Improving search techniques and using them on different environments.

Ariel Felner
Dept. of Mathematics and Computer Science

Ph.D. Thesis.

Submitted to the Senate of Bar-Ilan University,
Ramat-Gan, Israel
February 2001
This work was carried out under the supervision of Prof. Sarit Kraus,
The Department of Computer Science Bar Ilan University.

This work was also greatly advised by Prof. Richard E. Korf,
The Department of Computer Science, UCLA.
0.1 Acknowledgments

I would like to thank Prof. Sarit Kraus who was my supervisor. Her door was always open and her enlightening advises, guidance and support greatly contributed a lot to the quality of this thesis and enabled me to complete this work.

I convey my special thanks to Prof. Richard E. Korf who is the leading researcher in heuristic search. His willingness to help, to answer my questions and to share his knowledge with me was extraordinary. I would not have been able to work on heuristic search without his help.

I would like to thank Eitan Yarden and Moshe Malin for their help with writing the code and for their ideas on the GADB system of the tile puzzle. Thanks to Ziv Shalvi and Avidan Reich for their help with the code of the VC table for the DDB system.
Contents

0.1 Acknowledgments .................................................. 0

1 Introduction and overview ........................................... 1
  1.1 Best-First Search .................................................. 2
  1.2 Linear Space algorithms ......................................... 2
  1.3 The tile puzzle domain ............................................ 3
  1.4 Weighted A* ...................................................... 4
  1.5 Recent improvements in heuristic search ......................... 6
       1.5.1 New search algorithms .................................... 6
       1.5.2 Using memory for obtaining a better heuristic function ...... 6
       1.5.3 A comparison of the two approaches ........................ 8
  1.6 Overview of this thesis .......................................... 8

2 K-Best First Search ............................................... 10
  2.1 Attributes of KBFS. ............................................. 10
  2.2 Related work ................................................... 13
  2.3 Search trees with dead-ends .................................... 14
       2.3.1 Random trees with dead-ends ............................. 15
       2.3.2 Tests results ............................................. 16
  2.4 KBFS results on sliding-tile puzzles ........................... 18
       2.4.1 Results on the twenty-four puzzle ....................... 19
       2.4.2 KBFS results on the 15 puzzle .......................... 22
  2.5 KBFS on number partitioning .................................... 23
       2.5.1 The search tree for number partitioning .................. 24
       2.5.2 Empirical results ........................................ 26
  2.6 Conclusions and future work for the KBFS algorithm ............ 27

3 Near optimal perimeter search .................................... 28
  3.1 Memory bound search algorithms ................................ 28
  3.2 Perimeter Search ................................................. 29
3.2.1 Analysis of perimeter Search ........................................ 30
3.2.2 Complexity of generating the perimeter ............................ 31
3.3 The near-optimal version of perimeter search. ......................... 31
  3.3.1 Perimeter search with weighting the heuristics .................... 31
  3.3.2 Simple Near-Optimal Perimeter Search ............................ 32
  3.3.3 Time complexity of NOPS ......................................... 33
  3.3.4 Solution quality of NOPA* ...................................... 34
3.4 Empirical results of NOPA* on the fifteen puzzle ..................... 36
  3.4.1 Heuristic of nodes in the perimeters of the fifteen puzzle ...... 36
  3.4.2 Results of NOPIDA* and NOPRBFS on the fifteen puzzle .......... 37
3.5 NOPS and other near optimal algorithms ................................ 39
  3.5.1 Comparing WRBFS to pure PRBFS ................................ 39
  3.5.2 Weighted NOPRBFS (WNOPRBFS) ................................ 41
3.6 Combining both WA* KBFS and Perimeter Search ....................... 42
3.7 Intuition for NOPA* ................................................. 45
3.8 The twenty-four puzzle. .............................................. 45
3.9 NOPS as an anytime algorithm ....................................... 47
3.10 Using the perimeter for improving the heuristics .................. 49
3.11 Conclusions .................................................................... 50

4 Background: The additive pattern database theory .................... 51
  4.1 Better heuristics ...................................................... 51
  4.2 Heuristics from multiple subgoals ................................ 52
    4.2.1 The standard explanation .................................... 52
    4.2.2 A new explanation ............................................. 53
    4.2.3 Non-Additive Pattern Databases .............................. 53
    4.2.4 Generalization of the Manhattan distance .................. 55
    4.2.5 Triples of tiles ............................................... 57
  4.3 Computing an Admissible Heuristic .................................. 57
    4.3.1 Maximal Weighted Matching ................................ 58

5 Implementing the ADB on the tile puzzle ............................ 59
  5.1 The GADB system ..................................................... 59
    5.1.1 The size of the database .................................... 60
    5.1.2 Building the database ....................................... 61
    5.1.3 Constructing the graph ....................................... 61
    5.1.4 Vertex-Cover as an Admissible Heuristic .................. 62
5.1.5  Moving from pairs to triples ........................................ 63
5.1.6  Combining both pairs and triples. ................................ 65
5.1.7  Taking Vertex-Cover for a weighted graph ..................... 65
5.1.8  General schema of the program ................................... 66
5.1.9  Domain dependent enhancements for the Tile puzzle .......... 66
5.1.10 Future enhancements .............................................. 69
5.2  The Disjoint Database ................................................. 70
5.3  Using Vertex-cover for a selected number of groups. ............ 72
5.4  Empirical Results .................................................... 74
  5.4.1  Dividing the experiments into subgroups ....................... 74
5.5  The fifteen puzzle .................................................. 76
  5.5.1  Comparison with the first work of pattern database .......... 79
5.6  The twenty-four puzzle ............................................. 80
  5.6.1  Comparing the results of both version of the puzzle ........ 84
  5.6.2  Comparison to Previously Reported Results ................... 85
5.7  Summary and conclusions for the ADB work on the tile puzzles .. 85

6  Finding optimal solutions to the Vertex Cover Problem with Heuristic Search 87
  6.1  The Vertex Cover domain ........................................... 87
  6.2  Related work ..................................................... 89
  6.3  The search tree .................................................. 91
  6.3.1  Definitions .................................................. 91
  6.3.2  Eliminating degree-one vertices ................................ 92
  6.3.3  Neighbors of excluded vertices ................................ 92
  6.3.4  Ordering the vertices of the graph ............................ 93
  6.3.5  Dealing with degree-one or isolated vertices during the search process. 93
  6.4  The additive database heuristics for the VC problem .......... 94
  6.4.1  Lower bound on the VC of the free graph .................... 94
  6.4.2  Cliques of free vertices .................................... 94
  6.4.3  Constructing the database and computing the heuristics ...... 96
  6.5  The experiment design ............................................. 101
  6.5.1  The type of the database .................................... 101
  6.5.2  The type of the graph ....................................... 102
  6.5.3  The search algorithms ....................................... 103
  6.6  Empirical results ............................................... 104
  6.6.1  Random graphs with 150 vertices ............................ 104
7 Finding optimal solutions to the Graph-partitioning problem with Heuristic Search

7.1 The Graph partitioning problem ........................................ 112
7.2 Related work .......................................................... 114
7.3 The different cost functions and search algorithms. ................. 115
  7.3.1 Definitions ....................................................... 115
  7.3.2 \( f_0 \): Uniform cost search. ..................................... 117
  7.3.3 \( f_1 \): Adding edges from free vertices to assigned vertices. .... 117
  7.3.4 \( f_2 \): Pairwise costs: Following the additive pattern theory. .... 118
  7.3.5 \( f_3 \): Adding edges from two free vertices to \( f_1 \). ................. 120
  7.3.6 \( f_3 \): Sorting the free vertices ................................ 123
  7.3.7 \( f_4 \): Adding edges from two free vertices to \( f_3 \). ................. 125
7.4 Enhancements to the search process .................................. 127
  7.4.1 Ordering the vertices of the graph ................................ 127
  7.4.2 Calculating related heuristics ................................ 128
  7.4.3 Ordering nodes of the search tree ................................ 128
7.5 Empirical results ...................................................... 129
  7.5.1 Graphs with 50 vertices .......................................... 129
  7.5.2 Graphs with a degree of 8. .................................... 133
  7.5.3 Graphs of size 100 .............................................. 134
7.6 Discussion and conclusions for the GPP work .......................... 134

8 Summary and conclusions ................................................. 136
List of Tables

1.1 WA*: results on the fifteen puzzle. ........................................... 5
2.1 KBFS: Random trees with variable depth of dead-ends. ................. 17
2.2 KBFS: results on the twenty-four puzzle. .................................... 20
2.3 KBFS: winning and losing on the twenty-four puzzle. .................... 22
2.4 KBFS: comparison with hill-climbing on number partitioning. The values are the average cost of a solution after generating 6,000,000 nodes. ............... 26
3.1 Mistakes in the heuristics of nodes in the fifteen puzzle. .......... 36
3.2 Mistakes of the heuristics of nodes in the fifteen puzzle. ........... 38
3.3 WNOPRBFS results on the fifteen puzzle. .................................. 42
3.4 NOPKWA* results on the fifteen puzzle for p=18. ...................... 43
3.5 NOPKWA* results on the twenty-four puzzle. ............................. 46
5.1 ADB: sizes of the databases. .................................................. 75
5.2 Results of the various combination of the pattern databases on the fifteen puzzle. ................................................................. 77
5.3 Twenty-four puzzle results of the different combination of the ADB. .. 81
5.4 Twenty-four puzzle results of the different combinations of their ADB. .. 82
5.5 Twenty-four puzzle results of the different combinations of their ADB. .. 83
6.1 The average optimal vertex cover for graphs with 150 vertices. ....... 106
6.2 Performance of the different algorithms on graphs with 150 vertices and an average degree of 15. ............................................. 106
6.3 Performance of the different algorithms on Delaunay graphs with 100 vertices for finding vertex cover. ...................................... 109
6.4 Performance of the different algorithms on Delaunay graphs with 150 vertices for finding vertex cover. ...................................... 110
7.1 The H3 heuristic for the GPP. .................................................. 124
7.2 The number of nodes per second for graphs of size 50 with an average density
of 8 for the different heuristics ........................................ 130
7.3 Nodes visited and time elapsed for each of the algorithms. ............... 131
7.4 Graph with density of 8. ............................................. 133
7.5 Graph with size 100. .................................................. 135
List of Figures

1.1 The Fifteen Puzzle. ............................................. 4
2.1 Example search tree for KBFS. ................................. 11
2.2 A graph with a dead-end: The whole subtree rooted at node B is a dead-end. 15
2.3 Results on the dead-end trees. Each curve is a different KBFS. The figure shows that when the depth of the deadend subtrees grows, KBFS with larger k tends to generate fewer nodes. ......................... 18
2.4 Results on the dead-end trees. Presents the overhead factor versus the width of KBFS. There is one curve for each DD. The figure shows that for large values of DD, larger KBFS are better choices. .......................... 19
2.5 Example of an error in the Manhattan distance heuristic for the fifteen puzzle. 19
2.6 Path length versus generated nodes for twenty-four puzzle. ......................... 21
2.7 Path length versus generated nodes for the twenty-four puzzle. Larger scale. 21
2.8 Number partitioning: the different types of neighbors. ......................... 25
3.1 Nonoptimality of perimeter search. ............................... 33
3.2 Errors versus RBFS in the fifteen puzzle. ......................... 39
3.3 NOPIDA* versus NOPRBFS on the fifteen puzzle. ................. 40
3.4 WRBFS versus NOPRBFS. ........................................ 41
3.5 NOPWRBFS results on the fifteen puzzle. ......................... 44
3.6 NOPWRBFS results on the fifteen puzzle. ......................... 45
3.7 NOPWA* results on the fifteen puzzle. K is always 1. ................. 47
3.8 NOPKWA* results on the fifteen puzzle. .......................... 48
3.9 NOPWRBFS results on the twenty-four puzzle. ...................... 49
4.1 The Fringe Pattern for the Fifteen Puzzle. ....................... 53
4.2 The 24 tile puzzle. ............................................. 56
5.1 Triple linear conflict in the 24 tile puzzle. ....................... 62
5.2 Triple examples. ............................................. 64
First case example. .......................................................... 64
Triple examples. ............................................................. 67
Triple examples. ............................................................. 68
The 15 tile puzzle partitioning into disjoint databases. ............ 76
The 24 tile puzzle partitioning into disjoint databases. ............ 80

The search tree of the vertex cover domain .......................... 92
The free graph. Vertices in this graph are the free vertices ....... 93
Cliques of different sizes and their admissible heuristics .......... 96
The potential cliques-space tree. ....................................... 97
The actual cliques tree for a specific graph. ......................... 98
The general DB versus the disjoint db on the vertex cover domain .... 100
Random graph, 15 vertices, average degree of 5. ................. 103
Delaunay graph, 15 vertices. ........................................... 103
Time in seconds for the general additive database. The DFBnB algorithm and database of cliques of sizes 1 through 4 on random graphs of size 250. ................................. 105
Comparison of all types of algorithms with a database of cliques of a size up to 3 on random graphs of size 250. ......................... 107
Time in seconds for the disjoint database. The IDA* algorithm and the disjoint database of cliques of sizes 1 through 4 on random graphs. ..................... 108
Random graphs with average degree of 15. DFBnB, general database. ................................. 108

The search tree for the graph partitioning domain. ................ 113
The sets of the partition and the different edge types. ............ 116
The free graph. ............................................................. 120
Generalized matching of the free graph. .............................. 122
The optimal partition for graphs of size 50 with different densities. 130
The overall time for the algorithms on graphs with 50 vertices. DFBnB used. 131
The overall time for the algorithms on graphs with 50 vertices. IDA* used. 132
Comparing IDA* to DFBnB. ............................................. 132
Graphs with degree 8. Overall time. .................................. 134
Graphs with degree 8. Nodes generated. .............................. 134
Abstract

When using a heuristic search we have to decide upon two issues. The first issue is which search algorithm to use and the second one is what heuristic function to apply to the search space. Works on heuristic search can be classified into these two classes i.e improving search algorithms and finding better heuristic functions.

This thesis is divided into two parts based on these two classes. In the first part we will introduce two new general algorithms, K-Best-first search (KBFS) and Near-Optimal Perimeter Search (NOPS). Both of these algorithms propose general methods to improve the search process when a sub-optimal solution is sufficient. Both these algorithms are general and they do not depend on a special domain.

KBFS is a generalization of the well known Best-First Search (BFS)[18]. In KBFS, each iteration simultaneously expands the K best nodes from the Open-List (rather than just the best node as in BFS). We claim that KBFS outperforms BFS in domains where the heuristic function has errors in estimation of the real distance to the goal state or does not predict dead-ends in the search tree. We present empirical results that confirm this claim and show that KBFS outperforms BFS by a factor of 15 on random trees with dead-ends, and by a factor of 2 and 7 on the 15 and 24 tile puzzles, respectively. KBFS also finds better solutions than BFS and hill-climbing for the number partitioning problem. KBFS is only appropriate for finding approximate solutions with inadmissible heuristic functions.

Perimeter Search (PS) [13] is an admissible algorithm that was developed recently to improve the efficiency of search algorithms. In PS, first, a breadth-first search from the goal node is performed to depth d and all the nodes surrounding the goal node at that depth are saved in a hash table. Then, a forward search from the initial state is performed until one of the nodes in the perimeter is reached. The heuristic function that is used is the estimation of the distance between a node and the closest node to it in the perimeter. The disadvantage of such an algorithm is the large number of heuristic evaluations that need to be calculated for each node. We introduce the near optimal version of perimeter search where the heuristic that is used is the regular heuristic to the goal, while the search stops when reaching a node at the perimeter. We analyze the behavior of such an algorithm and the quality of the solution that it produces. We present empirical results that show that this algorithm outperforms other inadmissible algorithms like WA* and KWA* in the sense of path length versus generated nodes.

The second part of this thesis falls into the second class described above, namely finding better heuristic functions. We describe a general theory of deriving new heuristic functions which we call the additive pattern database theory. This theory was first presented by Korf in [33]. In this work we further develop, study and implement the theory on a number of
Many times a goal of a problem can be divided into a number of subgoals and a simple heuristic function like the Manhattan distance is based on how to solve each individual subgoal. The additive database is a general theory that generalizes this method. Instead of trying to solve each subgoal independently the additive pattern database considers the cost of solving 2, 3 or more subgoals simultaneously. We build a database that stores knowledge about the mutual cost of solving groups of subgoals. During the search, we consult this database to get an informed heuristic which is an estimation of the cost of solving groups of subgoals together. We introduce two techniques that allow the heuristics from different pattern database to be added together. The first one is called the General Additive Database (GADB) and the second one is called the Disjoint Additive database (DADB). In GADB we store the mutual cost of solving each pair or triple of subgoals in memory. Then, during the search, we try to combine these costs into an admissible heuristic while using as much knowledge as possible. In DADB we first divide the goal states into disjoint groups of subgoals. We subsequently store the mutual cost of solving each group in a database. Then, during the search, data from different databases can simply be added together.

We demonstrate and implement these techniques on both the sliding tile puzzle and on the vertex-cover problem. We show that on both domains such heuristics speed up the search by a great amount over the simple heuristics such as the Manhattan distance. The overall time needed to solve these problems with our new heuristics is reduced by a couple of orders of magnitude. With these techniques we can solve an average instance of the fifteen puzzle within a fraction of a second and an average instance of the twenty four puzzle can be solved in about 24 hours. For the vertex-cover problem we show that with our techniques for obtaining heuristic functions by looking at groups of subgoals, an optimal vertex-cover can be found much faster than using a simple heuristic function that looks at each subgoal separately. While both the GADB and the DADB greatly improve the simple heuristic, we conclude that for the sliding tile puzzle the DADB is better while for the VC problem the GADB is better.

We then show that generalizing the idea of the additive pattern database can result in new heuristics for other domains as well. We generalize the ADB idea and claim that looking into interactions between subgoals of the problem and adding them to the heuristic function result in a more accurate heuristic function and improves the search. We have applied this idea to the graph partitioning problem and introduce a sequence of heuristic functions that can optimally solve this problem. Each heuristic is more complicated and looks more deeply into interactions between vertices of the graph. The best heuristic function that we propose improves on the simplest one by a couple of orders of magnitude and an optimal graph partitioning can be found much faster by our techniques.
Chapter 1

Introduction and overview

Heuristic search is a general problem-solving mechanism in artificial intelligence. The search takes place in a problem space graph. Nodes represent states of the problem and edges represent legal moves. A problem instance consists of a problem space together with an initial state and a set of goal states. A solution is a path from the initial state to a goal state. The cost of a solution is the sum of the costs of its edges. A solution is said to be optimal if it is a lowest-cost path from the initial state to a goal state, otherwise it is considered suboptimal. The term generating a node refers to creating a data structure representing the node, while expanding a node means to generating all of its children. Actually we can view the search space as a search tree whose root is the initial state. We start the search from the initial state. The various search algorithms differ in the order that they decide to traverse the search tree by deciding which node to expand next. Different algorithms can be compared to each other according to the following three criterions:

- The time complexity: The time complexity is usually proportional to the number of generated nodes, since generating a node usually takes a constant time.
- The space complexity: An algorithm is measured by the amount of memory that it needs in order to solve a problem.
- The quality of the solution. An algorithm can return a solution of different qualities. A solution can be optimal i.e. a lowest cost path between the initial state and the goal state. A solution can also be suboptimal i.e. it is not guaranteed to be optimal. Sometimes we can bound the quality of a solution and say that it is not greater than the optimal solution by a constant factor.

A brief description of the various search algorithms is provided in the following sections.
1.1 Best-First Search

Best-First Search (BFS)[18] is a well known and common heuristic search algorithm. It keeps an open list of nodes that have been generated but not yet expanded and chooses the most promising node (the best) from it for expansion. When a node is expanded it is removed from the open-list and its neighbors are generated and added to the open list. The search usually terminates when a goal node is chosen for expansion or when the open list is empty.

Special cases of best-first search include breadth-first search, Dijkstra’s single-source shortest-path algorithm and the A* algorithm [18], differing only in their cost functions f(n). If the cost of a node is its depth in the tree, then best-first search becomes breadth-first search, expanding all nodes at a given depth before any nodes at any other depth. If the edges in the graph have different costs, then taking g(n), the sum of the edge costs from the initial node to node n yields Dijkstra’s algorithm. If the cost is f(n) = g(n) + h(n), where h(n) is a heuristic estimation of the cost from node n to a goal, then best-first search becomes the A* algorithm [18]. If h(n) is admissible, i.e., it never overestimates the actual cost from node n to a goal, then A* is guaranteed to return an optimal solution, if one exists.

1.2 Linear Space algorithms

A* was proved to be admissible, complete and optimally effective [12] when it uses an admissible heuristic. However, A*’s disadvantages include its memory and time complexities [26]. The main drawback of A* is its memory requirements. A* stores in memory all the open nodes in order to guarantee admissibility, and all the closed nodes in order to return the solution path once a goal is reached. The space complexity of A* is therefore equal to its time complexity which grows exponentially with the depth of the search. Therefore, A* cannot solve difficult problems since it usually exhausts all the available memory before reaching a goal. For the different versions of the sliding-tile puzzle for example, on current machines, A* can optimally solve only the eight puzzle but not the fifteen puzzle [27].

This problem has been addressed over the last two decades, and many algorithms that generate nodes in a best-first fashion but use only a linear or a restricted amount of memory have been developed. Iterative-Deepening A* (IDA*)[26] was introduced in 1985. It is actually a sequence of calls to DFS. In each iteration it expands all nodes having a total cost not exceeding a given threshold for that iteration. The threshold for the first iteration is the cost of the root of the search tree. The threshold for the next iteration is the lowest cost of a generated node from the previous iteration that exceeded the current threshold. IDA* was the first algorithm that solved the fifteen puzzle optimally. IDA* is the most popular algorithm for heuristic search since it has all the good attributes of A* but needs
only memory that is linear in the depth of the search. Its appearance was an important milestone in the field. IDA* generates new nodes in a best-first order only for monotonic cost functions.

Another famous algorithm is Depth First Branch and Bound. (DFBnB)\cite{34}. DFBnB is used when the depth of the solution in the search tree is known in advance. DFBnB performs a depth-first search of the tree but uses a global threshold for pruning nodes. This threshold is the best solution that has been found so far. At the beginning of the search, this threshold is set to infinity and DFBnB reaches a solution in one dive into the tree. The threshold is set to the cost of that solution and the DFS is continued until a better solution is found. When it reaches a node whose cost function is greater than the current threshold we know that solutions under that node will not be better than the best solution and the subtree under that node can be pruned. DFBnB starts with a solution and improves on it until it finds the optimal solution. Then, it keeps searching the tree to verify that a better solution is not feasible. Thus in this sense DFBnB is an anytime algorithm.

DFBnB is used only when we are guaranteed to have a feasible solution with every path or when we can bind the search to a specific level. Otherwise it might enter an infinite loop.

It was proven\cite{12} that an admissible search algorithm must at least visit all the nodes that are visited by A*. IDA* does not visit any other node, but visits some of the nodes more than once. DFBnB never visits any node more than once, but visit many nodes that are not visited by A*. See \cite{34} for a detailed comparison between these two algorithms.

IDA* generate nodes in a best-first order only when the cost function is monotonic. Recursive Best-First Search (RBFS)\cite{27} is also a linear-space algorithm. However, it expands nodes in a best-first order even when the cost function is nonmonotonic. While Iterative Deepening uses a global cost threshold, RBFS uses a local cost threshold for each recursive call. It takes two arguments, a node and an upper bound on cost, and explores the subtree below the node as long as it contains frontier nodes whose costs do not exceed the upper bound. It then returns the minimum cost of the frontier nodes of the explored subtree. The space complexity of both of these algorithms is linear in the depth of the search since they only need to store one branch of the tree and the siblings for nodes on that branch.

1.3 The tile puzzle domain

A famous search problem is the sliding-tile puzzle\cite{27} \cite{26}. The most popular version of this puzzle consists of a 4x4 array, with 15 square tiles numbered 1..15 and one blank or empty position. The object of the puzzle is to move the tiles to the goal state in which the tiles are in a special order. The tiles that are situated next to the blank position can move into the blank space and make their former position, the new blank position. Figure 1.1 shows the
Fifteen Puzzle. A common heuristic function for the tile puzzle is the Manhattan distance heuristic, which is the sum of the number of horizontal and vertical positions that each tile is away from its goal position. This heuristic is a lower bound on the actual solution cost, since each tile must move at least its Manhattan distance, and each move moves only one tile.

Finding optimal solutions to the tile puzzle was one of the first tasks researched in the field of heuristic search. The eight puzzle, whose search space contains $9!/2 = 181,440$ different states was optimally solved in the early days of the field. On current machines, the eight puzzle can be optimally solved by A* and even by a simple breadth-first search. The fifteen puzzle contains about $10^{13}$ different states and can not be solved even by A* since the available memory will be exhausted long before it reaches a goal state. This is because A* must store all generated nodes in its memory either in the open list or in the closed list. It was not until 1985, when Iterative Deepening A* (IDA*) was introduced by Korf in [26], that the fifteen puzzle was solved optimally. IDA* can solve the fifteen puzzle using the Manhattan distance in an average of about one minute for a single problem.

1.4 Weighted A*

The time complexity of A* and all the other linear space algorithms is proportional to the number of generated nodes, which is usually exponential in the depth of the search. The time complexity can be very large when addressing large problems. For example, finding optimal solutions to some of the 24-tile puzzle instances took several months to solve in [33] even though a heuristic function that is much more accurate than the Manhattan distance was used. One way to deal with this exponential complexity is to compromise on the quality of the solution and to settle for a suboptimal solution. With this compromise, a large reduction in the number of generated nodes can be obtained, at the cost of a small sacrifice in solution quality.
WA*: fifteen puzzle

<table>
<thead>
<tr>
<th>Wg</th>
<th>Wh</th>
<th>Moves</th>
<th>Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>99</td>
<td>145.37</td>
<td>6,957</td>
</tr>
<tr>
<td>1</td>
<td>19</td>
<td>127.65</td>
<td>7,924</td>
</tr>
<tr>
<td>1</td>
<td>9</td>
<td>116.49</td>
<td>9,527</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>103.29</td>
<td>10,460</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>88.15</td>
<td>15,818</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>78.41</td>
<td>22,840</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>63.51</td>
<td>78,870</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>56.61</td>
<td>496,384</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>53.05</td>
<td>363,028,079</td>
</tr>
</tbody>
</table>

Table 1.1: WA*: results on the fifteen puzzle.

Weighted A* (WA*) [42] [16] is a generalization of A* which accomplishes the above compromise. WA* uses \( f(n) = w_g \cdot g(n) + w_h \cdot h(n) \) as its heuristic function. If we define \( w \) as ratio \( w_h/w_g \), then the cost function can be written in the equivalent form: \( f(n) = g(n) + W \cdot h(n) \). The two extreme cases are A* when \( W = 1 \) and pure heuristic search when \( W \) is very large and actually the cost function is only \( h(n) \). When \( W > 1 \) the overall cost function is inadmissible since it may overestimate the real distance to the goal and therefore the solution it provides may have a greater cost than an optimal solution. It was shown in [16] that increasing \( W \) (i.e. giving more weight to \( h \) and less to \( g \)) results in finding solutions faster in terms of the number of generated nodes at the expense of an increased solution cost. It was proven in [17] that the solution cost returned by WA* cannot exceed the optimal solution cost by more than a factor of \( W \). Empirical results [27] show that the solution cost of WA* is much smaller than this upper bound in practice. In fact, one can control the tradeoff between the cost of the path and the number of generated nodes by tuning \( W \) to any desired value. The behavior of WA* can be perceived as paying a larger price for shorter solutions. Table 1.1 illustrates this tradeoff of WA* for different solution lengths of the fifteen puzzle.\(^1\). The values in the tables are the average over 100 random instances and were taken from[27]

As shown in the table, a small sacrifice in solution quality yields a large improvement in the running time. For example, obtaining a solution that is only 8% longer than the optimal solution yields an improvement of almost 3 orders of magnitude in the number of generated nodes. In this work we will present two new algorithms, namely KBFS and NOPS that outperform WA* in the sense of the number of generated nodes versus the quality of the solution obtained.

\(^1\)The last line of the table was calculated using IDA* because A* would exhaust the available memory.
1.5 Recent improvements in heuristic search

After IDA* was introduced, memory constraints were no longer a problem since IDA* is a depth-first search and therefore it uses memory which is only linear in the depth of the search. As computer memories got larger many methods for intelligent usage of the memory were developed in order to speed up the search. In order to improve heuristic search we can take one of the following approaches:

- Find a better search algorithm. Methods of this class introduce new ways to scan a given search tree. These methods are general and are domain independent. KBFS and NOPS are two types of such algorithms that are presented in this thesis.

- Find better heuristic functions. Methods of this class develop more accurate heuristic functions. With a better heuristic function the search will be guided to the goal state more quickly. These methods are usually domain dependent as the heuristic function is suitable only for a special domain.

1.5.1 New search algorithms

MREC [48], MA* [8] and SMA* [47] are all based on A* differing in the ways they proceed when the memory is full. They all start to execute A* until the memory is almost full. When the memory is full, MREC starts to execute IDA* on the leaves of the search tree that is currently stored in memory. MA* and SMA*, on the other hand, delete bad nodes from the open-list freeing memory for new nodes. Another work for using memory in an intelligent way using a finite state machine (FSM) for pruning duplicate nodes [32] by Taylor and Korf. The FSM is discovered automatically and it enforces pruning rules that eliminate the generation of duplicate nodes in a depth first search oriented search. This method of using FSM for pruning duplicate nodes [32] can be also included in this class of new search algorithms since it helps the search not to expand nodes that were already expanded. All these algorithms, however, use the same search tree as simple IDA* search and only suggest different methods of how to search that tree more efficiently than the standard IDA*.

1.5.2 Using memory for obtaining a better heuristic function

The second class of works are methods for using a more informed search tree by using large tables in memory in order to evaluate a more accurate heuristic function. Both Perimeter Search (PS) [13] and BIDA* [35] are algorithms that save a large table in memory which contains all the nodes that surround the goal node up to a fixed depth. In these perimeter searches, first, a breadth-first search from the goal node is performed to a fixed depth and
then all the nodes surrounding the goal node at that depth are saved in a hash table. Then, a forward search from the initial state is performed until one of the nodes in the perimeter is reached. The heuristic function that is used is the estimation of the distance between a node \( n \) and the closest node to it in the perimeter (or from a subset of nodes from the perimeter as in BIDA*). Adding this amount to \( g(n) \) and to the depth of the perimeter results in a much more accurate heuristic in comparison to the regular heuristic to the goal. The disadvantage of such an algorithm is the large number of heuristic evaluations that need to be calculated since at each step the distance from the new generated node to all the nodes in the perimeter must be estimated in order to find the closest node from the perimeter. However, empirical results show that despite the fact that much more work is done for each node in order to evaluate the heuristic function, in the overall calculation, it is worthwhile. The great reduction in the number of generated nodes compensates the constant time per node and yields a better overall time.

Another method for intelligent usage of memory in order to attain a better heuristic is the Pattern database which was used to solve both the fifteen puzzle in [10] and Rubik's cube in [25]. Both systems use a large hash table to store the exact distance of a part of the domain to its goal position. This distance is taken as a heuristic estimation to the real distance of the complete state. For example, in Rubik's cube part of the work was to calculate the exact distance of all the corner cubies of each state and then to store it in memory. This distance was then used when calculating the heuristic value of a particular state. In the fifteen puzzle, the reduction of the number of node generations was up to a factor of 1000, while the real time improvement was smaller and was no more than 12 over simple IDA* with Manhattan distance. Applying the pattern database technique to Rubik's cube was the first program that found an optimal solution to this problem. The weakness of the pattern database technique is the large size of the table which uses all the available memory. Even though today memory capacities are up to one Gigabyte of main memory these techniques nearly exhaust the memory.

In all the above works, the reduction in the number of node generations did not result in the same improvement in the overall real time since finding a better heuristic causes a larger constant time per node generation. This suggests that there is a tradeoff of an increased constant cost per node versus a better heuristic. When a new method of calculating a heuristic function is found, it should be determined if it is cost effective to use this heuristic in the sense of this tradeoff. For example, in PS [13], an optimal perimeter exists.

It is not worthwhile to use larger perimeters even though better heuristics are used, since the constant time per node dominates the heuristic improvements. In the improved perimeter search (BIDA*) [35] however, increasing the depth of the perimeter always resolves a better overall time. This is because in BIDA* only distances to a subset (Active set) of
the perimeter nodes are calculated.

Korf in [33] optimally solved the twenty-four puzzle by using a more informed heuristic than the Manhattan distance. He discovered, that when looking into interactions between pairs of tiles, there are some cases where the two tiles are in conflict and thus a heuristic that takes these conflicts into consideration is more accurate than the simple Manhattan distance heuristic. In this work, Korf also came up with the additive pattern database theory, but did not implement it and was sceptical whether it can be cost effective to implement it.

### 1.5.3 A comparison of the two approaches

Comparing the empirical results of the above two approaches, namely better search algorithms and better heuristic functions, shows that the second class is more powerful and yields a better time improvement over simple IDA*. However, methods in this class concentrate on the heuristic function and are usually domain dependent. These improvements should be implemented separately for each domain and there might be domains which will not be applicable. On the other hand the first class is domain independent since it only improves the search algorithms. Therefore this class of improvements can be used for almost any domain. For example, a necessary condition for using a perimeter is that only one or a small number of goal nodes exist. The pattern database method can be used only when a state in the search space can be divided to sub-states.

### 1.6 Overview of this thesis

In this thesis we consider both approaches that were stated above. In the first part we follow the general approach and introduce two new search algorithms that are orthogonal to each other and that are useful for finding suboptimal solutions. They are both simple to understand and implement but yet are very powerful for finding suboptimal solutions with almost no overhead to implement. They seem to outperform other suboptimal algorithms. In the second part we follow the other approach and introduce a general method for obtaining new and more accurate heuristic functions based on our additive pattern database theory. We then show empirical results in three domains that demonstrate the advantage of these new heuristics. This thesis is organized as follows.

In chapter 2 we introduce the K-Best-First Search (KBFS) algorithm, study its nature and behavior and describe the attributes of the search space where KBFS might be valuable. We then show empirical results on a number of domains that confirm that KBFS outperforms simple BFS on these domains.

In chapter 3 we introduce the Near-Optimal Perimeter Search (NOPS) algorithm. We
first discuss previous work on perimeter search and then describe the NOPS algorithm which is our modification of perimeter search. We then formally state the quality of the solution it produces and its time complexity. Then we present empirical evidence that NOPS outperform simple BFS. We then show that KBFS and NOPS are orthogonal and may be combined to improve the search.

In chapter 4 we present the additive pattern database theory (ADB) as was first presented by Korf in [33]. The ADB is a general way to derive powerful heuristic functions by storing a large table of solutions to subproblems in memory. We compare this new theory to existing heuristics and present both the General database system (GADB) and the Disjoint database (DADB). Both systems are based on the additive database theory and are efficient ways to build a search algorithms.

In chapter 5 we present empirical results of the ADB on both the fifteen puzzle and on the twenty four puzzle with both the GADB and the DADB systems. We show that both systems outperform simple heuristics like the Manhattan distance by a couple orders of magnitude.

In chapter 6 we present an algorithm that optimally solves the vertex cover problem with a heuristic search that uses the Additive database theory. We present empirical results on this domain with both the GADB and the DADB. Both systems outperform simpler heuristics.

In chapter 7 we show that the additive database theory can be generalized. We present an algorithm that optimally solves the graph partitioning problem (GPP) with heuristic search. The heuristics that we develop follow the same principle of the ADB, namely that looking into interaction between unsolved subgoal can result in a better heuristic function. Nevertheless, unlike the ADB theory in our work on the GPP no database is used.

The conclusions and summary are presented in Chapter 8.
Chapter 2

K-Best First Search

KBFS(k) is a generalization of BFS in that each cycle of KBFS(k) expands the best $k$ nodes from the open list and not only the first best node as in BFS.

We first describe the behavior of KBFS and claim that when there are errors in an inadmissible heuristic function, KBFS outperforms BFS. Then, we show empirical results on incremental random trees, on the sliding-tile puzzle, and on the number-partitioning problem that confirm our claim.

When addressing large problems, such as those larger than the twenty-four puzzle, obtaining a suboptimal solution is the only practical way to solve these problems within a reasonable amount of computing resources today. Also, while theoretically an optimal solution is more interesting, in practice many times a suboptimal solution suffices, and algorithms that return suboptimal solutions are sufficient for such cases. We propose KBFS as an algorithm that is simple but sufficiently powerful for these purposes.

