COS30019
Introduction to AI

Assignment 1

RobotNav

Jeremy Vun 2726092

Lab: Monday 5:30pm

Table of Contents

1. RobotNav

The robot's navigation can be modelled as a general state space using tree search as the model for a model-goal based agent. However, the actual state space data structure is represented as a two-dimensional array[x][y] as opposed to a linked list for performance and simplicity. Successor states are dynamically calculated using a successor function i.e. Adjacent(node n) = walkable nodes north, west, south, and east. Pruning of redundant paths is the responsibility of the algorithm.

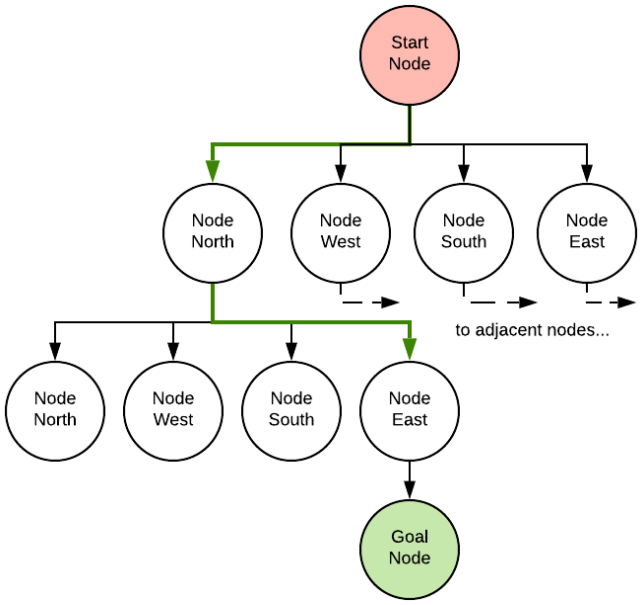


Figure 1.0 – tree representation of states, and the edges by which to reach them

Where there are multiple active goals, the agent behaves to choose the goal which it perceives to have the best value. Either explicitly using heuristics, or implicitly as baked into its reflex ruleset.

Using this model different search algorithms are applied to evaluate their completeness, optimality, and efficiency. Testing the relative performance of our different search algorithms, 6 standard maps (3 different map sizes with and without walls) are used. See Appendix A for map information.

f(x) = g(x) + h(x) is used as the performance metric to compare the value of different explored paths. Where f is the total cost of moving to node x, g is the realised cost, and h is the heuristic value. G cost is assumed to be 1 as the problem involves a uniform cost grid. This can be modified to account for different movement cost models as required.

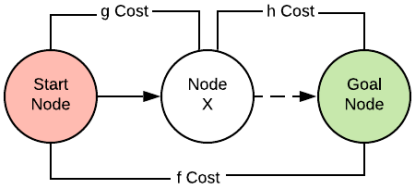


Figure 1.1 – g and h cost components of movement across state space

Importantly, we know that depth and branching factors are finite because the map is bounded (no cycles/looping) and each node leads to at most 4 other possible states (maximum b=4). This guarantees that our state space is finite and completely searchable if repeated states are ignored and the program is given sufficient time and memory.

Source Code can be found at: https://github.com/elodea/RobotNav

2. Uninformed Search

Uninformed searches do not make use of heuristic information about where the goal is. As such they are generally more expensive than informed searches. The cost of a path is represented as f(x) = g(x) where the heuristic component is set to 0.

While it is uncommon to use uninformed searches to solve pathfinding problems because knowledge of the goal's location is implicitly part of the problem (enabling the formulation of an admissible heuristic), uninformed searches are useful in situations where we don't know the location of the goal state or its heuristic distance.

2.1 Breadth First Search

2.1.1 Method

Breadth first search operates by completely exploring the frontier of possible nodes one step/edge away in each iteration. The shallowest nodes are fully explored before any deeper nodes. As shown in figure 2.1, because the goal is found on expansion of Node C, the algorithm stops and Nodes D, E, and F are left unexplored.

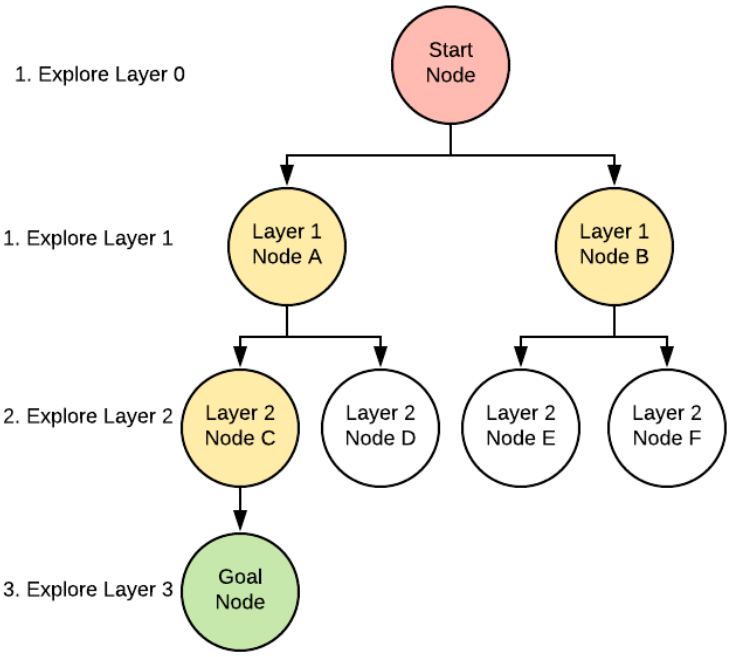


Figure 2.1.0 – Layer 0 is explored first, then layer 1, and finally layer 2 before we find out goal state as an adjacent node on layer 3. Nodes A, B, and C are explored. Nodes D, E, F are not explored.

Implementation involves using an open and closed set. The open set is first seeded with the starting node. Each node in the open set is then removed from the open set, put into the closed set to prevent looping, and inspected to find their adjacent nodes.

Successor/adjacent nodes are compared against the closed set to see if we have already seen them before to prevent looping. If they have not been seen before, they are given a g cost, attached as children to their parent node, and placed into the open set before we loop back. This continues until our open set is empty or we find the goal node.

If a the cost grid is non-uniform, BFS can be modified into a uniform cost grid search (dijkstra's algorithm), where the shallowness of a node and thereby it's priority for expansion is determined by its f cost. That is, breadth levels are determined by g cost.

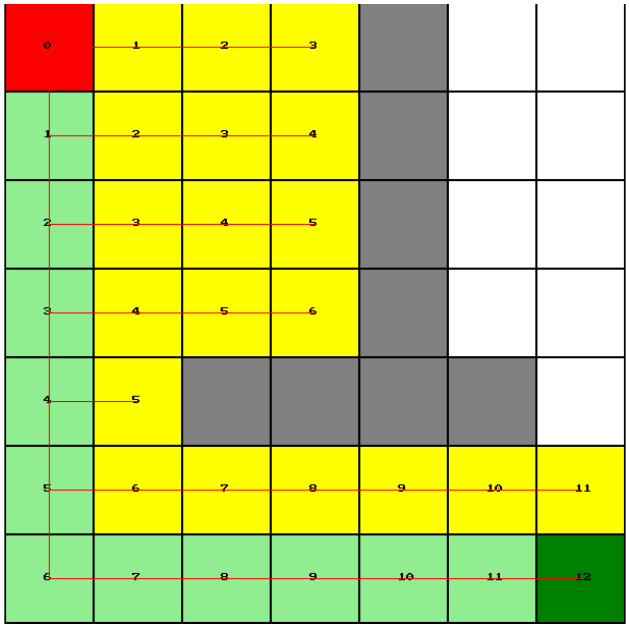


Figure 2.1.1 – BFS Search with each breadth layer indicated by increasing g costs.

2.1.2 Discussion

Completeness

If branching factor is finite, given enough time and memory breadth first search will find a path if one exists because it explores every single node/state until it finds a goal.

Optimality

Because explored nodes are assigned the lowest possible g cost from the start node, the path to any node n is guaranteed to be the shortest possible - see figure 2.1.1.

