Finding Patterns in an Unknown Graph

Roni Stern [a] Meir Kalech [a] and Ariel Felner [a]
[a] *Department of Information System Engineering*
Ben Gurion University of the Negev
Beer-Sheva, 84105, Israel
E-mails: roni.stern@gmail.com, kalech@bgu.ac.il, felner@bgu.ac.il

AbstractSolving a problem in an *unknown graph* requires an agent to iteratively explore parts of the searched graph. Exploring an unknown graph can be very costly, for example, when the exploration requires activating a physical sensor or performing network I/O. In this paper we address the problem of searching for a given input pattern in an unknown graph, while minimizing the number of required exploration actions. This problem is analyzed theoretically. Then, algorithms that choose which part of the environment to explore next are presented. Among these are adaptations of existing algorithms for finding cliques in a known graph as well as a novel heuristic algorithm ($Pattern^*$). Additionally, we investigate how probabilistic knowledge of the existence of edges can be used to further minimize the required exploration. With this additional knowledge we propose a Markov Decision Problem (MDP) formulation and a Monte-Carlo based algorithm ($RPattern^*$) which greatly reduces the total exploration cost. As a case study, we demonstrate how the different heuristic algorithms can be implemented for the *k-clique* pattern as well as for the *complete bipartite* pattern. Experimental results are provided that demonstrate the strengths and weaknesses of the proposed approaches on random and scale-free graphs as well as on an online web crawler application searching in Google Scholar. In all the experimental settings we have tried, the proposed heuristic algorithms were able to find the searched pattern with substantially less exploration cost than random exploration.

Keywords: Heuristic search, Unknown graphs, Subgraph isomorphism

1. Introduction

Many real-life problems can be modeled as problems on graphs, where one needs to find subgraphs that have a specific structure or attributes. Examples of such subgraph structures are shortest paths, any path, shortest traveling salesperson tours and cliques. Most classical algorithms that solve such graph problems assume that the structure of the graph is given either explicitly, in data structures such as adjacency list or adjacency matrix, or implicitly, with a start state and a set of computable operators (e.g., moving a tile in a sliding tile puzzle state). The complexity of such search algorithms is therefore measured with respect to CPU time and memory demands. We refer to such problems as problems on *known graphs*.

By contrast, there are domains that can be modeled as graphs, where the graph structure is not known a-priori, and exploring vertices and edges requires a different type of resource, that is, neither CPU nor memory. For example, a robot navigating in an unknown terrain, where vertices and edges correspond to physical locations and roads, respectively. Acquiring knowledge about the vertices and edges of the searched graph may require activating a physical sensor and possibly mobilizing the robot, incurring a cost of fuel (or any other energy resource). Another example is an agent searching the World Wide Web, where the web sites and hypertext links represent the vertices and edges of the searched graph, respectively. Since the web is extremely large and dynamic, accessing vertices requires sending and receiving network packets (e.g., HTTP request/response). We refer to such problems as problems on *unknown graphs*.

Solving problems in *unknown graphs* (as well as in any other type of graph problem) requires exploring some parts of the graph. We define an exploration action for a vertex as an action that discovers all its outgoing edges and neighboring vertices. Such exploration actions are associated with a cost, denoted hereafter as *exploration cost*. This exploration cost is often conceptually different than the traditional computational effort (of CPU and memory). In the web graph domain, for example, the exploration can correspond to sending an HTTP request, retrieving an HTML page and parsing all the hypertext links in it. The hypertext links are the outgoing edges, and the connected web sites are the neighboring vertices. The associated exploration cost includes the network I/O of sending and receiving IP packets. For a physical domain, where a robot is navigating in an unknown terrain, the exploration is done by using sensors at a location to discover the near area,

AI Communications
ISSN 0921-7126, IOS Press.

e.g., the outgoing edges and the neighboring vertices in the map. The associated exploration cost includes the cost of activating sensors at a vertex. In both cases the CPU and memory costs are often negligible in comparison to the other exploration costs. An important task, which is addressed in this paper, is to solve the problem while minimizing the exploration cost. This is especially important when computational CPU cost is of lesser importance and can be neglected as long as it is running in time that is tractable.

In this paper, we address the problem of searching for a specific pattern of vertices and edges in an unknown graph while aiming to minimize the exploration cost. Starting from a single known vertex, the search is performed in a best-first search manner. In every step, if the desired pattern does not exist in the known part of the graph a single "best" vertex is chosen and explored. This process is repeated until the desired pattern is found or until the entire unknown graph is explored. Several general heuristic algorithms are proposed for choosing which vertex to explore next: $KnownDegree$, $Pattern^*$ and $RPattern^*$. $KnownDegree$ is a straightforward adaptation of a common known graph heuristic, in which the vertex with the highest degree is explored first. $Pattern^*$ exploits the structure of the searched pattern by choosing to explore the vertex that is "closest" to being a part of the searched pattern. A metric for "closeness" of a vertex to a pattern is presented. With this "closeness" metric, $Pattern^*$ has the property of returning a tight lower bound on the number of exploration steps required to find the searched pattern. For scenarios where probabilistic knowledge of the unknown graph is available, we propose the $RPattern^*$ heuristic algorithm. $RPattern^*$ is a randomized heuristic algorithm that chooses the next vertex to explore by applying a Monte-Carlo sampling procedure in combination with $Pattern^*$ as a default heuristic.

To demonstrate the applicability of the proposed heuristic algorithms, we describe how to implement them for two specific patterns: a k-clique and a complete (p-q)-bipartite graph. We develop the concept of a *potential* k-clique and a *potential* complete (p-q)-bipartite graph, along with supporting corollaries that allow efficient implementation of the $Pattern^*$ heuristic algorithm for these patterns. Empirical evaluation were performed on the k-clique pattern, by applying the proposed heuristic algorithms when searching random and scale-free graphs. Results show that the performance of $Clique^*$ and $RClique^*$ (the k-clique variants of $Pattern^*$and $RPattern^*$respectively), in terms of exploration cost, is equal to and often much better than an adaptation of the state-of-the-art clique search algorithm. The strengths and weaknesses of the different heuristic algorithms are evaluated. We also implemented and evaluated the algorithms on a web crawler application, where the papers that are accessible via Google Scholar are the vertices of the searched unknown graph. Results show that using $Clique^*$, cliques are found more often and with less exploration cost compared to $KnownDegree$ and random exploration.

Beyond the value of investigating such a basic problem in the unknown graph setting, finding patterns in an unknown graph has practical applications in real-world domains. For example, finding a set of physical locations forming a clique suggests the existence of a metropolitan area. Another example is a set of scientific papers, where finding a set of papers that reference each other suggest resemblance in content. Therefore finding such a cluster of referencing papers can be useful in a data mining context (complemented by a textual data mining approach), where the goal is to find a set of scientific papers from a given subject. Section 8 describes experimental results of such a web crawler application where a k-clique of referencing papers is searched in Google Scholar.

This paper is organized as follows. First we formally define the problem of finding a pattern in an unknown graph (Section 2) and list related work (Section 3). Then, a best-first search framework for solving this problem is described (Section 4). Several deterministic heuristic algorithms are given for this best-first search framework (Section 5), as well as a heuristic algorithm that can exploit probabilistic knowledge of the searched graph (Section 6). Following, we analyze the proposed best-first search framework theoretically (Section 7) and compare the described heuristic algorithms experimentally (Section 8).The paper finally concludes with a discussion and future work (Section 9).

A preliminary version of this paper already appeared [1] focusing on searching for the k-clique pattern in an unknown graph. In this paper we take a big step beyond that work, as detailed in Section 3.

2. Problem Definition

Following are several definitions and notations required for formally describing the problem of finding a specific pattern in an unknown graph.

Some graph problems are given a graph as input explicitly, in data structures such as adjacency list or adjacency matrix. All the vertices and edges of the graph are easily accessible by searching the given data structure. We call such problems *explicitly known graph problems*. In other graph problems, the graph is given as input implicitly, by an initial set of vertices and a set of computational operators. These operators can be applied to a vertex to discover its neighbors. Consequently, the vertices and edges of the graph can be discovered by applying the given operators. We call such problems *implicitly known graph problems*. Prominent examples of implicitly known graph problems are various combinatorial puzzles such as the sliding tile puzzles and Rubik's cube. Similarly, state graphs of planning problems are also given implicitly. In general, we regard both *explicitly known graph problems* and *implicitly known graph problems* as *known graph problems*.

In this paper we focus on a different type of problems, that we call *unknown graph* problems. Similar to the *implicitly known* graph problems, in *unknown graph* problems, the vertices and edges of the graph are not given in advance, except for an initial vertex. By contrast, in an *unknown graph* problem the vertices and edges of the graph cannot be accessed via any computational operator alone. Exploring vertices and edges of the graph requires applying exploration action that incur a different cost than CPU cycles. In an *unknown graph* problem we aim at minimizing this cost.

Let $G = (V, E)$ be the initially unknown graph, i.e., the searched graph, and let $G_P = (V_P, E_P)$ be the searched pattern. The input to the problem addressed in this paper is the pattern graph G_P, a single vertex s from the searched graph G, and an exploration action, which is defined next.

Definition 1 *[Explore]*
$Explore{:}V \rightarrow 2^V$ is a function that returns all the neighbors of a given vertex in the searched graph.

This exploration model is inspired by the *fixed graph* model [2]. Each exploration action has a corresponding exploration cost which depends on the vertex:

Definition 2 *[Exploration Cost]*
The function $ExpCost : V \rightarrow \mathbb{R}^+$ returns the cost of exploring a given vertex.

The cost function is additive, meaning that the total cost of multiple exploration actions is the summation over all the exploration costs of the explored vertices. Now we can define the problem of finding a pattern in an unknown graph:

Definition 3 *[Subgraph isomorphism problem in an unknown graph]*
Given a pattern graph G_P, a vertex s in the searched graph G and an exploration action $Explore()$ with an associated exploration cost $ExpCost()$, the goal is to find a subgraph of G that is isomorphic to G_P with minimal exploration cost.

Note that finding a subgraph that is isomorphic to G_P requires finding all the vertices and edges of that subgraph. In this paper we simplify the problem by assuming a constant exploration cost C, i.e., $\forall v \in V \ ExpCost(v) = C$ for some C. This is motivated by a number of real world scenarios such as the following examples: (1) a central controller that can be queried to provide the exploration data, and (2) querying a web page from a single host (this example is further justified in Section 8). For simplicity and without loss of generality, we will assume in the rest of the paper that $C = 1$. This focuses the problem on minimizing the number of vertices that are explored before finding the desired pattern in the searched graph.

Also, for clarity of presentation we assume that the searched graph is a connected and undirected. Extending the results in the paper to directed graphs is straightforward. In Section 4 we briefly discuss the case of an unknown graph with more than a single connected component.

3. Related Work

There are several problems that have already been researched in the context of an unknown domain that can be represented as a graph. A prominent example is the exploration problem, where the goal is to visit *all* the locations (vertices) in an unknown environment. Much work has been previously done on exploration [3,4,5,6,7,8,2] for various types of graphs and agents. The key difference between our work and exploration is that while in an exploration task the goal is to explore all the vertices, we try in our work to avoid exploring the entire graph in order to minimize the exploration cost. Indeed, as the results in Section 8 show, the desired pattern can often be found without exploring large parts of the unknown graph.

3.1. Pathfinding

Another related topic is path finding in an unknown environment, where the goal is to find a path between

two locations. This challenge has been researched extensively in the fields of robotics and artificial intelligence. Navigation is a special variant of path finding in an unknown environment, in which exploring a vertex requires moving to it and the goal is to minimize the movements until a path is found. There are many navigation algorithms for numerous variations of the navigation problem [9,10,11,12]. Examples of navigation variants include performing a navigation task between 2 vertices repeatedly, with a single agent [13] and with multiple agents and various communication paradigms [14]. Another work studied the problem of distributed navigation of multiple agents. [15].

Pathfinding problems are often regarded in the context of real-time search. In real-time search the goal is to find an efficient path to a goal, but the amount of runtime allowed until a movement has to be performed is limited. As a result, the navigation planning and execution are interleaved. There are many real-time search algorithm, such as Real-Time A* and Learning Real-Time A* [10], Prioritized LRTA* [16], Time-Bounded A* [17] and more. In the unknown graph context we do not assume any real-time constraints on the runtime between exploration actions.

Many navigation algorithms try to find any path to a goal location. Roadmap-A* [18] is a pathfinding algorithm that does impose constraints on the length of the resulting path, while Physical A* [19,20] finds the shortest path in an unknown physical graph.

Notice that Physical A* and Roadmap-A*, as well as all the navigation algorithms described above, are designed for a physical unknown graph, i.e., where the cost of exploring a vertex is the distance from the last vertex explored (as a physical entity needs to move from the last vertex that has been explored to the new one). In this work we focus on a different problem (finding a pattern in a graph) with a different exploration cost model (constant exploration cost).

While navigation and path finding problems are very important, the goal of the work presented in this paper is to find a specific pattern of vertices and edges, and not a path to a single vertex.

3.2. Searching for a Pattern in a Graph

Finding patterns in a graph is a well known NP-Complete problem [21], also known as subgraph isomorphism. Ullman presented the classical backtracking and pruning algorithm [22]. Much work has followed that improves this algorithm with better vertex ordering [23] or by partitioning the graph to pieces and applying dynamic programming [24]. For the special case of planar graphs it is even possible to find patterns in linear time [25].

An important and well studied special case of searching for a pattern in a graph is the problem of finding a clique in a graph. This problem is NP-Complete as well [21]. Bron and Kerbosch presented the classical algorithm for finding all the maximal cliques in a graph [26]. Many algorithms exist for finding the largest clique in a graph [27,28,29]. However, most clique algorithms are designed for *explicitly known* graph problems, and exploit a priori knowledge of the graph. For example, a common heuristic when searching for a clique is to start the search with the vertex that has the highest degree or by pruning vertices that have low degree. In an unknown graph, pruning all the vertices that have such a property (a low degree), requires exploring the entire graph, which is not relevant if we wish to minimize the exploration cost.

The state-of-the-art algorithms for finding the largest clique in a known graph are based on local search [30,31,32,33]. In Section 8.1 we describe these in more details. Furthermore, we propose how to adapt them to the unknown graph setting and discuss the limitations of these local search algorithms.

A closely related work is the preliminary work on searching for k-cliques in physical unknown graphs with a swarm of agents [34]. In that work, each agent was directed to explore the closest largest clique in the known graph. First, this paper goes beyond the specific clique pattern and addresses the more general problem of searching for any specific pattern. Second, there is a key difference between the physical unknown graph setting and the setting described in this paper. We address unknown graphs that are not necessarily physical, which means that a vertex is not necessarily a physical location. Therefore the requirement that an agent should be physically located in a vertex v in order to explore it is dropped. Thus the two-level approach that is used for physical unknown graphs is not appropriate, as the lower level is redundant. Third, the focus of that work was mainly on task allocation for multiple agents. In this paper we consider a single agent, and provide theoretical analysis of our problem in addition to experimental results.

