Proceedings of the Seventeenth International Symposium on Combinatorial Search (SoCS 2024)

Generalized Longest Simple Path Problems: Speeding up Search Using SPQR Trees

Gal Dahan, Itay Tabib, Solomon Eyal Shimony, Yefim Dinitz
Dept. of Computer Science, Ben-Gurion University of the Negev, Israel
dahanga@post.bgu.ac.il, itaytab@post.bgu.ac.il, shimony@cs.bgu.ac.il, dinitz@bgu.ac.il

Abstract

The longest simple path and snake-in-a-box are combinatorial search problems of considerable research interest. Recent work has recast these problems as special cases of a generalized longest simple path (GLSP) framework, and showed how to generate improved search heuristics for them. The greatest reduction in search effort was based on SPQR tree rules, but it was posed as an open problem how to use them optimally. Unrelated to search, a theoretical paper on the existence of simple cycles that include three given edges answers such queries in linear time with SPQR trees. These theoretical results are utilized in this paper to develop advanced heuristics and search partitioning for GLSP. Empirical results on grid-based graphs show that these heuristics can result in orders of magnitude reduction in the number of expansions, as well as significantly reduced overall runtime in most cases.

Introduction

The longest simple path (LSP) problem is to find the Longest *Simple Path* (where no vertex is visited more than once) between two given vertices in an undirected 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 designed for minimization (MIN) problems to solve maximization (MAX) problems. They used LSP to demonstrate this and proposed an admissible heuristic for LSP. Then, (Palombo et al. 2015a) proposed several admissible heuristics for solving the *Snake-in-the-box* (SIB) problem (Kautz 1958). SIB is a variant of LSP that can help find efficient error correction codes. In SIB, a path may not use neighbours of vertices that are already in the path. 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.

These two longest path search problems (LSP and SIB), as well as others, were recast within a generalized longest simple path (GLSP) framework that includes these problems as special cases (Dahan et al. 2022). Heuristics based on biconnected components were shown to be applicable to many GLSP types, by discounting vertices that are guaranteed not to be on such a simple path. Using separation pairs and SPQR trees (Dahan et al. 2022), it was shown that the admissible heuristic bound could be greatly improved due to considering an independent set in an ”exclusion graph”, consisting of an edge for each vertex pairs that cannot be on a desired simple path. This was called the independent-set (IS) heuristic. Although the new SPQR-tree based heuristics led to a major improvement in number of expanded nodes and runtime, still two major issues remained open: the exclusion rules in (Dahan et al. 2022) did not deliver all possible exclusion pairs, and the independent set (being NP-hard to compute in general) was only approximated. A recent theoretical paper on conditions for the existence of a simple cycle that includes three given edges (Dinitz and Shimony 2023) provides an exact and efficient method (using SPQR trees) to find all exclusion pairs, but does not actually compute the heuristic value and does not apply the results to search.

This paper leverages off the theoretical results, proving that an exact independent-set heuristic can be computed efficiently (linear time). This heuristic dominates all the approximate versions of the IS heuristic from (Dahan et al. 2022). Implementing this heuristic results empirically in considerable savings in number of expansions and runtime for both LSP and Snake problems, our second contribution. Finally, we show that the SPQR structure allows for a partitioning of the search: solving a *weighted* longest simple path problem in individual triconnected components results in an overall longest path. This partitioning is valid beyond LSP: it holds for Snake problems, as well as constrained longest path problems with ”symmetric” constraint rules that are ”in between” LSP and Snakes. We develop a scheme exploiting partitioning, with empirical results again showing considerable search time reduction in hard problem instances.

Copyright © 2024, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

Background

We begin with background on longest path problems, solving them using exhaustive search, and admissible heuristics used therein in prior work. The latter requires concepts from graph decomposition: biconnected components and SPQR trees, also overviewed.

Generalized Longest Simple Path Problems

Let $G = (V, E, w)$ be a connected undirected weighted graph with no self-loops. A path from v_0 to v_m is an alternating sequence $P = (v_0, (v_0, v_1), v_1, (v_1, v_2), ...v_m)$ 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 - where constraining the path to be simple (LSP) is the most common.

The Generalized Longest Simple Path (GLSP) framework (Dahan et al. 2022) supports analysis of heuristics across several domains, including LSP and Snakes. We briefly repeat its essential definitions here.

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, a 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.*

As mentioned above, LSP and Snake are special cases of GLSP, as follows. Longest (vertex-wise) simple path (denoted as standard LSP): pairs in L are $(x, \{x\})$ for all vertices $x \in V$. Snake: pairs in L are $(x, N(x) \cup \{x\})$ where $N(x)$ are the immediate neighbours of x, for all $x \in V$. In this paper, we assume the G is unweighted (i.e. $w(x) = 1$ for all $x \in E \cup V$) unless explicitly stated otherwise.

Best-First Exhaustive Search in GLSP

We assume, unless stated otherwise, 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, and that the path has to end at target vertex t. Here $g(n)$ is naturally the length of P. Define the remaining graph, $G_r(G, P)$ to be G after removing all P vertices other than s' from G, as well as any other elements no longer accessible under constraint L: the edges incident to $P - \{s'\}$ in LSP, and also adjacent vertices in Snake. Clearly a state is not a dead end only if s', t are in the same connected component of $G_r(G, P)$.

In shortest-path problems, which aim to minimize a path length (called MIN problems) search algorithms such as A* prefer search nodes with lower $f(n) = g(n) + h(n)$ values. Symmetrically, for longest path problems (MAX problems), A* must be modified to prefer search nodes with the greatest $f(n)$ values, as done specifically for LSP (Stern et al. 2014).

