Solving the Longest Simple Path Problem With Heuristic Search

Submission #59

Abstract

The aim in the longest simple path (LSP) problem is to find the longest simple path in a graph. LSP is a fundamental problem in graph theory with applications in a variety of domains including VLSI design, error correction code, and robot patrolling. This problem is known to be NP-hard. Prior approaches for solving LSP included using constraints solvers and genetic algorithms. However, very few works have applied systematic heuristic search techniques such as A*. In this work, we make several advancements towards solving the LSP problem with heuristic search. First, we introduce several methods for pruning path prefixes that are dominated by others. Then, we propose several admissible heuristic functions for this problem, based on prior work on the snake-in-the-box problem. Experimental results demonstrate the large impact of the proposed heuristics and pruning rules.

1 Introduction

The longest simple path (LSP) problem is the problem of finding the longest simple path in a graph. LSP is a fundamental problem in graph theory, that is known to be NP-hard, and even hard to approximate within a constant factor (Karger, Motwani, and Ramkumar 1997). The motivation to solve the LSP problem 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). Snake-in-the-box is a particular variant of LSP that is important for a useful type of efficient error correction codes (Kautz 1958).

In this work we approach LSP as a heuristic search problem. While there are many theory and algorithm for solving the shortest path problem, LSP was hardly studied as a search problem. Indeed, there are several challenges in trying to solve LSP with heuristic search. In particular, the requirement that the solution path should be simple entails a blowup in the size of the search space. This is because a state represents a path and not a vertex. Furthermore, in LSP we are looking for a solution with the maximal cost not the minimal cost. Thus, applying heuristic search for such problems is not trivial (Stern et al. 2014).

To address this, we propose several methods to detect and prune states that are dominated by other states. We define this notion of domination between states formally, and provide two techniques for identifying when a state is dominated by another. We then show how to integrate these pruning methods into A* and into Depth-First Branch-and-Bound (DFBnB).

Then, we propose several admissible heuristics for LSP. This is especially challenging because most prior work in developing heuristics have been on minimization problems and on finding shorter paths. Finally, we perform a comprehensive set of experiments over grid map domains. Our results show that while some pruning is always useful, more sophisticated pruning is sometimes not worthwhile.

Several approaches where proposed for solving the LSP problem. Some prior work compiled a given LSP problem to a constraint optimization problem and used a constraints solver (Pham and Deville 2012). Others used genetic algorithms (Portugal, Antunes, and Rocha 2010). While LSP in general is NP-Hard, it can be solved in polynomial time for certain classes of graphs (Keshavarz-Kohjerdi, Bagheri, and Asgharian-Sardroud 2012). To the best of our knowledge, only two prior works approached LSP as a heuristic search problem. Palombo et al. (2015) proposed several admissible heuristics for solving the snake in the box (SIB) problem, which is a special case of LSP. We build on these heuristics, adapted them to LSP, and propose novel heuristics as well as pruning techniques. Stern et al. (2014) focused on general maximization problem and how to adapt heuristic search algorithms to solve them, and used LSP as a domain to demonstrate their results. Using a combination of our novel pruning techniques and heuristics, we are able to improve on their results by a up to a factor of 20.

2 Background

State space search algorithms are defined with respect to a state space. They are given (1) an initial state, (2) a set of operators that define how to move from one state to another, and (3) a goal condition that defines which states are goal states. The output of a state space search algorithm is a path, i.e., a sequence of operators, from the initial state to a goal state. In general, each state transition operator o is associated with a cost denoted $cost$(o), and the cost of a path is the sum of costs of its constituent operators. For ease of exposition, we will assume in this paper that the cost of all operators is one, and so the length of a path corresponds to its cost.

A natural and common application of state space search algorithms is the well-known shortest path (SP) problem. The SP problem is defined w.r.t a directed graph $G = (V, E)$, a start vertex $s \in V$, and a goal vertex $g \in V$. A solution to a SP problem is a path in G from s to g such that no other path from s to g is shorter. The corresponding state space is defined as follows: a state represents a vertex in V, the initial state is s, the state transition operators correspond to outgoing edges, and the goal state represents the vertex g.

2.1 A*

The A* (Hart, Nilsson, and Raphael 1968) algorithm is a popular state-space search algorithm that is often used to solve SP problems. A* maintains two lists of states called OPEN and CLOSED. For every generated state, A* maintains the length of the shortest path found so far from s to that state. The latter is referred to as the g value of that state. A* starts by adding the initial state to OPEN and setting its g value to zero.

