Journal of Artificial Intelligence Research 75 (2022) 747-793 Submitted 02/22; published 11/22

Solving the Watchman Route Problem with Heuristic Search

Shawn Skyler SHAWN@POST.BGU.AC.IL
Dor Atzmon DORAT@POST.BGU.AC.IL
Tamir Yaffe TAMIRY@POST.BGU.AC.IL
Ariel Felner FELNER@BGU.AC.IL
Ben-Gurion University of the Negev,
Be'er-Sheva Israel

Abstract

This paper solves the *Watchman Route Problem* (WRP) on a general discrete graph with Heuristic Search. Given a graph, a *line-of-sight* (LOS) function, and a start vertex, the task is to (offline) find a (shortest) path through the graph such that all vertices in the graph will be visually seen by at least one vertex on the path. WRP is reminiscent but different from graph covering and mapping problems, which are done online on an unknown graph. We formalize WRP as a heuristic search problem and solve it optimally with an A*-based algorithm. We develop a series of admissible heuristics with increasing difficulty and accuracy. Our heuristics abstract the underlying graph into a *disjoint line-of-sight graph* (G_{DLS}) which is based on disjoint clusters of vertices such that vertices within the same cluster have LOS to the same specific vertex. We use solutions for the *Minimum Spanning Tree* (MST) and the *Traveling Salesman Problem* (TSP) of G_{DLS} as admissible heuristics for WRP. We theoretically and empirically investigate these heuristics. Then, we show how the optimal methods can be modified (by intelligently pruning away large sub-trees) to obtain various suboptimal solvers with and without bound guarantees. These suboptimal solvers are much faster and expand fewer nodes than the optimal solver with only minor reduction in the quality of the solution.

1. Introduction and Overview

Imagine you are in a museum and you want to see all the exhibits on the floor. To do so, you want to tour the museum so that you can see every item in all the rooms. Similarly, the museum's security wants to have a known path so that during its traversal, it will be able to see every item in the exhibit to check that it was not damaged. As another example, assume you want to water your garden, and you wish to find a path that your funnel will be able to pour water to all of your plants. This problem is called the *Watchman Route Problem* (WRP), where the task is to find a route that 'sees' every point in the environment. WRP was proven to be NP-hard for polygons (Chin & Ntafos, 1986).

In our variant of WRP, we are given a graph $G(\mathcal{V},\mathcal{E})$, a start vertex (similar to fixed or anchored WRP, as described in Section 2.4) and a *line-of-sight* (LOS) function. The task is to (offline) find a path from the start vertex through the graph, such that all vertices in the graph were visually covered by *line-of-sight* (LOS) from at least one of the vertices on the path. Such a *watchman path* is the output for our problem. In the optimal variant of WRP, we seek for the shortest path with these attributes.

The LOS function ($LOS : \mathcal{V} \rightarrow P(\mathcal{V})$) determines which vertices a given vertex can see (i.e., has line-of-sight to) and it can be any arbitrary function. Trivial LOS functions on

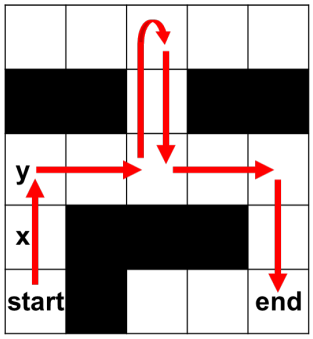


Figure 1: Optimal WRP Path

grids (four-connected graphs) are the cardinal lines (4-way) possibly combined with diagonal lines (8-way). An example of a non-trivial LOS function is a *transmission frequency function* that indicates for each vertex which are the vertices that can receive the transmission. Importantly, the exact map is known in advance and our task is to search offline for the requested (shortest) path. For example, Figure 1 shows the shortest possible tour such that each empty cell is seen by a vertical/horizontal LOS. An optimal solution for this problem is the shortest *Watchman Route* path. WRP has many practical applications, such as planning a patrol route, visually mapping a floor room in a building, finding a path for watering the garden and more.

In this paper we formally define WRP and formulate it as a heuristic-search problem by defining a corresponding search tree. We also investigate a number of variants for the LOS function. We then propose several admissible heuristics for WRP of increasing difficulty and accuracy that can be used on top of A* (Hart, Nilsson, & Raphael, 1968). In particular, we abstract the map into a *disjoint line-of-sight graph* (G_{DLS}), which is built from disjoint components, such that each has its own line-of-sight region. We prove that both the *Minimum Spanning Tree* (MST) and the solution tour of the *Traveling Salesman Problem* (TSP) on G_{DLS} are admissible heuristics for WRP.

We then introduce WRP-A*, an A*-based algorithm that uses these heuristics on top of the defined search tree. We also introduce a novel node expansion mechanism that significantly reduces the search tree by directly jumping to the different LOS regions. We theoretically study all these methods and heuristics. We prove their admissibility, show how to efficiently implement them, and study their trade-offs. We then provide an experimental study where we applied all our variants on a large number of different grid maps. We show that our strongest variant can optimally solve grids with up to 1,500 cells in several hundreds of seconds.

We also aim to solve large problems in short computation time by sacrificing the optimality of the solution. We therefore provide different search algorithms and methods which modify WRP-A* to two search cases:

1. **Bounded Suboptimal Search.** Given a bound W, the task is to find a solution path π where $\text{cost}(\pi) \leq W \times C^*$, where C^* is the cost of the optimal solution. In this paper, we compare variants of WA* as well as of XDP and XUP (Chen & Sturtevant,

2019). Experimental results show that these variants are much faster than WRP-A* while their solution is much better than the bound that they guarantee.

2. **Suboptimal Search.** The task is to find a solution as fast as possible without any guarantee on the quality. We introduce three methods that intelligently prune away subtrees that are less likely to contain a solution. When combining all these methods together, a solution is found orders of magnitude faster than baseline WRP-A*. Although not guaranteed, the quality of the solution was close to optimal.

We experiment with all these variants on mazes and maps of different size taken from the *movingai* benchmark (Sturtevant, 2012). Our experimental results show that we solve grids with up to 3,100 cells in less than a second while achieving close-to-optimal solutions.

This paper is organized as follows. Background and related work are covered in Section 2. A formal definition of WRP and the WRP-A* algorithm are given in Section 3. Next, we provide our admissible heuristics (Sections 4–5). An enhancement to the WRP-A* is given in Section 6 while Section 7 deals with the case of specific goal states. Section 8 provides a detailed experimental study. Then, Sections 9–10 introduce bounded suboptimal solvers and unbounded suboptimal solvers. Conclusions are given in Section 11.

All our definitions, algorithms and theoretical studies refer to general graphs. But, we always provide examples on grids. Similarly, in our experiments we focus on grids.

2. Background and Related Work

Next, we provide background on the Watchman Route Problem (WRP), on similar problems, and on algorithms that solve them.

2.1 Simultaneous Localization and Mapping (SLAM)

In the field of robotics, the *Simultaneous Localization And Mapping* problem (SLAM) is a prominent challenge that includes many variants and many solving approaches. The basic variant includes an autonomous moving agent that tries to explore the environment and build a map while simultaneously locating itself in the map (Dissanayake, Newman, Clark, Durrant-Whyte, & Csorba, 2001). A number of surveys on SLAM algorithms appeared in the literature (Aulinas, Petillot, Salvi, & Lladó, 2008; Taketomi, Uchiyama, & Ikeda, 2017). The main differences between SLAM and WRP is that in SLAM the environment is unknown and the task is to online explore the environment and build the map by a moving agent. By contrast, WRP is an offline search problem on a known map. Also, in SLAM the outcome of sensing is unknown until it is performed, while in WRP the actions of the agent are deterministic.

2.2 Inspection Planning

Another related problem is *inspection planning* (Fu, Kuntz, Salzman, & Alterovitz, 2019). A robot is equipped with a sensor and a set of *Points Of Interest* (POIs) in the environment to be inspected (i.e., physically seen with the sensor). This is a *Bi-Objective* problem (Hernandez, Yeoh, Baier, Zhang, Suazo, & Koenig, 2020a) where the solution is a motion plan for the robot that aims to maximize the number of POI inspected and minimize the cost of

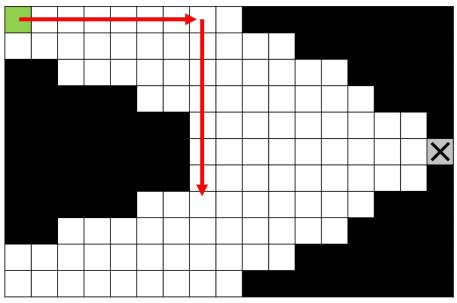


Figure 2: Differences between Art Gallery and Watchman

the plan. (see Fu et al. (2019) for more details). WRP can be seen as a special case where *all* POIs must be seen and that every vertex $v \in \mathcal{V}$ is a POI (i.e., we are interested only in one of the extreme parto-optimal solutions). In addition, in this paper we focus on discrete graphs whereas inspection planning works on a continuous high-dimensional configuration space (for the states of the robot) which is built on top of the physical environment.

2.3 Art Gallery Problem

A reminiscent classic problem is the *Art Gallery Problem* (AGP) which was proven to be NP-hard (Garey & Johnson, 1979). In AGP, we are given a map (e.g., an art gallery) and the task is to find the minimal set of points S on the map (to place guards) such that all points in the map can be seen by at least one point $s \in S$. A comparison between the two problems could be seen as the task of watching over an art gallery either by placing a minimal number of stationary guards/cameras to guard the gallery (AGP) or considering one mobile guard instead (WRP).

There are many algorithms that solve AGP on polygons. Such algorithms use geometric techniques such as triangulating a polygon. Then, some algorithms add internal diagonals between vertices until there are no more such diagonals that can be added (Fisk, 1978). After doing so, the problem can be transformed to the 3-coloring problem, which is NP-Complete (Appel & Haken, 1977), giving a solution to AGP (O'Rourke, 1987). An important algorithm for solving AGP on polygons (Honsberger, 1979) finds a solution with $\lfloor n/3 \rfloor$ points where n is the number of the polygon's vertices.

Indeed, the shortest possible tour between the points in S is a solution to WRP, but it may not be optimal (Kang, Kim, & Zhou, 2015). This is shown in Figure 2. The gray point, labeled X, is optimal with regards to the number of guards needed to cover the entire map (its LOS is the entire map). Yet, the optimal WRP path (the Red path) is much shorter than the path that travels to X.

2.4 WRP on Polygonal Domains

WRP has been extensively researched on polygonal domains in the field of computational geometry. The objective is to find a cyclic path that sees all internal points of a closed polygon. Here two points have line-of-sight if the straight line between them does not cross any edge. A specific variant of WRP is to define a start point that the guard must travel from. This is called *fixed* WRP or *anchored* WRP (Chin & Ntafos, 1991), as opposed

to *floating* WRP where no such point is required (Li & Klette, 2008). In general, the problem was proven to be NP-hard but a polynomial-time approximation algorithm exists with increased solution cost over the optimal solution of $O(log^2 n)$, where n is the number of vertices of the polygon (Mitchell, 2013). Such algorithms are used for complex polygons and polygons with internal holes.

For simple polygons (with no internal holes) there are polynomial-time algorithms (Chin & Ntafos, 1986) that identify a set of lines inside the polygon that the optimal watchman route must include. A full algorithm for optimal WRP on simple and complex polygons exists (Chin & Ntafos, 1991) which uses the same idea used for solving AGP (O'Rourke, 1987). An enhancement and corrections to that algorithm was introduced (Dror, Efrat, Lubiw, & Mitchell, 2003) with a proved running time of $O(n^3 logn)$. These algorithms use the following high-level framework. First, we place a horizontal line inside the polygon which cuts the polygon into two polygons. If every point on this line has line-of-sight to the entire smaller neighboring polygon then the smaller polygon can be deleted, and we recursively repeat this process for the larger polygon. Next, we repeat this process with vertical lines and this is called the "folding" process. Then, the remaining polygon is triangulated and now the triangles can be moved in such a manner that each triangle is connected to only two triangles (this is called the "roll-out" process). A shortest path that travels through all these triangles in their new formation can be found in polynomial time and it can be transformed back to the original (non-triangulated) polygon giving a (not necessarily optimal) solution for WRP on a simple or complex polygon.

2.5 The Uniqueness of Our Work

None of the other works on solving WRP just described is directly relevant to our problem because of the following differences. They assume a specific setting that includes a polygonal, continuous environment, with a specific line-of-sight such that two points can see each other if no edge of the polygon cuts the straight line between them. In addition, they assumed that the desired route is a *cycle*, i.e., that it ends at the start point. By contrast, we assume a general graph and may accommodate any LOS function, including asymmetric LOS functions. Furthermore, we do not require to end the path by returning to the start vertex but it can end at any vertex once all vertices have been seen.

2.6 The A* Algorithm and its Suboptimal Variants

A common framework for problem-solving in artificial intelligence is *heuristic search* for finding a minimum-cost solution path in a graph or a tree. The A* algorithm (Hart et al., 1968) is a general best-first search algorithm that is guided by the cost function $f(n) = g(n) + h(n)$ where $g(n)$ is the cost of the path from the start node to node n, and $h(n)$ is an estimation of the remaining cost to the goal. If h is *admissible* (i.e., it never overestimates) then A* is guaranteed to find an optimal solution. Also, under certain circumstances, A* was proved to be optimally efficient (Dechter & Pearl, 1985). A* and its variants are commonly used to solve search problems.

For large and complex problems, finding an optimal solution can take a lot of time and suboptimal solutions can be found more quickly and therefore are considered in the case of dealing with hard or complex problems. Naturally, suboptimal solvers trade the

optimality of the solution for faster running time. Suboptimal variants of A* exist. Some of these variants are *bounded suboptimal algorithms* (such as weighted A* (WA*) (Pohl, 1973)) which guarantee a certain bound on the quality of their returned solution compared to the quality of the optimal solution. Others provide no such guarantees.

2.7 Traveling Salesman Problem

Given a complete graph (clique) $G(V, E)$, the task in the Traveling Salesman Problem (TSP) is to find a path that travels through all the vertices of V and returns to the start vertex of this path. There are two variants of TSP. One variant allows multiple visits to the same vertex and one variant that does not allow multiple visits. A reminiscent problem is that of *Hamiltonian cycle* (Garey & Johnson, 1979) where the question to answer is whether a cycle exists that passes thorough all vertices exactly once. Finding the optimal (shortest path) solution for both versions of TSP is proven to be NP-hard (Papadimitriou, 1977). It is a common practice to transfer any graph G' to a complete graph G while adding missing edges with the cost of the shortest-path between the two vertices in case repetitions are allowed, and a cost of ∞ if repetitions are not allowed. Then, when solving TSP, one should find a permutation on the vertices of G. Because of the complexity of this problem, many suboptimal solvers were presented but below in various steps of our algorithm, we only use algorithms that solve TSP optimally. In our paper, when we talk about TSP, we refer to TSP where multiple visits are allowed.

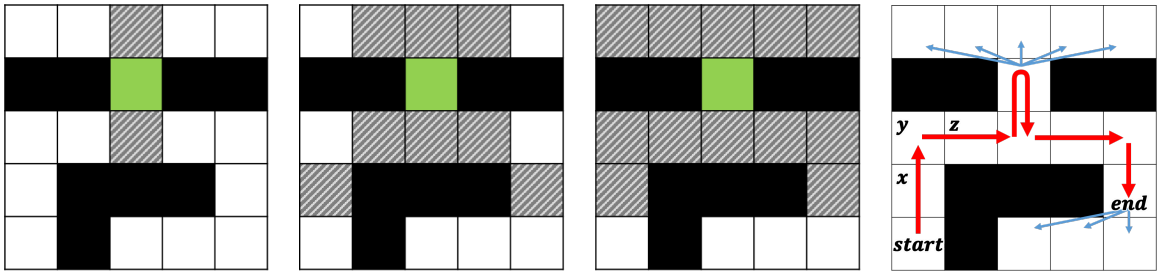
3. Problem Definition of WRP

We now turn to define the variant of WRP we focus on in this paper. The input is the tuple $\langle G, start, LOS\rangle$, where $G(\mathcal{V},\mathcal{E})$ is a graph (e.g., a map or a grid). For simplicity we assume that G is connected.[1] $start \in \mathcal{V}$ is the start location of the agent, and LOS is the line-of-sight function ($LOS : \mathcal{V} \rightarrow P(\mathcal{V})$). For grid-map instances, blocked (un-traversable) cells are denoted as *obstacles*, and the empty (traversable) cells are the vertices of the graph and correspond to $\mathcal{V}$. We say that cells p and q are *neighbors* iff there is a legal move from p to q (and visa versa). While our algorithms are general and apply to any graph, in this paper, for clarity, we provide examples for WRP and for the algorithms on grid maps, and assume that only the four cardinal moves are legal actions to move along the grid. However, our work applies to any general graph and LOS function. For example, in the context of grids, generalizing our work to allow diagonal moves or other moves (such as the 2^k neighborhood moves (Rivera, Hernández, & Baier, 2017)) should be easy.

