

# Tree-Based Search

## *Robot Navigation Problem*



**Jacob Milligan**  
100660682

Swinburne University of Technology  
*COS30019 - Introduction to AI*

April, 2017, Semester 1

## Acknowledgements

I would like to acknowledge the work of Russell and Norvig (Russell and Norvig 2010), whose book 'Artificial Intelligence: A Modern Approach' was referenced significantly during both research and implementation of the search program. I would also like to acknowledge the work of Amit Patel, found at [Red Blob Games](#), whose writings on A\* and heuristics-based search assisted greatly in understanding these algorithms (Patel 2014).

## Software Libraries and Code Repository

The code-base is version controlled via git and can be accessed on [github](#) and uses several external libraries. These include:

- [Catch](#) - A header-only unit-testing framework for C++
- [SDL2](#) - Platform abstraction media library used in the visualizer for opening a window, handling input and rendering graphics
- [SDL2\\_ttf](#) - TrueType rendering library extension for SDL2 for drawing text
- [Skyrocket.Path](#) - The platform agnostic file-system component from my game framework, Skyrocket, for navigating the file-system independent of the operating system being used.

# Contents

|          |  |           |
|----------|--|-----------|
| <b>1</b> | <b>Introduction</b>  | <b>1</b>  |
| <b>2</b> | <b>Search Algorithms</b>                                       | <b>1</b>  |
| 2.1      | A* Search (A*) . . . . .                                       | 1         |
| 2.2      | Breadth-First Search (BFS) . . . . .                           | 1         |
| 2.3      | Depth-First Search (DFS) . . . . .                             | 2         |
| 2.4      | Greedy Best-First Search (GBFS) . . . . .                      | 2         |
| 2.5      | Iterative-Deepening Depth-First Search (IDDFS)(cus1) . . . . . | 3         |
| 2.6      | Iterative-Deepening A* Search (IDA*)(cus2) . . . . .           | 3         |
| <b>3</b> | <b>Implementation</b>  | <b>3</b>  |
| 3.1      | Search Methods . . . . .                                       | 3         |
| 3.2      | Frontier . . . . .   | 5         |
| 3.3      | Explored Set . . . . .   | 5         |
| 3.4      | Environment . . . . .  | 5         |
| <b>4</b> | <b>Features and Known Bugs</b>                                 | <b>5</b>  |
| 4.1      | CLI application . . . . .                                      | 5         |
| 4.2      | Known bugs . . . . .   | 6         |
| <b>5</b> | <b>Research</b>  | <b>6</b>  |
| 5.1      | Visualizer . . . . .   | 6         |
| 5.2      | Search Method Analysis . . . . .                               | 7         |
| 5.2.1    | Methodology . . . . .  | 7         |
| 5.2.2    | Results . . . . .  | 7         |
| 5.2.3    | Discussion . . . . .   | 9         |
| <b>6</b> | <b>Conclusion</b>  | <b>10</b> |
| 6.1      | Improvements . . . . .   | 10        |
| 6.2      | Final Thoughts . . . . .                                       | 10        |
|          | <b>Acronyms</b>  | <b>10</b> |
|          | <b>Glossary</b>  | <b>11</b> |

## 1 Introduction

Tree and graph-based data structures are prevalent in many AI-related problems, especially in the implementation of search algorithms for finding a goal within a problem space. This report explores six such search methods and their [completeness](#), [optimality](#), and efficiency as applied to the [Robot Navigation Problem \(RNP\)](#). The RNP is a [Task environment](#) in which a simulated robot attempts to find an optimal route to a given *goal node* in an  $N \times M$  grid of nodes, where  $N = \text{width}$ ,  $M = \text{height}$  and both are non-zero. To demonstrate the execution of these search methods, a GUI-based program for building different states of the RNP, executing each algorithm, and displaying information about their results.

**Glossary of Terms** The concepts and terminology referenced here as they relate to graph and tree based search are defined and described for the reader in section 10, [glossaries](#). Links to these entries are found throughout the report, highlighted blue for reference.

## 2 Search Algorithms

Six search methods were implemented to provide solutions to the [RNP](#), three of which are [informed search](#) methods and three [uninformed search](#) methods. Each was tested using a combination of unit testing and statistical experiments outlined in subsection 5.2.1.

### 2.1 A\* Search (A\*)

A\* is an [informed search](#) search method that stores its frontier in a priority queue structure sorted by best node first, as determined by the nodes *evaluation function*  $f(n)$  which evaluates the cost of reaching a given node  $n$ . For A\*,  $f$  is defined as the cost to reach  $n$  from the starting node along the current path -  $g(n)$  and its heuristic function  $h(n)$  which provides an estimation of the cost of reaching the goal. A requirement for A\* to be correctly implemented is that  $h$  be [admissible](#), that is it never overestimates the quality of a path or node. In the context of the RNP, as the environment is a 2D grid, an admissible heuristic can be achieved by simply measuring the distance from the current node using either euclidean distance, calculated by  $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$ , or using a 'city-block' distance (the amount of nodes traversed without diagonal movements) referred to as *manhattan* distance, calculated by  $|x_1 - x_2| + |y_1 - y_2|$ . A\* is both complete and optimal (Hart, Nilsson, and Raphael 1968, p. 104), as well as efficient in terms of both time and space complexity. For this reason it's used as a comparison for all subsequent search methods described here.