In every iteration, a state n is popped out of OPEN and moved to CLOSE. Then, every applicable state transition operator is applied to n. This results in a set of states, one per applicable operator. This process is referred to as expanding n and the resulting set of states are referred to as the child states of n. Every child state c that was not generated before is added to OPEN, and its g value is set to $g(n)$+cost(o), where o is the operator applied to n to generate c. In some state spaces, there can be multiple paths to the same state. This may cause a state c to be generated more than once. In such a case, when expanding a state n and generating a child state c, it may be the case that c is already in OPEN or CLOSED, and already has a g value. When this occurs, A* checks if its current g value is smaller than or equal to $g(n) + cost(o)$. If it is, then c is not inserted again to OPEN. Otherwise, the g value of c is updated to be $g(n) + cost(o)$, and it is re-inserted to OPEN, removing the older copy of c if it is in OPEN (it may already be in CLOSED). This mechanism for keeping only the shortest path to each state is referred to as pruning dominating paths or duplicate detection, and is known to be very important in terms of runtime in some domains. Thus, many heuristic search algorithms implement pruning dominating paths mechanisms.

To choose which state to pop from OPEN in every iteration, A* uses a heuristic function, denoted h, that estimates the length of the shortest path from a given state to a goal state. If that heuristic function never over-estimates the length of the shortest path to a goal then it is called an admissible heuristic function. A* chooses in every iteration to pop the state n in OPEN with the lowest $g(n) + h(n)$ value. If h is an admissible heuristic, then when a goal state is expanded it is guaranteed that the lowest cost path to a goal has been found, and A* halts. Powerful admissible heuristics have been proposed over the years for a wide range of problems, and having a strong heuristic function can lead to orders of magnitude speedup.

2.2 Depth-First Branch-and-Bound (DFBnB)

DFBnB is a simple search algorithm: it performs a depth-first search, keeps track of the cost of the best solution found so far, and prunes any path in the state space that is more costly than the cost of the incumbent solution.

DFBnB can be coupled with an admissible heuristic, pruning every state n for which $g(n) + h(n)$ is equal to or greater than the cost of the incumbent solution. DFBnB is an anytime algorithm, and the solution it returns after pruning all branches in the state space is guaranteed to be optimal.

3 The Longest Simple Path Problem

A path in a graph is called simple if it never passes through the same vertex twice. Note that every solution to a SP problem must be simple, as otherwise a shorter path would exist.[1] Thus, there is no meaningful difference between the SP problem and the problem of finding the shortest simple path. This is not the case for the problem of finding the longest path: finding the longest path in a graph can be done in time that is polynomial in the number of graph vertices while finding the longest simple path is NP-Hard (Karger, Motwani, and Ramkumar 1997).

Definition 1 (The longest simple path problem). The Longest Simple Path (LSP) problem is defined w.r.t a directed graph $G = (V, E)$, a start vertex $s \in V$, and a goal vertex $g \in V$. A solution to LSP is a simple path from s to g such that no other simple path from s to g is longer.

In this work we explore how to apply heuristic search algorithms to solve the LSP problem. In particular, we focused on two popular heuristic search algorithms: A* and DFBnB with a heuristic.

To use A* or DFBnB to solve an LSP problem, one needs to first define a corresponding state space. For an SP problem, a state represented a vertex. This is not sufficient for LSP, since to know which operators are applicable in a vertex n one must know the vertices in

[1] This is not true if state transition operators can have zero or negative cost.

the path from s to n so as to make sure they are not added twice the resulting path. Thus, following Stern et al. (2014), we define a state in the LSP state space to represent a path in the underlying graph G from s to a vertex n.[2] To distinguish between a path in the underlying graph G and a path in the LSP state space, we will refer to the former as a graph-path, and for an LSP state N we use $N.\pi$ to denote its corresponding graph-path. The applicable state transition operators from an LSP state N correspond to extending $N.\pi$ by extending it with a single edge.

3.1 A* and DFBnB for LSP

LSP is a maximization (MAX) problem: the objective is to find the graph-path with the maximal length. In general, some modifications to textbook A* are needed in order to make it compatible to MAX problems (Stern et al. 2014). These modifications include changing the definition of admissibility and changing how A* choose which state to pop from OPEN in every iteration.