A path $\pi = \langle s_0 = start, \ldots, s_k\rangle$ is a sequence of neighboring vertices starting from *start*. The cost of the path is the sum of edge costs of that path. The task is to find a *watchman path* in the map. In a *watchman path* π, for every vertex $v \in \mathcal{V}$ there is line-of-sight from at least one vertex $s_i \in \pi$. An optimal watchman path is a watchman path with minimal cost.

In the main part of the paper, we assume that the watchman does not have to return to the start vertex. The important aspect of the problem is that *all* vertices were seen. The reason is that, practically, we do not want to restrict the whereabouts of the watchman

1. If the graph is not connected, then depending on the LOS function, there might still be a valid solution and in fact, our algorithms below are valid for such settings too.



(a) 4-way LOS (b) 8-way LOS (c) BresLOS (d) Optimal Path Example using *BresLos*

Figure 3: Example of WRP with each of the LOS functions.

after the task is completed. It might stay idle, it might leave through the nearest exit or might destroy itself. Nevertheless, our algorithms below can be adjusted to solve the cyclic case as well as the case where a specific goal or set of goals are given and we discuss such settings in Section 7.

3.1 Line of Sight

The *line-of-sight* (LOS) relation may be defined in many ways; it may or may not be symmetric. For every vertex v, we use $LOS_{From}(v)$ to denote the set of all the vertices that can be seen from v. In this case, we write that $q \in LOS_{From}(p)$ and say that an agent located at vertex p can *see* vertex q or equivalently, we say that p has line-of-sight to q or that q is in the line-of-sight of p. For asymmetric LOS functions, we use $LOS_{To}(v)$ to denote the set of all the vertices that can see v or equivalently, all vertices that have line-of-sight to v. We refer to each vertex in $LOS_{To}(v)$ as a *watcher* of v. A LOS is symmetric if $p \in LOS_{From}(q) \iff q \in LOS_{From}(p)$, or equivalently, v sees all the vertices that see v (i.e., $LOS_{From}(v) = LOS_{To}(v)$). All our algorithms and definitions are general and applicable for asymmetric LOS functions. Nevertheless, in this paper, we focus on the following three symmetric LOS functions for grids:

1. **4-way LOS (LOS4):** LOS4 exists between two cells p and q iff there is a path π from p to q such that all moves in π are the same cardinal move (e.g., they are all East). The Gray cells in Figure 3a represent LOS4 for the Green cell.

2. **8-way LOS (LOS8):** LOS8 exists between two cells p and q iff there is a path π from p to q such that all moves in π are the same cardinal or diagonal move (e.g., they are all North East). Figure 3b shows LOS8 for the Green cell.

3. **Bresenham LOS (*BresLos*):** *BresLos* is a LOS function commonly used in computer graphics, video games and bitmap pictures. It allows to see more cells beyond the cardinal and diagonal lines. *BresLos* is perhaps the most suitable LOS function that discretizes real-world continuous domains into grids and simulates a continuous

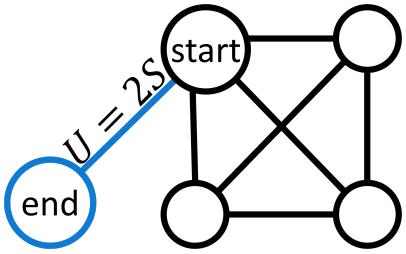


Figure 4: The graph G'. The new vertex is colored Blue.

field of view.[2] Computing *BresLos* involves many low-level details. But, intuitively, it approximates a straight line between two cells that does not pass through any obstacles. The exact definition of *BresLos* is complex and is given in (Bresenham, 1965). Figure 3c shows *BresLos* for the Green cell.

Sometimes, there is an upper-bound LOS radius R such that vertices at distance larger than R are not seen. Unless stated otherwise, for simplicity, we assume that $R = \infty$. But, all our algorithms below are applicable to any radius R as shown in Section 8.3 where we experiment with different R values.

3.2 WRP is NP-hard

We next prove that our version of WRP is NP-hard. We first define a *non-cyclic* variant of TSP (NC-TSP) and prove that it is NP-hard. Then, we reduce NC-TSP to WRP. Our proofs will be on the decision variants of these problems.

Definition 1. *In the decision variant of TSP, we are given a graph G and cost K, and the task is to decide whether there is a TSP path on G with cost $\leq K$.*

Definition 2. *In the decision variant of NC-TSP, we are given a graph G, a start vertex, and cost K. The task is to decide whether there is a path of cost $\leq K$ from start that passes through all vertices in G at least once. In the optimal variant of NC-TSP the task is to find the optimal such path.*

Lemma 1. *NC-TSP is NP-hard.*

Proof. We reduce TSP (assuming that vertices and edges can be visited multiple times) to NC-TSP. Assume a TSP instance with input graph $G(V, E)$ and cost K. We build a graph $G'(V', E')$. All vertices and edges in G also appear in G' but we add one more vertex denoted as *end*. We pick an arbitrary vertex *start* $\in V$ as the *start* vertex. Let S be the sum of all edge costs in E. We then connect *start* to *end* with an edge with cost $U = 2 \cdot S$. Figure 4 shows the new graph G'. We now ask whether there is a NC-TSP path on graph G' that starts in *start* with cost $\leq K + U$.

We next prove the reduction. That is, we prove that there is a TSP path of cost $\leq K$ on G iff there is a NC-TSP path from *start* of cost $\leq U + K$ on G'.

2. Originally, *BresLos* is asymmetric. But it can be converted to a symmetric LOS function by, e.g., taking the union of the two directions (LOS_{From} and LOS_{To}).

Direction 1: Assume that there is a TSP path on graph G with cost $X \leq K$. We can now construct a NC-TSP path on G' with cost $X + U \leq K + U$ as follows. The path will start at *start*, follow the TSP path on G which is a cyclic path with cost X. Then, it will go over the edge from *start* to *end* with cost U to a total cost of $X + U \leq K + U$.

Direction 2: Assume that there is a NC-TSP path on graph G' from *start* with cost $X \leq K+U$. We divide our discussion to two cases. Case 1: $K > U$. Observe that there is a trivial TSP tour as follows. Find an MST of G and assume that its cost is M. Then, cover the edges of the MST twice (from the root of the tree to the leaves and back). Now, observe that $M \leq S$. Thus, the cost of this TSP tour is $2M \leq 2S = U < K$. In fact, when $K > U$ then there is also always a NC-TSP path on G' with cost $\leq U + K$ by adding the edge $(start, end)$ to the TSP path on G that was just described. Case 2: $K \leq U$. In this case, the NC-TSP path cannot pass twice in the edge $(start, end)$ because it will incur a cost of C where $C > K + U$. So, the path only passes on this edge once with cost U (it ends at *end*). The rest of the path must be a TSP tour on G with cost $\leq X - U \leq U + K - U = K$ □

We next reduce NC-TSP to our version of WRP (where there is a specific start vertex and the path ends once all vertices in $\mathcal{V}$ have been seen).

Lemma 2. *WRP is NP-hard.*

Proof. We reduce NC-TSP to WRP. The reduction is easy. NC-TSP is a special case of WRP where LOS is defined such that $LOS_{From}(v) \equiv \{v\}$ for all vertices $v \in \mathcal{V}$. That is, a vertex can only see itself (LOS radius $R = 0$). Thus, one must pass through the entire set of vertices $\mathcal{V}$. □

We note that NC-TSP (i.e., WRP with LOS radius $R = 0$) is also known as *Coverage Planning* or *Shortest Hamiltonian Path* (Galceran & Carreras, 2013). In Appendix A, we provide a proof that WRP on a grid is also NP-hard.

3.3 WRP as a Search Problem

We next define the search tree of WRP that can be searched by any search algorithm. Then, we define admissible heuristics that can guide an A*-based search.

- **State:** A state is pair $\langle location, seen \rangle$ where *location* is a vertex (current location of the agent) and *seen* is a list of vertices (all the vertices that have been seen so far by the agent).

- **Node:** A node S in the search tree contains a *state*. The components of the state are referred to as $S.location$ and $S.seen$. The complement of *seen* is the *unseen* list; their union is the entire set of vertices ($seen \cup unseen = \mathcal{V}$). A node also has a parent pointer as well as a list of pointers to its children.

- **Root Node:** *Root* is a node such that $Root.location = start$ and $Root.seen = LOS_{From}(start)$.

- **Expansion:** Expanding node $S = \langle location, seen \rangle$ is to perform all legal movements on $S.location$. Each child S' of S corresponds to a given legal movement. $S'.location$ is the neighboring vertex of $S.location$ derived from the movement. $S'.seen$ is first inherited from the parent $S.seen$. Then, we add to $S'.seen$ all the vertices that are now seen from $S'.location$ and were not seen before. In other words, $S'.seen = S.seen \cup LOS_{From}(S'.location)$. The cost of the edge from S to S' is the cost of the movement action.

- **Goal Nodes:** Any node S where $S.seen = \mathcal{V}$. Thus $Goal.location$ may be any vertex in $\mathcal{V}$ as long as all vertices have been seen.

- **Path and Cost of Nodes:** Every node S in this search tree is associated with a path $\pi = \langle s_0 = start, ..., s_k = S.location \rangle$ which is determined by the branch of the search tree that leads to S. This enables the pruning of duplicate nodes that have the same state (same location and the same *seen* list) $S.seen$ includes all the vertices such that at least one of the locations in the path associated with S has line-of-sight to them. The g-cost of node S is the sum of the costs of applying the operators from *Root* to S, i.e., the cost of the path associated with that branch in the tree.

We use WRP-A* to denote the variant of A* that works on the search tree defined for WRP. *OPEN* denotes the open list of WRP-A*. Any A* search relies on an admissible heuristic to return optimal solutions. Below we introduce admissible heuristics for WRP-A*.

3.4 Preprocessing

All our heuristics and algorithms rely on the following three functions (defined on u and v which are two vertices from the input graph):

- $LOS_{From}(v)$**:** returns a set of all vertices that vertex v can see (vertices that have line-of-sight from v).

- $LOS_{To}(v)$**:** returns a set of all vertices that can see vertex v (vertices that have line-of-sight to v). Naturally, if LOS is symmetric then these two functions are identical.

- $d(u, v)$**:** returns the cost of the shortest path from u to v.

For efficiency, these functions can be implemented via three lookup tables that can be generated in a preprocessing phase and can be looked up during the A* phase. The table that stores $d(u, v)$ is also known as the *All Pairs Shortest Path* (APSP) table (Salvetti, Botea, Saetti, & Gerevini, 2017).

Let $n = |\mathcal{V}|$, the number of vertices. The size of each of these tables is $O(n^2)$. Building the APSP table may take up to $O(n^3)$ time. If p is the time to compute whether there is LOS between two vertices, then building the LOS tables may take up to $O(p \cdot n^2)$ time. These lookup tables can be fully built in a preprocessing phase or lazily during the search process. Nevertheless, these tables are polynomial in nature while the main problem is NP-hard. Therefore, the time and memory needed to build these tables is negligible compared to the resources needed to solve the main problem. In addition, the tables can be built once

and be used for solving multiple instances of the main problem. In our implementation, the APSP table was built lazily but the LOS tables were built in a preprocessing phase.

Next, we introduce several admissible heuristics to be used by WRP-A*.

4. Singleton Heuristic

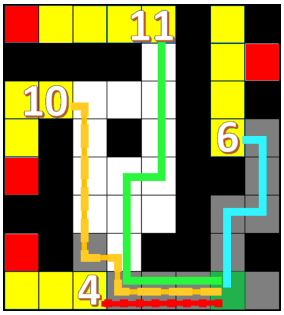


Figure 5: Example of Singleton heuristic

Our first heuristic is based on the idea that, in order to solve WRP, the watchman agent must see each of the vertices from $S.unseen$. Thus, for each vertex $p \in S.unseen$ (denoted as the *pivot* p) we define its *Singleton heuristic* to be the minimal distance from $S.location$ to a vertex $q \in LOS_{To}(p)$. Formally, given a pivot vertex $p \in S.unseen$:

$$h_p(S) = \min_{q \in LOS_{To}(p)} d(S.location, q) \tag{1}$$

where $d(x, y)$ is the cost of the shortest path between vertices x and y. For every $p \in S.unseen$, $h_p(S)$ is admissible because the agent will surely travel to some vertex that has LOS to p and $h_p(S)$ takes the minimum among all those vertices.

4.1 Aggregating Singleton Heuristics

Every possible pivot has its own singleton heuristic. Therefore, we can take the maximum of each of these heuristics in order to maintain admissibility (Holte, Felner, Newton, Meshulam, & Furcy, 2006). There exist a spectrum of possibilities to decide how many and which pivots to use. Naturally, adding more pivots has a diminishing return in terms of accuracy of the overall heuristic vs. time overhead. While we tried many combinations, in our experiments below we took the extreme case of using *all* cells in the *unseen* list as pivots. We calculated the singleton heuristic for all of them and took the maximum. We denote this heuristic by $h_{Singleton}$. It is formally defined by:

$$h_{Singleton}(S) = \max_{p \in S.unseen} h_p(S) \tag{2}$$

$h_{Singleton}$ is *larger than or equal to* any other combination of pivots. $h_{Singleton}$ is first fully calculated for the root node. Then, every vertex that is added to *seen* is removed from the set of pivots. This significantly reduces the computation of the heuristic as the search progresses.

In Figure 5 we show an example of the calculation of $h_{Singleton}$. Let S be the start location where *S.location* (the location of the agent at the start) is the Green cell, the Gray cells are in *S.seen* according to the *BresLos* LOS function, the Red cells are the chosen pivots (many permutations available) and each pivot has its watchers colored Yellow. Figure 5 shows that when aggregating the singleton heuristic to all these pivots we get that $h_{Singleton}(S) = 11$, represented by the Green line.

5. Graph Heuristics

Our next two heuristics are based on a graph called the *Disjoint LOS graph* ($G_{DLS}(V, E)$). $G_{DLS}(S)$ is abstracted from the input graph $G(\mathcal{V},\mathcal{E})$ for every node S in *OPEN*. For clarity, the following notations are used in this context. Since we always give examples with grids, then, when discussing G_{DLS} we use the term *cell* to denote a cell of the grid (= the vertex of the original input graph G) and the term *vertex* to denote a vertex of G_{DLS}. The term *node* refers to nodes of the search tree.

We say that two cells $x, y \in \mathcal{V}$ are *LOS-disjoint* if $LOS_{To}(x) \cap LOS_{To}(y) = \emptyset$ (i.e., there is no mutual cell that can see both vertices). Additionally, we say that a set of cells LD is *LOS-disjoint* if every two cells in LD are *LOS-disjoint* (i.e., $\bigcap_{v \in LD} LOS_{To}(v) = \emptyset$).

We next define $G_{DLS}(S)$ while demonstrating it on an example node S on a grid G with traversable cells $\mathcal{V}$. We then introduce admissible heuristics that are based on G_{DLS}. Figure 6a presents a grid G where the agent is located in the *start* cell X and LOS4 is used. The root node S is generated with $S.location = X$ and *S.seen* includes the cells in the grid which are horizontally right or left of X or vertically above X (all the Gray cells as well as A and D). $G_{DLS}(S)$, shown in Figure 6b[3], is built as follows.

