Generalized Longest Path Problems

Gal Dahan, Itay Tabib, Eyal Solomon Shimony, Ariel Felner

Ben-Gurion University of the Negev, Israel
dahanga@post.bgu.ac.il, itaytab@post.bgu.ac.il, shimony@cs.bgu.ac.il, felner@bgu.ac.il

Abstract

The longest simple path and snake-in-a-box are combinatorial search problems of considerable research interest. We create a common framework of longest constrained path in a graph that contains these two problems, as well as other interesting maximum path problems, as special cases. We analyze properties of this general framework, and produce bounds on the path length that can be used as admissible heuristics for all problem types therein. For the special cases of longest simple path and snakes, these heuristics are shown to reduce the number of expansions when searching for a maximal path, which in some cases leads to reduced search time despite the significant overhead of computing these heuristics.

1 Introduction

In the LSP problem the aim is to find the Longest *Simple Path* (where no vertex is visited more than once) between two given vertices in a graph. LSP is a fundamental problem in graph theory, known to be NP-hard, and even hard to approximate within a constant factor (Karger, Motwani, and Ramkumar 1997). The motivation to solve LSP comes from a variety of domains such as information retrieval on peer to peer networks (Wong, Lau, and King 2005), estimating the worst packet delay of Switched Ethernet network (Schmidt and Schmidt 2010), multi-robot patrolling (Portugal and Rocha 2010), and VLSI design where the longest path should be found between two components on a printed circuit board (Chen 2016).

Several prior works approached LSP as a heuristic search problem. Stern et al. (2014) showed how to modify common heuristic search algorithms that were designed for minimization (MIN) problems to solve maximization (MAX) problems. They used LSP to demonstrate their findings and proposed an admissible heuristic for LSP. Then, Palombo et al. (2015) proposed several admissible heuristics for solving the *Snake-in-the-box* (SIB) problem (Kautz 1958). SIB is a reminiscent of LSP that is important for a useful type of efficient error correction codes. IN SIB, a path may not use neighbours of vertices that are already in the path. A followup work (Cohen, Stern, and Felner 2020) focused on LSP and proposed several methods to detect and prune states that are *dominated* by other states. In addition specific grid-based heuristics for LSP were proposed.

This paper continues that line of research, making the following contributions. We first define the *Generalized Longest Simple Path* (GLSP) problem as finding the longest possible path under a set of constraints on the behaviour of that path. The problems mentioned above are special cases of GLSP. A theoretical study on different properties of the problem follows. In particular, we prove theorems about the complexity of deciding path existence, with or without a *must-include set* of vertices (that must be on a path) specified in the input. We then introduce methods for generating admissible heuristics for GLSP based on exclusion sets and must-include paths. We also extend the notion of pattern-based heuristics for Snake-in-the-box (Palombo et al. 2015) taken from a bi-connected projection of the graph to include components that are not necessarily disjoint. Finally, we describe an implementation, including a new incremental implementation of the heuristics. Empirical evaluation of our heuristics on LSP and Snakes shows significant savings in the number of expanded nodes compared to previous approaches. Frequently this translates into a reduced runtime despite the increased overhead of our new heuristics.

2 Background

Solving the shortest-path problem using search typically involves derivations of the A* algorithm. These are called MIN problems as the task is to find a solution with the least possible cost. In these problems one always prefers nodes with lower $f(n) = g(b) + h(n)$ values. But, in some cases (such as LSP) solutions are needed with the maximal possible weight. These are called MAX problems. The following modifications to textbook A* are needed to adapt it to MAX problems and in particular to LSP (Stern et al. 2014).

Admissibility in MAX problems. A function h is said to be admissible for MAX problems iff for every state n in the search space it holds that $h(n)$ is *greater than or equal to* the remaining optimal weight to the goal (the length of the longest simple path from $N.head$ to g in the case of LSP).

Choosing from OPEN. In MIN problems, A* pops from OPEN in every iteration the node n with the lowest $g(n) + h(n)$. In MAX problems, A* pops from OPEN in every iteration the node n with the highest $g(n) + h(n)$.

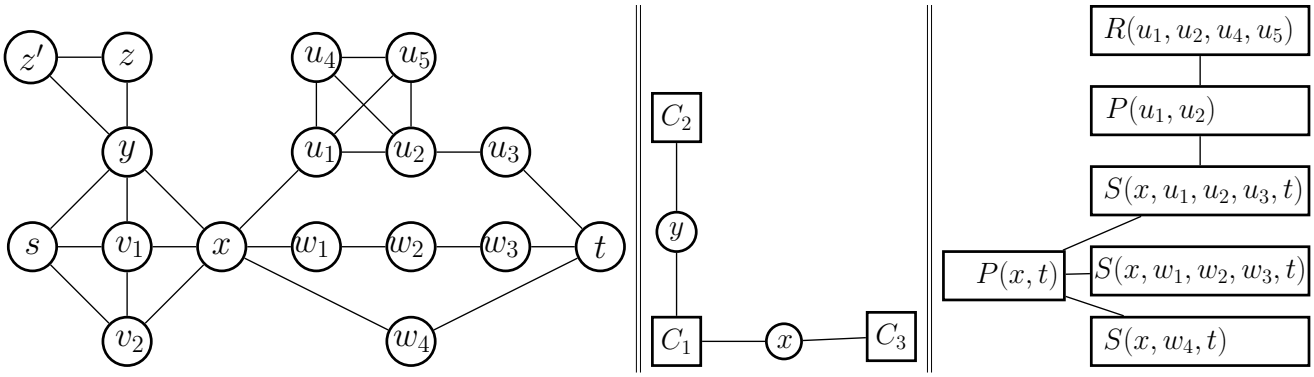

Figure 1: A graph (left), its biconnected block-cut tree (center), and simplified SPQR tree of block C_3 (right)

2.1 Graphs and Connectivity Types

Let $G = (V, E, w)$ be a connected undirected weighted graph with no self-edges. A path from v_0 to v_m is an alternating sequence $P = (v_0, e_0, v_1, e_1, ...v_m)$ of vertices and edges in G such that following elements in the sequence are adjacent, i.e. each edge is incident on the preceding and following vertices in P. Path P is simple if no vertex appear is P more than once. In this paper we aim at finding longest paths under constraints - one such commonly used constraint is that the path is simple. This case is called longest simple path (LSP) problem.