Admissibility in MAX problems. A function h is said to be admissible for MAX path problems iff for every state n in the search space $h(n)$ is *greater than or equal to* the weight of the longest constrained path (longest simple path for LSP) from s' to t in $G_r(G, P)$.

Graphs and Connectivity Types

Undirected graph $G = (V, E)$ is *connected* if for every pair of vertices $u, v \in V$ there is a path from u to v in G. Obviously the longest simple path cannot have more edges than the number of vertices in G minus one. Different notions of connectivity play a crucial role in designing admissible heuristics in searching for longest paths (Dahan et al. 2022).

Path search literature (Cohen, Stern, and Felner 2020) uses the notion of *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.

Especially relevant to LSP in past work are 2-connectivity (biconectivity), and 3-connectivity (triconnectivity). A biconnected component (block) of G is any maximal biconnected subgraph of G. Every pair of biconnected blocks 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 biconnected block 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. taken from (Dahan et al. 2022), shows a graph (left), with two separators: y and x, splitting the graph into 3 biconnected blocks; and the corresponding block-cut tree (center).

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; Dinitz and Shimony 2023). SPQR trees have many technical details, one must read the cited papers to fully understand them. Here, we describe details essential to our work. SPQR trees consist of *components* of 4 types; each component represents a graph fragment. R (”Rigid”) components represent triconnected parts of G. P (”Parallel”) components represent a separator pair that separates the graph into three or more parts. S (”Series”) components represent a circular series of vertices. Q components represent individual edges, but are not used in our SPQR representation. See Figure 1 (right) for an SPQR tree of block B_3, which contains all vertices between x and t. The pair x, t gives rise to a component of type P which separates the graph into 3 parts, each of type S. In the figure, for each S

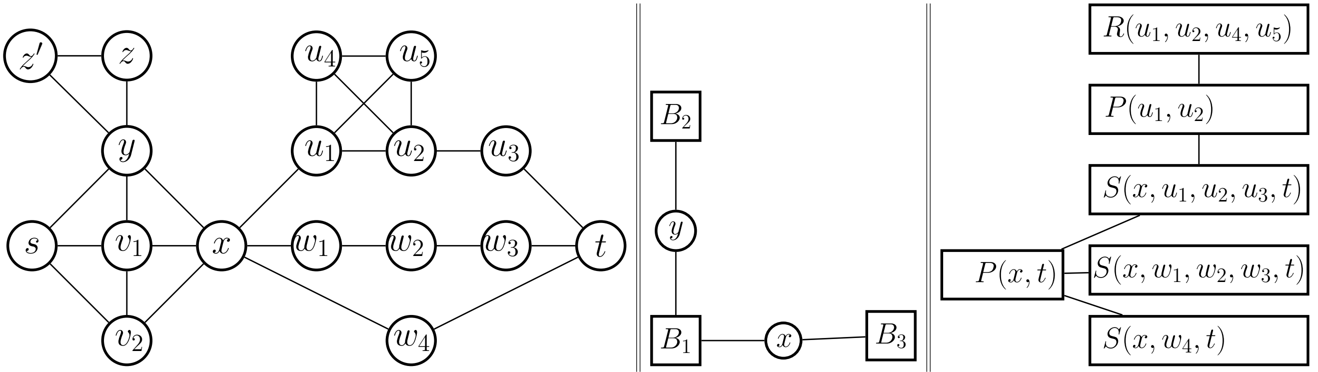

Figure 1: A graph (left), its biconnected block-cut tree (center), and simplified SPQR tree of block B_3 (right).

component 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 component, which abuts a triconnected component (an R component). Inside each component C, for each separation pair $u_1.u_2$, a *virtual edge* $e = (u_1, u_2)$ is added to *represent* each part of the SPQR tree (and thus also G) separated off by u_1, u_2. The part of the graph so represented is denoted by $\mathcal{T}(C, e)$. We denote by $V(\mathcal{T}(C, e))$ the set of vertices in $\mathcal{T}(C, e)$, and also use $V^-(\mathcal{T}(C, e)) = V(\mathcal{T}(C, e)) \setminus \{u_1, u_2\}$. Figure 2 is another example, with internal component structure visible, and virtual edges dashed. Virtual edge e_1 represents the top-left S component in the R component, and vice-versa. See (Westbrook and Tarjan 1992; Battista and Tamassia 1996) for details on SPQR trees, their properties, and constructing them in linear time (Gutwenger and Mutzel 2000). SPQR trees were used in (Dahan et al. 2022) to design heuristics, described below and improved in this paper.

Must-Include Paths and Cycles

A fundamental graph problem used in GLSP problems (Dahan et al. 2022; Palombo et al. 2015b), is: given a graph $G = (V, E)$ and a set of elements S, is there a simple path (or cycle) that includes all elements in S? Since this problem is hard in general, typically only $|S| \leq 3$ is used.

The following known relations between connectivity and must-include paths and cycles are useful to define GLSP search heuristics (Dahan et al. 2022):

Theorem 1. *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.*

Conversely, a biconnected block can only be entered or exited through a separator. Thus, a simple path (which cannot use a separator more than once) between vertices s and t can only visit biconnected blocks B obeying the following condition **B**: B is on the path from (a block containing) s to (a block containing) t in the block-cut tree of G. This observation leads directly the BCC heuristic (see next subsection). Together with Theorem 1 we have: vertex x can be on a simple path from s to t if and only if and only if x is in a biconnected block for which condition **B** holds.