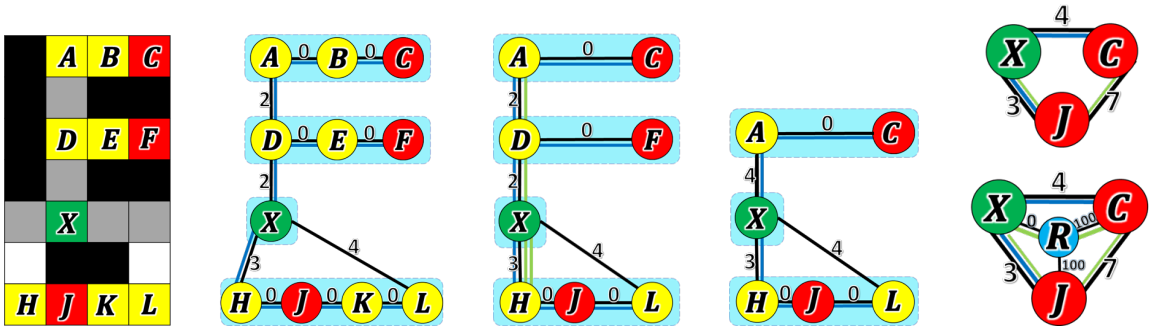
5.1 Vertices of G_{DLS}

The set of vertices of $G_{DLS}(S)$ is created from a subset of cells from G. These vertices are classified into the following three types (each with a different color in Figure 6):

- **Agent Vertex.** The *AgentVertex* (X – colored Green) is associated with *S.location*.
- **Pivots.** The *Pivots* vertices (C, F and J – colored Red) are a *LOS-disjoint* set of cells from *S.unseen*. Importantly, there are many possible ways to choose the set of *Pivots*, and thus $G_{DLS}(S)$ can be built in many ways. We describe our own method in Section 5.5.
- **Watchers.** The *Watchers* (A, B, D, E, H, K and L – colored Yellow) are all the cells that have *LOS* to one pivot from *Pivots*. Because the pivots are *LOS-disjoint*, each watcher has *LOS* to exactly one pivot. Formally, $Watchers = \bigcup_{p \in Pivots} LOS_{To}(p)$.

Note that some cells in $\mathcal{V}$ do not have vertices in $G_{DLS}(S)$ and are omitted. These cells are either the Gray cells which are in *S.seen*, or the White cells which are in *S.unseen*. A cell is White if it is not a watcher of any pivot (Yellow) and it cannot be made another

3. The colored edges will be explained later and can be ignored for now.



(a) The Grid G (b) G_{DLS} (c) $G_{DLS}1$ (d) $G_{DLS}2$ (e) $G_{DLS}3$ and $G_{DLS}4$

Figure 6: Example of a grid with LOS4 and the derived G_{DLS} graphs.

LOS-disjoint pivot (Red) in $G_{DLS}(S)$ because it has a watcher that is also a watcher of another pivot. For example, the White cell above L cannot be made a pivot because it will share the watcher L with J.

A pivot p and its watchers are denoted as *component* p of $G_{DLS}(S)$. Similarly, the *AgentVertex* (without its LOS) is its own component. There are four components marked with Blue background in Figures 6b and 6c: One for each of the pivots C, F and J and one for *AgentVertex*.

5.2 Edges of G_{DLS}

First, all edges $(u, v) \in G$ where both u and v exist in G_{DLS} are projected onto G_{DLS}, e.g., the edge (A, B) in Figure 6b. Next, we iterate over all cells that were omitted from G_{DLS} and *contract away* (Geisberger, Sanders, Schultes, & Delling, 2008) these vertices. A vertex v is contracted away by iterating over all pairs u, w that are both neighbours of v and adding an edge (u, w) that replaces vertex v. For example, the Gray cell above X is contracted away and X is connected to D in $G_{DLS}(S)$. Edges in $G_{DLS}(S)$ can be classified into two types. Their cost represents the distance that the agent needs to travel to see the pivots:

- **Watching edges.** These are edges inside a component that connect two watcher vertices or a watcher to its pivot in $G_{DLS}(S)$. Such edges have a cost of 0 (e.g., in Figure 6b, $c(A, B) = 0$).
- **Traveling edges.** These are inter-component edges. The cost of these edges is the APSP distance between the two vertices (e.g., $c(X, D) = 2$, $c(X, H) = 3$, $c(D, A) = 2$, as shown in Figure 6b).

5.3 Properties of G_{DLS}

Consider a valid solution path $\pi = \langle s_0 = start, ..., s_k = Goal.location \rangle$. Since $Goal.seen = \mathcal{V}$ then it includes all cells that are pivots in $G_{DLS}(S)$. Therefore, we get the following observation:

Observation 1: π will include at least one watcher cell for every pivot (the pivot is considered a watcher of itself).
G_{DLS} enables the following lemmas:

Lemma 3. *The minimum distance that the agent has to travel in G from S.location to see pivot cell p is the cost of the shortest path from S.location to p in $G_{DLS}(S)$.*

Proof. This is a direct result of the fact that each *watching edge* has a cost of 0 and a traveling edge has a cost of the APSP distance. □

Lemma 4. *A* **pivot-covering path** *is a path in $G_{DLS}(S)$ that starts at S.location and passes through all the pivots. The cost of the minimum pivot-covering path is a lower bound for the remaining cost in node S to complete a full solution to WRP.*

Proof. This is a direct result of Observation 1, Lemma 3 and the fact that the pivots are *LOS-disjoint*. It is only a lower bound because $G_{DLS}(S)$ may not cover all the cells in $\mathcal{V}$ as some cells are left out. Also, it assumes that traveling within a component has a cost of zero but this cost might be larger in practice, e.g., when the path travels through a component on its way to another component. □

The heuristics described below calculate different lower bounds to pivot-covering paths.

5.4 Simplifying G_{DLS}

Since we are interested in a *minimum pivot-covering path*, we can simplify G_{DLS} by contracting away many of the vertices of G_{DLS} while maintaining the fact that a minimum pivot-covering path is a lower bound on the remaining cost from node S. This is done via the following two procedures.

5.4.1 Contracting Away Internal Watchers

Consider cells A and B in Figure 6a. They are both watchers of pivot C. However, any path that arrives to the component of C (from outside that component) must first arrive at A. Therefore, B can be contracted away from G_{DLS} because it does not add any information. In general, we divide the watchers into two classes. *Frontier watchers* which are watchers that have neighbours in G outside of the component. *Internal watchers* which are (non-pivot) watchers that have all their neighbours within the component (either other watchers of the same pivot or the pivot itself). We thus simplify G_{DLS} by contracting away the *internal watchers*; only *frontier watchers* remain in G_{DLS}. The resulting graph is denoted $G_{DLS}1$ and is shown in Figure 6c, where cells B, E and K are internal watchers and are contracted away. Contracting away internal watchers does not change the pivot-covering path, it only reduces the number of vertices for some components in G_{DLS}. The cost of this path is also preserved because internal watchers are connected only with a 0 cost edges (watching edges) as described in Section 5.2.

5.4.2 Redundant Components

We define that component P is *redundant* with respect to component Q if all paths in $G_{DLS}(S)$ from *S.location* to Q pass through component P. We similarly say that the pivot of a *redundant component* P is a *redundant pivot*. In that case, component P can be deleted from $G_{DLS}(S)$ because that component must be passed when traveling to Q. For example, in Figure 6, the component of F is redundant with respect to the component of C. The redundant component is contracted away and neighbors of redundant components are then directly connected to cover the removal. The resulting graph is denoted $G_{DLS}2$ and is shown in Figure 6d where X is now connected to A with cost 4. Components that are not redundant are called *cardinal* and they remain in $G_{DLS}2$. Contracting away redundant components does not change the pivot-covering path, it only reduces the number of the components in G_{DLS} that will surely be on that path. However, by doing so, the cost of this path may increase because traveling inside a component that was not contracted away means traveling only through watching edges with cost 0. But, when this component is contracted away we use the real shortest distance that passes through that component.

5.5 Choosing Pivot Vertices

We now turn to discuss how pivots are chosen. We say that a set L of *LOS-disjoint* pivots is *maximal* if no other cell is *LOS-disjoint* to any of the cells of L (i.e., we cannot further increase L). Naturally, in order to produce high heuristic values we would like the size of the *Pivots* set to be as large as possible and thus obtain a *maximal LOS-disjoint* set.

We used the following greedy method to obtain a *maximal LOS-disjoint* set. We first set $Pivots \leftarrow \emptyset$. We then iterate over all cells $c \in S.unseen$. If c is *LOS-disjoint* to all existing pivots in *Pivots*, c is added as a new pivot. In our implementation, we iterated over all cells in increasing order of $|LOS_{To}(c)|$ so as to prefer pivots with small components (which allows a larger set of pivots). But, any other order of iterating over the cells will also produce a *maximal LOS-disjoint* set. We tried other orders (e.g., to find vertices that are farthest from the previous ones) and found this method to have the best performance.

We note that G_{DLS} is built from scratch for every node S in the search tree. The reason is that the *unseen list* is getting smaller on the fly. So, maximal sets of *LOS-disjoint* pivots can also dynamically change and even increase. For example, consider a cell d that was not *LOS-disjoint* to all pivots at a given node S. d might be *LOS-disjoint* to all pivots in its child C. This happens if there exists a cell e and a pivot p such that $e \in LOS_{To}(d)$, $e \in LOS_{To}(p)$ and $d \in C.unseen$. In the child C, if the agent has seen p from a cell that is not e, then p is in *C.seen* and d can now be pivot (assuming it is *LOS-disjoint* to all other pivots). This makes d *LOS-disjoint* to all pivots and d can be added to the set of pivots. The following steps summarize how $G_{DLS}(S)$ is built.[4]

1. Add *S.location* as *AgentVertex* to $G_{DLS}(S)$.

2. Choose a *maximal* set of *LOS-disjoint* pivots, identify their watchers and add all of them to $G_{DLS}(S)$.

4. A similar situation is illustrated in Figure 17 below.

3. Complete the edges by contracting away cells that are not in $G_{DLS}(S)$ and assign them their costs.

4. Build $G_{DLS}1$ by contracting away internal watchers.

5. Build $G_{DLS}2$ by contracting away redundant components.

Now that G_{DLS} is fully defined, we turn to introduce the heuristics that are based on it.

5.6 MST Heuristic

Given a graph $G(V, E)$, the *Minimum Spanning Tree* (MST) is a spanning tree of G so that the sum of its edge costs is minimal among all spanning trees. There are two classic greedy algorithms with time complexity of $O(|E| \times log|V|)$ (E and V refer to G_{DLS} in our case) which were given by Kruskal and by Prim (Greenberg, 1998; Cormen, Leiserson, Rivest, & Stein, 2009).[5] The MST heuristic $h_{MST}(S)$ computes a MST of $G_{DLS}(S)$. The sum of costs of edges of the MST is the minimum cost of a sub-graph that connects *AgentVertex* to all the *Pivots*, and is used for the heuristic value of h_{MST}. In Figures 6b-6e, the edges of the MST are marked by the Blue lines.

Lemma 5. *h_{MST} is admissible.*

Proof. Observe that all vertices within a component are connected with zero cost edges. Therefore, any MST also contains non-zero edges that connect the different components. The sum of costs of edges of the MST is the minimum cost of a sub-graph of G_{DLS} that connects *AgentVertex* to all the *Pivots*. Since the components are a disjoint set of vertices of G_{DLS}, this is a lower bound of a minimum pivot covering path. □

Since there exist quadratic-time algorithms for MST, h_{MST} is fast to compute.

5.7 TSP Heuristic

We now aim to find a *minimum pivot covering path* in G_{DLS}. Given G_{DLS}, we in fact want a path that starts at *AgentVertex* and passes through all the components. To do this, we further abstract $G_{DLS}2$ to $G_{DLS}3$ and then to $G_{DLS}4$. These graphs are *complete graphs* (cliques) and smaller than the previous ones so that we can apply TSP solvers on them as shown below. $G_{DLS}3$ is a *homomorphic abstraction* of $G_{DLS}2$. All vertices within a component in $G_{DLS}2$ are merged into a single vertex in $G_{DLS}3$ and internal edges within such components disappear. Edges between two components are merged into one edge. The cost of an edge between vertices in $G_{DLS}3$ is the minimal cost among all edges that connect these components in $G_{DLS}2$.

These new vertices are associated either with the corresponding pivot or with *AgentVertex*. $G_{DLS}3$ for our example is illustrated in Figure 6e(top). It has three vertices: X, C and J. As a result, $G_{DLS}3$ is always a complete graph (clique).

5. Faster, but much more sophisticated algorithms for MST also exist (Chazelle, 2000) but we used the traditional ones as G_{DLS} is relatively small.

We note that the MST of $G_{DLS}3$ includes exactly the non-zero edges of the MST of $G_{DLS}2$ (the zero cost edges disappeared because they belong to the same component). Furthermore, the MST of G_{DLS} and of $G_{DLS}3$ also use the same non-zero edges. Thus, one can compute h_{MST} on either of these versions of G_{DLS}. Nevertheless, we have found that it is most time-efficient to compute h_{MST} on $G_{DLS}2$. Indeed, to compute h_{MST} one can first generate $G_{DLS}3$ and then execute MST on $G_{DLS}3$. However, generating $G_{DLS}3$ requires to iterate over all edges of $G_{DLS}2$, but while doing so we can already compute MST so this duplication is not necessary.

We are interested in the minimum-cost path that starts at *AgentVertex* and visits all other vertices in $G_{DLS}3$. In fact, we are interested in the *minimum-cost Hamiltonian path* in a clique graph that starts at a specific vertex. This is different from the *Traveling Salesman Problem* (TSP) which calculates a minimal cycle in a clique which, by definition, does not have a specific start vertex. We therefore slightly modify $G_{DLS}3$ to yet a new graph $G_{DLS}4$ so that we can exploit TSP solvers. $G_{DLS}4$ adds a single *reference vertex* (denoted R) to $G_{DLS}3$. R is connected by edges to all other vertices in the graph. The cost of 0 is given to the edge connecting R to *AgentVertex* while all other edges have a cost of U. U is a constant larger than the sum of all edges in $G_{DLS}3$. $G_{DLS}4$ for our example is shown in Figure 6e(bottom).

Observe that a minimum traveling salesman tour (while visiting each vertex only once) for $G_{DLS}4$ must include the zero edge between R and *AgentVertex* (it is $\{R-X-J-C-R\}$ in our example). Furthermore, if we remove R and its two incident edges from the tour then we are left with a path whose one end is *AgentVertex* (this path is $\{X-J-C\}$). The cost of this path is an admissible heuristic for our problem, denoted by h_{TSP} (solve TSP on $G_{DLS}4$ and remove R).

Lemma 6. *h_{TSP} is admissible.*

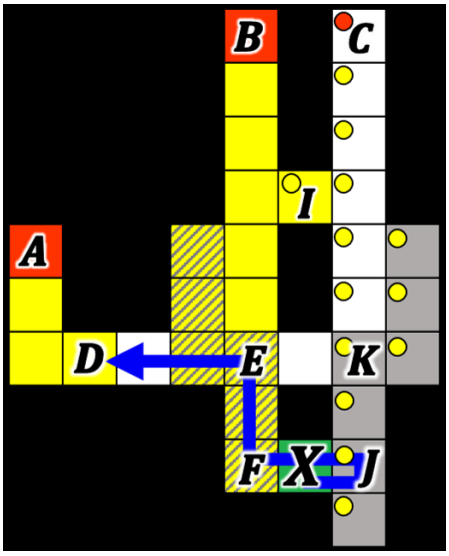
Proof. As built, a TSP tour on $G_{DLS}4$ without the edges connecting R is a lower bound on the shortest pivot covering path that travels from *AgentVertex* to see all the chosen pivots. □

Lemma 7. *$h_{TSP} \geq h_{MST}$.*

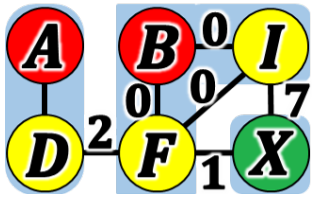
Proof. It is easy to see that given a graph G its MST is always *smaller than or equal to* a TSP tour on G. The reason is that an MST of G is the minimum-cost connected subgraph of G while TSP is the minimum-cost connected subgraph that is also a cycle. □

In our example (Figure 6e(top)), the MST is highlighted in Blue (4+3=7) while the path associated with h_{TSP} is highlighted in Green (3+7=10). However, unlike MST, TSP is NP-hard (Held & Karp, 1970; Garey & Johnson, 1979). Thus, there exists a natural trade-off of accuracy vs. time to compute the heuristic. We used an off-the-shelf TSP solver (Michail, Kinable, Naveh, & Sichi, 2020), based on an algorithm by Held and Karp (1970) that finds the optimal TSP tour. In our experiments below, we never had more than 12 pivots. However, if there are many more pivots, one can limit their number (and not use a maximal set of *LOS-disjoint* pivots). This will still be admissible and still competitive to the MST heuristic.[6]

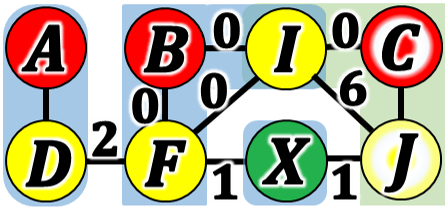
6. We experimented with such variants and confirmed this claim.