G is *connected* if for every pair of vertices $w, v \in V$ there is a path from w to v in G. Obviously the longest simple path cannot have more edges than the number of vertices in G minus one. Deciding whether the length of a simple path actually equals this bound is known as the NP-complete Hamiltonian path decision problem (Garey and Johnson 1979). This paper is about search for longest paths, and different notions of k-connectivity (see below) play a crucial role in designing admissible heuristics for this search. We assume that search progresses from a start vertex s by adding one edge and vertex at a time to a partial path P that has s' as a last vertex. A trivial admissible heuristic (bounding the maximum number of vertices by which P can be extended) is h_r, the number of vertices in the connected component in which s' resides at in the remaining graph, $G - P$.

Path search literature (Cohen, Stern, and Felner 2020) uses in addition the notion of 2-connectivity (also called biconnectivity): $G = (V, E)$ is (vertex-wise) *biconnected* if for every $v \in V$, $G - \{v\}$ is connected. In other words, there are at least two disjoint paths between any two vertices in a bi-connected component. A biconnected component of G is any maximal subgraph of G that is biconnected. Every pair of biconnected components G_1, G_2 has at most one vertex v in common; which is called an articulation point, or a separator. A graph consisting of one block-vertex $V(G_i)$ representing each connected component G_i of G, one vertex v for each separator, and an edge between v and $V(G_i)$ just when the separator $v \in G_i$ is known as the *block-cut graph* of G. For example, Figure 1 shows a graph (left), with two separators: y and x, splitting the graph into 3 biconnected blocks; and the corresponding block-cut tree (center).

A more general definition of connectivity used in this paper is *k-connectivity*. Graph G is (vertex) *k-connected* if there is no set of vertices S of size $k - 1$ which disconnect G, when removed. Such graphs have at least k vertex-disjoint paths between any two vertices. Here we mostly use 3-connectivity, also called triconnectivity, to develop admissible heuristics. In a biconnected graph, a pair of vertices is called a *separation pair* if deleting it makes G disconnected. *Triconnected components* are maximal subgraphs that are 3-connected. A biconnected graph can be organized into a tree-like structure of triconnected components and separation pairs, known as an SPQR tree (Battista and Tamassia 1996). SPQR trees have many technical details, one must read the cited paper to fully understand them. Here, we describe details essential to our work. SPQR trees consist of *super-vertices* of 4 types; each super-vertex represents a graph fragment. R ("Rigid") super-vertices represent triconnected components. P ("Parallel") super-vertices represent a separator pair that separates the graph into three or more components. S ("Series") super-vertices represent a series of vertices. Q super-vertices represent individual edges (not used in this paper). See Figure 1 (right) for an SPQR tree of block C_3, which contains all vertices between x and t. The pair x, t gives rise to a super-vertex of type P (parallel separator-pair) which separates the graph into 3 components, each of type S. In the figure, for each S super-vertex the sequence of vertices are listed in the parenthesis. The S vertex with the u_i vertices contains separator pair u_1, u_2, giving rise to a P super-vertex, which abuts a triconnected component (an R vertex). See (Westbrook and Tarjan 1992; Battista and Tamassia 1996) for details on SPQR trees, their properties, and how to construct them in linear time (Gutwenger and Mutzel 2000).

SPQR trees are used in this paper to design an admissible heuristic. Note that separators are pairs of vertices. Therefore, any triconnected component can be entered and exited from any other adjacent component - thus it is quite a challenge to use the separators to produce a heuristic.

2.2 Longest Path Search Heuristics

A biconnected block can only be entered or exited through a separator. Thus, a simple path (which cannot use a separa-

tor more than once) between vertices s and t can only visit biconnected blocks on the path from (a block containing) s to (a block containing) t in the block-cut tree of G. Thus, blocks not on such a path can be dropped from G before counting the number of unvisited vertices, resulting in the admissible biconnected component heuristic h_{BCC} (Cohen, Stern, and Felner 2020). This heuristic was found to significantly reduce the number of expanded nodes both for LSP and for Snake-in-the-box when compared to h_r. In Figure 1, s is in component C_1, and t is in component C_3. Component C_2 is not on the path from C_1 to C_3, and can be discarded.

Another admissible heuristic for Snake-in-the-box appears in (Palombo et al. 2015). The remaining graph is partitioned into disjoint sets of connected components. For each such component the largest set of vertices that can be in a snake was identified. One such component type used was a star-shape subgraph G' consisting of a vertex and its immediate neighbours, which cannot all be in the same snake path if the number of vertices in G' is greater than 3. The number of allowed vertices in each of these sets were added together into an admissible heuristic. This can be seen as the equivalent of the additive PDBs heuristics for MIN case (Felner, Korf, and Hanan 2004). An important remaining question is on the best way to find effective partitions quickly.

3 GLSP Problem Statement

As the longest simple path (LSP) (Cohen, Stern, and Felner 2020) problem is the best known special case, we call our framework generalized longest simple path (GLSP).

A pair (x, M_x) with $x \in (V \cup E)$ and $M_x \subseteq (V \cup E)$ is called a *local exclusion constraint*. The semantics of a constraint are as follows: **after** exiting x, the path cannot visit any member of M_x. A global exclusion constraint for G is a set of local exclusion constraints. Let L be a global exclusion constraint. If (x, M_x) is in L, we denote M_x (assuming it is unique) by $L(x)$. Path p violates global exclusion constraint L if it violates any of the local constraints of any element in p. Thus defined, the global constraint L is always monotonic: if p violates L, every extension of p also violates L. When the local constraints in L are defined uniformly, we call L a *constraint rule*. For example, the global constraint: $\forall x \in (E \cup V),\ L(x) = \{x\}$. i.e. "no vertex or edge of the graph may be visited more than once" is a constraint rule.

Definition 1 (Generalized LSP Problem)**.** *Given a graph $G = (V, E, w)$, and a global exclusion constraint L, find a path of maximum weight w in G (optionally starting at start vertex s, optionally ending at target vertex t) that does not violate L.*

Well known special cases that have been studied in the literature include, for unweighted graphs:

1. Longest (vertex-wise) simple path (denoted as standard LSP): pairs in L are $(x, \{x\})$ for all vertices $x \in V$. This is a variant of the Hamiltonian path problem.
2. Longest (edge-wise) simple path (denoted ELSP): pairs in L are $(x, \{x\})$ for all edges $x \in E$. This is a variant of the Euler path problem.
3. Snake: pairs in L are $(x, N(x) \cup \{x\})$ where $N(x)$ are the immediate neighbours of x, for all $x \in V$.
4. Snake in the box: Snake problem, with G being an n-dimensional binary hypercube.