Likewise, we might wish to find *exclusion pairs*: i.e. pairs of vertices v, w that cannot be on a simple path from s to t. A triconnected graph has no such pairs (Dahan et al. 2022):

Theorem 2. *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.*

Exclusion pairs can be found quickly using SPQR trees (Dahan et al. 2022; Dinitz and Shimony 2023). Let v, w be a separation pair that partitions G into disjoint (other than v, w) subgraphs $G_1, G_2, ..., G_k$, $k \geq 3$. This is called **case P** because it occurs in components of type P in the SPQR tree. 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. then vertices $x \in G_{i_1}$, $y \in G_{i_2}$, (both distinct from v, w) are exclusion pairs.

Case S (occurs in components of type S in the SPQR tree) is in a cycle of 4 or more separation pairs, noting that if s and t are on opposite sides of the cycle, only one side of the cycles can be traversed with a simple path.

In the graph of Figure 1 we have a P component (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 the subtrees rooted at these neighbors can be entered and exited, thus all pairs of vertices, one from each subtree, are exclusion pairs.

The above S and P rules were used to compute an SPQR-tree based heuristic (see next section), but do not deliver all possible exclusion pairs. An open question was how to find *all* exclusion pairs efficiently. A crucial idea in (Dinitz and Shimony 2023) was to add the edge (s, t) to graph G before constructing the SPQR tree $\mathcal{T}$. Then, one requires a simple cycle containing u, v, and the edge (s, t), which is equivalent to requiring a simple path from s to t via u, v without edge (s, t). If G was biconnected, this trick forces s, t to be in the same SPQT tree component C, thereby simplifying the required rules. If G is triconnected, for every set S of size 3 where not all elements are edges there exists in G a simple cycle that includes all elements in S. For the case of 3 edges, we have (Dinitz and Shimony 2023):

Theorem 3. *Let G be a triconnected graph, and e_1, e_3, e_3 be given edges in G. Then there exists a simple cycle that traverses e_1, e_2, e_3 if and only if these edges are not all incident on one vertex, and do not form an edge-cut of G.*

Together with Theorem 2, it was shown that the following scheme, called the Exclusion Pair Enumeration (EPE) algorithm, provably outputs exactly all exclusion pairs (Dinitz and Shimony 2023), as follows. Starting at the component containing the real edge (s, t) as the root, visit all components of the SPQR tree in a preorder. In each visited component C, exclusion pairs are emitted by calls to an EMITPAIRS(E', C) function that receives a set of virtual edges E' in C. Then, for each pair of edges $e_i, e_j \in E'$ with $i \neq j$, every vertex in $V^-(\mathcal{T}(C, e_i))$ forms an exclusion pair with each of the vertices in $V^-(\mathcal{T}(C, e_j))$. These exclusion pairs can be output explicitly, or implicitly by just listing the virtual edges in E'. For conciseness, define a predicate ExclusionP(E', C) which is true if and only if EMITPAIRS(E', C) is called by EPE when visiting component C.

It suffices here to define the output of EPE, using the ExclusionP predicate. Below are the only cases where ExclusionP is true. Denote by (s', t') the edge (real or virtual) by which EPE enters component C, with $(s', t') = (s, t)$ at the root. Denote by $E^{virt}(C)$ the set of virtual edges in C. If C is a P component, we have ExclusionP$(E^{virt}(C) \setminus (s', t')), C)$. This corresponds to **case P** above.

If C is an R component, we have ExclusionP(E', C) iff $E' \in E^{virt}(C) \setminus (s', t'))$ and one of the following conditions hold (corresponding to the cases in Theorem 3):

1. E' is a maximal set of virtual edges, all incident on s', or all incident on t', with $|E'| \geq 2$.
2. E' forms a 2-edge cut of $C' \setminus (s', t')$, and the edges of E' are not both incident on s' or t'.

To find all 2-edge cuts, EPE calls a linear runtime function FIND2EDGECUTS, also used below.

For example, in Figure 2 EPE starts with C being the R component and $(s', t') = (s, t)$ (additional edge (s, t) is not shown). We get ExclusionP$(\{e_1, e_2\}, C)$ (corresponding to exclusion pair (c_1, c_2)) and ExclusionP$(\{e_3, e_4\}, C)$ (exclusion pair (c_3, c_4)) due to case 1. Then EPE visits each S component, entering through the respective virtual edge e_i, but emitting no additional exclusion pairs as ExclusionP(E', C) is never true for S components. There are no exclusion pairs *inside* R components, due to Theorem 2. The runtime of the EPE algorithm is linear in the size of G with implicit output. Still, unlike the S and P rules above, EPE was never used to actually compute a search heuristic before, a contribution of this paper.

Existing Longest Path Search Heuristics

In searching for a path from s to t, biconnected blocks for which condition **B** above does not hold can be dropped from G, resulting in the admissible biconnected component heuristic h_{BCC} (Cohen, Stern, and Felner 2020). In Figure 1, s is in block B_1, and t is in block B_3. Block B_2 is not on the path from B_1 to B_3, and can be discarded. Equivalently, vertices z, z' cannot appear in any simple path from s to t.