A preliminary version of this paper appeared earlier [1], discussing only the specific k-clique pattern. This paper contains several contributions over the previously published paper. We generalize the problem from the specific pattern of a k-clique to the problem of searching for any pattern. This generalization

Procedure Explore

Input: $v \in V_{gen}$, The vertex to explore

1 $V_{exp} \leftarrow V_{exp} \cup \{v\}$
2 $V_{gen} \leftarrow (V_{gen} \setminus \{v\}) \cup (neighbors(v) \setminus V_{exp})$
3 $V_{known} \leftarrow V_{known} \cup neighbors(v)$
4 $E_{known} \leftarrow E_{known} \cup \{e \in E | e = (v, u), \forall u \in neighbors(v)\}$

includes theoretical analysis and new heuristic algorithms. Also, the empirical evaluation is more extensive than previous work, including comparison with an existing k-clique algorithm that was adapted to the unknown graph setting (Section 8.1).

4. Best-First Search in an Unknown Graph

According to our problem definition (Definition 3), the task in an unknown graph problem is to minimize the exploration cost and not the computational effort. Therefore, it is worthwhile to store all the parts of the searched graph that have been returned by an exploration action. Let V_{known} be a set containing the initial vertex s and all of the vertices that have been returned by an exploration action.

Definition 4 *[Known Subgraph]*
$G_{known} = (V_{known}, E_{known})$ *represents the subgraph of the searched graph that contains all the vertices in* V_{known} *and the edges between them.*

In this paper, G_{known} is referred to as the known subgraph of G, or simply the *known subgraph*. For ease of notation, we borrow the terms *expand* and *generate* from the classical search terminology in the following way. Applying an exploration action (Definition 1) to vertex v will be referred to as *expanding* v and *generating* the neighbors of v. During the search, we denote $V_{exp} \subseteq V_{known}$ as the set of all the vertices that have already been expanded, and $V_{gen} = V_{known} \setminus V_{exp}$ as the set of all the vertices that have been generated but not expanded. Notice that new vertices and edges are added to G_{known} only when expanding vertices from V_{gen}, as all the other vertices in G_{known} have already been expanded. A typical vertex goes through the following stages. First it is unknown. Then, when it is generated it is added to G_{known}. Finally, when it is expanded, it is moved to V_{exp} and its incident edges and neighboring vertices are also added to G_{known}.

Procedure *Explore()* lists how the lists described above (V_{exp},V_{gen}, V_{known} and E_{known}) are updated when a vertex v from V_{gen} is expanded. Vertex v is inserted to V_{exp} (line 1). All neighbors of v that have not been expanded yet are added to V_{gen} while v is removed from V_{gen} (line 2). Finally, G_{known} is updated with the vertices and edges that are connected to v (lines 3–4).

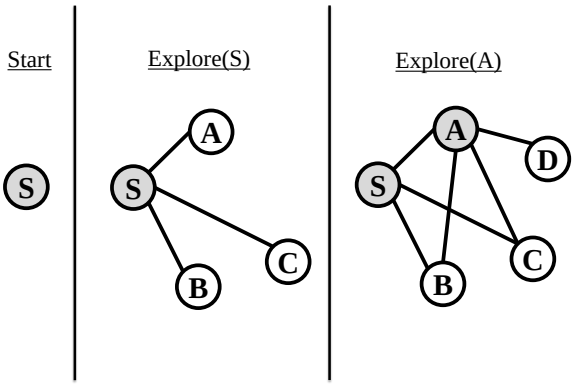

Figure 1. Example of exploring an unknown graph.

As an example of the exploration process, consider the two graphs displayed in Figure 1. Initially, only vertex S is known. Then, S is explored. The known subgraph G_{known} after exploring vertex S is shown on the middle graph in Figure 1. All the neighbors of S are added to V_{gen} and S is added to V_{exp}. Then, A is explored and the corresponding known subgraph is shown on the right graph in Figure 1. As can be seen, when vertex A is explored, vertex D is discovered, and will now be added to V_{gen}. Since vertex A has just been explored, it is moved from V_{gen} to V_{exp}. Also, the edge between A and C and the edge between A and B are now added to G_{known}.

Searching for a pattern in an unknown graph is inherently an iterative process, in which the graph is explored vertex by vertex. Since the goal is to minimize the exploration cost and not the computational effort, it is worthwhile to trade computation effort for saving unnecessary explorations. Therefore, we propose to search an unknown graph in a best-first search manner, as listed in Algorithm 1.

First, the algorithm checks whether there is a subgraph in the known subgraph (i.e., G_{known}) that is isomorphic to G_P (*test()* in line 3). If not, it chooses the next vertex to explore (line 4) and explores it (line 5). This process is repeated until the desired pattern is found or the entire graph has been explored (in case the desired pattern does not exist in G). Corresponding to a regular best-first search terms, V_{gen} is the *open-list*,

Algorithm 1: Best-first search in an unknown graph

```
   Input: s, Initial known vertex
   Input: G_P, the desired pattern
1  V_gen ← s
2  V_exp ← ∅
3  while (V_gen ≠ ∅) AND
   (test(G_P, G_known)=False) do
4  |  v ← chooseNext(V_gen)
5  |  Explore(v)
6  end
7  if test(G_P, G_known)=True then
8  |  Return True
9  else
10 |  Return False
11 end
```

V_{exp} is the *closed-list* and the *test()* action is the goal test.

Recall that the problem addressed in this paper is defined for the case where only a single vertex s is initially known, and the number of vertices in the searched graph is not known (Definition 3). Under this setting, if the searched graph is composed of a single connected component, Algorithm 1 is complete.

This is because G_{known} is initialized with s and in every iteration a vertex is expanded. Hence, eventually all the vertices on the same connected component as s will be expanded, finding the searched pattern or verifying that such a pattern does not exist in the searched graph.

However, Algorithm 1 can be easily extended to a partially known graph, by simply initializing G_{known} with the set of the initially known vertices. For example, if all the vertices in the unknown graph are known, but the edges are not, then G_{known} is initialized as $G_{known} = (V, \{\})$ (where V represents all the vertices in the searched graph). In such a case, every vertex in the graph can be chosen for exploration. If the graph is composed of more than a single connected component then it is easy to see that G_{known} must be initialized with at least one member of each connected component $C \subseteq G$.

4.1. Computational complexity

The goal in this paper it to minimize the number of exploration actions until finding the searched pattern. However, we also provide analysis of the computational complexity of the algorithms that are presented in this paper. The computational complexity of an iteration of Algorithm 1 is composed of the computational complexity of:

1. Checking if the desired pattern is isomorphic to a subgraph of G_{known} (*test()* in line 3).
2. Choosing the next vertex to explore (*chooseNext()* in line 4).
3. Performing the exploration action and update the known subgraph (*explore()* in line 5).

First, consider the computational complexity of the *test()* action (line 3). Searching for a pattern in a known graph can be preformed with a backtracking algorithm [22] that is polynomial in the number of vertices in the graph. The degree of the polynomial is the size of the pattern graph (in the worst case). Taking the k-clique pattern as an example, the computational complexity of searching for a k-clique pattern is $O(|V_{known}|^k)$ in the worst case.[1] This is a worst case analysis and there are many heuristic algorithms that are much more effective in practice [23], as well as more efficient algorithms for special types of graphs [25]. In addition, this search can be done incrementally, searching in every iteration only the subgraph of G_{known} that contains the newly added vertices and their neighbors. Notice also that this search is performed only on the known subgraph (G_{known}), and thus it does not require any exploration cost. Although the focus of this work is on minimizing the exploration cost and not computational cost, in all our experimental settings (Section 8) we have found that the computational effort of the *test()* actions is less than a second for all the k values we have tested.

Next, consider the *chooseNext()* (line 4) action. This is the main challenge when searching in an unknown graph, as this is the only part of the best-first search that affects the exploration cost. In the following sections we propose several heuristic algorithms for choosing the next vertex to explore. These heuristic algorithms have different computational complexity, ranging from O(1) heuristics to more computationally exhaustive heuristic algorithms (e.g., Sections 5.2 and 6.2). Finally, the computational complexity of updating the known subgraph, displayed in Procedure Explore(), is linear in the number of neighbors of the explored vertex: simply update G_{known} (and supporting data structures) with the newly added vertices and edges.

[1] If $k \in O(|V|)$ then the problem is NP-complete [21].

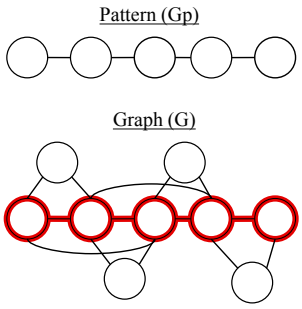

Figure 2. Example of a subgraph that is not an induced subgraph.

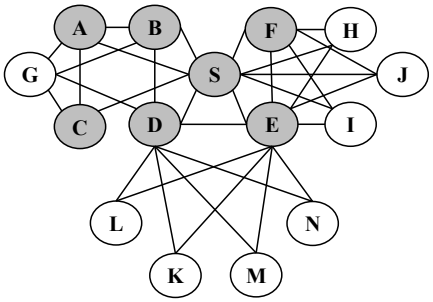

Figure 3. Example of $KnownDegree$.

5. Deterministic Heuristics

Next we describe several heuristic algorithms for choosing the next vertex to explore and discuss their analytical properties.

5.1. $KnownDegree$

Consider again the example of finding a k-clique in a graph. A very common and effective heuristic used for the k-clique problem in known graphs is to search first vertices with a high degree [28]. Vertices with a high degree are more likely to be a part of a k-clique, than vertices with a lower degree [35]. This is also true for any pattern - vertices with high degree are more likely to be part of any specific pattern. This is because the problem we address in this paper is to find a subgraph of G that is isomorphic to the pattern graph (See Definition 3 above), and not an *induced subgraph*. A graph $G' = (V', E')$ is a *subgraph* of a graph $G = (V, E)$ if all the vertices and edges in G' exist in G, i.e., $V' \subseteq V$ and $E' \subseteq E$. Thus, if the a vertex in the searched graph has more edges then its corresponding vertex in the pattern graph, it can still be "matched" to it.[2] For example, consider the pattern G_p and graph G displayed in Figure 2. G contains a subgraph that is isomorphic to G_p (marked by red circles), but G does not contain an induced subgraph that is isomorphic to G_P.

Since the real degree of a vertex $v \in V_{gen}$ is not known as it has not been expanded yet, we consider its *known degree*, which is the number of *expanded* vertices that are adjacent to v, i.e., v was seen when these nodes were expanded. We denote by $KnownDegree$ the algorithm that chooses to expand the vertex with the highest known degree in V_{gen}. For example, consider the graph in Figure 3. Throughout this paper, we will mark expanded vertices in gray, and generated vertices in white. The generated vertex G has a known degree of 4 (it was seen from vertices A,B,C and D when they were expanded), vertices H,I and J have a known degree of 3, and vertices K,L,M and N have a known degree of 2. Hence, $KnownDegree$ will choose to expand vertex G.

In terms of computational complexity, it is possible to implement $KnownDegree$ with only $O(D \cdot log(|V|)$ overhead in each iteration, where D is the maximum degree of a vertex in G. This can be done by storing all the vertices in V_{gen} in a priority queue ordered by the known degree of the vertices. In every iteration a maximum of D vertices will have their known degree updated, causing $log(D \cdot |V_{gen}|)$ operations to maintain the priority queue, and $|V_{gen}|$ is bounded by $|V|$.

An obvious shortcoming of $KnownDegree$ is that it ignores the actual pattern that is searched. Next we propose a more sophisticated heuristic algorithm that considers the structure of the searched pattern.

5.2. $Pattern^*$

The next heuristic algorithm that we present is called $Pattern^*$. $Pattern^*$ estimates the number of exploration actions required until the searched pattern is found. It then uses this estimation to choose the next vertex to expand.

We first define the concept of *extending* the known subgraph.

Definition 5 *[Extension of the Known Subgraph] A graph G' is denoted as an* m-extension *of G_{known} if it is possible that after expanding m vertices, G_{known} will be equal to G'.*

We say that a graph G' is an extension of G_{known} if there exists a number m such that G' is an m-

[2] By contrast, G' is an induced subgraph of G if all the vertices and edges in G' exist in G, and all the edges in G between the vertices in V' exist in E'.

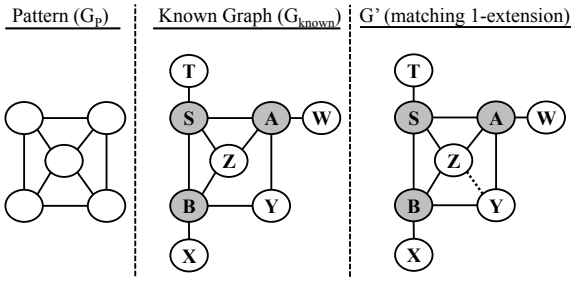

Figure 4. Example of a matching extension and the $Pattern^*$ heuristic algorithm.

extension of G_{known}. Next, we define the set of all m-extensions that contain a subgraph that is isomorphic to the searched pattern G_P.

Definition 6 *[Matching Extensions]*
The set of all m-extensions of G_{known} that include a subgraph that is isomorphic to the searched pattern G_P is referred to as the set of matching m-extensions *and denoted by $\mathcal{ME}_m$.*

For a given vertex v, we refer to the subset of matching m-extensions in which v is a part of the subgraph that is isomorphic to G_P as the *matching m-extensions of vertex v*, and denote it by $\mathcal{ME}_m[v]$. For every vertex v, the minimal m such that $\mathcal{ME}_m[v] \neq \emptyset$ is referred to as the *pattern distance* of v.

As an example, consider the graphs displayed in Figure 4. The left graph is the searched pattern G_P, and the graph in the middle is the known subgraph G_{known}. The rightmost graph in Figure 4 denoted by G' is a possible 1-extension. G' is a 1-extension because it might be discovered after one exploration action - if either vertex Z or Y are expanded next and an edge between them will be discovered. In addition, G' is a *matching 1-extension*, since it contains a subgraph that is isomorphic to the searched pattern - the subgraph with the vertices $\{S, A, B, Z, Y\}$. Thus in this case the *pattern distance* of vertex Z or Y is one. Using the notations described above, we have that $G' \in \mathcal{ME}_1$ as well as $G' \in \mathcal{ME}_1[Z]$ and $G' \in \mathcal{ME}_1[Y]$.

Informally, the $Pattern^*$ heuristic algorithm presented next chooses to expand the vertex that is the "closest" to the searched pattern, where "closeness" of a vertex v is measured by the *pattern distance* of v. Algorithm 2 describes $Pattern^*$ in details. m (the pattern distance) is initialized by one. If no vertex has any m-extensions, m is incremented. Otherwise, $Pattern^*$ returns a random vertex amongst the vertices that have a matching m-extension. Note that after at most $|G_P|$ iterations a vertex must be returned, since after expanding $|G_P|$ vertices there is an extension where new vertices (i.e., vertices that were not in G_{known}) form the desired pattern.