All the above are constraint rules, since the local constraints therein are defined uniformly. But it is also possible to require, for example, $L(x) = \{x\}$ for some vertices and $L(x) = \{x\} \cup N(x)$ for the rest of the vertices, resulting in a global constraint somewhere between vertex-wise simple path and Snake: cannot visit any vertex more than once, but some vertices also cause their neighbors to be disallowed.

Our algorithms provided in this paper are general and can be applied to any GLSP. But, to be focused we experiment and provide examples only for LSP and Snake.

4 Fundamental Issues of Constrained Paths

Henceforth we assume that the variant of GLSP we are addressing is finding a constrained longest path from a given source vertex s to a given target vertex t. Unless stated otherwise, we also assume uniform weights, or equivalently $w(e) = 1$ for all $e \in E$. We begin with the more basic constrained path existence problem.

4.1 Complexity of Finding Any Constrained Path

A crucial subproblem of GLSP is that of path existence. That is, for a given a graph G, a source vertex s, a target vertex t, and a global constraint L, is there **any path** (not necessarily shortest or longest) from s to t in G that does not violate L? It turns out that even this basic problem is intractable:

Theorem 1. *The path existence problem is NP-complete.*

Proof. (outline) By reduction from 3-SAT. Use a level graph with a vertex for every literal, and constraints disallowing vertices inconsistent with previously visited literals. □

Henceforth, we address the GLSP only for constraints L where we know that path existence is tractable. It is easy to see that with the constraints of standard LSP and Snake (which are the problems in the focus of this paper) the path existence problem is tractable: it can be decided by breadth-first search from s in time linear in $|E|$.

4.2 Must-Include Paths

Given that there exists a constrained path, the next fundamental question is, does there exist a constrained path that includes a given set of elements S? As this paper's focus is mostly on vertex-constrained paths, we limit this discussion to the case where S is a set of vertices. It is well known that even in the simplest case where the constraint of each vertex $x \in V$ consists of $\{x\}$, i.e. that of simple path, the problem of finding a path that must include *all* vertices in S is NP-hard (trivial reduction from Hamiltonian path). Therefore, we focus on cases where S is a very small set of vertices, mostly on $|S| = 1$ and $|S| = 2$.

Being able to quickly answer such must-include constrained path queries is important to generate admissible search heuristics, which we examine next.

5 Admissible Heuristics for GLSP

Consider any state of an A* maximum search where we have a path starting at s, for a global constraint at least as tight as simple path. The trivial bound h_r on the length of the remaining path is the number of as-yet unvisited vertices in G. From h_r we can subtract the number of vertices x through which there is no path in the remaining graph from the end of the path s' to t. For better clarity, we formally state this property in terms of the original graph G:

Theorem 2. *Given a constrained (s,t) longest path problem on graph $G = (V,E)$, with a constraint L such that $x \in L(x)$ for every vertex $x \in V$. Denote by S' the set of all vertices v for which there is no must-include $\{v\}$ constrained path from s to t. Then the length of the longest constrained (s,t) path is at most $|V| - |S'| - 1$.*

Proof. Constraint L forces every vertex to be visited at most once in a constrained path (may be less, e.g. for the snake constraint). The theorem thus follows immediately. □

For example, in the graph of Figure 1 (left) only vertices z, z' cannot appear in any simple path from s to t, so this upper bound on the LSP is $17 - 2 - 1 = 14$.

Moving to the case $|S| = 2$, we get the scheme with the following steps.

Step 1: Remove from G every vertex v for which there is no must-include path from s to t, resulting in graph $G' = (V', E')$. *Step 2:* Construct an auxiliary "exclusion" graph G_{ex} consisting of all vertices $V' - \{s,t\}$, and an edge $\{u,v\} \in G_{ex}$ just when there is no simple path (from s to t) in G' including $\{u,v\}$. [1] Then we have:

Theorem 3. *Every (s,t) path in $G = (V,E)$, constrained by L s.t. $\forall x \in V, L(x) \in \{x\}$, has length at most $\alpha(G_{ex}) + 1$, where $\alpha(.)$ is the maximum independent set size.*

Proof. Constraint L forces every vertex to be visited at most once in a constrained path (may be less, e.g. for the snake constraint). The vertices from G' along a path must form an independent set of G_{ex}, because no constrained path can include two vertices that are connected by an edge in G_{ex}. The theorem thus follows immediately. □

For example, in the graph of Figure 1 (left) after reaching vertex x one can traverse (only) either the top branch from x to t (the u_i vertices), or the middle branch, or the bottom vertex w_4. Thus G_{ex} contains all the graph vertices, except for s, t, and z, z', as the latter vertices are singleton exclusions and were deleted above in step 1. The edges in G_{ex} are between w_4 and all the other w_i vertices, between all u_i vertices and all w_j vertices. All other vertices have no edges. The maximum independent set in this G_{ex} consists of all five u_i vertices, and all the singleton vertices (y, v_1, v_2, x), total size 9. Thus the bound is 10. In this graph the bound happens to be tight, i.e. equal to the number of edges in a LSP.

The above bound based on the independent set is an upper bound, and can be used as an admissible heuristic. We thus define $h_{IS}(G,s,t) = \alpha(G_{ex}) + 1$. Computing the maximum independent set is also NP-hard, so we approximate it as follows. A *clique cover* of a graph is a set of cliques such that all vertices are in at least one of these cliques. The size of a maximum independent set is less than or equal to the number of cliques in any clique cover (the cliques do not have to be disjoint and/or maximal). We denote the latter clique cover approximated value of $h_{IS}(G,s,t)$ by $\hat{h}_{IS}(G,s,t)$. The clique cover approximation is also an admissible heuristic, and remains such even if it is evaluated based on an exclusion graph that is missing some edges. In our running example from Figure 1, if we pick all 4 singleton vertices in G_{ex}, as well as the cliques: $\{u_1, w_1, w_4\}, \{u_2, w_2, w_4\}, \{u_3, w_3, w_4\}, \{u_4\}, \{u_5\}$ we have a cover of size 9, which happens to be equal to the size of the maximum independent set.