Efficiency

In the worst case BFS can end up evaluating every single possible state if the goal is at the maximum depth of the tree. It is also increasingly slow as the size of the state space increases as the number of nodes on the frontier increases with every iteration. BFS is particularly inefficient when multiple best paths exist because it effectively considers each of them (see jump point search 3.3). Performance increases as the number of obstacles increases because the result is less nodes to explore on the frontier at any time.

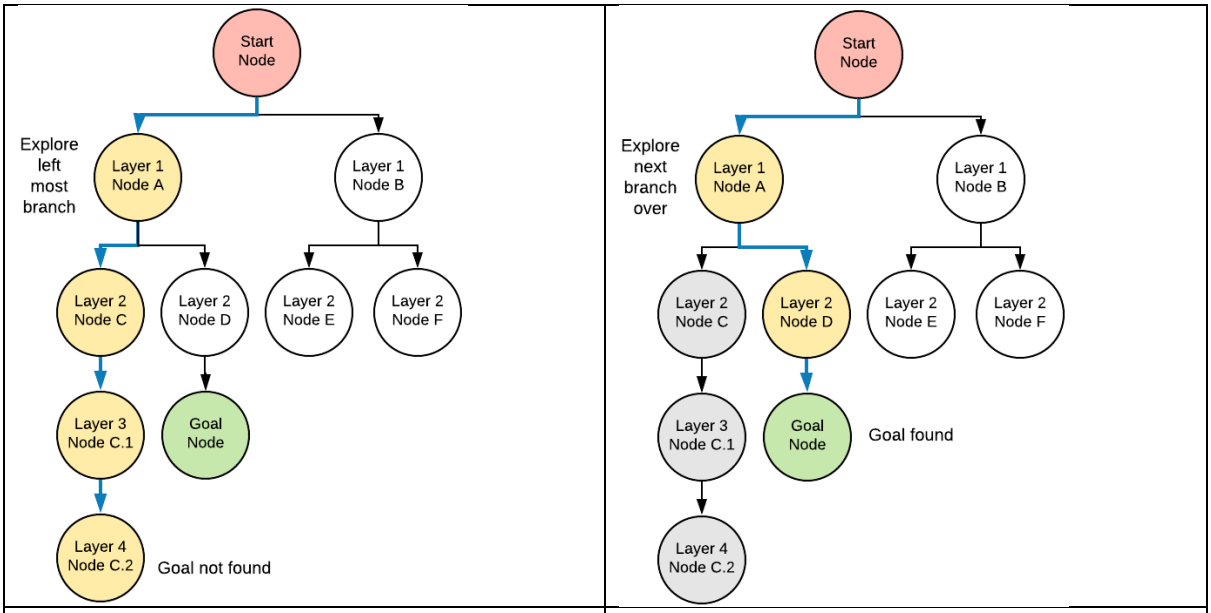
Breadth First Search (10 runs each)	
Map	**Average time (ms)**
Small – No walls	0.24
Medium – No walls	1.96
Large – No walls	6.68
Small – Walls	0.25
Medium – Walls	1.59
Large - Walls	5.20

Table 2.1.2 - BFS search time results

2.2 Depth First Search

2.2.1 Method

Depth first search preferences exploring nodes in deeper layers over nodes in shallower layers. When it reaches a dead end and cannot find any more unexplored successor nodes, it backtracks until it finds an open node in a shallower level that it did not previously explore.



Step 1: First explore left side in depth until Node C.2 - no more adjacent nodes are left to be explored.	Step 2: Path stack is unwound back to Node A and the next unexplored branch is chosen. Goal is found and search is stopped.

Figure 2.2.0 – graph representation of depth first search strategy

Depth first is traditionally implemented using a recursive function, but with a sufficiently large enough state space this can result in a stack overflow. As such, depth first search has been implemented iteratively with a simulated stack data structure.

2.2.2 Discussion

Completeness

Assuming finite depth and pruning of redundant paths (loops/cycles), depth first will find a path if one exists. In the worst case, every single branch and state will be explored before it finds a path.

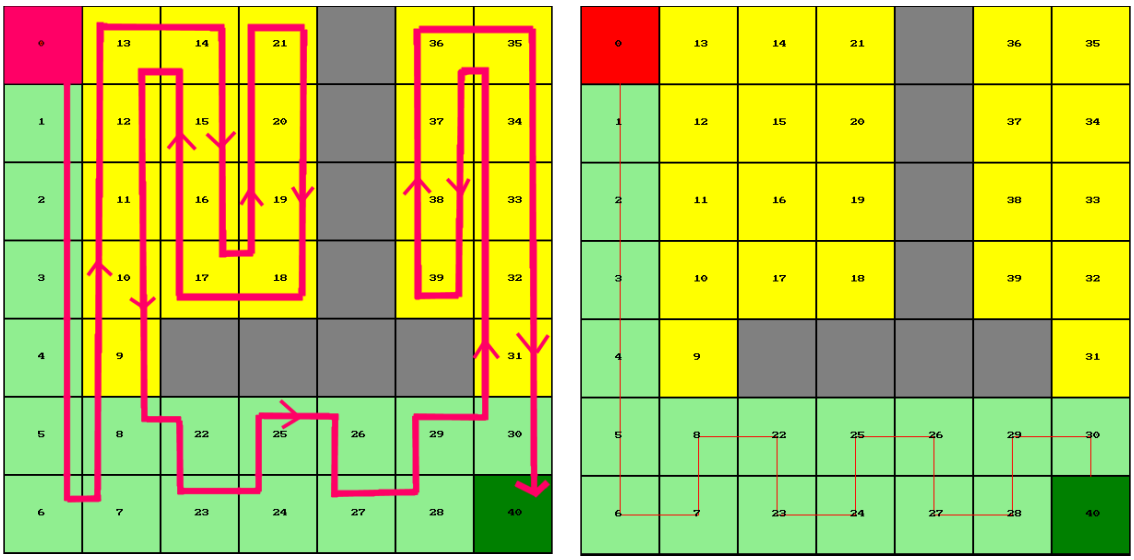


Figure 2.2.1 – DFS worst case explores every node before finding a suboptimal path to the goal state. Due to preferencing node exploration in the priority order ‘up, left, down, right’.

Optimality

Depth first does not guarantee finding the shortest path, only that we will find a path if one exists - see Figure 2.2.1.

Efficiency

DFS can be quicker than BFS if the goal happens to be in the branch/s that we choose to explore first and if the goal is deep. Depth first may be useful if we have an idea about where the goal is likely to be in the state space because we can explore those branches first.

10 runs each	**DFS**	**BFS**
Map	**Average time (ms)**	**Average time (ms)**
Small – No walls	0.30	0.24
Medium – No walls	0.62	1.96
Large – No walls	8.75	6.68
Small – Walls	0.27	0.25
Medium – Walls	1.54	1.59
Large - Walls	6.74	5.20

Table 2.2.2 – DFS search time results

2.3 Genetic Algorithm Search

2.3.1 Method

Genetic algorithms take inspiration from biological evolution and attempt to find solutions using the principles of variation, fitness selection, and reproduction. Instead of worrying about the specifics of a particular search instance, genetic algorithms provide feedback about the utility of each search in an attempt to provide gradual improvement over time.

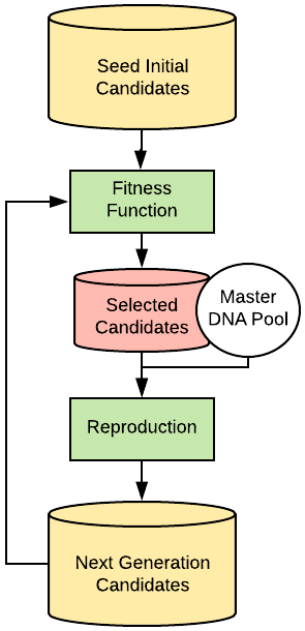


Figure 2.3.0 – Evolutionary operation of selection and reproduction over a pool of candidate solutions

Seeding

In robot navigation, the initial seed generation consists of a set of random paths. At each step, candidates randomly choose “N”, “W”, “S”, or “E” with equal 25% probability until they reach the goal state or the current iterative deepening limit. This generates a ‘dna sequence’ of cardinal directions e.g. “NESWWEESSN”. For efficiency, junk DNA is trimmed (see Reproduction).