Algorithm 2: $Pattern^*$

Input: G_P, The searched pattern
Input: G_{known}, The known subgraph
Input: V_{gen}, The list of vertices that can be expanded next
Output: The next vertex to expand

```
for m=1 to |G_P| do
    V_match ← {v ∈ V_gen || ME_m[v]| > 0}
    if V_match is not empty then
        return a random vertex from V_match
    end
end
```

As an example of $Pattern^*$, consider again the pattern and the known subgraph displayed in Figure 4. In the next iteration, either vertex Z, Y, X, W or V will be chosen for expansion. It is easy to see that vertices Z and Y have a matching 1-extension, displayed in the right part of Figure 4. By contrast, vertices X, W and V do not have a matching 1-extension, as they cannot be connected to the previously expanded vertices (S,A and B). Thus, $Pattern^*$ will not return X, W or V and choose randomly to return either Z or Y.

$Pattern^*$ has the following property.

Theorem 1 *The* pattern distance *of the vertex that is returned by $Pattern^*$ is a tight lower bound on the number of expansions required until the searched pattern is found.*

This lower bound is *tight* in the sense that no larger lower bound exists. [3]

Proof: Assume by negation that it is possible to reach the searched pattern after C expansions, while the vertex v returned by $Pattern^*$ has a *pattern distance* of $C' > C$. If the searched pattern can be reached after C expansions, then there exists a C-extension of G_{known} denoted by G' that contains a subgraph that is isomorphic to the searched pattern G_P. Therefore, there exists a vertex $v' \in V_{gen}$ that has a matching C-extension.

[3] If one knew the unknown graph, then clearly a higher lower bound could be achieved. However, this is not the case in an unknown graph problem. The described above lower bound is tight in the sense that given G_{known} and V_{gen} it is the highest lower bound that can be given. In other words, there is a graph G' that is an extension of G_{known} that violated any higher lower bound (i.e., in G' the searched pattern can be found with less node expansions).

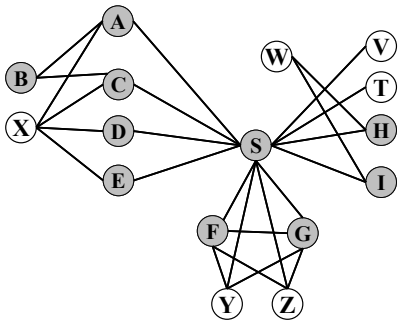

Figure 5. Example of potential patterns.

This contradicts the fact that v has been returned by $Pattern^*$, since by definition of $Pattern^*$ there is no vertex in V_{gen} with a m-matching extension for any $m < C'$.□.

A key challenge when implementing $Pattern^*$ is how to identify if a vertex has a matching m-extension for a given value of m. In general, this is intractable, as the number of matching m-extensions can be infinite, since we do not known the number of vertices in the searched graph. Also, checking if each m-extension is a matching m-extension requires solving the NP-Complete problem of subgraph isomorphism. Thus, $Pattern^*$ is clearly more computationally intensive than $KnownDegree$. However, it can be implemented more efficiently for specific patterns. Next, we demonstrate such an implementation for the k-clique pattern and the pattern of a complete bipartite graph.

5.2.1. The k-Clique Pattern

In order to implement $Pattern^*$ for the k-clique, we introduce the concept of a potential k-clique and supporting terms.

Definition 7 *[Potential k-Clique]*
A set of vertices $PC_k \subseteq V_{exp}$ is a potential k-clique if there exists an extension of G_{known} that contains a k-clique C_k such that $PC_k = C_k \cap V_{exp}$.

As an example, consider the graph in Figure 5. The set $\{S, F, G\}$ is a potential 5-clique, since if vertices Y and Z will be expanded, they might turn out to be connected to each other, and as a result the set of vertices $\{S, F, G, Y, Z\}$ will form a 5-clique. By contrast, the set $\{S, H, I\}$ is not a potential 5-clique, since vertices H and I will never have an edge between them, as they do not have an edge in the known subgraph, and they have already been expanded.

For ease of notation, we define the gcn function, that can be applied to any group of expanded vertices.

Definition 8 *[Generated Common Neighbors]*
Let V' be a set of expanded vertices. Then $gcn(V') = \bigcap_{v \in V'} neighbors(v) \cap V_{gen}$, if V'.
For example, $gcn(\{S, F, G\}) = \{Y, Z\}$ for the graph in Figure 5.

The relation between a potential k-clique and finding a matching m-extension for the k-clique pattern is as follows. A generated vertex v has a matching m-extension if there is a potential k-clique PC_k such that $m+|PC_k|+1 \geq k$ and v is connected to all the vertices in PC_k (i.e., $v \in gcn(PC_k)$). Conversely, it is possible to check if a vertex v has a matching m-extension for the k-clique pattern, by checking if there exists a potential k-clique PC_k such that $v \in gcn(PC_k)$ and $m \geq k - |PC_k| - 1$. The following corollary presents easy-to-compute conditions for checking if a set of vertices is a potential k-clique.

Corollary 1 *A set of vertices $PC_k \subseteq V_{exp}$, where $|PC_k| < k$, is a potential k-clique* if and only if

1. *PC_k is a clique.*
2. *$|gcn(PC_k)| + |PC_k| \geq k$.*

Proof: ($\Leftarrow$) Let m be the number of vertices in PC_k (i.e. $m = |PC_k|$). By definition, all the vertices in $gcn(PC_k)$ have been generated but have not been expanded yet. Therefore, there exists an extension G' where the vertices in $gcn(PC_k)$ form a clique. Hence, in G' the vertices in $gcn(PC_k) \cup PC_k$ also forms a clique, since all the vertices in $gcn(PC_k)$ are neighbors of all the vertices in PC_k (Definition 8). Since $|gcn(PC_k)| + |PC_k| \geq k$ we have that G' contains a k-clique $gcn(PC_k) \cup PC_k$, and thus PC_k is a potential k-clique as required (Definition 7).

($\Rightarrow$) Assume that a set of vertices PC_k is a potential k-clique. According to Definition 7 this means that there exists a k-clique C_k in an extension of G_{known} such that $PC_k \subset C_k$. Every subset of vertices of a clique also forms a (smaller) clique. Thus PC_k must also be a clique. Furthermore, every vertex in $C_k \setminus PC_k$ must be connected to every vertex in PC_k (or else C_k would not not be a clique). Thus PC_k must also have $k - |PC_k|$ common neighbors as required.□

Recall, that the $Pattern^*$ heuristic algorithm chooses to expand the vertex with the lowest *pattern distance*. It is easy to see that every vertex with the lowest *pattern distance* is in fact a member o the gcn of the largest potential k-clique. Implementing $Pattern^*$ for the k-clique pattern can therefore be done by choosing to expand a random vertex from the gcn of the largest potential k-clique.[4] For clarity and to conform with pre-

[4] If there are several potential k-cliques of the same largest size, choose a random vertex from the union of their gcn.

Procedure IncrementalUpdate

Input: v, The vertex that was just expanded

1 **foreach** PC *in* $\mathcal{PC}_k$ **do**
2 **if** v *is a neighbor of all the vertices in* PC **then**
3 $PC' \leftarrow PC \cup \{v\}$
4 **if** $|gcn(PC')| \geq k - |PC'|$ **then**
5 Add PC' to $\mathcal{PC}_k$
6 **end**
7 **if** $|gcn(PC)| < k - |PC|$ **then**
8 Remove PC from $\mathcal{PC}_k$
9 **end**
10 **end**
11 **end**
12 **if** $|gcn(\{v\})| \geq k - 1$ **then**
13 Add $\{v\}$ to $\mathcal{PC}_k$
14 **end**

vious publications, we denote by $Clique^*$ this implementation of $Pattern^*$.

There are several ways to implement $Clique^*$ with different computational complexities. We give details on our own implementation which was used in all our experiments (described in Section 8). $Clique^*$ was implemented by maintaining a global list of potential k-cliques, denoted by $\mathcal{PC}_k$. Every set of expanded vertices that form a potential k-clique is stored in $\mathcal{PC}_k$. Note that a vertex may be part in more than one potential k-clique. When a vertex is expanded, $\mathcal{PC}_k$ is updated incrementally, as follows.

In the beginning of the search, the initial vertex s is expanded. If s has more than $k-1$ neighbors, then $\{s\}$ is a potential k-clique, and $\mathcal{PC}_k$ is set to be $\{\{s\}\}$. Otherwise, $\mathcal{PC}_k$ is initialized as an empty set. Following, after a vertex v is expanded, $\mathcal{PC}_k$ is updated according to the pseudo code listed in Procedure IncrementalUpdate(), described next.

After a vertex v is expanded, every potential k-clique PC is examined.[5] If v is not a neighbor of all the vertices in PC, then PC remains unchanged. Otherwise, there are two (not mutually exclusive) options:

1. PC with v is a new potential k-clique that should be added to $\mathcal{PC}_k$ (line 5).
2. PC is no longer a potential k-clique, and should be removed from $\mathcal{PC}_k$ (line 8).

[5] In our implementation, we stored a mapping of every vertex $v \in V_{gen}$ to all the potential k-clique PC that it is in $gcn(PC)$. Then, only these potential k-cliques should be considered when v is expanded.

Checking these two options can be done easily according to Corollary 1. The first option is checked as follows. Since v is a neighbor of all the vertices in PC, then clearly $PC \cup \{v\}$ is a clique. Thus if $gcn(PC \cup \{v\}) \geq k - |PC \cup \{v\}|$, then $PC \cup \{v\}$ is a new potential k-clique and should be added to $\mathcal{PC}_k$ (line 5 in Procedure IncrementalUpdate()). For the second option, PC was a clique before v was expanded, and thus PC is still a clique. However, $gcn(PC)$ has decreased by 1 since v has now been expanded. Thus, PC will be removed from $\mathcal{PC}_k$ if $|gcn(PC)| < k - |PC|$ (line 8). Finally, v itself might start a new potential k-clique without any other expanded vertex. Thus if $|gcn(\{v\}| \geq k-1$ then $\{v\}$ is a new potential k-clique (line 13).

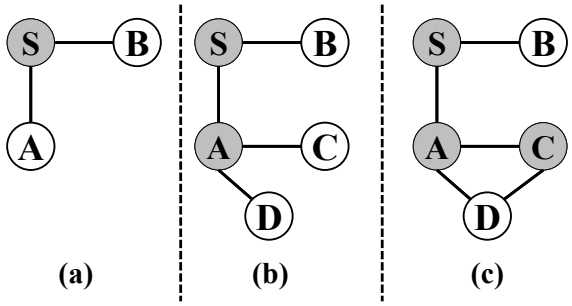

Figure 6. An example of the incremental update of the set of potential k-clique.

As an example of Procedure IncrementalUpdate(), consider the graphs in Figure 6 and assume that the size of the searched clique is 3. Initially, the known subgraph is the graph displayed in Figure 6(a). At this stage $\mathcal{PC}_3 = \{\{S\}\}$, i.e., there is only a single potential 3-clique, consisting of vertex S. $\{S\}$ is a potential 3-clique since $|gcn(\{S\})| = |\{A, B\}| = 2 \geq k - |\{S\}| = 2$. Next, vertex A is expanded, resulting in the known subgraph becoming the graph in Figure 6(b). Adding A to the potential 3-clique $\{S\}$ do not create a new potential 3-clique, since $gcn(\{S, A\}) = \emptyset$. Furthermore, after expanding A the set $\{S\}$ is no longer a potential 3-clique, since now $|gcn(\{S\})| = |\{B\}| = 1$. However, $\{A\}$ is a new potential 3-clique, since $|gcn(\{A\})| = |\{C, D\}| = 2$. Thus after expanding vertex S we have $\mathcal{PC}_3 = \{\{A\}\}$. Next, assume that vertex C is expanded, resulting in the graph in Figure 6(c). Now, the set $\{A, C\}$ is a potential 3-clique, since C and A form a 2-clique and $|gcn(\{C, A\}| = |\{D\}| = 1 \geq k - |\{C, A\}| = 2$. In fact, at this stage we can see that $\{A, C, D\}$ is a 3-clique and the search can halt.

The total computational complexity of the implementation of $Clique^*$ described above is composed of two parts: (1) updating $\mathcal{PC}_k$ according to Procedure IncrementalUpdate(), and (2) returning a vertex that is in the gcn of the largest member of $\mathcal{PC}_k$. The first step (Procedure IncrementalUpdate()) required $O(|\mathcal{PC}_k| \times |neighbors(v)|)$. The second step can be easily integrated into Procedure IncrementalUpdate(), by storing the largest potential k-clique and returning one of the vertices in its gcn. The computational complexity of this implementation of $Clique^*$ is therefore $O(|\mathcal{PC}_k| \times |neighbors(v)|)$.

5.2.2. Complete Bipartite Graph

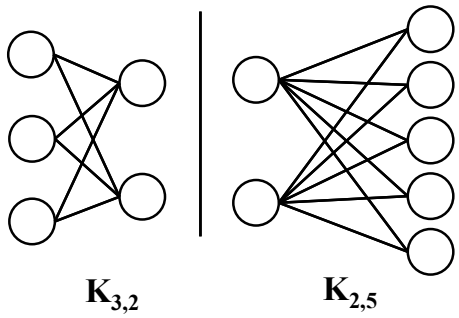

Figure 7. Examples of complete bipartite graphs.

Next, we demonstrate a method for applying $Pattern^*$ to the pattern of a *complete bipartite graph*. A *bipartite graph* is a graph whose vertex set can be partitioned into two subsets X and Y, so that each edge has one end in X and one end in Y. Such a partition is called a *bipartition* of the graph. A *complete bipartite graph* is a bipartite graph with bipartition (X, Y) in which each vertex of X is joined to *all* vertices of Y. If $|X| = p$ and $|Y| = q$, such graph is denoted by $K_{p,q}$ [36]. Figure 7 shows $K_{3,2}$ and $K_{2,5}$ [37].

Similar to the definition of a potential k-clique, we can define a *potential* $K_{p,q}$ as follows.

Definition 9 *[Potential $K_{p,q}$]*
A pair of sets of vertices $V_p, V_q \subseteq V_{exp}$ is a potential $K_{p,q}$ if there exists an extension of G_{known} that contains a pair of sets of vertices C_p and C_q such that the following conditions hold:

1. $\langle C_p, C_q \rangle$ *is a bipartition that forms a* $K_{p,q}$
2. $V_p = C_p \cap V_{exp}$
3. $V_q = C_q \cap V_{exp}$

As an example, consider again the graph displayed in Figure 5. The pair $\langle \{S\}, \{H, I\} \rangle$ is a potential $K_{2,4}$, since if vertex W is connected to vertices V and T, then the two sets $\{S, W\}$ and $\{H, I, V, T\}$ will form a complete bipartite graph $K_{2,4}$.

Finding a k-clique requires expanding at least $k-1$ members of that k-clique. A similar condition is described next for the $K_{p,q}$ pattern.

Lemma 1 *A $K_{p,q}$ that is composed of the bipartition $\langle C_p, C_q \rangle$ is found (i.e., it is in G_{known}) if either all the vertices in C_p are expanded or if all the vertices in C_q are expanded.*

Proof: If all the vertices in C_p are expanded, then the edges between them and all the vertices in C_q are added to G_{known} and the pattern is found. The same reasoning apply if all the vertices in C_q are expanded. By contrast, assume that there exists two vertices $v_p \in V_p$ and $v_q \in V_q$ there were not expanded. This means that the edge between them is not in G_{known}. Thus, $\langle C_p, C_q \rangle$ is not completely in G_{known}. □

The notion of which vertices need to be expanded to find a pattern is generalized later in this paper in Lemma 4.