(a) The Grid G



(b) $G_{DLS}1(S)$



(c) $G_{DLS}J$

Figure 7: JF example

6. Jump to Frontier (JF): an Enhanced Branching Mechanism

Trivially, when node S is expanded, then new nodes are generated for all the cells that are neighbours of $S.location$. However, we can significantly reduce the size of the search tree by adopting the enhanced branching mechanism which we describe in this section. This mechanism is called the *Jump to Frontier* enhancement (JF). JF can optionally be added to WRP-A*. JF is a reminiscent of Jump Point Search (JPS) (Harabor, Uras, Stuckey, & Koenig, 2019) — a framework that implements A* on grids. In JPS, only *jump point* cells (e.g., those who are in a corner of an obstacle and therefore break the so called *Canonical Ordering*) are considered neighbors of the current node and are placed in *OPEN*. Other cells (e.g., those along a cardinal line) are just passed through.

We next describe two types of *frontier cells* for WRP-A* that are added as children of a given node S. From node S, the search branches to these children and they are added to *OPEN*. The first type of frontier cells corresponds to watchers of pivots of $G_{DLS}(S)$ and the second type corresponds to watchers of the White cells. Both types are described next.

6.1 Frontier Watchers of Neighboring Pivot Components

Given a node S and a corresponding $G_{DLS}1(S)$, consider any watchman path π that continues further from $S.location$. Such a path must include at least one watcher for each of the neighbouring components. We are interested in the first such watcher that appears on π. Recall that *Frontier watchers* are watchers that have neighbours in G outside their own component. We define the set of *Frontier Watchers* with respect to node S (denoted $FW(S)$) as watchers from components that do not include $S.location$ (= $AgentVertex$) such that the shortest path (or any of the shortest paths in case there are multiple shortest paths) from $S.location$ to them (taken from the APSP table) does not include any other frontier

watcher. An optimal watchman path π that will continue from $S.location$ will include one of $FW(S)$. To be complete we add all $FW(S)$ as children of S in the search tree.

We demonstrate $FW(S)$ using Figure 7 with the *BresLos* LOS function.[7] Assume that the current node is S, that $S.location$ is the Green cell X and that the Gray cells are in $S.seen$. When we build $G_{DLS}1(S)$, we find two *LOS-disjoint* pivots — cells A and B (colored Red). Each has a component of watchers colored Yellow. Cells that are both watchers of a pivot and in $S.seen$ are colored with Gray and Yellow diagonal stripes. Cell D is a frontier watcher for the component of A, and cells F, E, and I are frontier watchers for the component of B. Cells E and D are not in $FW(S)$ because the shortest path from X to them must pass through F. Thus, $FW(S)$ includ the frontier watchers F and I. They are frontier watchers of neighbouring components and they are direct neighbours of X in $G_{DLS}1(S)$ as depicted in Figure 7b. F and I are candidates to be children of S as shown Figure 8a. They are only considered candidates because cell I will be removed from being a child in the next step below due to the addition of White components.

Note that F and I are not enough to find an optimal solution for this instance. After jumping from X to F and seeing pivot B, cell C becomes a pivot, and to see C the agent will have to go to J, K, or I. In addition, jumping from X to I (with cost 7) is not optimal either. An optimal path (depicted in Figure 7a) is shorter as it first goes to J to see C and then goes to see the component of B via cell F, and then ends at D with a cost of 8. Thus, we need to consider cells besides those in $FW(S)$ that are neighbours of $S.location$ in $G_{DLS}1$.

6.2 Frontier Watchers of White Components

So far we have taken care of all components in $G_{DLS}1(S)$ that are direct neighbors of the component of *AgentVertex*, i.e., the Green cell and the Red and Yellow cells (of component B) in the map of Figure 7a. We are left with Gray and White cells. Gray cells are trivially covered because they are all in $S.seen$. To preserve optimality, we need to take care of cases where the optimal path first goes to see the White cells. Recall that White cells are in $S.unseen$. They are not colored Yellow because they do not see any of the pivots. They could not become new pivots (cannot be colored Red) because such a new pivot will not be *LOS-disjoint* with other pivots since they have a mutual watcher. For example, a White cell in the column right of I in Figure 7a (e.g., cell C) cannot be colored Red because it has LOS to I, which is a watcher of pivot B. Nevertheless, any watchman path must also see the White cells. Therefore, to guarantee the optimality of the solution returned by WRP-A*, we also need to consider the possibility that the optimal path first passes via watchers of a White cell. To do so, we build "auxiliary" components for the White cells and add them to G_{DLS} as follows. We iterate over all White cells. Given a White cell Z, we add Z as an auxiliary pivot and all the cells with LOS to Z as its watchers. We then add this auxiliary component to $G_{DLS}1$ similarly to real pivots. In this manner, we will continue until we have dealt with all White cells. The new graph that also includes the auxiliary components is denoted $G_{DLS}J$. Then, we add $FW(S)$ of $G_{DLS}J(S)$ as children of S.

7. We use a more complicated example than the example presented in the former section because we need this example to demonstrate the algorithm presented in Section 10.1.

(a) (b)

Figure 8: (a) The search tree of Red pivots built from $G_{DLS}1(S)$. (b) The final search tree with the White pivots built from $G_{DLS}J(S)$.

We demonstrate this step using Figure 7a. Cell C is a White cell that becomes an auxiliary pivot (marked with a Red circle), and all the cells marked with a Yellow circle are its watchers. The frontier watchers of C are cells J, K, and I. However, only J is in $FW(S)$ but not K and I because the shortest paths to them pass through J (or F). $G_{DLS}J$ after this stage is shown in Figure 7c. So, J is added as a child of S to a total of two children $\{F, J\}$. We note that after considering the new components of White pivots, cell I is no longer in $FW(S)$ because the shortest path to it must pass either trough F or through J. Thus, I is *surplus* and not added as a child of S. In fact, After $G_{DLS}J(S)$ is built we just take the neighbours of $S.location$ in $G_{DLS}J(S)$ as the children of S.[8]

Importantly, these auxiliary components (of White cells) are not considered by the graph heuristics (MST and TSP) because their pivots are not *LOS-disjoint* with the other pivots. They are only added later (after the heuristic computation) for the JF branching process.[9] Expanding the root node in our example of Figure 7 will result with the tree in Figure 8b.

6.3 Generating the Children for $FW(S)$

When expanding a node S, after building $G_{DLS}J$ we generate one child S' for each edge in $G_{DLS}J$ that connects *AgentVertex* to a member $W \in FW(S)$ (= neighbour of *AgentVertex* in $G_{DLS}J(S)$) as follows. $S'.location = W$. Let $\pi(AgentVertex, W)$ be a shortest path in $G(\mathcal{V},\mathcal{E})$ from *AgentVertex* to W (taken from the APSP lookup table or calculated on the fly). The cost of edge (S, S') in the search tree is set to the cost of $\pi(AgentVertex, W)$ (= the cost of the corresponding edge in $G_{DLS}J(S)$). $S'.seen$ is copied from $S.seen$ and is updated to also include $LOS_{From}(W)$. We note that W is the first vertex in $\pi(AgentVertex, W)$ that sees a new vertex. Thus, if there are multiple shortest paths to W it does matter which one the watchman will finally take. The following steps summarize the JF enhancement:

1. Build $G_{DLS}1(S)$.

8. Strictly speaking, there are two other White cells in the figure that are being treated in the same way. However, we omit these from the discussion for simplicity of the description.
9. In Section 10.1, we further develop this process of adding White cells to G_{DLS} and suggest a (suboptimal) variant that does not add White cells (Ignore Whites). As we demonstrate there, on our example (that was carefully handcrafted for this purpose), the Ignore Whites variant might lose optimality of the solution but might run faster.

2. Build $G_{DLS}J(S)$.

3. Add neighbours of $S.location$ in $G_{DLS}J(S)$ as children of S.

4. For each new child, generate the corresponding node.

6.4 Theoretical Analysis

We next prove a number of claims about WRP-A* with JF.

Lemma 8. *WRP-A* with JF is sound.*

Proof. WRP-A* (with and without JF) halts only when $seen = \mathcal{V}$ (*unseen* is empty). Therefore, forcing the algorithm to jump to a specific cell will not cause it to halt without seeing all the cells. □

Lemma 9. *Applying JF on top of WRP-A* is complete.*

Proof. We first begin with the following simple observation.

Observation 1. *When expanding a node S, every generated child S' has a* seen *list that contains S.*seen *plus all the vertices seen when jumping from $S.location$ to $S'.location$. Therefore, the size of* seen *is always increasing with each JF operation, i.e., S.*seen $\subset$ S'.seen.

At the root node, $Root.seen$ contains all locations that are seen from $Root.location$. For any node S selected for expansion, as long as $S.unseen$ is not empty (S is not a goal), at least one child will be generated because there is always somewhere to jump to. According to Observation 1, each of the generated children nodes will have more vertices in *seen*. As the search progresses (in any best-first order), for some node selected for expansion, since the number of vertices in $\mathcal{V}$ is finite, *seen* will include all vertices. i.e., a solution will be found and returned. □

Lemma 10. *Applying JF on top of the WRP-A* preserves optimality.*

Proof. Let π be an optimal watchman path. π contains a list of vertices, starting from *start*. Some of the vertices in π add new vertices to *seen* and others do not. JF builds a search tree such that new generated nodes always add new vertices to *seen*, and between each two nodes the shortest path is considered. When expanding a node, JF generates nodes according to all pivots, as well as all the auxiliary pivots. These are all options for moving and adding new vertices to *seen*. Therefore, the search tree built by JF also represents π, and executing WRP-A* is guaranteed to return it because it expands nodes according to a best-first order of their f-costs.

□

7. Specific Goal States

Up until now, we assumed that any vertex can be the goal as long as all vertices in $\mathcal{V}$ have been seen. This is an *implicit goal*. But, in many scenarios, the input might include an *explicit goal* vertex (exit door) or even an explicit set of possible goal vertices (several exit doors). The task here is to find a path from *start* to one of the goals such that all vertices in $\mathcal{V}$ have been seen. A special case of this is where the start vertex is also the goal vertex and the task is to find a cycle that sees all vertices in $\mathcal{V}$.

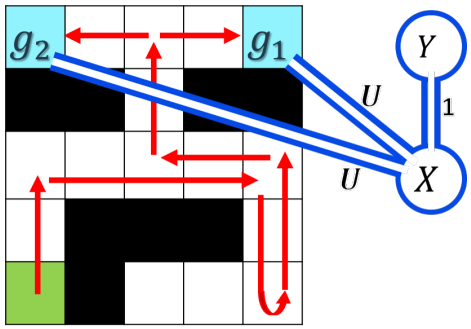


Figure 9: Multiple Goals Example

Given explicit goals, we next show how to modify the input graph $G(\mathcal{V},\mathcal{E})$ to a new graph $G'(\mathcal{V}',\mathcal{E}')$ such that a solution to the implicit goal case for G' is also a solution to the explicit goal case for G. G' is built as follows. For each explicit goal (one or more), we add an edge of cost U (where U is a very large constant) to a new vertex X. X is then connected to another vertex Y such that X is the only vertex that has LOS to Y (i.e., $\{X\} = LOS_{To}(Y)$). Figure 9 shows how our example graph from Figure 1 is modified where the top corners are the explicit goals.

Lemma 11. *An optimal path for the* implicit case *for G' must end at X via one of the U-cost edges.*

Proof. Any WRP path must pass through X because X is the only vertex that has LOS to Y. Now, since U is very large, every path that does not end in X must travel via such edge twice and incur the cost of U twice. Thus it is always cheaper to only incur the cost of U once. □

Deleting the last edge of cost U from the solution to the implicit case of G' results with the shortest path that ends at one of the explicit goals in G. Thus, all our algorithms are applicable to such cases after a small modification to the input graph.

8. Experimental Results for WRP-A*

We have performed extensive experiments with variants of WRP-A* (the optimal solver) that use the different heuristics with and without JF on many grid maps. We report our findings on representative example maps, but, it should be noted that similar trends were observed for other maps that are not reported. Indeed, each of our improvements achieved

a significant improvement in performance. The baseline breadth-first search algorithm (labelled BFS) which did not have any heuristic could only solve relatively small problem instances within reasonable computing resources while the best method (TSP+JF) could solve much larger problem instances. We next provide detailed experimental results.

8.1 Time Per Node Breakdown

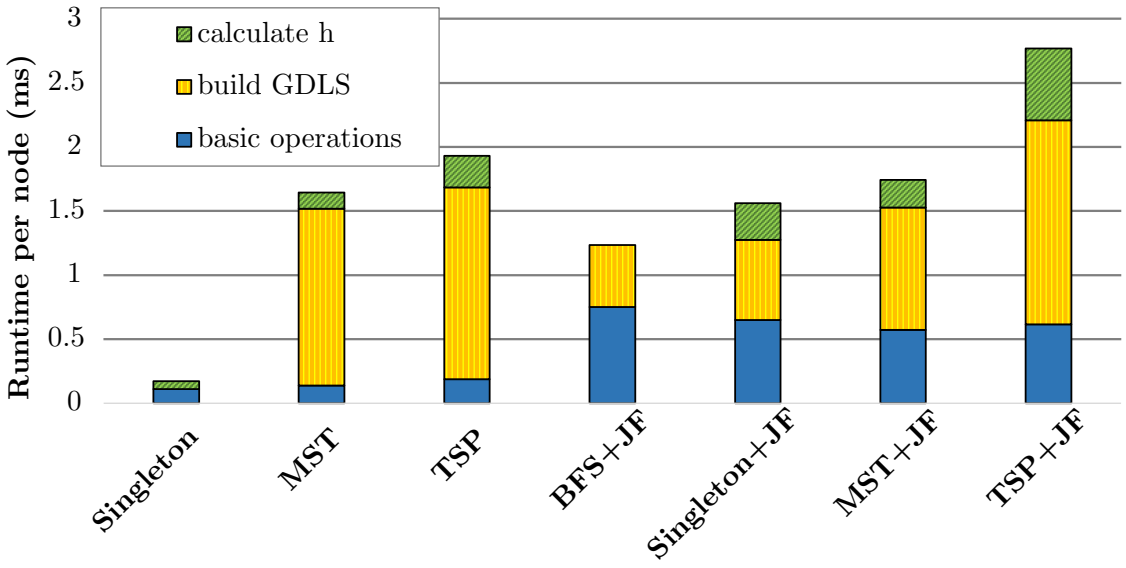
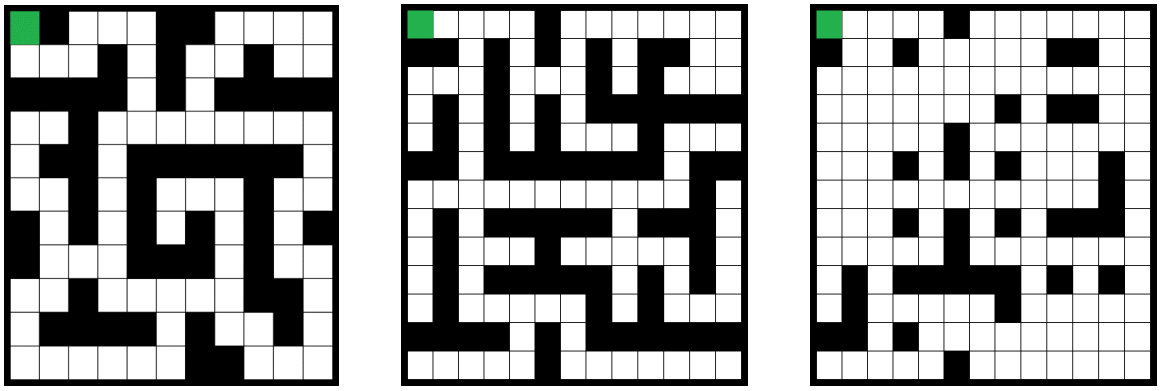


Figure 10: Runtime breakdown per node for the various variants

Figure 10 shows a breakdown to three categories of the average runtime per generated node on a large map while running WRP-A* using the various heuristics, without JF (basic expansion; left three) and with JF (right four). Only seven bars are shown because BFS without JF could not solve the problem and is not reported. The *basic operations* category includes all the Best-first Search overhead (e.g., operations on *OPEN*) as well as updating the *seen* and *unseen* lists. The *build* G_{DLS} category includes the overhead of building G_{DLS}. Finally, the *calculate h* category includes the overhead of calculating the heuristic.