- 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 larger than or equal to the length of the longest simple graph-path from n to g.
- Choosing from OPEN. In MIN problems, A* pops from OPEN in every iteration the state n with the lowest $g(N) + h(N)$. In MAX problems, A* needs to pop from OPEN in every iteration the state N with the highest $g(N) + h(N)$.

In general MAX problems, there is third change that needs to be done to A* so that it works for MAX problems — changing the stopping condition. In MAX problems, it may be the case that the optimal solution is to reach one goal state and continue the path to reach another (e.g., to gain more reward). Thus, expanding a goal is not a sufficient condition to guarantee optimality (Stern et al. 2014). However, in LSP this modification is not needed because there is a single goal vertex g and since a solution must be a simple path we know that every optimal solution to LSP is a path that reaches the goal vertex exactly once and as the last vertex it visit.

The changed required to make DFBnB return optimal solutions for MAX problems are simple and very similar to the changes detailed above for A*: (1) an admissible heuristic must now be an upper bound on the cost to go, and (2) a state N can be pruned only if $g(N)+h(N)$ is smaller than or equals to the cost of the incumbent solution (Stern et al. 2014).

3.2 Challenges

Using either A* or DFBnB to solve LSP is not trivial, and raises several major challenges.

[2]The initial state, which in LSP will be an empty path with one vertex – s.

State space size The size of the LSP state space is much larger than the SP state space defined over the underlying graph $G = (V, E)$: the SP state space is linear in $|V|$ while the LSP state space can be exponential in $|V|$. For example, consider an empty $X \times Y$ grid, where s is the grid cell in bottom-left and g is the grid cell on the top-right. The SP state space has $X \times Y$ states while the number of possible graph-paths from s to g is exponential in X and Y.

No pruning of dominated paths By construction, the LSP state space is a tree. Thus, there is exactly one path to reach each state, and consequently the duplicate detection mechanism used by A* and other algorithms will never detect a duplicate state. To address this, in Section 4 we propose more aggressive path pruning mechanisms that is able to prune states in the LSP state space.

Heuristics for Maximization Problems Most prior work in developing admissible heuristics have been for MIN problems. Standard techniques such as pattern database (Culberson and Schaeffer 1998; Felner, Korf, and Hanan 2004; Edelkamp 2014; Haslum et al. 2007), delete relaxation (Hoffmann and Nebel 2001), true distance heuristics (Sturtevant et al. 2009; Goldenberg et al. 2010) are all designed for MIN problems. The field of heuristic search has largely neglected the theory and practice of developing heuristics for MAX problems, and thus developing effective admissible heuristic for LSP is a challenge. We meet this challenge in Section 5.

4 Pruning Dominated States

Let $f^*(N)$ denote the length of the longest path from s to g that is an extension of $N.\pi$, that is, the length of the longest path that starts with $N.\pi$ and ends at g. We say that a state N dominates a state N‘ iff $f^*(N) \geq f^*(N$‘$)$. Clearly, any search algorithm that aims to solve LSP can safely prune N‘. In this section, we introduce several methods to identify this dominance relation between states, and show how to use these methods in A* and in DFBnB.

4.1 Basic Symmetry Detection

For an LSP state N we introduce the following notation: (1) $N.head$ is the last vertex in $N.\pi$, and (2) $N.tail$ is all other vertices in $N.\pi$. We say that that two states are symmetric if their heads are in the same location and their tail covers exactly the same set of cells, not necessarily in the same order. Figure 1 shows an example of two symmetric paths.

Observation 1. For every pair of symmetric states N and N‘, it holds that N dominates N‘ and vice versa.

A direct corollary from Observation 1 is that if a search algorithm generates two symmetric states, it can safely prune one of them. We call to this dominance detection method the Basic Symmetry Detection (BSD) method.

