Proceedings of the Thirtieth International Conference on Automated Planning and Scheduling (ICAPS 2020)

Solving the Watchman Route Problem on a Grid with Heuristic Search

Shawn Seiref
Ben Gurion Univ.
Be'er Sheva, Israel
shawn@post.bgu.ac.il

Tamir Jaffey
Ben Gurion Univ.
Be'er Sheva, Israel
tamiry@post.bgu.ac.il

Margarita Lopatin
Ben-Gurion University
Be'er Sheva, Israel
marglup@rnd-hub.com

Ariel Felner
Ben Gurion Univ.
Be'er Sheva, Israel
felner@bgu.ac.il

Abstract

In this paper we optimally solve the *Watchman Route Problem* (WRP) on a grid. 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. We formalize WRP as a heuristic search problem and solve it with an A*-based algorithm. We develop a series of admissible heuristics with increasing difficulty and accuracy. In particular, our heuristics abstract the problem into line-of-sight clusters graph. Then, solutions for the minimum spanning tree (MST) and the traveling salesman problem (TSP) on this graph are used as admissible heuristics for WRP. We theoretically and experimentally study these heuristics and show that we can optimally and suboptimally solve problems of increasing difficulties.

1 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 proven to be NP-hard for polygons (Chin and Ntafos 1986). Nevertheless, finding an optimal route is the main focus of this paper.

In our variant of WRP, 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. In the optimal variant of WRP we seek for the shortest path with these attributes.

The LOS function determines whether any given two cells can visually see each other and it can be any arbitrary function. 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(a) shows the shortest possible tour such that each empty cell is seen by a vertical/horizontal LOS.

Copyright © 2020, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

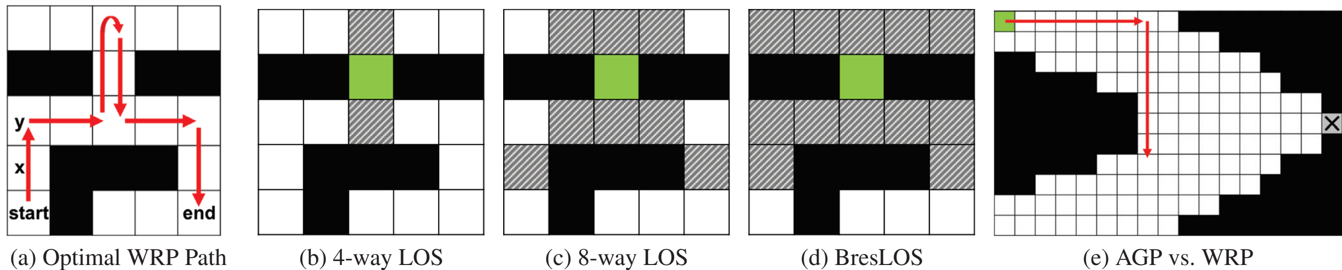
In this paper we formally define WRP on a grid map including a number of variants for the LOS capability and formulate it as a heuristic-search problem. We then propose several admissible heuristics for WRP of increasing difficulty and accuracy, which can be applied on top of the A* algorithm (Hart, Nilsson, and Raphael 1968). In particular, we abstract the map into a *disjoint line-of-sight graph* (G_{DLS}), which is built from disjoint components, 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 a novel node expansion mechanism that significantly reduces the the search tree by directly jumping to the different LOS regions of G_{DLS}.

Finally, we provide an experimental study where we applied all our variants on a large number of different grids maps. We show that our strongest variant can optimally solve grids with up to 1500 cells in several hundreds of seconds and sub-optimally in a few seconds.

2 Related Work

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 locate itself in the map (Dissanayake et al. 2001). Survey on SLAM algorithms appear in (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 offline problem is the *Art Gallery Problem* (AGP) which was proven to be NP-hard (Garey and 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 $s \in S$. Indeed, the shortest possible tour between these points is a solution to WRP, but it may not be optimal. This is shown in Figure 1(e). The gray point, labeled X, is optimal with regards to the number



(a) Optimal WRP Path (b) 4-way LOS (c) 8-way LOS (d) BresLOS (e) AGP vs. WRP

Figure 1: Example of WRP with each of the LOS functions.