The basic operations consumed more time for the JF methods compared to the basic expansion because JF jumped to farther locations, and updating the *seen* and *unseen* lists took more time. The basic operations for BFS+JF consumed more time because *OPEN* was larger. Note that for MST and TSP the *build* G_{DLS} category (which included simplifying G_{DLS} to $G_{DLS}2$) consumes more time than all other operations. In fact, solving MST and even TSP did not consume too much time because the resulting G_{DLS} was very small. We observed that the number of pivots varied from 9 in the root to 1 in the leaves and the average number of pivots in G_{DLS} per node was 5.8. Solving MST and even TSP on a few nodes can be done very fast and the *calculate h* category consumed on average less than 20% of the total time.

For the left part of the figure, JF was not applied and Singleton did not create G_{DLS} (it was not needed by the heuristic and not by JF). Therefore, Singleton consumed less runtime



(a) Maze 11×11 (b) Full 13×13 maze (c) 13x13 maze after removing 50% of the obstacles

Figure 11: Maze Maps

		Basic Expansion		JF expansion		Ratio: Basic/JF	
LOS	h	**Nodes**	**Time**	**Nodes**	**Time**	**Nodes**	**Time**
	BFS	666,370	27,670	28,485	7,499	23	3
LOS4	Singleton	296,264	19,166	13,980	3,772	47	7
OPT=73	MST	1,256	1,363	213	276	3,128	100
	TSP	**350**	**617**	**79**	**57**	**8,435**	**485**
	BFS	467,079	19,842	6,226	1,158	75	17
LOS8	Singleton	152,362	6,878	3,016	761	155	26
OPT=64	MST	1,122	509	119	203	3,925	98
	TSP	**507**	**412**	**75**	**59**	**6,228**	**336**
	BFS	149,450	6,345	2,316	503	64	13
BresLos	Singleton	61,959	2,639	1,175	246	127	26
OPT=57	MST	921	143	93	44	1,607	144
	TSP	**293**	**75**	**35**	**18**	**4,270**	**352**

Table 1: All algorithms with all LOS functions. 11x11 maze. We report the number of nodes and the CPU time in Msec.

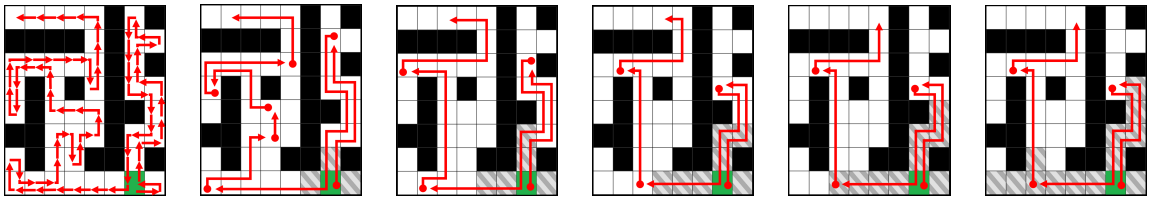
per node than the graph heuristics because the *build* G_{DLS} is the most time-consuming part per node. In Singleton, the total time is mostly influenced by the *basic operations* category and the *calculate* h category is negligible, and is a very thin slice in the graph.

8.2 11x11 Maze Grid

We next experiment on an 11x11 maze map shown in Figure 11a with a specific start state (the Green cell at the top left corner). We picked this relatively small domain so as to be able to provide a full comparison between all methods and all LOS functions. Results are presented in Table 1. The columns give the number of nodes expanded and the CPU time (in msec) to fully solve the problem for the *basic expansion* (left) and for the *jump to frontier expansion* (JF) (middle) for each of our LOS functions. Rows correspond to the different

heuristics. Consider LOS4 (top region). For basic expansion, the more informed heuristics significantly reduce the number of expansions over BFS by a factor of up to 1,900 (for TSP) and the CPU time by a factor of up to 44 (for TSP). The more liberal LOS functions further reduce the search effort. But, the relative advantage of using the heuristics increases. For LOS8 and *BresLos* the reduction of TSP over BFS was by a factor of ∼1000 for nodes and ∼100 for CPU time. The middle columns show the results when JF was used. Finally, the *Ratio* columns gives the improvement factor of the JF of each row compared to the basic expansion with BFS. For BFS, JF outperformed basic expansion by almost a factor of two orders of magnitude. This factor is naturally smaller when the heuristics were added. However, even when JF was applied, the heuristics further achieved a significant reduction over BFS. For *BresLos*, TSP+JF solved the entire problem in only 18ms. In the rest of the experiments we only focus on *BresLos* when JF was activated on top of WRP-A*.

8.3 Limiting the Line-Of-Sight Radius



(a) R=0 (b) R=1 (c) R=2 (d) R=3 (e) R=4 (f) R=∞

Figure 12: Limited radius R with an optimal path. *BresLos*.

Expansion Method	Radius	Generated	Expanded	Time(ms)	Cost
basic 4-way	0	$> 10^7$	$> 10^7$	$> 10^6$	61
	1	71,379	56,936	1,950	44
	2	9,829	7,634	333	35
	3	2,068	1,515	76	28
	4	1,237	921	123	27
	∞	457	339	107	27
JF	0	428	203	332	61
	1	37	23	19	44
	2	27	17	18	35
	3	16	10	14	28
	4	14	8	9	27
	∞	11	7	42	27

Table 2: Limiting the LOS Radius. The TSP heuristic was used with *BresLos*

A realistic way to modify an existing LOS is by limiting the sensing radius. Figure 12 illustrates a grid with 42 empty cells where the agent is located left to the bottom-right corner (colored Green). The frames are ordered in increasing LOS radius R. The LOS from

the cell of the agent is colored Grey. The Red arrows are the optimal WRP paths, and the Red dots inside these arrows indicate the jumping points of JF. Table 2 presents the data on the optimal paths on the map from Figure 12 where both expansion methods (Basic expansion 4-way movements and JF) were applied while varying the radius R. The TSP heuristic was used on top of *BresLos*. Naturally, increasing R results in a shorter optimal path because the agent can see farther cells. In fact, for $R = 4$ the path is identical to the path where $R = \infty$. Similarly, the search effort is reduced with larger values of R. Time increases when moving to $R = \infty$ because updating the *seen* and *unseen* lists incur more time. As mentioned above, when $R = 0$, the agent can only see the vertex it is currently located at, and the problem becomes equivalent to NC-TSP where the task is to travel through all the vertices of the graph.

8.4 Increasing Density of Obstacles

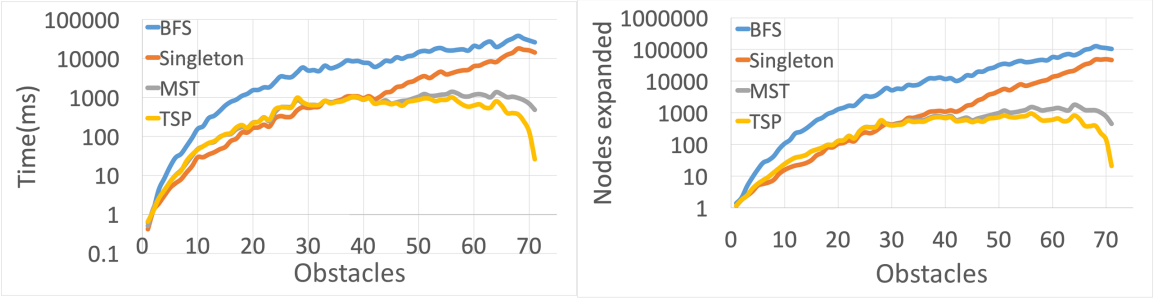


Figure 13: Varying the number of obstacles.

Figure 11b shows a 13x13 maze with 71 obstacles. From this we built 71 instances $(1 \ldots 71)$ where instance $\#n$ only had n obstacles randomized from the 71 obstacle set. Figure 11c shows the half way point where 35 obstacles were present. Figure 13 presents results on a log scale for all 71 instances averaged over 25 trials that randomly chose the subset of obstacles of the given size. When going from left to right in the figure more obstacles were added until the full maze was built. The right frame presents the number of nodes expanded for BFS and for our three heuristics, while the left frame is for CPU time. All curves are for the *BresLos* LOS when JF was activated. Here too, one can observe the significant superiority of the strong heuristics. TSP outperformed BFS by up to a factor of 100 in nodes and time. One can observe a phase transition; after 55 obstacles the problems become easier.

Observe that Singleton seems to be the best heuristic in relatively open maps (up to 40 obstacles). The reason is that MST and TSP might lose information due to the fact that edges inside a component cost 0 as we explain in Section 8.5. By contrast, the most costly Singleton pivot has exact distance to its watchers and in open maps this might be very close to the real cost.

After reaching the halfway point one can observe a trend where the curves of BFS and Singleton continue to increase while MST and TSP are more stable. The reason is that,

when more obstacles are added (this decreases the size of the map), the informativeness of MST and TSP remains stable because pivots are placed inside rooms and corridors. Singleton and BFS do not have this attribute and they continue to lose their effectiveness when more obstacles are added.

8.5 Advantages of the Singleton Heuristic

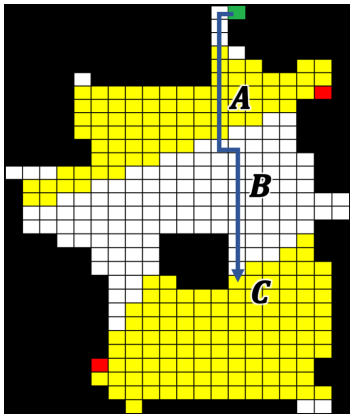


Figure 14: Orz106d from movingai.com

While the TSP heuristic is usually superior, there are cases where the Singleton heuristic should be preferred. We show an example for that next. Figure 14 shows the orz106d map from the *movingai* repository (Sturtevant, 2012) with two pivots colored Red and their watchers colored Yellow (components A and C). The Blue arrow shows an optimal path from the start cell (colored Green) which is at the top of the figure. When expanding the root, G_{DLS} is built with the two Red pivots shown in Figure 14. For the TSP heuristic, the path calculated on $G_{DLS}(root)$ has the cost of reaching the component of C minus the cost of traveling inside the component of A because the edges between the watchers and their pivots cost 0. Therefore, traveling within component A is not considered in the calculation of the TSP heuristic. The same applies for the edges of the MST in $G_{DLS}(root)$ (the MST will contain the 0 edges). By contrast, the Singleton heuristic takes only the maximum distance of the closest watcher of each cell of the map. In this example, the Singleton heuristic will take the cost of fully reaching component C. This causes the Singleton heuristic to return higher heuristic values than the other graph heuristics for some expanded nodes.

In general, the Singleton heuristic outperforms the graph heuristics if the solution path contains a very small number of jumps. This phenomenon usually happens when the map is a large open area. In such maps, each cell of the map has many watchers (i.e., large coverage of the map) and the cost of the pivot-covering path (see Lemma 4) will be small. In the example of Figure 14, one jump to a frontier of component C is a valid solution.

Table 3 shows data on this instance (of Figure 14). Indeed, Singleton outperforms the other heuristics in the node counts by a factor of up to 10 and, since it is faster to compute, its time improvement was by a factor of up to 30. This is an example where a simple fast heuristic outperforms a more complicated heuristic.

Movement	Heuristic	Expanded	Generated	Time
Basic Expansion	TSP	2,478	4,786	11,367
	MST	3,831	6,961	6,753
	Singleton	**331**	**801**	**345**
JF	TSP	328	1,086	2,887
	MST	542	1,277	1,915
	Singleton	**37**	**185**	**136**

Table 3: Results for Orz106d

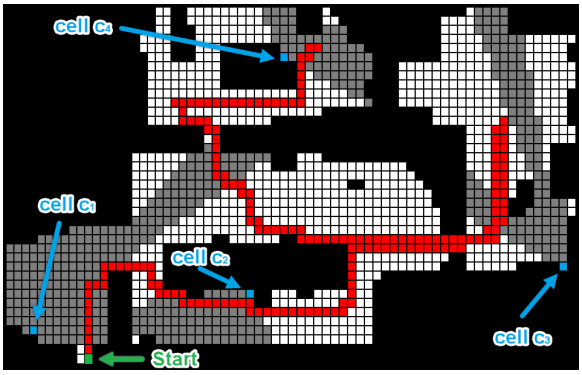


Figure 15: Den101d

8.6 Larger Map

We also experimented with the *den101d* map from the *movingai* repository (Sturtevant, 2012) shown in Figure 15. This map is of size 36x69 and has 1,360 empty cells. Our implementation was not optimized to save memory and each node consumed a few Kbytes. The start cell is marked in Green (bottom left) and the Red path is the optimal path found. The Gray cells are those who have *BresLos* LOS to the Blue cells. We executed our 4 heuristics on this map (with JF and *BresLos*). BFS and Singleton exhausted the available memory of 2GB and a failure was reported. Table 4 presents the maximal f-value that an algorithm reached, the number of generated nodes and the amount of time passed until the failure point. TSP and MST solved the problem optimally (OPT=137) but the other

Heuristic	Max$_f$	Expanded	Generated	Time(sec)
Optimal solvers				
BFS	66	>11,314	>50,000	>2,340
Singleton	103	>9,977	>45,000	>1,094
MST	**137**	13,271	41,837	1,577
TSP	**137**	**5,096**	**15,476**	**471**

Table 4: The den101d map and its experimental results

variants reported failure after a few minutes while only reaching f-values of 66, and 103 for BFS and Singleton, respectively.

We have also built suboptimal variants of WRP-A* which further reduced the branching factor and the size of the search tree. In the next sections, we discuss these variants for bounded suboptimal search and for unbounded suboptimal search.

9. Bounded Suboptimal Algorithm

It is common, especially in NP-hard problems, to trade the quality of the solution with running time. A common setting for this is that of *Bounded Suboptimal Search* (BSS). Given a bound W, the task is to find a solution with cost $\leq W \times C^*$. Many BSS algorithms exist but there is no universal winner and each algorithm has pros and cons. We studied a number of variants of *Weighted A** (WA*) (Pohl, 1970). WA* is simple and performs relatively robustly across domains (Gilon, Felner, & Stern, 2016). WA* is a best-first search algorithm that prioritizes nodes in *OPEN* according to $f_W(n) = g(n) + W \cdot h(n)$ where $W \geq 1$. The solution cost returned by *WA** is guaranteed to be $\leq W \times C^*$ (Pohl, 1973).

New members of the WA* family have been introduced recently (Chen & Sturtevant, 2019). When placing the g-value on the x-axis and the h-value on the y-axis then with WA* the set of nodes with the same f_W-value form a straight diagonal line. *Convex Downward Parabola* (XDP) modifies *WA** by changing the straight line to a parabola as follows:

$$f_{XDP} = \frac{1}{2W} \cdot (g + (2W-1)h + \sqrt{(g-h)^2 + 4Wgh}) \tag{3}$$

XDP focuses the search on nodes with near-optimal g-values (with respect to the optimal path) near the start and nodes with g-values are up to $(2W-1) \times C^*$ near the goal. Overall, the paths found are still bounded suboptimal (with W), and re-openings are not required. *Convex Upward Parabola* (XUP) is similar to XDP but focuses on nodes with near-optimal g-values near the goal and nodes with g-values are up to $(2W-1) \times C^*$ near the start. XUP is defined as:

$$f_{XUP} = \frac{1}{2W} \cdot (g + h + \sqrt{(g+h)^2 + 4W(W-1)h^2}) \tag{4}$$

We experimented with WA*, XDP and XUP on the Den405d map (Sturtevant, 2012) (Figure 16) with different values for W. In our implementation we did not allow to re-open closed nodes.[10] Table 5 provides the results (time in msec, number of expanded and generated nodes, and the solution cost) averaged over 30 random start cells. Chen and Sturtevant (2019) report that XDP tends to expand fewer nodes than WA* on the domains

10. The policy of not repeoning nodes was shown to produce solutions within the desired bound while usually expanding fewer nodes than the policy of always repoening (Sepetnitsky, Felner, & Stern, 2016). Additionally, we did not experiment with Explicit Estimation Search (Thayer & Ruml, 2011) and Dynamic Potential Search (Gilon et al., 2016) because they re-order their open lists every time the minimal f-value changes. Reordering is very costly especially in domains with many possible f-values such as WRP. This can probably be done more effectively if nodes that have the same g- and h-values are clustered together in g-h-buckets and if there is a small number of unique g- and h-values. This is valid mostly in permutation puzzles but not in WRP where valid paths are very long.

