# Suboptimally Solving the Watchman Route Problem on a Grid with Heuristic Search

**Tamir Yaffe, Shawn Skyler, Ariel Felner**
SISE Department, Ben-Gurion University, Be'er-Sheva, Israel
{tamiry,shawn}@post.bgu.ac.il, felner@bgu.ac.il

## Abstract

In the Watchman Route Problem (WRP) we are given a grid map with obstacles and the task is to (offline) find a (shortest) path through the grid such that all cells in the map can be visually seen by at least one cell on the path. WRP was recently formalized and optimally solved with heuristic search. In this paper we show how the previous optimal methods can be modified (by intelligently pruning away large subtrees) to obtain suboptimal solvers that are much faster than the optimal solver without sacrificing too much the quality of the solution. In particular, we derive bounded suboptimal solvers, suboptimal solvers without bounds and anytime variants. All these algorithms are backed up with experimental evidence that show their benefits compared to existing approaches.

## Introduction

Imagine you are in a museum and you want to see all the exhibits on the floor. To do so, you want to take a tour around the museum such that you can see every item in all the rooms. Similarly, the security of the museum wants to have a known path such that during its traversal it will be able to see every item in the exhibit to check that it was not damaged. 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 and Ntafos 1986).

In this paper we focus on WRP on grid maps although many of our methods can be generalized to any type of graph. We are given a grid map with obstacles and a start state. The task is to (offline) find a path from the start state through the grid such that all empty cells in the map were visually covered by line-of-sight (LOS) from at least one of the cells on the path. The LOS function determines whether any given two cells can visually see each other and it can be any arbitrary function. Trivial LOS functions on grids are the cardinal lines (4-way) possibly combined with diagonal lines (8-way). An example of a non-trivial LOS function on a general graph 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 shortest path.



Seiref et al. (2020) recently worked on this problem. They proved that the grid variant of WRP is NP-hard and then formalized and optimally solved it with heuristic search. They abstracted the grid map into a *disjoint line-of-sight graph* ($G_{DLS}$) and used this graph as a source for admissible heuristics. Their best heuristic was based on a solution to a variant of the traveling salesman problem (TSP) applied on $G_{DLS}$. Then, they executed an A*-like search with these heuristics. We call their algorithm WRP-A*. WRP-A* optimally solved the problem for grids of up to 1,500 cells in a number of seconds. Since the problem is NP-hard, there is a limit to the size of grids that can be solved.

In this paper we build on their work but aim to solve much larger problems by sacrificing the optimallity of the solution. We therefore provide different search algorithms and methods which modify WRP-A* to three search cases:

**(1) Bounded Suboptimal Search.** Given a bound $W$ the task is to find a solution path $\pi$ where cost($\pi$)$\leq W \times C^*$, where $C^*$ is the cost of the optimal solution. Here we compare variants of WA* as well as of XDP and XUP (Chen and Sturtevant 2019). Experimental results show that these variants are much faster than WRP-A* while their solution is much better then the bound that they guarantee.

**(2) Suboptimal Search.** Here 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.

**(3) Anytime Search.** Here we find a first solution fast by applying our suboptimal solver. Then by iterating on increasing costs of operators allowed (see below) we get better and better solutions until we converge to optimal.

We experiment on mazes and maps of different size taken from 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.

## Related Work

In the field of robotics, the *Simultaneous Localization And Mapping* problem (SLAM) includes an autonomous moving agent that tries to explore the environment and build a map while simultaneously locate itself in the map (Aulinas

et al. 2008; Taketomi, Uchiyama, and 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 study the map by a moving agent. By contrast, WRP is an offline search problem on a known map.

A reminiscent problem is the Art Gallery Problem which was proven to be NP-hard (Garey and Johnson 1979). We are given a map 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 $s \in S$. Indeed, the shortest possible tour between these points is a solution to WRP. But, Seiref et al. (2020) showed that such a solution may not be optimal for WRP (while still being NP-hard).

Another related problem is *inspection planning* (Fu et al. 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). The solution is a motion plan for the robot that maximizes the number of POI inspected and minimizes the cost of the plan. WRP can be seen as a special case where *all* POIs must be seen and that every cell of the grid is a POI. In addition, we focus on 2D grids while inspection planning works on a continuous high-dimensional configuration space (for the actions of the robot) which is built on top of the physical environment.

WRP has been extensively researched on polygonal domains in the field of computational geometry. The objective was to find a cyclic path that sees all internal points of a closed polygon. Two points can see each other (have LOS) if no edge of the polygon cuts the straight line between them. The problem was proven to be NP-hard but a polynomial-time approximation algorithm exists with increased cost over the optimal solution by a factor of $O(\log^2 n)$, where $n$ is the number of vertices of the polygon (Mitchell 2013). For simple polygons (with no internal holes) there are polynomial-time algorithms (Chin and Ntafos 1986; Dror et al. 2003); the best time achieved was $O(n^3 \log n)$. A specific variant of WRP is to define a start point that the agent must travel from. This is called *fixed* WRP or *anchored* WRP (Xu 2014), as opposed to floating WRP where no such point is required. Our setting is significantly different as assume a discrete grid (not a continuous polygon) and may accommodate any LOS function. Furthermore, we do not require to end the path by returning to the start location but it can end at any cell once all cells have been seen.