2.1 Attributes of KBFS.

KBFS(k) is a generalization of BFS in the sense that in each node expansion cycle, KBFS(k) expands the best $k$ nodes from the open list. Their children are generated and added to the open list, and only then a new expansion cycle begins and the next $k$ best nodes are selected and expanded. The number of nodes expanded in each iteration is the width or simply the $k$ parameter (BFS is therefore KBFS(1)). The difference between BFS and KBFS is that in BFS, after the children of the best node are added, a new expansion cycle begins and if one of these children is the best it is selected and expanded. In KBFS, the new children of a node are not examined until the rest of the previous $k$-best nodes are expanded and their children are added to the open list. While KBFS(1) is simply BFS, KBFS($\infty$) is actually Breadth-first search since at each expansion cycle all the nodes in the open list are expanded.
All these nodes are actually at the same depth of the search tree. In that sense KBFS is a hybrid of best-first search and breadth-first search. BFS makes its decisions of which node to expand while relying completely on the cost value of the best node. As we will show below, if that value is mistaken, BFS can jump to wrong conclusions. KBFS relaxes that dedication and tries to explore other possibilities as well. KBFS introduces diversity into the search. The constant time per node for KBFS is the same as BFS since they both expand best nodes from the open-list.

![Figure 2.1: Example search tree for KBFS.](image-url)

Consider the behavior of KBFS(2) versus BFS on the search tree of Figure 2.1, where the numbers represent the complete evaluation of each node. BFS expands node A generating nodes B and C. Then it expands node B generating nodes D, E and F. It expands nodes D, E and F and only then goes back to expand node C generating node H and then the GOAL node. KBFS(2) however, after generating nodes B and C, expands them both generating nodes D, E, F and H. Then, nodes H and D are expanded and the GOAL node is reached. BFS explores the subtree under node B until all frontier nodes in that subtree have costs greater than node C. KBFS(2) looks at both subtrees simultaneously and reaches the goal faster.

A cost value given to a node is usually based on a heuristic estimation of the real cost from that node to the goal, and therefore is not exact by nature. If there is a large deviation between the heuristic value of a node and the exact cost we can say that there is an error in the heuristic estimation of the node.\(^1\) When BFS chooses a best node for expansion it believes that the closest goal node is a descendent of that node. However, when the heuristic value of the best node has an error, that belief may not be true and thus causes BFS to make a mistake in its decision of which node to expand next. When B and C in Figure 2.1

\(^{1}\)Since heuristic functions usually underestimate, the typical error will be a heuristic value that is much smaller than the real distance.
are at the front of the open list, BFS makes a mistake and expands nodes under B while the
goal is under C. In such cases, BFS may incur unnecessary overhead by expanding nodes
in a wrong subtree. When that subtree is large and the values of the nodes in that subtree
also contain errors, then the overhead grows. On the other hand, when the goal node is
indeed under the best node as was suggested by the heuristic function, then BFS makes a
correct decision in expanding it. In that case KBFS(k) incurs the overhead of expanding
\( k - 1 \) unnecessary nodes for that cycle, since expanding only the best node will suffice. If
the value of node B were 22 for example, BFS would expand A,C,H and reach the GOAL.
KBFS(2) would expand A,C,B,H and D and only then get to the GOAL. In this case, when
BFS does not make any mistakes, KBFS(2) expands two nodes more than BFS (B and D),
one in each cycle. KBFS can be viewed as insurance against remaining in a misleading
subtree. The question is of course whether this insurance is worth the cost. The question
of which algorithm causes more overhead arises: KBFS while exploring unnecessary nodes
when there are no mistakes made by BFS, or BFS when entering a wrong subtree. The main
hypothesis of this chapter is therefore:

"As the number of errors in the heuristic values of nodes in the search space
increases, BFS makes more mistaken decisions and therefore KBFS with greater
widths tends to outperform BFS".

One observation must be made here before continuing. KBFS will not be effective when
the heuristic function is \textit{admissible} and therefore the corresponding cost function is usually
\textit{monotonic}, i.e. the cost values cannot decrease down a branch of the problem space graph.
To ensure obtaining optimal solutions, an admissible heuristic is required. To guarantee
optimality, all nodes whose cost is less than the optimal solution cost must be expanded.
Therefore, the order that they are expanded does not really matter. In the above example,
if the cost function had been monotonic, then generating the subtree of B would eventually
reach a node with the same value as C, and then C would be expanded. With monotonic cost
functions, BFS tends to correct itself when there are errors in the heuristic values. However,
when the cost function is \textit{nonmonotonic}, BFS may never reach C before the whole subtree of
B is explored, and therefore KBFS is preferable here. A* for example is normally used with
an admissible cost function, and therefore we do not believe that KA* will systematically
outperform A*. KBFS should be considered for cost functions that are not admissible, such
as the weighted cost function of WA* [42] [27], where optimal solutions are not required, or
are too expensive to calculate.
2.2 Related work

The idea that errors in the heuristic values can lead to poor behavior of ordinary search strategies has been examined by a number of researchers in the past few years. Limited Discrepancy Search (LDS)[19] was introduced in 1995. Given a heuristic estimate to order successors of a node, a discrepancy is to not follow the heuristic preference at some node, but to examine some other node that was not suggested by the heuristic value. LDS first looks at the path with no discrepancies at all, then at all paths with one discrepancy. Then it continues to look at paths with increasing numbers of discrepancies. It was shown that LDS outperforms depth-first search (DFS) on hard problems. The LDS approach has been enhanced by the Improved Limited Discrepancy Search [30] and the Depth-Bounded Discrepancy Search [50].

Parallel Depth-First Search (PDFS) [19] is also related to KBFS. PDFS activates a number of DFS engines simultaneously. It was shown that in difficult problems where the heuristic function makes many mistakes at shallow levels, parallel DFS outperforms sequential DFS.

An interesting simultaneous DFS is Interleaved Depth-First Search [37]. It searches a path until a leaf is found, and if that leaf is not the goal, it does not backtrack to the father of the leaf, but rather starts another DFS path from the root node. IDFS was shown to outperform DFS on randomized Constraint Satisfaction Problems (CSP).

None of the above algorithms always follow the advice of the heuristic function. Sometimes they expand a node in contrast to the suggestion of the heuristic function. Nevertheless, they all deal with improving DFS and they were tested on CSP problem-space trees where all goals are at the lowest level, and therefore once reaching this level these algorithm can backtrack. Our work is novel in the fact that we try to analyze and improve the behavior of best-first search when there are errors in the heuristic function.

The term 'K-Best' is not new in the field of heuristic search and has been used to describe other algorithms. However none of these algorithms generalizes Best-First Search to its K-Best version. For example, an algorithm similar to KBFS is the classic K Beam Search. Beam Search always chooses to expand the best child of the current expanded node while the other children are discarded. K Beam Search simultaneously expands the best k nodes from the nodes that were just generated and discards the rest of them. Even though this algorithm is reminiscent of our algorithm it is completely different since it is a Depth-First Search algorithm and it does not maintain an Open list. The new K nodes in K-Beam Search are chosen from a limited number of nodes that are children of the last K expanded nodes. K-Beam Search never moves to another part of the tree, but rather only makes local decisions of where to continue. Therefore the decision of which K nodes to expand is very important in K-Beam search, since it is unrecoverable. It is usually used for real-time applications
or certain CSP problems. KBFS however, behaves in a best-first manner and chooses the K-Best among all generated nodes. KBFS can always backtrack to another part of the tree when it realizes that it is proceeding in a poor direction.

K-Beam Search was used in CRESUS, an expert system for cash management [49]. It also adds “diversity” to the search by choosing to also expand nodes that seem very different from other nodes in the set of the K nodes that are chosen for expansion. This approach can only be implemented in environments where the states are a set of transactions on the world. This is not true for the general case of a graph. Errors in the heuristic lead the search towards a 'local minimum' at the expense of reaching the ’global optimum’ faster. Introducing diversity into the search, for example by expanding the K-best nodes, can help to try other directions as well.

Many attempts have been made to generalize best-first algorithms to their parallel versions. Some of these include Parallel IDA*, PIDA* [43] and IDPS [1]. The most closely related parallel best-first search to KBFS is PRA* [36]. In PRA*, each generated node is assigned by a hash function to one of the working processors and to a bucket of memory related to that processor. Each processor then expands the best node from its bucket, generates its children and broadcasts each of them to the processor they are assigned to by the hash function. This processor now keeps the new child in its bucket. If there are K processors, then some K promising nodes are always selected and expanded. However PRA* is different from KBFS even in the sense of which nodes it expands. In PRA* each processor locally chooses the best node from its bucket and expands it, while KBFS globally chooses the best K nodes among all the nodes that were generated. Also, PRA* was designed for a parallel architecture and will not be useful on a serial machine. PRA* is an admissible algorithm while KBFS is inadmissible. PRA* has a complicated retracting mechanism and incurs the overhead of communications between different processors, while KBFS is very simple to understand and implement.

Best-First Minimax Search was developed in [28]. The traditional method for traversing a game tree is to use depth-first search with a bound for the maximum depth. Thus the tree that is search by this method is balanced. Best-First Minimax uses the best-first principal and expands nodes in a game tree in a best-first order. Here, the tree may not be balanced and some paths might be explored to deeper levels than others. This algorithm was compared to Alpha-Beta minimax and the results showed somewhat mixed success.

### 2.3 Search trees with dead-ends

A special case of an error in the heuristic is when a node has a low heuristic value implying that it is close to a goal, but it is a root of a Dead-end subtree. Dead-end subtrees can occur
in real-world problems. For example, in road navigation every peninsula or neighborhood
surrounded by a fence is usually a dead-end subtree. *Pure heuristic search*, which is BFS
with \( f(n) = h(n) \) might visit every little town in the peninsula before trying to go around.
KBFS will explore nodes also outside that subtree and therefore may reach the goal faster.

Consider the graph of Figure 2.2 where the value written next to a node represents its
heuristic value. The two subtrees rooted at B and C are structurally symmetric, but node
B is the root of a dead-end subtree. KBFS(1), i.e., BFS, expands the nodes in the following
order: A, B, F, M, K, Q, E, D, C, H, N, Goal. It expands all the nodes of the left subtree
since their values are all smaller than 75 which is the value of C. Only then does it move to
the subtree rooted at C. KBFS(2), however, expands the nodes in the following order: A, B,
C, F, H, M, N, Goal. Most of the nodes of the dead-end subtree of Figure 2.2 are not visited
by KBFS(2).

2.3.1 Random trees with dead-ends

A common testbed for heuristic search algorithms are *incremental random trees*. They were
used in [21], [24] and [28]. Such trees allow experimental control over the branching factor,
search depth and the heuristic evaluation function. An incremental random tree is generated
by assigning independent random values to the edges of the tree, and computing the value of
a node as the sum of the edge costs from the root to that node. This produces a correlation
between the heuristic value of a node and that of its descendants. The first tests that we
conducted were on random trees that are very similar to incremental random trees, but the
mechanism we used to generate them is a little different and will be described below.

In our trees, the value of a node represents the estimated heuristic distance to the goal.
Thus we first assigned a value to the root of the tree and then for each node we assigned a
random value that could not deviate from its parent's value by more than a constant value.
This deviation is similar to the edge cost in the incremental trees. A node with a value of
0 or less is considered a goal node. In order to produce dead-end subtrees we installed a
mechanism that introduces dead-end subtrees into the procedure that generates the tree.
This mechanism randomly declares a node as a root of a dead-end subtree and then allows that subtree to have descendants only to a fixed depth. This depth is a variable parameter. All the heuristic values assigned to the nodes in such a subtree are errors since the real distance is $\infty$. We are aware of the fact that this is not a natural way to produce random trees and that we added a somewhat artificial dead-end subtree generator into the random trees. Our objective was to produce such trees that simulate the phenomena of dead-end subtrees that occur in the real world, for example, in road navigation or in CSP problems.

For each of the random trees that we generated we used the following parameters:

1. The heuristic value of the root node was set to 2000.
2. The maximum branching factor was set to 5. Each node has a random number of children with uniform distribution from 1 to 5.
3. The maximum deviation of a node’s value from that of its parent was set to 50. In pure heuristic search, the cost values usually decrease during the search since the search gets closer and closer to the goal. In order to simulate this we forced 80% of the nodes to have smaller values than their parents. This causes the cost function to be non-monotonic.
4. The dead-end probability was set to 20%. Thus, each node not already inside a dead-end subtree has a probability of 20% to be declared the root of a dead-end subtree.
5. The maximum depth of a dead-end tree was labeled DD (Dead-end Depth). Each dead-end subtree has a depth of uniform distribution from $[0..DD]$. For example, DD of 5 means that the subtree under each node that was selected to be the root of a dead-end subtree had a maximum depth of 5 or less. A DD of 0 means that the root itself is a dead-end and has no children at all. During the tests the DD parameter varied from 0 to 12.

After the tree was generated, we searched for a goal node using different widths of KBFS. We counted the number of generated nodes for each algorithm. For comparison, we also searched with simple Depth-First Search which chooses to expand an arbitrary child and with Depth-First Search with node ordering where children are chosen in increasing order of their heuristic values.

2.3.2 Tests results

The variable parameter on our tests was DD increasing from -1 to 12. -1 or ‘nod’ is where there were no dead-ends at all. The results are shown in Table 2.1. Each row corresponds to
Table 2.1: KBFS: Random trees with variable depth of dead-ends.

<table>
<thead>
<tr>
<th>DD</th>
<th>DFS</th>
<th>DFS+</th>
<th>k0</th>
<th>k1</th>
<th>k2</th>
<th>k3</th>
<th>k4</th>
<th>k5</th>
<th>k6</th>
<th>k7</th>
<th>k8</th>
<th>k9</th>
<th>k10</th>
<th>k11</th>
<th>k12</th>
</tr>
</thead>
<tbody>
<tr>
<td>nod</td>
<td>2.34</td>
<td>1.01</td>
<td>1</td>
<td>1.74</td>
<td>2.54</td>
<td>3.19</td>
<td>3.93</td>
<td>4.67</td>
<td>5.37</td>
<td>6.08</td>
<td>6.79</td>
<td>7.49</td>
<td>8.25</td>
<td>8.97</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1.88</td>
<td>1.04</td>
<td>1</td>
<td>1.41</td>
<td>1.89</td>
<td>2.38</td>
<td>2.88</td>
<td>3.37</td>
<td>3.86</td>
<td>4.35</td>
<td>4.84</td>
<td>5.33</td>
<td>5.81</td>
<td>6.31</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1.99</td>
<td>1.03</td>
<td>1</td>
<td>1.39</td>
<td>1.8</td>
<td>2.24</td>
<td>2.69</td>
<td>3.16</td>
<td>3.62</td>
<td>4.07</td>
<td>4.52</td>
<td>4.96</td>
<td>5.4</td>
<td>5.86</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1.62</td>
<td>1.11</td>
<td>1</td>
<td>1.08</td>
<td>1.37</td>
<td>1.65</td>
<td>1.99</td>
<td>2.29</td>
<td>2.57</td>
<td>2.87</td>
<td>3.19</td>
<td>3.55</td>
<td>3.86</td>
<td>4.18</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2.15</td>
<td>1.06</td>
<td>1</td>
<td>1.09</td>
<td>1.3</td>
<td>1.53</td>
<td>1.78</td>
<td>2.01</td>
<td>2.28</td>
<td>2.53</td>
<td>2.81</td>
<td>3.06</td>
<td>3.32</td>
<td>3.6</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2.79</td>
<td>1.31</td>
<td>1</td>
<td>1.04</td>
<td>1.27</td>
<td>1.49</td>
<td>1.69</td>
<td>1.76</td>
<td>1.93</td>
<td>2.14</td>
<td>2.35</td>
<td>2.56</td>
<td>2.78</td>
<td>2.98</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>3.61</td>
<td>0.98</td>
<td>1</td>
<td>1.18</td>
<td>1.28</td>
<td>1.25</td>
<td>1.39</td>
<td>1.5</td>
<td>1.58</td>
<td>1.66</td>
<td>1.79</td>
<td>1.89</td>
<td>2.1</td>
<td>2.23</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>2.66</td>
<td>1.3</td>
<td>1</td>
<td>0.92</td>
<td>0.95</td>
<td>0.96</td>
<td>0.96</td>
<td>0.93</td>
<td>0.96</td>
<td>1.02</td>
<td>1.08</td>
<td>1.14</td>
<td>1.23</td>
<td>1.28</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>2.5</td>
<td>1.36</td>
<td>1</td>
<td>0.66</td>
<td>0.66</td>
<td>0.56</td>
<td>0.56</td>
<td>0.56</td>
<td>0.57</td>
<td>0.63</td>
<td>0.64</td>
<td>0.63</td>
<td>0.64</td>
<td>0.67</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>2.15</td>
<td>1.32</td>
<td>1</td>
<td>0.77</td>
<td>0.51</td>
<td>0.45</td>
<td>0.39</td>
<td>0.44</td>
<td>0.55</td>
<td>0.56</td>
<td>0.56</td>
<td>0.52</td>
<td>0.67</td>
<td>0.48</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>4.21</td>
<td>1.57</td>
<td>1</td>
<td>0.88</td>
<td>0.84</td>
<td>0.83</td>
<td>0.83</td>
<td>0.83</td>
<td>0.63</td>
<td>0.59</td>
<td>0.54</td>
<td>0.57</td>
<td>0.58</td>
<td>0.59</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>3.84</td>
<td>1.26</td>
<td>1</td>
<td>0.83</td>
<td>0.71</td>
<td>0.5</td>
<td>0.44</td>
<td>0.4</td>
<td>0.38</td>
<td>0.37</td>
<td>0.35</td>
<td>0.34</td>
<td>0.32</td>
<td>0.33</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>4.06</td>
<td>1.09</td>
<td>1</td>
<td>0.56</td>
<td>0.42</td>
<td>0.37</td>
<td>0.32</td>
<td>0.29</td>
<td>0.25</td>
<td>0.22</td>
<td>0.22</td>
<td>0.21</td>
<td>0.19</td>
<td>0.18</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>4.04</td>
<td>1.14</td>
<td>1</td>
<td>0.29</td>
<td>0.27</td>
<td>0.23</td>
<td>0.18</td>
<td>0.15</td>
<td>0.14</td>
<td>0.23</td>
<td>0.13</td>
<td>0.09</td>
<td>0.08</td>
<td>0.07</td>
<td></td>
</tr>
</tbody>
</table>

a different value of the DD parameter. Each column specifies the results of different search methods: simple DFS, DFS with node ordering, BFS, and then KBFS(2) up to KBFS(12). The values represent the ratio of the number of nodes generated by each algorithm divided by the number of nodes generated by BFS averaged over 500 different random trees. Thus, Table 2.1 presents the overhead of the tested algorithm relative to BFS. When that factor is smaller than 1, then the tested algorithm outperforms BFS. The bold values in Table 2.1 represent the algorithm that performed best for each row (DD).

It is clear from Table 2.1 that as the dead-end subtrees become deeper, larger widths should be used for KBFS. In the first row where there were no dead-ends at all, BFS is the algorithm of choice. In this case there were no errors in the heuristics and therefore larger widths yield greater overhead. As long as the depth of the dead-end subtree, DD, does not exceed 5, BFS is the best option, because there are not enough nodes under a mistaken subtree to make it worth the overhead of expanding the other best nodes. From depth 6 and up, BFS is no longer the best. As the dead-ends go deeper and deeper there are more nodes with errors which make it more worthwhile to expand other best nodes i.e. increase the width of KBFS. When DD is 12, KBFS(12) expands only 7% of the number of nodes expanded by BFS, constituting a factor of 15. Both Figures 2.3 & 2.4 illustrate the data presented in Table 2.1 Figure 2.3 presents the overhead factor of KBFS versus the depth of the dead-end trees (DD). Each curve represents a different k. Figure 2.4 presents the overhead factor versus the width of KBFS. Each curve corresponds to a different DD.

Results of additional simulations with other values for the parameters of the tree were similar to the above results. The results from the random trees support our hypothesis that greater widths should be used where there are deeper dead-ends in the search space.
Figure 2.3: Results on the dead-end trees. Each curve is a different KBFS. The figure shows that when the depth of the deadend subtrees grows, KBFS with larger k tends to generate fewer nodes.

### 2.4 KBFS results on sliding-tile puzzles

In this section we present the results of testing KBFS on sliding-tile puzzles with the Manhattan distance heuristic.

The Manhattan distance is not accurate for many states. For example, consider a state where all but two pairs of tiles are correctly placed but each of the tiles in these two pairs are swapped as shown in Figure 2.5. The heuristic value given to that state by the Manhattan distance function will be 4 while the actual number of moves required is of course much greater since other tiles must be moved as well. Nodes in this class behave like roots of dead-ends in the sense that the heuristic value given to them is very small compared to the correct distance from the goal. The heuristic values of many nodes in a subtree under such nodes contain errors. Looking down into their subtrees will cause a lot of overhead and a goal node will not be reached as quickly as was suggested by the heuristic value. KBFS might overcome this overhead because the value of another best node might be more accurate and the algorithm may bypass the problem.

We tested KBFS both on the $4 \times 4$ fifteen puzzle and on the $5 \times 5$ twenty-four puzzle with the WA* heuristic function, $f(n) = g(n) + W \times h(n)$. We ran the resulting algorithm, called KWA*, with different values for K and W. For W we took the values that were used in [27]. Each KWA* algorithm was run over the 100 instances of the fifteen puzzle that were first used in [26]. For the twenty-four puzzle we generated 100 solvable instances. Both the
Figure 2.4: Results on the dead-end trees. Presents the overhead factor versus the width of KBFS. There is one curve for each DD. The figure shows that for large values of DD, larger KBFS are better choices.

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>1</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>15</td>
<td>14</td>
</tr>
</tbody>
</table>

1 is swapped with 2 and 14 with 15.

Manhattan distance=2+2=4.

Figure 2.5: Example of an error in the Manhattan distance heuristic for the fifteen puzzle.

Solution length and the number of nodes generated by each of these runs were recorded and then averaged over these 100 instances.

2.4.1 Results on the twenty-four puzzle

On the twenty-four puzzle we tested 30 different values of W on 100 randomly generated solvable problem instances producing 3000 different searches. The tests were performed on a 233MHz pentium with the ability to store up to 13 million states in its memory. Table 2.2 shows the results of KWA* on the 24 puzzle. Each number represents the average of the 100 instances for different values of K and W. When using simple WA* which is actually BFS, smaller values of W produce shorter solutions at the expense of more generated nodes as shown in the first column. The results given in Table 2.2 suggest a better method for getting shorter solutions, i.e., increasing the width of KWA*. The table shows that KWA*
Table 2.2: KBFS: results on the twenty-four puzzle.

<table>
<thead>
<tr>
<th>W = wh/wg</th>
<th>BFS</th>
<th>KBFS(10)</th>
<th>KBFS(50)</th>
<th>KBFS(100)</th>
<th>KBFS(200)</th>
<th>KBFS(300)</th>
</tr>
</thead>
<tbody>
<tr>
<td>9/10</td>
<td>329.96</td>
<td>55853</td>
<td>285.62</td>
<td>59061</td>
<td>224.7</td>
<td>60934</td>
</tr>
<tr>
<td>24</td>
<td>302.54</td>
<td>71500</td>
<td>257.38</td>
<td>87973</td>
<td>206.74</td>
<td>84322</td>
</tr>
<tr>
<td>191</td>
<td>286.86</td>
<td>97697</td>
<td>242.56</td>
<td>103464</td>
<td>198.96</td>
<td>93488</td>
</tr>
<tr>
<td>47/3</td>
<td>274.50</td>
<td>91208</td>
<td>237.48</td>
<td>128308</td>
<td>193.12</td>
<td>97641</td>
</tr>
<tr>
<td>23/2</td>
<td>264.68</td>
<td>118756</td>
<td>236.66</td>
<td>143079</td>
<td>187.68</td>
<td>113878</td>
</tr>
<tr>
<td>9/1</td>
<td>248.86</td>
<td>142051</td>
<td>215.96</td>
<td>168899</td>
<td>181.06</td>
<td>122100</td>
</tr>
<tr>
<td>8/1</td>
<td>243.18</td>
<td>131928</td>
<td>200.38</td>
<td>218125</td>
<td>177.54</td>
<td>143871</td>
</tr>
<tr>
<td>7/1</td>
<td>235.42</td>
<td>154343</td>
<td>204.46</td>
<td>225661</td>
<td>173.82</td>
<td>160285</td>
</tr>
<tr>
<td>87/13</td>
<td>231.56</td>
<td>189788</td>
<td>200.84</td>
<td>276119</td>
<td>172.16</td>
<td>181496</td>
</tr>
<tr>
<td>6/1</td>
<td>226.1</td>
<td>227516</td>
<td>196.74</td>
<td>308922</td>
<td>169.54</td>
<td>171483</td>
</tr>
<tr>
<td>17/3</td>
<td>221.18</td>
<td>321137</td>
<td>192.74</td>
<td>348828</td>
<td>168.22</td>
<td>170388</td>
</tr>
<tr>
<td>5/1</td>
<td>211.60</td>
<td>297459</td>
<td>184.40</td>
<td>405825</td>
<td>163.70</td>
<td>236763</td>
</tr>
<tr>
<td>41/9</td>
<td>200.26</td>
<td>340494</td>
<td>179.02</td>
<td>457011</td>
<td>159.52</td>
<td>249794</td>
</tr>
<tr>
<td>4/1</td>
<td>191.19</td>
<td>468727</td>
<td>172.64</td>
<td>527791</td>
<td>155.54</td>
<td>369183</td>
</tr>
<tr>
<td>39/11</td>
<td>186.5</td>
<td>652865</td>
<td>164.94</td>
<td>754533</td>
<td>150.20</td>
<td>528438</td>
</tr>
<tr>
<td>19/6</td>
<td>169.09</td>
<td>781549</td>
<td>157.58</td>
<td>867552</td>
<td>144.29</td>
<td>764153</td>
</tr>
</tbody>
</table>

As stated earlier we had 3000 searches (100 initial states with 30 different values for W). In each such tree we can compare the behavior of the tested algorithms. Table 2.3 presents the percentage of instances of the 3000 in which each algorithm beats BFS. The following data is included in the columns of the table:

- **Both**: represents the percentage of instances out of 3000 that an algorithm generates both fewer nodes and a shorter solution compared to BFS.
- **Only len**: counts the cases in which an algorithm produces a shorter solution length but generates more nodes.

Table 2.2 shows the percentage of instances out of 3000 that an algorithm generates both fewer nodes and a shorter solution compared to BFS. The figures in Table 2.2 present a comparison of different KBFS algorithms by plotting node generations as a function of solution length. Each curve corresponds to a different value of K. The data in Figure 2.7 is the same as that in Figure 2.6, but on a larger scale. The figures show that WA* is indeed outperformed by all other values of KWA*.
Figure 2.6: Path length versus generated nodes for twenty-four puzzle.

Figure 2.7: Path length versus generated nodes for the twenty-four puzzle. Larger scale.
Table 2.3: KBFS: winning and losing on the twenty-four puzzle.

- **Only Gen**: Fewer nodes but longer solution.
- **None**: More nodes and longer solution.
- **Len**: This column counts the percentage of search trees for which the algorithm produced a shorter solution compared to KBFS1 without taking into consideration how many nodes were generated. This column is the sum of the first two columns.
- **Generated**: Counts the percentage that each algorithm generated fewer nodes regardless of the solution length. This number is the sum of the first and third columns.

If when comparing two algorithms we define a *win* as a case that one algorithm generated fewer nodes and a shorter solution and a *loss* as vise versa, then the Both and None columns of Table 2.3 represent winning and losing. If we compare the Both column to the None column we see that KBFS2 is the worst KBFS on the twenty-four puzzle. It outperforms KBFS1 in 31.41% of the cases while it loses in 25.48%. Still, it wins more than it loses. KBFS30 for example wins 49.93% and loses only 2.44%. From the width of 400 and up, KBFS always produces shorter solutions, often with fewer numbers of generated nodes as well.

### 2.4.2 KBFS results on the 15 puzzle

On the fifteen puzzle we tested each K on 44 different ratios of W. Each pair of K and W (KWA*) was tested on the 100 random instances from [26]. Here, the improvement factor of KWA* over WA* was 2-3. For example, BFS finds a solution of length 78.41 moves at a cost
of 22,840 node generations, while KBFS(50) finds a path of length 77.41 moves with only 9,987 generated nodes. For a path with a length of 68 moves BFS generates 45,736 nodes while KBFS(100) generates only 18,896 nodes. The data of winning and losing of the fifteen puzzle was not significantly different from that of the twenty-four puzzle. We do not present the data of the fifteen puzzle in more detail since the fifteen puzzle can be solved optimally in a matter of minutes by techniques like IDA* or RBFS, so near-optimal solutions are no longer interesting for this version of the puzzle.

The results on the twenty-four puzzle are significantly better than on the fifteen puzzle. This may be due to the fact that errors in the twenty-four puzzle are more serious since their discrepancies from the real value are greater. Therefore, the corresponding subtrees with errors are larger. These results correlate with the results of the dead-end subtrees that show that where the subtrees with errors are larger, KBFS is more effective. The KBFS technique is not only more powerful on larger problems, but is also more valuable there, since finding an optimal solution for large problems can be very expensive. For example, some instances of the twenty-four puzzle take weeks to solve optimally, even though the heuristic function is more sophisticated than Manhattan distance. In such cases, settling for a near-optimal solution is the only practical way to solve a problem within reasonable time constraints.

We also tried to solve the 6x6 35 puzzle with KWA*. Nevertheless, even with pure heuristic search none of the algorithms could solve an entire set of 100 random problem instances with the available memory, which could save up to 9 million nodes. Only 85% of the instances could be solved and the partial results on these 'easy' instances show the same tendency, namely KWA* outperforms WA*.

## 2.5 KBFS on number partitioning.

Neither the sliding-tile puzzles nor the random tree are real-world problems. Furthermore, since KBFS does not return optimal solutions, it might be compared with other algorithms which also do not guarantee optimum solutions. For these reasons, we implemented KBFS on the number partitioning problem and compared it to hill-climbing which also does not guarantee optimal solutions.

Given a set of N numbers, the two-way number partitioning problem is to divide them into two disjoint subsets, such that the sum of the numbers in each subset are as nearly equal as possible. This problem has applications in the real world. For example, given two machines, a set of jobs and the time required to solve each job on either machine, the task is to assign the jobs to the machines such that all jobs are completed in the shortest elapsed time. This problem is known to be NP-complete [15].

There are many algorithms that were specially designed to solve this problem. One of
these algorithms is the set differencing algorithm also known as the KK [20] heuristic. The KK algorithm first sorts the numbers. Then, it removes the largest two numbers, computes their difference and treats the difference as another number, adding it to the sorted list of numbers. The rationale behind this move is that putting one of these numbers in one subset and the other one in the other subset is equivalent to putting their difference in one of the subsets. KK stops when there are no numbers left in the sorted list. KK runs in $O(n \log n)$ time, but doesn’t return optimal solutions.

A fast optimal algorithm was proposed in [31]. It was called CKK which stands for Complete KK algorithm. The basic KK takes the largest two numbers and places their difference in the sorted list indicating that these numbers are being placed in different subsets. The only other option is to place their sum in the sorted list to indicate that both numbers are being placed in the same subset. CKK explores both these options and thus finds an optimal solution to the problem. It outperforms all known algorithms for the number partitioning problem. Our goal in implementing KBFS on this problem is not to try to compete with algorithms that are specially designed for this problem, such as CKK, but to check whether increasing K improves the search in this domain.

### 2.5.1 The search tree for number partitioning

There are $2^N$ possible ways to partition N numbers into two disjoint sets. A state in our search space is a complete partition of the numbers into two disjoint sets. The cost of a state is simply the difference between the sum of the numbers in each of the subsets.

We specify two types of neighbors of a state:

1. A single number may be moved to the other subset while all the other numbers remain stationary. Thus the number of neighbors of this type for every state is N.

2. Sometimes, moving a number from subset A to subset B will produce a large increase in the cost of a state. However, moving another number from B to A will cancel this increase and might even decrease the cost. In other words, sometimes swapping two numbers might decrease the cost. Thus if moving a number to the other subset increases the total cost we look at all the possibilities to swap this number with numbers from the other subset. We therefore also consider all these swapping moves as generating neighbors of the original state.

Figure 2.8 illustrates both types of neighbors. There are three numbers to partition with values of 1, 2 and 3. In the root node, subset A includes 1 and 3 while subset B includes 2. Moving 1 to subset B is a neighbor of type 1, and the cost decreases from 2 to 0. This scenario is presented in the left child. Moving 3 to the other subset increases the cost from...
2 to 5. Thus, any possible way to swap 3 with members of subset B is also considered a
neighbor. In particular swapping 3 with 2 will decrease the cost from 2 to 0 as illustrated in
the right child. When many neighbors of type 1 increase the cost then the branching factor
of our search space is very large and can contain up to \( N + \left( \frac{N^2}{4} \right) \) different neighbors.

KBFS is a very simple and general algorithm for traversing a search tree. Thus, a
candidate algorithm for comparison should also be simple and general. We have chosen to
implement hill climbing as it is the simplest local search algorithm and can be implemented in
almost every domain. Hill climbing is much better than KBFS in its memory requirements
which is \( O(N) \) where \( N \) is the number of numbers to be partitioned. Hill climbing was
implemented with random restart as follows. We first generate a random state. Then, we
generate all its neighbors and choose one with the lowest cost. If this cost is better than
the previous cost we discard all other neighbors and perform this process on the new node.
This process is continued until a local minimum is reached. Then, we generate another
random state and perform this search again until another local minimum is reached. This
hill-climbing procedure is repeated until some external condition is reached and then returns
the best solution found. In our experiments the terminating condition was a bound of 6
million nodes generated. Once this number of nodes were generated the search stopped.
The reason that we picked this bound is because KBFS stores all generated nodes in an
open list simultaneously and this is the number of nodes that we could store in our memory.

After hill-climbing returned its best solution, we moved to KBFS. We took the same
initial random partitionings that were generated by hill-climbing and inserted them into
the open list. Then we activated KBFS with different values for K on this open list. In
this way both hill-climbing and KBFS start the search from the same set of nodes. Hill-
climbing performs local search for each such node and tries to reach a local optimum in its
neighborhood. KBFS searches all these nodes simultaneously while comparing the cost of all
these nodes and choosing the best K nodes to be expanded. KBFS also stops after 6,000,000
nodes were generated and returns the best solution that was found during the search.

As stated above, KBFS might be efficient when there are errors in the evaluation function.
In this domain, a state X may have a very small cost but all its neighbors may have a much

Figure 2.8: Number partitioning: the different types of neighbors.
larger cost. In this case we can say that we have a *mistake* in the cost of state X, since it does not have a large correlation with its neighbors. Thus KBFS might be effective in this domain.

### 2.5.2 Empirical results

Table 2.4 presents the results of KBFS on number partitioning. Each row corresponds to a different algorithm. The columns correspond to the size of the set of numbers that were partitioned which was varied from 20 to 60. The numbers to be partitioned were randomized and uniformly distributed from 0 to 10 billion, i.e. they contain up to 10 decimal digits. The values in the table are the best solutions found by the algorithms after generating 6,000,000 nodes. The values are all averages of 100 instances of the same size.

The first column corresponds to a set of size 20. Since there are only \(2^{20} = 1,048,576\) different ways to partition such a set, the whole search space is spanned by all versions of KBFS and an optimal solution is obtained. With larger sets only suboptimal solution were obtained. While the difference between the solution of the different algorithms is not very large here we can see here a tendency that increasing \(K\), especially to large values such as 1000, improves the search and a better solution is found. For example, for problems of size 35, KBFS(1000) returned a solution which is less than half the size of the solution obtained by both hill-climbing and BFS. The advantage of hill-climbing over KBFS is the fact that it is easier to implement. Also, KBFS uses an open list and incurs the overhead of both time and memory needs for handling this list. If we only compare BFS to KBFS then once again, these results show that KBFS outperforms simple BFS.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>20</th>
<th>25</th>
<th>30</th>
<th>35</th>
<th>40</th>
<th>45</th>
<th>50</th>
<th>55</th>
<th>60</th>
</tr>
</thead>
<tbody>
<tr>
<td>hill-climbing</td>
<td>62883</td>
<td>6708</td>
<td>4919</td>
<td>6837</td>
<td>5389</td>
<td>5543</td>
<td>5460</td>
<td>5460</td>
<td>5480</td>
</tr>
<tr>
<td>BFS</td>
<td>62883</td>
<td>6175</td>
<td>6192</td>
<td>7605</td>
<td>7794</td>
<td>10007</td>
<td>7638</td>
<td>9118</td>
<td>10664</td>
</tr>
<tr>
<td>KBFS3</td>
<td>62883</td>
<td>6175</td>
<td>6192</td>
<td>7805</td>
<td>7966</td>
<td>9851</td>
<td>7551</td>
<td>7793</td>
<td>7630</td>
</tr>
<tr>
<td>KBFS10</td>
<td>62883</td>
<td>6175</td>
<td>6180</td>
<td>7859</td>
<td>7118</td>
<td>6740</td>
<td>7767</td>
<td>5879</td>
<td>6290</td>
</tr>
<tr>
<td>KBFS30</td>
<td>62883</td>
<td>6175</td>
<td>6093</td>
<td>6662</td>
<td>5367</td>
<td>5827</td>
<td>5763</td>
<td>6252</td>
<td>5834</td>
</tr>
<tr>
<td>KBFS100</td>
<td>62883</td>
<td>6124</td>
<td>6206</td>
<td>4049</td>
<td>5795</td>
<td>5373</td>
<td>4695</td>
<td>4621</td>
<td>5359</td>
</tr>
<tr>
<td>KBFS300</td>
<td>62883</td>
<td>5406</td>
<td>4252</td>
<td>3924</td>
<td>4580</td>
<td>4737</td>
<td>4715</td>
<td>4568</td>
<td>4809</td>
</tr>
<tr>
<td>KBFS1000</td>
<td>62883</td>
<td>4396</td>
<td>3723</td>
<td>3547</td>
<td>4333</td>
<td>4400</td>
<td>3514</td>
<td>3646</td>
<td>3987</td>
</tr>
</tbody>
</table>

Table 2.4: KBFS: comparison with hill-climbing on number partitioning. The values are the average cost of a solution after generating 6,000,000 nodes.
2.6 Conclusions and future work for the KBFS algorithm

We have shown a simple method to increase the effectiveness of best-first search by generalizing it to KBFS. We have implemented KBFS on various domains and the results confirm that expanding a number of nodes is more efficient than expanding only the best node. The reason is because errors in the cost function hurt simple best-first search. These errors do not affect KBFS as much because it adds diversity into the search.

