A Report on Optimal Path Finding and Performance Enhancements on 2D Lattices using A*

1st Christoff van Zyl
Honors Computer Science
Stellenbosch University
Stellenbosch, South Africa
20072015@sun.ac.za

I. Functionality Checklist

Core:

- Reading in a lattice using a grid map of ones and zeros.
- Reporting mechanisms integrated into algorithms.
- Depth-First Iterative Deepening.
- A* , Iterative Deepening A* and Bi-Directional A*.
- Various heuristics, including Dijkstra.

Extended:

- Creating a lattice programmatically. This implies infinite lattices and non-rectangular lattices.
- Visualization mechanism integrated into algorithms.
- Optimizations on positional comparison.
- Optimal path guarantee for admissible heuristics.
- Multiple start and end points.

II. Definitions

- Position : A two-dimensional position on the lattice.
- Path length ($g(n)$) : A function representing the distance which was travelled on the lattice to reach node n.
- Heuristic function ($h(n)$) : A function representing an estimate of the distance a node n will need to travel still in order to reach a goal position.
- Cost Function ($f(n)$) : A function representing an estimate of the total path length to the goal. Mathematically: $f(n) = g(n) + h(n)$.
- Node (n) : An element inside the search space. It is uniquely described by a position, a current path length ($g(n)$), a heuristic function value ($h(n)$) and parent node (n_p).
- Open Set (**OPEN**) : The ordered set of nodes representing the search frontier or open set.
- Closed Set (**CLOSED**) : The set of nodes representing all explored nodes.
- Expand : The act of moving a node from **OPEN** to **CLOSED** and adding all of its valid children to **OPEN**.
- Dijkstra Heuristic: Refers to the uniform cost heuristic that allows A* to function in a similar manner as a Dijkstra graph search, with $h(n) = 0$.
- Straight Line Distance (SLD) Heuristic : This heuristic returns the Euclidean distance between the current position and the goal (k) , hence:

$$h(n) = \sqrt{(k_x - n_x)^2 + (k_y - n_y)^2}$$

- Manhattan Heuristic : This heuristic returns the sum of the difference in the x and y directions:

$$h(n) = abs(k_x - n_x) + abs(k_y - n_y)$$

III. Implementation and Performance Enhancements

A. Language and Libraries

Java was used as the language of implementation. Furthermore an open source OpenSimplex noise library was used to create most of the problem sets.

B. Design Decisions

The algorithms described here were written in such a way as to guarantee optimal path finding and the report also evolves around optimal path finding. This implies that whenever a heuristic is mentioned, it can be assumed to be at least admissible. Furthermore many optimizations were done which relies heavily on the geometry of a 2-D lattice, such as backtracking always leading to worse paths, and that any path from a starting position to an end position is guaranteed to be an integer. Furthermore although the report does focus on small cases for iterative deepening, the algorithms were designed with scale in mind, thus large lattices will be used for the most part in order to investigate the general behaviour of the algorithms and heuristics on a large scale.

C. OPEN and CLOSED efficient implementation

For Graph searches that are not iterative deepening, literature on A* usually defines **OPEN** and **CLOSED** as two mutually exclusive sets, with nodes in **OPEN** adhering to the order: $n < n' \iff f(n) < f(n')$. **OPEN** is also guaranteed to contain only nodes that represent unique positions on the lattice. Whilst this works well in theory, programmatically it is difficult to keep a single ordered structure which allows ordering by one metric ($f(n)$) and searching by another (position) with both operations having $\mathcal{O}(log(|\mathbf{OPEN}|))$ worst case performance. One solution is to divide these expectations up by introducing $\mathbf{OPEN_1}$ that allows for searching with respect to node positions in $\mathcal{O}(1)$ time and $\mathbf{OPEN_2}$ that is ordered with respect to $f(n)$ and uses $\mathcal{O}(log(|\mathbf{OPEN_2}|))$ time for its operations. The following actions are then modified as described:

1) **Expanding a node n:** Remove the smallest node n from $\mathbf{OPEN_2}$ (removing it from $\mathbf{OPEN_1}$ as well via a search on n's position) and find all its valid children ($\mathbf{n'}$).
2) **Check the closed set :** Given a child node n' only continue to the next step if **CLOSED** does not contain $'n$.
3) **Inserting a new node n':** Check if $\mathbf{OPEN_1}$ contains a node at the same position, call it n_k if it exists. If it exists, and $f(n') < f(n_k)$, remove n_k from all three sets using appropriate searches. Finally add n' to all three sets using appropriate searches.

This guarantees $\mathcal{O}(log(|\mathbf{OPEN_2}|))$ worst case time complexity during the process of expanding any node. For the current implementation $\mathbf{OPEN_1}$ and **CLOSED** are implemented as Hash Tables and $\mathbf{OPEN_2}$ as Linked Lists for iterative deepening and Tree Sets otherwise.

D. Optimal Path Guarantees

Given an admissible heuristic $h(n)$, which is not consistent, it is possible for the heuristic to return non-optimal paths in a graph search, as can be seen from Figure 8 on the left. Here the heuristic uses Manhattan distances or Uniform Cost depending which path is taken. Two steps can be taken to amend this. Firstly let $\mathbf{SEEN} = \mathbf{CLOSED} \cup \mathbf{OPEN_2}$ and replace all actions involving either **CLOSED** or $\mathbf{OPEN_2}$ with **SEEN** in III-C,

making the second action redundant. For the second step, since $f(n)$ from the Uniform Cost path at the incident location is still smaller than that of the Manhattan path, we amend action three in III-C to compare using $g(n)$ instead of $f(n)$. This leads to the optimal path on as can be seen on the right in Figure 8. This does however prevent the algorithm from being a pure graph search for inconsistent heuristics and in such cases turns it into a hybrid between a graph search and a tree search. It is also important to note that this is not a general fix for inconsistent heuristics and relies heavily on the geometry of 2-D Lattices, namely that given two g values for the same position, the smaller one guarantees a shorter path to the goal if it exists.

E. Bi-Directional Refinement

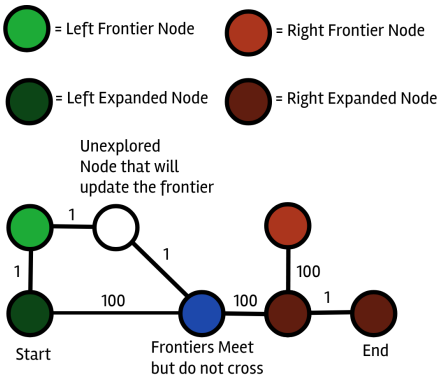


Fig. 1: Uniform cost heuristic leading to non-optimal path if the stopping condition is the minimum f in either frontier is less than current path length as opposed to both if the frontiers do not cross each other.

For Bi-Directional searches the stopping condition requires the minimum f value for a node in **both**, assuming we do not allow the frontiers to cross (see Fig 1) , the forward and backward frontiers be at least equal to the length of the current shortest path to the goal (assume an initial path length of ∞). Most literature suggests that both the forward and backward direction needs to keep being explored until the condition is met. Assuming that both directions are searched concurrently or alternately until the frontier meets for the first time, the condition can be met by having one frontier having a minimum f value at least equal to the path length, and the second frontier guaranteeing that it will not update its frontier nodes. This implies that all nodes in the second frontier represent the shortest possible paths to reach their respective positions. This can be accomplished by continuing to run both frontiers but prevent one of them from expanding, and only allow that one to refine its current frontier. This allows the non-expanding frontier to explore less, leading to an overall decrease in the amount of nodes we need to explore, see Table I. We see a median of about a 13% less nodes that requires expanding. It is also visualized in Fig 2 with the SLD (Straight Line Distance) heuristic (A full comparison that includes Dijkstra and Manhattan is found in Fig 7).