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.

WRP has been extensively researched on polygonal domains in the field of computational geometry. The object was to find a cyclic path that sees all internal points of a closed polygon. In general the problem was proved 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) that identify a set of lines inside the polygon that the optimum watchman route must include. The best known algorithm for simple polygons has a running time of $O(n^3 logn)$ (Dror et al. 2003). 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 (Xu 2014), as opposed to floating WRP where no such point is required.

None of these works is directly relevant to our problem because of the following differences. They assume a polygonal, continuous environment, with a specific LOS 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 discrete grid and may accommodate any line-of-sight function. Furthermore, we do not require to end the path by returning to the start state but it can end at any cell once all cells have been seen.

3 Problem Definition

We now define the variant of WRP we focus on in this paper. The input is a grid-map M. The set of empty (traversable) cells is labeled hereafter by $\mathcal{C}$ and $n = |\mathcal{C}|$. Blocked (untraversable) 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 or other moves (such as the 2^k neighborhood moves (Rivera, Hernández, and Baier 2017)) should not be too hard.

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* π, 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.

In the main part of the paper 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. 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 9.

3.1 Line of Sight

The *line-of-sight* (LOS) relation may be defined in many ways and may or may not be symmetric. In this paper, for simplicity, we assume a symmetric LOS function and focus on the following three LOS definitions:

(1) 4-way LOS (LOS4): LOS4 exists between two cells p and q iff there is a path $\mathcal{P}$ from p to q such that all moves in $\mathcal{P}$ are the same cardinal move. (e.g., they are all East). The gray cells in Figure 1(b) represent LOS4 for the Green cell.

(2) 8-way LOS (LOS8): LOS8 exists between two cells p and q iff there is a path $\mathcal{P}$ from p to q such that all moves in $\mathcal{P}$ are the same cardinal or diagonal move. (e.g., they are all North East). Figure 1(c) 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 is perhaps the most suitable LOS function that discretizes real-world continuous domains and simulates a continuous field of view. It allows to see more cells beyond the cardinal and diagonal lines. The exact definition of *BresLos* is complex and is given in (Bresenham 1965). Figure 1(d) shows *BresLOS* for the Green cell.

Sometimes, there is an upper-bound LOS radius R such that cells at distance larger than R are not seen. For simplicity we assume ($R = \infty$). But, all our algorithms are applicable to any radius R. In addition, for every cell c we use $LOS(c)$ to denote the set of all the cells that have LOS to c. We say that an agent located at cell p can *see* cell q iff p and q have LOS which is equivalent to the condition that $q \in LOS(p)$ and vice versa.

4 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.

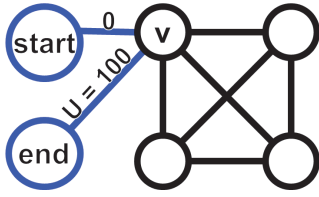


Figure 2: The graph G'. The new vertices are colored Blue.

Definition 1. *In NC-TSP we are given a graph G and a* $start$ *state. The task is to find a path from* $start$ *that passes through all vertices in G at least once.*

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

Proof. We reduce TSP (assuming that a node can be visited multiple times) to NC-TSP. Assume an input graph $G = (V, E)$. We build a graph $G' = (V', E')$. All vertices and edges in G also appear in G' but we add two more vertices, $start$ and end. We pick an arbitrary vertex $v \in V$ and connect $start$ to v with a zero cost edge. We also connect v to end with an edge with cost U, where U is a constant that is larger than the sum of all edges in E. Figure 2 shows the new graph G'. Now, observe that an optimal path for NC-TSP passes through the entire set of vertices V and halts at end, while passing through edge (v, end) (with cost U) only once. Otherwise, it will have to pass through edge (v, end) twice incurring a cost of at least $2U$. Next, it is easy to see that a solution to NC-TSP from $start$ on G' is optimal iff it includes a shortest cycle in G. □

We next reduce NC-TSP to our version of WRP (where there is a specific start state and the path ends once all cells in $\mathcal{C}$ 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 there is LOS between $c_1 \in \mathcal{C}$ and $c_2 \in \mathcal{C}$ iff $c_1 \equiv c_2$. That is, a cell can only see itself (LOS radius of 0). Thus, one must pass through the entire set of cells $\mathcal{C}$. □

5 WRP as a Search Problem

We next define the search tree of WRP followed by introducing admissible heuristics.