KBFS is a simple and general formulation for expanding nodes from an open list. Many problems can be solved with algorithms that were specially developed for these particular problems. KBFS, as a general algorithm, may not compete with these hand-tooled algorithms. However, if one chooses to use a simple and general algorithm such as best-first search this paper suggests that one might also consider KBFS because it is as simple and general as best-first search but may outperform it with no additional cost of machine resources nor implementation complexity.

KBFS might be further explored in the following directions:

1. Determine under what circumstances KBFS outperforms BFS and what is the optimal value of $K$?

2. A linear or memory-bounded space version of KBFS must be introduced in order for KBFS to be widely used. We are currently working on generalizing RBFS and SMA* to their Generalized K version.

3. Best-First Minimax search results presented in [28] were inconclusive. We believe that K-Best-First Minimax might provide better results.

4. We would like to develop a parallel version of KBFS (PKBFS) in which $K$ processors expand the $K$ best nodes simultaneously.
Chapter 3

Near optimal perimeter search

Perimeter Search (PS) is an admissible algorithm that was developed recently to improve the efficiency of search algorithms. In PS, first, a breadth first search from the goal node is performed to depth $d$ and all the nodes surrounding the goal node at that depth are saved in a hash table. Then, a forward search from the initial state is performed until one of the nodes in the perimeter is reached. The heuristic function that is used is the estimation of the distance between a node and the closest node to it in the perimeter. The disadvantage of such an algorithm is the large number of heuristic evaluations that need to be calculated for each node. We introduce the near optimal version of perimeter search where the heuristic that is taken is the regular heuristic to the goal, while the search stops when reaching a node at the perimeter. We analyze the behavior of such an algorithm and the quality of the solution that it produces. We present empirical results that show that this algorithm outperforms other inadmissible algorithms like WA* and KWA* in the sense of path length versus generated nodes.

3.1 Memory bound search algorithms

The main drawback of A* is the fact that it usually exhausts the available memory before solving the problem. This is because A* stores all the nodes that it generates in memory and the number of such nodes grows exponentially with the depth of the search. This problem was addressed in the last two decades and many linear-space best-first searches such as IDA* [26] and RBFS [27] were developed to overcome the memory limitation. These algorithms actually perform different variations of depth-first search using memory which is only linear in the depth of the search. During the past decade however, the size of computer memories have continuously become larger and current machines comprise typical capacities of up to 1 Gigabyte of main memory. This can not help pure A* which still exhausts the memory.
when it addresses large problems. On the other hand the linear space algorithms do not take full advantage of the large memory available. Many memory-bound algorithms such as MA*, SMA* and MREC were developed to intelligently use the available memory in order to improve the efficiency of the search. These algorithms are summarized in [27].

They all save a portion of the open list, as large as possible, in the memory and provide different methods of proceeding when the memory is exhausted. The new method of pattern database where specific heuristic values are kept in a large table [10] [25] can also be viewed as a new memory bound search.

### 3.2 Perimeter Search

Perimeter Search (PS) is an admissible algorithm that was developed by Delinberg and Nelson [13] to improve the efficiency of search algorithms by taking full advantage of the available main memory. In PS, first, a breadth-first search from the goal node is performed to a fixed depth and then all the nodes surrounding the goal node at that depth (the perimeter nodes) are stored in a hash table. Then, a forward search from the initial state is performed until one of the nodes in the perimeter is reached. The heuristic function that is used is the estimation of the distance between a node and the closest node to it in the perimeter.

We now present definitions and terms that will be used in the following sections. We assume that there exists an admissible heuristic function \( h(m, n) \) between any 2 states \( m \) and \( n \). For example, the heuristic value between an arbitrary node \( n \) to the goal state \( g \) will be referred to as \( h(n, g) \). We will use \( P \) for the set of nodes that are in the perimeter. We will use the term \( h(n, P) \) as \( \min_{x \in P} h(n, x) \) i.e. the distance from \( n \) to the closest node on \( P \) to \( n \) that is suggested by the heuristic function. We will use \( S \) to denote a regular Search algorithm, PS for a Perimeter Search which its forward search is done with \( S \). Finally, we will use NOPS for our new algorithm Near-Optimal Perimeter Search which we present below.

The heuristic function that is used by the forward search of PS is \( h(n) = h(n, P) + d \) where \( d \) is the depth of the perimeter. This heuristic function is more accurate than \( h(n, g) \) because it estimates only the distance from \( n \) to \( P \), whereas the distance from \( P \) to the goal, \( d \), is taken exactly. Another improvement of PS is the fact that it halts when reaching one of the perimeter nodes and does not continue until the goal node is reached. The tradeoff is that calculating this new heuristic function involves many evaluations (one for each node of the perimeter since we need the minimum among them). Perimeter search can use any kind of admissible search algorithm for its forward search. In the following sections we will talk about perimeter search with A* as its forward search PA* and its near-optimal version NOPA*. We will also deal with IDA* (PIDA* and NOPIDA*) and RBFS (PRBFS and...
A comprehensive analysis of the behavior of Perimeter Search and its admissibility is presented in [13] and readers not familiar with their work are encouraged to first see that paper. Nevertheless, our text can be read independently. We will now present and analyze the improvements of perimeter search over regular search which are important for our Near-Optimal Perimeter search which we present below. For example, we will use A* and PA*.

The forward search in PA* has the following improvements over regular A*:

1. The search in PA* stops sooner than A* when it reaches one of the nodes in the perimeter and does not have to wait until the goal node is found as in A*. The solution will be the path from the initial state to that node on the perimeter (which we will call p) plus the path from that p to the goal state g.

2. In order to guarantee admissibility the heuristic which is used is the minimum distance estimation from the current node n to any node on the perimeter i.e. \( \min_{x \in P} h(n, x) \) which will provide an optimal path to the perimeter. Adding this to the depth of the perimeter will provide an optimal path to the goal. The cost function of PA* for a node n is \( f(n) = g(n) + h(n, P) + d \) where d is the depth of the perimeter. This heuristic is more accurate than the ordinary heuristic since the path from the perimeter to the goal is calculated exactly and is not underestimated.

These two improvements cause PA* to examine less nodes than A* since it uses a better heuristic and it stops the search at a shallower level. The tradeoff for this method is however, the large number of heuristic evaluations that should be calculated for each generated node. In particular, for each generated node n, PA* evaluates the estimated distance from that node n to all the nodes in the perimeter in order to find the minimum among them. This number of evaluations is exponential in the depth of the perimeter. When a large perimeter is used, the heuristic function is more accurate and thus a smaller number of nodes are generated. On the other hand the time to evaluate such a heuristic function may surpass the improvement in the number of nodes generations. As shown in [13], for each domain there exist an optimal perimeter which is domain dependent and is a function of several variables concerning the domain. The optimality is the time complexity of the search algorithm which takes into account both the number of generated nodes and the number of heuristic evaluation for each node. A formula to calculate the optimal perimeter can be found in [13]. For example, the optimal perimeter for the fifteen puzzle was proved to be a perimeter with a depth of 3, both
analytically and empirically in [13]. With such a perimeter, they report an improvement of a factor of 3 in the time needed to solve a random instance of the fifteen puzzle.

An improved perimeter search, BIDA*, was introduced by Manzini [35] in order to reduce the number of heuristic evaluations to nodes in the perimeter. In BIDA*, evaluations are made only to nodes in an active set from the perimeter, which are essential for maintaining admissibility. This active set becomes smaller and smaller as the search proceeds. In his experiments the improvement of using a perimeter increases with larger perimeter and he obtained an improvement of a factor in the total time required to solve the fifteen puzzle when using a perimeter of depth 14.

The main drawback of PS algorithm is the fact that an optimal perimeter exists and when using larger perimeters the effectiveness of the algorithm drops even though a smaller number of nodes are generated. If we define a full memory algorithm to be one such that all the available memory is intelligently used, then in that sense this algorithm is not full memory. This is because perimeters which are larger than the optimal perimeter should not be used even if there is a great portion of free memory. In BIDA*, this drawback is not present and it can be observed as a memory bound algorithm. However it is not known whether BIDA* will continue to improve with perimeters that are larger than 14.

3.2.2 Complexity of generating the perimeter

We should also take into consideration the overhead complexity of generating the perimeter which is exponential in the depth of the perimeter. However, for very large problems this overhead is insignificant. Also, once the perimeter was generated it can be used for a great number of search tasks (if they share the same goal node) and its time complexity can be amortized over many problem instances. Also, there is no need to generate the perimeter even for each session of problem instances. The perimeter can be generated only once and can be saved on the disk which yields a better constant time for making the hash table.

3.3 The near-optimal version of perimeter search.

We now turn over to describe the Near-Optimal perimeter search algorithm (NOPS).

3.3.1 Perimeter search with weighting the heuristics

Weighted A* (WA*) [42] [16] was developed to reduce the complexity of A* by using

\[ f(n) = w_g * g(n) + w_h * h(n) \]
as its heuristic function. The ratio of \( w_n/w_g \) is labeled as \( W \) where \( 1 \leq W \leq \infty \). The cost function can therefore be written in the following simplified version: \( f(n) = g(n) + W \cdot h(n) \).

The two extreme cases are \( A^* \) when \( W = 1 \) and are pure heuristic search when \( W \) is very large and actually the cost function is only \( h(n) \). When \( W > 1 \) the overall cost function is inadmissible since it may overestimate the real distance. It was shown [27] that increasing \( W \) (i.e. giving more weight to \( h \) and less to \( g \)) results in finding solutions faster in terms of the number of generated nodes at the expense of an increased solution length. Actually the behavior of \( WA^* \) can be seen as paying a higher price for acquiring shorter solutions. It was proven [17] that the solution length of \( WA^* \) cannot be greater than the optimal solution by a factor of \( W \) while empirical results [27] show that in practice the solution of \( WA^* \) is much smaller than that upper bound of \( W \).

There may be modifications of \( PS \) and \( BIDA^* \) for suboptimal solutions, namely using the \( WA^* \) weighting mechanism for the forward search. However, for \( PS \) it seems irrational to calculate all these heuristics and then compute the weighted version of them. As for \( BIDA^* \), the reduction of the number of evaluations of \( BIDA^* \) is based on the fact that the function is monotonic. This is not the case for heuristics which are based on \( WA^* \) and are nonmonotonic. Therefore, the \( BIDA^* \) mechanism of using an active set for reducing the number of evaluations would not work with weights on the heuristics because the cost function is nonmonotonic.

### 3.3.2 Simple Near-Optimal Perimeter Search

Our new algorithm the near-optimal perimeter search (NOPS) uses only the first improvement of perimeter search that is stated above. In this version of perimeter search the heuristic function is the usual heuristic which is used by regular search algorithms of the estimated distance from a node \( n \) to the goal node \( g \), \( h(n, g) \) and not \( h(n, P) + d \) as in \( PS \). However, like \( PS \), in NOPS the search halts when there is a match between a new generated node and a node from the perimeter. The only difference between this algorithm (NOPS) and the corresponding regular algorithm \( S \) is the condition for halting the algorithm. In \( S \) each node which is selected for expansion is first matched against the goal node. If it is the goal node the algorithm halts and the current path to that node is returned as a solution. In NOPS this node is matched against all the nodes of the perimeter which are saved in a hash table with a constant time complexity. If a match is found NOPS halts and returns the current path to that node plus the path from that node to the goal node which was already calculated by the breadth first search that generated the perimeter.
### 3.3.3 Time complexity of NOPS

In NOPS a constant time is needed for processing each generated and/or expanded node. The match of a new node against the perimeter nodes that are saved in a hash table can be performed in a constant time just like the test in S which is performed to check whether a node is a goal node. Calculating the heuristic of a node can also be performed in a constant time. Therefore, the time complexity of NOPS can be measured by the number of generated nodes just like S. This is in contrast to PS where the complexity should also take into consideration the complexity of calculating the $h(n,P)$ heuristic function which depends on the size of the perimeter. The nodes that are generated by this algorithm will be exactly the same nodes that are generated by the regular search and in the same order. The only difference is that in NOPS, the search will stop earlier when the first node from the perimeter is reached. This implies that the number of generated nodes in NOPS will be smaller than that of S since all the nodes which should be generated by S after the first node of the perimeter is reached will not be generated by NOPS. When a larger perimeter is used the solution will be found faster since the perimeter is larger and a match to a node in the perimeter will eventually occur more quickly. Therefore, in the sense of time complexity larger perimeters improve the behavior of NOPS and the depth of the perimeter should be the larger perimeter that can be stored in memory. NOPS can be viewed as a memory bounded algorithm in contrast to PS which has an optimal perimeter depth as was stated above.

![Figure 3.1: Nonoptimality of perimeter search.](image-url)
3.3.4 Solution quality of NOPA*

If we use the above analysis in the context of A*, then NOPA* generates a smaller number of nodes than A*. However, even if the heuristic is admissible the solution returned by NOPA* may not be optimal since it only generates an optimal solution to the node in the perimeter that was reached. This node may not be in an optimal path to the goal state. Fig. 3.1 illustrates this situation.

Comparison of A*, NOPA* and PA*

Let’s examine the behavior of A*, NOPA* and PA* on the graph which is illustrated in Fig. 3.1. I is the initial state, A and B are nodes in the perimeter of depth 3 and G is the goal state. Both A* and NOPA* expand I and generate nodes A and B. When calculating $f(n) = g(n) + h(n)$ then the cost function of A is $f(A) = 5 + 1 = 6$, while that of B is $f(B) = 4 + 3 = 7$. Node A is now chosen for expansion, it is matched against the perimeter nodes and NOPA* halts. The solution returned is I, A, C, G with a cost of $g(A) + d = 5 + 3 = 8$. The problem here is that the heuristic value of A is inaccurate and underestimates the correct distance by 2. A* would expand A generating C. C has an accurate heuristic and the overall cost of C is $f(C) = 6 + 2 = 8$. A* would then backtrack from the mistake of choosing A and move over to B which has $f(B) = 7$. This will lead A* to provide an optimal solution of 7 but of course entailing more work than NOPA*. PA* (the regular perimeter search with A*), evaluates $f(A) = g(A) + h(A, P) + d = 5 + 0 + 3 = 8$. For B it evaluates $f(B) = g(B) + h(B, P) + d = 4 + 0 + 3 = 7$. PA* will choose to visit B which is an optimal solution and will not expand A as both A* and NOPA* did.

Theoretical analysis of the solution quality of NOPA*.

**Theorem 1**: Let $D$ be the depth of the perimeter and let $H_P$ be the minimum heuristic value of a node from the perimeter to the goal. i.e. $H_P = \min_{n \in P} h(n, G)$. Let $P_{opt}$ be the length of the optimal solution. The solution length returned by NOPA*, $P_{NOPA*}$, will always be bounded as follows:

$$P_{opt} \leq P_{NOPA*} \leq P_{opt} + (D - H_P)$$

$D - H_P$ can be viewed as $M_P$, the maximum mistake of heuristic values among nodes in the perimeter, and therefore the above theorem claims that $P_{opt} \leq P_{NOPA*} \leq P_{opt} + M_P$.

1Note that the heuristic function gives values to nodes of the perimeter despite the fact that these nodes are in the perimeter. These values are used when these nodes are generated. The fact that a node is in the perimeter is taken into consideration only when that node is chosen for expansion.

2Node A has a heuristic value of 1 even though the real distance to the goal is 3. The correct distance is not used yet since the match to the perimeter nodes is done only when a node is chosen for expansion.
**Proof:** Since NOPA* uses an admissible heuristic, the values are never overestimated. Therefore $0 \leq H_P \leq D$ and thus $D - H_P \geq 0$. As in A*, the overall cost function of nodes in the search is always monotonic increasing and is never greater than the optimal path length. Let $n$ be the first node in the perimeter which is reached by NOPA* (When choosing this node for expansion, NOPA* halts.) and let $h(n)$ be the heuristic value of $n$, i.e., $h(n) = h(n, G)$. The path returned at this point will be $g(n) + D$, while the cost function of $n$ is $f(n) = g(n) + h(n, G) = g(n) + h(n)$.

Now, since the cost function of A* and therefore also of NOPA* always fulfills $f = g + h \leq P_{opt}$ then

$$g(n) + h(n) \leq P_{opt}$$

$$g(n) + h(n) + D - D \leq P_{opt}$$

$$g(n) + D \leq P_{opt} + D - h(n).$$

In the worst case we can take $H_P$, the minimum heuristic value of nodes in the perimeter and we get

$$P_{NOPA*} \leq P_{opt} + (D - H_P).$$

If we define $m(n)$ to be the mistake of the heuristic of node $n$ i.e., the difference between the real distance and the heuristic value of a node, then for node $n$ in the perimeter $m(n) = D - h(n)$. Therefore we obtain

$$P_{NOPA*}(n) = g(n) + D \leq P_{opt} + m(n).$$

If $M_P$ is the maximal mistake we can complete the proof and attain

$$P_{NOPA*}(n) = g(n) + D \leq P_{opt} + M_P.$$  

The above theorem gives us an upper bound to the quality of the solution. However we believe that the average solution will be much shorter. Since the optimal solution length to the goal is $P_{opt}$ then the optimal path from $S$ to any node in the perimeter is $P_{opt} - D$. When NOPA* arrives at the first node in the perimeter, $n$, then

$$P_{opt} - D \leq g(n) \leq P_{opt} - D + m(n).$$

The right side of the inequality is the worst case and the left side is the best case. In the average case we would expect $g(n) = P_{opt} - D + \frac{m(n)}{2}$. When taking $m(n)$ as $\bar{M}$, the average mistake of heuristic values of the nodes of the perimeter, we should obtain that at the average case $g(n) = P_{opt} - D + \frac{\bar{M}}{2}$ and the average solution length of NOPA* should be $g(n) = P_{opt} + \frac{\bar{M}}{2}$. However, since tie breaking usually prefers nodes with smaller heuristics,
such nodes are more likely to be visited first. Also there are much more of these nodes since they are deeper in the tree and thus using $\frac{M}{2}$ as the average case would be incorrect and thus nodes at deeper levels with larger mistakes are more likely to be visited. Therefore, we expect that the average $g(n)$ will be larger. We expect $g(n) \approx P - D + \bar{M}$ and the average solution returned by NOPA* should be $P_{NOPA} \approx P + \bar{M}$. In the following sections we will try to calculate this $\bar{M}$ and empirically measure the solution quality of NOPS with respect to $\bar{M}$.

### 3.4 Empirical results of NOPA* on the fifteen puzzle

NOPS is valuable for obtaining near optimal solutions. While finding optimal solutions for the fifteen puzzle can be accomplished very quickly with current techniques, we chose to make our experiments on this version of the puzzle and not on larger versions because on this version we can attain a large range of results and consequently we can better understand the behavior of NOPS with different perimeters and variables.

#### 3.4.1 Heuristic of nodes in the perimeters of the fifteen puzzle

In order to check the above theoretical claims regarding the quality of NOPS and the
fact that it depends on the mistakes of the heuristic function we should evaluate these mistakes. For the fifteen puzzle we counted the deviations (mistakes) of the Manhattan distance heuristic of nodes of different perimeters from the real distance to the goal, which is actually the depth of the perimeter. The results are presented in Table 3.1. We used different perimeters from 0 to 21. The results are averaged on the 100 random instances of the fifteen puzzle from [26]. The rows are the different perimeters from 0 to 21. The columns are as follows:

- The first column provides the depth of the perimeter.
- The second column, 0, presents the number of nodes in that perimeter with no mistakes at all i.e. the heuristic value is accurate and was exactly the depth of the perimeter.
- The next column shows the number of nodes in the perimeter with a mistake of 2 (i.e. it counts the number of nodes that have a Manhattan distance smaller by 2 than their real distance to the goal which is the depth of the perimeter) Then 4, 6 until 16.
- The TOTAL column counts the total number of nodes in that perimeter.
- The AVR column presents the average mistake.
- The MAX column shows the maximum mistake for that perimeter.

For example at perimeter 20 there are 232256 nodes. The maximum mistake is 16 i.e. there exists a node with a heuristic value of 4. The average mistake of the heuristic values of this perimeter is 3.94. Note that there are no mistakes in nodes in depth 1-5 and that the Manhattan distance is accurate for these depths. When reaching a node in these depth we are guaranteed to have an optimal solution.

### 3.4.2 Results of NOPIDA* and NOPRBFS on the fifteen puzzle

The most natural candidate for the forward search is A*. However, since A* runs out of memory very quickly, we used IDA* and RBFS which both simulate A* but use a linear amount of memory. The heuristic that was used was the regular admissible heuristics $f(n) = g(n) + h(n)$ where $h(n)$ is the Manhattan distance to the goal node.

We ran NOPS with perimeters from 0 to 21. For the forward search we tested both NOPIDA* and NOPRBFS and the results are presented in Table 3.2. For each algorithm we include both the solution length and the number of generated nodes averaged over the 100 random instances of the fifteen puzzle from [26]. Table 3.2 shows that both NOPIDA* and NOPRBFS provide roughly the same results and as the depth of the perimeter increases they
Table 3.2: Mistakes of the heuristics of nodes in the fifteen puzzle.

<table>
<thead>
<tr>
<th>Perimeter</th>
<th>PATHA* nodes</th>
<th>PATHBFS nodes</th>
<th>mistakes of NOPRBFS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>53.05</td>
<td>35003890</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>53.05</td>
<td>35003890</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>53.05</td>
<td>35003890</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>53.05</td>
<td>35003890</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>53.05</td>
<td>35003890</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>53.05</td>
<td>35003890</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>53.05</td>
<td>35003890</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>53.05</td>
<td>35003890</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>53.05</td>
<td>35003890</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>53.05</td>
<td>35003890</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>53.05</td>
<td>35003890</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>53.05</td>
<td>35003890</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>53.05</td>
<td>35003890</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>53.05</td>
<td>35003890</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>53.05</td>
<td>35003890</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>53.05</td>
<td>35003890</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>53.05</td>
<td>35003890</td>
<td>0</td>
</tr>
<tr>
<td>17</td>
<td>53.05</td>
<td>35003890</td>
<td>0</td>
</tr>
<tr>
<td>18</td>
<td>53.05</td>
<td>35003890</td>
<td>0</td>
</tr>
<tr>
<td>19</td>
<td>53.05</td>
<td>35003890</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>53.05</td>
<td>35003890</td>
<td>0</td>
</tr>
<tr>
<td>21</td>
<td>53.05</td>
<td>35003890</td>
<td>0</td>
</tr>
</tbody>
</table>

Tend to generate smaller number of nodes while the solution length increases. For example, a perimeter of 21 the number of node generations was improved by a factor of 100. The solution length was longer than the optimal solution by no more than 8%.

The rest of the columns include data from the results of NOPRBFS. They all refer to deviation (mistake) from the optimal. The columns are as follows:

- The **AVR** column provides the average mistake of nodes in the perimeter.

- The **DIF** column presents the difference between the average solution length return by NOPRBFS for that perimeter from the average optimal solution length which is 53.05.

- The **MAX** column shows the maximum mistake for that perimeter.

These results support the claim that the deviation of the provided solution length from the optimal solution will be roughly the same as the average mistake of nodes in that perimeter as we predicted earlier.

Fig. 3.2 presents a graph of the last 3 columns of Table. 3.2 The curve of the actual average solution provided by PRBFS for the fifteen puzzle is always below the maximum mistake for that perimeter and is very close to the average mistake.
Fig. 3.3 illustrates both algorithms in the sense of the solution depth versus the number of generated nodes. There is a curve for each algorithm and each point in the graph represents a different perimeter.

Notice from both Fig. 3.2 and Table 3.2 that on small perimeters up to 5 there is no deviation from the optimal solution and since there are no mistakes in heuristics in nodes on these perimeters and therefore the maximum mistake is 0. Thus the deviation of the solution provided by NOPA* must be optimal.

3.5 NOPS and other near optimal algorithms

By now we know that NOPS can solve problems faster when using larger perimeters at the expense of an increased solution length. However, we must compare the performance of NOPS to other inadmissible algorithms such as WA* or WRBFS in order to see which algorithm offers better prices for a given solution length. The cost function that we used above was the usual admissible heuristic \( f(n) = g(n) + h(n) \). However, since NOPS provides only near optimal solutions then admissible heuristic is not necessary anymore and thus we can also try to combine both NOPS and an inadmissible cost function such as WA*.

3.5.1 Comparing WRBFS to pure PRBFS

Weighted A* (WA*) [42] [16] was developed to reduce the complexity of A* by using

\[
 f(n) = w_g \cdot g(n) + w_h \cdot h(n)
\]
as its heuristic function. The ratio of \( \frac{w_h}{w_g} \) is labeled as \( W \) and varies from 1 (\( A^* \)) to \( \infty \) (pure heuristic search). It was shown \([27]\) that increasing \( W \) (i.e. giving more weight to \( h \) and less to \( g \)) results in finding solutions faster in terms of the number of generated nodes at the expense of an increased solution length. It was proven \([17]\) that the solution length of \( W A^* \) cannot be greater than the optimal solution by a factor of \( W \) while empirical results \([27]\) show that in practice the solution of \( W A^* \) is much smaller than the upper bound of \( W \).

RBFS generates nodes in a best first manner also for inadmissible heuristics \([27]\). And indeed the weighted version of RBFS (WRBFS) (i.e. where the cost function for RBFS is \( f(n) = g(n) + W * h(n) \)) shows the same tendency as \( W A^* \). i.e., increasing \( W \) results in finding solutions faster in terms of the number of generated nodes at the expense of an increased solution length.\(^3\)

Since both WRBFS in \([27]\) and NOPRBFS in our results above provide near optimal solutions it will be interesting to compare their behavior. By NOPRBFS we mean RBFS with perimeters (no weightings) and by WRBFS we mean RBFS with weightings (no perimeters). We ran these algorithms over the 100 problem instances of the fifteen puzzle and Fig. 3.4 illustrates a comparison of these two algorithms in this domain. Since both NOPRBFS and WRBFS are generalizations of RBFS they both have RBFS as a trivial special case. In NOPRBFS when the perimeter is 0 and in WRBFS when \( W = 1 \), both algorithms converge and provide an optimal solution. We can see that WRBFS outperforms PRBFS for solutions from 53-56. PRBFS is better for solutions that are longer than 56.

\(^3\)Weighted \( IDA^* \) (\( WIDA^* \)) did not perform well since \( IDA^* \) is a Best-First search only for admissible heuristics, See \([27]\).
3.5.2 Weighted NOPRBFS (WNOPRBFS).

NOPRBFS alone does not really outperform WRBFS. However, since the weighted heuristic function is a good method for acquiring inadmissible solutions, we can combine both NOPS and weighted heuristic function techniques. If we use RBFS for the forward search, the resulting algorithm would be labeled by WNOPRBFS. Now we have 2 independent parameters, namely the depth of the perimeter and the weights assigned to the heuristic function. When using weights, WRBFS needs to generate more nodes in order to attain a better solution length. We would like to know whether using perimeters in addition to weights (NOPWRBFS) would outperform WRBFS.

We ran WNOPRBFS for the 21 perimeters and various values of W for the 100 instances of the fifteen puzzle. The results are presented in Table 3.3. Note that when the perimeter depth is 0 (The first pair of result columns) it is the special case of WRBFS. With a perimeter of depth 21 WNOPRBFS outperforms WRBFS by a factor of up to 100. For example, for a path of 112 WNOPRBFS0 (When the perimeter is 0), which is actually WRBFS, needs 689,468 nodes while NOPRBFS21 needs only 20071. For a path of 83.09 NOPRBFS0 needs 551,144 while NOPRBFS21 needs only 5,370 nodes. The graph in Fig. 3.4 is actually a comparison of the left column of Table 3.3 and the bottom row of Table 3.3.

The phenomena of WRBFS, namely having larger number of nodes generated when both Wg and Wh are large integers, can be explained by the fact that for such values there are
more global cost values and therefore less tie breaking. This phenomena was explained in [27]. The first column of Table 3.3 where the perimeter is 0 is actually pure RBFS with weights on the cost function. Our results for this column are slightly better than those of [27] since the tie breaking that we used in our work was a little better. For example, for finding the optimal solution of 53.05 [27] reports an average of over 545 million generated nodes while our version of RBFS needs less than 400 million.

Fig. 3.5 shows the performance of WNOPRBFS for a selection of perimeters Fig. 3.6 shows the same data but focuses on the solutions that are very close to the optimal. In general we can say that combining both WRBFS and NOPRBFS into a new algorithm WNOPRBFS creates a very powerful algorithm which outperforms each of them alone since it combines the advantages of both of them. WNOPRBFS is up to 100 times faster than either of them.

### 3.6 Combining both WA* KBFS and Perimeter Search

We have already seen that expanding the K best nodes in each iteration (KBFS) is an algorithm that outperforms Best-first search for inadmissible heuristic functions. Actually we have 3 ways of producing near optimal solutions, namely weighting the heuristic func-

<table>
<thead>
<tr>
<th>Pw</th>
<th>Pw</th>
<th>path</th>
<th>nodes</th>
<th>path</th>
<th>nodes</th>
<th>path</th>
<th>nodes</th>
<th>path</th>
<th>nodes</th>
<th>path</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>19</td>
<td>128.87</td>
<td>33133065</td>
<td>130.45</td>
<td>850163</td>
<td>124.17</td>
<td>7033355</td>
<td>119.97</td>
<td>527690</td>
<td>112.92</td>
</tr>
<tr>
<td>7</td>
<td>93</td>
<td>3475.87</td>
<td>2845817</td>
<td>124.57</td>
<td>803959</td>
<td>119.71</td>
<td>184721</td>
<td>117.23</td>
<td>45320</td>
<td>110.36</td>
</tr>
<tr>
<td>1</td>
<td>9</td>
<td>115.47</td>
<td>3651005</td>
<td>116.27</td>
<td>131940</td>
<td>114.07</td>
<td>626298</td>
<td>110.07</td>
<td>16870</td>
<td>106.44</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>112.45</td>
<td>608468</td>
<td>113.51</td>
<td>565511</td>
<td>112.49</td>
<td>88855</td>
<td>109.87</td>
<td>21373</td>
<td>106.28</td>
</tr>
<tr>
<td>17</td>
<td>7</td>
<td>107.27</td>
<td>250677</td>
<td>109.43</td>
<td>119423</td>
<td>108.73</td>
<td>46082</td>
<td>107.17</td>
<td>23859</td>
<td>104.63</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>102.27</td>
<td>331000</td>
<td>104.59</td>
<td>164717</td>
<td>103.69</td>
<td>57767</td>
<td>102.65</td>
<td>25537</td>
<td>100.66</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>96.23</td>
<td>193241</td>
<td>90.07</td>
<td>81013</td>
<td>98.19</td>
<td>26643</td>
<td>96.79</td>
<td>13264</td>
<td>95.04</td>
</tr>
<tr>
<td>9</td>
<td>4</td>
<td>91.00</td>
<td>506316</td>
<td>93.25</td>
<td>464636</td>
<td>93.51</td>
<td>80932</td>
<td>92.95</td>
<td>40109</td>
<td>90.68</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>87.81</td>
<td>208888</td>
<td>90.27</td>
<td>90498</td>
<td>90.53</td>
<td>36310</td>
<td>90.21</td>
<td>18118</td>
<td>87.68</td>
</tr>
<tr>
<td>11</td>
<td>39</td>
<td>83.00</td>
<td>551144</td>
<td>85.69</td>
<td>277949</td>
<td>86.13</td>
<td>116490</td>
<td>86.61</td>
<td>40774</td>
<td>84.91</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>77.43</td>
<td>167430</td>
<td>80.99</td>
<td>76006</td>
<td>81.55</td>
<td>20392</td>
<td>81.93</td>
<td>15410</td>
<td>80.68</td>
</tr>
<tr>
<td>18</td>
<td>18</td>
<td>71.57</td>
<td>1386546</td>
<td>74.81</td>
<td>688986</td>
<td>75.11</td>
<td>164905</td>
<td>75.35</td>
<td>103579</td>
<td>74.86</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>67.83</td>
<td>222347</td>
<td>71.01</td>
<td>206604</td>
<td>72.77</td>
<td>94546</td>
<td>72.35</td>
<td>52940</td>
<td>72.68</td>
</tr>
<tr>
<td>17</td>
<td>17</td>
<td>64.81</td>
<td>2223588</td>
<td>67.95</td>
<td>1355925</td>
<td>69.75</td>
<td>441199</td>
<td>70.45</td>
<td>176555</td>
<td>70.38</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>63.19</td>
<td>604782</td>
<td>66.29</td>
<td>244007</td>
<td>67.71</td>
<td>100541</td>
<td>68.95</td>
<td>54236</td>
<td>69.36</td>
</tr>
<tr>
<td>9</td>
<td>16</td>
<td>59.23</td>
<td>6853706</td>
<td>62.85</td>
<td>2729601</td>
<td>64.77</td>
<td>802568</td>
<td>65.23</td>
<td>338032</td>
<td>65.72</td>
</tr>
<tr>
<td>19</td>
<td>31</td>
<td>57.45</td>
<td>16655806</td>
<td>60.31</td>
<td>6218712</td>
<td>62.41</td>
<td>1535900</td>
<td>63.26</td>
<td>688900</td>
<td>63.62</td>
</tr>
<tr>
<td>30</td>
<td>61</td>
<td>56.59</td>
<td>43259753</td>
<td>59.91</td>
<td>14376107</td>
<td>61.49</td>
<td>2722388</td>
<td>62.45</td>
<td>1689806</td>
<td>63.09</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>55.83</td>
<td>13035040</td>
<td>58.83</td>
<td>4853659</td>
<td>60.61</td>
<td>714449</td>
<td>61.77</td>
<td>326645</td>
<td>62.60</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>54.51</td>
<td>36744687</td>
<td>56.51</td>
<td>14006538</td>
<td>58.55</td>
<td>3382170</td>
<td>59.53</td>
<td>1592470</td>
<td>59.98</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>53.85</td>
<td>105506613</td>
<td>55.85</td>
<td>37302229</td>
<td>57.41</td>
<td>8910064</td>
<td>58.85</td>
<td>3489409</td>
<td>59.22</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>53.06</td>
<td>295388909</td>
<td>54.87</td>
<td>111142550</td>
<td>55.91</td>
<td>22978655</td>
<td>56.67</td>
<td>1242181</td>
<td>57.32</td>
</tr>
</tbody>
</table>

Table 3.3: WNOPRBFS results on the fifteen puzzle.
<table>
<thead>
<tr>
<th>W = g/h</th>
<th>KBFS1</th>
<th>KBFS5</th>
<th>KBFS10</th>
<th>KBFS50</th>
<th>KBFS100</th>
<th>KBFS1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/49</td>
<td>144.20</td>
<td>7001</td>
<td>130.85</td>
<td>7499</td>
<td>112.91</td>
<td>7343</td>
</tr>
<tr>
<td>1/19</td>
<td>127.65</td>
<td>7924</td>
<td>114.71</td>
<td>8755</td>
<td>102.07</td>
<td>7735</td>
</tr>
<tr>
<td>7/93</td>
<td>121.90</td>
<td>7768</td>
<td>109.25</td>
<td>7683</td>
<td>98.17</td>
<td>7508</td>
</tr>
<tr>
<td>1/9</td>
<td>116.49</td>
<td>9527</td>
<td>110.53</td>
<td>7857</td>
<td>93.17</td>
<td>7185</td>
</tr>
<tr>
<td>1/1</td>
<td>109.23</td>
<td>10796</td>
<td>98.07</td>
<td>10156</td>
<td>89.89</td>
<td>8511</td>
</tr>
<tr>
<td>1/6</td>
<td>103.20</td>
<td>10460</td>
<td>94.17</td>
<td>11309</td>
<td>86.37</td>
<td>10059</td>
</tr>
<tr>
<td>1/5</td>
<td>97.31</td>
<td>12689</td>
<td>90.23</td>
<td>13448</td>
<td>82.97</td>
<td>11928</td>
</tr>
<tr>
<td>1/4</td>
<td>88.15</td>
<td>15181</td>
<td>82.81</td>
<td>14966</td>
<td>78.37</td>
<td>13281</td>
</tr>
<tr>
<td>1/3</td>
<td>78.41</td>
<td>22840</td>
<td>75.13</td>
<td>24905</td>
<td>71.81</td>
<td>21045</td>
</tr>
<tr>
<td>7/18</td>
<td>71.71</td>
<td>38255</td>
<td>69.63</td>
<td>35287</td>
<td>67.45</td>
<td>27564</td>
</tr>
<tr>
<td>3/7</td>
<td>68.43</td>
<td>45736</td>
<td>66.51</td>
<td>47226</td>
<td>65.41</td>
<td>44676</td>
</tr>
<tr>
<td>1/7</td>
<td>64.93</td>
<td>62177</td>
<td>64.09</td>
<td>60659</td>
<td>63.31</td>
<td>57869</td>
</tr>
<tr>
<td>1/6</td>
<td>61.53</td>
<td>78870</td>
<td>62.81</td>
<td>72826</td>
<td>61.83</td>
<td>71914</td>
</tr>
<tr>
<td>1/5</td>
<td>59.47</td>
<td>84763</td>
<td>63.84</td>
<td>59371</td>
<td>57.97</td>
<td>54027</td>
</tr>
<tr>
<td>1/4</td>
<td>56.61</td>
<td>93634</td>
<td>64.51</td>
<td>48976</td>
<td>63.37</td>
<td>51586</td>
</tr>
<tr>
<td>2/3</td>
<td>55.90</td>
<td>71900</td>
<td>59.53</td>
<td>738464</td>
<td>55.91</td>
<td>721070</td>
</tr>
</tbody>
</table>

Table 3.4: NOPKWA* results on the fifteen puzzle for p=18.
tion (WA*), using perimeter (NOPS) and using KBFS. Combining all 3 algorithms together (KWNOPA*) yields a very powerful algorithm which takes advantage of all these improvements. The forward search in these algorithms is the classic Best-First search which keeps all frontier nodes in an OPEN LIST and all internal nodes in a CLOSED LIST.