5.1 On Deciding Must-Include Paths

We have shown that the notion of must-include paths can be used to define admissible heuristics for longest path. However, to compute these heuristics we must decide, given a set of vertices S, whether there exists a must-include S path from s to t, which we do next.

Although theorems 2 and 3 hold for any constraint rules where $x \in L(x)$, we focus mostly on vertex-disjoint paths, i.e. the constraint rule $L(x) = \{x\}$ for every vertex x. We thus begin with this better-known case (LSP). As stated above, for S of unbounded size deciding existence of a must-include path for S is NP-complete. However, the problem is in P for a bounded set size, due to a result on finding K vertex-disjoint paths (Robertson and Seymour 1995). Although showing that the problem complexity is $O(|V|^3)$, the authors also state "the algorithm is not practically feasible, since it involves the manipulation of enormous constants".

To be useful as a heuristic, one must be able to quickly guarantee that there is no must-include path. We begin with sets of size 1, which can be handled efficiently using biconnected components. As stated in the background, heuristic h_{BCC} computes the biconnected blocks in G, and drops every block not on the block-path from s to t.

It is easy to show that every vertex v that h_{BCC} discards has no must-include $\{v\}$ path from s to t. The converse, that for all remaining nodes v there exists a must-include $\{v\}$ path from s to t, follows from Menger's theorem:

Theorem 4. *Let $G = (V,E)$ be a biconnected graph, and $s, t, v \in V$. Then there is a simple s to t path through v.*

Proof. Let G' be G extended by adding new vertex w, and edges $\{s,w\}$ and $\{t,w\}$. By construction, G' is biconnected. By Menger's theorem (Menger 1927), two internal vertex-disjoint paths from v to w exist. Joining these paths at v and removing w gives the desired simple path from s to t. □

Theorem 4 implies that the vertices in S' from Theorem 2, i.e. every v for which there is no must-include simple path from s to t through v; are exactly those not in the biconnected components on the block-cut-tree from s to t. Therefore, the bound of Theorem 2 is in fact exactly the biconnected component heuristic from (Cohen, Stern, and Felner 2020). Computing this bound can be done efficiently using

[1] Using the exclusion graph was originally suggested by Avinoam Yehezkel for the standard LSP.

the biconnected components. We thus denote this heuristic as $h_{BCC}(G, s, t) = |V| - |S'| - 1$.

Since to detect must-include sets of size 1 we can use biconnected components, it is reasonable to expect that for must-include sets of size 2 (exclusion pairs) we can use triconnected components, which can also be computed efficiently (Battista and Tamassia 1996). First, we show that if the graph is triconnected there are no exclusion pairs.

Theorem 5. *Let $G(V, E)$ be a triconnected graph. Then, for every $s, t, v, w \in V$ there exists a simple path in G from s to t that includes v and w.*

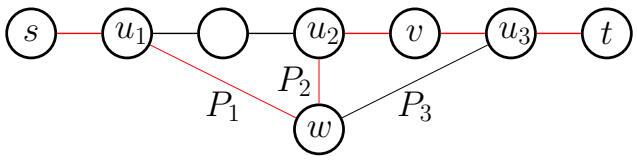

Figure 2: Path splicing to get path through s, w, v, t (red)

Proof. Since G is triconnected, Dirac's second theorem (Dirac 1952, 1960) implies that there exists a simple cycle containing any 3 vertices. Then s, v, t are on a simple cycle, and there is a simple path P from s to v and then to t. From Menger's theorem it can be shown (also called Dirac's Fan Lemma (Dirac 1960)) that in a k-connected graph, given a vertex w and a set of vertices V' there exist k vertex-disjoint (except at w) paths from w to vertices in V'. Let V' in our case be the set of vertices of P. Then there exist 3 vertex-disjoint (except for w) paths from w to the vertices of P, which we denote by P_1, P_2, P_3 (see Figure 2), W.l.o.g. assume that P_i are also all disjoint from P except for their last vertex which is in P. Now consider the vertices u_1, u_2, u_3 in P ending the paths P_1, P_2, P_3, respectively, and assume w.l.o.g. that u_1 comes before u_2 which comes before u_3 on P when traversing from s towards t. If u_1, u_2 are both on the path segment of P from s to v, merge the following path segments: s to u_1 along P, then from u_1 along P_1 to w, then P_2 to u_2, then from u_2 to v and then to t along the remaining part of P. We thus get a simple path from that starts from s, traverses w, then v, and ends at t. (Note that we allow $u_1 = s$, in which case the segment of P from s to u_1 is a zero-length path, and likewise for the case where $u_2 = v$, but the result is still the desired simple path). If u_2 is not on the path segment of P from s to v, then it must be on the path segment from v to t, and due to the ordering this is true for u_3 as well. In this case the desired path is constructed from the segments: s to v and then u_2 along P, then w using P_2, then u_3 using P_3, and finally t using the remainder of P. We thus get a simple path from s to v to w to t, i.e. with the ordering of w, v reversed. (Either order is acceptable since we only required that both w, v be on the path.) □

To find exclusion pairs, we must thus consider a partition of the graph into triconnected components. Assume w.l.o.g. that G is biconnected (otherwise create the biconnected components and handle each biconnected component separately). G can be decomposed into its triconnected components, as done by SPQR tree algorithms, which find all pairs of vertices u, v which, when removed, make G unconnected. Such pairs of vertices are called *separation pairs*.

Let v, w be a separation pair that partitions G into disjoint (other than v, w) subgraphs $G_1, G_2, ..., G_k$, $k \geq 3$. Clearly, a simple path can only enter and then exit at most one of the G_i, because entering and exiting G_i uses up both w and v which cannot be used any more. The simple path may in addition start and/or end at some G_j, G_m with $j \neq i \neq m$ although either $j = m$ or $j \neq m$ are possible. Therefore, let G_{i_1} and G_{i_2} be components that do not contain either s or t, and let vertices $x \in G_{i_1}$, $y \in G_{i_2}$, both distinct from v, w. We call this "case P" because this case occurs in super-vertices of type P in the SPQR tree. Note that for the exclusion pairs set generated here to be non-empty we need at least two components (neighbors of the P node) that do not contain s, t except as separators, so either $k \geq 3$ with s, t in the same component or as separators, or we need $k \geq 4$.