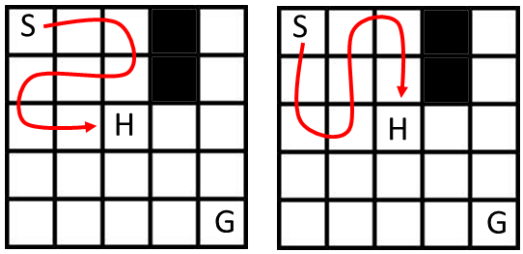


Figure 1: Example of two symmetric states

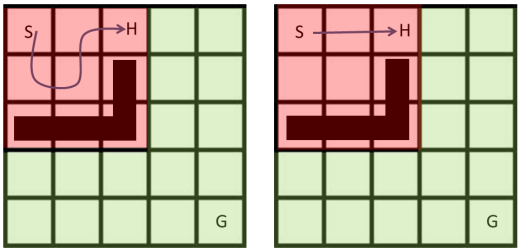


Figure 2: The two depicted LSP states N (left) and N‘ (right), where N dominates N‘ according to Theorem 1.

One way to implement BSD is to perform a linear match between the lists of the two tails. To speedup performance, we implemented BSD in a more advanced way: the vertices in the tail are stored in lexicographic order, and we used a hash function over these values for each tail. This will map all tails that occupy the same region to the same key, allowing quick identification of states that are not symmetrical. This method was significantly faster in our experiments and we only report its results below.

4.2 Reachable Dominance Detection

For a given LSP state N, we can partition the vertices in the underlying graph into three sets. The first one is the set of visited vertices, which is the union of the head and the tail of N. From the remaining vertices, we denote by reachable the set of vertices that may be visited by a path that is an extension of N. We call the remaining vertices the blocked vertices, as they represent vertices that may never be visited by a path that is an extension of N. For brevity, we denote by $N.V$, $N.R$, and $N.B$ the visited, reachable, and blocked sets of vertices.

Theorem 1. State N dominates state N‘ if the following conditions hold (1)$N.head = N‘.head$, (2) $|N.\pi| \geq |N‘.\pi|$, and (3) $N‘.R \subseteq N.R$.

Proof. Let π_N be the longest graph-path that starts from $N.head$, ends in g, and only includes the vertices in $N.R$, and let $N.\pi^*$ be the longest graph-path from s to g that has $N.\pi$ as a prefix. By definition, $N.\pi^* = N.\pi \circ \pi_N$, where $\circ$ denotes path concatenation. Thus, $f^*(N) = |N.\pi^*| = |N.\pi| + |\pi_N|$. Due to condition 2, $|N.\pi| \geq |N‘, \pi|$. Due to conditions 1 and 3, $|\pi_N| \geq |\pi_{N‘}|$. Thus, $f^*(N) \geq f^*(N‘)$, as required. □

Figure 2 shows an example of two LSP states, N (left) and N‘ (right), that satisfy the dominance condition given in Theorem 1. The red areas represents the blocked vertices and green areas represents the reachable vertices. As can be seen, both states have the same head, the same sets of reachable states ($N.R = N‘.R$), and the length of $N.\pi$ is 4 while the length of $N‘.\pi$ is 2. Thus, RDP will prune the state of the left.

Implementing RDP in an efficient manner is a challenge, as it requires fast set inclusion computation. In our implementation, we used a simple linear search over the reachable states to check if one set of reachable states is included in the other. This indeed causes RDP to be sometimes slower than BSD, as can be seen in our experimental results.

4.3 Pruning Dominated States During a Search

Algorithm 1: The A* Algorithm

```
1  OPEN ← initial
2  g(initial) ← 0
3  CLOSED ← ∅
4  while OPEN is not empty do
5  |  N ← pop best state from OPEN
6  |  Add N to CLOSED
7  |  if N is a goal state then
8  |  |  return N
9  |  end
10 |  foreach operator o applicable in N  do
11 |  |  C ← apply o on N
12 |  |  newg C ← g(N)+ cost(o)
13 |  |  if ∃N‘ ∈OPEN∪CLOSED s.t.
   |  |    Dominate(N‘, C)  then
14 |  |  |  Goto line 10
15 |  |  end
16 |  |  foreach N‘ ∈OPEN s.t. Dominate(C,N‘)
   |  |    do
17 |  |  |  Remove(N‘, OPEN)
18 |  |  end
19 |  |  g(C) ←newg
20 |  |  Insert(C, g(C) + h(C))
21 |  end
22 end
23 return no solution found
```

Given a method for detecting dominated states we now describe how to use it in the context of A* and in the context of DFBnB. We denote by Dominate(N,N‘) the application of the chosen dominance detection method. Algorithm 1 lists the pseudo-code for such an A* implementation. The highlighted lines are those that use the Dominate method. When a child state C is generated, we search for a state N in OPEN or CLOSED that dominates C (line 13 in Algorithm 1). If such a state is found, then C is not inserted into

OPEN. If there is no such dominating state in OPEN or CLOSED, we check if C dominates a state N that is already in OPEN (line 16). If this occurs, then N is removed from OPEN. Finally, C is added to OPEN (line 20).