Fitness selection

Candidates are passed through a fitness function to select the fittest 10%. Selected candidates are then mixed into a ‘master dna pool’ from which the next generation is built. The fitter a candidate, the greater the probability that their genetic information will be passed on.

It is important that the fitness function be continuous. Boolean fitness functions are inefficient because they are unable to differentially select between candidates. If all paths that reach the goal simply evaluate to true, we cannot distinguish shorter paths from longer paths. For efficiency, selection pressure can be increased by changing the fitness multiplier (see program readme.txt)

$$fitness\,(x) = \left(\frac{avg\ generation\ dna\ length}{x\ dna\ length}\right) fitness\_multiplier$$

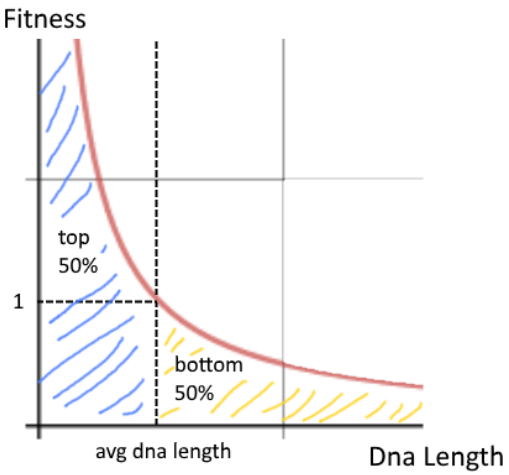


Figure 2.3.3 – Exponentially higher fitness as paths decrease in length from the average.

Reproduction

A master dna pool is used to keep track of the probability spread of each gene/cardinal direction at each location in the genome, based on both the frequency of the gene's occurrence at that dna location as well as the fitness of its owner. For example, given the following four candidates,

Candidate	DNA	Fitness score
1	NNNNNN	10
2	NWSEE	30
3	SSWS	20
4	ESS	40

Table 2.3.4 – Parent dna sequences to be combined into the master dna pool

Master Dna Pool – at Dna[0] gene		
Gene	**Total fitness score**	**Probability of passing on to a candidate**
North	10+30 = 40	40%
West	0	0%
South	20	20%
East	40	40%

Table 2.3.5 – Probability of each cardinal direction to be reproduced at location 0 in the dna sequence of each candidate in the next generation

All candidates in the next generation have a 40% probability of inheriting an "N" gene, 20% "S", and 40% "E" at the first position of their dna sequence. If a candidate reaches the goal state before walking through the master dna pool, sequencing is stopped.

Often a dna sequence will contain "junk dna", resulting in unnecessarily long paths and wasted computation. For example, the dna expression "NSNSNSE" is often equivalent to the expression "E" because of the repeated North/South instructions. As part of the implementation, genes are ignored if they result in symmetrical back and forth movement. Further, genes that instruct the candidate to move into walls are ignored.

While in some respect a genetic algorithm might be considered an informed search because subsequent generations have implicit knowledge about the goal reached by prior generations, the algorithm and dna sequence are still fundamentally naïve and don't make direct use of heuristics. Further, most of the compute time spent by the Genetic algorithm is during the beginning iterative deepening phase where no goal heuristic is available.

Iterative Deepening

The first implementation of the genetic algorithm did not include iterative deepening and came against two significant problems – performance, and maps with no goal states. Because each candidate would randomly make moves until it reached a goal, each generation was expensive to compute and would become stuck if there was no goal to reach. Without iterative deepening, the genetic algorithm could not deal with infinitely deep search spaces e.g. due to looping or lack of a goal state.

By putting a limit on the amount of moves a candidate can make every generation and slowly increasing it over time, the evolution of the algorithm was smoothed out into a gradual process that not lonely looks quite life like but greatly reduced the search tree. Without iterative deepening, some generational candidates were making thousands of random moves to reach the goal going this way and that. Iterative deepening eliminates those unnecessarily convoluted variations and cuts the search off quicker, allowing quicker evolution around shorter candidates that make it to the goal.

It was found that once the algorithm found the goal, it would very rapidly be able to bring its fitness function to bear and improve the path. Instead of taking a full depth first approach, iterative deepening attempts to combine the best of both breadth and depth first.

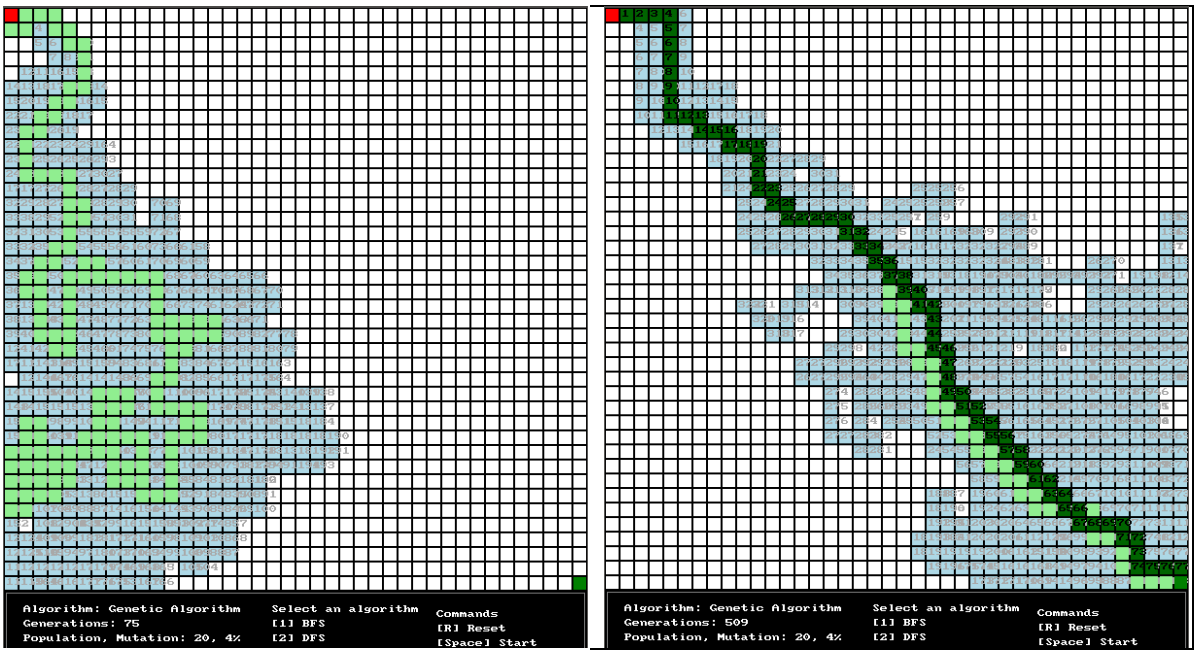


Figure 2.3.6 – Significantly reduced search area using iterative deepening (light blue squares represent candidate variation within the generation). Without iterative deepening, the algorithm would attempt to explore in all directions at once with very high depth, causing the grid to be significantly more blue coloured. Also, a solution is quickly optimised for once a path to the goal is found.

Although no heuristic has been used in order to keep this an uninformed search, iterative deepening opens up the possibility of using goal heuristics if required.

Variation

One of the main issues with genetic algorithms is that they can lock into optimising for fitness around a local minima/maxima. There may exist a more globally optimal solution but the genetic algorithm may be unable to pay the short-term traversal cost of lower fitness in order to get to it.

To test this, the following map was used (Appendix B for map definition).