In this section we define $gcn(\{\}) = V_{gen}$. Using this extended definition of gcn and Lemma 1, we give the following method to check if a set of vertices is a potential $K_{p,q}$.

Corollary 2 *A pair of sets of vertices $V_p, V_q \subseteq V_{exp}$, where $|V_p| \leq p$ and $|V_q| \leq q$, is a potential $K_{p,q}$* if and only if

1. $\langle V_p, V_q \rangle$ *is a* $K_{|V_p|,|V_q|}$
2. $|gcn(V_p)| + |V_q| \geq q$
3. $|gcn(V_q)| + |V_p| \geq p$

We omit the proof due to its simplicity.

The relation between a potential $K_{p,q}$ and a matching m-extension is similar to the relation described in Section 5.2.1 between a potential k-clique and a matching m-extension. To describe this relation, the following notations are used. $PK_{p,q} = \langle V_p, V_q \rangle$ denotes a potential $K_{p,q}$ that is composed of the set of vertices V_p and V_q. A generated vertex v is said to be *connected* to a potential $K_{p,q} = \langle V_p, V_q \rangle$ if $v \in gcn(V_p)$ or $v \in gcn(V_q)$. Finally, the *distance* of a potential $K_{p,q}$, denoted by $dist(\langle V_p, V_q \rangle)$, is defined as $\min(p - |V_p|, q - |V_q|)$.

The following lemmas describe the relation between a matching m-extension and a potential $K_{p,q}$.

Lemma 2 *Let $PK_{p,q} = \langle V_p, V_q \rangle$ be a potential $K_{p,q}$. There is a matching m-extension for all the vertices in $\langle V_p, V_q \rangle$, where $m = dist(\langle V_p, V_q \rangle)$.*

Proof: Since $PK_{p,q}$ is a potential $K_{p,q}$ there exists an extension of G_{known} with a $K_{p,q} = \langle C_p, C_q \rangle$ such that V_p and V_q are part of C_p and C_q respectively. Thus, by expanding either the remaining $p - |V_p|$ vertices from C_p, or the remaining $q - |V_q|$ vertices from C_q, the searched pattern (the $K_{p,q}$) will be found (Lemma 1). Hence, there is a matching m-extension for every vertex in $\langle V_p, V_q \rangle$ where $m = min(p - |V_p|, q - |V_q|) = dist(PK_{p,q})$ as required. □

Lemma 3 *For any vertex v, if there exists a matching m-extension of v, such that $m \leq \min(p, q)$, then v is connected to a potential $K_{p,q}$ with distance equal to or smaller than m.*

Proof: Assume that G' is a matching m-extension of vertex v such that $m \leq \min(p, q)$. Let $\langle C_p, C_q \rangle$ be the $K_{p,q}$ in G' that contains v. Since $m \leq \min(p, q)$, this means that at least one vertex u from $\langle C_p, C_q \rangle$ has already been expanded. Thus, there exists a corresponding potential $K_{p,q} = \langle V_p, V_q \rangle$. As vertex v is part of $\langle C_p, C_q \rangle$, it is either connected to all the vertices in C_p or it is connected to all the vertices in C_q. Thus, v must either be in $gcn(V_p)$ or in $gcn(V_q)$. By definition, this means that v is *connected* to $\langle V_p, V_q \rangle$. Finding $\langle C_p, C_q \rangle$ requires expanding at least all p vertices in C_p or all q vertices in C_q. Thus m cannot be smaller than $\min(p - |C_p|, q - |C_q|) = dist(\langle V_p, V_q \rangle)$.□

A direct result from Lemma 2 and 3 is the following. A vertex with the minimum pattern distance is connected to a potential $K_{p,q}$ with minimum distance. Also, any vertex v that is connected to the potential $K_{p,q}$ with minimum distance, is the vertex with the minimum pattern distance. Thus, one can implement $Pattern^*$ for this pattern in a similar manner to the way described for the k-clique pattern: maintain all potential $K_{p,q}$, and in any iteration choose to expand one of the generated vertices that is connected to the potential $K_{p,q}$ with the smallest distance.

Let $\mathcal{PC}_{p,q}$ be the set of all potential $K_{p,q}$ (which is initially an empty set). Maintaining all the potential $K_{p,q}$ can be done in an incremental manner similar to Procedure IncrementalUpdate. When a vertex v is expanded, every potential $K_{p,q} = \langle V_p, V_q \rangle$ in $\mathcal{PC}_{p,q}$ is examined. Let $V'_p = V_p \cup \{v\}$ and let $V'_q = V_q \cup \{v\}$. If $\langle V'_p, V_q \rangle$ is a potential $K_{p,q}$ according to Corollary 2 add it to $\mathcal{PC}_{p,q}$. If $\langle V_p, V'_q \rangle$ is a potential $K_{p,q}$ according to Corollary 2 add it to $\mathcal{PC}_{p,q}$. If $\langle V_p, V_q \rangle$ is not a potential $K_{p,q}$ anymore - remove it from $\mathcal{PC}_{p,q}$. Finally, if $|gcn(\{v\})|$ is larger than p, add $\langle \{\}, \{v\} \rangle$ to $\mathcal{PC}_{p,q}$, and if $|gcn(\{v\})|$ is larger than q, add $\langle \{v\}, \{\} \rangle$ to $\mathcal{PC}_{p,q}$.

The purpose of the above discussion on the complete bipartite graph pattern is to demonstrate another pattern for which the concepts described for the k-clique pattern can be applied. In this paper we present experimental results (Section 8) only for the k-clique pattern.

6. Probabilistic Heuristic

The heuristic algorithms described in Section 5 assumed that nothing is known about the searched graph besides an initial vertex. However, there are many domains where the exact searched graph is unknown but some knowledge on the probability of the existence of edges is available. Formally, for every two vertices u and v, assume that a function $\hat{Pr}(u, v)$ is available that estimates the probability of having an edge between u and v. For example, if the searched graph is the World Wide Web then it is well-known that it behaves like a *scale-free graph* [38], which is a graph where the degree distribution of its vertices follows a power law (i.e., the probability of a vertex having a degree x is x^β for some exponent β). Furthermore, it is possible to classify web pages according to their URL [39,40]. Another example that is common in Robotics is a navigator robot in an environment represented by a graph. The robot may have an imperfect vision of the environment, wrongly identifying routes as passable with some probability [41]. Such a probabilistic knowledge should affect the choice of which vertex to expand next. In this section we present heuristics for such graphs.

6.1. MDP Approach

By adding probabilistic knowledge, the problem of searching for a pattern in an unknown graph can be modeled as a Markov Decision Problem (MDP) [42] as follows. The *states* are all possible pair combinations of (G_{known}, V_{gen}). The *actions* are the exploration actions applied to each vertex in V_{gen}. The *transition function* between two states (old and new states) is affected by the existence probability of edges that were added to G_{known} in the new state. Finally, the reward of every action is the negation of the exploration cost of the expanded node (in our case this is -1). A policy will be an algorithm for choosing which vertex to expand

next in every iteration of Algorithm 1. If one knows the number of vertices in the searched graph, or at least an upper bound to the number of vertices in the searched graph, then this MDP is a finite-horizon MDP, where the horizon is the given upper bound on the number of nodes in the searched graph. In such a case, an optimal policy could theoretically be computed, such that the expected cost would be minimized. By contrast, if one does not know anything about the number of vertices in the searched graph, then computing the optimal policy is problematic even in theory. Note that in the unknown graph model described in this paper, every expanded node incurs the same negative reward. Thus, a discounted infinite reward MDP model does not adequately fit our problem.

An alternative formulation of our problem is as a Deterministic Partially Observable MDP (DET - POMDP) [43,44], where the partially observable state is the entire searched graph (and not just G_{known} as in the MDP formulation described above) along with the list of explored nodes - (G, V_{exp}), and an observation is the known subgraph G_{known}. An action corresponds to expanding a single vertex. Note that in DET-POMDP, unlike standard POMDPs, the outcome of performing an action from a state is completely deterministic - in our problem this is simply adding the expanded vertex to V_{exp}. Furthermore, in DET-POMDP, an observation is a deterministic function of a state and the performed action. Similarly, in our problem, given a graph G and a set of expanded vertices V_{exp} the known subgraph (i.e., the observation) will be the subgraph of G that contains all the vertices and edges that are connected to at least a single vertex in V_{exp}.

The main problem in using an off-the-shelf MDP or POMDP algorithms is the number of possible states. Initially, only a single vertex is known. Since we do not know the number of vertices in the searched graph, the number of states is infinite. Thus it is impossible to explicitly store the entire belief state or enumerate all the states in the state space. If the number of vertices in the searched graph is known, then it is theoretically possible to employing an MDP or POMDP solver [45,46,47,48] in order to find the optimal policy, i.e., the policy that minimizes the expected exploration cost. Unfortunately, the size of the MDP state space grows exponentially with the number of vertices in the graph, as it contains all possible subgraphs of G. If the number of vertices in the unknown graph is n, then the number of states in the corresponding MDP will be $(n-1)! \cdot 2^{\binom{n}{2}}$: $(n-1)!$ for every possible order of expanding $n-1$ vertices, and $2^{\binom{n}{2}}$ for all the possible graphs with n vertices (an undirected graph with n vertices has at most $\binom{n}{2}$ edges). For example, a graph with 10 vertices requires $9!2^{45}$ MDP states. This combinatorial explosion prohibits any algorithm that requires enumerating a large part of the state space, as well as algorithms that require explicit representation of the belief state.

6.2. $RPattern^*$

In order to still exploit an available probabilistic knowledge, we propose a Monte-Carlo based sampling technique that combines sampling of the MDP state space with $Pattern^*$ as a default heuristic. We call this heuristic algorithm *Randomized* $Pattern^*$ or $RPattern^*$ in short. Like the previously presented heuristic algorithms, $RPattern^*$ is invoked when choosing which vertex to expand next (line 4 in Algorithm 1). The basic idea is to estimate for every generated vertex v the future exploration cost until the desired pattern is found by sampling the possible extensions of G_{known} in which v was expanded. $RPattern^*$ will then choose to expand the vertex with the lowest estimated future exploration cost.

Algorithm 3 presents the pseudo code for $RPattern^*$. $RPattern^*$ requires the following parameters: (1) $MaxDepth$, the maximum depth of every sample and (2) $NumOfSampling$, the number of samples to construct. Every vertex v in V_{gen} is assigned a value $Q[v]$, initialized by zero (line 2 in Algorithm 3). The value of $Q[v]$ is updated by the sampling process described next, and it is used eventually to estimate the cost of finding the desired pattern given that v is expanded next

The value of $Q[v]$ is updated as follows. In each sample, G'_{known} is initialized by G_{known} (line 4) and V'_{gen} is initialized by V_{gen} (line 5). Then, the outcome of expanding v is simulated (line 7) using the available probabilistic knowledge (i.e., the $\hat{Pr}$ function described earlier). G'_{known} and V'_{gen} are updated according to the outcome of the simulated exploration. Following, a vertex v' is chosen (from V'_{gen}) for simulated exploration using the $Pattern^*$ heuristic (line 9). The outcome of exploring v' is then simulated, again possibly adding new edges from v to other vertices (line 10) and updating G'_{known} and V'_{gen} accordingly. This process continues until either G'_{known} contains a subgraph that is isomorphic to G_P or after $MaxDepth$ iterations have been performed. If the desired pattern has been found after d simulated exploration actions, then $Q[v]$ is incremented by d (line 13) . Otherwise, $Q[v]$ is

Algorithm 3: $RPattern^*$

Input: G_P, The desired pattern
Input: $MaxDepth$, Max sample depth
Input: $NumOfSampling$, Number of samples
Output: The next vertex to expand

1 **foreach** v *in* V_{gen} **do**
2 $Q[v] \leftarrow 0$
3 **loop** $NumOfSampling$ **times**
4 $G'_{known} \leftarrow G_{known}$
5 $V'_{gen} \leftarrow V_{gen}$
6 $d \leftarrow 1$
7 $simulatedExplore(v, G'_{known}, V'_{gen})$
8 **while** $d<MaxDepth$ *And* $G_P \nsubseteq G'_{known}$ **do**
9 $v' \leftarrow chooseNext(V'_{gen})$
10 $simulatedExplore(v', G'_{known}, V'_{gen})$
11 $d \leftarrow d + 1$
12 **end**
13 $Q[v] \leftarrow Q[v] + d$
14 **if** *hasPattern*(G'_{known},k)=*False* **then**
 $Q[v] \leftarrow Q[v]$ + the pattern distance of v'
15 **end**
16 $Q[v] \leftarrow \frac{Q[v]}{NumOfSampling}$
17 **end**
18 return $argmin_{v \in V_{gen}} Q[v]$

incremented by $MaxDepth$ plus the *pattern distance* of the vertex chosen by $Pattern^*$ (line 14), which is the best lower bound of the remaining exploration cost (Theorem 1). All this (lines 4–14) is done for a single sample. The average value of $NumOfSampling$ samples is then stored in $Q[v]$ (line 16). This is repeated for all vertices in V_{gen} and the vertex with the smallest Q value is chosen for expansion by $RPattern^*$.[6] Note that the *pattern distance* of a vertex v is a lower bound on $Q[v]$, since the pattern distance is a lower bound on the cost of finding the desired pattern given that v is expanded next (Theorem 1).

An important part of $RPattern^*$ is how the outcome of an exploration is simulated (line 7 & 10). This greatly depends on the available probabilistic knowledge of the graph. As described above, in this paper we assume that the available probabilistic knowledge is the probability of any two generated vertices $u, v \in V_{gen}$ having an edge between them (denoted by $\hat{Pr}(u, v)$). By contrast, we do not assume any knowledge on vertices that have not been generated. We therefore take a myopic approach, considering only future connections between generated vertices, and no new vertices are added during a simulated exploration.[7] The simulated exploration of vertex v is therefore performed by adding an edge between v and any other generated vertex u with probability $\hat{Pr}(v, u)$.

Consequently, the computational complexity of the simulated exploration of vertex v is $O(|V_{gen}|)$. In the worst case, $RPattern^*$ performs $MaxDepth \times NumOfSampling$ simulated explorations. After every simulated exploration, $Pattern^*$ is executed to choose on which vertex to perform a simulated exploration next. Thus, the computational complexity of $RPattern^*$ is $MaxDepth \times NumOfSampling$ times $O(|V_{gen}|)$, plus $MaxDepth \times NumOfSampling$ times the computational complexity of $Pattern^*$.

In summary, $RPattern^*$ exploits the available probabilistic knowledge to generate samples that are extensions of G_{known} (line 7 and 10) and utilizes $Pattern^*$ for choosing the next vertex to explore during the sampling (line 9) and for estimating the exploration cost in case the maximum depth has been reached (line 14). Note that $RPattern^*$ can be easily implemented as an anytime algorithm: one can always add more samples in order to improve the estimated expected exploration cost. [8]

7. Theoretical Analysis

Next, we analyze the effect of the heuristic algorithms described in the previous sections for performing the *chooseNext()* action in Algorithm 1 on the exploration cost of searching for a pattern in an unknown graph. For pedagogical reasons we first focus on the k-clique pattern, and then generalizing to any pattern.