Algorithm	Nodes Explored	Nodes Explored (refined)
Manhattan	632513	545966
Dijkstra	6526889	4590687
SLD	1314865	1020939

TABLE I: Total nodes explored on in Bi-Directional search with and without the refinement mechanism.

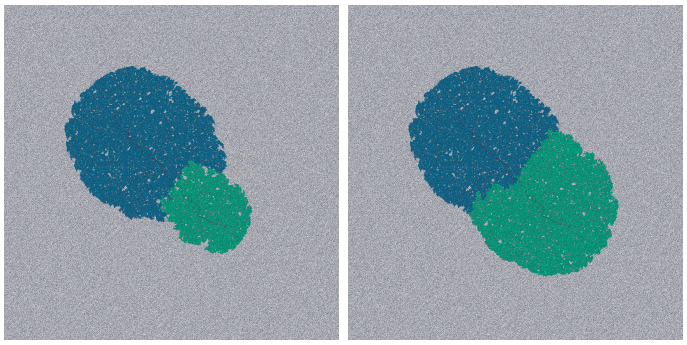


Fig. 2: Effects of refinement on Bi-Directional Search (left is refined, right is not).

IV. Results

A. Iterative Deepening Algorithms

Iterative deepening search, although having virtually no memory footprint, does suffer from an exponential growth in the nodes it explores with each depth, as can be seen from Table II. This forces heuristic functions to be extremely well calibrated to the problem at hand, in order to minimize the branching factor, whilst also retaining a large granularity to minimize the amount of unique depth levels in $f(n)$ that needs to be explored before the goal is reached. We can clearly see this effect by comparing Depth-First ID (Iterative Deepening) and A* Manhattan ID. Whilst the former has the smallest branching factor, it much more depths to be explored since it has $f(n) = g(n)$, as opposed to the latter. The results for A* SLD (Straight Line Distance) ID is also shown, which might cause confusion as it is an extremely granular heuristic. The reason it provides good results is due to another optimization which relies on the geometry of the problem. Given any non-integer admissible heuristic value, we can guarantee that the remaining distance to any goal is at least as big as its ceiling. Therefore, when comparing the $f(n)$ values of nodes to the current path length, as well as the current depth, and when computing the next depth, they use $\lceil f(n) \rceil$ instead. Nodes inside the frontiers are however still compared to each other without alterations.

Algorithm	Nodes Explored	Branching Factor	Depths Explored
Depth-First ID	304362523	1.76	34
A* SLD ID	22252	1.91	15
A* Manhattan ID	5517	7.83	5
A* Dijkstra	829	-	-

TABLE II: Total nodes explored and branching factor on an easy noise generated lattice (see Fig 6) by different algorithms and heuristics.

It is clear that even for small cases, such as this where the minimum path length is only 34, that poor heuristic functions explores several orders of magnitude more nodes in its iterative deepening version compared to its non-iterative deepening version or to more finely calibrated or dominating heuristics. The prime example of this here is Depth-First ID compared to A* Dijkstra, both of which use $h(n) = 0$. Furthermore to show iterative deepening algorithms' lack of scalability due to their exponential growth in nodes explored each depth, the dominating heuristic was used on the same maze but with the goal shifted a bit further away such that the minimum path length is 65. The results are tabulated in Table III.

Algorithm	Nodes Explored
A* Manhattan ID	140560035
A* Dijkstra	3547
A* SLD	597
A* Manhattan	282

TABLE III: Total nodes explored on a medium lattice.

B. A* vs Bi-Directional A* for Optimal Path Finding

Algorithm	Nodes Explored
BD A* Dijkstra	4253180
A* Dijkstra	3734959
BD A* SLD	938376
A* SLD	851926
BD A* Manhattan	491525
A* Manhattan	470850

TABLE IV: Total nodes explored on a large lattice.