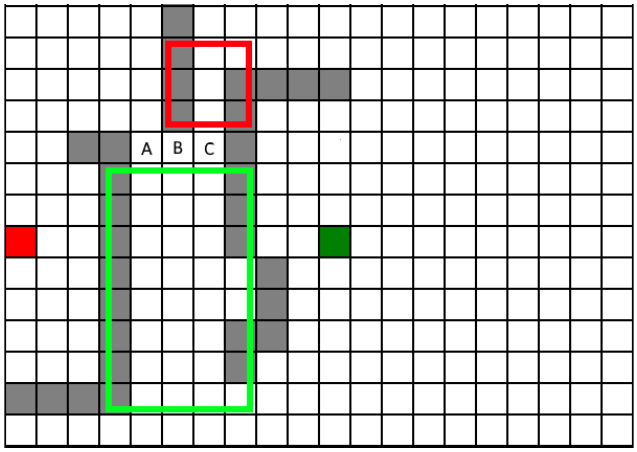


Figure 2.3.7 – Large local optimality in green, small global optimality in red.

The red area can only be reached via the north edge on node C while the green area is easily accessible and rich with possible states. In addition, solutions through the red area are more fragile. If a mutation occurs at C or the squares north of it, the red path likely inverts and passes through the green area instead. Meanwhile, mutations at A and B are relatively less likely to cause differences in the best path as there are many possible best paths through the green open space (see jump point search 3.3).

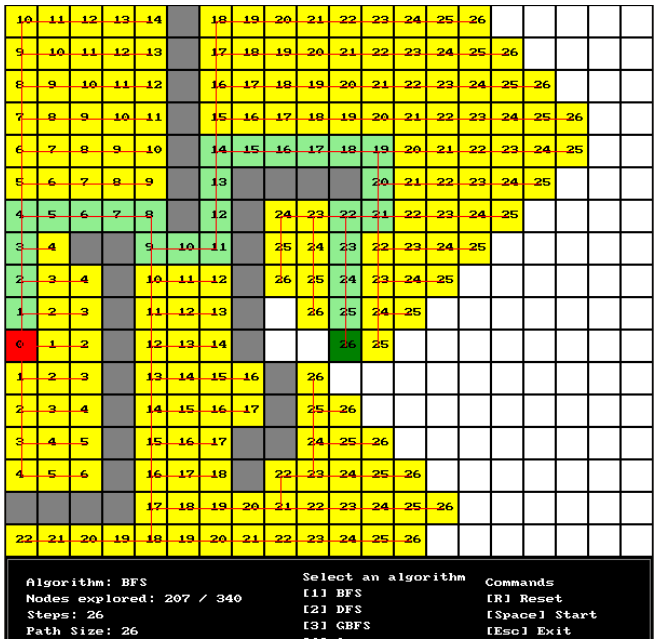


Figure 2.3.8 – Breadth first search finds global minima with path size of 26.

BFS gives us the globally shortest path going through the red area with a path size of 26. However, the genetic algorithm overwhelmingly prefers to go through the green area even though the shortest path in this direction is 28 steps.

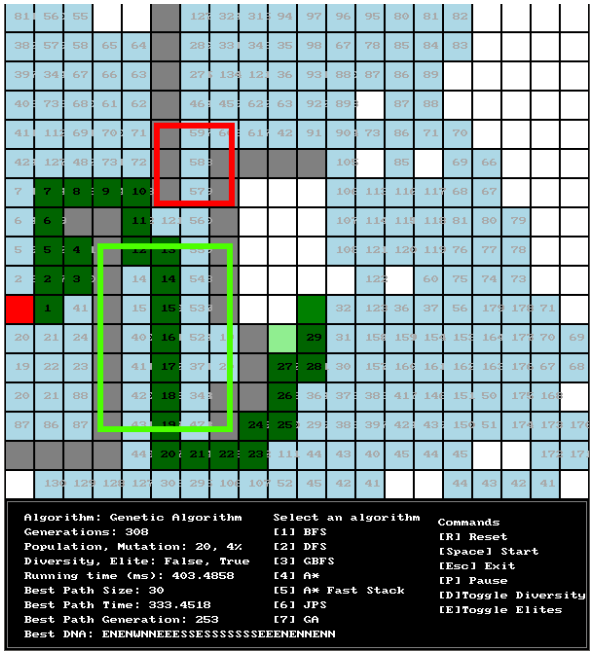


Figure 2.3.9 – Genetic algorithm locks onto a local solution family.

In 10 test runs (300 generations), the algorithm settled on a path through the green area 9 times and settled on a path through the red area 1 time.

Another problem is the reinforcing feedback loop of generational inheritance where the generational dna pool dramatically loses variation the moment it finds a solution to the goal, becoming unable to change whatsoever.

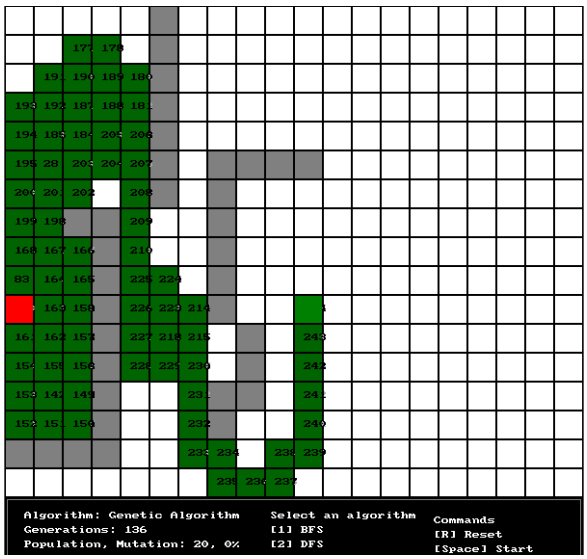


Figure 2.3.10 – With 0% mutation, the algorithm eventually stops exploring different paths and becomes a genetic monoculture (no light blue squares, indicating limted candidate variation within the generation).

To bridge these issues, two techniques have been used – mutation, and diversity selection.

Mutation

A mutation rate allows the algorithm to explore paths outside of its normal tendency and prevents the dna pool from being stuck on a local minima. Where a child might have a 90% chance of inheriting a “North” gene, a mutation event temporarily resets the probabilities to 25% for each cardinal direction.

Using the map from figure 2.3.6, the effect of mutation was tested over 10 runs, population size 20, for 200 generations each at mutation rates 1%, 5%, 10%, 15%, 25%, 40%, and 60%. Unfortunately, there did appear to be any noticeable effect on the proportion of paths that were able to settle in the red area.

However, the mutation rate appears to strongly affect the algorithms efficiency and effectiveness. Too low mutation and the algorithm gets stuck and cannot improve, too high mutation and the search takes too long, causing a high miss rate (no solution found). It appears that a mutation rate somewhere between 1 – 10% is good, after which the speed in which solutions are found decreases with increased miss rates. Under no circumstance should 0% mutation be used.

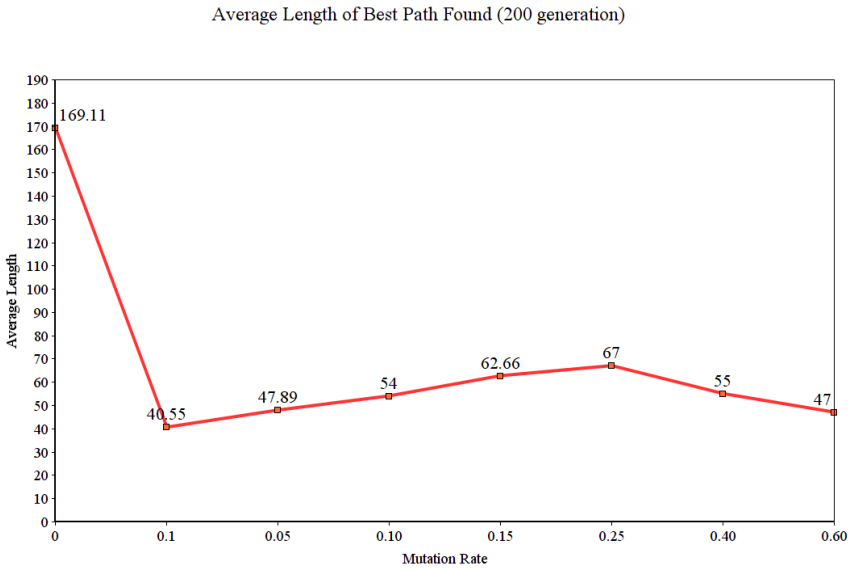


Figure 2.3.11 – Average length of best path found increases with mutation. Decreases in higher mutation rates likely due to increasing miss rates causing low sample size of paths to the goal.