Algorithms for *chooseNext()* can be viewed as online algorithms. An online algorithm is an algorithm that must process each input in turn, without detailed knowledge of future input [49]. An offline algorithm for *chooseNext()* would be an algorithm that receives the entire searched graph G as input (and could potentially detect whether a k-clique exists) but

[6] Actually, dividing $Q[v]$ by $NumOfSampling$ is redundant, as this is constant for all the vertices. We leave this for clarity of presentation, to emphasize that $Q[v]$ is designed to estimate the expected exploration cost.

[7] If knowledge is available on the existence of new vertices, it can be easily incorporated into the simulated exploration.

[8] This requires a slight modification of Algorithm 3, mainly swapping lines 1 and 3 in the pseudo code.

is still required to choose vertices for exploration until G_{known} contains a k-clique. This is because the search only halts when G_{known} contains the desired pattern, and *chooseNext()* can only choose a generated vertex for exploration. An *optimal offline algorithm* for *chooseNext()* is an algorithm that if used in the best-first search described in Algorithm 1 will result in finding the searched pattern with minimum exploration cost. Using any other algorithm for *chooseNext()* will require expanding at least the same number of vertices as the optimal offline algorithm.

Let $d(u, v)$ be the length of the shortest path between u and v in the searched graph, and denote by $d(v, V')$ the distance between a vertex v and a set of vertices V', which is defined as $d(v, V') = \min_{u \in V'} d(v, u)$.

Theorem 2 *[Optimal offline algorithm] Let C be the k-clique in the unknown graph that is closest to s, where closeness is measured by $d(s, C)$. The optimal offline algorithm for chooseNext() will choose to expand the vertices that are on the shortest path from s to C, as well as $k-1$ members of C.*

Proof: Let C_k be the first k-clique that will be found using the optimal offline algorithm. It is easy to see that G_{known} contains the k-clique C_k only after $k-1$ members of C_k have been expanded. Since only vertices from V_{gen} may be expanded, C_k cannot be explored until $k-1$ of its members have been added to V_{gen}. With the exception of the initial vertex s, a vertex is inserted into V_{gen} only when one of its neighbors is expanded. Therefore, the minimal set of vertices that must be expanded until C_k is in G_{known} contains the shortest path from s to C_k plus $k-1$ members of C_k. Consequently, the minimal exploration cost will be achieved when C_k is the k-clique that is closest to s. □

Note that Theorem 2 and all the analysis in this section are for undirected graphs. Converting these theorems to directed graphs and pattern is simple. For example, a vertex cover of a k-clique in a directed graph contains all the vertices of that clique. Thus, there is a graph in which even the optimal offline algorithm will have to expand all the vertices in the graph before finding the searched clique.

Theorem 3 *[Exploration cost analysis] For any $k > 2$ the following statements hold:*

1. [Best case lower bound]: *There is no graph $G = (V, E)$ such that the optimal offline algorithm will find a k-clique after expanding less than $k-1$ vertices.*

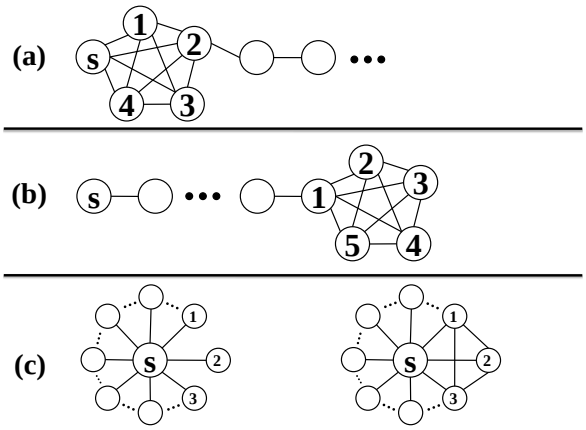

Figure 8. Scenarios for best and worst case exploration cost

2. [Best case upper bound]: *There exists a graph $G = (V, E)$ such that the optimal offline algorithm will find a k-clique after expanding exactly $k-1$ vertices.*
3. [Worst case upper bound]: *There is no graph $G = (V, E)$ such that the optimal offline algorithm will find a k-clique after expanding more than than $|V|-1$ vertices.*
4. [Worst case lower bound]: *There exists a graph $G = (V, E)$ such that the optimal offline algorithm will find a k-clique after expanding exactly $|V|-1$ vertices.*

Proof: We first prove the best case result. Consider a graph in which s is a part of a clique of the desired size, denoted by C_k. Figure 8(a) shows an example of such a graph for $k = 5$. If only $k-2$ vertices were expanded, then there are two vertices $v, u \in C_k$ that have not been expanded. Since neither of them has been expanded, the edge between them cannot be in E_{known} and thus the desired k-clique has not been found. On the other hand, if $k-1$ members of C_k have been expanded, all the edges and vertices of C_k are in G_{known} and therefore C_k is found.

Next, we prove the worst case results. After expanding $|V|-1$ vertices, all the edges and vertices in the graph have been discovered, and thus a k-clique will be found without expanding the last vertex. For proving the last statement, consider a graph that is composed of a chain of vertices starting from s and ending with a clique of the desired size, denoted by C_k. An example of such a graph is presented in Figure 8(b), where k=5. C_k is the only k-clique in G and is therefore the closest k-clique to s. Hence, according to Theorem 2 all the $|V|-k$ vertices in the chain must expanded by

the optimal offline algorithm, along with $k-1$ members of C_k. This totals to $|V|-1$ vertices that will be expanded by the optimal offline algorithm until the k-clique is found in G_{known} (causing the search to halt) □

Online algorithms are commonly analyzed using *competitive analysis* [50]. In competitive analysis we try to calculate (or at least estimate or bound) the *competitive ratio*. This is the maximal ratio over all possible inputs between the performance of the evaluated algorithm and the performance of the optimal offline algorithm. For the problem of finding a k-clique with minimum exploration actions, this corresponds to the maximal ratio between the exploration cost of an evaluated algorithm and the exploration cost of the optimal offline algorithm over all possible unknown graphs and initial vertices. Theorem 4 presents a lower bound of the competitive ratio of any algorithm for this problem.

Theorem 4 *[Competitive ratio of exploration cost] For any $k > 2$ and every deterministic online algorithm A for $chooseNext()$, there exists a graph $G=(V,E)$ for which the competitive ratio of A is at least $\frac{|V|-1}{k-1}$.*

Proof: Assume by negation that A is an algorithm that has a competitive ratio that is smaller than $\frac{|V|-1}{k-1}$. Let $G=(V,E)$ be a star-shaped graph without a k-clique, where all the vertices in the graph are connected only to s. The left graph in Figure 8(c) is an example of such a graph. Assuming $k>2$ then running algorithm A on G starting from s will expand all the vertices in V, until concluding that no k-clique exists. Let C_{k-1} be the last $k-1$ vertices chosen for exploration by A. Let G' be a graph that is identical to G except for having additional edges between all pairs of vertices in C_{k-1}. Thus, C_{k-1} form a k-clique with the initial vertex. The right graph in Figure 8(c) is an example of such a graph for k=4. G and G' provide exactly the same input to A until one of the last $k-1$ vertices is expanded. Thus if A is deterministic, it will choose to expand exactly the same $|V|-(k-1)$ vertices in G' as it chose for G until one of the vertices in C_{k-1} is expanded. Then, A will have to expand at least $k-2$ vertices from C_{k-1} (as s has already been expanded) until finally exploring the k-clique, totaling in $|V|-1$ vertices expanded until the k-clique has been found. On the other hand, the optimal offline algorithm will only expand the $k-1$ members of the clique. Thus, the competitive ratio of algorithm A is at least $\frac{|V|-1}{k-1}$, contradicting the assumption that A has a competitive ratio lower than $\frac{|V|-1}{k-1}$. □

7.1. Generalizing to Any Given Pattern

The above theorems can be generalized to any specific pattern $G_P=(V_P,E_P)$, using the following supporting claims.

Lemma 4 *G_{known} contains a subgraph C_P that is isomorphic to G_P iff V_{exp} contains a (not necessarily minimal) vertex cover of C_P.*

Proof: ($\Leftarrow$) Let C' be a vertex cover of C_P that has been expanded. By definition of $Explore()$ (Definition 1), all the connecting edges and neighboring vertices of members C' have been inserted to G_{known}. Since C' is a vertex cover of C_P, then all its edges and vertices are in G_{known}.

($\Rightarrow$) Assume by negation that V_{exp} does not contain a vertex cover of C_P. Then there exists an edge (u,v) in C_P such that u and v are not in V_{exp} (i.e., have not been expanded yet). This means that (u,v) cannot be in G_{known}, contradicting the definition of C_P □

Note that a vertex cover of the k-clique pattern is any $k-1$ vertices of the clique. Therefore, as stated in Theorem 2, even the optimal offline algorithm requires exploring at least $k-1$ vertices until a k-clique is found.

For ease of notation we use the term "a vertex cover of G_P in G" to denote a vertex cover of a subgraph of G that is isomorphic to G_P.

Definition 10 *[Smallest connected subgraph with a vertex cover]*
Let $minConnectedVC(s,G,G_P)$ be a function that returns the smallest connected subgraph of G (in terms of number of vertices) that contains the vertex s and contains a vertex cover of G_P in G.

Corollary 3 extends the result from Theorem 2 to the general case of searching for any pattern.

Corollary 3 *[Any pattern optimal offline algorithm]*
The optimal offline algorithm will choose to expand only the vertices in $minConnectedVC(s,G,G_P)$.

Proof: First we prove that there is an algorithm that expands only the vertices in $minConnectedVC(s,G,G_P)$ and finds the desired pattern. Then we prove that no algorithm can find the desired pattern without expanding at least the same number of vertices. Let $G'=(V',E')=minConnectedVC(s,G,G_P)$. Since V' contains a vertex cover of G_P, then after

expanding the vertices in V', a subgraph of G_{known} that is isomorphic to G_P has been found (Lemma 4). Since G' is connected and contains s, all the vertices in G' can be expanded in a breadth-first order, starting from s. This expansion order ensures that a vertex is expanded only after it has been previously generated. Therefore there is an algorithm that finds the desired pattern by choosing to expand only the vertices in G'.

Assume by negation that the optimal offline algorithm finds the desired pattern by expanding the set of vertices V'', such that $|V''| < |V'|$. Recall that G' is the minimal connected graph that contains the initial vertex s and a vertex cover of G_P. Clearly, any algorithm will expand the initial vertex s. Thus, either V'' does not contain a vertex cover of G_P in G, or there is no connected subgraph of G that contains only the vertices in V''. If V'' does not contain a vertex cover G_P in G, then after expanding only the vertices in V'', the known subgraph does not contain an isomorphic subgraph of G_P (Lemma 4). This contradicts the fact that the desired pattern has been found after expanding only the vertices in V''. On the other hand, assume that V'' does contain such a vertex cover but there is no connected subgraph in G that contains only the vertices in V''. Then there are vertices in V'' that cannot be expanded without previously expanding a vertex that is not in V''. It is easy to show that a vertex cannot be chosen for expansion if there is no path from s to it that contains only expanded vertices. Thus it is not possible to expand only the vertices in V'', resulting in a contradiction.

Concluding the proof, there is an algorithm that finds the desired pattern by expanding only the vertices in $minConnectedVC(s, G, G_P)$, and no algorithm can find the desired pattern by expanding less vertices. Thus it is optimal $\square$.

The worst case and best case results for the k-clique pattern shown in Theorem 3 can be seen as a special case of Corollary 3. As explained above, a vertex cover of a k-clique is any $k-1$ vertices from that clique. When there is a k-clique connected to the start state, $minConnectedVC(s, G, G_P)$ will contain exactly $k-1$ vertices. This corresponds to the best case result stated in Theorem 3. On the other hand, if the searched graph contains only a single k-clique that is $|V|-k$ vertices away from the initial vertex s, then $minConnectedVC(s, G, G_P)$ will contain exactly $|V|-1$, corresponding to the worst case results in Theorem 3.

Consequently, the best-case result given for k-cliques in Theorem 3 can be generalized as follows.

Theorem 5 *[Exploration cost best-case, searching for any pattern] For any pattern G_P there exists an unknown graph $G = (V, E)$ such that the optimal offline algorithm will expand the number of vertices that is equal to minimal vertex cover of G_P in the best case.*

Proof: The best case scenario occurs when s is a part of the minimal vertex cover of G_P in G and has edges to all the other vertices in the minimal vertex cover. The optimal offline algorithm will then expand s and only the vertices in the minimal vertex cover, until the desired pattern is found. Since the desired pattern cannot be found without expanding a vertex cover of G_P (Lemma 4) then this is the best case.$\square$

The best-case analysis above is correct for any pattern. Next, we generalize the worst-case and competitive-ratio results from Theorems 3 and 4 from the k-clique case to any pattern. This is done for every pattern that contain at least two vertices with degree higher than two. We call patterns that do not fulfill this requirement *degenerated* patterns, and do not discuss them in this paper.

Theorem 6 *[Exploration cost worst-case, searching for any pattern] For any non-degenerated pattern G_P, the following claims holds for the optimal offline algorithm:*

1. *For any unknown graph that contains G_P, the optimal offline algorithm will find the searched pattern after expanding $|V| - (|G_P| - |minConnectedVC(v, G_P, G_P))$ vertices.*
2. *There exists an unknown graph $G = (V, E)$ that contains G_P, where the optimal offline algorithm will find the searhced pattern only after expanding $|V| - (|G_P| - |minConnectedVC(v, G_P, G_P))$ vertices.*

Proof Since G contains G_P, then after expanding $|V| - |G_P|$ vertices at least one vertex from G_P is generated. Following, the optimal offline algorithm will expand the minimal vertices that are required to find G_P. It is easy to see that this is exactly the vertices in $minConnectedVC(v, G, G_P)$. Thus, the optimal offline algorithm will never expand more vertices than $|V| - |G_P| + minConnectedVC(v, G, G_P)$. Since G_P is a subgraph of G, the size of $minConnectedVC(v, G, G_P)$ cannot be larger than the size of $minConnectedVC(v, G_P, G_P)$. Thus, the optimal offline algorithm will never expand more vertices than $|V| - (|G_P| - |minConnectedVC(v, G_P, G_P))$.

Notice that size of $minConnectedVC(v, G_P, G_P)$ depends on v, and can be different for the different vertices in G_P. For example, consider a pattern of a star (one vertex in the middle, with many neighbors), displayed in the left part of Figure 8(c). If v is the vertex in the middle (vertex S in Figure 8(c)), then $|minConnectedVC(v, G_P, G_P)| = 1$, containing only S. By contrast, if v is not the vertex in the middle, then $|minConnectedVC(v, G_P, G_P)| = 2$, containing v and the vertex in the middle S. Thus, the optimal offline algorithm will never expand more vertices than $|V| - (|G_P| - \max_{v \in G_P} |minConnectedVC(v, G_P, G_P)|)$. This proves the first claim in the theorem above.