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* ∪ *unseen* = $\mathcal{C}$).[1]

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

[1]We note that the path associated with a given node S is not part of the description of the node, and is only a result of the current branch of S in the search tree. This enables to prune duplicate nodes which have the same location and the same *seen* list as they represent exactly the same situation for the search task.

Expansion: Expanding node $S = \langle location, seen \rangle$ is to perform all legal movements on $S.location$. Each child S' of S is associated with a given legal movement. $S'.location$ is the neighboring 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. In other words, $S'.seen = S.seen \cup LOS(S'.location)$. The cost of the edge from S to S' is the cost of the movement action.

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. $S.seen$ includes all the cells that have LOS to at least one of the locations in the path associated S. The cost of node S in the tree 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 *Open* to denote the Open list of the A* search that is activated on this search tree.

5.1 Preprocessing

All our algorithms relay on the following two lookup tables that are generated in a preprocessing phase and can be looked up during the A* phase. **(1) LOS(c):** For each cell c this table contains a list of all cells in $LOS(c)$. **(2) All Pairs Shortest Path (APSP):** For each pair of cells we store the minimal distance between them. While these lookup tables can be fully built in a preprocessing phase, in our implementation they were built lazily, on demand. 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.

A* relies on an admissible heuristic to return optimal solutions. We introduce such heuristics next.

6 Singleton Heuristic

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

$$h_p(S) = \min_{q \in LOS(p)} d(S.location, q)$$

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

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 et al. 2006; Tolpin et al. 2013). 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.

CPU 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:

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

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

7 Graph Heuristics

Our next two heuristics are based on a graph called the *Disjoint LOS graph* ($G_{DLS} = (V, E)$). G_{DLS} is abstracted from the grid map M for every node S in *Open*. The following notations are used in this context. We say that two cells $x, y \in \mathcal{C}$ are LOS-$disjoint$ if $LOS(x) \cap LOS(y) = \emptyset$, i.e., there is no cell that they both see. In general, we say that a set of cells LD is LOS-$disjoint$ if every two cells in LD are LOS-$disjoint$ (i.e., $\bigcap_{c \in LD} LOS(c) = \emptyset$).

7.1 Vertices of G_{DLS}

Figure 3(a) presents a grid M where the agent is located in the *start* cell D and LOS4 is used. The root node S is generated with $S.location = D$ and $S.seen$ includes the cells in the grid which are horizontally right or left of D or vertically above D (all the gray cells as well as A). $G_{DLS}(S)$, shown in Figure 3(b), is built as follows. The set of vertices of $G_{DLS}(S)$ is created from a subset of cells from M as defined below. These vertices are classified into three types, each with a different color (Green, Red and Yellow):

- **Agent Cell.** The *AgentCell* (D – colored Green) is associated with $S.location$.
- **Pivots.** The *Pivots* vertices (C and F – colored Red) are a LOS-$disjoint$ set of cells from $S.unseen$. Importantly, there are many possible way to choose the set of *Pivots*, and thus $G_{DLS}(S)$ can be built in many ways. We describe our own method below in Section 7.5.
- **Watchers.** The *Watchers* (A, B, E, H and I – 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.

Note that some cells in M do not have vertices in $G_{DLS}(S)$ and are omitted (the gray cells which are in $S.seen$ and the white cells which are in $S.unseen$). A pivot p and its watchers are denoted as *component* p of $G_{DLS}(S)$. Similarly, *AgentCell* (without its LOS) is its own component. There are three components circled in Figure 3(b): one for each of the pivots C, F and one for *AgentCell*.