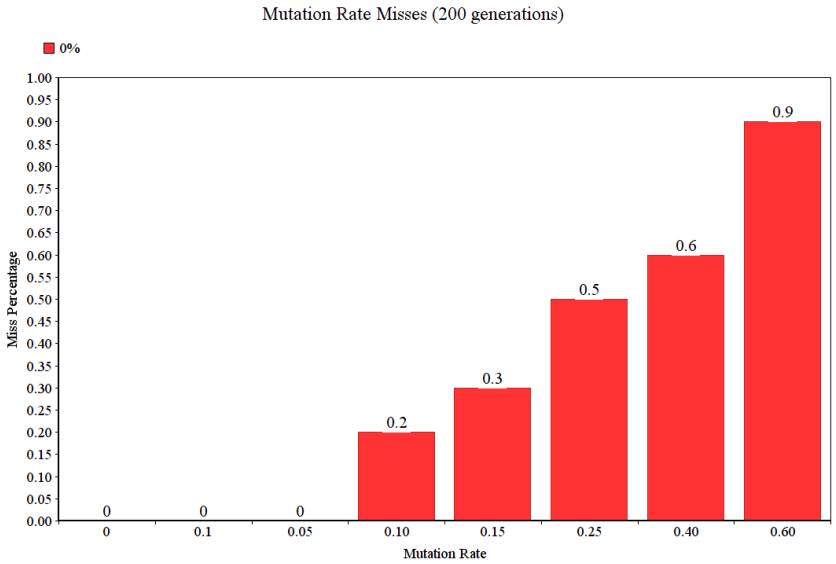


Figure 2.3.12 – Increasing miss rates as mutation rate increases. 90% of searches did not find a solution by the 200th generation with a 60% mutation rate.

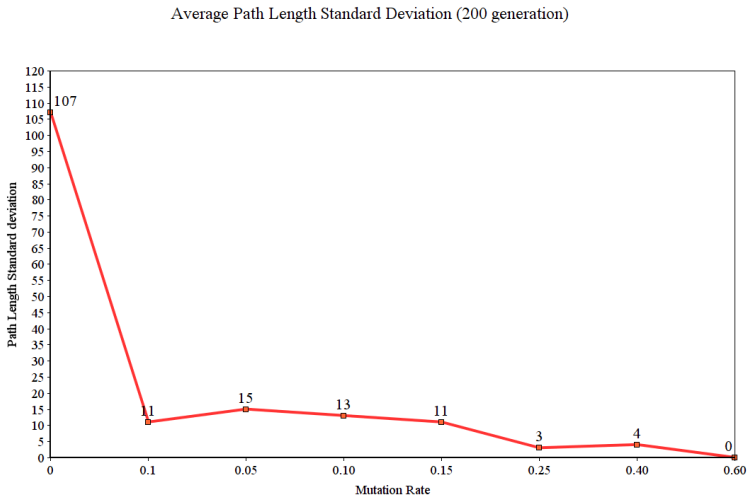


Figure 2.3.13 – Deviation between resulting paths to the goal. 0% mutation has high variability because it gets stuck in local minimas.

Diversity Selection

In addition to fitness selection, we can also value a candidate's genetic distance from the norm. The idea is to select candidates that are both maximally diverse and maximally fit. For example, if two candidates are equally fit, we prefer the one with a higher genetic diversity score.

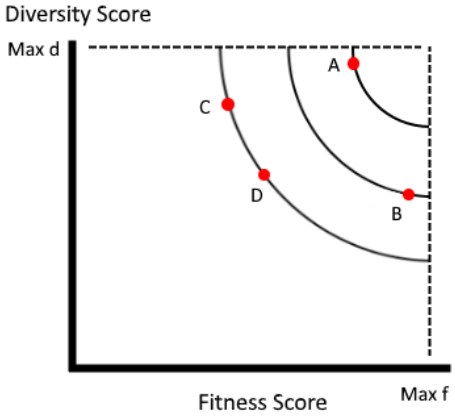


Figure 2.3.14 – A is valued more than B even though A is less fit. C and D are equally valued.

Diversity is measured as the genetic difference between each candidate and the pool of candidates selected purely on their fitness f score. For example, an "E" gene is 1 unit distance from a "W" gene and 0.5 units from an "N" or "S" gene. Each of the candidate's genes is scored, summed, averaged, and scaled by the generation's average fitness f score. This diversity score is then added to all candidates. The highest scoring candidates (fitness + diversity) are selected and a new master dna pool is built to reflect the newly selected set of candidates.

Testing over the map in figure 2.3.6, diversity selection appears to increase proportion of paths that end up settling in the red area.

Diversity Selection Test – 10 runs (20 pop, 200 generations)		
	No Diversity selection	**Diversity selection**
Proportion of best paths that settle in the red area	15%	35%

Table 2.3.15 – results of enabling diversity selection

Elitism

With the change to iterative deepening, it was found that sometimes the algorithm would find a good solution in the current generation but then lose it going forward due to the randomness of dna inheritance. This was especially prominent in complex maps with lots of obstacles and walls.

By persisting the best inter-generational candidate found and mixing it into every generation, the algorithm ensures that good genetic information is not lost. The use of elitism greatly reduces the amount of time needed to optimise a found solution.

Testing the effect of elitism using the medium walls map (Appendix A 6.2), it was found that elitism reduced the number of generations required to find a best path approximate to BFS by about 30 generations or 13%.

Medium Walls - 10 runs (20 pop, 4% mutation)	
	Average generations
Elitism	241.25
No Elitism	277.5

Table 2.3.16 – elitism reduces the number of generations needs to find an approximately optimal solution

2.3.2 Discussion

Completeness

Because the algorithm randomly chooses adjacent nodes until the goal is found, a path will be found if one exists given enough time and memory. Further, because the algorithm is using depth limits, infinitely deep search spaces are not a concern.

Optimality

While it is possible that a shortest path will be found, it is not guaranteed. Genetic algorithms do not have mathematical basis. As the size of the state space increases and local minima's start emerging in the topology, the likelihood of finding the globally shortest path decreases. GA's are good at finding local clusters of solutions that are not fragile to mutation, which might be useful if we are interested in the behaviour of agents that act in an evolutionary/herd like (not rational), manner.

Efficiency

The genetic algorithm is relatively slower than solutions like BFS and A*, especially as the demand for precise navigation around obstacles increases. Presumably because walls decrease the number of path symmetries, make solutions more fragile, and require more precise navigation. Iterative deepening increment per generation can be increased to speed up the search if required (see program readme.txt). e.g. Small no walls test slower due to small deepening increment.

Genetic Algorithm Search (10 runs each)				
Pop: 20	*Mutation: 4%*	*Diversity: No*	*Elitism: No*	*Deepening increment: 1*
Map (shortest path length)		**Iterative deepening Avg time(ms)**	**No iterative deepening Avg time(ms)**	**Performance Increase**
Small – No walls (18)		6.18	1.8	-243%
Medium – No walls (43)		79.88	83.61	+4.5%
Large – No walls (78)		100.31	135.56	+26%
Small – Walls (22)		86	176.90	+51%
Medium – Walls (45)		388	891.72	+56%
Large – Walls (78)		2090	2527.87	+17%

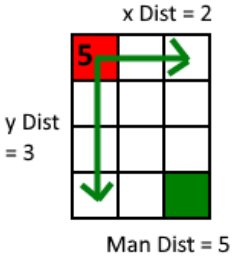
Table 2.3.10 – GA search time results

3. Informed Search

Informed searches utilise information about the goal's location to compute a heuristic h that roughly indicates the distance of a given state from the goal state, represented using f(x) = g(x) + h(x). Heuristics allow more informed decisions to be taken about which nodes are likely to best lead to the goal state. This has the effect of reducing the number of nodes that need to be explored.