We next summarize how WRP was formulated and optimally solved as a heuristic search problem (Seiref et al. 2020) and then provide our new suboptimal algorithms.

## WRP as a Heuristic Search Problem

**Problem Definition:** The input for our variant of WRP is a grid-map $M$. The set of empty (traversable) cells is labeled hereafter by $\mathcal{C}$ and un-traversable cells are denoted as *obstacles*. We are also given a cell $start \in \mathcal{C}$ as input. In this paper, for simplicity, we assume that only the four cardinal moves are legal. Cells $p$ and $q$ are *adjacent* iff there is a legal move from $p$ to $q$ (and vice versa). Generalizing our work to allow diagonal moves (8-way) or other moves (such as the $2^k$ neighborhood moves (Rivera, Hernández, and Baier 2017)) as well as to other types of graphs should not be hard because our algorithm is defined on a general graph.

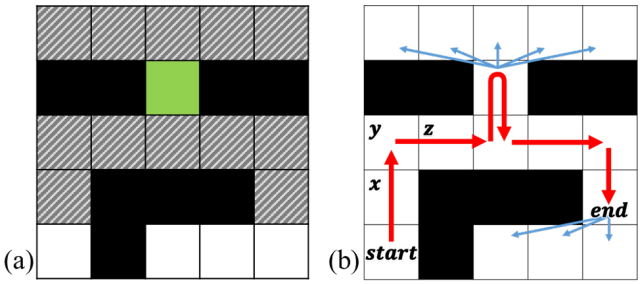


Figure 1: Example of *BresLos* and Optimal Path Example

A path $\pi = \langle s_0 = start, \ldots, s_k \rangle$ is a sequence of adjacent cells starting from *start*. The task is to find a *watchman path* in the grid. In a *watchman path* $\pi$, for every cell $c \in \mathcal{C}$ there is *line-of-sight* from at least one cell $s_i \in \pi$. An optimal watchman path is a watchman path with minimal cost.[1]

The *line-of-sight* function (LOS) determines whether any given two cells can visually see each other and it can be any arbitrary function. Seiref et al. (2020) experimented with three possible LOS functions on grids: 4-way, 8-way and *Bresenham LOS* (*BresLos*) (Bresenham 1965). *BresLos* is commonly used in computer graphics, video games and bitmap pictures. *BresLos* discretizes real-world continuous domains and simulates a continuous field of view. 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 Gray cells in Figure 1a have *BresLos* to/from the Green cell. The red arrows in Figure 1b show an optimal solution for this map.

### The Search Tree of WRP

Seiref et al. (2020) formulated WRP as a search problem and defined its corresponding search tree as follows.

**Node:** A node is a pair $\langle location, seen \rangle$ where $location$ is a cell (current location of the agent) and *seen* is a list of cells (all the cells that have been seen so far by the agent). The complement of *seen* is the *unseen* list; their union is the entire set of cells (*seen* $\cup$ *unseen* $= \mathcal{C}$).

**Root Node:** $Root$ is a node such that $Root.location = start$ and $Root.seen = LOS(start)$.

**Expansion:** Expanding node $S = \langle location, seen \rangle$ is to perform all legal movements on $S.location$. For each child $S'$ of $S$, $S'.location$ is the adjacent cell 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 cells that are now seen from $S'.location$ and were not seen before (i.e., $S'.seen = S.seen \cup LOS(S'.location)$). The cost of the edge from $S$ to $S'$ is the cost of the movement action.

[1] We assume that the watchman does not have to return to the start cell. The importance of the problem is that *all* cells were seen. The reason is that, practically, we do not want to restrict the whereabouts of the watchman after the task is completed. It might stay idle, it might leave through the nearest exit or might destroy itself.

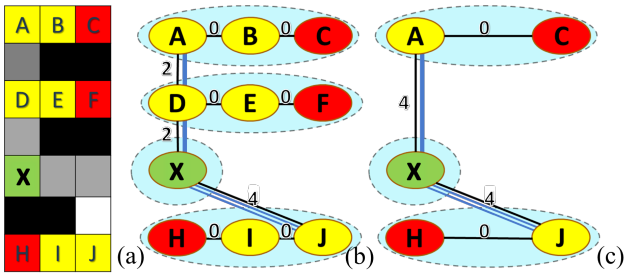


Figure 2: A grid (a). Its $G_{DLS}$ (b). Its abstracted $G_{DLS}$ (c).