We have seen that the improvement of KWA* alone over WA* is a factor of 2-3 on the fifteen puzzle. The improvement of NOPA* alone is a factor of 4-5 in the fifteen puzzle as shown below. We would expect that combining them together will yield an improvement of the multiplication of both these factors since they are not related to each other.

Table 3.4 and Figures 3.7 and 3.8 illustrate the behavior of NOPKWA* on the 100 instances of the fifteen puzzle. Indeed we see that NOPKWA* outperforms WA* by a factor of 10. For example, for a path of 68.47 P0K1_WA* (i.e where the perimeter was 0 and the K used for KBFS was 1) Which is simply WA* needs 45763 generated nodes while for the same path length of 68.87 P18K20_WA* needs only 4635. For a path of 78.41 WA* needs to generate 22840 nodes while for a path of 77.21 P18K20_WA* needs only 2880 nodes. These results show that indeed we have the multiplication of the improvements of both methods constituting an improvement factor of 10 over pure WA*.
Figure 3.6: NOPWRBFS results on the fifteen puzzle.

### 3.7 Intuition for NOPA*

Following is a suggested explanation explanation of why WNOPA* is better than both WA* and pure NOPA*.

The h part of the heuristic guides the search to proceed towards the goal while the g part prevents it from going too far and when the g is large the search tries other parts of the tree to seek shorter solutions. This means that the g part undertakes most of the overhead of the algorithm when it prunes long paths (while increasing the weight of g the number of generated nodes increase). With a perimeter, the search stops with h much larger than 0 meaning that the g part had less of a chance of causing this overhead. On the other hand less overhead of g resolves with longer solutions. To handle this we should increase the weight of g. In this case it will produce a shorter solution will cause the overhead only at the beginning of the search before reaching the perimeter. In this manner the g role of pruning long path is done more effective on a small number of nodes at the beginning of the search since there are less nodes at shallower levels of the tree than at deeper levels.

### 3.8 The twenty-four puzzle.

We conducted the same experiments on the twenty-four puzzle and the results showed the same tendency the Near-Optimal Perimeter Search generates a smaller number of nodes
<table>
<thead>
<tr>
<th>W/s</th>
<th>KBFS1</th>
<th>KBFS10</th>
<th>KBFS50</th>
<th>KBFS100</th>
<th>KBFS200</th>
<th>KBFS300</th>
</tr>
</thead>
<tbody>
<tr>
<td>g/h</td>
<td>Path</td>
<td>Nodes</td>
<td>Path</td>
<td>Nodes</td>
<td>Path</td>
<td>Nodes</td>
</tr>
<tr>
<td>-----</td>
<td>-------</td>
<td>--------</td>
<td>-------</td>
<td>--------</td>
<td>-------</td>
<td>--------</td>
</tr>
<tr>
<td>1/4</td>
<td>229.16</td>
<td>55200</td>
<td>264.84</td>
<td>55461</td>
<td>223.80</td>
<td>57133</td>
</tr>
<tr>
<td>1/19</td>
<td>286.86</td>
<td>79607</td>
<td>242.56</td>
<td>103464</td>
<td>198.86</td>
<td>93488</td>
</tr>
<tr>
<td>7/93</td>
<td>290.12</td>
<td>105409</td>
<td>231.88</td>
<td>133840</td>
<td>190.32</td>
<td>104928</td>
</tr>
<tr>
<td>1/5</td>
<td>248.86</td>
<td>146251</td>
<td>215.96</td>
<td>168889</td>
<td>181.06</td>
<td>122100</td>
</tr>
<tr>
<td>1/8</td>
<td>243.18</td>
<td>154343</td>
<td>204.48</td>
<td>235600</td>
<td>173.82</td>
<td>166265</td>
</tr>
<tr>
<td>1/4</td>
<td>236.48</td>
<td>227516</td>
<td>195.74</td>
<td>368822</td>
<td>169.54</td>
<td>171483</td>
</tr>
<tr>
<td>1/5</td>
<td>216.00</td>
<td>267450</td>
<td>184.44</td>
<td>498528</td>
<td>163.70</td>
<td>230626</td>
</tr>
<tr>
<td>9/41</td>
<td>200.26</td>
<td>430404</td>
<td>159.02</td>
<td>457011</td>
<td>150.52</td>
<td>249769</td>
</tr>
<tr>
<td>1/4</td>
<td>191.98</td>
<td>468727</td>
<td>172.64</td>
<td>527791</td>
<td>155.69</td>
<td>389544</td>
</tr>
<tr>
<td>1/20</td>
<td>179.46</td>
<td>652085</td>
<td>164.94</td>
<td>754593</td>
<td>150.20</td>
<td>520438</td>
</tr>
</tbody>
</table>

**PERIMETER 0**

<table>
<thead>
<tr>
<th>W/s</th>
<th>KBFS1</th>
<th>KBFS10</th>
<th>KBFS50</th>
<th>KBFS100</th>
<th>KBFS200</th>
<th>KBFS300</th>
</tr>
</thead>
<tbody>
<tr>
<td>g/h</td>
<td>Path</td>
<td>Nodes</td>
<td>Path</td>
<td>Nodes</td>
<td>Path</td>
<td>Nodes</td>
</tr>
<tr>
<td>-----</td>
<td>-------</td>
<td>--------</td>
<td>-------</td>
<td>--------</td>
<td>-------</td>
<td>--------</td>
</tr>
<tr>
<td>1/4</td>
<td>321.94</td>
<td>51656</td>
<td>272.86</td>
<td>51200</td>
<td>215.04</td>
<td>52532</td>
</tr>
<tr>
<td>1/19</td>
<td>286.62</td>
<td>58731</td>
<td>243.34</td>
<td>67979</td>
<td>198.14</td>
<td>62581</td>
</tr>
<tr>
<td>7/93</td>
<td>309.18</td>
<td>51854</td>
<td>232.16</td>
<td>64295</td>
<td>189.84</td>
<td>60614</td>
</tr>
<tr>
<td>1/5</td>
<td>251.08</td>
<td>62974</td>
<td>218.14</td>
<td>103388</td>
<td>180.02</td>
<td>75012</td>
</tr>
<tr>
<td>1/8</td>
<td>244.44</td>
<td>89066</td>
<td>210.28</td>
<td>134520</td>
<td>178.16</td>
<td>88006</td>
</tr>
<tr>
<td>1/4</td>
<td>236.68</td>
<td>90629</td>
<td>205.76</td>
<td>136267</td>
<td>175</td>
<td>106162</td>
</tr>
<tr>
<td>1/5</td>
<td>229.08</td>
<td>123470</td>
<td>198.36</td>
<td>207838</td>
<td>169.24</td>
<td>127148</td>
</tr>
<tr>
<td>1/5</td>
<td>215.42</td>
<td>202545</td>
<td>186.64</td>
<td>263855</td>
<td>165.22</td>
<td>147000</td>
</tr>
<tr>
<td>9/41</td>
<td>203.04</td>
<td>215061</td>
<td>182.90</td>
<td>232224</td>
<td>161.38</td>
<td>125032</td>
</tr>
<tr>
<td>1/4</td>
<td>194.90</td>
<td>254887</td>
<td>175.65</td>
<td>241219</td>
<td>158</td>
<td>192084</td>
</tr>
<tr>
<td>1/20</td>
<td>183.36</td>
<td>361072</td>
<td>167.86</td>
<td>374063</td>
<td>152.82</td>
<td>278463</td>
</tr>
</tbody>
</table>

**PERIMETER 12**

<table>
<thead>
<tr>
<th>W/s</th>
<th>KBFS1</th>
<th>KBFS10</th>
<th>KBFS50</th>
<th>KBFS100</th>
<th>KBFS200</th>
<th>KBFS300</th>
</tr>
</thead>
<tbody>
<tr>
<td>g/h</td>
<td>Path</td>
<td>Nodes</td>
<td>Path</td>
<td>Nodes</td>
<td>Path</td>
<td>Nodes</td>
</tr>
<tr>
<td>-----</td>
<td>-------</td>
<td>--------</td>
<td>-------</td>
<td>--------</td>
<td>-------</td>
<td>--------</td>
</tr>
<tr>
<td>1/4</td>
<td>311.54</td>
<td>48127</td>
<td>258.35</td>
<td>47741</td>
<td>204.96</td>
<td>48553</td>
</tr>
<tr>
<td>1/19</td>
<td>283.52</td>
<td>45210</td>
<td>239.66</td>
<td>52096</td>
<td>193.84</td>
<td>53834</td>
</tr>
<tr>
<td>7/93</td>
<td>267.92</td>
<td>40148</td>
<td>231.08</td>
<td>53282</td>
<td>186.54</td>
<td>51450</td>
</tr>
<tr>
<td>1/5</td>
<td>250.16</td>
<td>54210</td>
<td>217.64</td>
<td>57513</td>
<td>178.72</td>
<td>58661</td>
</tr>
<tr>
<td>1/8</td>
<td>243.98</td>
<td>60159</td>
<td>208.96</td>
<td>60064</td>
<td>177</td>
<td>66074</td>
</tr>
<tr>
<td>1/4</td>
<td>236.86</td>
<td>61671</td>
<td>203.38</td>
<td>96283</td>
<td>173.88</td>
<td>65455</td>
</tr>
<tr>
<td>1/5</td>
<td>226.90</td>
<td>71011</td>
<td>197.86</td>
<td>127947</td>
<td>169.04</td>
<td>73008</td>
</tr>
<tr>
<td>1/5</td>
<td>214</td>
<td>125655</td>
<td>186.82</td>
<td>164780</td>
<td>164.68</td>
<td>109315</td>
</tr>
<tr>
<td>9/41</td>
<td>203.12</td>
<td>138325</td>
<td>182.35</td>
<td>143345</td>
<td>161.10</td>
<td>91073</td>
</tr>
<tr>
<td>1/4</td>
<td>195.22</td>
<td>187280</td>
<td>175.33</td>
<td>174275</td>
<td>158.08</td>
<td>115015</td>
</tr>
<tr>
<td>1/3</td>
<td>184.92</td>
<td>264793</td>
<td>168.92</td>
<td>232700</td>
<td>152.40</td>
<td>198062</td>
</tr>
<tr>
<td>1/3</td>
<td>172.16</td>
<td>406042</td>
<td>159.62</td>
<td>304126</td>
<td>147.34</td>
<td>484153</td>
</tr>
</tbody>
</table>

Table 3.5: NOPKWA* results on the twenty-four puzzle.
but the solution length is only near-optimal. However, the largest perimeter that we could store in our computer was to a depth of 16. Indeed, the improvement factor for the twenty-four puzzle for a perimeter of 16 was not as large as a perimeter of 21 for the fifteen puzzle. For example, for NOPA* the improvement factor for the twenty-four puzzle was not greater than 2.2 while for the fifteen puzzle it was 5. Even for a perimeter of 16 for the fifteen puzzle the improvement was a little better than that of the twenty-four puzzle. This is due to the fact that a perimeter of depth 16 is about 33% of the average optimal solution while for the twenty-four puzzle a perimeter of a depth of 16 is 15% of the optimal solution. This suggests that for larger problems we need larger perimeters to provide the same improvements. This is in contrast to KBFS which showed that for larger problems KBFS improvements are larger.

Figure 3.9 illustrates weighted RBFS for different perimeters. Table 3.5 shows the results of WKNOPA* for the twenty-four puzzle for a selection of k values and for different perimeters.

3.9 NOPS as an anytime algorithm

As we have seen both theoretically and empirically, NOPS reduces the solution length but increases the number of generated nodes. We can therefore create a dynamic algorithm that incrementally improves the quality of the returned solution (An Anytime Algorithm). In this version of NOPS all the perimeters up to the maximum depth D are saved in a hash
table with an internal field indicating the depth of each such node from the goal.

Now NOPA* is executed until it reaches a node of the perimeter at depth D. The solution is returned and now NOPA* continues to run until it finds a node at depth D-1. As we have seen this solution is shorter and is an improvement over the solution that was found for level D. Then, NOPS continues until a node at level D-2 is found. This process continues on and on until level 0 is found and the goal is reached. At this stage NOPA* is exactly the same as A* itself and the optimal solution was found. The overall time complexity is exactly the same as that of A* while near optimal solutions are provided much earlier. With this method NOPA* can be executed and run as much time as we want and it will provide the best solution that it has found until that period. If we have more time a better solution can be found. This method actually runs A* with possibilities of stopping the search earlier at the cost of a longer solution length. However, no overhead in time complexity is involved. And indeed in our experiments we used this version and once other variables such as K and W were determined we could obtain the results for all the different values of the depth of the perimeter in one execution of NOPS.

Note that no other inadmissible algorithm such as KBFS and WA* has the capability of continuing to run and improve its solution after it was found. After K for KBFS or W

\footnote{In fact we need to store them anyway in order to return the rest of the solution from the node that was reached in the perimeter to the Goal node.}

\footnote{Or any other kind of search like NOPRBFS or NOPIDA* etc}
for WA* are determined and the solution is found, if we want a better solution then a new value for K or W should be assigned and the whole process needs to be started again. In NOPS however, after the perimeter at depth D was reached we can continue on to find the perimeter at shallower levels.

### 3.10 Using the perimeter for improving the heuristics

When running A* or IDA* the heuristic function is used for ordering the frontier nodes. However, when keeping nodes that surround the goal node in memory, when a node form the perimeter is reached since we know the exact distance from it to the goal we can use this accurate distance instead of the heuristic function which is underestimating. In a sense, for these nodes there is a lookup table for the h part of \( f = g + h \). When the number of nodes that are kept in memory are larger more nodes fall into that category and in the overall process we have a better heuristic which implies that the solution will be found earlier with a smaller number of generated nodes. Also, whenever a node is reached which is outside the perimeter and its heuristic function is smaller than the depth of the perimeter we can use the depth of the perimeter plus one for its heuristic. This is because we know that we are far from the perimeter by at least 1.
We have tried this version of A* for the 15 tile puzzle with storing all nodes up to depth of 21 from the goal and using the exact distance to the goal for their heuristic value. Unfortunately, the improvement in the number of generated nodes was insignificant and was not larger than 25%. This is because most of the search was performed far away from the perimeter and only a small fraction of the nodes that were generated were close enough to the goal node to have heuristics that were smaller than the perimeter.

3.11 Conclusions

Most of the work in single agent heuristic search was done for admissible algorithms. Therefore, most algorithms were developed in order to find an optimal solution of minimum cost. This approach is theoretically oriented since in practice many problems are so large that they can not be solved optimally with current techniques and technology. For example many instances of the twenty-four puzzle that were solved optimally in [33] needed a few months of computation. Therefore a near optimal solution is the only considerable solution for large problems.

In the above chapters we have shown some simple and yet very powerful modification to known search techniques, namely KBFS and NOPS. These modifications help the new algorithm to find near-optimal solution much more efficiently. Since they are orthogonal, they can be combined together. The improvement factor was up to 100, two orders of magnitude.
Chapter 4

Background: The additive pattern database theory

4.1 Better heuristics

As the search space becomes larger and as problems scale up, a more accurate heuristic function is needed. With a better heuristic function, the search will be guided faster towards the goal state and a smaller number of nodes will be generated. However, in order to obtain a more accurate heuristic function, a larger overhead will be caused by each node, resulting in a larger constant time per node. However, in general, the smaller number of generated nodes tends to compensate for the larger overhead, resulting in a significant improvement in the running time.

In the following chapters, we will present a general theory for developing better heuristic functions. This theory suggests taking a simple heuristic that looks at isolated subgoals and to generalize it. Instead of simply looking at each subgoal alone, we now gather data from the interactions of these subgoals by looking on groups of subgoals resulting in a more informed heuristic. These interactions can be calculated in two ways. The first way is to precalculate a large database with such interactions and this database is consulted during the search. We call this method the Additive database method (ADB). This theory was first presented by Korf in [33]. In the following chapters, we take this theory much further, study it, and generalize it. We implemented it and will show empirical results on both the tile puzzles and on the vertex-cover problem. The second way is not to store any database but to calculate these interactions on the fly. We present such an implementation on the graph partitioning problem.

The additive database theory is general and can be applied to many domains. However, we choose to describe the ADB theory on the environment of the tile puzzle since it was this
environment where this theory was first presented.

The search space of the twenty-four puzzle contains nearly \(10^{25}\) states which is a Trillion times larger than the fifteen puzzle. In spite of considerable work on the fifteen puzzle nobody has solved a version of the puzzle that is larger than the 15 puzzle until the work of Korf and Taylor in [33] which was the first attempt to optimally solve the twenty-four puzzle. The main contribution of their work was using a heuristic that is much more accurate than the Manhattan distance. Three enhancements to the Manhattan distance were implemented in that work, namely The Linear conflict, the corner tile and the last move enhancement. An automatic mechanism, which we call the additive pattern database for deriving these heuristics, was first presented in [33]. These enhancements as well as the automatic mechanism will be presented in the following chapter.

The rest of the present chapter is completely copied from [33] with only minor changes.

### 4.2 Heuristics from multiple subgoals

Let’s take a closer look at the well know Manhattan distance heuristic. A natural question that arises, is how this heuristic could be automatically learned by a computer program. We give two answers to this question, the standard explanation, and a new explanation that suggests the additive pattern database extension.

#### 4.2.1 The standard explanation

The standard explanation for the origin of admissible heuristic functions is that they are the cost of exact solutions to a simplified version of the original problem. For example, to move a tile in the tile puzzle from position \(x\) to position \(y\), \(x\) and \(y\) must be adjacent to each other, and position \(y\) must be empty. If we ignore the empty constraint, we get a new problem where any tile can be moved to any adjacent position and multiple tiles can occupy the same position. In this simplified problem, the tiles are independent of each other. Thus, we can solve any instance optimally by taking each tile at a time, and moving it along the shortest path to its goal position, counting the number of moves made. The cost of an optimal solution to this simplifies problem is just the Manhattan distance from the initial state to the goal state. Since we simplified the problem by removing a constraint on the moves, any solution to the original problem is also a solution to the simplified problem. Thus the cost of an optimal solution to the simplified problem is a lower bound on the optimal solution to the original problem. Any heuristic derived this way is therefore admissible.
4.2.2 A new explanation

An alternative derivation of the Manhattan distance function comes from the observation that the sliding-tile puzzle consists of a set of subproblems, namely to get each individual tile to its goal location. This allows us to consider the cost of solving each individual subproblem, assuming no interaction between the subproblems. In other words, we count the minimum number of moves of each tile required to get it to its goal location, which is just the Manhattan distance of that tile. Since each move only moves one tile, we can add all these individual distances together to get an admissible heuristic for the original problem.

The key idea here, which makes it possible to efficiently compute the Manhattan distance, is the assumption that the individual tiles do not interact with one another. The reason the problem is difficult, and why the Manhattan distance is only a lower bound on actual cost, is that the tiles get in each others way. By taking into account some of these interactions, we can compute more accurate admissible heuristic functions.

4.2.3 Non-Additive Pattern Databases

Pattern databases\cite{10} represent one way to do this, and were originally developed for the Fifteen Puzzle. Figure 4.1 shows the Fifteen Puzzle, with a particular subset of the tiles labeled, called the fringe pattern by Culberson and Schaeffer. The minimum number of total moves required to get just the fringe tiles from their initial positions to their goal positions, including any required moves of other tiles as well, is a lower bound on the minimum number of moves needed to solve the entire Fifteen Puzzle.

![Fringe Pattern for the Fifteen Puzzle](image)

It would be too expensive to calculate the moves needed to solve the fringe tiles for each heuristic calculation in the search. This number, however, depends only on the positions of the fringe tiles, and the blank position, but is independent of the positions of the other tiles. Since there are only a limited number of such configurations, we can precompute all of these values, store them in memory in a pattern database, and look them up when
they are needed during the search. Since there are seven fringe tiles and one blank, and sixteen different locations, the total number of different possible configurations of these tiles is $16!/(16 - 8)! = 518,918,400$. For each table entry, we store the number of moves needed to solve the fringe tiles from their corresponding locations, which takes less than a byte of storage. Thus, we can store the whole table in less than 500 megabytes of memory, which is feasible on modern workstations.

We can compute this entire table by a single breadth-first search backward from the goal state shown in figure 4.1. In this search, the unlabeled tiles in the figure are all considered equivalent, and a state is uniquely determined by the positions of the labeled tiles and the blank. As each configuration of these tiles is encountered for the first time, the number of moves made to reach it is stored in the corresponding entry of the pattern database. The search continues until all entries of the database are filled. Note that this table need only be computed once for a given goal state, and its cost can be amortized over the solution of multiple problem instances with the same goal state.

Once the table is stored, we can use IDA* to search for an optimal solution to a particular problem instance. As each state is generated, the positions of the fringe tiles and the blank are used to compute an index into the pattern database, and the corresponding entry, which is the number of moves needed to solve the fringe tiles, is used as the heuristic value for that state.

Using this pattern database, Culberson and Schaeffer were able to reduce the number of nodes generated to solve the Fifteen Puzzle by a factor of 346, and reduce the actual running time by a factor of 6[^10]. Combining this with another pattern database, and taking the maximum of the two database values as the heuristic value reduced the nodes generated by about a factor of thousand, and the running time by a factor of 12, compared to Manhattan distance.

**Limitations of Non-Additive Pattern Databases**

The main limitation of non-additive pattern databases is that they don’t scale up to larger problems. For example, the state space of the Twenty-Four Puzzle is more than 100,000 times larger than that of Rubik’s Cube. Since the Twenty-Four puzzle contains 25 different positions for the tiles and the blank, a pattern database covering $n$ tiles plus the blank would occupy $25!/(25 - n + 1)!$ locations of memory. For example, just five tiles plus the blank generate almost 128 million possibilities, requiring over 121 megabytes of storage. A pattern database of six tiles and the blank would require over 2.3 gigabytes. Even with this large a database, since only six tiles are covered by it, the values would almost certainly be smaller than the simple Manhattan distance of all the tiles. Even with multiple five-tile databases,
the best way to combine them admissibly is to take the maximum of their values, even if the tiles in the different databases don't overlap. The reason is that the non-additive pattern database values include all moves required to solve the tiles in the pattern, not just moves of the pattern tiles.

Instead of taking the maximum value of different pattern databases, we can sum the Manhattan distances of each tile to derive an admissible heuristic. This is because each move of the puzzle moves only one tile, and the Manhattan distance of a tile counts only the moves of that tile. In general, what is needed is a mechanism that takes into account the interactions of different tiles, but for which individual values can be added together in order to get a more accurate admissible heuristic. This is the basic idea of additive pattern databases.

4.2.4 Generalization of the Manhattan Distance

We can automatically calculate the Manhattan distance as follows. For each possible location of an individual tile, we perform a search of the original problem space to correctly position that tile, ignoring all other tiles, and only counting moves of the tile in question. Since the operators of the tile puzzle are invertible, we can perform a single search for each tile, starting from its goal position, and record how many moves of the tile are required to move it to its goal position. Doing this for all tiles resulted in a table which gives, for each tile, its Manhattan distance from its goal position. Then, since each move only moves one tile, for a given state we add up the Manhattan distance of each tile to get an admissible heuristic for the state. Of course, we don’t really need to do the search in this case, since we can easily determine the Manhattan distance values from the problem, but this requires some human insight, whereas the search can be done automatically.

The value of this reconstruction of Manhattan distance is that it suggests a further generalization. Rather than considering each tile in isolation, we could compute the cost of solving each possible pair of tiles. In other words, for each pair of tiles, and each combination of positions they could occupy, perform a search to their goal positions, and count only moves of the two tiles of interest. We'll call this value the pairwise distance of the two tiles from their goal locations. A state of this search consists of the positions of the two tiles and the position of the blank, since all other tiles are equivalent. Again for efficiency, for each pair of tiles we perform a single search starting from the goal state, and store the pairwise distances to all other positions in a table.

For almost all pairs of tiles and positions, their pairwise distance will equal the sum of their Manhattan distances. However, there are three types of cases where the pairwise distance exceeds the combined Manhattan distances. The first case is when the two tiles
are in a *linear conflict*. It applies when two tiles are in their goal row or column, but are reversed relative to their goal positions. For example, assume that the top row of a given state contains the tiles (2 1) in that order, but in the goal state they appear in the order (1 2). To reverse them, one of the tiles must move down out of the top row, to allow the other tile to pass by, and then move back up into the top row. Since these two moves are not counted in the Manhattan distance of either tile, two moves can be added to the sum of the Manhattan distances of these two tiles without violating admissibility.

The second case involves tiles near the corners of the puzzle. Referring to the goal state of the Twenty-four Puzzle in figure 4.2, consider a state where the 3 tile is in its goal position, but some tile $X$ other than the 4 tile is in the adjacent corner. In that case, the 3 tile will have to move temporarily out of position, and then back into position, in order to correctly position the 4 tile. Thus, the pairwise distance of the 3 and 4 tiles will exceed the sum of their Manhattan distances, since the Manhattan distance of the 3 tile is zero. Similarly, the pairwise distance of the 3 tile and the tile $X$ that is in the corner will exceed the sum of their Manhattan distances. This also applies to the 9 and 4 tiles if the 9 is correctly positioned, but the 4 is not, and also to the 9 tile and whichever tile is in the adjacent corner in that case. This same situation occurs around the lower left and lower right corners of the puzzle.

The third case involves the 1 and 5 tiles in the Twenty-four Puzzle. Since the blank is in the upper left-hand corner in the goal state, the last move of any solution must either move the 1 tile to the right, or the 5 tile down. Thus, in the next-to-last state of a solution, either the 1 or 5 tile must be in the upper left-hand corner. Since the Manhattan distance of these tiles is computed to their goal positions, unless the 1 tile is in the left-most column, its Manhattan distance will not accommodate a path through the upper-left corner. Similarly, unless the 5 tile is in the top row, its Manhattan distance will not accommodate a path through the upper-left corner. Thus, if the 1 tile is not in the left-most column, and the 5 tile is not in the top row, their pairwise distance will be two greater than the sum of their Manhattan distances.
Computing all the pairwise distances by simple search “discovers” Manhattan distance, along with all three of the heuristic enhancements described above, with very little domain-specific reasoning. No other enhancements are discovered by the pairwise searches.

Since the states of these searches are only distinguishable by the positions of two tiles and the blank, the size of these search spaces is at most $O(n^3)$, where $n$ is the number of tiles. There are $O(n^2)$ such searches to perform, one for each pair of tiles, for a time complexity of $O(n^5)$. The size of the resulting tables is at most $O(n^4)$, one entry for each pair of tiles in each combination of positions. Note that these pairwise searches only have to be performed once for a given goal state, and their cost can be amortized over all subsequent problem-solving trials with the same goal state.

4.2.5 Triples of tiles

Clearly, this idea can be extended to heuristics based on solving three or more subgoals simultaneously. As in the case of pairwise distances, the triple distance for each triple of tiles can be computed by a single breadth-first search backward from the goal state, and stored in a table. In this case, there are $O(n^4)$ states of each search, and $O(n^3)$ different searches, for a total time of $O(n^7)$, where $n$ is the number of tiles. The total size of the triples database is $O(n^6)$ in the worst case.

4.3 Computing an Admissible Heuristic

The next question is how to handle the interactions between the pairwise, triple, or higher-order distances to compute an admissible heuristic function for a given state. We can’t simply sum all of the pairwise distances, for example, because moves of the same tile will appear repeatedly in pairings with different tiles.

Assume that we have precomputed all the pairwise distances and stored them in a table. The pairwise distance for tiles $x$ and $y$ represents the minimum number of total moves of both of these tiles to get them to their goal positions. For a given state, we can represent the relevant pairwise distances by a pairwise-distance graph, with a node for each tile, and an edge between each pair of tiles, weighted by the pairwise distance of those tiles. In practice, we only include edges for those pairs of tiles whose pairwise distance exceeds the sum of their Manhattan distances, and then add the total Manhattan distance to the moves from this graph to get the corresponding heuristic value, but the theory is the same in either case. We want the largest admissible heuristic value that is consistent with this graph.
4.3.1 Maximal Weighted Matching

One way to get an admissible heuristic is to choose a set of edges such that no two edges in the set are incident to the same node. Thus, moves of the same tile cannot be counted more than once. This is called a matching of the graph. To get the largest admissible heuristic, we want a matching for which the sum of the weights of the included edges is maximized. If all the weights are the same, this is known as the maximum matching problem, and for different edge weights, it is known as the maximum weighted matching problem. It can be solved in $O(n^3)$ time, where $n$ is the number of nodes in the graph, or tiles in the puzzle in our case.

If we want to generalize these heuristics we need to extend the idea of the pairwise distance to include triples of tiles, quadruples, etc. The corresponding matching problem is a hypergraph matching, when a single hyperedge connects three or more nodes, and unfortunately is NP-complete [15]. Thus, we may have to rely on a greedy approach to the higher-dimensional matching problem and a lower but still admissible heuristic value.

This is a fairly general theory for the discovery and implementation of admissible heuristic functions. Many admissible heuristics are constructed by considering the solution of each individual subproblem in isolation, and ignoring the interaction with other subproblems. These scheme proposed heuristics based on the simultaneous consideration of two, three or more subgoals.
Chapter 5

Implementing the ADB on the tile puzzle

Let’s look again at the three types of enhancements to the Manhattan distance based on pairs of tiles. All these enhancements are also discovered when one looks on the Pairwise distance. In [33], Korf and Taylor implemented these enhancements to the Manhattan distance and were the first ones to optimally solve the twenty four puzzle.

While the ADB theory was presented in that paper, the weakness of their implementation in [33] is that these enhancements were hardwired to the program and are domain dependent. They were sceptical whether an ADB system will be cost effective and they did not build such a database but used only hardwired enhancements of pairs. Nevertheless the results were very significant since they contained the first optimal solutions to random twenty-four puzzle instances.

In the following section we will describe the GADB (General Additive Database system) that we have built for the fifteen and the twenty four puzzle. In the section after that we will describe the DADB (Disjoint additive database) which is another way of using the additive database for calculating heuristics.

Then, we will present empirical results on both versions of the puzzle.

5.1 The GADB system

In the following section we will describe the structure and the behavior of the General Additive Database Search for the Tile Puzzle (GADB).

While Korf and Taylor in [33] were somewhat sceptical if this method could be implemented, after a closer observation we found that indeed it is possible. We have built a working system that is capable of finding optimal solutions to the twenty-four puzzle when
using a heuristic function that is based on a database that stores the conflicts between pairs and triples.

5.1.1 The size of the data base

The first observation that we have made is that the upper bounds in [33] are much larger than in practice. For the twenty-four puzzle, there are $\binom{24}{2} = 276$ different pairs. Each pair can be located in $25 \times 24 = 600$ different pairs of locations. Therefore, there are $276 \times 600 = 165,600$ different combinations for pairs and thus the size of table that contains all this data should be the same. However, we only need to store data for pairs and locations such that the two tiles involved are in a conflict and the corresponding pairwise distance is more than the sum of the individual Manhattan distances. For other cases, when there is no conflict, the pairwise distance will be exactly the sum of the Manhattan distances and does not need to be included in the table. Actually, the table can store only the additional moves that each conflict adds to the Manhattan distance and therefore, when there is no conflict the corresponding edge is 0 and can be discarded.

For example, tile 8 and tile 16 in the twenty-four puzzle will always have their pairwise distance equal to the sum of their individual Manhattan distance and thus should not be included in the table. Therefore, the size of the table is much smaller than $O(n^4)$. Each tile in the twenty-four puzzle can have a linear conflict with 8 other tiles, 4 in its row and 4 others in its column. Therefore, the number of different pairs in the table is $(24 \times 8) / 2 = 96^1$. We don’t need to add the 6 pairs of tiles that are situated next to a corner (for the corner tile enhancement) since they are already counted. Adding the 1 and 5 pair for the last move gets to a total of only 97 pairs for the twenty-four puzzle and only 46 for the fifteen puzzle. Moreover, even for pairs of tiles that might have a conflict between them, this conflict will only happen for a small number of locations. For example, tiles 2 and 3 will be in a linear conflict only if they are located in the top row but in reversed order. There are only 10 combinations for such a conflict. This is very small compared to the 600 different combinations to locate these two tiles. After building such a database we found that the number of different conflicts for pairs of the twenty-four puzzle is only around 3000. This is a great reduction compared it to the upper bound of 165,600.

Calculating the corresponding upper bounds for triples show that there are 2024 different triples which can be located in 13,800 combinations yielding a table with 27,931,200 entries. In practice we found that there are no more than 148,000 entries in the table.

\footnote{For simplicity, we do not count the fact that tiles at the top row and at the left column can have a conflict with only 7 tiles.}
5.1.2 Building the database

In order to build the database we did the following. For each pair or triple of tiles, we performed a breadth-first search backwards from the goal location of the corresponding tiles. For every new combination of locations for these tiles we checked whether the path in the breadth-first search tree is larger than the sum of the individual Manhattan distances. If the path was indeed larger than the Manhattan distance, then this specific pair or triple in the specific combination of locations, should be stored in the database as a case that the pairwise or triplewise distance exceeds the sum of individual Manhattan distances. It takes about 10 minutes to build this database for the twenty four puzzle but once the database is stored in a file on the disk it takes a few seconds to load it to a table in the main memory.

Actually, we have two files, one for pairs and one for triples. These files contain lists of pairs or triples, the locations where they have a conflict and the addition of this conflict over simple Manhattan distance. At the initialization phase of the program the lists from these files are loaded into the database in the main memory.

5.1.3 Constructing the graph

We now turn to the method that the database is used in order to calculate the heuristic function for a particular state. According to the additive database theory, we need to build a conflict graph where the nodes are the tiles and the edges or hyper edges are pairs or triples of tiles where the mutual distance to their goal location exceeds the sum of their individual Manhattan distance. In order to find all the conflicts for each state of the twenty-four puzzle, one should look at all the different pairs and triples of that state and then to look in the conflicts table to see whether these tiles in the current combination of locations have an entry in the table. With this knowledge we can construct the corresponding conflict graph. There are 2024 triples and 276 pairs for each state. Only a few have an entry in the table but we do not know that in advance and should look in the table for all of them. Fortunately, this massive checking of the table should be done only once at the initial state. Then, when using a depth-first oriented search like IDA*, since only one tile moves from a parent to its child in the search tree, we should only check pairs and triples that include this specific tile. A particular tile can be involved in 23 pairs and in 253 triples. (We will later reduce this number even more). Therefore, keeping track of the dynamic graph that represents the different conflicts between tiles is not so expansive as it first seems.
5.1.4 Vertex-Cover as an Admissible Heuristic

After the graph is constructed for the state, a heuristic evaluation should be calculated, based on that graph.