Because the robot navigation problem is a pathfinding grid with uniform cost, the heuristic chosen is Manhattan Distance not only because it is admissible but because it can be very rapidly calculated. With manhattan distance, obstacles are ignored and treated as walkable.

$$ManDist\,(node) = Abs\,(Goal.X - node.X) + Abs\,(Goal.Y - node.Y)$$



3.0 – Finding Manhattan distance

3.1 Greedy Best First Search

3.1.1 Method

Greedy best first only considers the heuristic h component, based on the assumption that this is likely to lead to the goal the fastest. GBFS is represented as f(x) = h(x) where g cost evaluates to 0.

The open set is first seeded with a starting node. The node in the open set with the lowest heuristic value is chosen, removed from the open set, put into the closed set to prevent looping, and its valid adjacent nodes are found. Each of these adjacent nodes are assigned a heuristic value, assigned a parent node, put into the open set, and then the algorithm loops back until a goal is found or the open set is empty.

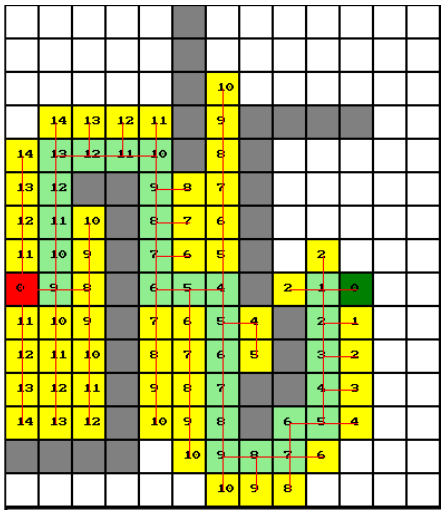


Figure 3.1.0 – Operation of GBFS path finding. Numbers represent the h cost at each node.

3.1.2 Discussion

Completeness

Because GBFS uses a closed set to prevent looping, it will find a path to the goal if one exists.

Optimality

GBFS does not guarantee finding the shortest path because it does not consider g cost. Instead, it depends on the heuristic method used. While heuristics are generally calculated using quick and dirty methods e.g. Manhattan distance ignores walls, it is possible to use BFS or Dijkstra's to calculate "heuristic" values to find the best adjacent nodes to take. However, this obviously defeats the point of using cheaply computable heuristics in the first place.

Efficiency

GBFS is relatively fast to compute as it tends to minimise the number of nodes explored. It is also often correct in assuming that directly moving towards the goal results in short if not shortest paths.

However, this strategy generates strange paths under certain situations because Manhattan distance does not consider walls. Furthermore, heuristics result in strange behaviour with multiple active goals because a child node's h value may refer to a different goal than its parent's h value.

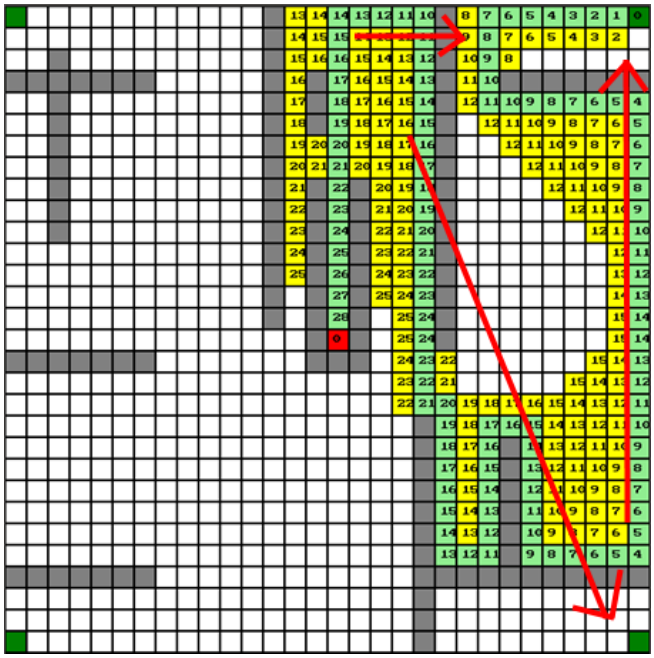


Figure 3.1.1 – GBFS first seeks the top right goal, then the bottom right goal, and then the top right goal again.

GBFS has limited practical use other than quickly finding paths where compute time is relatively more valuable than finding optimal paths. The use of heuristics allow a lot of states to be pruned.

10 runs each	**GBFS**	**DFS**	**BFS**
Map	**Average time (ms)**	**Average time (ms)**	**Average time (ms)**
Small – No walls	0.11	0.30	0.24
Medium – No walls	0.39	0.62	1.96
Large – No walls	1.08	8.75	6.68
Small – Walls	0.20	0.27	0.25
Medium – Walls	0.84	1.54	1.59
Large - Walls	1.35	6.74	5.20

Figure 3.1.2 – GBFS search times

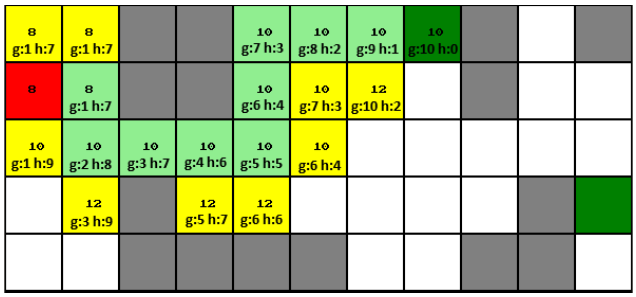
3.2 A* Search

3.2.1 Method

A* considers both g and h values when choosing what node to explore, represented as f(x) = g(x) + h(x).

The open set is first seeded with the starting node. A* chooses the node with the lowest f value from the open set, removes it from the open set and adds it to the closed set. Each successor of the chosen node is expanded, added as a child of their parent node, added to the open set, and both their g and h costs are calculated. If the calculated g cost is lower than the g cost that the node already has, then the lower g cost overwrites the old g cost. The algorithm then loops until the goal is reached or there is nothing left in the open set.

G and h costs need to be stored separately because where the f values of two nodes are the same, the algorithm preferentially explores the node with the lower h cost. This preferences nodes that are closer to the goal, as well as the greater reliability of a higher g cost.



3.2.0 – Operation of A* search

3.2.2 Discussion

Completeness

Given an admissible heuristic, A* will find a path to the goal if one exists.

Optimality

Given an admissible heuristic, A* will find the shortest path to the goal.

Efficiency

Heuristics greatly reduce the number of nodes to expand by indicating a direction to the goal. However, this comes at the cost of having to use a sorted open list. As state space increases, A* tends to build up very large open sets. Maintaining an effectively sorted structure to find the node with the lowest f cost becomes increasingly expensive.

One way to mitigate this is to use a 'fast stack'. Because the f cost of a discovered adjacent node will never be smaller than the f cost of its parent (assuming an admissible heuristics that doesn't overestimate), all child nodes with the same f value as their parent are automatically fast tracked.

This is because the child is at the lower bound of possible f costs and would be evaluated first in the next iteration. Using a separate fast stack to store these nodes for priority processing avoids having to search/scan through the open set. Using a fast stack results in significant performance increases as the state space increases.

	A* search (unoptimized)	**A* search (Fast stack)**	**Fast Stack % Improvement**
Map	**Average time 10 runs (ms)**	**Average time 10 runs (ms)**	
Small – No walls	0.15	0.15	0%
Medium – No walls	0.55	0.53	3.7%
Large – No walls	1.59	1.51	5.2%
Small – Walls	0.39	0.38	3.1%
Medium – Walls	1.84	1.39	24.66%
Large - Walls	5.07	3.35	33.91%

Figure 3.2.2 – A* search time results

3.3 Jump Point Search

3.3.1 Method

With uniform cost grids, it is often the case that there exist open spaces that contain many possible best paths between the start and goal. The purpose of jump point search is to break these path symmetries by pruning out duplicate paths and reducing the number of nodes that need to be processed through the open set (Harabor & Grastien, 2011).

For example, in an open space with w width and h height, the total number of symmetric best paths between start and goal is given by (w+h)! / (w!*h!). In a 10 by 5 grid, there are 3,003 (15! / 5!*10!) duplicate best paths, all with identical length 13. One of the problems with A* and other informed search algorithms is that they will often consider all these best paths, resulting in duplicate work.