7.2 Edges of G_{DLS}

First, all edges $(u, v) \in M$ where both u and v exist in G_{DLS} are projected onto G_{DLS}, e.g., the edge (A, B). Next we iterate over all cells that were omitted from G_{DLS} and *contract away* (Geisberger et al. 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 D is contracted away and D is connected to A in $G_{DLS}(S)$. Edges in $G_{DLS}(S)$ can be classified into two types. Their cost represent how much the agent needs to travel to see the pivots:

- **Watching edges.** These are edges inside a component that connect watcher vertices to their pivot in $G_{DLS}(S)$. Such edges have a cost of 0 (e.g., $c(A, B) = 0$).
- **Traveling edges.** These are inter-component edges. The cost of these edges is the APSP distance between the two vertices (e.g., $c(A, D) = 2$, $c(D, E) = 3$, $c(D, I) = 4$).

7.3 Attributes of G_{DLS}

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

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 M 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.*

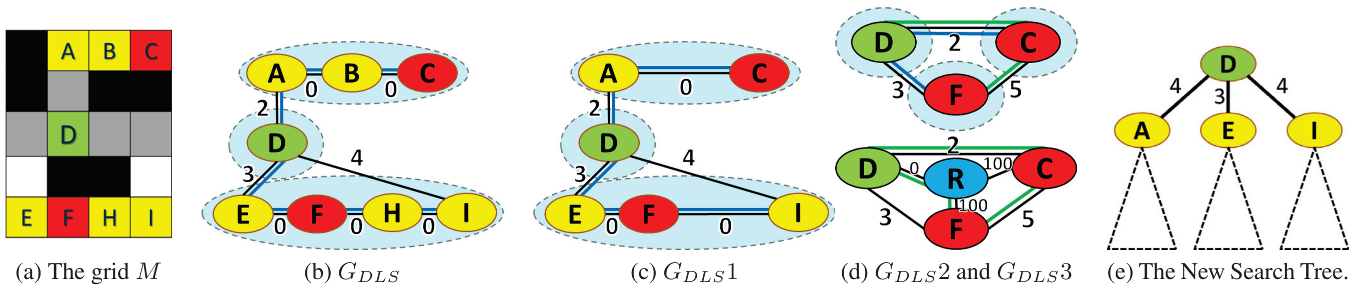
Proof. This is a direct result of Observation 1, lemma 3 and the fact that the pivots are LOS-$disjoint$. It is only a lower bound because $G_{DLS}(S)$ may not cover all the cells in $\mathcal{C}$ as some cells are left out. Also, it assumes that traveling within a component has a cost of zero but this cost might be larger in practice. □

The heuristics described below are all computing different lower bounds to such pivot-covering paths.

7.4 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 procedure.

Contracting away internal watchers. Consider cells A and B. 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* are watchers that have neighbours in M



(a) The grid M (b) G_{DLS} (c) $G_{DLS}1$ (d) $G_{DLS}2$ and $G_{DLS}3$ (e) The New Search Tree.

Figure 3: Example of a grid with LOS4, derived G_{DLS} graphs and the advanced search tree.

outside of the component. *Internal watchers* 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 3(c)), where cells B and H are internal watchers and are contracted away.

7.5 Choosing Pivot Vertices

We now turn to discuss how pivots are chosen. We say that a set L of $LOS\text{-}disjoint$ pivots is *maximal* if no other cell is $LOS\text{-}disjoint$ 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* $LOS\text{-}disjoint$ set. We used the following greedy method to obtain a *maximal* $LOS\text{-}disjoint$ set. We first set *Pivots* $= \emptyset$. We then iterate over all cells $c \in S.Unseen$. If c is $LOS\text{-}disjoint$ 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 $|LOS(c)|$ 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* $LOS\text{-}disjoint$ set. We tried other ways of choosing 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 $LOS\text{-}disjoint$ pivots can also dynamically change and even increase. For example, consider a cell d that was not $LOS\text{-}disjoint$ to all pivots at a given node S. d might be $LOS\text{-}disjoint$ to all pivots in its child C. This happens if there exists a cell e and a pivot p such that $e \in LOS(p)$, $e \in LOS(d)$ and $e \in S.unseen$. In the child C, e is added to $C.seen$ and is removed from the component of p. This makes d $LOS\text{-}disjoint$ to all pivots and d can be added to the set of pivots. The following steps summarize how $G_{DLS}(S)$ is built.