If the pairwise distance of tiles $X$ and $Y$ in a given state is $a$, there will be an edge in the corresponding pairwise-distance graph between nodes $X$ and $Y$, weighted by $a$. If $x$ is the number of moves of tile $X$ in a solution, and $y$ is the number of moves of tile $Y$, then their pairwise distance represents the constraint that $x + y \geq a$. Each edge of the pairwise-distance graph represents a similar constraint on any solution. In addition, the Manhattan distance of each individual tile represents a unary constraint, which is a lower bound on the number of moves of that tile. Here we consider the general case where the edge weights include their Manhattan distances as well.

The problem is to assign a number of moves to each node of the graph such that all the unary and pairwise distance constraints are satisfied. Since the constraints are all lower bounds, assigning large values to each node will satisfy all the constraints. In order for the resulting heuristic to be admissible, however, the sum of the values of all nodes must be the minimum sum that satisfies all the constraints. This sum is then the maximum admissible heuristic for the given state. Furthermore, each node must be assigned a non-negative integer.

![The pairwise graph](image)

**Figure 5.1:** Triple linear conflict in the 24 tile puzzle.

Let's look at the example in Figure 5.1. We have three tiles in a linear conflict, namely 3, 2 and 1. All three pairs have an edge with a weight of 2 in the graph since for each pair one of the tiles should move down. For this particular case we have the pairs (1, 2), (2, 3) and (1, 3). Maximal matching will take one of these edges, add 2 to the overall heuristic and then delete the corresponding nodes from the graph. While matching is admissible, it is obvious that we can add more to the heuristic without losing admissibility.

In the general case, an edge $(X,Y) = 2$, means that nodes should have values such that $X + Y \geq 2$. With matching we added this value of 2 and deleted these nodes from the graph. For the special case of the tile puzzle, since the odd-even parity is the same for all solutions, then for each pair, one of the tiles should move at least two moves more than

---

2For the meanwhile, we will limit our discussion to pairs only.
its Manhattan distance in order to solve the pairwise conflict. Therefore, this edge actually means not only that \((X + Y) \geq 2\) but also \((X \geq 2 \mid Y \geq 2)\).

Now we have a list of equations in the form: \((X \geq 2 \mid Y \geq 2)\). In order to satisfy all these equations, one of the variables from each equation must be set to at least 2. An admissible heuristic will be a minimal assignment to variables such that all equations are satisfied. If these equations are edges, the the minimal assignment is a minimal vertex cover.

The value of a node is the number of moves that it should move above the Manhattan distance according to this assignment.

In the above example, a minimal vertex cover of these 3 edges will needs two tiles to cover all the edges and solve this linear conflict of 3 2 1 and the corresponding heuristic will have four additional moves. Maximal matching takes only one edge and the corresponding heuristic will have two additional moves. In a general example, when we have a clique of \(n\) nodes, matching will add \(n/2\) where vertex-cover will add \(n-1\). This is a great improvement.

Of course, if the VC assigns: \(X=2\), in the final solution it might not move, and actually be: \(X=0\). However, this VC assignment is a lower bound on the total number of overall moves. This is because any other solution must also be an assignment of values such that all equations will be satisfied, but VC is minimal among all such assignments.

Another nice property of the minimal vertex cover, is that it gives the full benefit of all linear conflicts with only pairwise costs. With maximal matching in order to capture multiple linear conflicts in the same row or column in the \(N\times N\) puzzle we needed \(N\)-tuple enhancements.

### 5.1.5 Moving from pairs to triples

Looking at conflicts of triples and storing them also in the database introduce more conflicts and therefore the corresponding heuristic function will be much more accurate. While there are only 3 classes of enhancements for pairs there are many more different classes of triple enhancements that are not captured by pairs alone. We do not think that it is worthwhile to classify\(^3\) them all but the following are a couple of examples.

Figure 5.2 illustrates different triples that have a mutual distance greater than their individual distance. Note, that in all cases none of the internal pairs capture the conflict and only the three tile together have the conflict.

In Figure 5.2a. the tiles 7, 8 and 13 are involved. In the left scenario, the 8 tile should either go around tiles 7 and 13 or one of them should move to let 8 pass between them. The same goes if it is 9 and not 8. In the right scenario, 8 should move to the current location

\(^3\)In fact, we do not think that we are aware of all of them since they were all discovered automatically by our program
of 13 and 13 should move down one position. However, if 8 goes first it will create a linear conflict with 13. If 13 moves first it will cause the same situation as the left case.

In Figure 5.2b If 19 moves right it has a linear conflict with 14. Otherwise, 14 or 18 should move away to let it pass through or 19 must move left to go around them.

All of these scenarios are actually variations of linear conflicts but generalized to triples.

In Figure 5.2c tiles 1, 2 and 5 are involved. This scenario is a combination of both the linear conflict of 1 and 2 and the last move enhancement of 1 and 5. This conflict is an example for an edge with a weight of 4, since 4 additional moves must be done. It could be either the 1 that will move these 4 additional moves or any two tiles should move 2 moves each.

Pairs: (10,15), (1,5), (1,7)
Triples: (3,5,14), (1,6,10)
VC= 1, 3, 10.
Total additive heuristic = 6.

Figure 5.3: First case example.
Figure 5.3 illustrates the initial state of case number 1 from [33] and its additive heuristic. Note that there are 3 pairs and two triples. The minimal vertex cover is 3 and therefore the addition to the Manhattan distance heuristic is 6.

5.1.6 Combining both pairs and triples.

Combining both pairs and triples needs a little care. First, we should count only triples that none of the internal pairs are already in conflict. Otherwise for each pair we can add any other third tile to generate a triple. Thus we stored only the triples whose conflicts are not captured by any of the internal pairs. Also, if there is a triple such that each of its internal pairs has an edge of 2 then this triple will have an edge of 4. For example, the linear conflict between 1, 2 and 3 in Figure 5.1. In this case the triple hyperedge can be deleted since all the information is also captured by the three pair edges. The vertex cover will indeed take two nodes for these triples. On the other hand we have a scenario as in 5.2.c. In this case we have the pair edges of (1,2) and of (1,5), both with a weight of 2. We also have the triple hyper edge of (1,2,5) with a weight of 4. In this case we do not delete this triple hyperedge from the database since its information is not all captured by its internal pairs. In fact tile 1 can solve the internal pairs while actually we need 4 extra moves.

5.1.7 Taking Vertex-Cover for a weighted graph

The assumptions in [33] when suggesting the maximal-matching was that every edge or hyperedge in the graph requires some number of additional moves, not necessarily all the same. Taking a maximal matching and adding the resulting moves is certainly admissible. In vertex-cover things are a little more complicated since the classic vertex cover takes one node for each edge regardless of the weight of that edge. If we use such a vertex-cover we might loose information when we have an edge of 4.

When we have an edge of 4 then \((X,Y,Z)=4\) actually means:

\[
(X >= 2 & Y >= 2) \land (Y >= 2 & Z >= 2) \land (X >= 2 & Z >= 2) \land (X >= 4) \land (Y >= 4) \land (Z >= 4)
\]

In our system we ignored the assignment of 4 to a single tile. Therefore, we have generalized vertex-cover as follows: Each edge with a weight of 2 should be covered by one tile. Each edge with a weight of 4 should be covered by two tiles. There is however, an extreme case where an edge of 4 can only be solved with one tile that moves 4 additional moves. We do not refer to this case in our system because if it exists it is very rare\(^4\). However, this case can

\(^4\)In fact we did not find a case in which the only way to solve the conflict was with one tile that moves 4 moves. Nevertheless, we cannot prove that it does not exist.
only increase the heuristic value and if we take two tiles for that edge we surely do not lose admissibility. For example, suppose we have the following edges: \((1, 2, 3) = 4\) and \((3, 4) = 2\). Suppose the only way to solve \((1, 2, 3)\) is for 1 to move 4 moves. In that case the vertex cover should have 6. We do not refer to this and instead take two tiles from the first edge and one from the second edge (2 and 3 for example) and reach a total of 4.

### 5.1.8 General schema of the program

The general schema of our program is as follows. For each new state in the search tree we keep the conflicts graph by storing a list of edges and their weights. This list is first calculated for the root of the tree. Then, since we use IDA* which is a depth-first oriented search, the graph can be changed incrementally with each move by looking at the tile that moved and changing the graph according to its new location.

Then, a vertex cover of this graph is calculated. If the graph did not change, then the old vertex cover is taken and it is not calculated again. When the graph does change then unfortunately, since vertex cover is NP-Complete, there is no method to incrementally calculate the vertex cover. Otherwise, we can find a vertex cover for one edge and then incrementally add all other edges and find a dynamic programming method for vertex cover in polynomial time. Therefore, we had to calculate the vertex cover from scratch for each new node. Fortunately, since the graph is very sparse and does not include many edges this was done rather quickly. In fact, for a large graph, when calculating the vertex cover, we first divided the graph into connected components and then we evaluated the vertex cover for each connected component alone. This deviation reduced the time complexity of the vertex cover.

After calculating the vertex cover we added it to the simple Manhattan distance of that state and used the total amount as an admissible heuristic for IDA*.

### 5.1.9 Domain dependent enhancements for the Tile puzzle

The entire analysis above\(^5\) is described on the tile puzzle but can also be relevant for other domains. However, after many observations on the tile puzzle, we found many enhancements and improvements that are based on the nature and behavior of the puzzle. We have added many of them to our code and each of them contributed either to reducing the constant time per node or to reducing the number of generated node. Following is a list of a couple of these enhancements.

\(^5\)One may skip the following section and proceed directly to the Disjoint database section without loosing anything.
Dividing the puzzle into two components.

We have found that sometimes the existence of an edge does not depend on the location of the relevant tiles but also on the location of the blank. If the current positions of the tiles in a pair or a triple break the puzzle into two connected components then sometimes there is a difference depending at which component the blank tile is located as described below. If such a scenario happens then our program recognizes this situation and when there are two different values that depend on the blank’s location then that triple (or pair) is broken into two triples and both values are stored. When constructing the graph we take the relevant value from the relevant entry in the table.

In Figure 5.4, tiles 21, 22 and 23 break the puzzle into two components. The first component is the three locations in the top right corner and the second component is the rest of the puzzle. These three tiles alone do not have any conflict between them and they can each move to their goal locations with no interference of each other and no additional move should be added. This is true if the blank tile is in the large component as in Figure 5.4a. However, if the blank is in the small component as in Figure 5.4b then one of these tiles must move, against the suggestion of the Manhattan distance, into the small component in order to let the blank move to the other component. In this case we can add two additional moves for this triple. Our program is aware of this enhancement and takes full advantage of it. In fact we found that the number of generated nodes was reduced by nearly 25% after adding this enhancement.

This enhancement is actually a special case of a larger enhancement which can be named: the “first move enhancement”. It happens when the blank tile is surrounded by a perimeter of tiles such that the Manhattan distance for each of them is away from the blank. In this case, one of these tile must move to replace the blank against its Manhattan distance and two moves might be added to the total heuristic. Figure 5.4 illustrates a case of the first move enhancement. One of the tiles 4, 5, 8 or 25 must move to replace the blank and actually
move away from its goal location.

**Conditions for the graph to be stable**

Following is an example of another enhancement that we found and implemented into our code. The Manhattan distance suggests the closest path for a tile to reach its goal location. This path includes a number of either up moves or down moves and a number of right moves or left moves. If the tile moves as suggested by the Manhattan distance then the Manhattan distance decreases by one. If it moves to other directions against the Manhattan distance then the Manhattan distance increases by one. When observing closely we can see the following proposition:

**Proposition 1:** When a tile moves against its Manhattan distance, no new edges can be added and when a tile moves according to its Manhattan distance no edge can be deleted.

**Proof:** Suppose the contrary, that a tile X moved against its Manhattan distance and a new edge was added for this tile say (X,Y). Also, since X moved against the Manhattan distance, the Manhattan distance for X was increased by one. This means that now in order to solve this conflict one of the tiles X or Y must move away from its Manhattan distance direction. However, if tile X returns to its previous location, this edge is being removed without any of the tiles moving against the Manhattan distance. Contradiction. The second part of the proposition is the duality of the first one.

These two sides of the same fact are very useful and reduce the constant time for the incremental constructing of the graph. The main contribution is the fact that if a tile moved against its Manhattan distance then we know that no new edge can be added. When constructing the graph, most of the work is spent when searching for new edges which involves looking at all the triples that contains this tile. When taking advantage of the proposition, this search is avoided in half of the nodes, thereby saving a lot of time. In fact, when a tile moves against it Manhattan distance and no old edges were deleted then the graph did not change and it is not even necessary to evaluate the vertex cover once again.

---

6Since looking if old edges are still there is to only scan the list of edges of the old graph. Since the graph is very sparse this act is very fast.
We found that implementing this enhancement into the code reduced the average constant time by nearly 40%.

5.1.10 Future enhancements

We believe that the GADB system for the tile puzzle can be further improved by the following enhancements.

1. The database for pairs and triples needs no more than one Mega byte of memory. On current machines this amount is only a very small fraction of the available memory. Therefore, we believe that we may be able to add quadruples to our system without exhausting the memory. Of course, if the memory is exhausted we can always store as many quadruples as possible. Quadruples will add many more conflicts to the system and will certainly decrease the number of generated nodes. However, this enhancement is not trivial and it will take a lot of work in adding it to our code. Also we are concerned that the constant time per node with quadruples will be much larger since keeping track of all quadruples as well as calculating the vertex cover for quadruples will take a lot of time.

2. The Manhattan distance actually forms a rectangle which tells the tile to stay inside it (It can move only in 2 directions). For each tile of each edge, we should store 4 additional bits in the table, one for each direction: UP, DOWN, LEFT and RIGHT. If the corresponding tile can solve the conflict of the edge when moving up from the Manhattan distance rectangle then the UP bit will be set. This will be done for all 4 directions and this way we can determine if and how each tile can solve the conflict of a given edge. Now, two edges will be inserted to the same connected component of the graph only if they have a common tile that can solve both edges in the same way i.e. only if the common tile has at least one bit that is set on both edges.

An example for that would be a tile that is involved in both a row conflict (Where the DOWN and UP bits are set) and a column conflict (where the LEFT and RIGHT bits are set). In that case, the same tile cannot solve both edges in the same way and therefore these two edges can be put in separate connected components. Another example would be if the 1,5 last move edge exists but also the 1 tile is involved in a conflict at the corner of the right bottom. In that case it might have its LEFT bit set in the 1,5 edge and its RIGHT bit set in the other edge. 7

7In some cases a tile will not have any bit set which means that it cannot solve the conflict of its edges at all.
However, we are not sure that it would be cost effective to add this information to our program since these cases are very rare and it would increase the constant time per node.

3. We found that if we take out some of the triples from the database the number of generated nodes increases. On the other hand the constant time decreases by a larger factor resolving in a faster solution time. We reported the version that had all the triples since it is easier to understand and reproduce. Also it generated less nodes and this fact is most important when considering larger version of the tile. The explanation for that is that many conflicts are represented in more than one edge. This adds unnecessary work in processing these edges. For example, suppose that the 9 tile is in its correct location but some other tile X (not 4) is in the corner. In our implementation we have both edges (4,9) and (9,X). However, whenever there is a (4,9) and the 9 is in its correct location there will be some other edge (9,X). When building the database and finding an edge (9,X) where 9 is in its correct location and X is in the corner this edge can be discarded because it is already captured by the (4,9) edge. If we find a method for reducing the number of edges that represent the same conflict we will get the same heuristic but with a better constant time per node and with a smaller database.

4. In order to take advantage of the available memory many other known enhancements can be added to our system. For example, we believe that adding the FSM pruning would reduced the time by a factor of 3. Also, since the average number of generated nodes for the fifteen puzzle was less than a Million, we might want to try to implement A* for this problem. This will prune all the duplicate paths and reduce the number of nodes more than the FSM pruning. However, we might have large memory requirements for each node since each node will need to store the corresponding graph with it.

5.2 The Disjoint Database

The GADB is very complicated. In this section we present another way to use additive pattern databases which we call the Disjoint Database (DADB). A simple way to construct an additive pattern database for the sliding-tile puzzles is to partition the tiles into several disjoint groups, such that no tile belongs to more than one group. We then precompute tables of the minimum number of moves of the tiles in each group that are required to solve all the tiles in that group. We call the set of such tables, one per group of tiles, a disjoint additive pattern database, or a disjoint database for short. Then, given a particular
state in the search, for each group of tiles, we use the current positions of the tiles in the
group to compute an index into the corresponding database table, to access the number of
moves required to solve the tiles in the group. We then add together the values for each
group to compute an admissible heuristic for the given state. The disjoint database is much
more simple than the the general database because we simply add up the heuristics of the
individual groups. This is because no tile can be in more than one group. In the general
database a tile can be in more than one group (edge) and thus we needed a complicated
way such as maximal-matching or vertex cover to gather data from the different edges in an
admissible manner.

As an example, Manhattan distance is a special case of a disjoint database, where each
group contains only a single tile, and we have a separate group for each tile in the puzzle.
While Manhattan distance was initially discovered by hand, it also could have been “discovered” automatically by the following process. For each tile in each given position, perform a
search until it reaches its goal location, in which all other tiles are indistinguishable. A state
of this search is uniquely determined by the position of the tile in question and the position
of the blank, and the only moves that are counted are those of the tile of interest. Since
the operators of the sliding-tile puzzle are invertible, we can perform a single search for each
tile, starting from its goal position, and record how many moves of the tile are required to
move it to every other position. Doing this for all tiles results in a set of tables which give,
for each possible position of each tile, its Manhattan distance from its goal position. Since
we only counted moves of the tile of interest, and each move only moves a single tile, we can
sum the Manhattan distances to get an admissible heuristic.

A non-trivial example of a disjoint database for the sliding-tile puzzle would be to divide
the puzzle horizontally into rows, and have a separate group for each row of tiles. These
databases would then automatically capture all the linear conflicts in a row. As a general
rule, when partitioning the tiles, we want to group together tiles that are near each other in
the goal state, since these tiles are most likely to interact with one another. As we will see
in our experimental results, disjoint databases work extremely well for the Fifteen Puzzle,
outperforming every other method we tried. One reason for this is that computing the overall
heuristic by summing the individual database values is very efficient, and almost as fast as
computing the Manhattan distance.

In the DADB we don’t form a pairs graph or a triples graph and there is no need to
calculate a vertex cover or a maximal matching of such a graph. Therefore, the size of a
group (edge) can be much larger and is only bounded by the amount of memory that we
have.

In fact, we can partition the puzzle in disjoint groups more than once, calculate a database
for each partition and then take the maximum between the different partition as an admis-
sible heuristic. For example, we can have two disjoint databases, one that divides the tiles by rows and the other by columns.

A nice attribute of the tile puzzle is that it is symmetric in a number of manners. Therefore, a database of a certain partition of the tile can also be used for a symmetric partition. For example if we partition the puzzle into rows and create a database for each row, then these databases can be used for the same partition that is created when we reflect the puzzle about the main diagonal. What we get by doing that is is a partition of the puzzle into columns. Indeed we took advantage of this idea in our experiments and used the same database for symmetric groups.

While theoretically, both the GADB and the DADB can be implemented for different sizes of groups (edges), in practice we have memory and time constraints. Thus, in the DADB we have a small number of groups but the size of each group can be relatively large. In the GADB, since we have all the possible groups, the size of each group is relatively small. In the GADB we only have pairs and triples. In the DADB we implemented groups of sizes up to 7.

Both the disjoint database and the vertex-cover additive database are special cases of storing knowledge from groups of subgoals. If we put the disjoint database on the left side of the scale and the complete general additive database on the right side then as we move from left to right, more and more "edges" are added and we need a more sophisticated method to gather all this knowledge in an admissible way. Thus, moving from left to right yields a better heuristic at the cost of a larger constant time per node. There are more than just one way way to calculate the heuristic from the database. On the end of the left side, where we have disjoint groups, we can just add the heuristics together. When more disjoint groups are added we consider taking the maximum between sets of disjoint groups. When more are added, taking the maximum will no longer be effective and we should consider a better heuristic. Thus, at the end of the right side of the scale we have all the possible groups and take the vertex-cover of them.

The question is, what is the optimal combination under a given memory capacity and a given number of tiles that are allowed in one 'edge'. Then, the question would be what is the optimal combination between different sizes of edges.

5.3 Using Vertex-cover for a selected number of groups.

We have also developed a better method to gather data from different databases in an admissible manner. This method, is a continuum between the two methods above, namely the GADB and the ADB. In this method, just like the DADB, we select a limited number of groups of tiles with potentially high values of mutual distance over the Manhattan distance.
The number of such groups should not be large is bounded by 10. However, unlike the DADB, these groups should not be disjoint and a tile can belong to more than one group. Also, the groups can be of different sizes. Now, we first calculate their mutual values of addition over the Manhattan distance for all different combinations and store them in the pattern database just as in the DADB.

However, since a tile can belong to more than one group, we cannot add up these values. On the other hand, just like the GADB we can compute the vertex-cover of them and use it as an admissible heuristic.

In fact, it is easy to see that even for two sets of disjoint partitioning, vertex-cover is always greater or equal to the maximum of them. Many times it is indeed larger.

The drawback is that the vertex-cover slows things down and is complex to compute. However, in our case we do not need to supply a general function that will calculate the vertex cover of the edges on the fly. This is because in our case, we don’t have an arbitrary graph since throughout the search process we have exactly the same groups of tiles each with a limited number of different values that can be assigned to them.

For example, in the tile puzzle, for groups of sixes the values of each group can vary from 0 to 12 with jumps of 2. Suppose that we have n groups of 6 tiles each. In this case there are 7 different values allowed for each edge. If we have n different edges then we will have $7^n$ different combinations for the graph. (n edges, 7 values for each). We can therefore precompute the vertex-cover for all these different combinations of values and keep them in a lookup table. We call this table the vertex-cover table. If for example, $n = 8$ then we have $7^8 = 5,764,801$ different combinations. Since the search space is much larger the vertex cover for many combinations will be needed more than once. This is why storing them in a lookup table is very efficient.

Now, for each new state of the search space each of the groups is in a specific location and we can can have the addition over Manhattan distance of each group of tiles from their pattern databases in a constant time. We use these values as entries into the lookup vertex-cover table and have the vertex-cover of them in a constant time. Thus we attain a better admissible heuristic with a better a constant time per node which equals the time it takes to retrieve the above values from the corresponding tables that are located in main memory.

Note again, that both the pattern database tables and the vertex-cover table can be calculated only once and can be stored in files. Then, it will be very fast to load them into the computer's memory. The overhead for calculating the tables can be amortized over all problem instances.

This should be a very fast method for calculating heuristics since we only look into lookup tables and do not calculate anything. We call this method the Combined Additive Database (CDB).
5.4 Empirical Results

In this chapter we summarize all the empirical results that we attained for the different versions of the additive database on both the fifteen and the twenty four puzzle. These versions include, the full vertex-cover, the disjoint databases and the continuum of vertex-cover on a selected set of databases.

5.4.1 Dividing the experiments into subgroups

The different experiments can be divided into subgroups by a number of aspects. The, size of the puzzle, The number of tiles in each edge and finally the different methods for gathering data from the databases into an admissible heuristic.

The different sizes of the tile puzzle

Experiments were done, both on the fifteen and on the twenty-four puzzle. Since the fifteen puzzle has about $10^{13}$ different states, a random state can be solved rather fast. Therefore, we used a set of 1000 different random states for this version of the puzzle. We report the average results for these 1000 instances. The twenty-four puzzle is much larger and contains about $10^{25}$ different states. Some random instances might take a couple of weeks to solve. Therefore, for this version we used a set of 10 different random states and reported the individual results for each of them.

The number of tiles in a database

Another distinction between different experiments is the number of tiles in each group. For each group of tiles ('edge') we can search backwards from its goal state and build a corresponding database that indicates for each permutation of these tiles in the puzzle what is the smallest number of moves that is required to arrange these tiles back into their goal positions. For a group of $n$ tiles and a puzzle of size $x$, there are $x!/(x-n)!$ different combinations. If indeed a table if size $x!/(x-n)!$ would be used, then a mapping function from a permutation of the tiles into the table should be presented. However, in order to have fast access to each entry we used a simple direct address table of size $x^n$. Each one of the $n$ tiles in the edge can be located in $x$ different possible locations. Therefore, an n dimension array with a total size of $x^n$ was used. The locations of the tile were immediate references to the table. Nevertheless, a large amount of memory from the table was wasted since many of the entries were meaningless. For example, all entries where one of the indexes was the same is not valid since we cannot have two tiles in the same location. In any case, the size of both tables increase exponentially with the number of tiles in one edge.
Table 5.1: ADB: sizes of the databases.

Table 5.1 presents the different sizes in bytes of the databases for each number of tiles in an edge that we used for both puzzles. The table assumes that each entry was one byte. However, if we only store the addition above the Manhattan distance and calculate the Manhattan distance on the fly then each entry can be stored in only 4 bits because the addition to the Manhattan distance in all our experiments was bounded by 14. Therefore, for groups of sixes in the twenty-four puzzle and for groups of sevens in the fifteen puzzle we used only 4 bits per entry, and the numbers in the table can be divided by two for these entries. This memory saving was of course at the expense of a little longer access time since it is faster to retrieve a full byte than only four bits.

The different methods for calculating the heuristic

We used three different methods to calculate the heuristics from the databases.

1. **The general additive databases. GADB** We used this method for pairs and triples. In this system we have data for all the possible pairs and triples. The heuristic is calculated by computing the vertex-cover of the conflict graph which include the pairs and triples whose mutual distance from the goal state exceeds the Manhattan distance.

2. **The disjoint databases. DADB** We divide the puzzle into a number of disjoint groups (edges) of tiles. We then calculated and stored a database for each set. Then, since the tiles in these sets are disjoint, values from these databases can be simply added to get an admissible heuristic. With this method, we can further partition the puzzle to other combinations of disjoint groups. Each of these devisions will have an admissible heuristic as the sum of its databases values. Then we can take the maximum
of all these partitionings as an admissible heuristic. For this method we used groups of fives, sixes and even sevens.

3. **Vertex-cover for selected groups. CDB** This method, is a continuum between the two methods above. In this method, we select a number of groups of tiles with potentially high values of mutual distance over the Manhattan distance. However, these groups should not be disjoint and can also be of different sizes. Now after storing their mutual values of addition over the Manhattan distance, we compute the vertex-cover of all different combinations and store them in another table. We then use values from the pattern databases as entries into the vertex-cover table and get an admissible heuristic very fast.

### 5.5 The fifteen puzzle

![Figure 5.6: The 15 tile puzzle partitioning into disjoint databases.](image)
Table 5.2: Results of the various combination of the pattern databases on the fifteen puzzle.

<table>
<thead>
<tr>
<th>No</th>
<th>Heuristic</th>
<th>Value</th>
<th>Nodes</th>
<th>Seconds</th>
<th>Nodes/Sec</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Manhattan</td>
<td>36.940</td>
<td>401,189,630</td>
<td>53.424</td>
<td>7,509,527</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>Linear conflict</td>
<td>38.788</td>
<td>40,224,625</td>
<td>10.336</td>
<td>3,891,701</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>Pairs</td>
<td>40.432</td>
<td>9,983,886</td>
<td>10.401</td>
<td>959,896</td>
<td>1,000</td>
</tr>
<tr>
<td>4</td>
<td>Pairs+Triples</td>
<td>42.792</td>
<td>707,476</td>
<td>5.076</td>
<td>139,376</td>
<td>1,300</td>
</tr>
<tr>
<td>5</td>
<td>Disjoint5 A</td>
<td>40.938</td>
<td>7,318,428</td>
<td>1.258</td>
<td>5,817,510</td>
<td>3,145</td>
</tr>
<tr>
<td>6</td>
<td>Disjoint5 B</td>
<td>41.560</td>
<td>3,090,405</td>
<td>0.540</td>
<td>5,722,922</td>
<td>3,145</td>
</tr>
<tr>
<td>7</td>
<td>Disjoint5 C</td>
<td>40.970</td>
<td>5,708,806</td>
<td>1.006</td>
<td>5,674,757</td>
<td>3,145</td>
</tr>
<tr>
<td>8</td>
<td>Disjoint5 B+C</td>
<td>42.052</td>
<td>1,366,565</td>
<td>0.339</td>
<td>4,031,135</td>
<td>6,290</td>
</tr>
<tr>
<td>9</td>
<td>Disj5 B+C+B*+C*</td>
<td>42.512</td>
<td>812,723</td>
<td>0.293</td>
<td>2,773,799</td>
<td>6,290</td>
</tr>
<tr>
<td>10</td>
<td>Disjoint6 A</td>
<td>42.132</td>
<td>1,840,134</td>
<td>0.347</td>
<td>5,412,588</td>
<td>33,554</td>
</tr>
<tr>
<td>11</td>
<td>Disjoint6 A+A*</td>
<td>42.924</td>
<td>617,555</td>
<td>0.163</td>
<td>3,788,680</td>
<td>33,554</td>
</tr>
<tr>
<td>12</td>
<td>CDB6 7</td>
<td>43.211</td>
<td>397,107</td>
<td>0.134</td>
<td>2,963,485</td>
<td>34,377</td>
</tr>
<tr>
<td>13</td>
<td>CDB6 10</td>
<td>43.485</td>
<td>242,186</td>
<td>0.115</td>
<td>2,105,965</td>
<td>332,806</td>
</tr>
<tr>
<td>14</td>
<td>Disjoint7 A+A*</td>
<td>44.586</td>
<td>116,985</td>
<td>0.047</td>
<td>2,489,042</td>
<td>268,437</td>
</tr>
<tr>
<td>15</td>
<td>CDB7 (5)</td>
<td>44.563</td>
<td>97,730</td>
<td>0.044</td>
<td>2,221,136</td>
<td>402,669</td>
</tr>
<tr>
<td>16</td>
<td>CDB7 (6)</td>
<td>44.731</td>
<td>76,634</td>
<td>0.037</td>
<td>2,071,189</td>
<td>419,548</td>
</tr>
</tbody>
</table>

Table 5.2 presents the results that we got on the fifteen tile puzzle. The experiments were done on a 500MHZ Pentium 3 PC with 512MB of main memory. The different partitionings to groups are presented in Figure 5.6.

All the data in the table is averaged on the same 1000 different random initial states. The average optimal path that was returned by all the methods for these states was 52,522. Each row presents a different database and methods to calculate the heuristics. The first column indicates the heuristic method. The second column, **Value** presents the heuristic value of the initial state. The **Nodes** column shows the number of generated nodes. The **Seconds** column, has the amount of CPU time that was needed to solve a problem. The next column presents the ratio of nodes per second. Finally, the last column shows the size of memory in Kbytes that was needed for the databases as well as for the vertex-cover lookup table for these heuristics that it was applicable.

**Row 1.** has data for simple Manhattan distance.

**Row 2.** has results for Manhattan distance plus linear conflict which was hardwired into the code.
Row 3. has the results for the GADB system with pairs.

Row 4. Has the results for the GADB system for both pairs and triples.

Row 5. has data for the DADB labeled 5A in Figure 5.6.

Row 6. has data for the DADB labeled 5B in Figure 5.6.

Row 7. has data for the DADB labeled 5C in Figure 5.6.

Row 8. has data when we used both 5B and 5C and took the maximum of them.

Row 9. took the maximum between the following four databases: 5B, Its reflection about the main diagonal (which we label 5B*), 5C and 5C*. Note, that we can have the data for the reflected database directly by just assigning each tile to its reflected tile.

Row 10. has the data of the DADB labeled 6A in Figure 5.6.

Row 11. has data for the maximum between 6A and 6A* (The reflection of 6A about the main diagonal).

Row 12. has data for the combined database (CDB) system. We used 7 groups of sixes. The groups are shown if the center frame of Figure 5.6. which is labeled VertexCover6. The first group is the six that is labeled A. The other 6 groups of sixes are all the permutations of the 3^2 rectangle labeled by B. There are of two these for each of the three corners that are not the blank. In this system, all the possible values for the vertex cover of these groups were precalculated in advance and were stored in a vertex-cover table.

Row 13. has the results for the CDB when using 10 groups of six. They are the same 7 from row 12 plus the 3 permutations of the group labeled by C in the second frame labeled by VertexCover6.

Row 14. contains the results for a DADB that took the maximum between the databases that are labeled Disjoint7A and Disjoint7A*.

Row 15. gives data for the CDB that used the following 5 groups of seven: A1, A1* A2, A2* and B.

Row 16. presents data for a CDB that used all the groups of row 15 plus the group of six labeled by C or actually all the frames in the bottom of Figure 5.6.

The rows in the table are ordered by the time needed to solve an average problem. The table shows that as the heuristics use larger databases, we get a reduction of both the number of generated nodes and the actually time that is needed to solve the puzzle. In fact, our best combination at row 16, shows an improvement factor of 5,235 in the number of nodes and a factor of 1443 in the actual running time over simple Manhattan distance. While the Manhattan distance program solves an average problem in almost one minute, our fastest program solves 30 problems in one second.

However, another observation that we can make from this data is that as the heuristics become more accurate, a larger amount of memory is needed. This observation is very
accurate and we can see that the rows in Table 5.2 are also ordered by the amount of memory that were required to store the tables. While the Manhattan distance program did not need any amount of memory the GADB needed a Mega byte of memory and our fastest combination needed 420 Megabytes of memory. This observation makes sense because with more memory more combinations and interactions can be stored and a more accurate heuristic function can be obtained.

It is definitely clear that for the fifteen puzzle, the best tradeoff between efficiency and simplicity would be to use a DADB with the largest groups as possible. The combination of row 14 has this attribute. Note that even though both rows 15 and 16 are faster than row 14, they use the complex system of calculating an additional table, namely the vertex-cover table. Also, we can observe that here, the GADB shows relatively poor results when compared to all combinations of the DADB and CDB. While the GADB generates a relatively small number of nodes its drawback is the large constant time per node needed to compute the heuristic. Clearly, the GADB is not the system of choice for the tile puzzle. However, we choose to describe it here in great detail because it is a general idea and might be used to a greater benefit in other domains. In fact, we will see latter that for the vertex-cover problem in the next chapter the GADB system was the system of choice.

A nice attribute of all the combinations that we used, including the GADB, is that all of them outperform the simple Manhattan distance in both the number of generated nodes and the average time. While the Manhattan distance program was the fastest program in the sense of number of nodes per second it seems that for all the other combinations that we used, the larger overhead per node pays off since it produces a more accurate heuristic function and therefore a much smaller number of generated nodes.

5.5.1 Comparison with the first work of pattern database

At this stage we must refer to the work of pattern database which is presented in [10]. They report an improvement of a up to a factor of 1038 in the number of generated nodes over the simple Manhattan distance solver. However, in real time they report that their best version ran only 12 times faster than Manhattan distance in the sense of CPU time for solving an average single problem instance. Our GADB system for example, solved the fifteen puzzle 32 times faster than simple Manhattan distance. However, the weakness of their approach was the large amount of memory that they needed to store their database. They report that they used up to I gigabyte of memory for their database. Our systems clearly needed to use a smaller amount of memory. A significant advantage of our GADB system for example is that it needs only 274K bytes for the fifteen puzzle. Our GADB database was 4,000 times smaller then their database and yet we found solutions faster. In our system there is a lot
of free memory and therefore we can use other techniques for improving the search since most techniques are orthogonal to each other. For example, we can use the FSM pruning system\cite{33} to further reduce the number of generated nodes.

\section{5.6 The twenty-four puzzle}

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure5_7.png}
\caption{The 24 tile puzzle partitioning into disjoint databases.}
\end{figure}

Since the twenty-four puzzle is much larger, we only used 4 different heuristics. The first one was the full GADB system, i.e. we used databases of both pairs and triples and calculated the vertex cover of the conflict graph. The second combination that we used was 4 partitioning of the puzzle into 4 fives and one quadruple. The 4 partitioning were 5A, 5B, 5C and 5C* (which is 5C reflected about the main diagonal) from Figure 5.7. This combination needed approximately 78MB of memory. Each entry was stored in one byte. We then have partitioned the puzzle into 4 groups of sixes as in Disjoint 6A from Figure 5.7. Note that the 2*3 rectangle needed to be stored only once for all three permutations. Then, we reflected the whole partition of 6A about the main diagonal to a total of 8 groups of sixes. For the sixes we looked both at the maximum between 6A and 6A* (the third option) and we also built a vertex-cover table for that combination (forth option). The sixes were stored with 4 bits per entry each with only the addition over the Manhattan distance. This combination
needed 242MB of main memory. The vertex-cover lookup table was of size $7^8$ and needed no more than 6MB of memory.

Table 5.3 shows the average heuristic values for the above additive databases combinations averaged over 10,000 different random instances. It also shows the number of nodes per second that the search process generated for each of these combinations. The full GADB system of pairs and triples is the slowest since a vertex-cover function is applied for each new state. The partitioning into fives had actually 20 edges and 20 databases but it was faster than the partitioning into 8 groups of sixes since the sixes were stored in 4 bits per entry rather than a in full byte. More importantly is that since we used only two databases tables for the 8 different combinations of sixes, we had additional overhead in reflecting each combination into the database.