Table IV represents the total amount of nodes explored by several algorithms on a large lattice. It is clear that Bi-Directional A* does not provide a performance increase over A* on these types of lattices. The reason for this appears to lie with the fact that the heuristics does not account for the obstacles in the lattice, thereby severely underestimating the distance to the goal for nodes far away from the goal, such that a significant amount of search still needs occur after the frontiers meet in order to guarantee optimality of the path. This allows for both the forward and backwards frontiers to be expanded in the wrong directions and still provide f values lower than the current path length. This effect can be observed in Fig 3. Notice how the forward frontiers have similar explored area, however, the backwards frontier searches in the wrong direction as well. This leads to the slight increase in nodes explored by similar heuristics in their Bi-Directional versions.

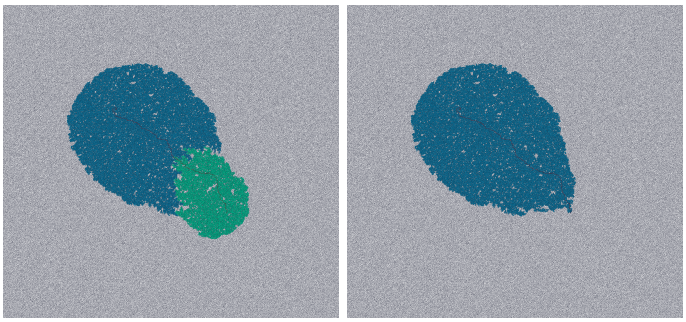
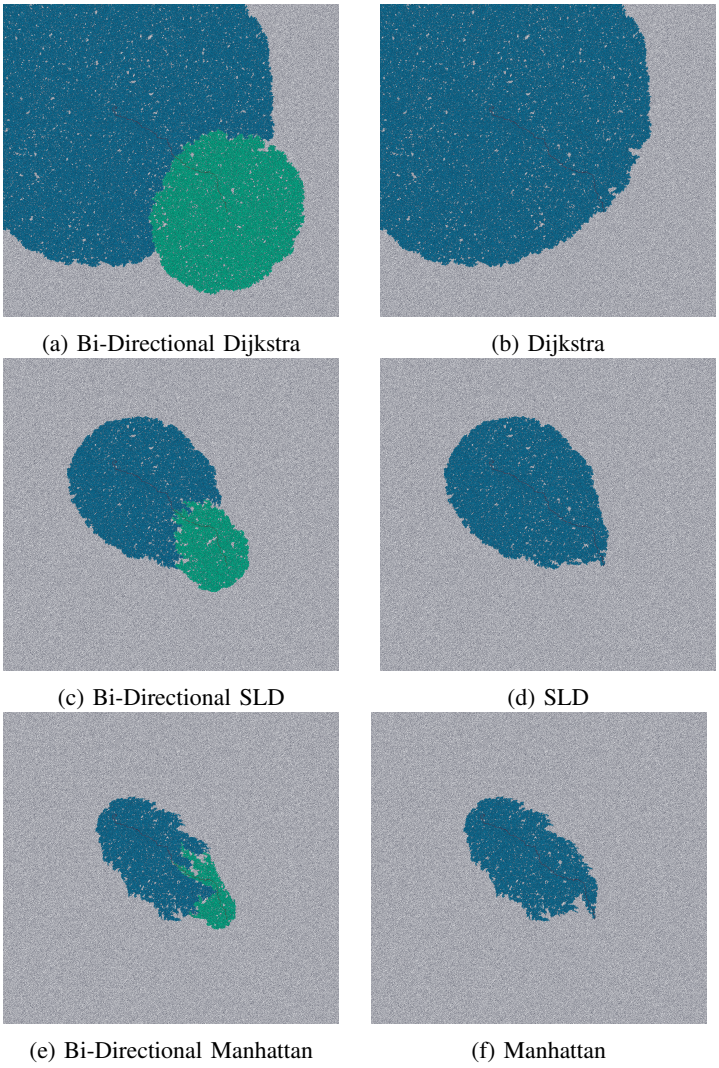