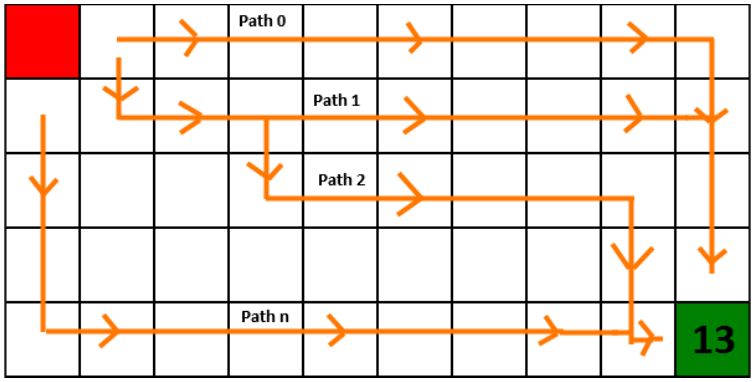


Figure 3.3.0 – Multiple symmetric best paths exist in uniform cost grids.

To break path symmetries, jump point search scans along direction vectors. While the original JPS algorithm was designed for 8 directional grid movement, it can be adapted to work with 4 directional grid movement by scanning left and right at each y position until a forced neighbour is found (figure 3.3.1). Figure 3.3.2 shows why we can ignore exploration of a lot of adjacent nodes. if we are coming to ***node x*** from the direction of ***node 4***, we only need to explore ***node a*** via the red arrows because all other nodes can be explored by alternate best paths shown in yellow. In this way, we prune out 75% of the nodes that need to be considered.

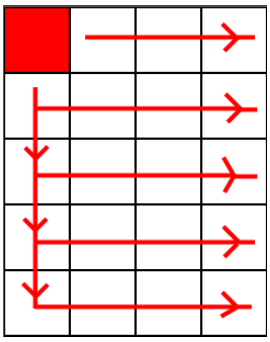
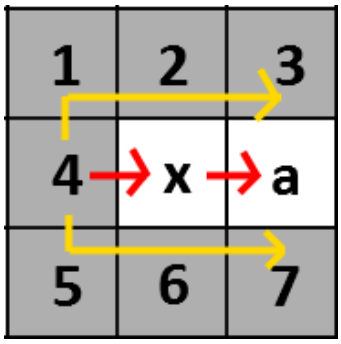


Figure 3.3.1 – JPS scanning in 4 directional state space (N, W, S, E).



3.3.2 – Horizontal scanning.

However, if a horizontal scan encounters a wall (figure 3.3.3), it is prevented from exploring nodes further along that row i.e. node 2 is unreachable from node 1. JPS refers to these as forced neighbours. A primary jump point at **node x** (a point of interest) is created to allow us to hook north.

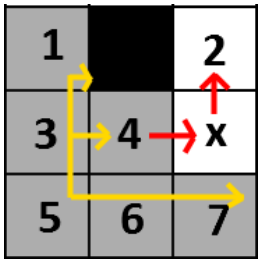


Figure 3.3.3 – Forced neighbour.

JPS directly connects node x as an successor of node 3 from the rightward direction, skipping the need to process intermediate nodes (i.e. node 4) through the open set. JPS also looks for the goal when scanning horizontally and places jump points accordingly.

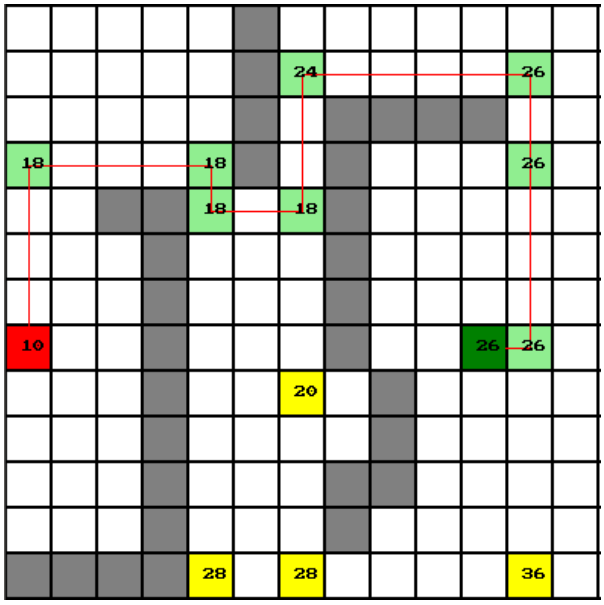


Figure 3.3.4 – Operation of Jump point search

To find the shortest path, node costs are evaluated the same as with A*: f(x) = g(x) + h(x). The jump point with the lowest f cost is removed from the open set and its successor jump points are searched for. Successor jump points are given a g and h cost, assigned a parent, and added to the open set. The algorithm then loops back until the open set is empty or the goal is found.

3.3.2 Discussion

Completeness

JPS will find a path to the goal if one exists.

Optimality

JPS is a pruned version of A* that takes advantage of uniform cost grids. Therefore, it will find the shortest path to the goal if the heuristic used is admissible.

Efficiency

Performance increases over traditional informed search strategies like A* are significant. However, JPS only works over uniform cost grids. If movement costs are not always the same e.g. moving north costs twice as much, or if some nodes cost more to navigate through, then JPS does not work.

10 runs each	**Jump Point Search**	**A* (Fast stack)**	**GBFS**
Map	**Average time (ms)**	**Average time (ms)**	**Average time (ms)**
Small – No wall	0.06	0.15	0.11

Medium – No wall	0.28	0.53	0.39
Large – No wall	0.67	1.51	1.08
Small – Walls	0.12	0.38	0.20
Medium – Walls	0.60	1.39	0.84
Large – Walls	0.77	3.35	1.35

Figure 3.3.5 – JPS search times

4. GUI

For debugging and learning purposes, a GUI has been developed to help visualise what the algorithms are doing as well as to provide some comparative and informative metrics.

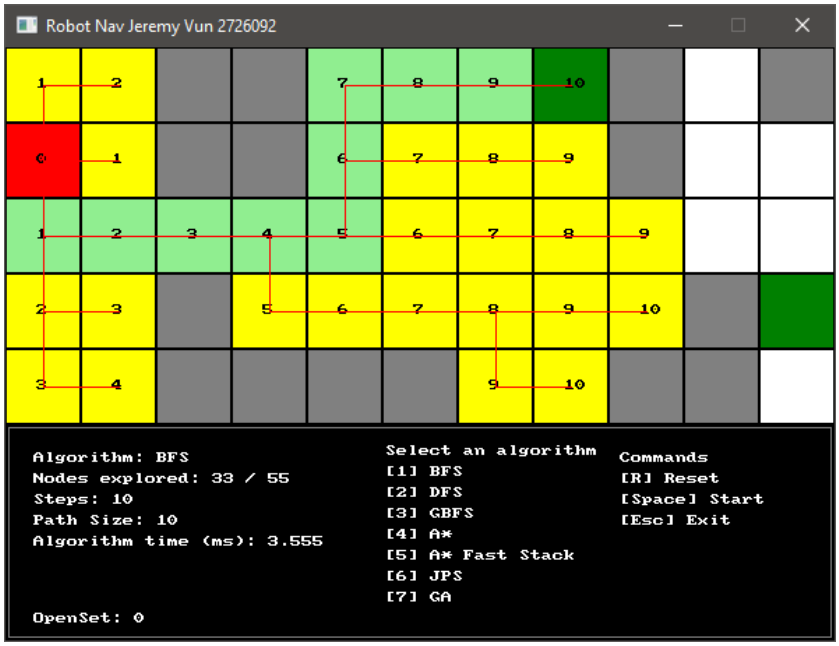


Figure 4.0 – GUI window

Press a **number key** to select an algorithm, space to start the search, **r** to reset the search, and **esc** to quit the program.