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

7.6 MST Heuristic

The *Minimum Spanning Tree* (MST) of a graph is the spanning tree with the minimal sum of edge costs. The MST heuristic $h_{MST}(S)$ computes a MST of $G_{DLS}1(S)$. In Figure 3(c) 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 that connects *AgentCell* to all the *Pivots*. Since the components are 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 (Cormen et al. 2009) then h_{MST} is fast to compute.

7.7 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 *AgentCell* and passes through all the components. To do this we further abstract $G_{DLS}1$ to $G_{DLS}2$. $G_{DLS}2$ is a *homomorphic abstraction* of $G_{DLS}1$ where all vertices within a component in $G_{DLS}1$ are merged into a single vertex in $G_{DLS}2$. These new vertices are associated either with the corresponding pivot or with *AgentCell*. $G_{DLS}2$ for our example is illustrated in Figure 3(d.top). It has three vertices: D, C and F. $G_{DLS}2$ is a complete graph (clique). The cost of an edge between vertices in $G_{DLS}2$ is the minimal cost among all edges that connect these components in $G_{DLS}1$.

We note that the MST of $G_{DLS}2$ includes exactly the non-zero edges of the MST of $G_{DLS}1$. Thus, in order to compute h_{MST} one can first generate $G_{DLS}2$ and then execute MST on $G_{DLS}2$. However, generating $G_{DLS}2$ requires to iterate over all edges of $G_{DLS}1$, 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 *AgentCell* and visits all other vertices in $G_{DLS}2$. 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 state. We therefore slightly modify $G_{DLS}2$ to yet a new graph $G_{DLS}3$ so that we can exploit TSP solvers. $G_{DLS}3$ adds a single *reference vertex* (denoted R) to $G_{DLS}2$. R is connected by edges to all other vertices in the graph. The cost of 0 is given to the edge connecting R to *AgentCell* while all other edges have a cost of U. U is a constant larger than the sum of all edges in $G_{DLS}2$. $G_{DLS}3$ for our example is shown in Figure 3(d.bottom).

Observe that a traveling salesman tour for $G_{DLS}3$ must include the zero edge between R and *AgentCell* (it is $\{R - D - F - C - R\}$ in our example). Furthermore, if we remove R and its two incident edges from the tour then we are left with the *minimal-cost Hamiltonian path* whose one end is *AgentCell* (this path is $\{D - F - C\}$). The cost of this path is an admissible heuristic for our problem denoted by h_{TSP} (solve TSP on $G_{DLS}3$ and remove R). Unlike MST, TSP is NP-hard (Held and Karp 1970). By contrast h_{TSP} is more informed than h_{MST}. In our example (Figure 3(e.top) the MST is highlighted in Blue (2+3=5) while the minimal Hamiltonian path associated with h_{TSP} is highlighted in Green (2+5=7). Thus, there exists a natural trade-off of accuracy vs. time to compute the heuristic.

8 Reducing the Size of the Search Tree

Trivially, when node S is expanded, then new nodes are generated for all the cells that are neighbours of $S.location$. However, based on $G_{DLS}1$ (Figure 3(c)) we can significantly reduce the size of the search tree. Given a node S and its corresponding $G_{DLS}1(S)$ consider any watchman path π that will continue further from $S.location$. Such a path must include at least one watcher for each component. We are interested in the first such watcher. All such possible watchers are direct neighbors of *AgentCell* in $G_{DLS}1(S)$ as shown in Figure3(b-c). The optimal watchman path will include one of them as the first such watcher. To have a complete search (and not lose any possible path) we must add all of them as neighbours of S in the search tree. Formally, when expanding a node S, we generate one child C for each edge in $G_{DLS}1(S)$ that connects *AgentCell* to a watcher W as follows. $C.location = W$. Let $\pi(AgentCell, W)$ be the shortest path in M from *AgentCell* to W (taken from the APSP lookup table). The cost of edge (S, C) in the search tree is set to cost of $\pi(AgentCell, W)$ (=the cost of the corresponding edge in $G_{DLS}1(S)$). $C.seen$ is updated to also include $LOS(p)$ for each cell $p \in \pi(AgentCell, W)$.