**Goal Node:** $Goal.location$ may be any cell in $\mathcal{C}$ such that $Goal.seen = \mathcal{C}$.

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 associated with $S$.[2] $S.seen$ includes all the cells that have LOS to at least one of the locations in the path associated with $S$. The cost of node $S$ (i.e., its $g$-value) in the tree is the sum of the costs of applying the operators along the branch from $Root$ to $S$.

Seiref et al. (2020) introduced WRP-A*, a variant of A* that works on the state-space defined for WRP. The admissible heuristic used by WRP-A* is based on an abstracted graph called $G_{DLS}$ which is described next.

## Graph Abstraction for WRP Heuristics

A *Disjoint LOS Graph*, $G_{DLS}(S) = (V, E)$ (Seiref et al. 2020) is abstracted from the grid map $M$ for every node $S$ in *Open*. This graph is used to obtain several admissible heuristics as discussed below. Figure 2 shows an example grid (2a) and its corresponding $G_{DLS}$ (2b) assuming 4-way LOS.

We say that two cells are $LOS\text{-}disjoint$ if there is no cell that they both see. We say that a set of cells $P$ is $LOS\text{-}disjoint$ if every pair of cells in $P$ are $LOS\text{-}disjoint$. $G_{DLS}$ is built by first identifying a set of $LOS\text{-}disjoint$ cells $P \subseteq S.unseen$. Each cell in $P$ is called a *pivot* and is added as a vertex to $G_{DLS}$. Pivots are colored Red in our figures. For each pivot $p \in P$ we also add all cells in $LOS(p)$ as *watcher vertices* to the graph (colored Yellow). The edges in $G_{DLS}$ between pivot $p$ and its watchers have weight of 0. A pivot $p$ and its watchers are referred to as the *component* of $p$ (circled in light blue) and the cost of traveling inside a component is therefore 0.[3] *Frontier watchers* are watchers of pivot $p$ that have at least one neighboring cell that does not have LOS to $p$. For example, cell $A$ is a frontier watcher but cell $B$ is not. Frontier watchers are connected with frontier watchers of other pivots if the shortest path between them does not pass through other components in $G_{DLS}$. For example, $A$ is connected to $D$ but not to $J$ (because the components of $F$ and $X$ are in between). The weight of edges between frontier watchers of different pivots is the cost of the true distance between them. For example, the weight between $A$ and $D$ is 2. The current location of the agent, $S.location$ ($X$, colored Green) is also added as a component to $G_{DLS}$. Note that watchers of $X$ (Gray cells) as well as all the rest of the cells (White cells, only one which is present in Figure 2a) are not added to $G_{DLS}$.

Two additional abstractions are done on $G_{DLS}$ (Figure 2c). First, non-frontier watchers (e.g., cells $B$, $E$ and $I$ in Figure 2a) are removed from $G_{DLS}$ since all the edges connecting them have weight 0. Second, 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, the component of $F$ is redundant with respect to the component of $C$. Neighbors of redundant components are then directly connected to cover the removal. For example, $X$ is now connected to $A$ with weight 4 as shown in Figure 2c. Components that are not redundant are called *cardinal* and they remain in $G_{DLS}$.

## TSP Heuristic

Seiref et al. (2020) provided several admissible heuristics based on $G_{DLS}$. We only describe and use their best heuristic - the TSP heuristic. A solution to *Traveling Salesman Problem* (TSP) on a graph $G = (V, E)$ is a cycle that passed through all the vertices of $V$. Finding optimal (minimum-cost) solutions to TSP is NP-hard (Held and Karp 1970).

When $G_{DLS}(S)$ is present we are interested in the minimum-cost Hamiltonian path from $X$ that visits all other vertices in $G_{DLS}$. Since edges between pivots and their watchers are of weight 0 then this is a lower bound on the path that will see all vertices in $S.unseen$. Seiref et al. (2020) slightly modified a TSP solver to find the minimal path that starts at location $X$ and passes in all pivots of $G_{DLS}$. In our example in Figures 2b and 2c, the path colored Blue (X,J,X,A) is the required minimum-cost Hamiltonian path from $X$ that visits all other components in $G_{DLS}$. Its cost 12 is used as the TSP heuristic.

For more implementation details and specific data structures (such as the all-pairs shortest-path table) for $G_{DLS}$ and the TSP heuristic, see Seiref et al. (2020).

## Reducing the Size of the Search Tree

We next cover an additional enhancement introduced for WRP-A*. Trivially, when node $S$ is expanded, then new nodes are generated for all the cells that are adjacent to $S.location$. However, Seiref et al. (2020) significantly reduced the size of the search tree by using edges of the abstracted $G_{DLS}(S)$ (Figure 2c) to generate the children of $S$. An optimal path must include at least one frontier watcher for each pivot in $G_{DLS}(S)$. To have a complete search (and not lose any possible path) we add *all* of the *frontier watchers* as children of $S$ in the search tree. Formally, when ex-

[2]Two nodes are duplicates if their current location and seen list are identical. This enables to prune nodes which have the same location and the same *seen* list as they represent exactly the same situation for the search task.