Another case, (called "case S", as it occurs in super-vertices of type S in the SPQR tree), is when we have a cycle of 4 separation pairs, i.e. $(v_1, v_2), (v_2, v_3), (v_3, v_4), (v_4, v_1)$. Assume for simplicity that each separation pair creates exactly one component that does not contain the other separation vertices, which we call $G_{12}, G_{23}, G_{34}, G_{41}$ respectively. Now if $s \in G_{12} - \{v_1, v_2\}$ and $t \in G_{34} - \{v_3, v_4\}$ then to reach t from s can traverse either $G_{23} - \{v_2, v_3\}$ or $G_{41} - \{v_4, v1\}$ but not both. That is, let vertices $x \in G_{23} - \{v_2, v_3\}$, $y \in G_{41} - \{v_4, v_1\}$. Note that if we have $s = v_1$ the above also holds, except that now traversing $y \in G_{41}$ excludes $G_{12} - \{v_1\}$ as well as G_{23}. So in this case let $y \in G_{41} - \{v_4, v_1\}$, $x \in G_{23} \cup G_{12} - \{v_1, v_3\}$. Symmetrically for t being in a separator. This case can obtained by examining the S super-vertices in the SPQR tree.

Property 6. *Under the conditions stated in either case P or in case S, there is no simple path in G from s to t that includes both (the above defined) x and y.*

And thus we can add, for each such x, y vertex pair, the edge $\{x, y\}$ to the exclusion graph G_{ex}.

In the graph of Figure 1 we have a P super-vertex (right), with three neighbors, none of which contain the "entry vertex" x or the "exit vertex" t, except in the separator. So only one of these subtrees can be entered and exited. Therefore we add to G_{ex} an edge between every vertex (not including the separator) of each component, and every vertex of each of the other components. The above happen to be all the edges that need to be in G_{ex}. Case S does not add any edges to G_{ex} in this example, because the entry and exit vertices are "virtually" adjacent in each of the S super-vertices.

The above S and P cases are incomplete in that they do not find all exclusion pairs, leading to an overestimate in the path length estimation, thus still leading to an admissible heuristic which we denote by $\hat{h}_{SPQR}$. In practice, these cases cover most of the exclusion pairs, according to the empirical evaluation, at relatively little computational cost.

5.2 Detecting Exclusion Pairs Using Flow

To achieve better coverage of exclusion pairs, we optionally check every potential exclusion pair of vertices not ruled out by Theorem 5, by using flow techniques. The most precise scheme (though still not proved complete) we use is multiflows. We require the flows: $f_{sw} = f_{wv} = f_{vt} = 1$ or $f_{sv} = f_{vw} = f_{wt} = 1$ between the respective vertex pairs.Keeping these flows separate requires having different variables to represent each potential flow over each edge, and solving the resulting inequalities using linear programming (LP). No LP solution means there is no such path, and the pair $\{v, w\}$ is added to G_{ex}. We denote this approximation of h_{IS} by $\hat{h}_{LP}$ We also have a method $\hat{h}_{Flow}$ which uses simple (rather than multi) flow, which is faster but less precise than multiflow.

5.3 Computing the Heuristics

Until now we have examined must-include path existence in isolation. However, numerous such computations are needed for each heuristic value computation. This overhead can be minimized by reusing computations. Even the computation of the biconndected components can be reused: if there is more than one biconnected block on the path from s', the current end of the partial path, to t in the block-cut tree, the only block where changes to the heuristic can occur (w.r.t. to the parent search node) is in the one containing s'.

Algorithm 1: Compute Heuristics Incrementally

Input: G, s, t, **current search node** N

if N is the root node **then**
  Let $s' = s, \quad t' = t, \quad \mathcal{B} = (G), B = G.$
else
  Let $\mathcal{B} =$ copy $\mathcal{B}$ from parent(N)
  Copy heuristic values and t' from parent(N)
  Remove s'(parent(N)) from $first(\mathcal{B})$.
end if
if $s' = t'$ **then**
  $\mathcal{B} = rest(\mathcal{B}), \quad B = first(\mathcal{B}). \quad t' = t'(B).$
  /* (We just moved to a new biconnected block) */
else
  Compute BCC block-tree BT in B
  Let $\mathcal{B}'$ be the BT blocks on the path from s' to t'
  For each $B \in \mathcal{B}''$, compute the heuristic value $h(B)$
    /* (h_{BCC}, h_{IS}, or $\hat{h}_{IS}$ as desired) */
  Replace $first(\mathcal{B})$ in $\mathcal{B}$ by the list $\mathcal{B}'$.
  Return sum of $h(B, s'(B), t'(B))$ over $B \in \mathcal{B}$.
end if

Thus, we use Algorithm 1, assuming $s' \neq t$, in which case we are at a goal node and do not need to compute a heuristic. At each search node, we keep lists of remaining vertices, of the remaining biconnected components on the path to t, and of the heuristics values for each biconnected component. For convenience we also keep t', the exit vertex in the component containing s', the end of the current partial path. For each generated search node, s' was just added to the partial path so we (re)compute the biconnected components (only) between s' and t'. For each biconnected block B along the clock-cut tree path, we denote its entry vertex by $s'(B)$ and its exit vertex by $t'(B)$. Note that if $s \in B$ then $s'(B) = s'$ and if $t' \in B$ then $t'(B) = t'$.

Computing the heuristic value in each block B is simply counting the number of vertices for $h_{BCC}(B, s'(B), t'(B)$. To compute $\hat{h}_{IS}(B, s'(B), t'(B)$ proceed as in Algorithm 2.