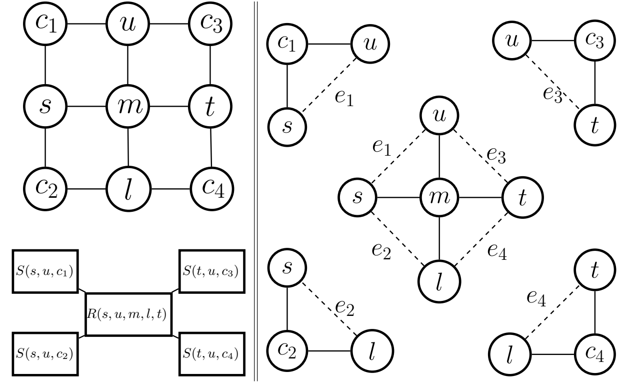

Figure 2: Graph, SPQR tree (left), SPQR components (right)

Theorem 4. *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 $h_{BCC}(G, s, t) = |V| - |S'| - 1$.*

Considering vertex pairs, (Dahan et al. 2022) proposed the following method. *Step 1:* Remove the above defined S' vertices from G, 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 $\{u, v\}$. Then we have (Dahan et al. 2022):

Theorem 5. *Every (s, t) path in $G = (V, E)$, constrained by L s.t. $\forall x \in V, x \in L(x)$, has length at most $\alpha(G_{ex}) + 1$, where $\alpha(.)$ is the maximum independent set size.*

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 vertices of block B_3, except for x, t. 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. The maximum independent set in this G_{ex} consists of all five u_i vertices, total size 5. Thus the bound is 6. In this graph the bound happens to be tight, i.e. equal to the number of edges in a LSP. Henceforth, denote $h_{IS}(G, s, t) = \alpha(G_{ex}) + 1$. Note that by construction h_{IS} dominates h_{BCC}.

Computing the maximum independent set is NP-hard, so it was approximated in (Dahan et al. 2022) by counting the number of maximal cliques in some (not necessarily disjoint) clique cover of an approximation to G_{ex} resulting from applying the S and P rules in the SPQR tree. The resulting heuristic was called $\hat{h}_{SPQR}$. This left two open issues: how to compute the exact exclusion graph efficiently, and how to compute or better approximate the size of the maximum independent set - both solved in this paper.

Heuristics for Snakes Constraints L tighter than simple path, such as Snake problems, allow for additional admissible heuristics. For Snake problems, the remaining graph is partitioned into disjoint sets of connected components (Palombo et al. 2015a). 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. It was observed that smaller patterns lead to better heuristics (Dahan et al. 2022), where the vertices in Y-shaped (star with only 3 spokes) and 2x2 square patterns also cannot all be visited by a Snake. In either case, the number of allowed vertices in each of these sets were added together into an admissible heuristic. With the introduction of exclusion-pair based heuristics in (Dahan et al. 2022), another open question is how to best combine such patterns with exclusion pairs (problematic when they overlap, e.g. when a vertex is in a 2x2 pattern as well as in an exclusion pair), another issue addressed in this paper.

Efficient Exact Computation of h_{IS}

We now adapt the methods of (Dinitz and Shimony 2023) to compute h_{IS}. Since our version computes the exact size of the maximum independent set of the complete G_{ex}, we denote this heuristic by h_{MIS}. Due to theorems shown therein, their EPE algorithm correctly emits all exclusion pairs. Thus, one can use the EPE algorithm to generate the exact exclusion graph G_{ex}, and then find its independent set size to compute h_{MIS}. However, we show that G_{ex} actually has a special structure allowing computation of its maximum independent set size without explicitly constructing G_{ex}. Thus, we modify the EPE algorithm to compute the value of h_{MIS} directly, without emitting the exclusion pairs.

This algorithm follows the same traversal of the SPQR tree as EPE, except that instead of emitting the exclusion-pairs in preorder, the number of vertices in a maximum independent set is computed post-order. The OPTCONSTRAINT Boolean function in line 5 is used to apply the heuristic to Snake and other problems, and always returns false for LSP. For conciseness, in recursive calls to TRAVERSE we use C_e to denote the component adjacent to C' in the SPQR tree that hangs off the virtual edge e.

As stated above, when EXCLUSIONP(E', C') and $e_1, e_2 \in E'$ with $e_1 \neq e_2$, all vertex pairs in $V^-(\mathcal{T}(C', e_1)) \times V^-(\mathcal{T}(C', e_2))$ are exclusion pairs in G_{ex}. Therefore an independent set among vertices in these subtrees cannot contain more vertices than $\max_{e \in E'} V^-(\mathcal{T}(C', e_i))$. However, subtrees hanging on virtual edges not in any EXCLUSIONP(E', C') are (mutually independent) independent sets of G_{ex}, so their sizes are summed.

Specifically, recall that EXCLUSIONP(E', C') never occurs in an S component, so the independent set sizes from the subtrees hanging on its virtual edges are summed. To that sum we add the number of vertices in the S cycle, not including the two vertices s', t'. In a P component EXCLUSIONP(E', C') where E' dentes all virtual edges in C^-, so the maximum number of vertices from the subtrees is taken. In an R component, we count all vertices other than s', t', as well as capturing the maximum independent set sizes of vertices due to the virtual edges: sum over subtrees from virtual edges not in any EXCLUSIONP(E', C') plus sum of maximum over edge sets E' for each EXCLUSIONP(E', C').