[3]Our pivot selection policy (Seiref et al. 2020) iteratively takes a cell $P \in S.unseen$ with the fewest watchers and adds it to $G_{DLS}$ until no cell can be added to the set of $LOS\text{-}disjoint$ pivots. This is a greedy policy that balances the pivots selection time and the quality of the heuristic derived from the $G_{DLS}$ that is formed.

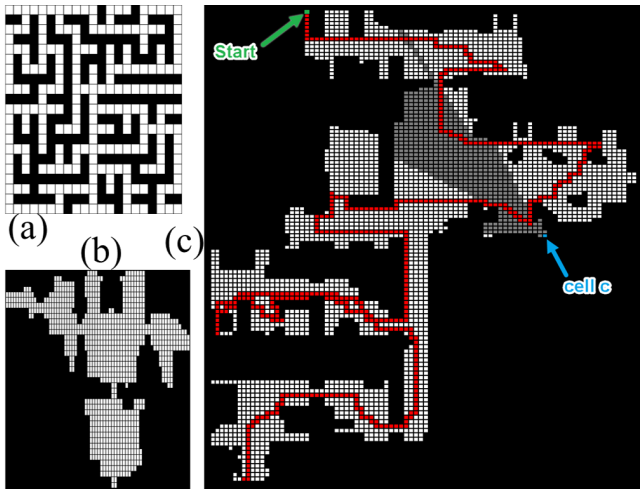


Figure 3: (a) Maze 21X21 (b) Den405d (c) Den020d

panding a node $S$, we generate one child $C$ for each edge in $G_{DLS}(S)$ that connects $S.location$ to a frontier watcher $W$ as follows: $C.location = W$. The cost of edge $(S, C)$ in the search tree is set to the cost of the corresponding edge in $G_{DLS}(S)$. $C.seen$ is updated to also include $LOS(p)$ for each cell $p \in \pi(S.location, W)$. For example, in Figure 2b $X$ is connected to $D$ with cost 2 and to $J$ with cost 3 and thus two children will be generated. This method is denoted as *Jump to Frontier expansion* (JF) and the neighbors generated with JF are called *jump points* (inspired by Jump Point Search (Harabor et al. 2019)).

To guarantee optimality we also need to take care of the possibility that the optimal path passes via a cell which is not in $G_{DLS}$. Such cells are colored White in Figure 2a (only one such cell is present in the figure above cell $J$). To do so, we artificially build components for the White cells and add them to $G_{DLS}$ as follows. We iterate over all White cells. Given a White cell $x$ we add $x$ as a pivot (Red) and add all the other White cells with LOS to $x$ as its watchers. We then add this artificial component to $G_{DLS}$ in the same manner as real pivots. We continue this process until all White cells are taken care of. We then add the frontier watchers of the White cells as children of $S$. Importantly, these components are not considered by the TSP heuristic because their pivots are not $LOS\text{-}disjoint$ with the other pivots. They are only added later (after the heuristic computation) for the branching process of JF. We use $G_{DLSJ}$ to denote the resulting $G_{DLS}$ after White cells were added. In Section  below, we further discuss the pros and cons of adding White cells to $G_{DLS}$ and also give illustrative examples.

WRP-A* with reasonable memory and time resources cannot solve large problems. We therefore introduce suboptimal (bounded and unbounded) and anytime algorithms.

## 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).

| 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 1: Results of WA*, XDP and XUP on Den405d map

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 robust across domains (Gilon, Felner, and Stern 2016).[4] 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 and 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 states 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:

$$f_{XDP} = \frac{1}{2w} \cdot (g + (2w-1)h + \sqrt{(g-h)^2 + 4wgh})$$

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:

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

We experimented with WA*, XDP and XUP on the Den405d map (Sturtevant 2012) (Figure 3b) with different values for $W$. Here, and in our experiments and explanations below we exclusively use the *BresLos* function. Table 1 provides the results (time in msec, number of expanded and

[4]We did not experiment with Explicit Estimation Search (Thayer and Ruml 2011) and Dynamic Potential Search (Gilon, Felner, and Stern 2016) because they re-order their open lists every time the minimal $f$-value changes. This is very costly and can be done effectively only 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.