To use BSD or RDP in DFBnB, we implemented DFBnB with a transposition table that contains all generated states. Applying BDS for DFBnB is easy: prune a newly generated state if there exists a symmetric state in the transposition table. The dominance relation detected by RDP is asymmetric, that is, one state dominates the other and not vice versa. Thus, pruning will only occur if DFBnB happened to generate the dominating state before the dominated states. This cuts down the effectiveness of RDP by half. Experimentally, this proved not cost-effective, and thus in our experimental results (Section 6) we only report results for DFBnB without any pruning and as well as with BSD only (but not with RDP).

5 Heuristics

In this section, we describe several admissible heuristic functions for solving LSP problems.

5.1 The Reachable Heuristic

For a state N, any graph-path that extends $N.\pi$ can never visit a vertex that is not in the reachable set of vertices. The Reachable heuristic relies on this observation, denoted h_R and defined as the size of the reachable set, i.e., $h_R(N) = |N.R|$. Clearly, h_R is admissible. The reachable heuristic will return 16 for both states in Figure 2, as there are 16 reachable vertices (the green area).

To compute the reachable heuristic for a state N, we run a simple depth-first search starting from $N.head$ and spanning all reachable vertices. The reachable heuristic was previously introduced by Stern et al. (2014).

5.2 The Bi-Connected Components Heuristic

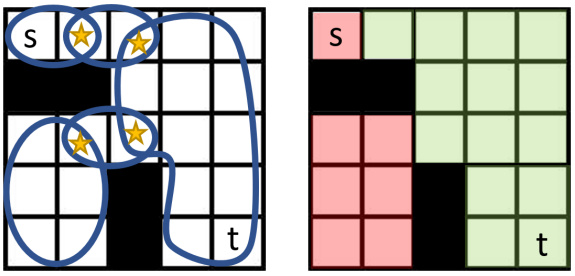


Figure 3: An example of using the BCC heuristic. Left: the bi-connected components. Right: the vertices on the BCT branch that reaches the goal, marked in green.

A bi-connected graph is a graph that remains connected after removing any single vertex. An equivalent definition is that every two vertices have at least two vertex-disjoint paths connecting them. A bi-connected component of a graph G is a maximal sub-graph of G that is bi-connected. A cut-point is a vertex that belongs to two different bi-connected components. Every graph can be decomposed into a set of biconnected components. The resulting set of cut-points and bi-connected components form a Block-Cutpoint-Tree (BCT) that can be constructed in linear time (Harary and Prins 1966; Hopcroft and Tarjan 1973).

Two bi-connected components cannot have more than a single cut-point connecting them. Thus, once a graph-path passes a cut-point, it can never return to that bi-connected component. Therefore, any simple graph-path passes through a single branch in the BCT. In particular, any solution to an LSP problem must pass through the BCT branch that starts in the bi-connected component that contains s and ends in the bi-connected component that contains g. The BCC heuristic returns the sum of the number of reachable vertices in each of bi-connected component along this BCT branch, starting from the bi-connected component that contains the head of the evaluated state. This heuristic is based on a similar heuristic proposed by Palmobo et al. (2015) for the snake-in-the-box problem.

Figure 3 shows an example of the BCC heuristic. Figure 3(left) shows the initial LSP state, which represents an empty graph-path where the head is still in s. The BCT for this graph is visualized: bi-connected components are marked with blue lines and cut points with yellow stars. Given this BCT, we can identify that the vertices on the bottom-left area are part of a bi-connected component that is not on the longest simple path from s to g. The vertices that are on the BCT branch that leads from s to g is marked in green in Figure 3(right). The BCC heuristic counts only these vertices, returning a value of 14. The reachable heuristic, will count all reach able states, returning 20 for this state.

5.3 Alternate Steps Heuristic

The following heuristic is applicable for cases where the underlying graph represents a 4-connected grid, i.e., every grid cell is a vertex and there is an edge between this vertex and the vertices that represented its 4-connected neighboring grid cells.