W	Alg.	Exp	Gen	Time	Cost
1	A*	840	3,662	20,608	**99.69**
1.1	WA*	203	**859**	6,285	99.76
	XUP	**201**	897	**5,486**	99.76
	XDP	206	906	6,134	99.76
2	WA*	**40**	**247**	**1,599**	101.14
	XUP	53	283	2,101	101.21
	XDP	91	498	3,113	99.83
5	WA*	**26**	**178**	813	106.59
	XUP	27	**178**	**668**	107.07
	XDP	34	229	1,366	103.07
10	WA*	**25**	173	**653**	107.07
	XUP	**25**	**172**	790	107.07
	XDP	38	252	1,597	104.79

Table 5: Results of WA*, XDP and XUP on Den405d map

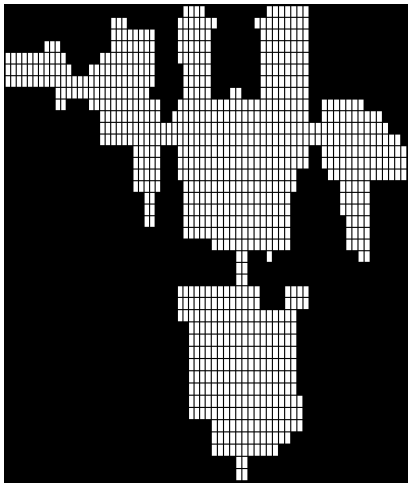
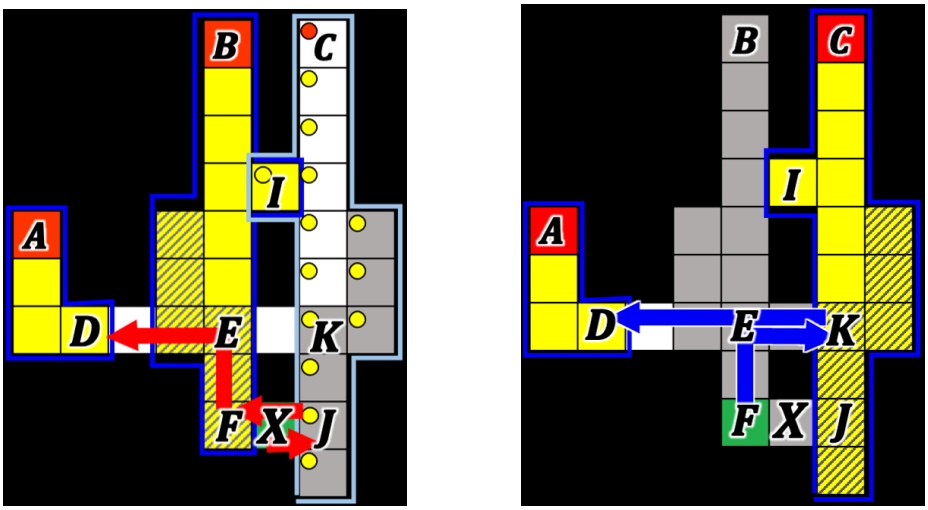


Figure 16: Den405d

they tested. By contrast, our results are mixed. XDP tends to provide better quality of solutions but tends to expand more nodes. Further study and comparison on the general behavior of these three variants is needed but is beyond the scope of this paper. But, one can see that all variants expand up to 34 times fewer nodes than WRP-A* while the solutions they return are never larger than $1.075 \times C^*$ (7.5%) which is much better than the bounds they guarantee which are up to $W = 10$.

10. Unbounded Suboptimal Algorithm

In BSS, there is a guarantee of W on the suboptimality. In this section we describe a suboptimal algorithm without a suboptimality bound. To do so, we modify WRP-A* by adding the following three techniques that reduce the size of the search tree (searched by



(a) Example with White Pivot C (b) Example After One Step

Figure 17: Suboptimality of Ignoring White cells

WRP-A*) by pruning subtrees that are less likely to contain the optimal path: **(1)** Ignoring White cells (those who do not have LOS to any of the pivots or the agent), thus reducing the branching factor of the tree. **(2)** A greedy method to remove pivots from G_{DLS} that are on the way to other pivots. This method is less strict than the removal of redundant pivots. **(3)** Bounding the jump points of JF to have distance no larger than by a constant factor more than the distance of the nearest neighbor. We cover these next.

10.1 Ignoring White Cells

Consider Figure 17 (which has the same grid as Figure 7a). Recall that *White cells* are unseen cells that do not have LOS to any of the pivots (Red cells) in G_{DLS} so they are not Yellow. They cannot be colored Red because they will not be *LOS-disjoint* to all other pivots as they have a common watcher with at least one existing pivot. As explained in Section 6, to ensure the optimality of the solution, given a node S we build (auxiliary) components for the White cells for the JF expansion (but not for the TSP heuristic computation). We use $G_{DLS}J$ to denote the resulting G_{DLS} after White cells were added (see Figure 7c). Neighbours of X in $G_{DLS}J$ are added as children of node S (see Figure 8).

We now suggest to ignore the White cells and not add White components, thus applying JF on the original $G_{DLS}1$ (and not $G_{DLS}J$). Cutting those components out will reduce the size of the search tree (no edges to frontier watchers of White cells). This will lose the guarantee of an optimal solution because, in rare cases, the optimal path passes through these White cells before any other cell from the other components as we now demonstrate using Figure 17a with *BresLos*.

As described in Section 6, assume that the current node is S, that $S.location$ is the Green cell X and that the Gray cells are in $S.seen$. Cells A and B are the two *LOS-disjoint* pivots and their frontier watchers (cells F, I) are added as children of S with JF.

Map	Alg	Exp	Gen	Time	BF	Depth	Cost
Maze 21X21	Opt	8,146	27,658	143,008	4.5	20.4	**182.52**
	IW	2,448	8,039	**30,662**	5	11.8	**182.52**
	WR	1,064	5,981	46,819	11	5.4	183.48
	Both	**1,020**	**5,678**	43,506	11	5.4	183.29
Den405d	Opt	840	3,662	20,608	9	11.2	**99.69**
	IW	46	177	792	9	4.3	**99.69**
	WR	26	153	818	13	2.4	100.17
	Both	**10**	**84**	**238**	15	1.9	100.17
Map from Figure 17	Opt	4.83	12.91	3.93	2.68	2.71	**10.37**
	IW	2.65	7.30	0.74	2.39	1.75	10.46
	WR	4.12	10.83	1.93	2.35	2.12	**10.37**
	Both	**2.18**	**5.28**	**0.49**	1.78	1.47	**10.37**

Table 6: Subsections 10.1 and 10.2 results

Next, based on the former method for JF of WRP-A*, White cell C becomes an auxiliary pivot and its frontier watcher J is also added as a child of S to a total of 2 neighbors: $\{F, J\}$ (I is reduandant). It turns out that the optimal path (that is colored Red in Figure 17a) first jumps to the new child J (adding the two rightmost columns to *seen*). Then, it jumps left to F (adding pivot B to *seen*). Finally, it jumps to D covering the left side of the figure and halts. This is the optimal path of length 8.

By contrast, when we ignore the White cells and not add White components to G_{DLS}, we do not add C as an auxiliary pivot. As a consequence, J is not added as a child of S which now has two children: F and I. In this setting of ignoring While cells, the minimal-cost path first jumps to F. The situation after this is shown in Figure 17b and the corresponding node is denoted by S'. Now, $S'.location = F$ (F is colored Green) and B and some of its watchers are covered (i.e., added to $S'.seen$, colored Gray). $G_{DLS}1(S')$ now includes A as a pivot but C also becomes a pivot because it is *LOS-disjoint* with A. So, now three cells are added as children of S': D ($\in LOS_{To}(A)$), J and K ($\in LOS_{To}(C)$). The minimal-cost path from F will go to K and cover the right side of the map, and then will go to D to cover the left side of the map and halt. The path returned (from the original location X), when ignoring White cells, is of length 10.

Here, we gave an example where components of White cells are crucial for finding the optimal solution. Ignoring White cells reduces the search tree but optimality is not guaranteed. We note that this example was carefully handcrafted. In practice, however, such cases are very rare and, usually, optimal solutions are returned (but not guaranteed), even when White cells are ignored from the JF procedure as we experimentally demonstrate next.

Table 6 presents results on a 21x21 maze (Figure 18) and on the Den405d map (Figure 16) averaged over 30 instances. It also provides results for the example of Figure 17. One can see that *Ignore Whites* (IW) reduced the number of expanded nodes and the time by up to a factor of 30 over WRP-A* (Opt). We also added the average branching factor (BF) and average depth of the leaves of the final search tree (Depth). One can see that the search tree is much shallower compared to the optimal tree. Indeed, BF can get larger but the depth of the search is significantly smaller, causing fewer nodes to be expanded. In

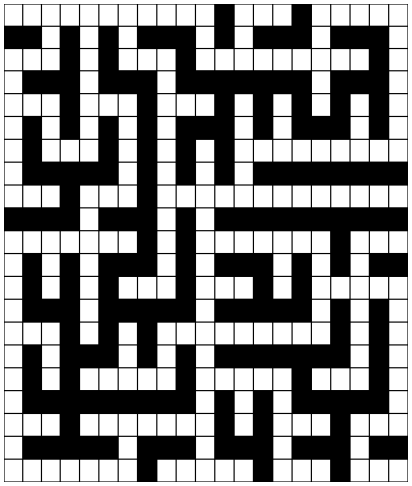


Figure 18: Maze 21X21

almost all instances, the optimal path was returned, even though it was not formally guaranteed. Thus, the phenomenon of losing optimality, as presented in Figures 17a and 17b, happens very rarely. We performed similar experiments on 17 other maps, 50 instances per map and IW always returned an optimal solution. This suggests that IW is practically effective and that the quality of the solution will not be hurt too much.

10.2 Removing Likely Redundant Pivots

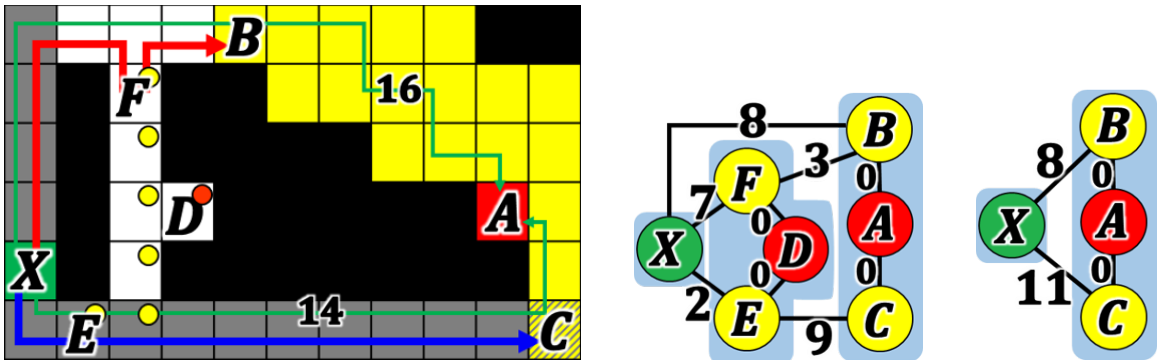
One of our goals is to solve larger problem instances. The number of pivots will increase accordingly because we will be able to identify more *LOS-disjoint* pivots to G_{DLS}. This will increase the TSP heuristic values because they are based on G_{DLS}. But, it will also increase the size of G_{DLS} as well as the branching factor of the main search (assuming that JF is applied) because there will be more jump points that will be added as children of a node. We next reduce the size of G_{DLS} by removing some of the pivots from it. To do so intelligently, we want to remove pivots with minimal effect on the quality of the solution returned.

Above, we defined *redundant components* and suggested deleting them from G_{DLS} keeping only *cardinal components* when calculating the graph heuristics. Recall that G_{DLS} is built from components of pivots and their watchers, and that to have an admissible heuristic, all inner edges of a component cost 0. Thus, the additional cost of passing through a component is 0. However, when removing redundant components, the costs of the paths inside these components are accumulated when jumping to the cardinal component because we take the true shortest path along an edge. So removing redundant pivots may reduce the time overhead of calculating the heuristics and may even improve their values. Nevertheless, for JF, redundant components are needed as we jump to frontier watchers of neighboring components to guarantee the optimality of the solution.

But, since we are now willing to forfeit optimality for running time we next suggest to delete redundant components from the search tree of JF. For this we provide a greedy method that further relaxes the definition of redundant component to a *weakly redundant*

component. We then suggest to modify JF such that it will not add such components as jumping points. Such modification will reduce the size of the search tree of JF but optimality might be sacrificed. We describe this process next.

Recall that the definition of redundant components in Section 5.4.2 is strict — *all* paths to a cardinal component Q must pass through component P to make P a redundant component. Let Q be a component with q as its pivot. A component P is *weakly redundant* with respect to Q if *one* shortest path in the underlying original map from $S.location$ to q passes through component P. Practically, when building G_{DLS} and finding a new *LOS-disjoint* pivot p, we also find a shortest path to p. There might be many such shortest paths. But, in order to reduce the computation, we chose the first one that we find with an A* search on the underlying graph G. Alternatively, we can use the *all pairs shortest paths* (APSP) database if one is built, as suggested in Section 3.4. If that path includes a watcher of q then component Q is weakly-redundant. So, Q is removed from G_{DLS}. Importantly, removing weakly-redundant components preserves completeness — as long as *unseen* is not empty, at least one point will be selected and the search continues.



(a) WR Example Map (b) WR Example G_{DLS} (c) G_{DLS} with WR

Figure 19: Suboptimality of weakly redundant pivots

Consider the example in Figure 19a. Assume we are at node S where $S.location = X$ and that $S.seen$ already contains all the Gray cells. Figure 19b shows $G_{DLS}(S)$ that has two pivots A and D. It is important to note that the component of D is not fully *redundant* because there exists a path to the component of A that does not pass through watchers of D (the topmost path of cost 8 to B which is a watcher of A). The optimal path (colored Red in Figure 19a) is to jump to watcher F of pivot D and then jump to watcher B of pivot A with solution cost of 10.

Now, observe that the shortest path from X to pivot A passes through E which is a watcher of pivot D. So, in this case, the component of D (which includes E, the cell to its right, and F as frontier watchers) is a *weakly redundant* component and we remove D and its watchers from $G_{DLS}(S)$, as seen in Figure 19c. Removing the component of D will lose the optimality of the solution. After removing the component of D there are now two possible solutions to develop in the search tree: (1) Jump to watcher C of pivot A while seeing pivot D on the path to C with cost of 11. (2) Jump back to watcher B of pivot A

and then jump to watcher F of pivot D (which will become a pivot again) also with solution cost of 11.

Table 6 also presents our greedy method to detect and prune weakly-redundant pivots (WR). A reduction in nodes and in time is seen when compared to the optimal WRP-A* variant (Opt) at a cost of a small loss in the quality of the solution. IW tends to be slightly faster in time than WR although it expanded slightly more nodes. When adding WR on top of IW (the *Both* line), we see that in the maze, results were not improved by a large margin. But, in Den405d there is a total reduction of a factor of 80 in nodes and of 86 in time over Opt. Again, this was at a negligible loss ($\sim$0.4%) in the quality of the solution. We also report the average branching factor and depth of the search tree. As can be seen, all these algorithms reduce the depth of the search tree significantly by up to a factor of 6. As can be seen in Table 6, WR has an effect which is similar to improving a heuristic: it reduces the number of nodes but increases the time overhead per node. By contrast, IW ignores some information in order to reduce the size of the search tree.