Algorithm 2: Compute $\hat{h}_{SPQR}$ in biconnected component

Input: $B, s'(B), t'(B)$

Compute T, the SPQR tree for B.
Let $V' =$ vertices(B)$-\{$s'(B),t'(B).$\}$, $B_{ex} = (V', \emptyset)$
For each P vertex in T, add edges to B_{ex} using case P.
For each S vertex in T, add edges to B_{ex} using case S.
Greedily select cliques in B_{ex}, preferring large cliques, until a clique cover CC of B is achieved.
Set $\hat{h}_{SPQR}(B, s'(B), t'(B)) = |CC| + 1$

5.4 Additional Heuristics for Snake Constraint

The A* variant described below, as well as the exclusion pairs heuristic, works correctly for all constraint types at least as tight as the LSP constraint rule: $x \in L(x)$ for every vertex x. However, for tighter constraints it may be possible to generate more informative heuristics. How to do so is a hard open question in general. Still, we briefly examine the special case of the Snake constraint rule, i.e. $\forall x \in V,\ L(x) = \{x\} \cup N(x)$. The star-shaped exclusion set proposed in (Palombo et al. 2015), where not all vertices can be on the same snake path, is one such exclusion set. Note that it is better to make such sets as small as possible, thus unlike (Palombo et al. 2015) we use a 4-vertex star shape, which we henceforth call a "Y" exclusion. In addition, note that the Snake constraint does not allow any path to include any set of vertices that form a cycle in G, and thus any cycle in G is an exclusion set.

Given a disjoint set of patterns (cycles, star-shapes), and a set of disjoint exclusion-pairs that are also disjoint from the cycles and Y-shapes, we can apply the exclusion-pairs bound and also deduct 1 for each cycle and each Y exclusion. As in (Palombo et al. 2015), for disjoint patterns we can easily use this as a bound on total path length.

For the Snake constraint we can use triconnected components to detect additional excluded vertices (rather then exclusion pairs): entering a region of the graph by a node v cuts off possible exit through any neighbours of v as well v itself. We call every separator pair v, w in G that also has an edge $\{v, w\}$ in G an *effective* separator. Every vertex v residing in an SPQR tree fragment separated from s and t by an effective separator cannot be on a simple path from s to t, even if v is in the same biconnected block as s and t. For example, in Figure 1, the separator-pair u_1, u_2 is an effective separator, and the top R component cannot be used in a Snake path from s to t, other than traversing just the separator vertices: that is, neither u_4 not u_5 can be traversed in a Snake path. This reduces the value of the bound on the Snake path length by 2. We denote this heuristic by $\hat{h}_{SPQR^+}$.

6 Lazy A* Search for GLSP

We search for the optimal constrained path using a variant of Lazy A* (Zhang and Bacchus 2012), modified to find maximal-valued, rather than minimal-valued states. We have several admissible (not necessarily consistent) heuristics h_i, some of which are expensive to compute. Thus not all heuristics are computed at node generation time.

Algorithm 3: Lazy Max A*

Input: G, L, s, t

$OPEN \leftarrow$ MakeHeap(MakeSearchNode(s))
while $OPEN$ is not empty **do**
  $curr \leftarrow$ GetMax($OPEN$)
  if $t \in path(curr)$ **then**
    return curr
  end if
  if ($i \leftarrow$ ComputeHeuristic?($curr$)) **then**
    Compute $h_i(curr)$, re-insert $curr$ into $OPEN$
  else
    $OPEN \leftarrow$ insert($OPEN$, expand($curr$))
  end if
end while
return $fail$

The heuristics are applied in order of index i. We assume w.l.o.g. that higher-index heuristics dominate, i.e. $h_{i+1}(n) \leq h_i(n)$ for every mode n in the search space. (Otherwise, just redefine each heuristic to be the minimum between itself and the previous ones.) We use heuristics $h_1 = h_{BCC}$, $h_i = \hat{h}_{IS}$, for $i > 1$, using increasingly precise versions of $\hat{h}_{IS}$, with an increasingly more complete version of the exclusion graph G_{ex}. Specifically we use $h_2 = \hat{h}_{SPQR}$, or $h_2 = \hat{h}_{Flow}$ and $h_3 = \hat{h}_{LP}$.

ComputeHeuristic? returns the number of the next heuristic h_i for $curr$ to be computed, false if none left. It can be made to bypass some expensive heuristics (e.g. $\hat{h}_{LP}$) in some cases, as in "rational" Lazy A* (Karpas et al. 2018), further reducing runtime, an option not used in this paper.

The expand function works as follows. For each possible edge e leading out of the end of $p = path(curr)$, add e to the end of p, denoting the extended path by p'. Check if there is still an L constrained path to t from the end of p', otherwise discard p'. Create a new search node for n with $path(n) = p'$ and $f(n) = g(curr) + w(e) + h_0(n)$. As search nodes contain the entire path, with only one way to generate any given path, there is no need for a closed list.

7 Empirical Evaluation

Experiments evaluating our heuristics vs. a baseline of the h_{BCC} heuristic searching for LSP and for Snake paths were in different 4-connected grid types. The code was implemented in Python using Sage for the advanced graph algorithms, such as computation of the SPQR tree. Experiments were run on AMD Ryzen 9/3900X 12-Core @3.80GHz with 64.0GB, 2667MHz RAM. We evaluate all heuristics both in naive implementation (recomputing biconnected components) and our new incremental version.

In our first experiment we compare *all* our heuristics in two types of graphs. The first type is a demonstration of the power of exclusion pairs on a favorable graph with parallel paths, in the "hall" graph of Figure 3, with s in purple and t in green. Unlike h_{BCC}, our new heuristics quickly realize that only one of the short parallel corridors at the bottom can be traversed. They thus direct the search towards the top bypass, immediately finding the optimal solution. The second type are randomly generated 2D-grid graphs, such as the one in Figure 3 (colors indicate unpruned biconnected blocks). Results for LSP are shown in Table 1.

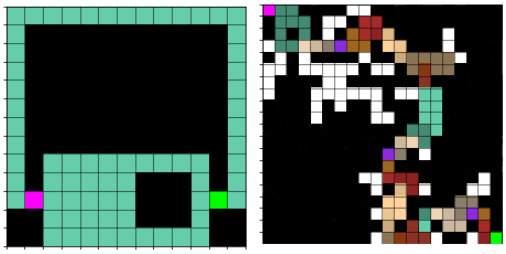

Figure 3: Hall Map (left), Random Map G1821 (right)

For each problem instance, we report the length of the longest simple path as f^*. Then, for each heuristic we report the following: (1) The number of expanded nodes, unless the search timed out. (2) The value of the heuristic at the initial state, denoted by $h(s)$. (3) Runtime in seconds both for the naive implementation (t_{ni}) and for the incremental version (t_{inc}) of the heuristic. **Bold** fonts indicate the best variant(s). The incremental implementation had little or no effect on the number of expanded nodes (not shown), but resulted in a very significant improvement in runtime, sometimes by an order of magnitude.