Fig. 3: Bi-Directional A* expansion (left) vs A* expansion (right)

C. Heuristic Comparison

By looking at Tables II and IV once more it is easy to determine a clear order in the effectiveness of the three heuristics. Dijkstra, which makes no attempt at estimating the distance to the goal, performs worst overall, with the straight line distance being the intermediate heuristic and the Manhattan distance heuristic performing best overall. This does not prove, but strengthens the general notion that if one heuristic dominates another, one can in general expect better performance from the dominating heuristic.

Using Fig 4 it is easy to see these effects in action. Starting with Dijkstra, one can clearly see the effect of the heuristic not respecting moving towards the general direction of the goal, and due to this does not direct the search towards the goal.The straight line distance heuristic already significantly improves on Dijkstra by providing a general direction for the algorithm to search towards, however due to it severely underestimating the distance to the goal at positions diagonal to the goal, it allows the algorithm to search in these areas for too long before moving on to more favourable regions. The Manhattan distance heuristic overcomes this by taking the geometry of the problem into account (positions diagonal to the goal give large heuristic values) and is thereby able to create the tightest search of all three heuristics.



(a) Bi-Directional Dijkstra (b) Dijkstra

(c) Bi-Directional SLD (d) SLD

(e) Bi-Directional Manhattan (f) Manhattan

Fig. 4: Bi-Directional A* and A* large lattice exploration visualizations

D. Smart Heuristics

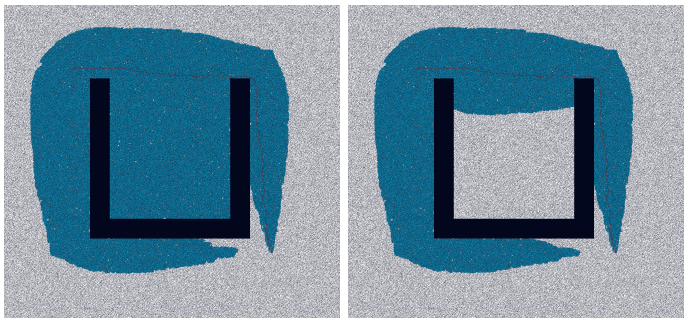


Fig. 5: Manhattan heuristic without (left) and with (right) environment specific knowledge

The heuristics dealt with thus far do not incorporate any information regarding the obstacles in the environment, relying only on the goal position to estimate the heuristic. This is useful since usually it is impossible to make guarantees regarding any region of a problem beforehand. If this is allowed however, and some, but not all information regarding a problem is known beforehand, it is possible to construct heuristics which incorporate this information. Suppose we know of a big obstacle in the lattice or environment, such as on the left of Fig 5. Here there exists a noisy lattice, and big bucket in the middle of the maze. Using both a normal Manhattan heuristic as well as one which is aware of the bucket, but not the noise of the lattice, it is able to significantly decrease the area it explores in order to find the goal. Approaches

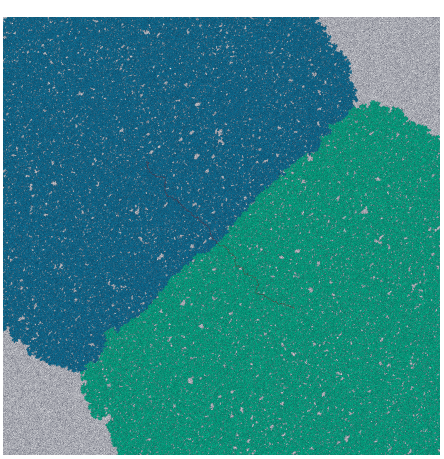
like these, which incorporate some large consistent features of the environment into the heuristic, whilst leaving the more granular inconsistent features up to the algorithm itself, can lead to a significant decrease in the amount of nodes expanded during the search.