10.3 Bounding the Jump Points

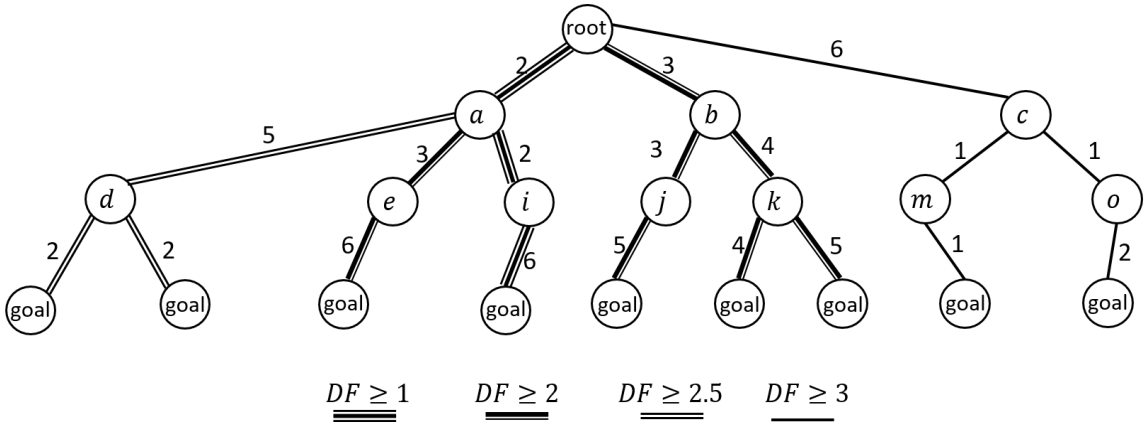
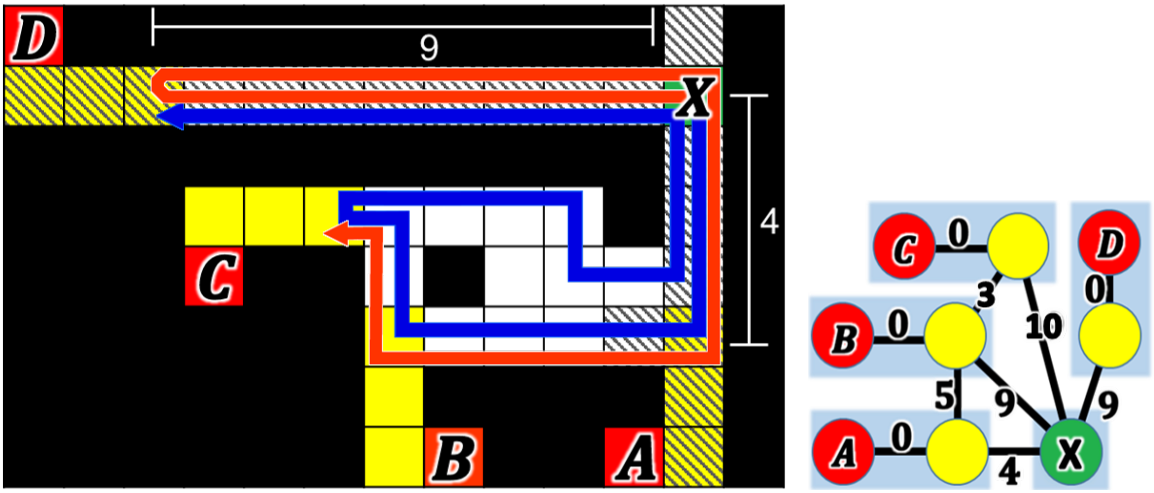


Figure 20: DF tree example

Observe the following two attributes in a search tree rooted at S when JF is applied. **(1)** Edges going out from S can have different costs. **(2)** All leaves of the search tree are goal nodes because the algorithm will keep generating children as long as there are unseen cells in the graph.

We would like to further reduce the branching factor of the search tree of JF. Inspired by the *nearest neighbor* heuristic in TSP (Hurkens & Woeginger, 2004), we only connect *S.location* to nearby jump points but prune away jump points that are far from *S.location*. This is done as follows. Let $\epsilon(S)$ be the cost of the edge to the closest jump point from *S.location*. Let *distance factor* (DF) be a constant factor. We now connect *S.location* only to jump points whose edges have costs C such that $C \leq DF \times \epsilon(S)$. For example, Figure 20 presents a search tree where edges costs are different and all leaves are goal nodes. The partial expanded trees of different DF values are marked with different line thickness. When $DF = 1$, and $\epsilon(root) = 2$ then at the root we will prune edges from *root* to b and c because their cost C is $C > DF \times \epsilon(S) = 2$, and thus the *root* is only connected to a.

When $DF = 2$, then $DF \times \epsilon(root) = 2 \times 2 = 4$ and we will now jump to a and to b but prune c. This algorithm will be complete (because of attribute 2 just described), but the solution might be suboptimal.



(a) Bounded Example Map (b) Bounded Example G_{DLS}

Figure 21: Suboptimality of Bounded Jump

The optimal path in Figure 21a is the Red arrow (X, D, A, B, C) and its cost is 30. The corresponding G_{DLS} is shown in Figure 21b. When $DF = 2$ is applied on X ($S.location$), the watcher of pivot A is the only child. Any of the frontier watchers of other pivots are with distance ≥ 9 which is more than twice than the nearest child which is at distance 4 from X. Thus, the Blue path (X, A, B, C, D) of cost 31 is returned.

Map	DF	Exp	Gen	Time	BF	Cost
Maze 21X21	1	**130**	**134**	**629**	1.04	207.48
	1.2	167	183	763	1.13	206.14
	1.5	597	723	3,421	1.32	191.67
	2	4,312	6,493	31,314	1.92	186.90
	4	6,075	12,766	60,114	2.84	**182.52**
	Opt	8,146	27,658	143,008	4.48	**182.52**
Den405d	1	**200**	**213**	**1,253**	1.45	115.97
	1.2	306	375	1,287	2.20	109.10
	1.5	441	755	3,729	3.29	102.66
	2	698	1,530	7,220	4.78	101.10
	4	780	2,373	12,432	6.83	**99.69**
	Opt	840	3,662	20,608	9.29	**99.69**

Table 7: Bounded Jump Operator

Table 7 presents results for the DF technique on our two domains (without applying IW and WR). Indeed, in both domains, smaller values of DF resulted in smaller branching factors and therefore faster solutions times and fewer node expansions. For the maze, the

improvement was up to ×60 in nodes and up to ×225 in time while for Den405d it was up to ×4 in nodes and up to ×20 in time. The quality of the solution was never more than 15% over Opt ($DF = \infty$).

10.4 Combining All Methods

Map	DF	Exp	Gen	Time	BF	Cost
Maze 21X21	1	**29**	**30**	**647**	1.08	218.48
	1.1	33	37	709	1.20	215.71
	1.2	84	104	1,118	1.52	204.05
	1.5	365	660	4,052	2.61	187.76
	2	503	1,403	10,112	4.43	183.29
	4	932	4,434	31,585	8.78	183.29
	∞	1,020	5,678	43,506	10.83	183.29
	Opt	8,146	27,658	143,008	4.48	**182.52**
Den405d	1	**4**	**5**	**14**	1.14	107.83
	1.1	6	24	65	7.00	101.21
	1.2	7	34	227	8.87	100.59
	1.5	8	54	155	12.50	100.45
	2	9	60	297	13.01	100.45
	4	9	74	211	13.52	100.17
	∞	10	84	238	14.93	100.17
	Opt	840	3,662	20,608	9.29	**99.69**

Table 8: All improvements on two maps

Table 8 shows the results when combining all our three techniques (IW, WR and DF). Different rows are for different DF values. Dramatic improvements are seen. The maze can be solved in less than a second and the map can be solved in 14ms. This comes at an increased cost above the optimal solution of 20% for the maze and 8% for Den405d.

We also studied the effect of increasing density of obstacles. We experimented on the same 13x13 maze with 71 obstacles used above. Here too, we built 71 instances $(1 \ldots 71)$ where instance $\#n$ only had n obstacles randomized from the 71 obstacle set. The full maze as well as the halfway point in which only 35 obstacles are present are shown in the top left corner of Figure 22(top). The results in that figure are for all 71 instances averaged over 25 trials that randomly chose the subset of obstacles. Figures 22(top) and 22(bottom) present the solution length and the CPU time, respectively (y-axis), as a function of increasing number of obstacles (x-axis).

The different curves are as follows. $DF = 1$ is for the DF technique only. $All = 1$ is for all three improvements where $DF = 1$ (similarly $All = 2$ has $DF = 2$). Five of our variants obtained almost-optimal solutions except $All = 1$ and $DF = 1$ which were a little larger but never off by more than 20%. As for runtime, one can observe a phase transition behavior for all variants. $All = 1$ was best on time – significantly better (almost an order of magnitude) than all other variants. $All = 2$ produced the best balance between solution quality (almost optimal) and running time (only worse than $All = 1$). Importantly, these

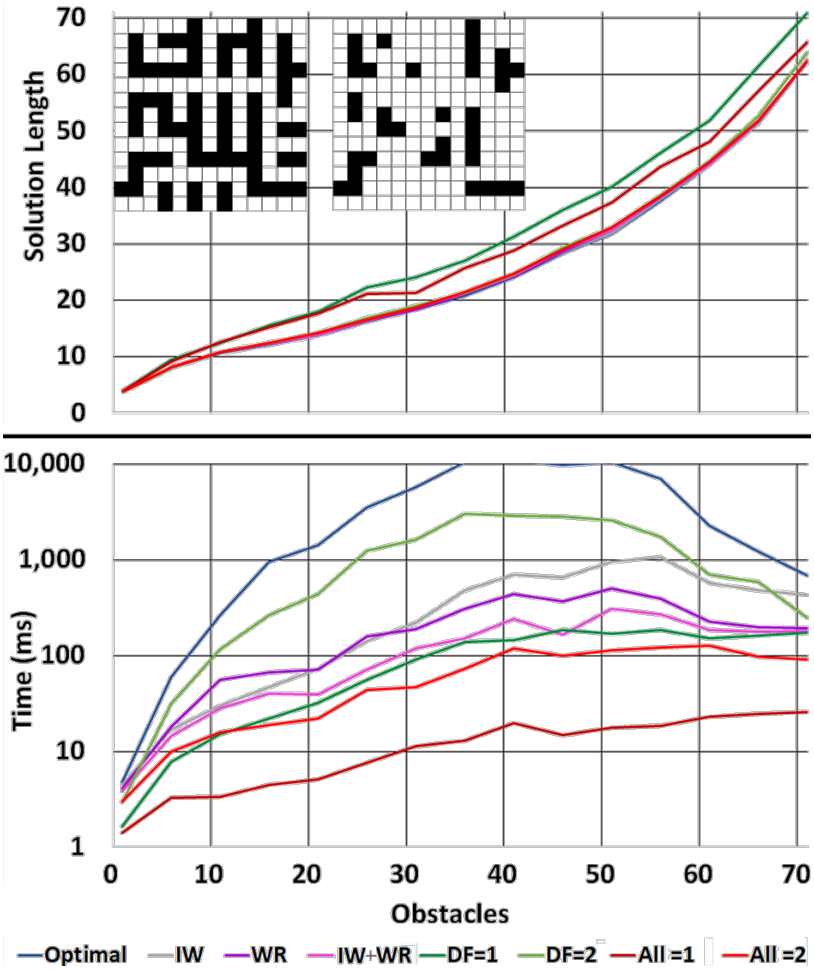


Figure 22: Different densities for the 13x13 maze

trends can be seen for all densities of obstacles, implying the robustness and generality of our improvements.

Finally, Table 9 compares combining WA* with all our methods tested on Den020d which is a complex large map with 3,100 free cells, shown in Figure 23. Both IW and WR are used and the tested parameters are DF and W for WA*. Increasing W somewhat speeds up the search at a cost of a longer solution. Reducing DF also speeds up the search but tends to better preserve the solution quality. When both are combined, we get better quality solutions (than each of them alone) in less time which implies that the combination is beneficial.

10.5 Anytime Algorithms

In many cases, the amount of time given to a problem solver is limited or unknown. Anytime algorithms are used for such cases (Zilberstein, 1996). A first (usually low quality) solution is found as fast as possible. Then, as time allows, better and better solutions are found. In our former paper (Yaffe, Skyler, & Felner, 2021), we introduced an anytime solver called

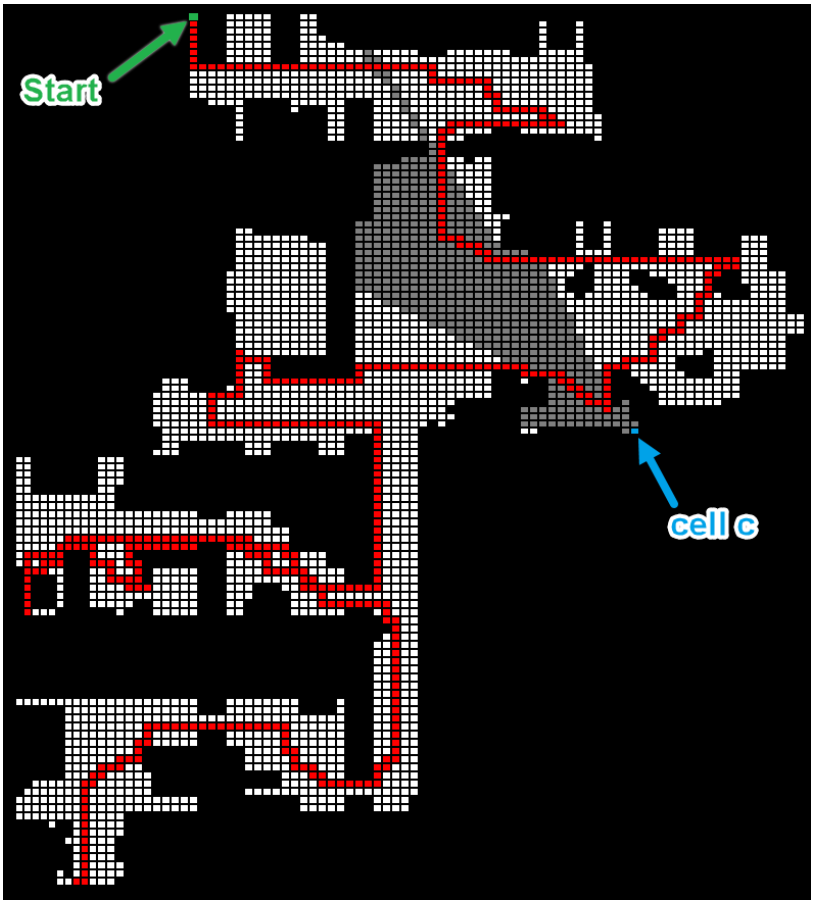


Figure 23: Den020d

Anytime DF (ADF) which iterates over increasing values of DF. ADF is a general algorithm that is applicable beyond WRP and it has many properties that were not yet studied. Thus, it deserves a paper on its own and discussing it is left for future work.

11. Discussion and Conclusions

In this paper, we solved the *Watchman Route Problem* with heuristic search and provided strong admissible heuristics for this problem. Our heuristics showed their effectiveness by allowing a vast reduction in the search effort. In particular, the TSP heuristic is usually most effective. However, if the environment has very large open areas then the Singleton heuristic should be used. We also presented algorithms that solve WRP for bounded and unbounded suboptimal solutions. A significant reduction is shown in the search effort with a relatively low increase in the cost of the solution when the suboptimal variants are used. Thus, we can now solve much larger problems.

We feel that we have touched the surface of an iceberg regarding this fascinating problem. Future work will continue in the following directions:

DF	W	Alg	Exp	Gen	Time	BF	Cost
1	1	A*	21	23	**488**	1.17	532.11
	2	WA*	**14**	**18**	646	1.25	532.11
	10	WA*	**14**	**18**	652	1.25	532.11
1.2	1	A*	105	154	4,292	3.85	514.22
	2	WA*	**15**	**55**	**746**	3.72	520.67
	10	WA*	**15**	57	1,147	3.83	523.33
1.5	1	A*	243	421	10,357	5.97	506.89
	2	WA*	**16**	106	**2,309**	6.62	514.00
	10	WA*	18	**98**	2,627	5.70	535.56
2	1	A*	422	998	43,195	8.90	506.22
	2	WA*	19	139	**4,934**	7.77	517.56
	10	WA*	**22**	**136**	6,172	6.60	543.89
∞	1	A*	761	5,246	175,221	25.75	**492.00**
	2	WA*	18	465	**7,860**	28.34	523.71
	10	WA*	**16**	**400**	8,831	26.56	681.71

Table 9: WA* variants with our algorithm on Den020d

- There are a number of approaches for adding sensing abilities to the problem. In this paper we assumed that at each location sensing is done for free to all directions. Another approach is adding a specific operator for sensing which the agent can activate once arriving at a node, or even during its movements. Additionally, different types of sensing can be applied, e.g., to a different direction, or with a different sensing radius or a different sensing angle. These approaches will significantly increase the search space due to a much larger branching factor of the different possible operators. Another approach is to add sensing costs in parallel to travel costs. This will make the problem a *Bi-Objective Search Problem* (BOS) (Hernandez, Yeohz, Baier, Zhang, Suazoy, & Koenig, 2020b). BOS is currently extensively researched, and WRP may be a nice application for that research.