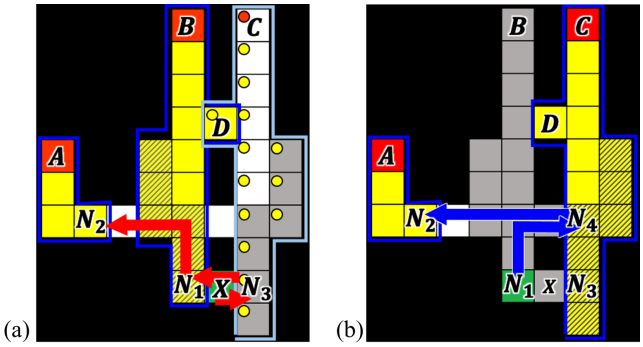


Figure 4: Suboptimality of Ignoring White cells

| Map | Alg | Exp | Time | BF | Depth | Cost |
|---|---|---|---|---|---|---|
| Maze 21X21 | Opt | 8,146 | 143,008 | 4.5 | 20.4 | **182.52** |
| | IW | 2,448 | **30,662** | 5 | 11.8 | **182.52** |
| | WR | 1,064 | 46,819 | 11 | 5.4 | 183.48 |
| | Both | **1,020** | 43,506 | 11 | 5.4 | 183.29 |
| Den405d | Opt | 840 | 20,608 | 9 | 11.2 | **99.69** |
| | IW | 46 | 792 | 9 | 4.3 | **99.69** |
| | WR | 26 | 818 | 13 | 2.4 | 100.17 |
| | Both | **10** | **238** | 15 | 1.9 | 100.17 |

Table 2: IW and WR Results

generated nodes and the solution cost) averaged over 30 random start states. Chen and Sturtevant (2019) report that XDP tends to expand fewer nodes on the domains 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 $W = 10$.

## Suboptimal Algorithm

In BSS, there is a guarantee of $W$ on the suboptimality In this section we describe a sub-optimal algorithm without a suboptimality bound. To do so, we modify WRP-A* (Seiref et al. 2020) by adding three techniques that reduce the size of the search tree produced by 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 be distanced no more than a constant more than the distance of the nearest neighbor. We cover these next.

### Ignoring White Cells

Consider Figure 2a again. Recall that *White cells* are unseen cells that do not have LOS to any of the pivots (Red cells) in $G_{DLS}$. As explained in section  to ensure the optimality of the solution, given a node $S$ we build (artificially) components for the White cells for the JF expansion (but not for the TSP heuristic computation) resulting in $G_{DLSJ}$.

We now suggest to ignore the White cells and not add White components thus applying JF on the original $G_{DLS}$ (and not $G_{DLSJ}$). 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 4a with *BresLos*.

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 $G_{DLS}(S)$ we find two $LOS\text{-}disjoint$ pivots — cells $A$ and $B$, each has a component of watchers colored Yellow which are surrounded by a Blue border. With JF we add cells $N_1$ and $N_2$ as children of $S$. Note that cell $C$ is White, i.e., it is not $LOS\text{-}disjoint$ with the other pivots because cell $D$ has LOS both to pivot $B$ and to cell $C$. Next, based on the former method of Seiref et al. (2020) we further add (artificial) components for White cells too resulting in $G_{DLSJ}$. So, the White cell $C$ becomes a pivot (marked with red circle on top) and all the cells with yellow circle on top are its watchers. This new component is surrounded by a Light-blue border. The frontier watchers of $C$ are cells $N_3$, $N_4$ and $D$ which are also added as neighbors of $S$ to a total of 5 neighbors: $\{N_1, N_2, N_3, N_4, D\}$[5]. It turns out that the optimal path (colored Red in Figure 4a) first jumps to the new child $N_3$ (adding the two rightmost columns to $seen$). Then, it jumps left to $N_1$ (adding pivot $B$ to $seen$). Finally, it jumps to $N_2$ 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 artificial pivot. As a consequence, $N_3$, $N_4$ and $D$ are not added as a children of $S$ which now only has two children: $N_1$ and $N_2$. In this setting the minimal path first jumps to $N_1$. The situation after this is shown in Figure 4b and the corresponding node is denoted by $S'$. Now, $S'.location = N_1$ ($N_1$ is colored Green) and $B$ and some of its watchers are covered (i.e., added to $S'.Seen$, colored gray). $G_{DLS}(S')$ now includes $A$ as a pivot but $C$ also becomes a pivot because it is $LOS\text{-}disjoint$ with $A$. So, now three cells are added as children of $S'$: $N_2$ (LOS to $A$),$N_3$ and $N_4$ (LOS to $C$). The minimal path from $N_1$ will go to $N_4$ and cover the right side of the map and then will go to $N_2$ 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 building components of White cells is crucial for finding the optimal solution. Ignoring the white cells reduces the search tree but optimality is not guaranteed. We note that this example was very carefully handcrafted. In practice, however, such cases are very rare.

Table 2 presents results on a 21x21 maze (Figure 3a) and

[5]Strictly speaking, there are two other white cells in the figure that are being treated in the same way. But, for simplicity of the description we omit these from the discussion.