We prove the second claim by showing that such a worst case scenario exists. Consider a graph that is similar to the graph presented in Figure 8(b). The graph is composed of a chain of vertices starting from S and ending with the desired pattern. The desired pattern can be connected to the chain via any of its vertices. In this worst case graph, the desired pattern will be connected to the chain via the vertex that will maximize the exploration cost of the optimal offline algorithm. Thus only $|G_P| - \max_{v \in G_P} |minConnectedVC(v, G_P, G_P)|$ will not be expanded □

Using a straightforward generalization of the proof of Theorem 4, it is possible to conclude that any algorithm cannot achieve a competitive ratio that is better (i.e., lower) than $\frac{|G|-|G_P|}{|G_P|}$ for any non-degenerated pattern.[9] As described in Theorem 4 for the specific patten of a k-clique, the competitive ratio is $\frac{|V|-1}{k-1}$. This is even higher than $\frac{|V|-k}{k}$ described in Theorem 4, because k is at least one and $|V| \geq k$.

When the size of the searched graph is much larger than the size of the pattern graph, then the upper bound for the competitive ratio of any algorithm is approximately $\frac{|G|}{|G_P|}$. Note that an algorithm that chooses which vertex to expand randomly has approximately the same competitive ratio of $\frac{|G|}{|G_P|}$. Hence, one might presume that all algorithms are as effective in finding the searched pattern as random exploration. However, this analysis is based on a worst case scenario. Next, we demonstrate experimentally that the heuristic algorithms presented in Sections 5 and 6 require significantly less exploration cost than random exploration in various settings.

[9] Note that the trivial patterns such as a chain of vertices are *degenerated* patterns, which we do not discuss in this paper.

8. Experimental Results

We empirically compared the different heuristic algorithms presented in this paper for the k-clique pattern by running experiments on various graphs. We chose to experiment on the k-clique pattern because it is a well known pattern in the computer science literature. In every experiment the searched graph was constructed according to the following parameters: (1) the graph structure (random or scale-free), (2) the number of vertices in the graph (100,200,300 and 400), (3) the initial vertex and (4) the size of the desired clique (5,..,9). We verified that the constructed graph contained a clique of the desired size. If it did not, a new graph was generated. We chose to experiment on relatively small cliques (i.e. the size of the desired clique is substantially smaller than the size of the searched graph), as we are interested in scenarios where the k-clique can be found without exploring almost the entire unknown graph.

The performance of the different heuristic algorithms was evaluated by running them on the constructed graphs, and comparing the exploration cost required until the desired pattern was found. For comparison reasons, we also ran the following algorithms:

- **Random.** An algorithm in which the next vertex to expand is chosen randomly from V_{gen}. This algorithm serves as a baseline for comparison.
- **Lower bound.** The optimal offline algorithm described in Section 7. This algorithm is used as a lower bound on the exploration cost of the optimal algorithm, since no algorithm can do better than the optimal offline algorithm.
- **RLS-LTM**. An adaptation of a clique search algorithm from the known graph setting to the unknown graph setting. This algorithm is described next.

8.1. RLS-LTM

The state-of-the-art algorithms for finding the largest clique in a known graph are based on local search [30,31,32,33]. These algorithms begin with a trivial clique containing a single vertex that is chosen randomly. Then, an iterative improvement process begins, in which the *current clique* is extended by adding a vertex that is connected to all other vertices in the current clique. This process continues until it is not possible to extend the clique any further, i.e., there is no vertex in the graph that is connected to all the ver-

tices in the current clique. Then a single vertex is removed from the current clique, possible allowing further extensions of the new current clique (without the removed vertex). After performing this process several times the search restarts, discarding the current clique and choosing a different initial vertex from which the search continues. The various local search algorithms differ by the policy they employ to choose which vertex to remove or add, and by the policy they employ in choosing when to restart the search.

Recently, it has been shown empirically that for random graphs and scale-free graphs the best algorithm in this local search framework is Reactive Local Search with Long Term Memory (RLS-LTM) [33]. In RLS-LTM, whenever a vertex is removed from the current clique, it is *prohibited* from being added to the current clique for the next T iterations. T is a parameter adjusted during the search reactively: T is increased whenever the current clique has already been visited, and decreased when the current clique is a new clique. This requires storing all cliques visited throughout the search (this is the long term memory of RLS-LTM). Among the vertices that are not prohibited, RLS-LTM chooses to add to the current clique the vertex that has the highest degree. If the size of the current clique does not increase within a A number of iterations, the search restarts, where A is a parameter. The parameter A was set to be 100 times the size of the largest clique found so far, following [33,31].

We have adapted the RLS-LTM clique algorithm to the unknown graph setting as follows. The current clique is initialized with the initially known vertex s. Vertices are added and removed according to the RLS-LTM algorithm, and when a generated vertex is chosen to be added to the current clique, it is first expanded (incurring an exploration cost). Furthermore, for vertices that have not been expanded yet the known degree is used instead of the actual (but unknown) degree, which is used for tie-breaking in RLS-LTM.

Note that there are two shortcomings for using RLS-LTM in the unknown graph setting:

1. **RLS-LTM is not complete.** RLS-LTM is a local search, and it is therefore not complete. In other words, a k-clique might exists in the graph and RLS-LTM will not find it. Due to the prohibition mechanism used by RLS-LTM, this rarely occurs for small cliques. This can be remedied by forcing RLS-LTM to expand a generated vertex after a fixed number of restarts. Thus all the unknown graph will eventually be expanded and the search will halt.
2. **RLS-LTM focuses on runtime.** RLS-LTM was developed to find cliques fast, not to reduce the number of vertices encountered (=explored) during the search. Thus it ignores the distinction between expanded and generated vertices.

Nonetheless, we provide experimental results for this algorithm as well.

8.2. Evaluating the Deterministic Heuristics

First, we have evaluated the proposed deterministic algorithms on random graphs. In random graphs the probability that an edge exists between any two vertices in the graph is constant. These graphs have been extensively used in computer science research as an analytical model and as benchmarks for evaluating algorithms efficiency [51,52,53]. In our experiments we generated random graphs as follows. Let n be the number of vertices in the graph, and k be the size of the desired clique. First, a graph with n vertices was generated. Then, edges were added between random pairs of vertices, until a given number of edges were added. The number of edges that was added to the graph was calculated such that the expected number of k-cliques in the graph would be one. This can be easily calculated by the linearity of expectation [35]. If no k-cliques existed in the generated graph, the graph was discarded. This process ensures that the graph contains a k-clique, but the expected number of k-cliques in the graph is not large, making the search for a k-clique more challenging.[10]

In Figure 9 we compare the average total exploration cost of searching for a 5-clique using the heuristic algorithms, $KnownDegree$ RLS-LTM and $Clique^*$ on random graphs. Every data point is the average over 50 random graphs generated as described above. The x-axis shows the number of vertices in the graph and the y-axis shows the average exploration cost, i.e., the number of vertices expanded until a clique of the desired size was found. The black brackets denote an error bar of one standard deviation.

The figure shows that all the heuristic algorithms significantly outperform the random approach. This improvement grows with the size of the graph but the difference between the lower bound (computed by the optimal offline algorithm described in Theorem 2) and the total exploration cost of all of the heuris-

[10] In a graph with many k-cliques, finding a k-clique is easy, and as a result it would have been harder to distinguish between the performance of the different algorithms.

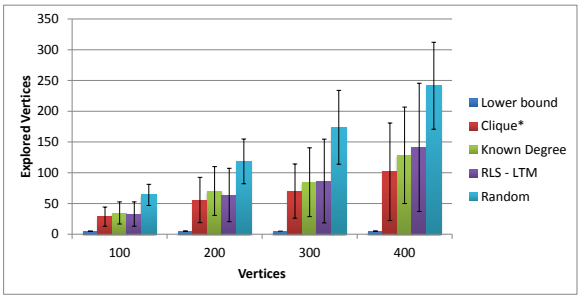

Figure 9. Non probabilistic heuristic algorithms on random graphs.

tic algorithms also increases. Another observation is that $Clique^*$ is more effective than $KnownDegree$ and RLS-LTM for random graphs by up to 20%. This difference also increases as the number of vertices in the graph grows.

Although the focus of the algorithms proposed in this paper is to minimize the exploration cost, we provide runtime results as well to demonstrate the feasibility of the proposed algorithms. Table 1 displays the average runtime in milliseconds until a k-clique was found in random graphs in the same experiment set described for Figure 9. The values in the *Total* column are the average total runtime in milliseconds and the values in the *Vertex* column are the average runtime per vertex, also measured in milliseconds. As can be seen, all algorithms found the k-clique under four seconds. Note that the reported runtime is for all the steps of our best-first search algorithm (Algorithm 1) including the *test()* step, where a search for the desired pattern is performed in the known subgraph. Indeed, searching for a 5-clique in a graph with several hundreds of vertices can be done very efficiently.

The runtime complexity per expanded vertex of $KnownDegree$ as well as random exploration is very small (see Section 5), and thus the total runtime as seen in Table 1 is small (less than 300 milliseconds). On the other hand, the runtime of RLS-LTM is relatively large (over a second for random graphs with 300 and 400 vertices). In every iteration of RLS-LTM, a local search is performed in the known subgraph, and a vertex is chosen for exploration only when a generated vertex is added to the current clique (see Section 8.1 for details). Consequently, the runtime until RLS-LTM chooses which vertex to expand next is larger than that of all the other heuristic algorithms. Although in a worst case analysis, the runtime of $Clique^*$ is large (explained in Section 5.2.1), under these settings it is the fastest. This is because the actual number of potential k-clique, which dominated the runtime of $Clique^*$, is in practice much smaller than the worst case number (which is exponential in the size of the searched clique).

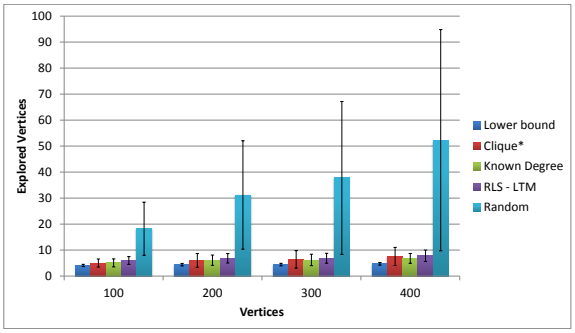

Figure 10. Non probabilistic heuristics on scale-free graphs.

The second set of experiments were performed on scale-free graphs. In scale-free graphs the degree distribution of the vertices in the graph can be modeled by power laws, i.e. $Pr(degree(v) \geq x) = x^{-\beta}$ for some $\beta > 1$. Many networks, including social networks and the Internet exhibit a degree distribution that can be modeled by power laws [54]. Since one of the domains which we are interested is the Internet, it is natural to run experiments on this class of graphs as well. A number of scale-free graph models exist [55,56,57,54]. We chose a simple scale-free graph generator model [58] requiring two parameters: (1) the number of vertices n and (2) the number of edges m. According to this model, a graph is generated in two stages. First, a connected graph with n vertices is generated. This is done incrementally, starting with an empty graph and adding vertices one at a time. A new vertex v is added to the graph by connecting it to an old vertex u that is selected with probability proportional to its degree. Then, after all the vertices have been added to the graph, m edges are added by selecting a vertex at random and connecting it to a vertex that is also selected with probability proportional to its degree.

Figure 10 presents results obtained by performing a set of experiments with scale-free graphs under settings similar to those of the random graph experiments described above. Two interesting phenomena can be observed. First, the improvement in the exploration cost of all the heuristic algorithms over the random approach significantly grows when the size of the graph increases. The improvement is more than a factor of 6 in graphs with 400 vertices. Second, according to a paired t-test, $KnownDegree$ outperforms RLS-LTM on all sizes of graphs, and it is even significantly better

Vertices	$Clique^*$		$KnownDegree$		$RLS\text{-}LTM$		Random	
	Total	Vertex	Total	Vertex	Total	Vertex	Total	Vertex
100	11	0	17	0	112	3	20	0
200	49	1	62	1	596	7	86	1
300	72	1	115	1	1,271	10	204	1
400	180	2	256	2	3,491	17	377	2

Table 1
Runtime in milliseconds, on random graphs.

(p-value < 0.1) than $Clique^*$ on graphs with 400 vertices. Moreover, the average exploration cost of these three algorithms ($Clique^*$, $KnownDegree$ and RLS-LTM) is almost the same, and very close to the lower bound (calculated by the optimal offline algorithm described in Theorem 2).

The improved performance of $KnownDegree$ and RLS-LTM in scale-free graphs can be explained as follows. In scale-free graphs a vertex is more likely to be connected to a vertex with a high degree (this is known as *preferential attachment*). Thus, vertices with high known degree are more likely to be connected to other generated vertices or to new vertices. Vertices with high known degree are chosen by $KnownDegree$ by definition. Furthermore, vertices with high known degree are more often considered in RLS-LTM, since such vertices are connected to more vertices than vertices with low known degree. These two arguments explain the improved performance of $KnownDegree$ and RLS-LTM on scale-free graphs in comparison with the results of $KnownDegree$ and RLS-LTM on random graphs.

In terms of runtime, all the algorithms expect random exploration found a 5-clique under 100 milliseconds. Furthermore, the differences between the runtime of the algorithms were insignificantly small. We therefore omit these results. This is reasonable, since as seen in Figure 10 all the heuristic algorithms (except random exploration) found the desired 5-clique with a very small number of exploration actions - almost equal to the optimal offline algorithm.

Concluding, on both random and scale-free graph we have seen that all the heuristic algorithms ($Clique^*$, $KnownDegree$ and $RLS\text{-}LTM$) outperform random exploration significantly. On random graphs $Clique^*$ is superior to all of the other algorithms in terms of exploration cost, while in scale-free all the heuristic algorithms required very similar exploration cost, with a slight advantage for $KnownDegree$.

8.3. A Real Domain of Unknown Graphs from the Web

We have also evaluated the deterministic heuristic algorithms on a real unknown graph - the World Wide Web. This was done by implementing an online search engine designed to search for a k-clique in the web. Specifically, we implemented a web crawler designed to search for a k-clique in academic papers available via the *Google Scholar* web interface (denoted hereafter as GS). Each paper found in GS represents a vertex, and citations between papers represent edges. We call the resulting graph the *citation web graph*. Naturally, the connection in context between papers is bidirectional (although two papers can never cross cite each other), thus we model the citation web graph as an undirected graph. The motivation behind finding cliques in the citation web graph is to find the relevant significant papers discussing a given subject (as discussed in the introduction). This can be done by starting the clique search with a query of the name of the desired subject or term. In our experiments we used computer science related terms (e.g., "Subgraph-Isomorphism", "Online Algorithms").

The web crawler we implemented operates as follows. An initial query with a name of a subject or a scientific term is sent to GS. The result is parsed and a list of hypertext links referencing academic work in GS is extracted. The crawler then selects which link to crawl next, and follows that link. The resulting web page is similarly parsed for links. This process is repeated, allowing the web crawler to explore more and more parts of the citation web graph. Figure 11 shows an example of a citation web graph generated by a random crawl, starting with a GS query of "Sublinear Algorithms".