Tables 5.4 and 5.5 present the data of the experiments on the twenty-four puzzle. For each case we also include the permutation of the tiles in the initial state.

It seems that in general, the more complicated methods solved the puzzle faster but this was not the case in all the instances. For example, in problem 1, the full vertex-cover system was better than the disjoint database of fives. In general our best system, CDB, tends to solve the puzzle about 5 times faster than the GADB system.

A phenomena that can be observed from the above data is that when using a couple of disjoint databases and then taking a vertex-cover of them does not improve the heuristic much more than simply taking the maximum of them. In fact, in more than 80% of a large random sample of cases they had the same heuristic value. The reason is as follows. Suppose for example that two groups (edges) have two tiles in common. If one of the groups has a price of 6 and the other has a price of 4 then both maximum and vertex-cover heuristics will have a price of 6. The vertex-cover will be 6 because the two tiles that belong to both group will only yield an addition of 4. We need another tile with an addition of 2 in order to get to a total of 6. Only if both have a price of 6 and only two tiles in common, then the vertex-cover will be 8 and the maximum will be 6. For the tile puzzle, these cases are very rare. Most of the edges of the tile puzzle have prices of no more than 2 or 4.
<table>
<thead>
<tr>
<th>Heuristic</th>
<th>No</th>
<th>Path</th>
<th>Nodes</th>
<th>seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>14 5 9 2 18 8 23 19 12 17 15 0 10 20 4 6 11 21 1 7 24 3 16 22 13</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pairs+Triples</td>
<td>1</td>
<td>95</td>
<td>306,958,148</td>
<td>1,757</td>
</tr>
<tr>
<td>Disjoint5</td>
<td>1</td>
<td>95</td>
<td>16,507,915,825</td>
<td>8,710</td>
</tr>
<tr>
<td>Disjoint6</td>
<td>1</td>
<td>95</td>
<td>2,031,102,635</td>
<td>1,446</td>
</tr>
<tr>
<td>vertex-cover 6</td>
<td>1</td>
<td>95</td>
<td>1,377,159,819</td>
<td>1,063</td>
</tr>
<tr>
<td>16 5 1 12 6 24 17 9 2 22 4 10 13 18 19 20 0 23 7 21 15 11 8 3 14</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pairs+Triples</td>
<td>2</td>
<td>96</td>
<td>65,125,210,009</td>
<td>692,829</td>
</tr>
<tr>
<td>Disjoint5</td>
<td>2</td>
<td>96</td>
<td>291,813,860,897</td>
<td>155,895</td>
</tr>
<tr>
<td>Disjoint6</td>
<td>2</td>
<td>96</td>
<td>211,884,984,458</td>
<td>147,493</td>
</tr>
<tr>
<td>vertex-cover 6</td>
<td>2</td>
<td>96</td>
<td>158,889,554</td>
<td>123,018</td>
</tr>
<tr>
<td>6 0 24 14 8 5 21 19 9 17 16 20 10 13 2 15 11 22 1 3 7 23 4 18 12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pairs+Triples</td>
<td>3</td>
<td>97</td>
<td>52,906,797,645</td>
<td>524,603</td>
</tr>
<tr>
<td>Disjoint5</td>
<td>3</td>
<td>97</td>
<td>63,793,892,595</td>
<td>33,915</td>
</tr>
<tr>
<td>Disjoint6</td>
<td>3</td>
<td>97</td>
<td>21,148,144,928</td>
<td>14,972</td>
</tr>
<tr>
<td>vertex-cover 6</td>
<td>3</td>
<td>97</td>
<td>14,448,309,001</td>
<td>11,294</td>
</tr>
<tr>
<td>7 13 11 22 12 20 1 18 21 5 0 8 14 24 19 9 4 17 16 10 23 15 3 2 6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pairs+Triples</td>
<td>4</td>
<td>98</td>
<td>8,465,759,895</td>
<td>72,911</td>
</tr>
<tr>
<td>Disjoint5</td>
<td>4</td>
<td>98</td>
<td>40,146,522,260</td>
<td>20,908</td>
</tr>
<tr>
<td>Disjoint6</td>
<td>4</td>
<td>98</td>
<td>10,991,471,966</td>
<td>7,809</td>
</tr>
<tr>
<td>vertex-cover 6</td>
<td>4</td>
<td>98</td>
<td>9,262,519,107</td>
<td>7,016</td>
</tr>
<tr>
<td>17 1 20 9 16 2 22 19 14 5 15 21 0 3 24 23 18 13 12 7 10 8 6 4 11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pairs+Triples</td>
<td>5</td>
<td>100</td>
<td>715,535,336</td>
<td>3,922</td>
</tr>
<tr>
<td>Disjoint5</td>
<td>5</td>
<td>100</td>
<td>16,810,328,542</td>
<td>8,824</td>
</tr>
<tr>
<td>Disjoint6</td>
<td>5</td>
<td>100</td>
<td>2,899,007,625</td>
<td>2,024</td>
</tr>
<tr>
<td>vertex-cover 6</td>
<td>5</td>
<td>100</td>
<td>2,480,350,516</td>
<td>1,894</td>
</tr>
</tbody>
</table>

Table 5.4: Twenty-four puzzle results of the different combinations of their ADB.
<table>
<thead>
<tr>
<th>Heuristic</th>
<th>No</th>
<th>Path</th>
<th>Nodes</th>
<th>seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 0 10 19 14 16 3 15 20 22 9 6 18 5 13 12 21 8 17 23 11 24 7 14</td>
<td>6</td>
<td>101</td>
<td>10,415,838,041</td>
<td>151,083</td>
</tr>
<tr>
<td>Pairs+Triples</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Disjoint5</td>
<td>6</td>
<td>101</td>
<td>56,831,491,399</td>
<td>30,603</td>
</tr>
<tr>
<td>Disjoint6</td>
<td>6</td>
<td>101</td>
<td>103,460,814,368</td>
<td>74,100</td>
</tr>
<tr>
<td>vertex-cover 6</td>
<td>6</td>
<td>101</td>
<td>86,134,496,298</td>
<td>65,252</td>
</tr>
<tr>
<td>21 22 15 9 24 12 16 23 2 8 5 18 17 7 10 14 13 4 0 6 20 11 3 1 19</td>
<td>7</td>
<td>104</td>
<td>46,196,984,340</td>
<td>717,454</td>
</tr>
<tr>
<td>Pairs+Triples</td>
<td>7</td>
<td>104</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Disjoint5</td>
<td>7</td>
<td>104</td>
<td>175,226,281,669</td>
<td>92,923</td>
</tr>
<tr>
<td>Disjoint6</td>
<td>7</td>
<td>104</td>
<td>106,321,596,792</td>
<td>76,522</td>
</tr>
<tr>
<td>vertex-cover 6</td>
<td>7</td>
<td>104</td>
<td>85,774,231,083</td>
<td>66,491</td>
</tr>
<tr>
<td>18 14 0 9 8 3 7 19 2 15 5 12 1 13 24 23 4 21 10 20 16 22 11 6 17</td>
<td>8</td>
<td>108</td>
<td>15,377,764,962</td>
<td>82,180</td>
</tr>
<tr>
<td>Pairs+Triples</td>
<td>8</td>
<td>108</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Disjoint5</td>
<td>8</td>
<td>108</td>
<td>96,568,770,957</td>
<td>51,235</td>
</tr>
<tr>
<td>Disjoint6</td>
<td>8</td>
<td>108</td>
<td>116,202,273,788</td>
<td>81,643</td>
</tr>
<tr>
<td>vertex-cover 6</td>
<td>8</td>
<td>108</td>
<td>83,209,058,152</td>
<td>64,424</td>
</tr>
<tr>
<td>3 2 17 0 14 18 22 19 15 20 9 7 10 21 16 6 24 23 8 5 1 4 11 12 13</td>
<td>9</td>
<td>113</td>
<td>135,129,533,132</td>
<td>747,443</td>
</tr>
<tr>
<td>Pairs+Triples</td>
<td>9</td>
<td>113</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Disjoint5</td>
<td>9</td>
<td>113</td>
<td>719,927,253,823</td>
<td>378,780</td>
</tr>
<tr>
<td>Disjoint6</td>
<td>9</td>
<td>113</td>
<td>428,017,148,189</td>
<td>299,133</td>
</tr>
<tr>
<td>vertex-cover 6</td>
<td>9</td>
<td>113</td>
<td>351,665,302,180</td>
<td>272,608</td>
</tr>
<tr>
<td>23 14 0 24 17 9 20 21 2 18 10 13 22 1 3 11 4 16 6 5 7 12 8 15 19</td>
<td>10</td>
<td>114</td>
<td>726,455,970,727</td>
<td>4,214,591</td>
</tr>
<tr>
<td>Pairs+Triples</td>
<td>10</td>
<td>114</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Disjoint5</td>
<td>10</td>
<td>114</td>
<td>1,789,774,310,778</td>
<td>3,401,728</td>
</tr>
<tr>
<td>Disjoint6</td>
<td>10</td>
<td>114</td>
<td>1,519,052,821,943</td>
<td>3,320,098</td>
</tr>
<tr>
<td>vertex-cover 6</td>
<td>10</td>
<td>114</td>
<td>1,331,681,205,551</td>
<td>3,390,445</td>
</tr>
</tbody>
</table>

Table 5.5: Twenty-four puzzle results of the different combinations of their ADB.
believe that if we took larger edges, the number of tiles that these edges have in common increase and vertex-cover will even be less effective. Thus, this idea of CDB does not yield a larger improvement over the simpler DADB because of the domain specific behavior if the tile puzzle that most of the edges have a low price, comparing it to the number of tiles that they have in common. On other domains, where the edges will have larger prices compared to the number of individual subgoals that they have in common, it may result in much better performance.

5.6.1 Comparing the results of both version of the puzzle

It seems that the the DADB and CDB systems are much more effective than the GADB on the fifteen puzzle. On the twenty-four puzzle the improvement was somewhat more modest. This is because when grouping together 6 or 7 tiles than we got almost half of the tiles of the fifteen puzzle. On the twenty-four puzzle, where the size is much larger, grouping of 6 tiles together is less effective.

We have observed that the conflict graph of the GADB for the twenty-four puzzle did not have significantly more edges than the GADB for the fifteen puzzle. The number of edges and conflicts tends to be the same for both versions of the puzzle. We believe that the reason is as follows. In the perspective of a single tile there are 3 stages that it has to go through until it reaches its goal location. First, it should leave the area of its current position. During that stage it might be in conflicts with other tiles. For example if it is located near a corner it might be in conflicts with other tiles that are located near that corner. At the second stage it has to move across the puzzle to the area of its goal location. During this stage, there are not many conflicts since traveling to the area of the goal location can be done usually via many paths and conflicts can be bypassed. Mostly, only linear conflicts might really influence this stage. The length of this stage is measured by the Manhattan distance. Finally, the third stage is when it is already in the area of the goal location, and it should be moved to its exact location. In the third stage there may be also many conflicts with other tiles in the area. Since many of the conflicts are caused by the corners and both puzzles have 4 corners then the number of conflicts due to corners tends to be stable for both puzzles. In fact, since the second stage for the twenty-four puzzle is longer, i.e., a tile needs to travel more in order to get to its goal area, we can determine that our heuristic function is more accurate for smaller versions of the puzzle. Actually, we have seen that the number of additional moves that this heuristic adds to the Manhattan distance is nearly the same for both puzzles and since the Manhattan distance for the twenty-four puzzle is larger we can say that the our enhancements are more effective for the fifteen puzzle.

In general, we can say that due to the small size of the fifteen puzzle we could have built
a nice disjoint database. However, for the twenty-four puzzle we could only save relatively smaller databases and the relative performance of the DADB in comparison to the GADB was smaller. We believe that for larger versions of the puzzle such as the 6x6, 35 puzzle we will only be able to build relatively very small disjoint database and the GADB for that puzzle might do relatively much better.

5.6.2 Comparison to Previously Reported Results

These are optimal solutions to random instances of the Twenty-Four Puzzle. The first reported optimal solutions for these initial states were presented in [33]. We use a different numbering of the problem instances here, based on their optimal solution lengths, but the correspondence is easily established, since the solution lengths of this set of problems happens to be unique.

Our work is different from the work in [33] in two important respects. The first is that the heuristic used in [33] was based only on pairwise distances, including all linear conflicts, and its implementation was domain-specific, handling the interactions between the different pairwise distances in an ad-hoc way. Here we extended the heuristic to triples, handled the interactions with the general vertex cover algorithms, and introduced disjoint pattern databases.

The second difference is that [33] used a technique, based on finite-state machines (FSMs), to prune duplicate nodes representing the same state arrived at via different paths. Such duplicate nodes increase the nodes generated by a depth-first search on a graph with cycles. The results in [33] indicated that this FSM pruning reduced the number of nodes generated by IDA* on the Twenty-Four Puzzle by a factor that ranged from 2.4 to 3.6.

For the current work, we choose not to use FSM pruning for several reasons. The first is that the efficiency gains were only modest. More importantly, however, the FSM technique is complex, making it difficult for other researchers to reproduce the same results. Thus, to facilitate future comparisons with this work, we choose to solve the problems without duplicate pruning, except for eliminating the parent of a node as one of its children.

5.7 Summary and conclusions for the ADB work on the tile puzzles

We have found the first optimal solutions to random instances of the Twenty-Four Puzzle, a problem with almost $10^{25}$ states. The branching factor of the problem space is 2.3676, and optimal solutions average over 100 moves. We also present the best results to date on the
Fifteen Puzzle, finding optimal solutions in about a third of a second on average.

To solve these problems, we implemented IDA* with new admissible heuristic functions, based on the idea of pattern databases [10]. Rather than computing the costs of solving individual subgoals independently, a pattern database considers the costs of solving several subgoals simultaneously, taking into account the interactions between them.

In general, a more complex heuristic function increases the constant time per node evaluation, but decreases the number of nodes generated to solve a problem. Even with the modest increases in average heuristic values, and the complexity of computing them, in this problem it pays off. As machines get faster and larger problems are addressed, this work suggests that even small improvements in admissible heuristic functions will be cost effective. We have developed three new techniques for doing this automatically.

We believe that this is a general approach to the discovery and implementation of admissible heuristic functions. The obvious next step is to apply it to other problems. All combinatorial problems involve solving multiple subgoals. We are proposing heuristics based on the simultaneous consideration of two, three, or more subgoals, in such a way that their values can be added together to create a more accurate admissible heuristic function.

In the following chapters we will present solution to other problems with heuristics that are based on the ADB ideas.
Chapter 6

Finding optimal solutions to the Vertex Cover Problem with Heuristic Search

The main idea of the additive pattern database is that looking into interactions between groups of unsolved subgoals of the problem can resolve with significant knowledge about the cost of solving these subgoals. Using this knowledge properly results in a more accurate heuristic function and improves the search. We apply this technique to solve the vertex cover problem. We build a precalculated database that stores data about a vertex cover of groups of individual vertices of a given graph. This database is then consulted during the search to get an admissible heuristic of the actual vertex cover. We show that storing larger groups of vertices in the database and then using this information for the heuristic is more cost effective than smaller groups. The improvement that we obtained in our experiments with such a database is a couple of orders of magnitude for both the number of generated nodes and the amount of time needed to solve a given problem.

6.1 The Vertex Cover domain

In a vertex cover problem (VC), we are given graph G, and are asked to find the smallest (optimal) vertex cover. A vertex cover is a subset V of vertices from G such that each edge of G is incident with at least one vertex of V. The weights of the edges in this problem are meaningless and therefore, we can assume that the edges have a uniform weight. VC is a famous NP-complete problem [15] and many time it appears in the form of a decision problem, (fixed parameter problem). In that form, we are given a graph G and a parameter K and and we are asked to decide whether the graph G has a vertex cover of size at most
K. In our work we will use the general form of finding the smallest vertex cover for a given graph.

Each individual vertex may be either included in or excluded from a vertex cover. Therefore, a brute force technique would be to cycle through all the $2^n$ different subsets of vertices and find the smallest one that is a vertex cover. The time complexity for such an algorithm is $O(2^n)$.

Solving this problem using a search tree is straightforward. We order the vertices\(^1\) of the graph in some order and then search a binary tree, where each level of the tree corresponds to a different vertex of the graph. At each node of the search tree, a decision has to be made whether to include the current vertex in the VC or not. The left branch will add the corresponding vertex to the VC and the right branch will ignore this vertex and exclude it from the VC. The leaves of this binary tree are the $2^n$ different subsets of the vertices of the graph.

At any given node of the search tree, some of the vertices have already been included in the partial vertex cover, some were excluded and the rest are pending and will be decided at lower levels of the tree. Therefore, for a node $n$ we call the number of vertices that were already added to the vertex cover $g(n)$, since they represent the cost already incurred by the partial solution represented by node $n$.

In the following chapters we will present a number of different admissible heuristics to get a lower-bound estimate on the remaining number of vertices that must be included in the VC, given the partial VC. We call this number $h(n)$ since it represents an estimation of the number of vertices that must be added to the VC in the future. These heuristics apply the additive database principle of storing a precalculated database with data about mutual heuristics and interaction between groups of unsolved subgoals.

We build databases for pairs of vertices, triples, quadruples etc, Then, we use the knowledge from these databases to get an admissible heuristic of the number of vertices that must be included in the VC.

As suggested by the additive database theory, we will show that even though the constant time per node for calculating heuristics from larger databases increases, a very large improvement is achieved both in the number of generated nodes and in the overall time to solve a problem.

\(^1\)Throughout this work will use the term vertex to denote a vertex of the input graph and the term node will be used for a node of the search tree.
6.2 Related work

Vertex cover is a problem of central importance in computer science. It was one of the first problems that was proved to be NP-complete [15]. The VC problem is very practical and has a lot of commercial applications. For example, in computational Biochemistry, it is used to resolve conflicts between sequences of DNA by excluding some of them for a sample. Therefore, many efforts were made to design efficient approximation algorithms. Instead of presenting them here we refer the reader to a comprehensive survey of such algorithms [9]. Most of these approximation algorithms use known local search methods such as hill-climbing, genetic algorithms [3], simulated annealing [23] etc.

In addition, many algorithms were developed to find optimal solutions to VC but most of them were designed to solve the fixed parameter form of the problem, i.e., to determine given a graph $G$ and an integer $K$, whether the graph has a VC of size at most $K$.

A famous algorithm to solve the fixed parameter form of the VC problem is the algorithm proposed by Buss et al. [6]. Their algorithm is based on a method which Downey and Fellows [14] classify as the method of reduction to a problem kernel. The main idea is to reduce the problem to an equivalent one where the size of the VC should be now bounded by $K'$ which is a function of $K$. We will not elaborate on this algorithm but we will only indicate that it is based on the fact that vertices with a degree greater than $K$ must be included in the VC because otherwise all their neighbors must be included in the VC. Nevertheless, this algorithm cannot be used in the general form of the problem where a minimal VC is needed.

Papadimitriou and Yannakakis [39] developed an algorithm to solve the VC problem when $K$ is restricted to be logarithmic in the input size. Their algorithm first finds a maximum matching to the graph and then expands it to a vertex cover.

Many efforts have been made in the last few years to develop algorithms that are based on what Downey and Fellows [14] classify as the bounded search tree method. Basically, at each node of the search tree we have a partial vertex cover and the resulting graph is obtained by deleting the vertices in the partial vertex cover and the edges incident with them from the original graph. The edges of the tree correspond to various possibilities for adding of vertices to the existing partial vertex cover. A node in the search tree is pruned if the partial vertex cover at that node has a size that exceeds the parameter $K$ or the resulting graph at that node is empty. The corresponding search tree is of size $2^K$.

A large portion of the works that were done on such search trees are enhancements that reduce the size of the tree by pruning special classes of nodes or by combining several nodes together. For example, a simple enhancement is that if a vertex is excluded from the VC then all of its neighbors must be included in it. Therefore, when reaching to these neighbors there is no decision to make whether to include these neighbors and the branches that do
not include them are pruned.

At this point many attempts were made to further reduce the size of the search tree. Balasubramanian et al. [44] presented an algorithm with a search tree of size $1.32472^k$. This bound was improved by Downey et al. [45] to $1.31951^k$. Niedermeier and Rossmanith [38] further improved this bound to $1.29175^k$ thus breaking the 1.3 barrier in the base of the exponent.

All these algorithms are very complicated and are hardwired to the special search tree and the vertex cover. In general, each of these algorithms divide the nodes of the search tree to a number of classes. The distinction between the classes is done by looking on special structures of vertices and edges of the input graph in the area of the vertex that is to be decided in the specific node of the search tree. At each class, because of the special structure of the graph, some decision on these vertices are forced and thus the size of the search tree decreases. Much of the effort of these works is to determine new classes and structures that can reduce the size of the search tree.

For example, Balasubramanian et al. [44] grow the following search tree. At each node of the search tree a vertex $x$ from the resulting graph is picked. Then, a depth first search for three levels is performed. This ends up either with a triangle $x, a, b, x$ or a path $x, a, b, c$ of length 3. In the first case observe that any vertex cover will have the vertices $x, a$ or $a, b$ or $x, b$. In the second case, any vertex cover must have the vertices $x, b$ or $a, c$ or $a, b$. Their algorithm branches according to all these possibilities instead of branching according to an individual vertex at a time. This is one example of an idea of to reduce the size of the search tree. The other ideas mentioned above are more complicated than this simple example.

Niedermeier and Rossmanith [38] divide the vertices to special classes which depend on the degree of the vertex. Then, special structures of the graph near that vertex are taken into consideration and sometimes some of the decision regarding some vertices are forced thus reducing the size of the search tree.

All these algorithms calculate the cost of a node as the size of the partial vertex cover. These algorithms are very complicated and it seems that a lot of thought and hard work were put into them. However, none of them try to gather knowledge from all the vertices in the remaining graph in order to get a lower bound on the number of vertices that must be added to the partial vertex cover. In the following sections we will develop such heuristics that reduces the size of the tree by a great deal. Also while the above mentioned algorithms were developed for the fixed parameter form of the problem we use the more general form.
6.3 The search tree

While many ideas of reducing the size of the search tree were suggested by the developers of the algorithms that were described in the previous section, we decided to include only the simple ideas from these works and use a somewhat more simple search tree. The reason for this decision was that these methods are very complicated and contain many classes and cases and therefore are hard to understand and implement. Our intention is to show a general method of using data from the resulting graph to get an admissible heuristic. Thus, our method is orthogonal to any of the other ideas of the algorithms above. One may want to add any of these enhancements to ours and might obtain a higher speedup to the search process.

While we did not implement the complicated ideas from the above methods, we did add some simple and straightforward ideas in order to reduce the size of the search tree. These ideas were also used by the above mentioned algorithms as the simple cases that they provide. We will describe these ideas in the following section.

6.3.1 Definitions

The basic structure of the search tree is as follows. We order the vertices of the graph in some order\(^2\) and at each level of the tree we decide to either include the corresponding vertex to the VC or exclude it from the VC. We branch at the tree according to that decision. We call this vertex the major vertex of that level. Whenever a vertex is added to the VC then all the edges incident with that vertex are removed from the graph since they are covered by this vertex. We call the growing set of vertices that are included in the VC the partial VC.

At any stage of the search we can therefore divide the vertices of the graph into the following three classes:

- Vertices that were included in the VC. We call them the included vertices.
- Vertices that were excluded from the VC. We call them the excluded vertices.
- Vertices that we have not decided about them yet. We call such vertices the free vertices. We call the subgraph that is spanned by the free vertices the free graph.

At the root of the search tree all the vertices are free vertices. Upon proceeding down the tree, free vertices are assigned to either the partial VC or to the set of excluded vertices. The heuristics that we will describe later are an admissible estimation of the number of vertices from the free vertices that must be added to the partial VC.

\(^2\)We will see later that ordering the vertices in decreasing order of degree is the most efficient order.
6.3.2 Eliminating degree-one vertices

A simple observation on the nature of a feasible VC yields that if seeing a degree-one vertex on the graph then without the loss of generality, we can exclude this vertex from the VC and include its neighbor. Thus the first stage of our algorithm is to find all the degree-one vertices and add their neighbor to the VC and delete both them and the edges incident with them from the graph. Note, that this is not a one pass iteration over the graph since a new degree-one vertex might be created during the process when deleting edges from the graph. After this stage is performed we are assured that every vertex of the free graph has a degree of at least two.

6.3.3 Neighbors of excluded vertices

If a vertex is not included in the VC then all its neighbors must be included in it. This is because one end point for each edge must be included in the VC. Therefore, if at any point of the search tree a vertex is excluded from the VC then its neighbors must be included in it. Thus, at each node of our search tree one branch adds the major vertex to the VC while the other branch adds all its neighbors that are not yet included. In the former case, the major vertex is moved from the free vertices to the included vertices. In the later case the major vertex is moved from the free vertices to the excluded vertices and its neighbors are moved from the free vertices to the included vertices. This is of course a very great improvement over the simple binary tree that either includes or excludes the major vertex from the VC.

![The graph](image1)

![The search tree](image2)

Figure 6.1: The search tree of the vertex cover domain

Figure 6.1 shows an example of a simple graph and the corresponding search tree. At the root, we either go left and add vertex 0 to the VC or we branch to the right and exclude it. In the right child of the root all the neighbors of vertex 0, namely 1, 2 and 3 are added to the VC and the search stops for the right subtree.
6.3.4 Ordering the vertices of the graph

Each vertex of the graph is associated with a level of the tree and at that level (if this vertex is still free) we decide to either include it or exclude it from the VC. When sorting the graph in decreasing order of their degree, nodes with many neighbors will be treated sooner. Since these nodes have more neighbors, they will have larger costs when being excluded from the VC. Therefore, before starting the search, we first sort the graph in decreasing order of degree and add vertices in that order rather than by a random order. We have found that this simple enhancement may speed up the search process by a factor of more than 10 for many cases. This improvement was found to be valid for all the algorithms and the versions of the heuristic functions that we used. We have tried other methods of sorting the graph but this one was significantly better than all the others.

6.3.5 Dealing with degree-one or isolated vertices during the search process.

Whenever a vertex is added to the VC we remove the edges that are incident with it from the graph. This may cause some of its neighbors to become a degree-one vertices or even isolated vertices with a degree of zero. In our search of the tree, isolated vertices are immediately added to the excluded vertices immediately. For degree-one vertices we activate the same process described above i.e., exclude them from the VC and add their neighbor to it. Note again that this might cause a chain reaction that by adding a vertex to the VC new isolated vertices or degree-one vertices may pop out.

![Diagram showing vertex ordering process](image)

Figure 6.2: The free graph. Vertices in this graph are the free vertices

Figure 6.2 demonstrates this process. If vertex 0 is added to the VC then the free graph now includes vertices 1, 2 and 3. Vertex 1 is a degree-one vertex and therefore vertex 2 can be safely added to the VC. This causes vertex 3 to be isolated and it can be added to the excluded vertices.

These lines of thinking can be taken much further and indeed the works that were mentioned above [44] [45] [38] do apply these kinds of approaches. While we could have imple-
mented more of these enhancements to our search tree for the reason of simplicity we decided not to include them.

6.4 The additive database heuristics for the VC problem

6.4.1 Lower bound on the VC of the free graph

All the above algorithms [44][45][38] calculate the cost of a node $n$ in the search tree as the number of vertices in the partial vertex cover associated with $n$. We call this number $g(n)$ as it represents the number of vertices that are already included in the VC. The next step is to find an admissible heuristic to the free graph. Or in other words, we are looking for an efficient way to produce a lower bound on the number of vertices that must be included in the vertex cover due to edges in the free graph i.e., edges that are not covered by vertices that are already included in the VC. If we find such a lower bound, we can call it $h(n)$ as it represents a heuristic estimation of the number of additional vertices that must be added to the partial VC in order to produce a feasible VC. The cost of a node $n$ will therefore be the well known cost function: $f(n) = g(n) + h(n)$.

6.4.2 Cliques of free vertices

The main idea of the additive database theory is to try to estimate the cost of solving groups of unsolved subgoals while looking at interactions and conflicts between them. The theory says to first consider individual subgoals alone and calculate the singleton heuristic which is the sum of solving each individual subgoal by itself. The next step is to look at all possible pairs, find out about their mutual interaction and find the pairwise heuristic associated with each pair. In this manner we construct the pairs graph, where each edge corresponds to the pairwise heuristic of its vertices. Then, we should take edges from that graph in an admissible way. Ideally we should get the larger set of disjoint edges or a maximum matching of the pairs-graph. The next step is to calculate a heuristic which is based on interactions and conflicts of triples of subgoals. With this knowledge we construct a hyper-graph with hyper-edges that represent costs of groups of triples. Then we should proceed to quadruples etc. It seems that these techniques can be implemented very efficiently for the VC problem as described below.
$h_1$: Singleton heuristic

When looking at individual vertices from the free graph we can not get any kind of lower bound because there are no edges in question. Thus the singleton heuristic\(^3\) is zero and is trivial for the VC problem. Therefore we get:

$$h_1(n) = 0$$

$h_2$: pairwise heuristic

Looking at pairs of vertices of the free graph does yield a non trivial lower bound. If a pair of vertices from the free graph has an edge between them then we know that at least one of these vertices must be included in the VC. Thus, an admissible pairwise cost of each pair of nodes $(u, v)$ from the free graph is

$$h_2(u, v) = \begin{cases} 
1 & \text{if } (u, v) \in E \\
0 & \text{otherwise}
\end{cases}$$

The pairs graph which is the graph with the mutual costs of all the pairs of the free graph is therefore identical to the free graph since each edge should be covered by at least one vertex. Now, if we find an edge that both its vertices were not included in the VC we can add 1 to the heuristic and mark both vertices as suspected (i.e. suspected to be in the VC and may not be used further for the heuristic). We should keep looking for edges in which both endpoints are not in the VC and are not marked yet as suspected. In other words, in order to get an admissible heuristic we can only add costs from disjoint edges. So we are actually looking for a maximum matching of the free graph. A maximum matching of the free graph is therefore an admissible heuristic that is associated with pairs of vertices and can be called the pairwise heuristic. Thus

$$h_2(n) = \text{maximum matching of the free graph.}$$

Higher order heuristic

The next step is to look on triples of vertices. A close observation shows that if three vertices in the free graph form a triangle then two of these vertices must be added to the VC.\(^4\) The triple cost associated with these three vertices is therefore two. When looking only on pairs of vertices, then taking disjoint edges from a triangle will resolve with a cost of one since only one edge can be taken.

---

\(^3\)Which was the Manhattan distance in the tile puzzle domain.

\(^4\)Balasubramanian et al.[44] used this fact but only on the vertices near the major vertex.
A triangle is actually a clique of size three just as an edge is a clique of size two and a single vertex is a clique of size one. We can take this further to the general case and say that an admissible heuristic for a clique of size $k$ is $k - 1$. This is because of the following proposition:

**Proposition 2:** $VC$ of a clique of size $k$ is at least $k-1$.

**Proof:** Suppose the contrary, that there exists a feasible $VC$ of size $k - 2$. In that case, there are two vertices from the clique that are not included in the $VC$. The edge between them is thus not covered by any vertex. A contradiction.

$$\text{C}=0 \quad \text{C}=1 \quad \text{C}=2 \quad \text{C}=3$$

Figure 6.3: Cliques of different sizes and their admissible heuristics.

Therefore, if we find a clique of size $k$ in the free graph we can add $k - 1$ to the heuristic as illustrated in Figure 6.3. In order to obtain admissibility, we can not allow a vertex to participate in more than one clique and we should take disjoint cliques. Grouping vertices in larger cliques as possible will improve the heuristic of these vertices. For example, suppose that vertices X, Y and Z form a triangle. If we look on cliques of size 3 then the heuristic for these 3 vertices is 2 since they are a clique of size 3. If however, we only look at the pairwise heuristic (cliques of size 2) then the heuristic for these three vertices is only 1 since we can only find 1 disjoint edge in a triangle.

### 6.4.3 Constructing the database and computing the heuristics

If we would like to use these cliques heuristics with a brute force technique then for each node of the search tree we would try to identify cliques in the free graph and use them for the heuristic.

However, a much more efficient way would be to use the additive database method and precalculate all cliques of the graph and store them in a database. Then, at each node of the search tree we should consult the database for relevant cliques for that node.

Just as we did in the tile puzzle, we used two different systems to construct the additive database, namely a *general* database that contains full knowledge about all the possible cliques and a *disjoint* database which contains only a set of disjoint cliques. In the former case upon reaching each new node of the search tree we will have to consult the database in
order to retrieve an efficient partition of the free vertices into disjoint cliques. In the later case, this partitioning to disjoint cliques is done only once in the beginning of the search and at each node we are forced to use this specific partition.

The advantage of the former case is that the partition into disjoint cliques is as suitable as possible for each node of the search tree. The advantage of the latter case is the lack of the overhead that is needed to partition the free graph to disjoint cliques for every node.

The general additive database

In this form of database we first decide what is the largest size of cliques that we want in our database. We denote this number by $q$. Then, we search the graph for all cliques that are of size of $q$ or smaller. These cliques are saved in the database.

The Clique problem is known to be NP complete. However, since we are only interested in cliques that are of size at most $q$, the time complexity to find these cliques is $n^q$ where $n$ is the size of the graph. In our experiments, we have found that for random graphs, cliques of size 5 or more are very rare. Therefore, we have limited our database to contain only cliques that are of size smaller or equal to 4. Thus building a database of cliques of size not larger than 4 took a fraction of a second and we can omit this complexity from the time complexity of the whole search process.

A simple way to store all the cliques would be just a simple array of cliques. However, we used a more time efficient data structure which retrieves the relevant cliques much faster.

The space of cliques of size $q$ is actually $\binom{n}{q}$ and can be represented by a tree of depth $q$ where each node of the tree corresponds to a different vertex of the graph. A node $X$ will have a child $Y$ if $Y$ is larger than $X$. In this manner the tree keeps the lexicographic order and each node can only have children with larger indexes.

![Figure 6.4: The potential cliques-space tree.](image)

Figure 6.4 illustrates the cliques-space tree of all potential possible cliques of a size up to 3 of an arbitrary graph with 4 vertices indexed from 0 to 3. Each node at level 1 corresponds to a clique of size 1 (a single vertex). Each node at level 2 corresponds to a clique of size
2 which include the node and its father. In general, each node at level \( q \) corresponds to a clique of size \( q \) which contains itself and its ancestors.

In order to build a general database of cliques for a specific graph we should actually take only a portion of the above cliques-space tree where a node \( X \) will have a child \( Y \) if \( Y \) and \( X \) and their ancestors form a clique. Other nodes from the cliques-space tree are pruned. Actually, each branch in such a tree is a clique in the graph. Visiting all leaves of this cliques-tree will give us all the largest cliques for the graph. If we bound the traversal of the cliques-tree to a certain level \( p \), we get all cliques of size \( p \).

![The general cliques DB](image)

![The graph](image)

Figure 6.5: The actual cliques tree for a specific graph.

Figure 6.5 presents a graph with 4 vertices and the cliques database for it. Note, that there are 2 cliques of size 3 \( \{0, 1, 2\} \) and \( \{0, 2, 3\} \) and therefore there are 2 nodes in the tree at level 3. Each of these nodes is the leaf of a branch that corresponds to that clique. There are 5 cliques of size 2 in the graph (5 edges) and therefore, there are 5 nodes at level 2.

The additive database theory talks about a hyper-graph where each hyper-edge contains the mutual cost of the vertices in that hyper-edge. This hyper-graph can be drawn from the cliques tree in the following way. Each hyper edge is identified by a node in the tree. The other vertices of such hyper-edges are the ancestors of those nodes including itself.

Once we have the input graph i.e., the graph that we want to find the VC for, the first stage is to precompute the additive database for it which is actually the cliques tree. After building the cliques tree, we start the search process and start developing the search tree. At each node of the search tree we consult the cliques tree database for a heuristic for the free graph. As the search proceeds and we get to deeper levels of the search tree, the free graph becomes smaller and we need to find the best heuristic for this particular free graph.

As suggested by the additive database theory, a maximum matching of the hyper graph is the best admissible heuristic. However, calculating it will take a very long time which might be exponential for cliques of size of more than 2 which form a hyper-graph with hyper-edges with more than 2 vertices. We decided to compromise and our program simply looks for a
maximal matching. Note that a maximal matching is not necessarily a maximum matching. A maximum matching is the largest set of disjoint edges, while a maximal matching is a set of disjoint cliques that can not be increased by adding more cliques to it.