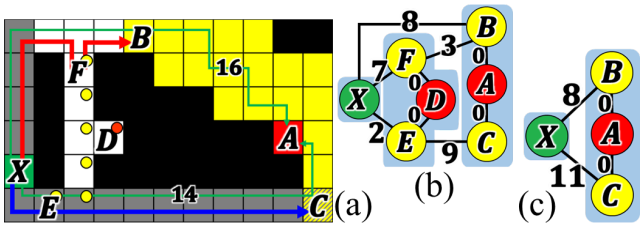


Figure 5: Suboptimality of weakly redundant pivots

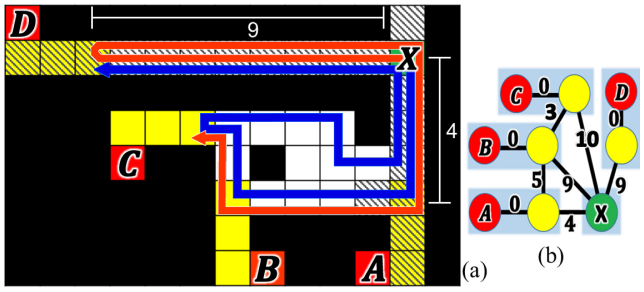


Figure 6: Suboptimality of Bounded Jump

on the Den405d (Figure 3b) map averaged over 30 instances. 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 is larger but the depth of the search is significantly smaller causing fewer nodes to be expanded as the table shows. In almost all instances the optimal path was returned, even though it was not formally guaranteed. Thus, the phenomenon of losing optimality as presented in Figure 4 happened very rarely in these instances. This suggests that IW is practically effective and that the quality of the solution will not be hurt too much.

## 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 $LOS\text{-}disjoint$ 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 because there will be more frontier watcher 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 to completely delete them from $G_{DLS}$ keeping only *cardinal components*. This results in two important positive effects. First, the depth of the search tree is reduced. Frontier watchers of redundant pivots will not be added as jump points. Instead, we add jump points directly to frontier watchers of the cardinal pivots which were connected to those redundant pivots that were deleted. Second, this may improve the quality of the TSP heuristic. 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 have weight of 0. Thus, the additional cost of passing through a component is 0. But, when redundant components are removed the cost of the paths inside these components is accumulated when jumping to the cardinal component because we take the true shortest path along an edge.

The definition of redundant components is strict — *All* paths to a cardinal component $Q$ must pass through component $P$ to make $P$ a redundant component. We next provide a greedy method that relaxes the notion of redundant component to *weakly-redundant component* to be deleted. Now, more components will be deleted and the search tree will be smaller but optimality might be sacrificed. 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 map from $S.location$ to $q$ passes through component $P$. Practically, when building $G_{DLS}$ and finding a new $LOS\text{-}disjoint$ pivot $p$ we also find a shortest path to $p$.[6] If that path includes a watcher of $q$ then component $Q$ is weakly-redundant. So, $Q$ is removed from $G_{DLS}$.

Consider the example in Figure 5. Assume we are at node $S$ where $S.location = X$ and that $S.seen$ already contains all the gray cells. Figure 5b shows $G_{DLS}(S)$ which 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). The optimal path (colored Red in Figure 5a) 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 5c. Removing the component of $D$ will lose the optimality of the solution. There are now two possible solutions: (1) Jump to watcher $C$ of pivot $A$ while seeing pivot $D$ on the path to $C$ with cost of 11. (2) Jump 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 2 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* (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. Also, IW always preserved the optimality of the solution (even though it was not guaranteed). 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.

[6]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 grid. Or, we can use the *all pairs shortest paths* database if one is built as suggested by Seiref et al. (2020).

| 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 3: Bounded Jump Operator

We also report the average branching factor and depth of the search tree and as can be seen all these algorithms reduce the depth significantly by up to a factor of 6.

## Bounding the Jump Points

Observe the following two attributes in the 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 and 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 of the closest jump point from $S.location$. Let *distance factor* (DF) be a constant factor. We connect $S.location$ only to jump points that their edges have weights $w$ such that $w \leq DF \times \epsilon(S)$. For example, with $DF = 1$ only the closest jump points will be generated. When $DF = 2$ only jump points that the distance between them and $S.location$ is at most twice than $\epsilon(S)$ will be generated. This algorithm will be complete (because of attribute 2 above), but the solution might be suboptimal.

The optimal path in Figure 6a is the Red arrow $(X, D, A, B, C)$ and its cost is 30. The corresponding $G_{DLS}$ is shown in Figure 6b. 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 $\geq 9$ which is more than twice than the nearest child which is at distance 4 from $X$. Thus, the Blue path path $(X, A, B, C, D)$ of cost 31 is returned.