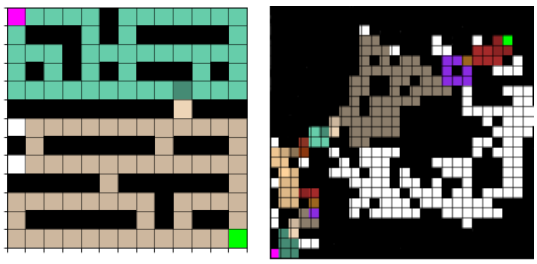

Figure 4: Maze (left), Rooms Instance (right)

All our new $\hat{h}_{IS}$ variants resulted in a major reduction in the number of expanded nodes, in some cases by more than an order of magnitude. The LP based technique had the smallest number of expanded nodes, (shown when it did not time out, at 1000 seconds). However, only the SPQR-based scheme had an overhead low enough to make its computation worthwhile when comparing runtime. Even for SPQR, the savings in number of expansions usually had to be by more than a factor of 2 in order to reduce the overall runtime. In cases where the reduction factor in number of expansions was only moderate, the incremental version of h_{BCC} was

Problem Instances	f^*	h_{BCC} exp	h(s)	t_{ni}	t_{inc}	$\hat{h}_{SPQR}$ exp	h(s)	t_{ni}	t_{inc}	$\hat{h}_{LP}$ exp	h(s)	t_{ni}	t_{inc}	$\hat{h}_{Flow}$ exp	h(s)	t_{ni}	t_{inc}
Hall	35	3295	71	9.50	6.35	**34**	38	0.20	**0.04**	**34**	35	128	128	**34**	38	27	27
G13	43	2122	47	4.64	**1.06**	**1546**	46	7.0	1.34	1550	46	144	16	1550	46	35	5
G21	121	>76K	135	T/O	T/O	**34874**	127	T/O	**264**		125	T/O	T/O		128	T/O	T/O
G53	47	304	52	0.67	**0.1**	208	51	2.34	0.3	**103**	49	96	13	**103**	49	24	3
G145	57	1761	60	7.44	1.52	**425**	58	3.20	**0.72**		58	T/O	347		60	T/O	622
G321	123	>63K	135	T/O	T/O	**31033**	131	902	**403**		131	T/O	T/O		132	T/O	T/O
G921	81	20339	86	159	69.0	**8769**	85	97.2	**20.1**		85	T/O	T/O		85	T/O	T/O
G1821	71	1267	78	2.07	0.26	**383**	71	1.88	**0.09**	**383**	71	11	0.7	**383**	71	4	0.2

Table 1: LSP Calibration Experiments Results

Problem Instances	Obstacles removed	f^*	h_{BCC} exp	h(s)	t_{ni}	t_{inc}	$\hat{h}_{SPQR}$ exp	h(s)	t_{inc}
M00	0	79	1108	106	1.44	**0.54**	**645**	90	0.84
M11	5	89	2200	111	4.37	**1.93**	**830**	99	4.03
M12	10	101	6195	116	20.96	**10.26**	**1400**	114	13.03
M13	15	111	3756	121	12.43	**8.69**	**1417**	119	21.47
M14	20	115	181716	126	14916.77	12681.40	**48170**	125	**1850.50**
M21	5	93	2746	111	4.83	2.05	**434**	97	**1.66**
M22	10	107	55602	116	777.51	692.94	**4518**	111	**42.85**
M31	5	85	2731	113	5.03	**2.45**	**1270**	103	5.39
M32	10	99	30297	118	261.95	223.48	**10837**	116	**128.75**
M33	15	111	82798	123	2439.5	2305.95	**22412**	121	**535.58**
M34	20	117	>113k	128	T/O	T/O	**68138**	127	**3726.85**

Table 2: Results for LSP on Maze, 3 random sequences all starting with M00

Problem Instances	Obstacles removed	f^*	h_{BCC+X} exp	h(s)	t	t_{inc}	h_{SPQR^+} exp	h(s)	t_{inc}	$\hat{h}_{SPQR^+ + Y}$ exp	h(s)	t_{inc}
S00	0	53	74	57	0.26	**0.01**	74	**55**	0.03	74	**55**	0.03
S11	1	53	143	62	0.56	**0.02**	143	**57**	0.04	143	**57**	0.04
S12	2	55	207	67	0.75	**0.04**	207	63	0.13	207	**62**	0.13
S13	3	55	2732	75	9.63	**1.0**	2678	69	2.31	2642	**66**	2.24
S21	1	55	159	63	0.59	**0.04**	153	60	0.12	153	**58**	0.12
S22	2	59	1463	96	13.62	**1.57**	1073	75	3.67	1047	**70**	2.3
S23	3	63	5831	111	57.66	13.49	2325	86	8.5	2147	**80**	**7.45**
S25	4	73	54181	162	1883.74	595.02	10735	99	70.04	4223	**88**	**16.5**
S31	1	53	350	71	1.22	**0.06**	234	63	0.1	230	**62**	0.09
S32	2	57	582	86	1.77	0.31	366	73	0.42	192	**69**	**0.24**
S33	3	63	3570	98	17.64	3.97	1693	79	2.35	1256	**73**	**1.79**
S34	4	77	8654	124	58.72	14.81	4841	103	11.35	3013	**93**	**7.14**

Table 3: Results for Snake on rooms map, all sequences starting with S00

Pr. Inst.	f^*	h_{BCC} exp	h(s)	t_{ni}	t_{inc}	$\hat{h}_{SPQR}$ exp	h(s)	t_{inc}
R1	55	78	57	0.1	**0.01**	**76**	55	0.02
R2	63	226	66	0.53	**0.03**	**84**	63	0.04
R3	69	654	72	1.15	**0.13**	**464**	70	0.32
R4	71	14721	91	88.27	79.4	**1215**	72	**1.08**
R5	79	230K	106	21K	18K	**7627**	82	**14.68**

Table 4: Results for LSP on Rooms Map Sequence

fastest. Also, obvious from these experiments is the fact that the flow techniques h_{Flow} and even more so F_{LP} reduce the number of expanded nodes. But they have a huge runtime overhead that makes them impractical. Thus they were dropped in the rest of the evaluation.