The algorithm for obtaining an admissible heuristic from the general database is first presented and then explained.

```c
get-heuristic(cliques tree, free graph)
{
    h=0;
    mark all vertices that are not in the free graph
    For p = q down to 2
        Traverse the cliques tree to depth p. Backtrack at mark nodes.
        if reached node at depth p then
            a new clique of size p was found
            mark the clique's vertices
            h = h + (p-1)
        endif
    endfor
    return(h)
}
```

If the larger clique in the DB is q, we first try to find a disjoint set of cliques of size q. Then we add cliques of size q-1 to the set, then q-2 etc. This procedure is done by q-1 traversals of the cliques-tree database from left to right to from depth q to depth 2. We visit the children of a node of the cliques-tree only if the corresponding vertex is a free vertex and was not taken yet to form a clique by this previous iterations of the traversals. Otherwise, we backtrack since this particular node can not be taken for a disjoint clique from the free graph. When we reach the depth of the search we have a clique and we set a flag to indicate that these vertices were already taken for a clique. Note that the cliques are built in a lexicographic order.

We found that this traversal of the cliques tree does not take a lot of time since the cliques tree is not large in the first place and many of its nodes are pruned. Also, since the search tree is a binary tree, most of the nodes are in deep levels. In these levels there are only a few free vertices so a large portion of the cliques-tree is pruned.
The Disjoint database

A disjoint database is simpler than the general database. Given a parameter $q$, we first find a disjoint set of cliques of size of $q$ or less. This is done in the same manner as in the general DB. However, while in the general DB this search for a disjoint set of cliques is done at each node of the search tree for the free graph that corresponds to this node, in the disjoint database this is done only once for the whole search process.

The disjoint database is actually a partition of the whole graph into disjoint cliques. At the beginning of the search when the free graph is identical to the whole graph, then a clique of size $q$ can contribute $q - 1$ to the heuristic. As the search proceeds, some of the vertices that form a clique in the DB are already included in the VC so a particular clique of size $q$ can add to the heuristic $q-1$ minus the number of included vertices from that clique. This can be calculated very fast since once we know the disjoint partition and the corresponding heuristic for it we can keep this number but subtract one each time one of its vertices is added to the VC.

While this method of the disjoint database is very fast to compute and incurs a small amount of overhead per node, it can lose a lot of its effectiveness during the search since cliques for which many of their vertices were taken to the VC are no longer helpful, but yet we are forced to look on cliques that include them.

An example of the different databases

General database:$(0, 1, 2), (1, 2, 3)$. (all cliques of size 2)

Disjoint database:$(0,1,2)$.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{example.png}
\caption{Figure 6.6: The general DB versus the disjoint db on the vertex cover domain.}
\end{figure}

Figure 6.6 illustrates the difference between the general database and the disjoint database. In the precalculation two cliques of size 3, namely $(0, 1, 2)$ and $(1, 2, 3)$ are being found as well as all the cliques of size 2. All these cliques are inserted into the general database. At this stage, in order to build the disjoint database we look for a disjoint set of cliques and find
the clique of \((0, 1, 2)\). This clique is the only clique inside the disjoint database. After the construction of the two databases the search begins. At the root of the tree all the vertices are free. This situation is illustrated in Figure 6.6.A. For the root node, the general database finds a clique of 3 \((0, 1, 2)\) and evaluates \(h_{\text{general}} = 2\). The disjoint database is forced to use the cliques in its database and since all vertices of \((0, 1, 2)\) are free we get that \(h_{\text{disjoint}} = 2\). At the next stage, vertex 0 is taken to the VC making \(g = 1\) as illustrated by Figure 6.6.B. The free graph now contains vertices 1, 2 and 3. The general database looks for a disjoint set of cliques in the free graph and finds that \((1, 2, 3)\) is a clique of size 3. Therefore it evaluates that \(h_{\text{general}} = 2\). The disjoint database is forced to use the partition from the beginning which is to look at the clique of \((0, 1, 2)\). One of these vertices is already taken to the VC so we have to subtract one from the heuristic of that clique and thus \(h_{\text{disjoint}} = 2 - 1 = 1\). The disjoint database looses information while the general database looks for new sets of cliques.

We can enhance the disjoint DB if we take more than one partition to disjoint sets of cliques. In other words, we can store a number of different disjoint databases, each of them storing a partition of the graph to disjoint cliques. Then we calculate the heuristic of each of these databases and take the maximum of them as an admissible heuristic. If for example we would take two disjoint databases of cliques for Figure 6.6, then one of them will contain \((0, 1, 2)\) and the other will contain \((1, 2, 3)\). After taking vertex 0 to the vc as in Figure 6.6.B, the disjoint database of \((0, 1, 2)\) will have a heuristic of 1. However, the disjoint database of \((1, 2, 3)\) was not effected yet and still has a heuristic of 2. Taking a maximum of these two partitionings will resolve with a heuristic of 2. But of course, we are limited to a small number of partitionings of disjoint sets. Otherwise the overhead per node will be large and will not have any advantage over the general database which is much more accurate.

### 6.5 The experiment design

We have conducted a number of experiments trying to solve the VC problem on different graphs. Each of these experiments was defined by the following parameters:

- The type of the database.
- The type of the graph.
- The search algorithms.

#### 6.5.1 The type of the database

We used both types of databases that were described above, namely the general DB and the disjoint DB. When using the general DB, at each node of the tree we look for a disjoint set
of cliques which are as large as possible.

For the disjoint DB we precalculated three disjoint databases which include different sets of disjoint cliques. Then, during the search, we take the maximum of the heuristics of these three databases. These three disjoint databases were calculated by three different passes on the cliques-tree\(^5\). The three passes are as follows:

- A pass from left to right. This one has more large cliques of vertices with a higher degree. This is because the vertices of the graph are ordered by their degree and this pass first looks at vertices with a higher degree.

- A pass from right to left. This one has larger cliques of vertices with a low degree.

- A mixed pass from the middle to left and then from the middle to the right. This one has larger cliques from the middle indexed vertices.

We then take the maximum of these three heuristics to represent the disjoint heuristic. The parameter \( q \), for the largest size of cliques varied in our experiments from 0 to 4.

### 6.5.2 The type of the graph

We have used two classes of graphs. The first is a random graph. We had two variables that were associated with each of the random graphs:

- The number of nodes in the graph \( n \).
- The average degree or branching factor \( b \).

The random graph was built as follows. First, we determined the values of these two variables and built an empty graph with \( n \) nodes. Then, each edge is independently added to the graph with a probability of \( b/n \). Therefore the graph that we obtained has an average degree of \( b \) but with some variance.

The other class of graphs comprises Delaunay triangulations of planar point patterns that are generated by a Poisson point process [2] that distributes points at random over a unit square using a uniform probability density function. Delaunay triangulation of a planar point pattern is constructed by creating a line segment between each pair of points \((u,v)\) such that there exists a circle passing through \( u \) and \( v \) that encloses no other point. In some sense it can be said that, in this triangulation, each point is joined by a line segment to all of its nearest neighbors and to no other points. The graph whose vertices are the points and whose arcs are the joined pairs of points will be called “Delaunay graph”.

\(^5\)The tree is built such that children of a node are ordered from left to right by increasing number of their indexes.
To construct Delaunay graphs for our experiments, we used the qhull [7] software package, which generates Delaunay triangulations. Figure 6.7 and Figure 6.8 illustrate a random graph and a Delaunay graph of size 15 respectively.

6.5.3 The search algorithms

We used two linear space admissible search algorithms. Both are from the A* [18] family. The first algorithm was Iterative deepening A* (IDA*) [26] which is a linear space version of A*. IDA* is actually a sequence of iterations of DFS. In each iteration it expands all the nodes having a total cost not exceeding a given global threshold for that iteration. The threshold for the next iteration is the lowest cost of a generated node from the current iteration that exceeded the current threshold. IDA* generates new nodes in a best-first order. In each
iteration, IDA* generates all the nodes from the previous iterations plus new nodes for the current iteration. Therefore, many modes are generated more than once.

The other algorithm that we used was Depth First Branch and Bound (DFBnB) [34]. DFBnB is very effective when the depth of the solution in the search tree is known in advance. DFBnB performs a depth-first search of the tree but uses a global threshold for pruning nodes. This threshold is the best solution that was found so far. At the beginning of the search, this threshold is set to infinity and DFBnB reaches a solution in one dive into the tree. The threshold is set to the cost of that solution and the DFS is continued until a better solution is found. When we get to a node where the cost function is greater than the current threshold we know that solutions under that node will not be better than the best solution and the subtree under that node can be pruned. DFBnB starts with a solution and improves on it until it finds the optimal solution. Then, it keeps on searching the tree to verify that a better solution is not feasible. Thus in that sense DFBnB is an anytime algorithm.

DFBnB is used only when we are guaranteed to have a feasible solution with every path or when we can bound the search to a specific level. Otherwise it can enter an infinite loop. In our case the depth of the tree is bounded by the number of vertices in the graph.

It was proven [12] that an admissible search algorithm must at least visit all the nodes that are visited by A*. IDA* does not visit any other node, but visits some of the nodes more than once. DFBnB never visits any node more than once, but visit many nodes that are not visited by A*. See [34] for a detailed comparison between these two algorithms.

6.6 Empirical results

6.6.1 Random graphs with 150 vertices

The first set of experiments are for random graphs with 150 vertices. The average degree was varied from 3 to 60 and we had 30 instances for each of the average degrees. Table 6.1 presents the optimal vertex cover for the different average degrees. The data here and in all the following tables and figures is the average over 30 instances. All the experiments were conducted on a 500MHZ pentium III PC.

Figure 6.9 illustrates the performance of using the general additive database with different sizes of cliques for the databases. There are four curves in the figure. $F_1$ is the curve where cliques of size 1 were used. This is a trivial case since the heuristic for this case is 0. $F_2$ is the curve where cliques of size 2 were used or in other words a maximal matching of the free graph was used as the heuristic. $F_3$ and $F_4$ are where cliques of size of up to 3 or up to 4 respectively were taken. The figure clearly shows that using a database with cliques of two
Figure 6.9: Time in seconds for the general additive database. The DFBnB algorithm and database of cliques of sizes 1 through 4 on random graphs of size 150.

significantly speeds up the search process. Moving to database with cliques of 3 speeds it up again. Database of 3 and 4 seems to behave about the same since cliques of 4 are very rare for random graphs. With sparse graphs, where the average degree is small, $F_3$ is a little better than $F_4$ while for dense graphs, where the average degree is large, $F_4$ is better. This is because with dense graphs there is a larger chance for the existence of cliques of size 4 and consulting such a database is more likely to pay off.

We also performed experiments where cliques of 5 were manually installed in these graphs and the results show that clique databases of 4 and 5 significantly improve the search over using only 3.

Another interesting phenomena is that the time complexity tends to increase when the average degree increases and then drops again. This is valid for all of the algorithms tested. The hardest degree that we obtained is 15. The reason for that is the behavior of the search tree. When the graph is very sparse, then during the search many vertices become degree-one or isolated vertices and we don’t branch at their level of the tree since the decision whether to include them in the VC or not was already determined. With very dense graphs pruning in the search tree occurs due to cases where we exclude a vertex from the VC. In such cases all the neighbors of that vertex are added to the VC and there is no decision to make when they become the major vertices. With a dense graph, after a couple of times when we exclude a vertex from the VC a large number of vertices are included and many subtrees are pruned.

Figure 6.10 shows a comparison of both types of databases and both DFBnB and IDA*. The curves represent database of cliques of a size up to 3. DFBnB with the general database seems to be the fastest algorithm.

Figure 6.11 shows the performance of the IDA* algorithm with the disjoint database.
<table>
<thead>
<tr>
<th>Degree</th>
<th>Vertex cover</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>72.43</td>
</tr>
<tr>
<td>6</td>
<td>89.30</td>
</tr>
<tr>
<td>10</td>
<td>100.70</td>
</tr>
<tr>
<td>15</td>
<td>113.29</td>
</tr>
<tr>
<td>20</td>
<td>118.86</td>
</tr>
<tr>
<td>30</td>
<td>127.14</td>
</tr>
<tr>
<td>40</td>
<td>131.07</td>
</tr>
<tr>
<td>50</td>
<td>134.86</td>
</tr>
<tr>
<td>60</td>
<td>137.14</td>
</tr>
</tbody>
</table>

Table 6.1: The average optimal vertex cover for graphs with 150 vertices.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Database</th>
<th>Cliques</th>
<th>VC</th>
<th>Nodes</th>
<th>seconds</th>
<th>nodes/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFBnB</td>
<td>General</td>
<td>1</td>
<td>113.29</td>
<td>720,582,454</td>
<td>4,134</td>
<td>174,306</td>
</tr>
<tr>
<td>DFBnB</td>
<td>General</td>
<td>2</td>
<td>113.29</td>
<td>19,159,780</td>
<td>231</td>
<td>82,982</td>
</tr>
<tr>
<td>DFBnB</td>
<td>General</td>
<td>3</td>
<td>113.29</td>
<td>4,261,831</td>
<td>70</td>
<td>60,451</td>
</tr>
<tr>
<td>DFBnB</td>
<td>General</td>
<td>4</td>
<td>113.29</td>
<td>4,170,981</td>
<td>73</td>
<td>57,136</td>
</tr>
<tr>
<td>DFBnB</td>
<td>Disjoint</td>
<td>1</td>
<td>113.29</td>
<td>720,582,454</td>
<td>4,134</td>
<td>174,306</td>
</tr>
<tr>
<td>DFBnB</td>
<td>Disjoint</td>
<td>2</td>
<td>113.29</td>
<td>187,467,358</td>
<td>1,463</td>
<td>128,139</td>
</tr>
<tr>
<td>DFBnB</td>
<td>Disjoint</td>
<td>3</td>
<td>113.29</td>
<td>105,669,961</td>
<td>849</td>
<td>124,464</td>
</tr>
<tr>
<td>DFBnB</td>
<td>Disjoint</td>
<td>4</td>
<td>113.29</td>
<td>103,655,233</td>
<td>844</td>
<td>123,252</td>
</tr>
<tr>
<td>IDA*</td>
<td>General</td>
<td>1</td>
<td>113.29</td>
<td>4,354,159,438</td>
<td>27,509</td>
<td>158,281</td>
</tr>
<tr>
<td>IDA*</td>
<td>General</td>
<td>2</td>
<td>113.29</td>
<td>43,452,913</td>
<td>590</td>
<td>73,648</td>
</tr>
<tr>
<td>IDA*</td>
<td>General</td>
<td>3</td>
<td>113.29</td>
<td>6,961,792</td>
<td>131</td>
<td>53,143</td>
</tr>
<tr>
<td>IDA*</td>
<td>General</td>
<td>4</td>
<td>113.29</td>
<td>6,733,959</td>
<td>140</td>
<td>48,099</td>
</tr>
<tr>
<td>IDA*</td>
<td>Disjoint</td>
<td>1</td>
<td>113.29</td>
<td>4,354,159,438</td>
<td>27,509</td>
<td>158,281</td>
</tr>
<tr>
<td>IDA*</td>
<td>Disjoint</td>
<td>2</td>
<td>113.29</td>
<td>583,847,228</td>
<td>4,867</td>
<td>119,990</td>
</tr>
<tr>
<td>IDA*</td>
<td>Disjoint</td>
<td>3</td>
<td>113.29</td>
<td>255,286,357</td>
<td>1,935</td>
<td>131,930</td>
</tr>
<tr>
<td>IDA*</td>
<td>Disjoint</td>
<td>4</td>
<td>113.29</td>
<td>250,776,912</td>
<td>1,936</td>
<td>129,533</td>
</tr>
</tbody>
</table>

Table 6.2: Performance of the different algorithms on graphs with 150 vertices and an average degree of 15.
The average degree of the graph

VC: time versus density. 150 vertices. Cliques of size 3

Figure 6.10: Comparison of all types of algorithms with a database of cliques of a size up to 3 on random graphs of size 150.

This combination is worse than all the other combinations. However, here also, databases with larger cliques produce faster solutions.

Table 6.2 shows the performance of both IDA* and DFbB and both the general database and the disjoint database on graphs with 150 vertices and an average degree of 15.

While IDA* is a little faster in terms of nodes per second, it seems that using the same database with DFbB is much faster usually by a factor of 8. DFbB was faster than IDA* in all the experiments that we conducted. This is because we searched a bounded tree and IDA* visits many nodes more than once.

The best combination that we obtained was DFbB with a general additive database. While using a database for the heuristic consumes more time per node it pays off in the overall overhead. A Uniform Cost Search with no heuristic (labeled in the table with cliques of 1), solves the problem in 4,134 seconds on average. Using a database with cliques of size 2 improves the performance to 231 seconds. Using a database with cliques of up to 3 or 4, further improves the performance to 70 seconds. The overall improvement is 2 orders of magnitude. The improvement is even better if we look at the number of generated nodes.

The disjoint database is faster than the general database in the overhead per node. However, it seems that for random graphs, the general database solves the problems faster by a factor of up to 10. The reason is that with the disjoint database we are forced to use a specific partition into cliques at the beginning. This partition may not be the best for deeper levels of the search tree. This is not true for the general database where at each node of the search tree we partition the free vertices into cliques in a manner that suites the current situation.

Figure 6.12 shows the behavior of the DFbB with the heuristic from the general
Figure 6.11: Time in seconds for the disjoint database. The IDA* algorithm and the disjoint database of cliques of sizes 1 through 4 on random graphs.

Figure 6.12: Random graphs with average degree of 15. DFBnB, general database. The size of the graphs varies from 100 to 150 with a constant average degree of 15. Here also we can see that a database with larger cliques significantly outperforms the database with smaller or no cliques at all.

6.6.2 Delaunay graphs

Table 6.3 shows the performance of the different algorithms on Delaunay graphs with 100 vertices. It seems that here the differences between all the different combinations of IDA* and DFBnB and the different databases is less significant than in random graphs. In random graphs the difference between DFBnB with the general database and IDA* with the disjoint database was almost a factor of 10. Here, in Delaunay graphs, the difference is no more
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Database</th>
<th>Cliques</th>
<th>VC</th>
<th>Nodes</th>
<th>seconds</th>
<th>nodes/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFBnB</td>
<td>General</td>
<td>1</td>
<td>67.87</td>
<td>23,595,355</td>
<td>102.45</td>
<td>231,327</td>
</tr>
<tr>
<td>DFBnB</td>
<td>General</td>
<td>2</td>
<td>67.87</td>
<td>1,817,620</td>
<td>13.97</td>
<td>130,108</td>
</tr>
<tr>
<td>DFBnB</td>
<td>General</td>
<td>3</td>
<td>67.87</td>
<td>64,482</td>
<td>0.84</td>
<td>76,764</td>
</tr>
<tr>
<td>DFBnB</td>
<td>General</td>
<td>4</td>
<td>67.87</td>
<td>61,466</td>
<td>0.89</td>
<td>69,062</td>
</tr>
<tr>
<td>DFBnB</td>
<td>Disjoint</td>
<td>1</td>
<td>67.87</td>
<td>23,595,355</td>
<td>102.45</td>
<td>231,327</td>
</tr>
<tr>
<td>DFBnB</td>
<td>Disjoint</td>
<td>2</td>
<td>67.87</td>
<td>2,012,129</td>
<td>13.27</td>
<td>151,629</td>
</tr>
<tr>
<td>DFBnB</td>
<td>Disjoint</td>
<td>3</td>
<td>67.87</td>
<td>239,530</td>
<td>1.86</td>
<td>128,779</td>
</tr>
<tr>
<td>DFBnB</td>
<td>Disjoint</td>
<td>4</td>
<td>67.87</td>
<td>211,153</td>
<td>1.64</td>
<td>128,751</td>
</tr>
<tr>
<td>IDA*</td>
<td>General</td>
<td>1</td>
<td>67.87</td>
<td>84,708,496</td>
<td>420.03</td>
<td>201,672</td>
</tr>
<tr>
<td>IDA*</td>
<td>General</td>
<td>2</td>
<td>67.87</td>
<td>2,981,445</td>
<td>26.73</td>
<td>111,593</td>
</tr>
<tr>
<td>IDA*</td>
<td>General</td>
<td>3</td>
<td>67.87</td>
<td>70,016</td>
<td>1.06</td>
<td>66,052</td>
</tr>
<tr>
<td>IDA*</td>
<td>General</td>
<td>4</td>
<td>67.87</td>
<td>65,826</td>
<td>1.10</td>
<td>59,841</td>
</tr>
<tr>
<td>IDA*</td>
<td>Disjoint</td>
<td>1</td>
<td>67.87</td>
<td>84,708,496</td>
<td>420.03</td>
<td>201,672</td>
</tr>
<tr>
<td>IDA*</td>
<td>Disjoint</td>
<td>2</td>
<td>67.87</td>
<td>3,343,357</td>
<td>24.72</td>
<td>135,249</td>
</tr>
<tr>
<td>IDA*</td>
<td>Disjoint</td>
<td>3</td>
<td>67.87</td>
<td>245,120</td>
<td>2.09</td>
<td>117,282</td>
</tr>
<tr>
<td>IDA*</td>
<td>Disjoint</td>
<td>4</td>
<td>67.87</td>
<td>213,115</td>
<td>1.82</td>
<td>117,096</td>
</tr>
</tbody>
</table>

Table 6.3: Performance of the different algorithms on Delaunay graphs with 100 vertices for finding vertex cover.
than a factor of 2. The disjoint database seems to perform better on Delaunay graphs than on random graphs. The reason for this is that on Delaunay graphs, while going deeper in the tree, neighbors of excluded nodes are more likely to be included in its cliques so the destruction of the initial partition is not that significant. On random graphs however, neighbors of a vertex can be anywhere in the graph so the initial partition becomes irrelevant sooner.

Table 6.4 presents the performance of DFBnB on Delaunay graph of size 150. Note that the improvement of using a general database with cliques of a size of up to 4 is a factor of 6874 over the simple algorithm with no heuristics. While the simple algorithm solves the problem in 281,828 seconds the general database with cliques of size 4 needs only 41 seconds.

### 6.7 Discussion and conclusions for the vertex cover work.

The primary purpose of our approach is finding optimal solutions and therefore we perform a global search on all the possible different solutions. However, the DFBnB search can also be used as an anytime algorithm and return the best solution found so far. In fact, we have observed that many times more than 70% of the running time was spent after the optimal solution was reached in order to assure that a better solution does not exist. So actually after 30% of the running time we already have an optimal solution in hand but we do not know that until the search halts.

Our approach cannot be directly compared to other VC algorithms that are designed

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Database</th>
<th>Cliques</th>
<th>VC</th>
<th>Nodes</th>
<th>seconds</th>
<th>nodes/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFBnB</td>
<td>General</td>
<td>1</td>
<td>102</td>
<td>60,676,456,977</td>
<td>281,828.01</td>
<td>215,164</td>
</tr>
<tr>
<td>DFBnB</td>
<td>General</td>
<td>2</td>
<td>102</td>
<td>556,993,145</td>
<td>5,313</td>
<td>104,835</td>
</tr>
<tr>
<td>DFBnB</td>
<td>General</td>
<td>3</td>
<td>102</td>
<td>2,542,114</td>
<td>43.33</td>
<td>58,668</td>
</tr>
<tr>
<td>DFBnB</td>
<td>General</td>
<td>4</td>
<td>102</td>
<td>2,098,207</td>
<td>41.33</td>
<td>50,767</td>
</tr>
<tr>
<td>DFBnB</td>
<td>Disjoint</td>
<td>1</td>
<td>102</td>
<td>60,676,456,977</td>
<td>281,828.01</td>
<td>215,164</td>
</tr>
<tr>
<td>DFBnB</td>
<td>Disjoint</td>
<td>2</td>
<td>102</td>
<td>1,101,077,823</td>
<td>7,264.25</td>
<td>151,580</td>
</tr>
<tr>
<td>DFBnB</td>
<td>Disjoint</td>
<td>3</td>
<td>102</td>
<td>36,046,592</td>
<td>264.97</td>
<td>136,540</td>
</tr>
<tr>
<td>DFBnB</td>
<td>Disjoint</td>
<td>4</td>
<td>102</td>
<td>18,129,883</td>
<td>131.46</td>
<td>138,396</td>
</tr>
</tbody>
</table>

Table 6.4: Performance of the different algorithms on Delaunay graphs with 150 vertices for finding vertex cover.
to return suboptimal solutions and usually perform local searches since our algorithm is
designed for finding optimal solutions and performs a global search. However, we can combine
our global search with any other suboptimal algorithm as follows. We first run the suboptimal
algorithm and get a suboptimal solution from it. We can then run our DFBnB search with
that solution as the initial threshold in order to either improve that solution or verify that
it is an optimal solution.

Our algorithms is more comparable to the algorithms that solve the fixed parameter
VC problem. While they only concentrate on reducing the size of the tree they only use
Uniform Cost Search, i.e. they only calculate the $g(n)$ cost function. We show a way to
further expand that cost function to also use the $h(n)$ part and resolve with the traditional
$f(n) = g(n) + h(n)$ cost function. Our improvement can be added to any of these algorithms.

In conclusion, we have shown a number of admissible heuristics that optimally solve
the VC problem using the additive database that stores knowledge about the mutual cost of
groups of subgoals together. We have shown here another domain where this general method
of developing an admissible heuristic function is applicable and effective. Once again we have
shown that even though more complicated heuristics incur a larger overhead per node they
seem to outperform simpler heuristics in the overall time since they reduce the number of
generated nodes by a greater factor.
Chapter 7

Finding optimal solutions to the Graph-partitioning problem with Heuristic Search

The general idea of the additive pattern database is that looking into interactions between subgoals of the problem and adding them to an existing heuristic function results in a more accurate heuristic function and improves the search. We have applied this idea to the graph partitioning problem and we introduce a sequence of heuristic functions that can optimally solve this problem. Each heuristic is more complicated and looks more deeply into interactions between vertices of the graph. The best heuristic function that we have improves on the simplest one by a couple of orders of magnitude.

7.1 The Graph partitioning problem

In a graph partitioning problem (GPP), we are given a set of vertices and edges that constitute a graph, and must divide all the vertices into two equal-sized subsets, so that the number of edges that go from a vertex in one subset to one in the other is minimized. This problem has applications in the physical layout of VLSI circuits, where the vertices are circuit elements, and the edges are wire connecting them. In the simple form of this problem, we assume that all edges have a uniform weight of one.

To solve this problem using heuristic search, we form a search tree where each node corresponds to a partial partition of some of the vertices of the graph.¹ We order the vertices in the graph, and then search a binary tree, where each level of the tree corresponds

¹Throughout this work will will use the term vertex to denote a vertex of the input graph and the term node will be used for a node of the search tree.
to a different vertex. At each node of the search tree, the left branch would assign the corresponding graph vertex to one subset and the right branch would assign it to the other subset. If the number of vertices in the graph is $n$ then this tree is a binary tree of depth $n - 1$. While leaves of such a binary tree correspond to the $2^n$ different subsets of $n$, we are only interested in the leaves that partition the vertices into equal size subsets. Therefore, each subtree rooted by a node where one of the partitions has more than half of the vertices is pruned.

![Diagram of a search tree for the graph partitioning domain.](image)

**Vertices 1, 2 and 3 are assigned.**

**Vertices 4, 5, ..., n are free vertices.**

Figure 7.1: The search tree for the graph partitioning domain.

Figure 7.1 illustrates such a tree. At any given node of the search tree, corresponding to a partial solution, some of the vertices have already been assigned and some have not. As a result, some edges of the graph already go from a vertex in one subset to a vertex in the other. We call this number of such edges $g(k)$, since they represent the cost already incurred by the partial solution represented by node $k$. In the following sections we will present a number of different admissible heuristics to get a lower-bound estimate on the remaining number of edges that must cross the partition, given the partial partition. We call this number $h(k)$. Each one of these heuristics looks more deeply than its predecessor into interactions between unassigned vertices and into edges that are connected to such vertices. We will show that even tough the constant time per node for calculating more complicated heuristics is larger, we achieve a very large improvement in both the number of generated nodes and in the overall time to solve a problem.

At first, the purpose of this work was to demonstrate the additive pattern database principle that was very successful in the tile puzzles. We first tried to develop heuristics that are based on the knowledge given by each of the unassigned vertices individually and add up these values to an admissible heuristic. Then, we looked at pairs of vertices and interactions between them and searched for an admissible way to gather their mutual values. However, After looking more deeply into the GPP domain we found that we can improve the performance.

---

1 We assume without the loss of generality that the first node goes to the first subset of the partition.

113
these heuristics much more, and at that point our work began to deviate a little from the additive database theory i.e. looking at pairs of subgoals that are not yet solved is better than looking at singles alone etc. However, in our opinion this work on GPP can still be part of the additive theory since it shows the same principle that looking more deeply into interactions between subgoals that are not yet solved resolves in a large improvement in the search effort.

7.2 Related work

The GPP is very famous, and many researchers have looked into this problem in the past [22] [11] [41] [4] [46]. This problem is very practical and has a lot of commercial applications and thus a large number of algorithms were developed over the years to solve it. However, since the problem is known to be NP-hard, most of these algorithms were developed to find a sub-optimal solution to the problem.

A large portion of these approaches are based on local search. In a local search, we start with any feasible solution that divides the vertices into two subsets. Then, we start swapping pairs of vertices between the subsets as long the cost of the solution decreases. This local search may get caught up in a local optimum, and therefore does not guarantee an optimal solution. Many algorithms were developed in order to improve the above local search schema. They all differ in the way they generate the neighbors of a given feasible partition.

Perhaps the most popular of these improvements, is the algorithm that was presented by Kernighan and Lin[22] known as the KL algorithm. The KL algorithm utilizes the fact that some vertices are more strongly connected than others. The KL algorithm swaps groups of nodes instead of only a pair of nodes between the two partitions. With this method there is a potential of escaping some local optimum.

Another famous improvement to the above local search schema was developed by Johnson et al.[11]. They developed an algorithm for solving the GPP based on simulated annealing (SA). SA is a stochastic optimization algorithm which starts with a bisection and improves upon it using a probabilistic hill-climbing strategy.

Several studies were conducted using the concept of genetic algorithm (GA). In these works there is a population of a large number of different partitionings. New generations are being evolved from a current population by using genetic algorithms techniques. The most recent approach for genetic algorithms is presented in [5]. That paper also summarizes previous genetic algorithm approaches. An approach to solve the GPP with the help of a learning automata was presented in [4]. Another known algorithm is the Extended Local Search algorithm (XLS) presented in [41]. While in most of the local searches a node can
only be swapped once between the two partitions, XLS allows a node to be swapped a couple of times. This way previous bad decisions can be undone and we can escape local optimum more effectively.

All the above algorithms are based on local search and their search space is actually all the different complete partitionings. In that sense, their search space is limited to the leaves of the tree that was presented above. They provide different methods of how to move between these leaves.

A somewhat different and new approach was presented by Rolland and Pirkul in [46]. It is an efficient mathematical programming technique for solving the GPP, based on Lagrangian relaxation and subgradient search. This mathematical programming technique offers an advantage in that it produces a lower bound on the solutions. This bound information is very useful as we are able to evaluate the quality of any given solution at any time.

Our work is much different then all the above since we would like to find an optimal solution to the GPP and not just a sub-optimal solution. In order to find an optimal solution to an NP complete problem a systematic global search must be used. We use such a schema but develop methods to prune a large portion of the search space.

7.3 The different cost functions and search algorithms.

We have already defined the search tree above. In order to search the tree to find an optimal solution, we need an admissible cost function for that. This cost function should be a lower bound on the number of edges that must cross the partition given the partial partition of the node. In the following chapter we propose different methods that suggest such heuristics. They all take different steps in analyzing the unassigned vertices and calculating a lower bound on the number of edges that must cross the partition due to these vertices.

7.3.1 Definitions

We first want to make the following definitions and assumptions:

- We assume that the number of vertices in the graph is n.
- We will use x to denote a vertex in the graph and k to denote a node in the search tree.
- For each node of the search tree we call the vertices that were already assigned to one of the parts of the partition the assigned vertices. Vertices that are have not been assigned yet are called the free vertices.
We denote the two sets of the partitions that are being built during the search as A and B. Vertices in A and B are the assigned vertices.

Within each node of the search tree some of the vertices are free. Each of the suggested heuristic functions below, divides the free vertices also into two sets that will complete the sets A and B of the assigned vertices. We denote the set of vertices that a given heuristic function suggests to add them to A by A’ and the set of vertices that a given heuristic function suggests to add them to B by B’.

For each node of the search tree after applying a heuristic function we can divide the edges of the graph into 4 types:

1. Type I: edges within A and A’ or within B and B’. These edges do not cross the partition and therefore will not be counted by any heuristic.
2. Type II: edges from A to B. These edges are crossing the partition.
3. Type III: edges from a free vertex to an assigned vertex of the opposite set i.e. either edges from A’ to B or from B’ to A.
4. Type IV: edges within two free vertices from opposite sets. i.e. edges from A’ to B’.

Figure 7.2: The sets of the partition and the different edge types.

For each free vertex x we define \(d(x, A)\) and \(d(x, B)\) as the number of edges that are connected between x and vertices from A or B respectively. If we place a free vertex x in A’ then we know that \(d(x, B)\) edges that connect x to vertices in B must cross the partition.

Figure 7.2 Illustrates a node of the search tree. The search tree assigned vertices 1, 4 to A and 2, 3 to B. The heuristic function then completes the partitioning by putting 5, 8 in A’.
and 6, 7 in B’. Therefore the edge (1, 2) is of type I, (1, 4) is of type II, (2, 7) is of type III while edge (6, 8) is of type IV.

The following heuristics try to add edges of the different edges types. As these heuristics become more complicated, they include more types of edges.

7.3.2 \( f_0 \): Uniform cost search.

The first cost function that we use, mainly for comparison reasons, is the cost function that is associated with uniform cost search (UCS). If \( k \) is a node in the search tree let \( g(k) \) be the number of edges that cross the partial partitioning associated with that node. \( g(k) \) is therefore the number of edges between vertices that are assigned to different parts of the partial partitioning or actually edges of type II. Thus we define:

\[
f_0(k) = g(k).
\]

This cost function is usually called uniform cost search since it only evaluates these steps that have already been done for the current node. We call this cost function \( f_0 \) because it does not have any heuristic evaluation for any of the free vertices. \( f_0 \) is very simple to compute but on the other hand is inefficient.

7.3.3 \( f_1 \): Adding edges from free vertices to assigned vertices.

For each free vertex \( x \), we have defined \( d(x, A) \) and \( d(x, B) \) as the number of edges from \( x \) to nodes that are already assigned to A or to B. If \( x \) will be assigned to A then \( d(x, B) \) more edges will cross the partition. If it will be assigned to B then the number will be \( d(x, A) \). An admissible heuristic for \( x \) will therefore be

\[
h_1(x) = \min(d(x, A), d(x, B)).
\]

This number is a safe lower bound on the number of edges that will cross the partition due to \( x \). Summing \( h_1(x) \) for all free vertices will yield an admissible heuristic on the number of edges that must cross the partition. Therefore, for each node \( k \) in the search tree we define:

\[
h_1(k) = \sum_{x \in \text{free}(k)} h_1(x).
\]

The corresponding cost function that we use is thus

\[
f_1(k) = g(k) + h_1(k).
\]

\( h_1 \) assigns each free vertex to either A’ or B’ by looking at potential edges of type III.
The $h_1$ heuristic looks only at each free vertex alone and is not concerned with the interaction between any two free vertices. In that sense, $h_1$ can be associated with the Manhattan distance (MD) in the tile puzzle domain. The MD heuristic, also, counts the number of positions that a tile is away from its goal position and does not look at interactions between two tiles that are not in their goal positions. Thus, $h_1$ can be called the singleton heuristic since it only looks on a single unsolved subgoal at a time.

### 7.3.4 $f_2$: Pairwise costs: Following the additive pattern theory.

The first approach that we tried was to follow the idea of the additive pattern database theory$^3$, regarding costs that are associated with pairs or triples of free vertices. Applying the additive pattern database ideas for a heuristic based on pairs and triples of unassigned vertices is straightforward at this point. Given a partial assignment of some of the vertices to one set or the other, we should consider all possible pairs of the free vertices just as we considered each one of these vertices alone in the $h_1$ heuristic. For each pair of free vertices, there are 4 ways to assign them, and each way may result in a different number of edges being cut. We should take the minimum of those 4 cuts, and that becomes the cost associated with the pair. Actually, we are only interested in those pairs that have an internal edge between them in the input graph. Otherwise, since there is no interaction between them, there is no new knowledge in their pairwise cost. It is just the sum of their singleton distance.

We then want to build the pairwise graph$^4$ where the nodes are the free vertices and the weight of each edge corresponds to the pairwise cost of its nodes. After building this pairwise graph, we would like to add up weights of edges from this graph in order to get a lower bound for a heuristic. We must do that of course in an admissible way. For simplicity (as in the tile puzzle), we only store the addition of the pairwise cost over the singleton heuristic and in order to get the heuristic we have to add the singleton heuristics of all the nodes.

Let $x$ and $y$ be a pair of the free vertices with an internal edge. We should look for the minimum between the following four possibilities:

- $d(x, A) + d(y, A)$: In this case both $x$ and $y$ are being placed in $B$ and the edge between them does not cross the partition.
- $d(x, B) + d(y, B)$: The opposite, both are placed in $A$.