Now, consider a grid marked like a board of chess, i.e., with grid cells are divided into black and white. A path along the board must alternate between white and black cells, that is, if a vertex v in a path is on a white cell then the next vertex in that path must be a black cell, and vice versa. For an LSP state N, let Δ be the difference between the number of white cells and black cells in the set of reachable vertices ($N.R$). Assume the head of N is a white cell. If the goal is a black cell, we can subtract Δ from the number reachable grid cells. If the goal is white cell, then we subtract $\Delta - 1$ from the number of reachable grid cells. We denote by Reachable+alternate (R+ALT) the reachable heuristic enhanced with this method, i.e., $h_{R+ALT}(N) = h_R(N) - \Delta$ of the head of

Heuristic	A*			DFBnB	
	NP	BSD	RDP	NP	BSD
R	92.5	97.5	91.4	100	100
R+ALT	94.4	98.3	92.2	100	100
BCC	99.4	100.0	99.7	100	100
BCC+ALT	99.7	100.0	99.7	100	100
BCC+S+ALT	99.7	100.0	99.7	100	100

Table 1: Success rate, open grids with obstacles.

Heuristic	A*			DFBnB	
	NP	BSD	RDP	NP	BSD
R	45,211	21,815	16,494	49,772	24,823
R+ALT	34,271	15,708	14,051	38,100	18,286
BCC	8,366	2,703	2,271	9,623	3,447
BCC+ALT	7,491	2,187	2,077	8,651	2,869
BCC+S+ALT	7,348	2,097	2,025	8,499	2,771

Table 2: Average expanded states, open grids with obstacles.

N and the goal are of the same color and $h_{R+ALT}(N) = h_R(N) - \Delta + 1$ otherwise.

There are two ways to apply this enhancement on top of BCC. The first one is to apply it on the entire set of all vertices in all bi-connected components of the branch in the BCT. This is called BCC+ALT. The second method is to apply this enhancement separately for each bi-connected component and then add them up. This is called BCC+separate+alternate (BCC+S+ALT).

The BCC formulation can also be used to prune states. This is done by identifying in a preprocessing phase all the vertices that are not on the BCT branch leading to the goal vertex, and considering all other vertices as blocked vertices. We found experimentally that doing this preprocessing proved beneficial, saving an average of 25% of the number of expanded states, and saving approximately 30% of the CPU time. Thus, we implemented this in all our variants and the experimental results included using this method.

This alternating heuristic, whether on top of the BCC heuristic or the Reachable heuristic, is more general than just for 4-connected grids. In fact, it is applicable for any underyling graph that can be represented as a bi-partite graph: the vertices on one part of the graph will be the "black" vertices and the vertices on the other will be the "white" vertices.

6 Experimental Results

In this section, we compared experimentally A* and DFBnB using all the proposed pruning methods (no prunning, BSD, and RDP), and heuristics (R, R+ALT, BCC, BCC+ALT, and BCC+S+ALT). The underlying graphs used in our experiments are based on 4-connected grids. Specifically, we used open grids with random obstacles and room map grids.

Heuristic	A*			DFBnB	
	NP	BSD	RDP	NP	BSD
R	10,808	2,626	17,576	601	448
R+ALT	6,155	1,417	14,386	463	338
BCC	377	110	294	195	105
BCC+ALT	300	82	262	150	86
BCC+S+ALT	311	86	261	149	94

Table 3: Average runtime (ms), open grids with obstacles.