Algorithm 1: Compute $h_{MIS}(G)$

Input: a biconnected graph G and two of its vertices s, t
1: Compute SPQR tree $\mathcal{T} = \mathcal{T}(G \cup \{(s, t)\})$
2: Root $\mathcal{T}$ at component C where $e = (s, t)$ is a real edge
3: **return** (TRAVERSE(C, e)+1)
4: **function** TRAVERSE($C', (s', t')$)
5: **if** OPTCONSTRAINT(G, s', t') **then**
6: **return** (0)
7: Let $C^- = C' \setminus (s', t')$; Let $\mathcal{E}$ = virtual edges of C^-
8: **if** C' is a P component **then**
9: **return** $(\max_{e \in \mathcal{E}}$ TRAVERSE$(C_e, e))$
10: **if** C' is an S component **then**
11: **return** $(|V(C')| - 2 + \sum_{e \in \mathcal{E}}$TRAVERSE$(C_e, e))$
12: /* (C' is an R component) */
13: Let $Tot = |V(C')| - 2$; Let $E_{NEX} = \mathcal{E}$
14: **for** $u \in \{s', t'\}$ **do**
15: Let $E_{EX} = \{(u, v) | (u, v) \in \mathcal{E})\}$
16: **if** $|E_{EX}| \geq 2$ **then**
17: Tot += $\max_{e \in E_{EX}}$ TRAVERSE(C_e, e)
18: $E_{NEX} = E_{NEX} \setminus E_{EX}$
19: Let $Cuts$ = FIND2EDGECUTS$(C^-, (s', t'))$
20: **for** all $E_{EX} \in Cuts$ s.t. $E_{EX} \subseteq \mathcal{E}$
21: and E_{EX} not incident on s' or t' **do**
22: Tot += $\max_{e \in E_{EX}}$ TRAVERSE(C_e, e)
23: $E_{NEX} = E_{NEX} \setminus E_{EX}$
24: Tot += $\sum_{e \in E_{NEX}}$ TRAVERSE(C_e, e)
25: **return** (Tot)

In order to prove corectness of the computation, we introduce the concept of an undirected abstract exclusion graph $A = A(G, s, t) = (V_A, E_A)$, which is defined using the the SPQR tree $\mathcal{T}$ of $G \cup (s, t)$ rooted at the component that includes the real edge (s, t). The "abstract" vertices of A consist of all the virtual edges in $\mathcal{T}$. There is an edge (v_1, v_2) in E_A just when there exist an exclusion pair $u, w \in G$ such that u is in the subtree of $\mathcal{T}$ represented by v_1, and w is in the subtree represented by v_2. By construction, this occurs if ExclusionP(E', C) for some component C, and exists E' such that $v_1, v_2 \in E'$. The following is an important lemma:

Lemma 1. *All the connected components of abstract exclusion graph $A(G, s, t)$ are cliques.*

Proof. By construction, if EXCLUSIONP(E', C) then there is a clique in $A = A(G, s, t)$ among the abstract vertices (standing for abstract edges) E'. Conversely, there is an edge between abstract vertices $v_1, v_2 \in A$ only if EXCLUSIONP(E', C) for some C and E' such that $v_1, v_2 \in E'$. So it is sufficient to prove that these cliques do not overlap, or formally:

Claim 1. *Let C be a component, and $E_1, E_2 ... E_k$ be different edge sets such that ExclusionP(E_i, C) for all $1 \leq i \leq k$. Then $e \in E_i$ for at most one $1 \leq i \leq k$.*

The claim holds vacuously for S components, and trivially for P components. Therefore, assume that C is an R component. To prove the claim by contradiction, let e be a

virtual edge in $C^- = (C - (s', t'))$ and $e \in E_i, e \in E_j$ for some $i \neq j$. There are three cases to consider:
1. $E_i, E_j \in Cuts$. This is impossible, since (Dinitz and Shimony 2023) showed that 2-edge cuts cannot share an edge in C^- of an R component.
2. E_i are all incident on s', E_j are all incident on t', or vice versa, in C^-. Then $e = (s', t') \in C^-$ (contradiction).
3. E_i is incident on s' (or on t'), and $E_j \in Cuts$. Let $E_j = \{(s', w), (u, v)\}$ s.t. u is on the s' side of the cut E_j. Then s', v separates w from u in both C^- and C, and is thus a vertex 2-cut of C. So C is not triconnected, a contradiction. A similar contradiction occurs for E_i incident on t'. □

Lemma 1 enables the main result (proved in Appendix):

Theorem 6. *The h_{MIS} algorithm correctly computes $h_{MIS}(G)$ in time linear in the size of G. (That is, $O(|E|)$.)*

Linear time is due to the complexity of EPE with implicit output. Actually achieving the linear time requires careful attention to implementation detail, e.g. set operations and membership tests in lines 15-18, 20, and 23 should be done by appropriate tagging of elements, rather than general-purpose set operation functions. (These optimizations were not done for the empirical evaluation in this paper.)

Adapting h_{MIS} to Other Longest Path Problems

Recall that a GLSP problem is an LSP problem if its constraint rule has $L(x) = \{x\}$ for every vertex x, and is a Snake problem if its constraint rule likewise has $L(x) = N(x) \cup \{x\}$. Consider constraints "in-between" these cases: L such that $\{x\} \subseteq L(x) \subseteq N(x) \cup \{x\}$; we call such constraints *local vertex constraints*. A local vertex constraint L is called *symmetric* if for every pair of vertices $x \neq y$, we have $x \in L(y) \Leftrightarrow y \in L(x)$.

Obviously h_{MIS} is admissible for all problems with local vertex constraints. Additionaly, when $x \in L(y)$, a path traversing both x and y must have x and y adjacent on the path. So h_{MIS} can be tightened for symmetric local vertex constraints by defining the function OptConstraint(G, x, y) to return true when $x \in L(y)$ and $(x, y) \in G$. Note that OptConstraint defined this way always returns false for LSP, and returns true for Snake just when $(x, y) \in G$.