Node Color Code	
Colour	**Description**
Red	Start Node
Green	Goal Nodes
Yellow	Explored nodes in the closed set
Light Green	The best path found
Light Green (GA Only)	The best path found in the current generation
Dark Green (GA only)	The best path found across all generations
Light Blue	Nodes in the open set
Dark Blue (ASFS Only)	Nodes in fast stack
White	Unexplored nodes
Grey	Walls

Figure 4.1 – colour codes

Grid numbers represent the path cost metric that the algorithm is measuring. For example, BFS, DFS and GA show ***g cost*** while GBFS, A* and JPS show ***f cost***.

Red lines show the tree graph as explored by the algorithm (parents -> children).

5. References

Harabor, D. & Grastieint, A. 2011, 'Online Graph Pruning for Pathfinding On Grid Maps', In Association for the Advancement of Articial Intelligence, *Twenty-Fifth AAAI Conference on Artificial Intelligence,* pp. 1114-1119.

Swingame library used to render GUI, *http://www.swingame.com/*

6. Appendix A

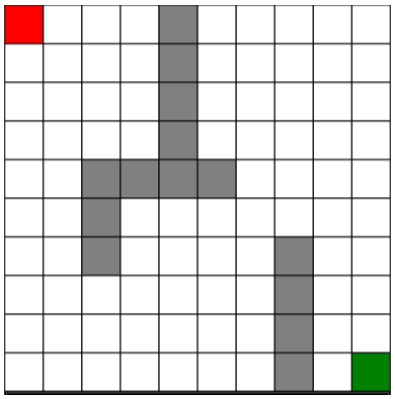
6.1 Small

No walls

```
[10,10]
(0,0)
(9,9)
```

Walls

```
[10,10]
(0,0)
(9,9)
(2,4,4,1)
(4,0,1,4)
(2,5,1,2)
(7,6,1,4)
```



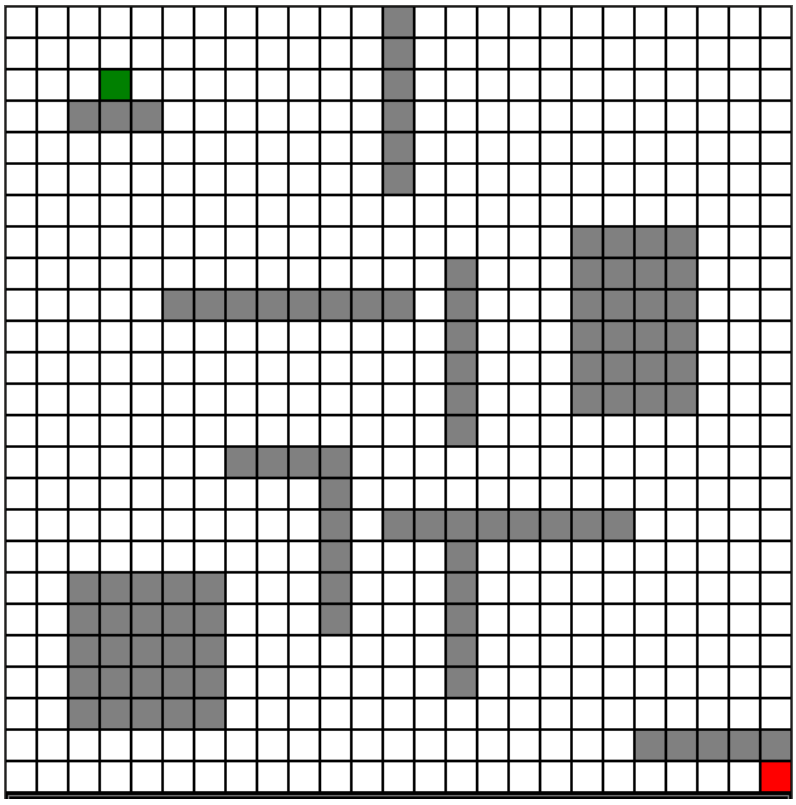
6.2 Medium

No walls

```
[25,25]
(24,24)
(3,2)
```

Walls

```
[25,25]
(24,24)
(3,2)
(2,18,5,5)
(2,3,3,1)
(5,9,8,1)
(7,14,3,1)
(10,14,1,6)
(12,0,1,6)
(12,16,8,1)
(14,8,1,6)
(14,17,1,5)
(18,7,4,6)
(20,23,5,1)
```



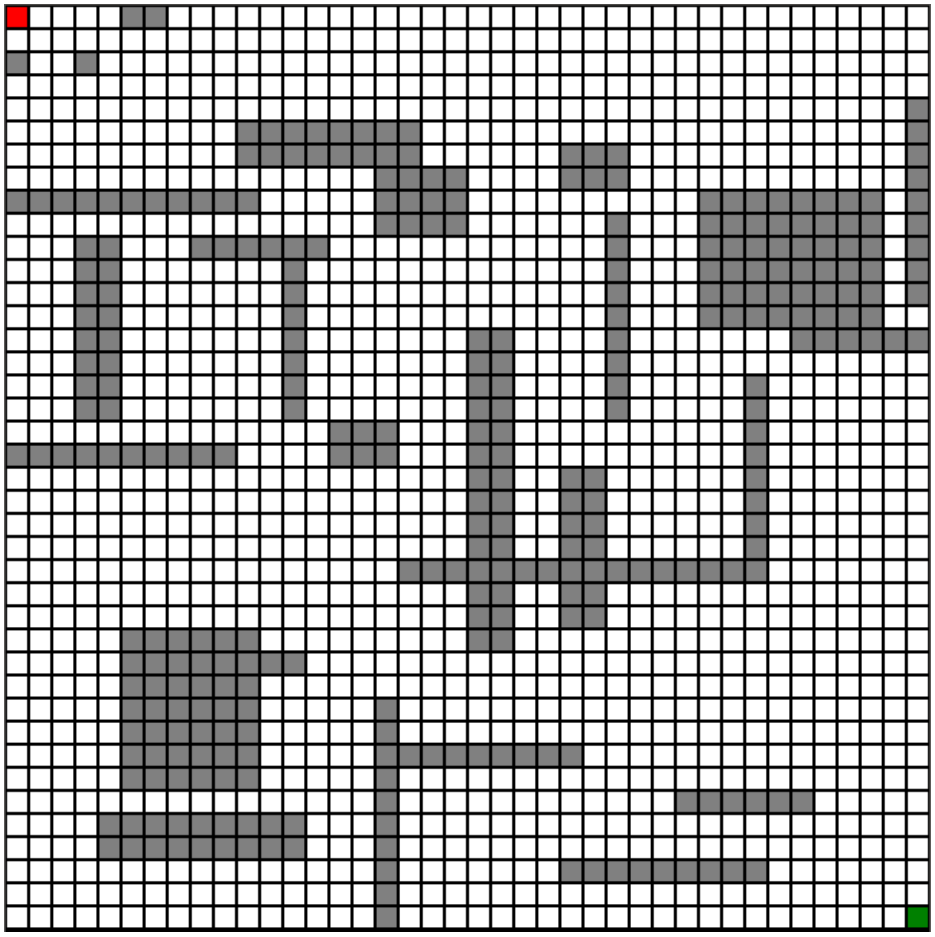
6.3 Large

No walls

```
[40,40]
(0,0)
(39,39)
```

Walls

```
[40,40]
(0,0)
(39,39)
(0,2,1,1)
(3,2,1,1)
(3,10,2,8)
(24,6,3,2)
(5,0,2,1)
(24,20,2,7)
(39,4,1,9)
(20,14,2,14)
(30,8,8,6)
(29,34,6,1)
(26,9,1,9)
(7,28,6,1)
(10,5,8,2)
(14,18,3,2)
(17,24,15,1)
(32,16,1,9)
(8,10,6,1)
(12,10,1,8)
(24,37,9,1)
(4,35,9,2)
(34,14,6,1)
(5,27,6,7)
(16,7,4,3)
(0,19,10,1)
(16,30,1,10)
(0,8,11,1)
(17,32,8,1)
```



7. Appendix B (local minima test)

```
[17,20]
(0,10)
(10,10)
(2,7,1,1)
(3,7,1,9)
(0,15,4,1)
(5,0,1,7)
(7,5,4,1)
(7,6,1,5)
(8,11,1,3)
(7,13,1,2)
```