Note that this method took care of all components in $G_{DLS}1(S)$, i.e., Red, Yellow and Green cells in the map of Figure 3(a). We are left with the Gray and White cells. Gray cells are trivially covered because they are all in $S.Seen$. But, we need to take care of the possibility that the optimal path passes via a White cell. To do so, we also build (temporary) components for the White cells. We iterate over all White cells. Given a White cell we temporarily add it as a pivot (Red) and add all the other White cells with LOS to it as watchers. We then (temporarily) add this component to $G_{DLS}1(S)$ in the same manner as real pivots. We continue this process until all White cells are taken care of. We then add the first watchers of the White cells as children of S. Importantly, these temporary components are not considered by the graph heuristics (MST and TSP) because their pivots are not LOS-$disjoint$.

This method is called the *Jump to Frontier* enhancement (JF). JF is inspired by Jump Point Search (JPS) (Harabor et al. 2019) — a framework that implements A* on grids.[2] Expanding the root node in our example will result with the tree in Figure 3(e).

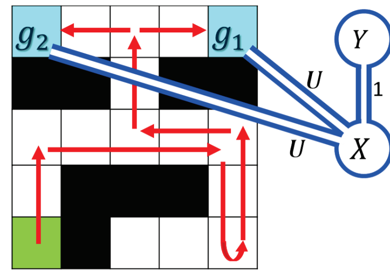


Figure 4: Multiple Goals Example

9 Specific Goal States

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

We next show how to modify the grid map M to a new map M' such that a solution to the implicit goal case for M' is also a solution to the *explicit goal* case for M. M' is built as follows. (Figure 4 shows how our example graph from Figure 1 is modified where the top corners are the explicit goals.) For each explicit goal (one or more) we add an edge of cost U (where U is a very large constant) to a new cell X. X is then connected to another cell Y such that X is the only cell that has LOS to Y.

Lemma 6. *An optimal path for the* implicit case *for M' must end at X via one of the U-cost edges.*

Proof. Any WRP path must pass through X because X is the only cell 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. □

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

[2]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 placed in the open list. Other cells (e.g., those along a cardinal line) are just passed through.

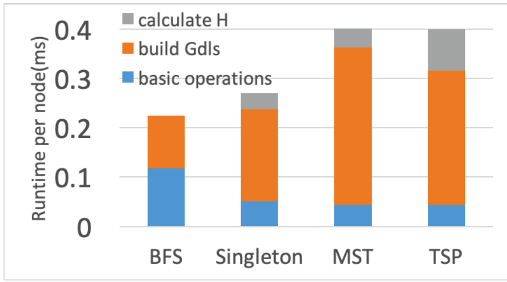


Figure 5: Runtime per node generated using Jump to Front

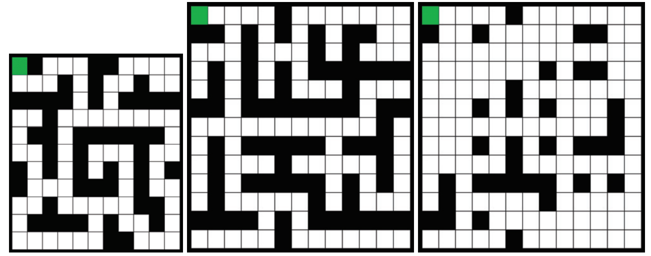


Figure 6: **Left:** 11x11 maze. **Center:** Full 13x13 maze. **Right:** 13x13 maze after removing 50% of the obstacles.