To combine h_{MIS} with pattern heuristics, we note that after setting Tot to $V(C') - 2$ in R components, we can use any disjoint pattern-based heuristic, such as 2x2 squares for Snakes, and deduct 1 from Tot for any such pattern S, as long as $V(S) \subseteq C'$. The resulting heuristic is still admissible, despite the fact that a vertex v in such a pattern may also be in an exclusion pair. That is due the following property: let $EX(v)$ be the set of vertices which form an exclusion pair with v. Then $EX(u) = EX(v)$ for all $u, v \in S \subseteq C'$.

Search Partitioning with SPQR Trees

Obviously, search for simple paths (both shortest or longest) can proceed independently in different biconnected components. A similar, albeit somewhat more complicated, scheme can also partition search among SPQR tree components. The crucial observation is that a simple path in the parts of G corresponding to a subtree of the SPQR tree hanging on any virtual edge e is independent of any simple path in the rest of the SPQR tree. So in fact the equations used to agglomerate the heuristic value in the h_{MIS} computation algorithm can also be used as-is for the length of the longest simple path between the ends of e in each subtree.

By calling TraverseP (algorithm) 2 that traverses the tree the same way as Traverse, we we get an actual longest constrained path. Here, in a P component, the max means comparison of paths by path length. The function SpliceVirtuals(p) receives a path p consisting of real and virtual edges, and replaces each virtual edge e in p by the appropriate path $p(e)$ computed by a recursive call. In an S node, p is the unique path from s' to t' that does not include (s', t'). In an R node, do the traversal for every virtual edge, and give each virtual edge a weight equal to the length of the returned path. All other edges have a weight of 1. Then search in the current R component for the heaviest weighted simple path between s', t', (function MaxWeightedPath) using, for example, A*.

Algorithm 2: TraverseP for Computing LSP

```
1:  function TraverseP(C'_e, (s', t'))
2:      if OptConstraint(G, s', t') then
3:          return ((s', t'))
4:      Let C^- = C' \ (s', t'); Let 𝓔 = virtual edges of C^-
5:      for e ∈ 𝓔 do
6:          Let p(e) = TraverseP((C_e, e), C')
7:      if C' is a P component then
8:          return (max_{e∈𝓔} p(e))
9:      if C' is an S component then
10:         Let p be the unique path from s' to t' in C^-.
11:         return (SpliceVirtuals(p))
12:     /* (C' is an R component) */
13:     for e ∈ E(C^-) do
14:         Let w(e)=1
15:         if e ∈ 𝓔 then
16:             Let w(e) = |p(e)|
17:     Let p =MaxWeightedPath(C^-, s', t')
18:     return (SpliceVirtuals(p))
```

Partitioning in Other GLSP Problem Types

TraverseP supports Snake problem partitioning when OptConstraint is defined to return true just when $(s', t') \in G$. More in general, defining OptConstraint appropriately as above, the decomposition supports any symmetric local vertex constraint. Obviously, MaxWeightedPath must be a function returning the path of maximum weight that does not violate constraint L within the R component C', rather than just a simple path.

Empirical Evaluation

We have tested the effectiveness of our new heuristics and partition algorithms mostly on 4-connected grids, with the exception being binary hypercube for SIB. We expect our

Problem Instances	f^*	$\hat{h}_{BCC}$ exp	$\hat{h}_{BCC}$ h(s)	$\hat{h}_{BCC}$ t	(old) $\hat{h}_{SPQR}$ exp	(old) $\hat{h}_{SPQR}$ h(s)	(old) $\hat{h}_{SPQR}$ t	Naive $\hat{h}_{MIS}$ exp	Naive $\hat{h}_{MIS}$ h(s)	Naive $\hat{h}_{MIS}$ t	h_{MIS} exp	h_{MIS} h(s)	h_{MIS} t	Partition exp	Partition t
Showcase	128	158K	286	T/O	104K	160	T/O	127	**128**	0.51	127	**128**	**0.09**	153K	T/O
Maze	85	694	109	**0.19**	364	103	1.10	268	**99**	0.53	268	**99**	0.52	**210**	0.45
Maze-A-1	91	102K	113	1160	43K	109	264	28K	**105**	100	28K	**105**	98	**824**	**1.37**
Maze-A-2	101	267K	117	T/O	257K	**113**	T/O	273K	**113**	T/O	279K	**113**	T/O	**22K**	**126**
Maze-B-1	93	28K	113	85	5724	107	24	3749	**103**	7.4	3749	**103**	7.0	**1389**	**2.4**
Maze-B-2	95	280K	117	T/O	112K	114	2923	57K	**110**	631	57K	**110**	645	**14K**	**44**

Table 1: LSP Results

h_{MIS} to gain mostly when the true exclusion graph has exclusion pairs not detected by SPQR rules in prior work, i.e. when the SPQR tree of the graph contains converging virtual edges, as in Figures 2, and 3 (left). To be more systematic, we also generate problem instances in sequences of increasing difficulty. We use grid maps of mazes, with a pre-determined s and t. We start with the maze in Figure 3 (right), and remove a 5x5 obstacle region (black) shaped as depicted below at a random location to get the next instance, repeating until all runs time out at 15K seconds:

```
  X
 XXX
XXXXX
 XXX
  X
```

Experiments were run on AMD Ryzen 9/3900X 12-Core @3.80GHz with 64.0GB, 2667MHz RAM, programmed in Python and Sagemath.