APPENDIX A
FIGURES

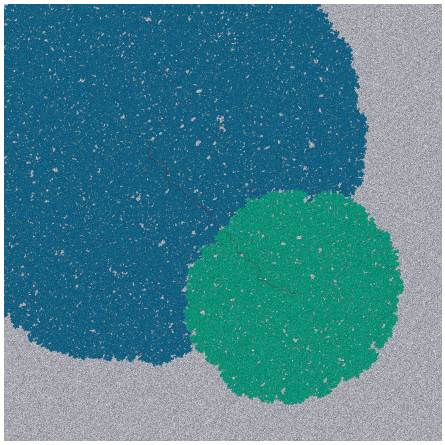
APPENDIX B
TABLES

(a) Dijkstra (Uniform Cost) (b) Manhattan (c) Bi-Directional Manhattan

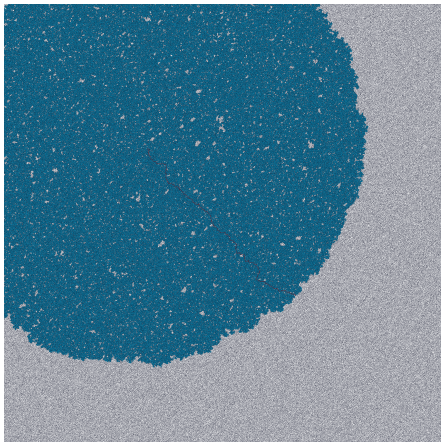
Fig. 6: Sparse Maze Search Results Visualization



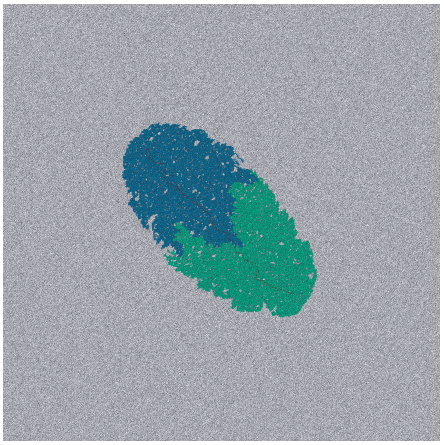
(a) Bi-Directional Dijkstra



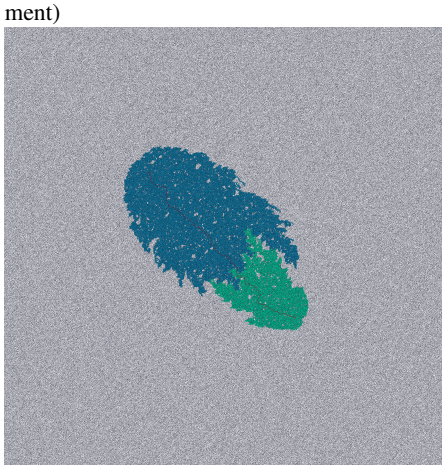
(b) Bi-Directional Dijkstra (Using Refinement)