- Algorithms can be developed for more advanced search settings, such as *bounded cost search* (Stern, Puzis, & Felner, 2011) where the task is to find a solution with cost $\leq C$ where C is a constant. Similarly, anytime algorithms that improve the quality of the solution on the fly should be developed. These will be compared to existing general solvers for these settings that are based on A*.

- Future work will solve this problem in a multi-agent setting where multiple agents need to find watchman routes while combining their efforts under different communication paradigms, such as centralized controller and centralized knowledge sharing or different variants of distributed decision abilities and distributed sharing of information. In addition, ADF is a general framework which can be applied to other domains such as TSP and other graph problems, especially when all leaves are solutions and the costs of edges will be used to apply the algorithm.

- From a robotics point of view, one may complicate the system by adding noise, discretization problems and online errors. This will enable to modify this work to a real environment with real robots.

12. Acknowledgments

Portions of this work have been previously published (Seiref, Jaffey, Lopatin, & Felner, 2020; Yaffe et al., 2021). This paper ties together all the results, provides more insights, more theoretical understandings, more experimental results, and presents a comprehensive manuscript that summarizes this line of work. The research was supported by Rafael Advanced Defense Systems and by the United States-Israel Binational Science Foundation (BSF) under grant numbers 2017692 and 2021643. We deeply thank Shahaf Shperberg for his comments and his help.

Appendix A. WRP on a Grid is NP-hard

We next prove that our version of WRP is NP-hard also on grids. We label the grid map G and use $\mathcal{C}$ to denote the set of empty (traversable) cells.

Definition 3 (Hamiltonian Cycle Problem)**.** *The* Hamiltonian Cycle Problem *(HCP) in 4-connected grids gets as input a grid and the output is a cyclic path that travels through each of the empty (non-obstacle) cells of the grid exactly once.*

HCP is proven to be NP-Complete (Garey & Johnson, 1979) and in particular in grids (Itai, Papadimitriou, & Szwarcfiter, 1982).

Up until now, we only used the TSP variant that allows repetitions, i.e., multiple visits at each cell (or vertex). We next prove the following 4 claims. **(1)** We prove that TSP without repetitions (TSP_{once}) on a grid is NP-hard. **(2)** We then reduce TSP_{once} to TSP (with repetitions). **(3)** We then show that NC-TSP on a grid is NP-hard by reducing TSP to it. **(4)** Finally, we show that WRP on a grid is NP-hard because NC-TSP on a grid is a special case of WRP on a grid.

Lemma 12. *TSP_{once} (i.e., without repetitions) on 4-connected grids is NP-hard.*

Proof. In 4-connected grids the cost of moving between two neighboring empty cells is 1. Thus, both problems TSP_{once} and HCP are identical in 4-connected grids. So, the proof that HCP on grids is NP-hard (Itai et al., 1982) is also a proof for TSP_{once} on a grid. □

Definition 4. *In the decision variant of TSP_{once} in 4-connected grids, we are given a grid G and cost K, and the task is to decide whether there is a (cyclic) TSP_{once} path (that visits each cell exactly once) on G with cost $\leq K$.*

Definition 5. *In TSP (repetitions allowed) in 4-connected grids, we are given a grid G and cost K. The task is to decide whether there is a (cyclic) TSP path of cost $\leq K$ that passes through all cells in G at least once. In the optimal variant of TSP in 4-connected grid domains the task is to find the optimal (shortest) such path.*

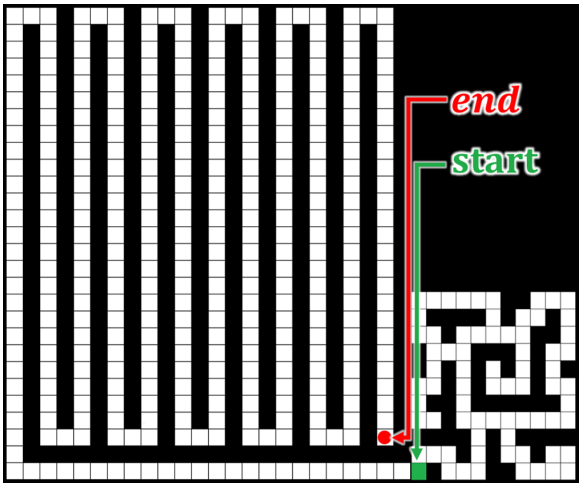


Figure 24: The new grid G'. The original grid G is shown in the bottom right of the figure.

Observation 2. *In any 4-connected grid G (with $|\mathcal{C}|$ cells), a cyclic path that travels through all cells of G has a cost of at least $|\mathcal{C}|$. Moreover, the cost of such path is exactly $|\mathcal{C}|$ iff every cell is visited exactly once.*

Lemma 13. *TSP on 4-connected grids is NP-hard.*

Proof. We reduce TSP_{once} to TSP. Assume a TSP_{once} instance with input grid G and cost K. According to Observation 2, if $K < |\mathcal{C}|$ then no solution exists for both problems. If $K \geq |\mathcal{C}|$, any solution for TSP_{once} (if exists) costs exactly $|\mathcal{C}|$. We thus build a respective TSP problem where we set $K' = |\mathcal{C}|$ and we need to decide whether there is a TSP path on grid G with cost $\leq K' = |\mathcal{C}|$.

Direction 1: Assume that there is a TSP path on grid G with cost $\leq K' = |\mathcal{C}|$. According to Observation 2, the cost of the path is $|\mathcal{C}|$ and, hence, visits each cell exactly once. Therefore, this is also a valid TSP_{once} path that costs $\leq K$ ($K \geq |\mathcal{C}|$).

Direction 2: Assume that there is a TSP_{once} path on grid G with cost $K \geq |\mathcal{C}|$. According to Observation 2, the cost of the path must be exactly $|\mathcal{C}|$ and, hence, the path is also a valid TSP with cost $\leq K' = |\mathcal{C}|$. □

Observation 3. *The solution path returned by a TSP solver is cyclic. Any cell (or vertex for general graph cases) on that path can be considered as the start and end of that path.*

Lemma 14. *NC-TSP (Defined above - non-cyclic path with a start cell) on 4-connected grid domains is NP-hard.*

Proof. We reduce TSP (assuming that cells can be visited multiple times) to NC-TSP. Assume a TSP instance with input grid G with traversable cells $\mathcal{C}$ and cost K. We build a grid G' with traversable cells $\mathcal{C}'$. All cells in G also appear in G' but we add more cells to the map (labeled *cor*) that are connected to some cell $start \in \mathcal{C}$[11] (i.e., $cor = \langle start, ..., end \rangle$).

11. According to Observation 3, any cell in $\mathcal{C}$ can be chosen as the *start* cell.

start is at the border of G (or next to an obstacle which has a path of obstacles to a border cell). We denote the cell at the other side of *cor* from *start* as *end*, and the length of the corridor *cor* is $|cor| = U = 2 \cdot |\mathcal{C}|$. We now ask whether there is a NC-TSP path on grid G' that starts in *start* with cost $\leq K + U$.

We next prove the reduction. That is, we prove that there is a TSP path of cost $\leq K$ on G iff there is a NC-TSP path from *start* of cost $\leq U + K$ on G'.

Direction 1: Assume that there is a TSP path on grid G with cost $X \leq K$.[12] We can now construct a NC-TSP path on G' with cost $X + U \leq K + U$ as follows. The path will start at *start*, follow the TSP path on G which is a cyclic path with cost X. Then, it will go over *cor* from *start* to *end* with cost U to a total cost of $X + U \leq K + U$.

Direction 2: Assume that there is a NC-TSP path on graph G' from *start* with cost $X \leq K+U$. We divide our discussion to two cases. Case 1: $K > U$. Observe that there is a trivial TSP tour as follows. Find an MST of G and assume that its cost is M. Then, cover the edges of the MST twice (from the root of the tree to the leaves and back). Now, observe that since this is a 4-connected grid then $M = |\mathcal{C}| - 1$. Thus, the cost of this TSP tour is $2 \cdot M = 2 \cdot |\mathcal{C}| - 2 < U < K$. In fact, when $K > |cor|$ then there is also always a NC-TSP path on G' with cost $\leq K + U$ by adding *cor* from *start* to *end* to the TSP path on G that was just described. Case 2: $K \leq U$. In this case, the NC-TSP path cannot pass twice in *cor* because it will incur a cost of $2 \cdot U$ for *cor*, and X for G, where $2 \cdot U + X > K + U$. So, the optimal path only passes on *cor* once with cost U (it ends at *end*). The rest of the path must be a TSP tour on G with cost $\leq X - U \leq U + K - U = K$

□

We next reduce NC-TSP on a grid to our version of WRP on a grid (where there is a specific start vertex and the path ends once all cells in $\mathcal{C}$ have been seen).

Lemma 15. *WRP on a grid is NP-hard.*

Proof. We reduce NC-TSP on a grid to WRP on a grid. The reduction is easy. NC-TSP on a grid is a special case of WRP on a grid where LOS is defined such that $LOS_{From}(c) \equiv \{c\}$ for all cells $c \in \mathcal{C}$. That is, a cell can only see itself (LOS radius $R = 0$). Thus, one must pass through the entire set of cells $\mathcal{C}$. □

References

Appel, K., & Haken, W. (1977). Every planar map is four colorable. part i: Discharging. *Illinois J. Math.*, *21*(3), 429–490.

Aulinas, J., Petillot, Y. R., Salvi, J., & Lladó, X. (2008). The SLAM problem: a survey.. *CCIA*, *184*(1), 363–371.

Bresenham, J. E. (1965). Algorithm for computer control of a digital plotter. *IBM Systems journal*, *4*(1), 25–30.

Chazelle, B. (2000). A minimum spanning tree algorithm with inverse-ackermann type complexity. *J. ACM*, *47*, 1028–1047.

12. X might be $> |\mathcal{C}|$ because multiple visits are allowed

Chen, J., & Sturtevant, N. R. (2019). Conditions for avoiding node re-expansions in bounded suboptimal search. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, pp. 1220–1226.

Chin, W.-P., & Ntafos, S. (1986). Optimum watchman routes. In *Proceedings of the second annual symposium on Computational geometry*, pp. 24–33. ACM.

Chin, W.-P., & Ntafos, S. (1991). Shortest watchman routes in simple polygons. *Discrete & Computational Geometry*, *6*(1), 9–31.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms*. MIT press.

Dechter, R., & Pearl, J. (1985). Generalized best-first search strategies and the optimality of A*. *J. ACM*, *32*(3), 505–536.

Dissanayake, M. G., Newman, P., Clark, S., Durrant-Whyte, H. F., & Csorba, M. (2001). A solution to the simultaneous localization and map building (SLAM) problem. *IEEE Transactions on Robotics and Automation*, *17*(3), 229–241.

Dror, M., Efrat, A., Lubiw, A., & Mitchell, J. S. (2003). Touring a sequence of polygons. In *Proceedings of the Thirty-Fifth Annual ACM Symposium on Theory of Computing*, pp. 473–482. ACM.

Fisk, S. (1978). The nonexistence of colorings. *Journal of Combinatorial Theory, Series B*, *24*(2), 247–248.

Fu, M., Kuntz, A., Salzman, O., & Alterovitz, R. (2019). Toward asymptotically-optimal inspection planning via efficient near-optimal graph search. *CoRR*, *abs/1907.00506*.

Galceran, E., & Carreras, M. (2013). A survey on coverage path planning for robotics. *Robotics and Autonomous Systems*, *61*(12), 1258–1276.

Garey, M., & Johnson, D. (1979). *Computers and Intractability, A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, New York, NY, USA.

Geisberger, R., Sanders, P., Schultes, D., & Delling, D. (2008). Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In McGeoch, C. C. (Ed.), *Experimental Algorithms, 7th International Workshop, WEA 2008, Provincetown, MA, USA, May 30-June 1, 2008, Proceedings*, Vol. 5038 of *Lecture Notes in Computer Science*, pp. 319–333. Springer.

Gilon, D., Felner, A., & Stern, R. (2016). Dynamic potential search - A new bounded suboptimal search. In *Proceedings of the Ninth Annual Symposium on Combinatorial Search (SoCS)*, pp. 36–44.

Greenberg, H. J. (1998). Greedy algorithms for minimum spanning tree..

Harabor, D. D., Uras, T., Stuckey, P. J., & Koenig, S. (2019). Regarding Jump Point Search and Subgoal Graphs. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*, pp. 1241–1248.

Hart, P. E., Nilsson, N. J., & 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.

Held, M., & Karp, R. M. (1970). The traveling-salesman problem and minimum spanning trees. *Operations Research*, *18*(6), 1138–1162.

Hernandez, C., Yeoh, W., Baier, J., Zhang, H., Suazo, L., & Koenig, S. (2020a). A simple and fast bi-objective search algorithm. In *Proceedings of the International Conference on Automated Planning and Scheduling*, pp. 143–151.

Hernandez, C. U., Yeohz, W., Baier, J. A., Zhang, H., Suazoy, L., & Koenig, S. (2020b). A simple and fast bi-objective search algorithm. In *ICAPS*, pp. 143–151.

Holte, R. C., Felner, A., Newton, J., Meshulam, R., & Furcy, D. (2006). Maximizing over multiple pattern databases speeds up heuristic search. *Artificial Intelligence*, *170*(16), 1123–1136.

Honsberger, R. (1979). *Mathematical plums*. Mathematical Association of America.

Hurkens, C. A., & Woeginger, G. J. (2004). On the nearest neighbor rule for the traveling salesman problem. *Operations Research Letters*, *32*(1), 1–4.

Itai, A., Papadimitriou, C. H., & Szwarcfiter, J. L. (1982). Hamilton paths in grid graphs. *SIAM J. Comput.*, *11*(4), 676–686.

Kang, S. H., Kim, S. J., & Zhou, H. (2015). Path optimization with limited sensing ability. *Journal of Computational Physics*, *299*, 887–901.

Li, F., & Klette, R. (2008). An approximate algorithm for solving the watchman route problem. In *International Workshop on Robot Vision*, pp. 189–206. Springer.

Michail, D., Kinable, J., Naveh, B., & Sichi, J. V. (2020). Jgrapht—a java library for graph data structures and algorithms. *ACM Trans. Math. Softw.*, *46*(2).

Mitchell, J. S. (2013). Approximating watchman routes. In *Proceedings of the twenty-fourth annual ACM-SIAM symposium on Discrete algorithms*, pp. 844–855. SIAM.

O'Rourke, J. (1987). *Art Gallery Theorems and Algorithms*. Oxford University Press, Inc., USA.

Papadimitriou, C. H. (1977). The euclidean travelling salesman problem is np-complete. *Theoretical Computer Science*, *4*(3), 237–244.

Pohl, I. (1973). The avoidance of (relative) catastrophe, heuristic competence, genuine dynamic weighting and computational issues in heuristic problem solving. In *International Joint Conference on Artificial Intelligence (IJCAI-73)*, pp. 12–17.

Pohl, I. (1970). First results on the effect of error in heuristic search. *Machine Intelligence*, *5*, 219–236.

Rivera, N., Hernández, C., & Baier, J. A. (2017). Grid Pathfinding on the $2k$ Neighborhoods. In *Proceedings of the Thirty-First Conference on Artificial Intelligence AAAI*, pp. 891–897.

Salvetti, M., Botea, A., Saetti, A., & Gerevini, A. E. (2017). Compressed path databases with ordered wildcard substitutions. In Barbulescu, L., Frank, J., Mausam, & Smith, S. F. (Eds.), *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling, ICAPS 2017, Pittsburgh, Pennsylvania, USA, June 18-23, 2017*, pp. 250–258.