Typical results appear in Table 1 for the following, all using A*: (I) BCC heuristic (h_{BCC}, to verify (Dahan et al. 2022) results), (II) the SPQR heuristic $\hat{h}_{SPQR}$ from (Dahan et al. 2022), (III) "naive" new SPQR heuristic ("Naive" $\hat{h}_{MIS}$), with G_{ex} created using EPE, and greedy clique cover to approximate the MIS, (IV) our h_{MIS} heuristic, and (V) the SPQR-based partitioning scheme, using h_{MIS} inside R components. 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. (2) The value of the heuristic at the initial state, denoted by $h(s)$. (3) Runtime in seconds, with T/O indicating time out. For the partitioning scheme, the number of expansions reported is the total for all search runs (one in each R component), and $h(s)$ is undefined and thus not reported. **Bold** fonts indicate the best variant(s). We can see that, confirming the results in (Dahan et al. 2022), for easy problem instances $\hat{h}_{BCC}$ is the fastest due to having the smallest overhead. As the instances grow harder, the SPQR-based schemes become relatively faster, with an advantage for our new h_{MIS} schemes (in Showcase this is several orders of magnitude). In most cases h_{MIS} performed no better than the naive version, but is still recommended, as it is guaranteed to compute h_{MIS} exactly and is simpler to implement. For hard instances, partitioning achieves several orders of magnitude improvement; but performs poorly in the Showcase example: it attempts to find a longest path in the large R component, where, due to our h_{MIS} heuristic, non-partitioned A* never expands paths at all.

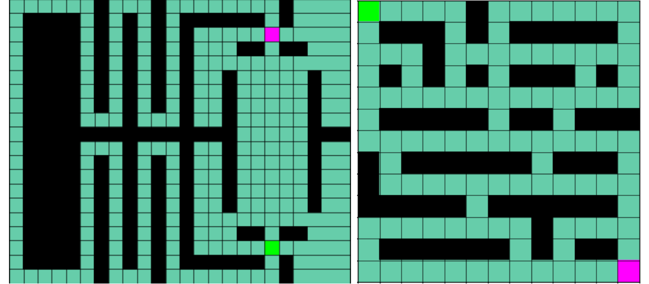

Figure 3: Showcase (left), Maze (right). Green: s. Purple: t

Prob. Inst.	f^*	$\hat{h}_{BCC}$ exp	$\hat{h}_{BCC}$ t	$\hat{h}_{SPQR}$ exp	$\hat{h}_{SPQR}$ t	h_{MIS} exp	h_{MIS} t	Partition exp	Partition t
6D-H SIB	26	48K	1486	22K	493	**19K**	**383**	96K	T/O
Maze	85	394	0.48	177	0.76	**129**	**0.43**	135	0.56
Maze-B-1	87	1541	2.13	556	2.60	328	1.07	**228**	**0.56**
Maze-B-2	87	13K	37.6	2216	19.2	**853**	6.89	875	**4.87**
Maze-B-3	89	46K	380	10K	82	3322	25	**1648**	**11**

Table 2: Snake Results

Finally, Table 2 depicts typical results for Snake and Snake-in-a-box. We compare the same heuristics as above (only one h_{MIS} version as their perfomance was roughy equal), combined with the pattern heuristics Y and 2x2. These results show that our new heuristics and partitioning also perform well in the Snake domain,

Conclusion

We have leveraged new theoretical results on must-include cycles in graphs to improve both the effectiveness and runtime of state-of-the-art admissible heuristics in a range of longest path problem types. The insights from our new scheme also allow a new type of longest-path problem partitioning, leading to orders-of-magnitude additional reduction in the number of search expansions and runtime. Despite that, there is clear room for future work, low-hanging fruit would be to improve the partitioning scheme by first running a round of h_{MIS} computation, sort the subtrees by that value, and then when computing the max in P components, avoid recursive calls unless their heuristic value is greater than the length of all previously computed paths.

That should overcome the shortcoming of partitioning evident in the showcase example. One could also consider whether to employ dynamic partitioning (during search), a much more complicated issue.

Appendix: Proof of Theorem 6

Denote by $EX(v)$ the set of vertices u such that (u, v) are an exclusion pair. Let us denote by $G(C, e)$ the subgraph of G induced by $V^-(\mathcal{T}(C.e))$, and $G_{ex}(V)$ the subgraph of G_{ex} induced by V.

Lemma 2. *G_{ex} is hierarchical, with a hierarchy corresponding to the SPQR tree, rooted at the component containing real edge (s,t). That is:*

1. *For every component C, and every $v, v' \in V^-(C)$, we have: $EX(v) = EX(v')$, $v \notin EX(v')$,*
2. *Let C, C' be components such that C' is a descendant of C, and vertices $v \in V^-(C), v' \in V^-(C')$. Then $EX(v) \subseteq EX(v')$ and $v' \notin EX(v)$.*
3. *For every component C, and every virtual edge pair $e_1, e_2 \in C$, $e_1 \neq e_2$ and every pair of vertices $(x, y) \in V^-(\mathcal{T}(C, e_1)) \times V^-(\mathcal{T}(C, e_2))$, we have $(x, y) \in G_{ex}$ if and only if* EXCLUSIONP(E', C) *for some E' where e_1, e_2 in E'.*

Proof. Immediate from the definition of EXCLUSIONP. □

Note that the first claim above implies that if $u \in C^-, v \in C'^-$ are an exclusion pair, then so is any other pair of vertices from C^-, C'^- respectively. The third claim means that C is a "central" for $x, y, (s,t)$, i.e. is the only component determining whether $\{x, y\}$ is an exclusion pair w.r.t. (s, t).