In order to gather descriptive statistics of this process, 25 graphs were generated by the process described above, starting from queries of 25 different computer science topics and generating 25 corresponding citation web graphs. As could be expected, the distribution of node degrees in the graphs followed a power law distribution. Interestingly, many of the cita-

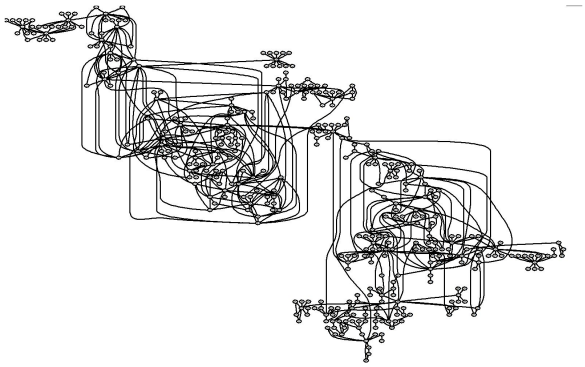

Figure 11. Citation web graph from a random walk in GS.

tion web graphs contained 4-cliques (95% out of the 25 generated web graphs) and 5-cliques (70% of the generated web graphs). On the other hand, very few (only 25% of the generated web graphs) contained larger cliques. Note that every random walk was halted after exploring several hundreds of vertices, and larger cliques may be found by further exploration. In addition, we measured the runtime used by every exploration, for over 2,500 random explorations of web pages in GS. The exploration of a vertex included sending an HTTP request, waiting for the corresponding HTML page to return and parsing it. Figure 12 presents the histogram of the runtime required for exploring a single vertex, grouped into bins of 200 milliseconds. As can be seen over 50% of the explorations are in the same runtime bin. Moreover, 90% of the explorations are performed in the range of the bins 0.6 and 0.8. This result agrees to some extent with our simplifying assumption of a constant exploration cost.

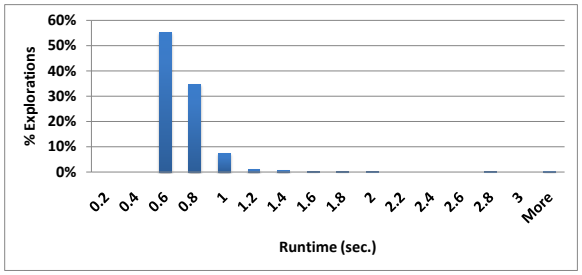

Figure 12. Runtime of exploring a web page

For finding a k-clique in the citation web graph, we implemented a best-first search (as described in Algorithm 1) on top of our web crawler as follows. The initial vertex s is an initial query that will be sent to GS. The $Explore()$ action (line 5 in Algorithm 1) consists of sending a query to GS and extracting from the

k	Random	$KnownDegree$	$Clique^*$
4	15	14	22
5	0	6	17

Table 2

Number of instances where the desired clique was found.

Algorithm	Cost	Runtime	Time per vertex
$Clique^*$	10.1 (7.5)	11.3 (6.4)	1.3 (0.4)
$KnownDegree$	27.8 (30.5)	15.4 (13.4)	0.8 (0.3)
Random	43.7 (27.7)	41.0 (26.4)	1.0 (0.2)

Table 3

Online search for a 4-clique in the GS web citation graph.

HTML of the resulting web page the list of hypertext links that references academic work. Each new link is added to G_{known} as a new generated node. The crawler then selects which link to crawl next (line 4 in Algorithm 1), and follows that link. The resulting web page is similarly parsed for links. This process is repeated until a k-clique is found (line 3 in Algorithm 1) or after 100 web pages were explored. The number of explorations was limited to 100 in order to prevent the crawler from being classified as a "denial of service" (DOS) attack (and as a result be blocked from access to the web page).

Table 2 presents the results of 22 online GS web crawls, performed using the $KnownDegree$ and $Clique^*$ heuristic algorithms, as well as the random baseline. As mentioned above, we used computer science related terms to start the crawl in GS. The values in the column *k* represents the size of the desired clique. The values in the other column represent the number of instance where the desired clique was found before reaching the exploration limit of 100 described above. As can be seen in Table 2, with $Clique^*$ the desired clique was found in substantially more instances than both $KnownDegree$ and random exploration. For example, with $Clique^*$ a 5-clique was found in almost three times more instances than $KnownDegree$ (17 vs. 6), and random exploration did not manage to find a 5-clique in any of the instances (before reaching the exploration limit).

Table 3 presents detailed results for the instances where all the algorithms have successfully found the desired clique. There were 9 such instances (instances that were solved by random, $KnownDegree$, and $Clique^*$), and the desired clique size in all these instances was 4. The *Cost* column represents the average number of vertices expanded until the desired clique was found and the '*Runtime*' column represents the av-

erage runtime in seconds. The *Time per vertex* column displays the average number of seconds required to explore a vertex. The values in brackets in each column are the standard deviation.

As can be seen in Table 3, $Clique^*$ requires expanding significantly fewer vertices than both $KnownDegree$ and $Random$. For example, finding a 4-clique requires expanding an average of only 10.11 vertices using $Clique^*$, which is 4 times less than the average number of vertices required random, and more than two times less vertices than required by $KnownDegree$. The difference in runtime between $KnownDegree$ and $Clique^*$ is smaller than the difference in cost between $KnownDegree$ and $Clique^*$ (11.33 for $Clique^*$ vs. 15.44 for $KnownDegree$). This is due to the larger overhead per vertex required by $Clique^*$, as demonstrated in the *Time per vertex* column. When searching for a 4-clique, the average number of seconds required to explore a vertex was 0.81 for $KnownDegree$ while it was 1.29 for $Clique^*$. This corresponds to the complexity analysis given in Section 5.1 and Section 5.2.1: $KnownDegree$ requires only $O(log(|V_{gen}|))$ operations to choose the next vertex to expand, while $Clique^*$ requires higher computational effort, due to the overhead of maintaining the set of potential k-cliques.

Surprisingly, the average runtime per vertex of $KnownDegree$ is even smaller than $Random$. This is counter-intuitive, as the complexity of choosing the next vertex to expand in $Random$ is $O(1)$. However, this can be explained as follows. In every iteration of the search (Algorithm 1), we first *test* if the searched pattern has been found in the known subgraph (G_{known}), and then run a heuristic algorithm for choosing the next vertex to expand. As more vertices are expanded, G_{known} grows, demanding more time to test if the searched pattern has been found. Since $KnownDegree$ finds a k-clique by expanding less vertices than $Random$, the resulting runtime per vertex of $KnownDegree$ is slightly smaller than $Random$.

In summary, although all the proposed heuristic algorithms are similar in terms of theoretical competitive ratio (Theorem 4), they performed much better than random exploration in all our experimental settings. We have seen as well that $Clique^*$ outperforms all the other algorithm in terms of exploration cost, except for in scale-free graphs, where all the heuristic algorithms exhibited similar performance with a slight advantage for $KnownDegree$. Furthermore, the runtime of $Clique^*$ has always remained feasible.

8.4. Simulated Graphs with Probabilistic Knowledge

$RClique^*$ assumes that probabilistic knowledge about the existence of an edge connecting two vertices in V_{gen} is available. This knowledge is used by $RClique^*$ when simulating the outcome of expanding vertices. To empirically evaluate $RClique^*$ we simulated this probabilistic knowledge as follows. Let $noise$ be a real number in the range $[0, 1]$. For a graph $G = (V, E)$ we define the following function:

$$P(e) = \begin{cases} 1 - rand(0, noise) \; if & e \in E \\ rand(0, noise) & if \quad e \notin E \end{cases}$$

In our experiments $RClique^*$ uses $P(e)$ when performing simulated exploration, assuming that an edge $e \in V \times V$ exists with probability $P(e)$.

Consider the effect of the $noise$ parameter. If $noise = 0$ then $RClique^*$ is given exact knowledge about all edges in G. That is, $P(e) = 1$ for all existing edges and $P(e) = 0$ for edges that do not exist in the real searched graph. In this case a simulated exploration of vertex v will reveal all the real edges connecting v to the other generated vertices. By contrast, if $noise = 1$ then the probabilistic knowledge given to $RClique^*$ is completely random. That is, $P(e)$ assigns random values for every possible edge (whether it exists or not). If $noise$ is between these two extremes then edges that do exist in the real searched graph are assigned high probabilities while edges that do not exist are assigned low probabilities. For example if $noise = 0.5$ then existing edges are assigned $P(e)$ from the range $[0.5, 1]$ and non-existing edges are assigned $P(e)$ from the range $[0, 0.5]$. We performed experiments with different levels of uncertainty by using different values of $noise$.

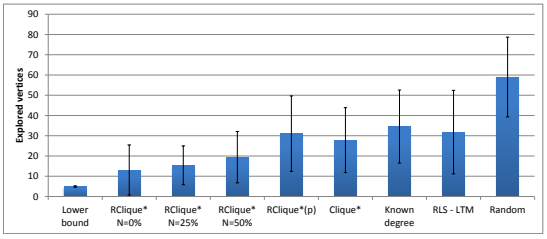

Figure 13. Random graphs, various levels of noise.

In the first set of experiments we compared $RClique^*$ with various levels of noise to the non-probabilistic approaches. Figure 13 shows the average exploration cost of searching for a 5-clique in random graphs with 100 vertices, generated as described

Max depth	1	2	3	5	9
Exploration cost	20.02	18.48	16.56	16.8	16.84
Runtime	157	215	240	331	409
Runtime per vertex	6	9	11	16	19

Table 4
Exploration cost and runtime, different max sample depth.

in Section 8.2. Every data point is the average over 50 randomly generated graphs.The bars denote the different heuristics, including (1) random exploration, (2) $KnownDegree$, (3) $RLS\text{-}LTM$, (4) $Clique^*$, (5) $RClique^*$ with various settings, and (6) the lower bound provided by the optimal offline algorithm. In all $RClique^*$ settings we set $NumOfSampling$ to 250 and $MaxDepth$ to 3, which we have found empirically to be effective. $RClique^*$with N=0%, 25% and 50% represents $RClique^*$ with various levels of noise $(0\%, 25\%, 50\%)$. $RClique^*$(p) denotes $RClique^*$ which is only given the average vertex degree in the graph. Therefore, in the simulated explorations $RClique^*$(p) assumes that the probability of having an edge between any two vertices is the average vertex degree in the graph divided by the number of vertices in the graph.[11]

As expected, $RClique^*$ with smaller noise (i.e., more precise knowledge) achieves a lower exploration cost. However, even with a setting of 50% noise $RClique^*$ outperforms $Clique^*$ significantly.[12] By contrast, $RClique^*$(p) expands approximately the same amount of vertices as $Clique^*$ (p-value of 0.15 according to a paired t-test). This suggests that knowing the average vertex degree does not contain enough probabilistic knowledge to be exploited by $RClique^*$.

Next, we evaluated the effect of different sampling depths ($MaxDepth$) of $RClique^*$ on the total exploration cost. Table 4 presents results for $RClique^*$ with a maximal sampling depth of 1,2,3,5 and 9, on the same set of random graphs described above, having $NumOfSampling$ set to 250 and 50% noise. The *Exploration cost* row presents the average exploration cost, i.e., number of vertices expanded until the desired clique was found. The *Runtime (sec.)* row presents the average runtime in seconds, and the *Runtime per vertex (sec.)* row presents the average runtime per vertex in seconds. These results indicate that deeper sampling reduces the exploration cost of the search down to some limit, after which the exploration cost remains approximately the same. This diminishing return effect is important, as deeper sampling requires higher runtime computational effort. Another observation with respect to the runtime results, is that the runtime of $RClique^*$ is much higher than the runtime needed for the deterministic algorithms (see Table 1). This is reasonable, as $RClique^*$ calls $Clique^*$ many times during the search to evaluate leaf nodes of the sampling (see Algorithm 3).

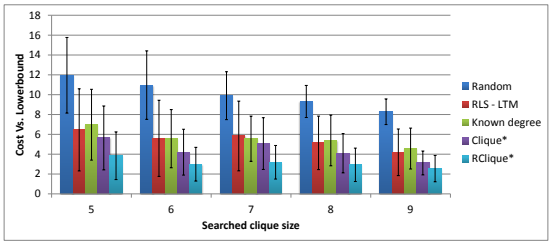

Figure 14. Random graphs, different desired clique size.

Figure 14 presents the exploration cost of the different heuristic algorithms when searching for different sizes of cliques (5,6,7,8 and 9) in random graphs. As explained in the beginning of Section 8.2, the graphs were generated such that the expected number of cliques of the desired size is one, and experiments were run only on graphs with at least one clique of the desired size. We set the sampling depth, $NumOfSampling$ and noise parameters of $RClique^*$ to be 3, 250 and 50% respectively.

The x-axis of Figure 14 denotes the size of the searched clique, and the y-axis represents the exploration cost normalized by the exploration cost of the optimal offline algorithm. As can be seen, all heuristic algorithms require a substantially lower exploration cost than random exploration, $Clique^*$ is the best deterministic algorithm, and $RClique^*$ is better than $Clique^*$. This shows that the same trends observed when searching for a 5-clique are kept for larger cliques.

An additional trend that can be observed is that the difference between the performance of all the algorithms is narrowed as the size of the searched clique grows. This is explained as follows. As the size of the searched clique grows, the number of vertices expanded by the optimal offline algorithm also grows. Exploring a 5-clique requires expanding at least 4 vertices, while more than twice are required for finding a 9-vertices. By contrast, random exploration required

[11] The average vertex degree in a random graph is simply the number of edges divided by the number of vertices.

[12] Note that even with %0 noise, $RClique^*$ is still not equivalent to the optimal offline algorithm. This is because unlike the optimal offline algorithm $RClique^*$ only considers the vertices already found (i.e., expanded vertices or neighbors of expanded vertices).

expanding 59 vertices on average until a 5-clique was found. Since the graph size in this experiment was 100, then random exploration will find a 9-clique without expanding twice as many vertices as needed for finding a 5-clique. The same logic applies to all the other heuristic algorithms. Also, as explained previously, the graphs were generated such that the expected number of cliques of the searched size is one. Thus the "distribution" of cliques of the same size will tend to stay the same for the various clique sizes, under this setting.

Table 5 presents average runtime results corresponding to the same set of experiments described above, of searching for cliques of sizes 5,6,7,8 and 9 in random graphs with 100 vertices. The values in the *Total* column are the average total runtime in seconds and the values in the *Vertex* column are the average runtime per vertex, also measured in seconds. We omit the runtime results for k=5 as it was presented in Table 1 and Table 4. Again, we set the sampling depth, $NumOfSampling$ and noise parameters of $RClique^*$ to be 3, 250 and 50% respectively.

While the runtime of all the algorithms increase when searching for larger cliques, all the deterministic algorithms have found the desired clique in less than a second on average. On the other hand, the $RClique^*$ algorithm is much more computationally intensive. For example, $RClique^*$ required on average 111.67 seconds to expand a vertex when searching for an 8-clique, while all of the other algorithms required less than 10 milliseconds. Correspondingly, the runtime per vertex in $RClique^*$ is substantially larger than all of the other algorithms. This is reasonable as $RClique^*$ is based on performing a set of simulated explorations before every exploration.

In this paper, we focus on minimizing the total exploration cost, regardless of the computational runtime. In that sense, $RClique^*$ is superior to all the other algorithms presented in this paper, in all the experimental settings we have used. However, $RClique^*$ is substantially more computationally expensive, introducing a natural tradeoff between computational effort and exploration cost.