10 Experimental Results

We have performed extensive experiments on many maps and report our findings here on representative example maps. Indeed, each of our improvements achieved a significant reduction 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.

10.1 Constant Time Per Node Breakdown

Figure 5 shows a breakdown to three categories of the runtime per generated node on a large map for all algorithms; JF was added on top. The *basic operations* category includes all the BFS overhead (e.g. operations on *Open*) as well as updating the *seen* and *unseen list*. BFS consumed more time for the basic operations because *Open* was larger. Also, JF caused it to jump to farther locations and updating the *seen* and *unseen* lists took more time. Note that for MST and TSP the *build* G_{DLS} category (which included simplifying it to $G_{DLS}1$) costs 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.

10.2 11x11 Maze Grid

We next experiment on an 11x11 maze map shown in Figure 6 (left) with a specific start state (the Green cell in the middle of the top row). We picked this relatively small domain so as to be able to provide a full comparison between all methods and all LOS functions as 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) (right) for each of our LOS functions. Rows correspond to the different algorithms. 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

		Basic Expansion		Jump to Front	
LOS	h	**Nodes**	**Time**	**Nodes**	**Time**
LOS4 OPT=73	BFS	666,370	27,670	28,485	7,499
	Singleton	296,264	19,166	13,980	3,772
	MST	1,256	1,363	213	276
	TSP	350	617	79	57
LOS8 OPT=64	BFS	467,079	19,842	6,226	1,158
	Singleton	152,362	6,878	3,016	761
	MST	1,122	509	119	203
	TSP	507	412	75	59
BresLos OPT=57	BFS	149,450	6,345	2,316	503
	Singleton	61,959	2,639	1,175	246
	MST	921	143	93	44
	TSP	293	75	35	18

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

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 $\sim$1000 for nodes and $\sim$100 for CPU time. Finally, the right columns show the results when JF was used. 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.

10.3 Increasing Density of Obstacles

Figure 6 (center) 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 6 (right) shows the half way point where 35 obstacles were present. Figure 7 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 bottom frame presents the number of nodes expanded for BFS and for our three heuristics, while the top 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 an easy-hard-easy behaviour; after 65 obstacles

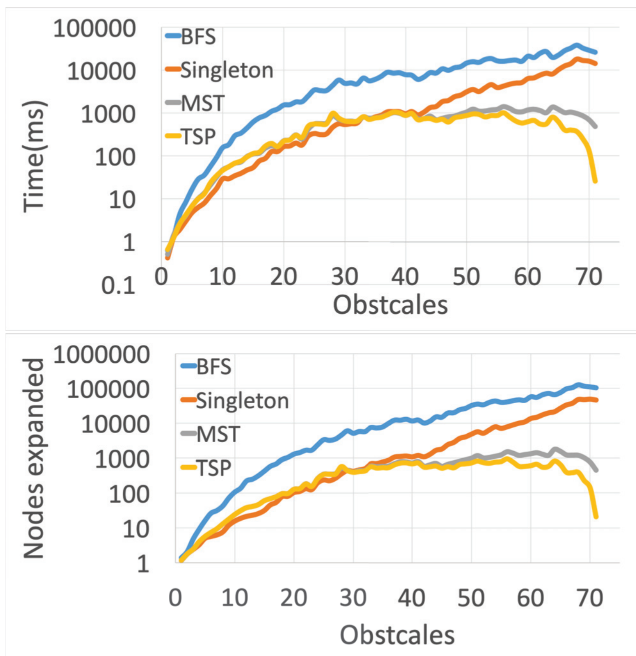
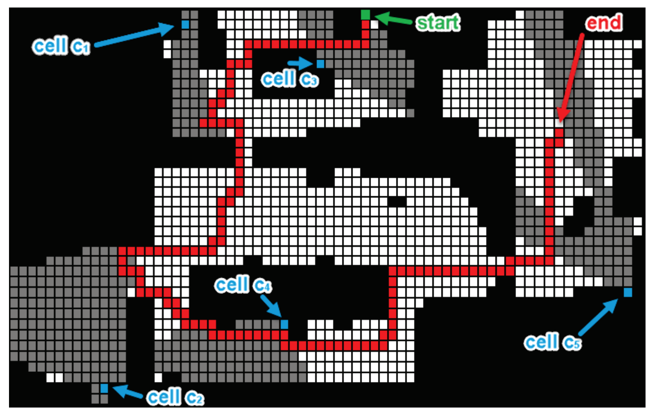