Algorithm 3: Compute $MIS(G_{ex})$

1: **function** TRAVERSES$(C', (s', t'))$
2: **if** OPTCONSTRAINT(G, s', t') **then**
3: **return** (0)
4: Let $C^- = C' \setminus (s', t')$; Let $\mathcal{E}$ = virtual edges of C^-
5: **if** C' is a P component **then**
6: **return** $(arg \max_{e \in \mathcal{E}}$ TRAVERSES$(C_e, e))$
7: **if** C' is an S component **then**
8: **return** $(V^-(C') \cup \bigcup_{e \in \mathcal{E}}$TRAVERSES$(C_e, e))$
9: /* (C' is an R component) */
10: Let $I = |V^-(C')|$; Let $E_{NEX} = \mathcal{E}$
11: **for** $u \in \{s', t'\}$ **do**
12: Let $E_{EX} = \{(u, v) | (u, v) \in \mathcal{E}\}$
13: **if** $|E_{EX}| \geq 2$ **then**
14: $I = I \cup \max_{e \in E_{EX}}$ TRAVERSES(C_e, e)
15: $E_{NEX} = E_{NEX} \setminus E_{EX}$
16: Let $Cuts$ = FIND2EDGECUTS$(C^-, (s', t'))$
17: **for** all $E_{EX} \in Cuts$ s.t. $E_{EX} \subseteq \mathcal{E}$
18: and E_{EX} not incident on s' or t' **do**
19: $I = I \cup \max_{e \in E_{EX}}$ TRAVERSES(C_e, e)
20: $E_{NEX} = E_{NEX} \setminus E_{EX}$
21: $I = I \cup \bigcup_{e \in E_{NEX}}$ TRAVERSES(C_e, e)
22: **return** (I)

Proof. (of theorem) We can show that TRAVERSE with minor modification, called TRAVERSES actually computes a MIS of G_{ex}. Instead of returning numbers, make TRAVERSES return vertex-sets: every maximization returns a set of maximum set cardinality from its arguments, and addition is treated as (disjoint) set union. The max operators among sets should be read as returning a set of maximum cardinality. For an S node always add all its vertices except for s', t' (union with the union of all sets from recursive calls, if any), and for an R node always add all its vertices (other than s', t') to the appropriate independent set(s) from recursive calls, if any.

We prove by structural induction that the set of vertices returned by TRAVERSES(C, s', t') is a MIS of $G_{ex}(V(G(C, (s', t'))))$. In the base case of a leaf component, C^- has no virtual edges. There are no exclusion pairs within any component C. Indeed in both S and R components, all vertices of $V^-(C)$ form an independent set of $G_{ex}(V(G(C, (s', t'))))$ and are returned, as required. P components can never be leaves of the SPQR tree, as a P component must have at least 2 virtual edges.

Now to show the inductive step for each component type. **For a S component**, there are no exclusion pairs between any $V^-(\mathcal{T}(C', e_i))$, $V^-(\mathcal{T}(C', e_j))$. Denote by I_i an arbitrary independent set of $G_{ex}(V(G(C', e_i)))$, and:

$$I = \bigcup_{e_i \in E^{virt}(C^-)} I_i$$

Then I is an independent set of $G_{ex}(V(G(C, (s', t'))))$, and in particular for each I_i being a MIS of $G_{ex}(V(G(C', e_i)))$. Due to Lemma 2, there is no exclusion pair x, y with $x \in V^-(C)$ and $y \in \mathcal{T}(C, e_i) \cup V^-(C)$ for any i, therefore, by Lemma 2 item 3, $I \cup V^-(C)$ is an independent set of $G_{ex}(V(G(C, (s', t'))))$, as required. Since the independent sets I_i are disjoint, the union size is maximized when the size of each I_i is maximized, as required.

For a P component, every vertex pair from $V^-(\mathcal{T}(C'_{e_i}, e_i)) \times V^-(\mathcal{T}(C_{e_j}, e_j))$ forms an exclusion pair. Therefore, an independent set in $G_{ex}(V(G(C, (s', t'))))$ must be an independent set I_i in some $G_{ex}(V(G(C', e_i)))$. Since the I_i are disjoint, the maximum independent set is achieved by taking the maximum-size I_i, as required.

For an R component, from each clique in the abstract exclusion graph, only vertices from one abstract vertex (i.e. virtual edge) can be in any independent set. Since all the (maximal) cliques in the abstract exclusion graph are independent, then the union of independent sets, one from each clique, is an independent set, and has maximum size iff the maximum size set is taken from each of the cliques. Again due to Lemma 2, there is no exclusion pair x, y with $x \in V^-(C)$ and $y \in \mathcal{T}(C, e_i) \cup V^-(C)$ for any i, so all vertices of $V^-(C)$ are also in the MIS of $G_{ex}(V(G(C, (s', t'))))$ as required; this is exactly what is computed by TRAVERSES.

This completes the inductive proof that TRAVERSES returns a MIS of G_{ex}. By construction, TRAVERSE returns the size of the set returned by TRAVERSES, thus TRAVERSE returns $\alpha(G_{ex})$. □

Acknowledgements

Supported by ISF grant #844/17.

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.

Dahan, G.; Tabib, I.; Shimony, S. E.; and Felner, A. 2022. Generalized Longest Path Problems. In *Symposium on Combinatorial Search*, 56–64.

Dinitz, Y.; and Shimony, S. E. 2023. On Existence of Must-Include Paths and Cycles in Undirected Graphs. *CoRR*, abs/2302.09614.

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.

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

Palombo, A.; Stern, R.; Puzis, R.; Felner, A.; Kiesel, S.; and Ruml, W. 2015a. 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.

Palombo, A.; Stern, R.; Puzis, R.; Felner, A.; Kiesel, S.; and Ruml, W. 2015b. Solving the Snake in the Box Problem with Heuristic Search: First Results. In *Symposium on Combinatorial Search (SoCS)*, 96–104.

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.

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)*.

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.