In our second experiment, to be more systematic, we generate problem instances in sequences of increasing difficulty. We use grid maps of mazes and rooms, with a predetermined s and t. For mazes, we start with the maze in Figure 4 (instance M00), and randomly remove 5 obstacles (black) to get the next instance. Here timeout=15K seconds.

For rooms, we start with room map lak105d.map from the *movingai* repository (Sturtevant 2012), picked as it is reasonably small with interesting room structure. We randomly *add* obstacles until there is no path from s to t. Algorithms were run starting from the easiest (containing most obstacles) problem. Sequences extend until some methods time out (Tables 2, 4).

Finally, Table 3 depicts results for instance sequences of Snake on different randomizations of obstacles in the rooms map. We compare our incremental h_{BCC} implementation to the baseline standard h_{BCC} implementation (both with X (star) patterns), and to our new SPQR-based heuristics, where $\hat{h}_{SPQR^+}$ uses only pruning and exclusion pairs and $\hat{h}_{SPQR^+ + Y}$ adds in the Y partitions. The advantage of our new heuristics increases with problem instance difficulty.

8 Conclusion

We developed a framework called Generalized LSP (GLSP) of constrained longest path problems, that generalizes most standard longest simple path problems, including LSP, Snake, and Euler paths. This enables using a common scheme for analysis, bounds, and for developing admissible search heuristics for these problems.

A non-trivial scheme using exclusion sets of size 2 is especially powerful. New heuristics based on this were developed, as well as efficient ways to approximate them. Our new heuristics, especially the variant based on SPQR trees, were shown to reduce the number of expanded nodes during search compared to existing methods, frequently leading to improved overall runtime despite their significant overhead.

Despite performing well, our rules based on SPQR trees still do not deliver all exclusion pairs, and it is challenging future work to find and prove a complete set of rules which can still be implemented efficiently. Combining the different types of exclusions in a way that does not require mutual disjointness is also a challenging issue that can further improve the accuracy of the resulting heuristics.

Acknowlegements

Supported by ISF grant #844/17, and by the Lynne and William Frankel center for CS at BGU.

References

Battista, G. D.; and Tamassia, R. 1996. On-Line Maintenance of Triconnected Components with SPQR-Trees. *Algorithmica*, 15(4): 302–318.

Chen, W. 2016. *The VLSI Handbook, Second Edition*, 77-31. Electrical Engineering Handbook. CRC Press. ISBN 9781420005967.

Cohen, Y.; Stern, R.; and Felner, A. 2020. Solving the Longest Simple Path Problem with Heuristic Search. In Beck, J. C.; Buffet, O.; Hoffmann, J.; Karpas, E.; and Sohrabi, S., eds., *Proceedings of the Thirtieth International Conference on Automated Planning and Scheduling, Nancy, France, October 26-30, 2020*, 75–79.

Dirac, G. A. 1952. Some Theorems on Abstract Graphs. *Proceedings of the London Mathematical Society*, s3-2(1): 69–81.

Dirac, G. A. 1960. In abstrakten Graphen vorhandene vollständige 4-Graphen und ihre Unterteilungen. *Mathematische Nachrichten*, 22(1-2): 61–85.

Felner, A.; Korf, R. E.; and Hanan, S. 2004. Additive pattern database heuristics. *Journal of Artificial Intelligence Research*, 22: 279–318.

Garey, M. R.; and Johnson, D. S. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences)*. W. H. Freeman, first edition edition. ISBN 0716710455.

Gutwenger, C.; and Mutzel, P. 2000. A linear time implementation of SPQR-trees. In *Proceedings of the 8th International Symposium on Graph Drawing*.

Karger, D.; Motwani, R.; and Ramkumar, G. D. 1997. On approximating the longest path in a graph. *Algorithmica*, 18(1): 82–98.

Karpas, E.; Betzalel, O.; Shimony, S. E.; Tolpin, D.; and Felner, A. 2018. Rational deployment of multiple heuristics in optimal state-space search. *Artif. Intell.*, 256: 181–210.

Kautz, W. H. 1958. Unit-Distance Error-Checking Codes. *j-IRE-TRANS-ELEC-COMPUT*, EC-7(2): 179–180.

Menger, K. 1927. Zur allgemeinen Kurventheorie. *Fundamenta Mathematicae*, 10(1): 96–115.

Palombo, A.; Stern, R.; Puzis, R.; Felner, A.; Kiesel, S.; and Ruml, W. 2015. Solving the Snake in the Box Problem with Heuristic Search: First Results. In Lelis, L.; and Stern, R., eds., *Proceedings of the Eighth Annual Symposium on Combinatorial Search, SOCS 2015, 11-13 June 2015, Ein Gedi, the Dead Sea, Israel*, 96–104. AAAI Press.

Portugal, D.; and Rocha, R. 2010. MSP Algorithm: Multi-robot Patrolling Based on Territory Allocation Using Balanced Graph Partitioning. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, 1271–1276. New York, NY, USA: ACM. ISBN 978-1-60558-639-7.

Robertson, N.; and Seymour, P. D. 1995. Graph Minors .XIII. The Disjoint Paths Problem. *J. Comb. Theory, Ser. B*, 63(1): 65–110.

Schmidt, K.; and Schmidt, E. G. 2010. A Longest-Path Problem for Evaluating the Worst-Case Packet Delay of Switched Ethernet. In *SIES*, 205–208. IEEE.

Stern, R.; Kiesel, S.; Puzis, R.; Felner, A.; and Ruml, W. 2014. Max is More than Min: Solving Maximization Problems with Heuristic Search. In *Symposium on Combinatorial Search (SOCS)*.

Sturtevant, N. 2012. Benchmarks for Grid-Based Pathfinding. *Transactions on Computational Intelligence and AI in Games*, 4(2): 144 – 148.

Westbrook, J. R.; and Tarjan, R. E. 1992. Maintaining Bridge-Connected and Biconnected Components On-Line. *Algorithmica*, 7(5&6): 433–464.

Wong, W. Y.; Lau, T. P.; and King, I. 2005. Information Retrieval in P2P Networks Using Genetic Algorithm. In *Special Interest Tracks and Posters of the 14th International Conference on World Wide Web*, WWW '05, 922–923. New York, NY, USA: ACM. ISBN 1-59593-051-5.

Zhang, L.; and Bacchus, F. 2012. MAXSAT Heuristics for Cost Optimal Planning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 26(1): 1846–1852.