Table 3 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 $\times 60$ in nodes and up to $\times 240$ in time while for Den405d it was up to $\times 4$ in nodes and up to $\times 20$ in time. The quality of the solution was never more than 15% over Opt ($DF = \infty$).

| 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 |
| | $\infty$ | 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 |
| | $\infty$ | 10 | 84 | 238 | 14.93 | 100.17 |
| | Opt | 840 | 3,662 | 20,608 | 9.29 | **99.69** |

Table 4: All improvements

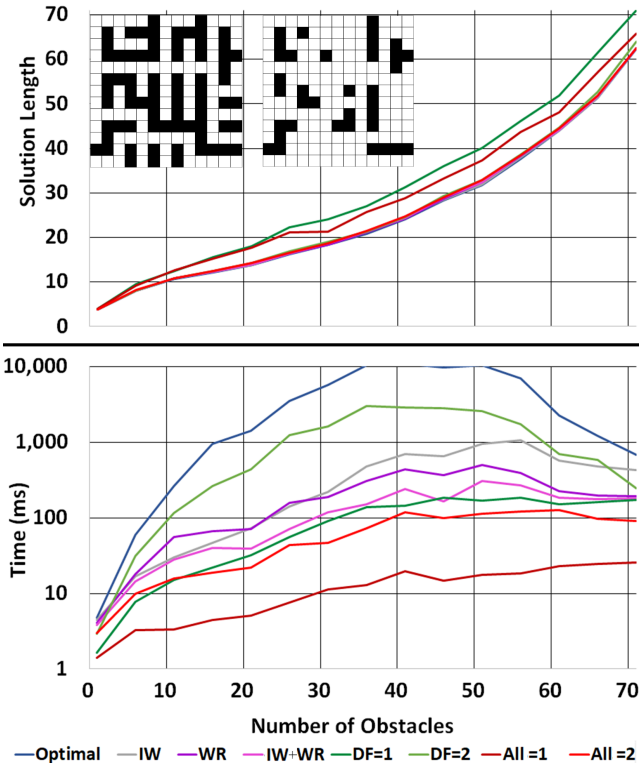


Figure 7: Different densities for the 13x13 maze

## Combining All Methods

Table 4 shows the results when combining all our three techniques. 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 less than 15 msec. 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 a 13x13 maze with 71 obstacles, from which we built 71 instances $(1 \ldots 71)$ where instance

| 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 |
| $\infty$ | 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 5: WA* variants with our algorithm on Den020d

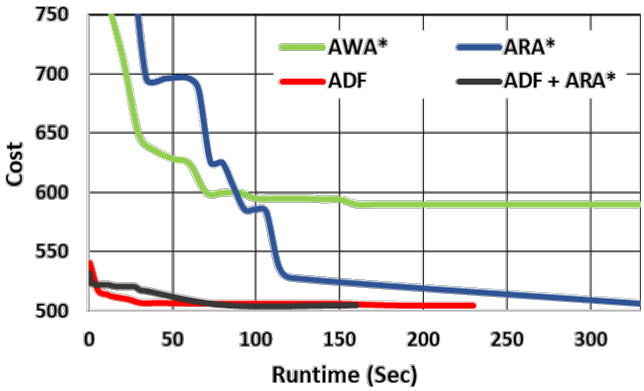


Figure 8: Anytime Algorithms Comparison

$\#n$ 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 topleft corner of Figure 7a. Results for all 71 instances averaged over 25 trials that randomly chose the subset of obstacles. Figures 7a and 7b present the solution length and the CPU time, respectively, ($y$-axis) as a function of increasing number of obstacles ($x$-axis). $DF = 1$ 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 an easy-hard-easy 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 trends can be seen for all densities of obstacles implying the robustness and generality of our improvements.

Finally, Table 5 compares combining WA* with all our methods tested on Den020d which is a complex large map with 3,100 free cells shown in Figure 3c. Both IW and WR are used and the tested parameters are DF and $W$ for WA*. Increasing $W$ 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 in less time which implies that the combination is beneficial.

## Anytime DF

In many cases the amount of time given to 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. Usually, anytime algorithms converge to the optimal solution if enough time is given.

Anytime heuristic search (AWA*) (Hansen and Zhou 2007) is a simple, general anytime search framework. AWA* executes WA* until a first solution is found. Then, the execution continues on the remaining open list until more solutions are found. An alternative formulation is that of ARA* (Likhachev, Gordon, and Thrun 2003). The main idea is to execute iterations of WA*. The first iteration starts with a high value of $W$ and then $W$ is reduced from iteration to iteration according to some *reducing schedule*. If that schedule has $W = 1$ as its last value then it will converge to optimal because the last iteration is identical to A*. An enhanced version of ARA* can pass information from iteration to iteration thereby expanding each node only once.

Here, we propose an anytime search algorithm based on operator costs which we call *Anytime DF* (ADF). ADF iterates over increasing values of DF. We start with $DF = 1$ and find a first solution. Then, we increase $DF$ according to an *increasing schedule*. When the scheduled value converges to $\infty$ the entire search tree is generated and an optimal solution will be at hand. A basic implementation starts each run of a new DF value from scratch. Similar to ARA*, an enhanced method may maintain special data structures that enable to pass information from iteration to iteration (such as nodes that were pruned in the previous iteration). This will allow to expand each node only once.