---

$^3$The following heuristic is based on the additive pattern theory and was not implemented and tested because it is outperformed by the $f_2$ heuristic that is presented in the next section. The readers unfamiliar with the additive pattern theory might want to skip this section. We present it here only because it uses exactly the same heuristics as suggested by the Additive database theory.

$^4$Similar to the pairwise graph in the tile puzzle work.
• \(d(x, A) + d(y, B) + 1\): Here, \(x\) is placed in \(B\), \(y\) is placed in \(A\) and the edge between them crosses the partition.

• \(d(x, B) + d(y, A) + 1\): The opposite.

This minimum will be called the pairwise distance of \(x\) and \(y\) and it denotes a lower bound on the number of edges that must cross the partition when looking at the free nodes \(x\) and \(y\) and at interactions between them. In our case, the interaction between them, is the edge between them.

If one of the first two possibilities is smaller than the other possibilities, it means that it is better to place both \(x\) and \(y\) in the same part of the partition. In that case the pairwise cost will be equal to the sum of the singleton costs and the corresponding edge in the pairs graph can be omitted because for simplicity, we only store in the 'pairwise graph' edges whose pairwise cost is greater than the sum of the singleton costs.

If however, one of the last two possibilities is the minimum it means that while looking on each of these vertices alone it would be preferable to placed them in different parts but when looking at both of them together then we have an edge in the pairs graph between \(x\) and \(y\) with a weight of 1. In this case \(x\) and \(y\) should be located in different components but the pairwise cost also adds the edge between them.

In contrary to the tile puzzle, there is no database here, and this heuristic is being computed on the fly. Actually, one can only compute the incremental changes from the parent node and once the pairwise graph for the parent is known, changes in the graph can only occur between neighbors of the new assigned nodes and their neighbors.

After computing all pairwise edges, the pairwise graph is created, and now we need to find a method that will take full benefit of the knowledge of pairwise graph. The additive pattern database theory suggests maximal matching as a general admissible heuristic for the pairwise graph.

While maximal matching is always admissible there might be domain dependent enhancements. For example in the tile puzzle we have seen that taking the vertex cover of the pairwise graph is better than maximal matching. Further enhancements can be found also for the GPP. In fact, because of the domain dependent behavior of the GPP, we observed that there is a much better way to look at interactions between two free vertices than to strictly follow the additive pattern theory. Therefore, we did not implement the conservative pairwise heuristic as suggested by the additive database theory and we present the better heuristic in the next section. We found that is more suitable to change the name of the pairwise graph to the free graph as described in the next section. Nodes of this graph are the free vertices and the edges are edges from the input graph where the two vertices incident with them were assigned to the opposite partition by \(h_i\) i.e. edges of type D. We want to
add to $h_1$ as many edges from the free graph without losing admissibility. We will go into this matter more deeply in the next section.

At this point we stop talking about pairs of vertices from the pairs graph but rather on edges from the free graph. Thus, we abandon the additive database theory at this point. However, the pairwise heuristic is important since the better heuristic in the next section was developed from the pairwise heuristic.

The next step, based on the additive database theory, is to move to triples of vertices of the free graph. However, since conflicts in the GPP domain are associated with edges then when talking about triples, we are only interested in edges between them. These edges were already being scanned by the $f_2$ heuristic below. Thus, unlike the tile puzzle there is no new knowledge in looking into groups of triples such that this knowledge is not already captured by $f_2$.

### 7.3.5 $f_2$: Adding edges from two free vertices to $f_1$.

Let’s look closer at the free vertices. After applying $h_1$ each of these vertices was placed either in $A'$ or in $B'$. Now suppose that $x$ and $y$ are free vertices that were placed by $h_1$ in $A'$ and $B'$ respectively. Suppose that there is also an edge connecting $x$ and $y$. We would like to find out under what conditions we can add this edge to the $h_1$ heuristic without loosing admissibility.

For this purpose we define the free graph. Nodes of the free graph are the free vertices and the edges are edges from the input graph where the two vertices incident with them were assigned to the opposite partition by $h_1$. The edges of the free graph are actually all the edges of type IV. Thus the free graph is actually a bipartite graph where the two sets of the graph are $A'$ and $B'$.

![Figure 7.3: The free graph.](image)

Figure 7.3 illustrates the free graph. The nodes of this graph are nodes 5, 6, 7 and 8. The edges of this graph are the bold edges and connect only edges between free vertices that
were placed in different sets.

We would like to add as many edges to \( h_1 \) from the free graph without losing admissibility. We do this as follows. For each free vertex \( x \), we define:

\[
N(x) = \text{abs}(d(x, A) - d(x, B)).
\]

\( N(x) \) stands for the "number of edges that are allowed for \( x \)". \( N(x) \) is the number of edges of type III that will be added to \( h_1 \) if we move \( x \) from the best component of the partition to the other component. \( N(x) \) is an upper bound on the number of edges that are incident to a vertex of the free graph, \( x \), that might be added to \( h_1 \). When adding a number of edges that does not exceed \( N(x) \) we do not loose admissibility since it is not worse than moving \( x \) to the other component.

For example, let \( x \) be a vertex in the free graph. Let’s assume that all the other free vertices are static and can not move to the other component. Suppose that \( d(x, B) = 2 \) and that \( d(x, A) = 3 \). \( h_1 \) assigns \( x \) to \( A' \) since the number of crossing edges to \( B \) is smaller. Suppose that we have a vertex \( y \) that is assigned to \( B' \) with an edge connecting \( y \) to \( x \) in the free graph. Placing \( x \) in \( A' \) will have \( h_x = d(x, B) = 2 \), but another edge \((x, y)\) will cross the partition. Moving \( x \) to \( B' \) will cancel this edge but \( d(x, A) = 3 \). Either way, from the perspective of \( x \) we have 3 edges crossing the partition. Therefore, this edge, \((x, y)\) can be added to the heuristic value. If however, we further have another vertex in \( B' \), \( z \), with edge \((x, z)\) in the free graph, then adding this edge as well will resolve with \( d(x, B) + 2 = 4 \). In that case \( x \) would choose to move to the other component, \( B' \), and the heuristic will only take \( D(x, A) = 3 \).

If we want an admissible heuristic for \( x \) we can take as many edges that are connected to \( x \) in the free graph as long as the overall heuristic for \( x \) does not exceed the possibility of placing \( x \) in the other component. In other words, for \( x \), we can take no more then \( N(x) \) edges (of type IV). As long as we do not take more than \( N(x) \) for \( x \) then from the perspective of \( x \) (assuming that the rest of the graph is stable) there is no reason to transfer \( x \) to the other component because transferring it will surly add \( N(x) \) more edges of type III. The graph is of course not stable and there are other free vertices as well. Thus we have to calculate \( N(x) \) for all of the nodes of the free graph and deal with interactions between vertices.

We can now draw the free graph and for each node \( x \) we write \( N(x) \) next to it as shown in Figure 7.4. Each node is associated with a number of edges of type IV that are connected to that node and might be added to the heuristic. Because if we move this vertex to the other component we must add a number of edges of type III that are uniquely connected to this vertex from assigned vertices and are not related whatsoever to the free graph. This number is at least as large as the number of edges of type IV that we took.

Therefore, in order to get an admissible heuristic we should solve the following problem.
We want to take as many edges as possible from the free graph and form a subgraph such that no node \( x \) will be connected in the subgraph to more than \( N(x) \) neighbors. This problem is a generalization of the maximal matching problem, since the simple matching problem is a special case where \( N(x) = 1 \) for all the vertices.

Figure 7.4 illustrates the generalized matching problem of the free graph. Each node \( x \) is associated with a number which is its \( N(x) \). We want to take as many edges such that \( N(x) \) is an upper bound on the number of edges that are connected to a node \( x \). We want to take as many edges from the free graph as long as the number of edges that are connected to a node \( x \), does not exceed \( N(x) \).

This generalized matching problem can be solved very easily as a flow problem [40]. We connect the source node \( s \) to each of the nodes of \( A' \), with each edge that connects \( s \) to \( x \) having a capacity of \( N(x) \). Each of the edges of the bipartite free graph (edges from \( A' \) to \( B' \)) will have a capacity of one. Then, we connect each of the free nodes from \( B' \) to the target node \( t \). The capacity of an edge from \( y \) in \( B' \) to \( t \) will be \( N(y) \). Now we are looking for a maximal flow from \( S \) to \( t \). The size of this flow is the maximum of the generalized matching which can be added to the \( h_1 \) heuristic without losing admissibility. If \( V \) is the number of the free vertices and \( E \) is the number of edges in the free graph then this is done in \( O(E \times V) \).

To summarize we define:

\[
f_2(k) = g(k) + h_1(k) + h_2(k)
\]

where \( h_2(n) \) is the above generalized matching of the free graph. \( f_2 \) is therefore calculated in the following steps:

1. calculate \( h_1 \) for each of the free vertices and construct the free graph i.e. the sets \( A' \) and \( B' \) and edges between them.

2. for each vertex \( x \) in the free graph calculate \( N(x) \).
3. calculate $h_2$ by solving the generalized matching problem for the free graph such that no node $x$ will be connected to more than $N(x)$ neighbors.

We also suggest a brute force technique for solving the generalized matching problem. This technique looks at an edge, $(x, y)$ and if both $N(x) > 0$ and $N(y) > 0$ then it adds this edge to the overall heuristic, and decreases both $N(x)$ and $N(y)$ by one. This is done in $O(E)$. However, this method is not the optimal method of solving this generalized matching problem and there might be a larger set of edges which satisfied the constraints. For the special case of regular matching, when having $N(x) = 1$ for all nodes, this brute force technique means that we simply take the first assignment and do not look at augmenting paths. In fact, in the experiment that we will present later we used this brute force technique since it was more cost effective.

7.3.6 $f_3$: Sorting the free vertices

Even though $f_1$ is associated with singleton heuristic and $f_2$ in some way reminds the pairwise heuristic, it seems that we can take these heuristic much further, due to a domain specific behavior of the GPP. Thus, we now present two other heuristics $h_3$ and $h_4$ that are somewhat similar to $h_1$ and $h_2$ but these new heuristics also take care of balancing the size of each part of the partition.

Let $G$ be a graph with $V$ vertices. Assume that $G$ already has a partial partition into sets $A$ and $B$, with the cardinalities of $A$ and $B$ being $V_A$ and $V_B$, respectively. When the partition will be completed, the $V - V_A - V_B$ unassigned vertices should also be partitioned into $A$ and $B$. Since in a balanced partition there must be $V/2$ vertices in each set then we know that $V/2 - V_A$ free vertices must go into set $A$, and $V/2 - V_B$ other vertices must go into set $B$. Thus, we have to partition the $V - V_A - V_B$ free vertices into two sets with cardinalities $V/2 - V_A$ and $V/2 - V_B$. With $f_1$, we do not look into this but rather place each free vertex in one of the partitions without looking at where other free vertices went. Thus, for $f_1$, an unbalanced partial partition that has most of the vertices on one side will tend to have a low heuristic, since assigning most free vertices to the set with most of the assigned vertices will generate few crossing edges.

We solve this problem as follows. For each vertex $x$ from the $n$-a-b free vertices, we define:

$$N_A(x) = d(x, A) - d(x, B)$$
$$N_B(x) = d(x, B) - d(x, A)$$

$N_A(x)$ denotes the advantage of placing $x$ in $A$ over placing it in $B$. If, for example, $N_A(x) = 3$, it means that it is preferable in 3 edges if we place $x$ in $A$ than if we place it in $B$. It is
straightforward that $N_B(x) = -N_A(x)$. Note that $N(x)$ from $f_2$ is actually the absolute value of $N_A(x)$ and $N_B(x)$.

Now, we sort the $V - V_A - V_B$ unassigned vertices in decreasing order of $N_A(x)$. We take the first $V/2 - V_A$ vertices and place them in A (we call these nodes $A'$). The rest of the vertices (which are the $V/2 - V_B$ vertices that have the best $N_B(x)$ value) will be placed in B (we call them $B'$). This is an admissible way to partition these vertices to A and B and then have both partitions with $V/2$ vertices. After constructing $A'$ and $B'$ we define:

$$h_3(x) = \begin{cases} \quad d(x, B) & \text{if } x \in A' \\ \quad d(x, A) & \text{if } x \in B' \end{cases}$$

Summing $h_3(x)$ for all $x$ in the free graph will yield an admissible heuristic which is much better than $h_1$ since in $h_1$ we took the minimum between $d(x, A)$ and $d(x, B)$ while in $h_3$ we sometimes take the maximum between them.

For example, suppose we are left with 6 unassigned vertices as illustrated in Table 7.1. Suppose that we need to take 4 nodes to $A'$ and 2 to $B'$. The four nodes with the best $N_A(x)$ are a, b, c, and d (They are $A'$). The total $h_3$ heuristic for these nodes will be the sum of their $d(x, B)$ which is $1 + 1 + 1 + 2 = 5$. Nodes e and f will go to B ($B'$) with a heuristic of $2 + 1 = 3$. So the $h_3$ heuristic here is $3 + 5 = 8$. Note that node d was placed in $A'$ even though from its own perspective it would rather be placed in $B'$. Thus $h_1$ would assign node d to $B'$ and take $d(d, A)$ as its heuristic while $h_3$ would assign it to $A'$ and take $d(d, B)$. $h_3$ is still admissible but in this case is larger than $h_1$. While $f_1$ divided the free vertices into two sets with arbitrary sizes $f_3$ divides them so that the resulting sets would be balanced.

Thus we define:

$$f_3(k) = g(k) + h_3(k)$$

where $h_3(k)$ is calculated by sorting the free vertices by decreasing order of $N_A(x)$, splitting the sorted list into $A'$ and $B'$ and then summing the number of edges (of type III) that cross
the partition.

The sorted list can be calculated once for the root node of the search tree and only incremental changes are done when traversing the tree.

7.3.7 $f_4$: Adding edges from two free vertices to $f_3$

Just like $h_1$, $h_3$ only looks on edges from free vertices to assigned vertices (type III). The next step would be similar to $h_2$ i.e. look on edges from two free vertices and add them to $h_3$. Once again we build the free graph which has edges from $A'$ to $B'$. Now again, in the same manner as $h_2$ the question is how many edges from the free graph (edges of type IV) can be added to $h_3$ in an admissible way. In $h_2$, if $x$ was in $A'$ we defined

$$N(x) = d(x, B) - d(x, A).$$

As long as we add a number of edges of $x$ that is less or equal to $N(x)$ it will not be worthwhile to transfer $x$ to $B$. Here also, we can add edges to the heuristic of $x$ as long as we know that it is not worthwhile to move $x$ to $B'$. However, we want both parts of the partition to be balanced and have exactly $N/2$ nodes each. Thus, if we want to move node $x$ from $A'$ to $B'$ then some other node $y$ from $B'$ must move to $A'$. Therefore, it is not enough to ask how many edges of type III will be added if $x$ moves to $B'$. We should find another node from $B'$, $y$, and swap it with $x$ in order to keep the cardinality of both parts of the partition balanced. We should also count the number of edges of type III that will be added because of the fact that $y$ moves from $B'$ to $A'$.

The best candidate for this would be the node from $B'$ with the best (smallest) $N_A$, (or the worst $N_B$), because it will add the minimum number of edges of type III if it moves from $B'$ to $A'$. Since for $h_3$, we already have all the vertices of the free graph sorted by their $N_A$ value it is very simple to spot a vertex in $B'$ with the smallest $N_A$ value. We call this node the swappable vertex and denote it $s_{B'}$ Once this $s_{B'}$ is spotted, the number of edges from the free graph that are allowed to $x$ is $N_A(x) - N_A(s_{B'})$. This is because if $x$ will move from $A'$ to $B'$ then $N_A(x)$ will be added to the cut because of $x$. However, vertex $y$ will move from $B'$ to $A'$ so $N_B(y) = -N_A(s_{B'})$ edges should also be added. In a symmetric manner we spot a swappable node $s_{A'}$ in $A'$ with the best (smallest) $N_B$. We assume that each vertex $x$ in $B'$ will be at least swapped with $s_{A'}$. Thus, the number of edges that is allowed for a node $x$ in $B'$ is $N_B(x) - N_B(s_{A'})$. Therefore, after these swappable vertices from $B'$ and $A'$, $s_{B'}$ and $s_{A'}$, are spotted we define:

$$\tilde{N}(x) = \begin{cases} 
N_A(x) - N_A(s_{B'}) & \text{if } x \in A' \\
N_A(s_{A'}) - N_A(x) & \text{if } x \in B' 
\end{cases}$$
Where $s_{A'}$ is a swappable vertex from $A'$ and $s_{B'}$ is a swappable vertex from $B'$. $\bar{N}(x)$ is the number of edges that are allowed to x if we want the two parts of the partition to remain balanced. Let’s take a look at the example given in Table 7.1 The swappable vertex in $B'$ is (the node with the best $N_A$ value) vertex $e$ with $N_A(e) = -2$. Now, consider vertex $a$. $\bar{N}(a) = N_A(a) - N_A(e) = 2 - (-2) = 4$. As long as the number of edges from the free graph that we add to vertex $a$ is not greater than 4 we do not lose admissibility. This is because only if it is greater than 4 it might be worthwhile to exchange $a$ and $e$. The swappable vertex in $A'$ is node $d$ with $N_B(d) = 1$. If we take $f$, for example, then $\bar{N}(f) = N_B(f) - N_B(d) = 3 - 1 = 2$. Only two edges are allowed for $f$. Otherwise we might want to swap $f$ and $d$. Note that for nodes in $A'$ we now allow more edges than $N(x)$ of $f_2$. This is because the best $N_A$ vertex in $B'$ is negative. On the other hand, for vertices in $B'$ we now allow less edges since the best $N_B$ vertex in $A'$ is positive.

Just as in $f_2$ we now want to take as many edges from the free graph as long as for each node $x$ we do not take more than $\bar{N}(x)$ edges. We have the same generalized matching problem.

The general rule for $F_4$ will therefore be

$$f_4(k) = g(k) + h_3(k) + h_4(k)$$

where the calculation of $h_4(k)$ is done as follows:

1. Calculate $N_A$ for each of the free vertices.
2. Sort the free vertices in a decreasing order of $N_A$.
3. Construct the free graph i.e. the sets $A'$ and $B'$ and edges between them.
4. Calculate $h_3$ for each of the free vertices.
5. For each vertex $x$ in the free graph calculate $\bar{N}(x)$.
6. Calculate $h_4$ by solving the generalized matching problem for the free graph such that no vertex $x$ will be connected to more than $\bar{N}(x)$ neighbors.

After determining the correct $\bar{N}(x)$ values for all the vertices in the free graph we have to find a generalized maximum matching for that graph. Here also we found that our brute force technique (that was described earlier) is more cost effective than finding the optimal matching.

---

5The bar is to indicate that the two components of the partition are balanced.
6Note that the two swappable nodes are next to each other in the sorted list.
There is a small restriction to the above calculations. If, for example, vertex \( x \) from \( A' \) was allowed 4 edges, then either adding 4 edges to \( x \) or moving \( x \) to \( B' \) by swapping it with \( y \) (a swappable vertex from \( B' \)) will yield the same number. If indeed we find 4 edges and add them up to \( h_4(x) \) then it means that \( x \) and \( y \) can either stay as they are or be swapped. In that case we must change \( \bar{N}(y) \) to 0 since we are no longer certain that it is in \( B' \). Maybe it was swapped with \( x \). In the same manner, if we have several vertices from \( B' \) with the same \( \bar{N} \) values as \( y \) they are all fit to be the swappable vertex and once we find a vertex \( x \) from \( A' \) that we took all the allowed number of edges for it, then all these swappable nodes are suspected to be swapped and their \( \bar{N} \) value is changed to 0. Edges that were taken for the swappable nodes must be subtracted from the matching in this case. In practice this case occurs very frequently and therefore, we found that it is more convenient and cost effective to set \( \bar{N} \) values of all the swappable nodes from both \( A' \) and \( B' \) to 0 in advance.

### 7.4 Enhancements to the search process

Now that we have defined the search tree and the heuristic functions we should described the search algorithms that we used. Once again we used IDA* [26] and DFBBnB [34]. A comprehensive description of these algorithm was presented in section 6.5.3. However, in addition the above heuristics we also implemented the following 3 enhancements into the search.

#### 7.4.1 Ordering the vertices of the graph

Each vertex of the graph is associated with a level of the tree and at that level it is assigned to one of the sets of the partial partitions. When sorting the graph in decreasing order of their degree (the number of neighbors for each vertex), vertices with many neighbors will be treated sooner. Since these vertices have more edges connected to them they add more edges that cross the partial solution and the heuristic function in this case should also be larger and more accurate. Therefore, we first sort the graph in decreasing order of degree and add vertices in that order rather than a random order. We have found that this simple enhancement may speed the search process by a factor of more than 10 for many cases. This improvement was found to be valid for both IDA* and DFBBnB and for all the versions of the heuristic functions that we used. This is because a better search tree is formed and it effects every search of that tree. In fact, this ordering might be useful for other graph problems since it deals with more important vertices sooner. In all the experiment below we used this ordering of vertices for the search tree.

Another idea is to sort the vertices of the input graph on a breadth-first order. A breadth-
first order sorts the vertices of the graph by the order that the breadth-first search visits them. In that case, clusters of connected vertices will be treated together which might improve the search process. We tried this method but the above order by degree was much better.

7.4.2 Calculating related heuristics

Another improvement is more general and concerns the heuristic functions. This improvement can be applied whenever a more complicated heuristic is based on first calculating a simpler heuristic. With both IDA* and DFBnB, since we are searching trees, most of the nodes generated are at the bottom and get pruned during a given iteration. Instead of computing the complicated heuristic and then checking to see if a node should be expanded or pruned we should first check if the simpler heuristic already exceeds the threshold for pruning. If it does, we can prune the node without computing any more heuristics, which is the expensive part of the search. In our case of the GPP heuristics, $h_2$ is an addition to $h_1$ and $h_4$ is an addition to $h_3$. When using $h_4$, for example, we first calculate $h_3$. If for a given node, $h_3$ already exceeds the threshold, there is no need to calculate $h_4$ for that node. While applying this idea does not decrease the number of generated nodes it does decrease the constant time per node for many nodes. Applying this enhancement to the GPP search reduced the average constant time per node by more than 20% for many cases.

7.4.3 Ordering nodes of the search tree

A known enhancement to DFBnB is to order the descendents of a node by increasing order of their cost value and to visit them in this order. This method helps the search to first explore a subtree whose root seems to be more promising than its siblings. Nevertheless, this ordering of the nodes need not to be according to the same cost function that the search uses. We can take any heuristic to order these nodes. In fact, we have observed that the most cost effective heuristic that we obtained was using $h_4$ for the heuristic of a node but ordering the descendents of a node with $h_1$. The reason for using a more simple heuristic for the ordering of the nodes is that we want a very fast method for ordering these nodes since sometimes some of these nodes will not be visited by the search process. Thus, in all the experiments below we used $h_1$ to order the descendents of a node in the search tree for all the search algorithms except for $f_0$. 

128
7.5 Empirical results

The testbed for our algorithms were random graphs. Just as in the vertex cover work we had two variables that were associated with each graph:

- The number of vertices in the graph $n$.
- The average degree or branching factor $b$.

The graph was built as follows. First we determined the values of these two variables. We then built an empty graph with $n$ vertices. Then, each edge is independently added to the graph with probability of $b/n$. Therefore the graph that we obtained has an average degree of $b$ but with some variance. Note that since the branching factor of the resulting graph is not uniform then before starting the search process we first sort the vertices of the graph by decreasing order of their degrees. We did not use Delauney graphs here because they seem to have a special structure that might affect the heuristics of the vertex cover search. However, here, we do not believe that Delaunay graphs will behave differently then random graphs. All the experiments below were conducted on a 500MHZ pentium III PC.

7.5.1 Graphs with 50 vertices

The first set of experiments that we conducted were for graphs with 50 vertices. We used degrees of 2, 4 .. 20. For each type of graph we generated 30 random graphs and all the data below are the average of these 30 instances.

In many previous works in heuristic search, when comparing two algorithms the comparison is done on the number of generated nodes [27] [34] [10] [29]. We believe that presenting only the number of generated nodes for each algorithm here would be inappropriate, since the constant time per node is quite different for each algorithm. When the heuristic function is more complicated the number of generated nodes decreases at the expense of a larger constant time per node. Hopefully this should resolve with a better overall time for solving a given problem. Many times, papers on heuristic search only report about the improvement in the number of generated nodes while the improvement in real time is extremely smaller [10]. Therefore, we believe that the comparison should be between the overall time that an algorithm solves a given problem.

Table 7.2 shows the number of nodes per second for each of the algorithms on graphs of size 50 with average degree of 8. We refer to the average degree of the graph as the density of the graph$^7$.

---

$^7$The reason for choosing 8 here and later on, is because 8 is the smallest density that DFBnB was faster than IDA* for all the heuristics and all sizes of the graph.
Table 7.2: The number of nodes per second for graphs of size 50 with an average density of 8 for the different heuristics

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Nodesspersecond</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_0$</td>
<td>2,412,132</td>
</tr>
<tr>
<td>$f_1$</td>
<td>1,280,940</td>
</tr>
<tr>
<td>$f_2$</td>
<td>435,270</td>
</tr>
<tr>
<td>$f_3$</td>
<td>445,553</td>
</tr>
<tr>
<td>$f_4$</td>
<td>194,191</td>
</tr>
</tbody>
</table>

Figure 7.5: The optimal partition for graphs of size 50 with different densities.

As could be expected, the table shows that as the heuristic function is more complicated the constant time per node increases. We observed that for a given evaluation function the constant time per node does not change significantly for all sizes and densities of graphs. As shown below, with better heuristics the number of generated nodes decreases yielding a better overall time needed for solving a given problem.

As a general rule we have observed that as the size of the graphs grows larger and also as the graph becomes more dense the number of generated nodes for each algorithm increases. However, the ratios between the different algorithms of the constant time per node generation tends to remain about the same with only minor changes.

Figure 7.5 illustrates the average size of the optimal partition for graphs of size 50 with different degrees. The size of the optimal solution tends to increase linearly when increasing the average degree.

Figure 7.6 shows the overall time of the different algorithms on graphs with 50 vertices. Each curve corresponds to a different algorithm where each point is an average of 30 different graphs of that size and degree. The search for this figure was performed by DFBnB. The
Figure 7.6: The overall time for the algorithms on graphs with 50 vertices. DFBnB used.

<table>
<thead>
<tr>
<th>Density</th>
<th>algorithm</th>
<th>optimal cut</th>
<th>nodes</th>
<th>seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>$f_0$</td>
<td>36.90</td>
<td>53,161,224.20</td>
<td>1,836.04</td>
</tr>
<tr>
<td>6</td>
<td>$f_1$</td>
<td>36.90</td>
<td>18,923,257.97</td>
<td>11.76</td>
</tr>
<tr>
<td>6</td>
<td>$f_2$</td>
<td>36.90</td>
<td>13,090,520</td>
<td>36.57</td>
</tr>
<tr>
<td>6</td>
<td>$f_3$</td>
<td>36.90</td>
<td>655,027.73</td>
<td>1.47</td>
</tr>
<tr>
<td>6</td>
<td>$f_4$</td>
<td>36.90</td>
<td>44,404.33</td>
<td>0.22</td>
</tr>
<tr>
<td>20</td>
<td>$f_1$</td>
<td>184.80</td>
<td>13,664,811,427.40</td>
<td>20,590.75</td>
</tr>
<tr>
<td>20</td>
<td>$f_3$</td>
<td>184.80</td>
<td>342,200,788.97</td>
<td>1,139.51</td>
</tr>
<tr>
<td>20</td>
<td>$f_4$</td>
<td>184.80</td>
<td>33,850,497.73</td>
<td>269.16</td>
</tr>
</tbody>
</table>

Table 7.3: Nodes visited and time elapsed for each of the algorithms.

Figure clearly shows that using a more complex heuristic function resolves with a much better overall time. The improvement factor between two succeeding algorithms is around 10. We can also observe that in general, the overall time increases exponentially when increasing the dense of the graph.

Table 7.3 focuses on two points of this data, namely on graphs with an average degree of 6 and 20. We can see that the difference between the two extreme cases, $f_0$ and $f_4$, for graphs with a degree of 6 is almost 5 orders of magnitude in the number of generated nodes and 4 orders for the overall time. While a simple search based on $f_0$ needs a complete hour to solve such a problem, a complicated search heuristic based on $f_4$ solves it in less than one half of a second. With a more dense graph this difference becomes even larger and if we compare $f_1$ to $f_4$ we see that for a density of 6, $f_4$ outperforms $f_1$ by a factor of 50 in the
Figure 7.7: The overall time for the algorithms on graphs with 50 vertices. IDA* used.

Figure 7.8: Comparing IDA* to DFBnB.

overall time while for a density of 20 the improvement factor is 100.

Note that $f_2$ does not improve the overall time over $f_1$. While it generates a smaller number of nodes, the constant time per node is much larger and does not resolve with a better overall time. Therefore, we did not include $f_2$ in the figures.

Figure 7.7 shows data for a search that was performed by IDA*. Once again, it is on graphs of size 50 and the density of the graphs was variable. Figure 7.8 compares both DFBnB and IDA* for $f_0$, $f_1$ and $f_4$. These figures show that at a density of 4 and smaller, IDA* outperforms DFBnB while for larger degrees DFBnB is much better. The explanation for this phenomenon is as follows. With 50 vertices in the graph, the search tree is fixed with $O(2^{50})$ nodes. Each node is given a heuristic value of the best cut associated with it. With larger degrees, the optimal cut is larger and therefore the number of unique values in the search tree is larger. This number directly affects the number of iterations performed by
IDA*. With more different unique values there are more iterations of IDA* and many nodes are generated many times. This phenomenon does not seems to effect DFBnB in the same manner and therefore in graphs that are more dense DFBnB outperforms IDA*.

For example, for a graph with a density of 2 and an optimal cut of 4, IDA* using $f_4$ has 5 iterations with thresholds increasing from 0 to 4. On the other hand, DFBnB on the same graph first found a solution of 31 and then improved it until it found a solution of 4. If we can associate each improvement with an iteration, then DFBnB had 27 iterations for that graph. However, for a graph with density of 8 and an optimal cut of 51, IDA* performed 52 iterations while DFBnB reduced its solutions from 91 to 51, with only 41 iterations.

### 7.5.2 Graphs with a degree of 8.

We conducted another set of experiments. In this set the average degree of the graphs was always set to 8 but the size of the graph varied from 20 to 80. The search was done with DFBnB. Again, each number in the following data is an average over 30 different graphs with the same size and average degree.

Table 7.4 shows data for that set of experiments. The table shows that the optimal cut grows linearly with the size of the graph while the overall time grows exponentially. We tried only 3 of our heuristics, namely $f_1$, $f_3$ and $f_4$ and only with DFBnB. We can see that as the graphs grow larger the gap between these algorithms also grows. While for graphs of size 20, $f_4$ is twice as fast as $f_1$, for graphs of size 80 the improvement factor is almost 1000.

Figures 7.9 and 7.10 illustrate the overall time and the number of generated nodes for these two algorithms. The figures show that for graphs of size 80 the improvement factor in the number of nodes generated of $f_4$ over $f_1$ is almost 100,000, five orders of magnitude.

<table>
<thead>
<tr>
<th>Size</th>
<th>Cut</th>
<th>$f_1$:seconds</th>
<th>$f_3$:seconds</th>
<th>$f_4$:seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>27.07</td>
<td>0.04</td>
<td>0.02</td>
<td>0.01</td>
</tr>
<tr>
<td>30</td>
<td>34.67</td>
<td>0.15</td>
<td>0.04</td>
<td>0.03</td>
</tr>
<tr>
<td>40</td>
<td>47.07</td>
<td>3.89</td>
<td>0.59</td>
<td>0.15</td>
</tr>
<tr>
<td>50</td>
<td>54.43</td>
<td>80.88</td>
<td>7.53</td>
<td>1.06</td>
</tr>
<tr>
<td>60</td>
<td>63.13</td>
<td>1,482.14</td>
<td>59.70</td>
<td>4.87</td>
</tr>
<tr>
<td>70</td>
<td>69.87</td>
<td>30,922.20</td>
<td>1,085.62</td>
<td>55.10</td>
</tr>
<tr>
<td>80</td>
<td>83.59</td>
<td>502,238.58</td>
<td>4,741.89</td>
<td>119.66</td>
</tr>
</tbody>
</table>

Table 7.4: Graph with density of 8.
Another domain studied was graphs of size 100. Here we only solved five graphs for each degree and only with our best algorithm $f_4$. The results are shown in Table 7.5. For these graphs we can also see that as the density of the graph increases linearly, the time needs to solve the problem increases exponentially. We solved the most difficult graph that we had i.e. with a density of 10 in 227,607 seconds.

7.6 Discussion and conclusions for the GPP work

The primary purpose of our approach is to find optimal solutions and therefore we perform a global search on all the possible different solution. However, the DFBnB search can also be
used as an anytime algorithm and return the best solution found so far. In fact, we observed that many times, over 70% of the running time was spent after the optimal solution was already reached, in order to assure that a better solution does not exist. So actually after 30% we already have an optimal solution in hand but we do not know this until the search halts.

Our approach cannot be directly compared to other GPP algorithms since they are designed to return suboptimal solutions and usually perform local searches. However, we can combine our global search with any other suboptimal algorithm as follows. We first run the suboptimal algorithm and get suboptimal solution from it. We can then run our DFBnB search with that solution as the initial threshold in order to either improve that solution or verify that it is an optimal solution.

To the best of our knowledge, we are the first to develop an algorithm for optimal solutions for the GPP. However a very interesting approach would be a combination of our algorithm and the work in [46] with the Lagrangian relaxation. While their algorithm does not guarantee a feasible optimal solution, it provides the size of the optimal solution rather quickly. Once we know the size of the optimal solution we can run our algorithm with that size as a bound and stop as soon as the first solution is found. In this search both IDA* and DFBnB converge and only develop nodes whose cost value is smaller or equal to the size of the expected optimal solution. No node will be visited more than once, and no node with a cost that is larger than the optimal solution will be visited. This can save a great amount of time.

In conclusion, we have shown a number of admissible heuristics that optimally solve the graph partitioning problem. This work once again shows that finding better heuristics by looking deeper into interactions between unsolved subgoals seems to pay off at the end with a much better overall time for obtaining a solution. The heuristics that we used here on the GPP do not exactly follow the additive database theory. However, they use the same principle, namely that looking deeper into interactions between unsolved subgoals yields a better heuristic and thus improves the search process.

<table>
<thead>
<tr>
<th>Dense</th>
<th>Cut</th>
<th>( f_4 ):Nodes</th>
<th>( f_4 ):seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5.57</td>
<td>170,103.70</td>
<td>0.82</td>
</tr>
<tr>
<td>4</td>
<td>30.97</td>
<td>5,333,677.50</td>
<td>51.85</td>
</tr>
<tr>
<td>6</td>
<td>66.63</td>
<td>122,199,646.13</td>
<td>1,542.94</td>
</tr>
<tr>
<td>8</td>
<td>106.30</td>
<td>2,004,165,640.23</td>
<td>29,214.19</td>
</tr>
<tr>
<td>10</td>
<td>144.75</td>
<td>13,464,048,386.75</td>
<td>227,607.09</td>
</tr>
</tbody>
</table>

Table 7.5: Graph with size 100.
Chapter 8

Summary and conclusions

We have presented in this thesis two kinds of approaches of dealing with heuristic search. The first one is more traditional and deals with improving search techniques. We have presented the KBFS algorithm and the Near-Optimal Perimeter Search. We have shown that both algorithms are simple and yet very powerful in finding solutions faster. Both algorithms are general and can be applied to many domains.

The other part of our work was a little more domain specific. In that part we tried to find a general schema for finding better heuristics. With better heuristics, the number of generated nodes is reduced by a great deal and we are able to find a solution much faster. We have presented the additive database theory for finding better heuristics and implemented it on both the tile puzzles domain and the vertex-cover problem. We obtained very good results on both domains. We have also used the idea behind the additive database theory to optimally solve the graph partitioning problem. While we did not strictly follow the ADB theory, we used its principle that looking into interactions between unsolved subgoals can improve the heuristic function. So in fact, we have shown three domains where this general idea of looking into interactions between subgoals and using this information can resolve with a much more accurate heuristic function and thus a solution can be found much faster. In all our experiments, the improvement factor of our best heuristics over the simple heuristic was a couple of orders of magnitude.

Many search algorithms were developed over the years but since IDA* and DF BnB were presented, none of the other algorithms can really compete with the simplicity and effectiveness of these algorithms. We believe that works for finding solutions faster should concentrate on domain specific enhancements. We believe that given a specific problem, finding a better heuristic is the best way to improve the search and that this should be the direction for future works. We have developed a general method for improving heuristics. The importance of our approach is the generality of it and the fact that it can be applied to many domains.
We believe that future work can try to implement our approach to other domains and try to improve this approach by generalizing it or finding new ways to combine knowledge from different subproblems.
Bibliography


140