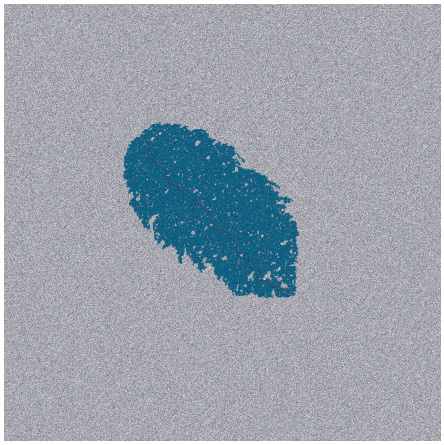
(c) Dijkstra



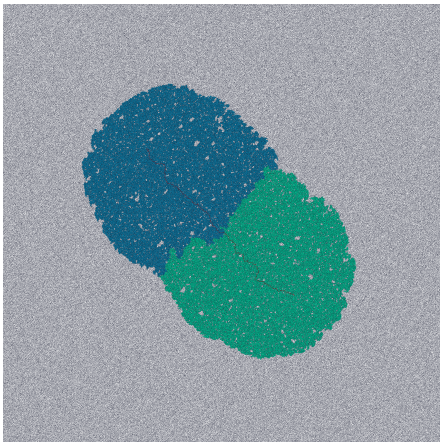
(d) Bi-Directional Manhattan



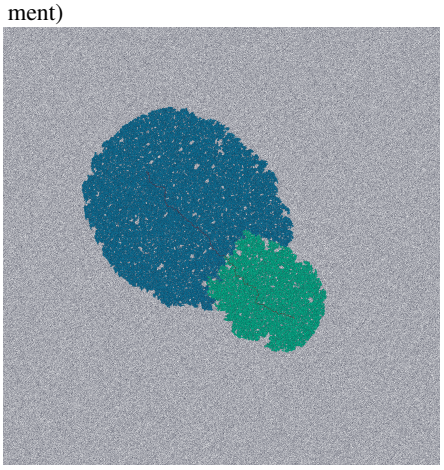
(e) Bi-Directional Manhattan (Using Refinement)



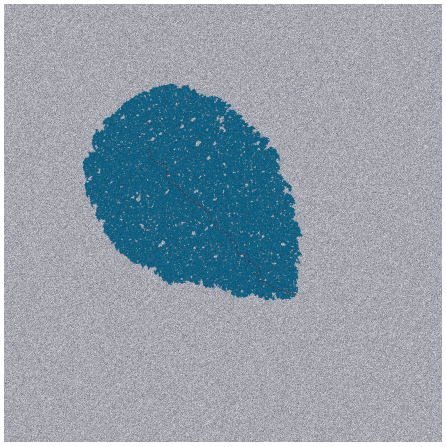
(f) Manhattan



(g) Bi-Directional SLD

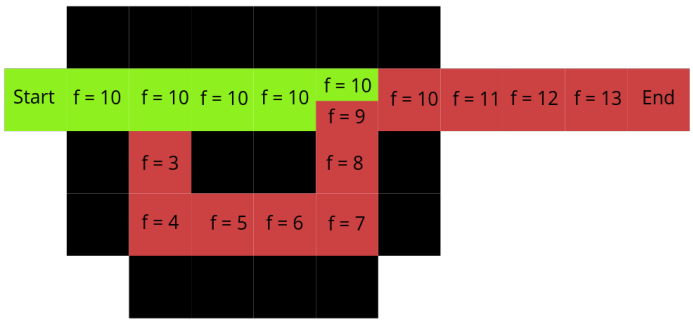


(h) Bi-Directional SLD (Using Refinement)

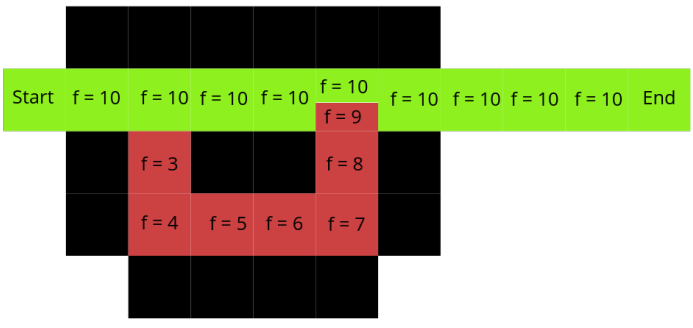


(i) SLD

Fig. 7: Large Dense Maze Search Results



(a) Without path correction



(b) With path correction

Fig. 8: Non-optimal path correction for inconsistent heuristics