6.1 Open Grids with Random Obstacles

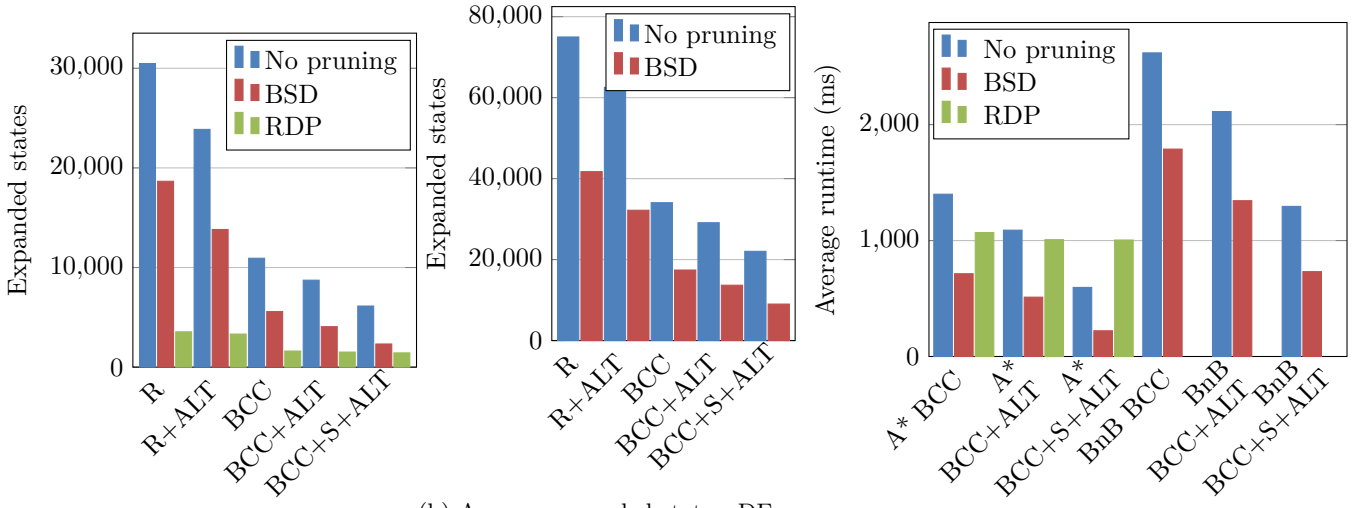
In this set of experiments we generated open grids with all integer combinations of sizes between 5x5 and 7x8 (total of 9 combinations). To each of these grids we set 4%, 8%, 12% or 16% cells as obstacles. For each of these percentiles we created 10 random LSP problems, yielding a total of 40 instance per gird size. In total, we had 360 random instances. A timeout of 10 minutes was set for solving each problem instance.

Table 1 shows the success rate, i.e., the percentage of instances (out of the 360 available problems) that were solved within the 10 minute limit. The rows are the different heuristics and the columns are the different search algorithms coupled with the different pruning techniques. We denoted by "NP" the columns that represent not using any pruning technique. This table shows that DFBnB and A*+BSD using any of the BCC heuristics are able to solve all the problem instances.

Table 2 compares the number of expanded states per solver averaged over all instances that could be solved by all methods. The main expected trends are very clear. First, the more advanced heuristics yielded fewer state expansions. The main advantage comes from using the BCC heuristics, as the differences between the BCC heuristics is quite small. Second, RDP is able to prune more states than BSD, and BSD prunes a significant number of states (when compared to NP). Note that the best method of A*+RDP with the BCC+S+ALT heuristic is better than the baseline A* variant by a factor of 22. Table 3 gives similar results but for runtime in milliseconds. Here, we see that the RDP was slower than the BSD method, although it expanded fewer states. This is reasonable since RDP incurs a large CPU overhead. Neverthelss, when considering the best method (A*+BSD), it is 12 times faster than the baseline A* variant (no prunning, reachable heuristic).

6.2 Room Maps

For this set of experiments, we generated grids shaped like a house with rooms. Every grid contains multiple rooms connected via narrow doors, the doors



(a) Average expanded states, A*.

(b) Average expanded states, DFBnB.

(c) Average runtime, A* and DFBnB.

Figure 4: Room maps result charts, all comparisons are for instances that was were solved by all algorithms, heuristics and pruning

	A*			DFBnB	
Heuristic	NP	BSD	RDP	NP	BSD
R	58.3	62.5	72.0	68.5	70.0
R+ALT	61.3	65.0	72.8	70.5	71.3
BCC	72.0	76.0	81.5	76.0	78.0
BCC+ALT	74.0	77.8	81.5	77.0	79.0
BCC+S+ALT	75.3	79.8	82.3	78.5	80.3

Table 4: Success rate, room maps.

are randomly positioned. The number of rooms in a grid varies between 3x2 and 5x6, where the room size was between 2x2 and 5x5. 400 such room maps were created for our experiments. Table 4 shows the success rate of the different LSP algorithms on this type of grids. In this domain, the A*+RDP method outperforms all other methods, but DFBnB+BSD provides comparable results. The observed trends here are that indeed stronger heuristic and stronger pruning method yielded higher success rate.