Note that results for $RClique^*$ are not presented for the web graph search, since it requires probabilistic knowledge of the web citation graph. Such knowledge can be estimated by comparing the text in the titles of referenced papers, or by learning common properties of such graphs (e.g., the node degree distribution). However, this is beyond the scope of this paper.

9. Conclusion and Future Work

In this paper we proposed and analyzed several algorithms for the problem of finding a pattern in an unknown graph. The problem was formally defined and a best-first search algorithm was described for solving it. Several theoretical attributes of this problem were presented. We theoretically proved that *all* algorithms will have to expand almost all the vertices in the searched graph in the worst case. We also proved that the competitive ratio of any algorithm is close to that of random exploration. However, several heuristic algorithms were presented, namely $KnownDegree$, $Pattern^*$ and $RPattern^*$, that substantially outperform random exploration in practice.

$KnownDegree$ is a straightforward adaptation of a common known graph heuristic, in which the vertex with the highest degree is expanded first. The $Pattern^*$ heuristic algorithm is based on a "closeness' measure to the searched pattern, where the vertex that is the "closest" to the searched pattern is selected for exploration. $RPattern^*$ is designed for scenarios where some probabilistic knowledge of the searched graph is available. This knowledge is exploited by $RPattern^*$ by applying a Monte-Carlo sampling procedure in combination with $Pattern^*$ as a default heuristic.

To demonstrate the applicability of the proposed heuristic algorithms, we have shown how to implement $Pattern^*$ for two specific patterns: a k-clique and a complete bipartite graph. This was done by introducing the concept of a potential k-clique and potential complete bipartite graph, along with supporting corollaries that allow efficient implementation of the $Pattern^*$ heuristic algorithm. The $Pattern^*$ and $RPattern^*$ implementations for the k-clique pattern were denoted by $Clique^*$ and $RClique^*$ respectively.

Experimental evaluation on the k-clique pattern showed that both $Clique^*$ and $KnownDegree$ are much more efficient in terms of exploration cost than random exploration. In random graphs $Clique^*$ outperformed $KnownDegree$ significantly, while in scale-free graphs the $Clique^*$ and $KnownDegree$ performed very similarly, with a slight advantage for $KnownDegree$. $RClique^*$ significantly outperformed all other algorithms, even when the probabilistic knowledge was not very accurate ($noise = 50\%$). However, the runtime of running $RClique^*$ was much larger than the runtime required to apply $Clique^*$ or $KnownDegree$. This introduces a natural tradeoff between exploration cost and runtime.

k	Random		$RLS\text{-}LTM$		$KnownDegree$		$Clique^*$		$RClique^*$	
	Total	Vertex	Total	Vertex	Total	Vertex	Total	Vertex	Total	Vertex
6	0.05	0.00	0.11	0.00	0.03	0.00	0.02	0.00	552.85	25.79
7	0.10	0.00	0.14	0.00	0.06	0.00	0.05	0.00	1,483.24	57.00
8	0.26	0.00	0.15	0.00	0.15	0.00	0.08	0.00	3,043.21	111.67
9	0.49	0.01	0.18	0.00	0.25	0.01	0.12	0.00	5,484.83	206.86

Table 5
Runtime (in sec.) when searching for cliques of different sizes.

In addition to the experimental evaluation on random and scale-free graphs that were generated by a mathematical model, we also implemented a web crawler application that used the proposed best-first search with the proposed heuristic algorithms to search for a k-clique of academic papers via Google Scholar. In this domain as well, the results showed that in terms of exploration cost, $Clique^*$ is superior to $KnownDegree$, and both were much more efficient than random exploration. $RClique^*$ was not evaluated in this domain since extracting probabilistic knowledge of the web graph from the URLs of web pages is beyond the scope of this paper.

There are many open questions and future directions. We intend to perform a large scale online crawling experiment using the proposed algorithms, combining the clique search with a content based textual data mining. A natural extension of our work is to study imperfect matching of the searched pattern, and explore the tradeoff between finding exactly the searched pattern versus investing further exploration cost. For example, in the citation graph, a set of nodes which are *almost* a clique is also an indication for strong context relation.

Another possible research direction is to consider complex exploration cost models, where each vertex may have a different exploration cost. An example of such an exploration cost is a physical based model in which exploring a vertex requires an agent to move to its geographic location. Moreover, how multiple agents can search for a pattern cooperatively in an unknown graph is another interesting future direction.

Acknowledgements

This research was supported by the Israeli Science Foundation (ISF) grant No. 305/09.

References

[1] R. Stern, M. Kalech, A. Felner, Searching for a k-clique in unknown graphs, in: the International Symposium on Combinatorial Search (SoCS), 2010, pp. 83–89.

[2] B. Kalyanasundaram, K. R. Pruhs, Constructing competitive tours from local information, Theoretical Computer Science 130 (1) (1994) 125–138.

[3] L. Gkasieniec, R. Klasing, R. Martin, A. Navarra, X. Zhang, Fast periodic graph exploration with constant memory, Journal of Computer and System Sciences 74 (5) (2008) 808–822.

[4] I. C. Kim, Real-time search algorithms for exploration and mapping, in: the 10th International Conference on Knowledge-Based Intelligent Information and Engineering (KES), Vol. 4251, 2006, pp. 244–251.

[5] S. G. K. Christian A. Duncan, V. S. A. Kumar, Optimal constrained graph exploration, ACM Transactions on Algorithms (TALG) 2 (3) (2006) 380–402.

[6] R. Fleischer, G. Trippen, Exploring an unknown graph efficiently, in: the 13th Annual European Symposium on Algorithms (ESA), Lecture Notes in Computer Science, Springer, 2005, pp. 11–22.

[7] P. Panaite, A. Pelc, Exploring unknown undirected graphs, in: the 9th Symposium on Discrete Algorithms (SODA), Philadelphia , PA, USA, 1998, pp. 316–322.

[8] B. Awerbuch, M. Betke, R. L. Rivest, M. Singh, Piecemeal graph exploration by a mobile robot, in: Information and Computation, 1995, pp. 321–328.

[9] P. Cucka, N. S. Netanyahu, A. Rosenfeld, Learning in navigation goal finding in graphs, International Journal of Pattern Recognition and Artificial Intelligence (IJPRAI) 10 (5) (1996) 429–446.

[10] R. E. Korf, Real-time heuristic search, Artificial Intelligence 42 (2-3) (1990) 189–211.

[11] Y. Gabriely, E. Rimon, CBUG: A quadratically competitive mobile robot navigation algorithm, in: the International Conference on Robotics and Automation (ICRA), 2005, pp. 954–960.

[12] J. Bruce, M. Veloso, Real-time randomized path planning for robot navigation, in: G. A. Kaminka, P. U. Lima, R. Rojas (Eds.), RoboCup 2002: Robot Soccer World Cup VI, Vol. 2752 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2003, pp. 288–295.

[13] S. Argamon-Engelson, S. Kraus, S. Sina, Interleaved vs. a priori exploration for repeated navigation in apartially-known graph, International Journal of Pattern Recognition and Artificial Intelligence (IJPRAI) 13(7) (1999) 963–968.

[14] R. Meshulam, A. Felner, S. Kraus, Utility-based mulit-agent system for performing repeated navigation tasks, in: the 4th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS), ACM, Net York, NY, USA, 2005, pp. 887–894.

[15] A. Gilboa, A. Meisels, A. Felner, Distributed navigation in an unknown physical environment, in: the 5th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS), 2006, pp. 553–560.

[16] D. C. Rayner, K. Davison, V. Bulitko, K. Anderson, J. Lu, Real-time heuristic search with a priority queue, in: International Joint Conference on Artificial Intelligence (IJCAI), 2007, pp. 2372–2377.

[17] Y. Björnsson, V. Bulitko, N. R. Sturtevant, TBA*: Time-bounded a*, in: International Joint Conference on Artificial Intelligence (IJCAI), 2009, pp. 431–436.

[18] L. Shmoulian, E. Rimon, Roadmap-A*: an algorithm for minimizing travel effort in sensor based mobile robot navigation, in: the International Conference on Robotics and Automation (ICRA), Leuven, Belgium, 1998, pp. 356–362.

[19] A. Felner, R. Stern, A. Ben-Yair, S. Kraus, N. Netanyahu, PhA*: finding the shortest path with A* in unknown physical environments, Journal of Artificial Intelligence Research (JAIR) 21 (2004) 631–679.

[20] A. Felner, R. Stern, S. Kraus, PhA*: performing A* in unknown physica environments, in: the First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS), Bologna, Italy, 2002, pp. 240–247.

[21] M. R. Garey, D. S. Johnson, Computers and Intractability : A Guide to the Theory of NP-Completeness, W. H. Freeman, 1979.

[22] J. R. Ullmann, An algorithm for subgraph isomorphism, Journal of the ACM 23 (1) (1976) 31–42.

[23] H. Shang, Y. Zhang, X. Lin, J. X. Yu, Taming verification hardness: an efficient algorithm for testing subgraph isomorphism, in: the VLDB Endowment, Vol. 1, 2008, pp. 364–375.

[24] G. Valiente, C. Martínez, An algorithm for graph pattern-matching, in: the 4th South American Workshop on String Processing, 8, International Informatics, 1997, pp. 180–197.

[25] D. Eppstein, Subgraph isomorphism in planar graphs and related problems, in: the 6th annual Symposium on Discrete Algorithms (SODA), Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1995, pp. 632–640.

[26] C. Bron, J. Kerbosch, Algorithm 457: finding all cliques of an undirected graph, Communications of the ACM 16 (9) (1973) 575–577. doi:http://doi.acm.org/10.1145/362342.362367.

[27] P. R. J. Östergård, A fast algorithm for the maximum clique problem, Discrete Applied Mathematics 120 (1-3) (2002) 197–207.

[28] E. Tomita, T. Kameda, An efficient branch-and-bound algorithm for finding a maximum clique with computational experiments, Journal of Global Optimization 37 (1) (2007) 95–111. doi:http://dx.doi.org/10.1007/s10898-006-9039-7.

[29] P. M. Pardalos, J. Rappe, Mauricio, G. C. Resende, An exact parallel algorithm for the maximum clique problem, in: High Performance and Software in Nonlinear Optimization, 1997, pp. 279–300.

[30] R. Battiti, M. Protasi, Reactive search, a history-based heuristic for maximum clique, ACM Journal of Experimental Algorithmics 2.
URL citeseer.ist.psu.edu/battiti97reactive.html

[31] R. Battiti, M. Protasi, Reactive local search for the maximum clique problem, Algorithmica 29 (4) (2001) 610–637.

[32] W. Pullan, H. H. Hoos, Dynamic local search for the maximum clique problem, Journal of Artificial Intelligence Research (JAIR) 25 (2006) 159–185.

[33] R. Battiti, F. Mascia, Reactive and dynamic local search for max-clique: Engineering effective building blocks, Computers & Operations Research 37 (2009) 534–542.

[34] Y. Altshuler, A. Matsliah, A. Felner, On the complexity of physical problems and a swarm algorithm for the k-clique search in physical graphs, in: the First European Conference on Complex Systems (ECCS), Paris, France, 2005.

[35] B. Bollobás, P. Erdős, Cliques in random graphs, in: Mathematical Proceedings of the Cambridge Philosophical Society, Great Britian, 1976, pp. 419–427.

[36] J. A. Bondy, U. S. R. Murty, Graph Theory With Applications, Elsevier Science Ltd, 1976.

[37] E. W. Weisstein, Mathworld – a wolfram web resource, complete bipartite graph, http://mathworld.wolfram.com/CompleteBipartiteGraph.html (2011).

[38] A. Bonato, W. Laurier, A survey of models of the web graph, in: Combinatorial and Algorithmic Aspects of Networking (CAAN), 2004, pp. 159–172.

[39] M. Y. Kan, Web page classification without the web page, in: the 13th international World Wide Web conference (WWW) on Alternate track papers & posters, New York, NY, USA, 2004, pp. 262–263. doi:http://doi.acm.org/10.1145/1013367.1013426.

[40] M. Y. Kan, H. O. N. Thi, Fast webpage classification using url features, in: the 14th ACM International Conference on Information and Knowledge Management (CIKM), New York, NY, USA, 2005, pp. 325–326. doi:http://doi.acm.org/10.1145/1099554.1099649.

[41] S. Thrun, W. Burgard, D. Fox, Probabilistic Robotics (Intelligent Robotics and Autonomous Agents), The MIT Press, 2005.

[42] M. L. Puterman, Markov Decision Processes: Discrete Stochastic Dynamic Programming, Wiley-Interscience, 1994.

[43] M. L. Littman, Algorithms for sequential decision-making, Ph.D. thesis, Brown University (1996).

[44] B. Bonet, Deterministic POMDPs revisited, in: Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence, UAI '09, 2009, pp. 59–66.

[45] J. Pineau, G. Gordon, S. Thrun, Point-based value iteration: An anytime algorithm for POMDPs, in: Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI), 2003, pp. 1025 – 1032.

[46] E. A. Hansen, S. Zilberstein, LAO*: A heuristic search algorithm that finds solutions with loops, Artificial Intelligence 129 (1-2) (2001) 35–62.

[47] A. Cassandra, M. L. Littman, N. L. Zhang, Incremental pruning: A simple, fast, exact method for partially observable markov decision processes, in: In Proceedings of the Thirteenth Conference on Uncertainty in Artificial Intelligence, UAI '97, Morgan Kaufmann Publishers, 1997, pp. 54–61.

[48] R. Bellman, Dynamic programming treatment of the travelling salesman problem, J. ACM 9 (1) (1962) 61–63.

[49] P. A. Laplante, Dictionary of Computer Science, Engineering and Technology, CRC Press, 2001.

[50] A. Borodin, R. El-Yaniv, Online computation and competitive analysis, Cambridge University Press, New York, NY, USA, 1998.

[51] P. Erdős, A. Rényi, On random graphs I, Publicationes Mathematicae Debrecen 6 (1959) 290–297.

[52] B. Bollobás, Random Graphs, Cambridge University Press, 2001.

[53] M. D. Santo, P. Foggia, C. Sansone, M. Vento, A large database of graphs and its use for benchmarking graph isomorphism algorithms, Pattern Recognition Letters 24 (8) (2003) 1067–1079. doi:http://dx.doi.org/10.1016/S0167-8655(02)00253-2.

[54] A. L. Barabási, R. Albert, Emergence of scaling in random networks, Science 286 (5439) (1999) 509–512.

[55] D. Donato, L. Laura, S. Leonardi, S. Millozzi, Simulating the webgraph: a comparative analysis of models, Computing in Science and Engineering 6 (6) (2004) 84–89.

[56] B. Donnet, T. Friedman, Internet topology discovery: a survey, IEEE Communications Surveys and Tutorials 9 (4) (2007) 2–15.

[57] G. Siganos, M. Faloutsos, P. Faloutsos, C. Faloutsos, Power laws and the AS-level internet topology, IEEE/ACM Transactions on Networking 11 (4) (2003) 514–524. doi:http://dx.doi.org/10.1109/TNET.2003.815300.

[58] D. Eppstein, J. Wang, A steady state model for graph power laws, in: the 2nd International Workshop on Web Dynamics, 2002.