Figure 7: Varying the number of obstacles



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	>7,864	>28,500	>3,000
TSP	137	5,096	15,476	471
		Suboptimal solvers		
BFS	101	>9,334	>46,214	>432
Singleton	137	4,642	19,338	94
MST	137	723	4,043	34
TSP	137	521	2,962	40

Figure 8: The den101d map and its experimental results

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. 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.

10.4 Larger Map

We also experimented with the *den101d* map from the *movingai* repository (Sturtevant 2012) shown in Figure 8. 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 state is marked in Green (top 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 algorithms on this map (with JF and *BresLos*). BFS, Singleton and MST exhausted the available memory of 2GB and a failure was reported. The table in Figure 8 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 solved the problem optimally (OPT=137) but the other variants reported failure after a few minutes while only reaching f-values of 66, 103, and 137 for BFS, Singleton and MST, respectively.

10.5 Suboptimal Variants

We have also built a suboptimal variant of our algorithm which further reduced the branching factor of the JF step as follows. As JF described above jumped to the watchers cells (Yellow cells) but also created new temporary components for the White cells and then jumped to them too thus increasing the branching factor. Here, we omitted the White cells and only jumped to Yellow cells. This reduced the branching factor but loses optimality in very rare cases.

The results of the suboptimal variants on *den101d* are presented in the bottom of the table in Figure 8. BFS exhausted the resources after reaching layer 101. However, a solution was found by all other variants very fast, around 30 seconds for MST and TSP and 1.5 minutes for Singleton. This is a reduction in time of an order of magnitude compared to the optimal variant. This map is sprase with obstacles and we were lucky to get an optimal solution (137), but there was no guarantee on that in general. A deeper treatment of suboptimal variants is left to future work.

11 Conclusions and Future Work

In this paper we solved the *Watchman Route Problem* with heuristic search and provided strong heuristics for this problem. We feel that we have touched the surface of an iceberg regarding this fascinating problem. Future work will continue in the following directions. First, new LOS functions should be studies such as limiting the radius as well as incurring real costs for physically observing at the environment. Second, suboptimal, bounded suboptimal and anytime algorithms for this problem should be built, e.g., by modifying the algorithms presented in this paper. Finally, a study will be done for generalizing this problem to the case where multiple watchman agents exist and for different communication paradigms between them.

12 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. We deeply thank Shahaf Shperberg for his comments and his help.

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.

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

Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; and Stein, C. 2009. *Introduction to algorithms*. MIT press.

Dissanayake, M. G.; Newman, P.; Clark, S.; Durrant-Whyte, H. F.; and 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.; 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.

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.

Geisberger, R.; Sanders, P.; Schultes, D.; and Delling, D. 2008. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *Experimental Algorithms, 7th International Workshop, WEA*, 319–333.

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.

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

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

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

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

Rivera, N.; Hernández, C.; and Baier, J. A. 2017. Grid pathfinding on the 2*k* neighborhoods. In Singh, S. P., and Markovitch, S., eds., *AAAI*, 891–897.

Sturtevant, N. R. 2012. Benchmarks for grid-based pathfinding. *IEEE Trans. Comput. Intellig. and AI in Games* 4(2):144–148.

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.

Tolpin, D.; Beja, T.; Shimony, S. E.; Felner, A.; and Karpas, E. 2013. Toward rational deployment of multiple heuristics in A. In Rossi, F., ed., *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, 674–680. IJCAI/AAAI.

Xu, N. 2014. On the watchman route problem and its related problems. *Dissertation Proposal*.