We compared basic ADF, AWA* ($W = 2$), basic ARA* and a combination of ADF and ARA* where both $W$ and DF are iteratively changed. IW and WR were always used in this specific experiment. Therefore, the algorithms will converge to the solution length of IW+WR. Start points were taken from different regions of the Den20d map (figure 3c). Figure 8 plots the cost of the solution achieved ($y$-axis) as a function of the runtime in seconds ($x$-axis). As could be expected, at first all algorithms find high cost solutions and as time passes solution costs are improved. At the beginning AWA* finds better solutions than ARA*. But, given our time limit, AWA* is unable to further improve the quality of the solution beyond 590. After 90 seconds ARA* outperforms AWA* and finds better solutions. Clearly, our new algorithms ADF and ADF+ARA* are superior to the WA* variants. They find first solution faster with lower cost than ARA* and AWA* and converge faster to a low cost solution. We note that the optimal solution is unknown as the optimal solver could not solve these instances. ADF+ARA* finds the first solution with better quality than and converge

to its minimum faster than ADF. But in the range between 2 to 80 seconds ADF finds better solutions, which means that there is a tradeoff between them.

## Conclusions and Future Work

We presented algorithms that solve WRP for bounded and unbounded suboptimal solutions. We also presented an anytime framework based on these. In general, significant reduction is shown in the search effort with a relatively low increase in the cost of the solution.

Future work will solve this problem in a multi agent setting where multiple agents need to find watchman routes while combining their efforts. In addition, ADF is a general framework which can be applied to other domains such as TSP and other graph problems.

## Acknowledgments

The research was supported by Rafael Advanced Defense Systems, by Israel Science Foundation (ISF) grant $\#844/17$ to Ariel Felner and by the Cyber grant by from the Prime Minister office.

## References

Aulinas, J.; Petillot, Y. R.; Salvi, J.; and 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.

Chen, J.; and 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*, 1220–1226.

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

Dror, M.; Efrat, A.; Lubiw, A.; and Mitchell, J. S. 2003. Touring a sequence of polygons. In *Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*, 473–482. ACM.

Fu, M.; Kuntz, A.; Salzman, O.; and Alterovitz, R. 2019. Toward Asymptotically-Optimal Inspection Planning via Efficient Near-Optimal Graph Search. *CoRR* abs/1907.00506. URL http://arxiv.org/abs/1907.00506.

Garey, M. R.; and Johnson, D. S. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co. ISBN 0716710447.

Gilon, D.; Felner, A.; and Stern, R. 2016. Dynamic Potential Search - A New Bounded Suboptimal Search. In *Proceedings of the Ninth Annual Symposium on Combinatorial Search (SoCS)*, 36–44.

Hansen, E. A.; and Zhou, R. 2007. Anytime heuristic search. *Journal of Artificial Intelligence Research* 28: 267–297.

Harabor, D. D.; Uras, T.; Stuckey, P. J.; and Koenig, S. 2019. Regarding Jump Point Search and Subgoal Graphs. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*, 1241–1248. doi:10.24963/ijcai.2019/173. URL https://doi.org/10.24963/ijcai.2019/173.

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

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

Likhachev, M.; Gordon, G. J.; and Thrun, S. 2003. ARA*: Anytime A* with provable bounds on sub-optimality. *Advances in neural information processing systems* 16: 767–774.

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

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

Pohl, I. 1973. The avoidance of (relative) catastrophe, heuristic competence, genuine dynamic weighting and computational issues in heuristic problem solving. In *IJCAI*, 12–17.

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

Seiref, S.; Jaffey, T.; Lopatin, M.; and Felner, A. 2020. Solving the Watchman Route Problem on a Grid with Heuristic Search. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 30, 249–257.

Sturtevant, N. R. 2012. Benchmarks for Grid-Based Pathfinding. *IEEE Trans. Comput. Intellig. and AI in Games* 4(2): 144–148. doi:10.1109/TCIAIG.2012.2197681. URL https://doi.org/10.1109/TCIAIG.2012.2197681.

Taketomi, T.; Uchiyama, H.; and Ikeda, S. 2017. Visual SLAM algorithms: a survey from 2010 to 2016. *IPSJ Transactions on Computer Vision and Applications* 9(1): 16.

Thayer, J. T.; and Ruml, W. 2011. Bounded Suboptimal Search: A Direct Approach Using Inadmissible Estimates. In *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011*, 674–679.

Xu, N. 2014. On The Watchman Route Problem and Its Related Problems. *Dissertation Proposal* .

Zilberstein, S. 1996. Using anytime algorithms in intelligent systems. *AI magazine* 17(3): 73–73.