Similar trends are observed when considering the number of expanded states and runtime. Figures 4a and Figure 4b plots the average number of expanded states for the different pruning methods and heuristics for A* and for DFBnB, respectively. The impcat of using both pruning and heuristics is very clear: RDP provides more pruning compared to BDS, which in terms provides more pruning compared to not performing any pruning. A comparison of the run-times over instances solved by all appears in 4c. The results show a more complex picture, where the overhead of using RPD is not worthwhile in terms of runtime. BSD provides a good balance in this domain, of runtime overhead and strength of the chosen dominating systems.

To summarize, we observe the following. First, the BCC heuristic is always better than Reachable. Second, adding the alternating step on top of the BCC heuristic is helpful, but not by a large margin. Third, it is always worthwhile to perform the BSD prunning. The RDP pruning always saves state expansions, but its runtime is sometimes worst than that of BSD. Forth, when comparing DFBnB to A*, we observe that their performance is usually quite similar, when equipped with a good heuristic.

7 Conclusion and Future Work

In this paper we proposed several techniques to help solve the longest simple path problem. These techniques include methods for pruning states that are dominated by other states, and a range of admissible heuristics. We explain how to use these pruning methods and heuristics in two popular search algorithms: A* and DFBnB. We evaluted the impact of each of the newly proposed techniques experimentally on two types of grid maps. The results show that the proposed pruning methods and heuristics improves the ability to solve LSP problems in reasonable time.

This work opens several compelling directions for future work. First, we focused on finding an optimal solution to LSP. Future work can explore how to tradeoff optimality for runtime. The suggested pruning methods, and in particular RDP, can be time consuming. Future work may develop more efficient data structures that will allow fast searching of dominated states.

We consider a multi-level jury problem in which experts We consider a multi-level jury problem in which experts We consider a multi-level jury problem in which experts We consider a multi-level jury problem in which

experts We consider a multi-level jury problem in which
experts We consider a multi-level jury problem in which
experts

References

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

Culberson, J. C., and Schaeffer, J. 1998. Pattern databases. Computational Intelligence 14(3):318–334.

Edelkamp, S. 2014. Planning with pattern databases. In European Conference on Planning (ECP).

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

Goldenberg, M.; Felner, A.; Sturtevant, N.; and Schaeffer, J. 2010. Portal-based true-distance heuristics for path finding. In Symposium on Combinatorial Search (SoCS).

Harary, F., and Prins, G. 1966. The block-cutpoint-tree of a graph. Publ. Math. Debrecen 13(103-107):19.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. IEEE transactions on Systems Science and Cybernetics 4(2):100–107.

Haslum, P.; Botea, A.; Helmert, M.; Bonet, B.; Koenig, S.; et al. 2007. Domain-independent construction of pattern database heuristics for cost-optimal planning. In AAAI, volume 7, 1007–1012.

Hoffmann, J., and Nebel, B. 2001. The ff planning system: Fast plan generation through heuristic search. Journal of Artificial Intelligence Research 14:253–302.

Hopcroft, J., and Tarjan, R. 1973. Algorithm 447: efficient algorithms for graph manipulation. Communications of the ACM 16(6):372–378.

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.

Keshavarz-Kohjerdi, F.; Bagheri, A.; and Asgharian-Sardroud, A. 2012. A linear-time algorithm for the longest path problem in rectangular grid graphs. Discrete Applied Mathematics 160:210–217.

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 Symposium on Combinatorial Search (SoCS), 96–104.

Pham, Q., and Deville, Y. 2012. Solving the longest simple path problem with constraint-based techniques. In Integration of AI and OR Techniques in Contraint Programming for Combinatorial Optimzation Problems - 9th International Conference, CPAIOR 2012, Nantes, France, May 28 - June1, 2012. Proceedings, 292–306.

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.

Portugal, D.; Antunes, C. H.; and Rocha, R. P. 2010. A study of genetic algorithms for approximating the longest path in generic graphs. In Proceedings of the IEEE International Conference on Systems, Man and Cybernetics, Istanbul, Turkey, 10-13 October 2010, 2539–2544.

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. R.; Felner, A.; Barrer, M.; Schaeffer, J.; and Burch, N. 2009. Memory-based heuristics for explicit state spaces. In International Joint Conference on Artificial Intelligence (IJCAI).

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.