### 2.2 Breadth-First Search (BFS)

Breadth-First search is an uninformed search method that uses a [First-In, First-Out queue structure \(FIFO\)](#) as its frontier, resulting in the shallowest node encountered expanded first. As a consequence of this expansion method, BFS is [complete](#) in all spaces as eventually every node in the search space will be expanded. Furthermore, BFS is technically [optimal](#) when the step cost of a node is non-decreasing and equivalent to it's depth (Russell and Norvig 2010, p. 82). However, while BFS is both complete and optimal it can be inefficient in terms of both memory and time. As BFS eventually analyses every node in the search space, larger grids can increase its memory consumption much more than other algorithms, as demonstrated in figure 2.2. Furthermore, it's runtime increases proportional the time it takes to scan it's list of neighbors (Cormen et al. 2009, p. 597), which will always be the amount of nodes that exist at depth  $d$  extending from the start, minus the ones explored.

### 2.3 Depth-First Search (DFS)

Depth-First search is an uninformed search method that uses a **Last-In, First-Out queue structure (LIFO)** as its frontier. It will always follow the last-encountered node until it hits either a leaf node (a node with no children) or the goal. In theory this poses obvious advantages over BFS, namely in terms of its memory usage - as it only ever contains the current path and each of the path nodes immediate neighbors. However, DFS is non-optimal - for example, if it chooses to expand a node to its left (based a pre-defined order of expansion), it will follow that path even if the goal was located one node to the right. In the worst case, this could result in DFS following the longest possible path possible rather than a shorter alternative. Figure 2 highlights this problem, showing the difference in distribution of path lengths between BFS and DFS across 300 randomly generated grids. Furthermore, DFS is only complete in graph-based searches whereas the tree-based version is *incomplete* (Russell and Norvig 2010, p. 86). Both of these factors result in large run-time consequences, especially as the search space increases in size as demonstrated in section 5.2.

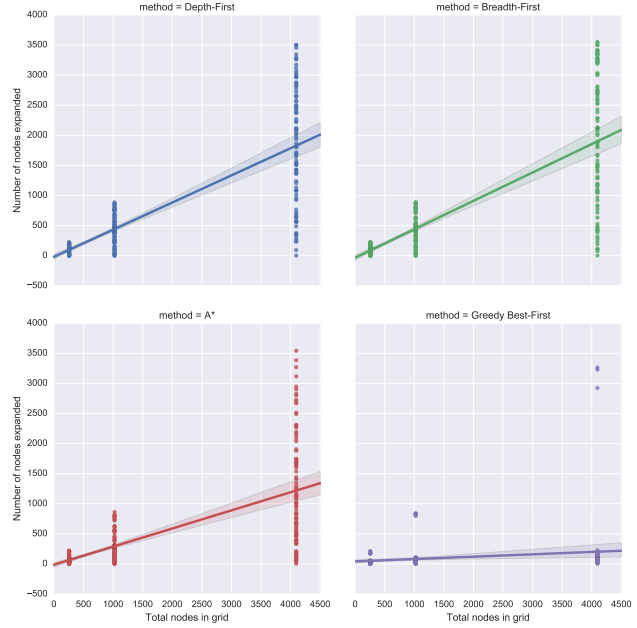


Figure 1: Linear relationship between size of RNP grid and the number of nodes expanded for each search algorithm

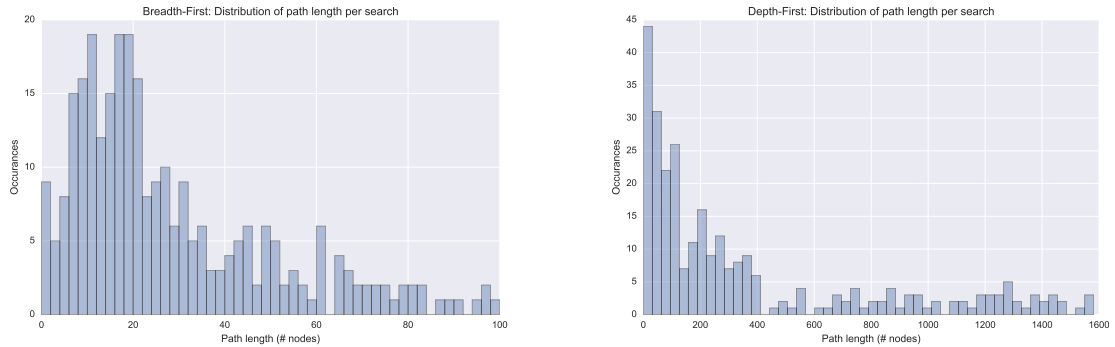


Figure 2: Comparison between the distributions of total path length for Breadth-First Search and Depth-First Search

### 2.4 Greedy Best-First Search (GBFS)

Greedy Best-First is an *informed* search method which, like all informed methods outlined in this report, uses a priority queue frontier ordered by lowest  $f$  value. In contrast to A\*, GBFS expands the node that is closest to the goal first, therefore its evaluation is  $f(n) = h(n)$ . GBFS, in contrast to A\*, is not optimal nor is it complete (Jones 2008), this is because its heuristic is

not *admissible* due to often overestimating the quality of a node, for example in scenarios when the goal is behind a wall but unreachable, the node closest will be evaluated as the best neighbor when it isn't. However, often GBFS can return a *close to optimal path* while expanding less nodes in a faster time than other search methods, demonstrated in figure 3.

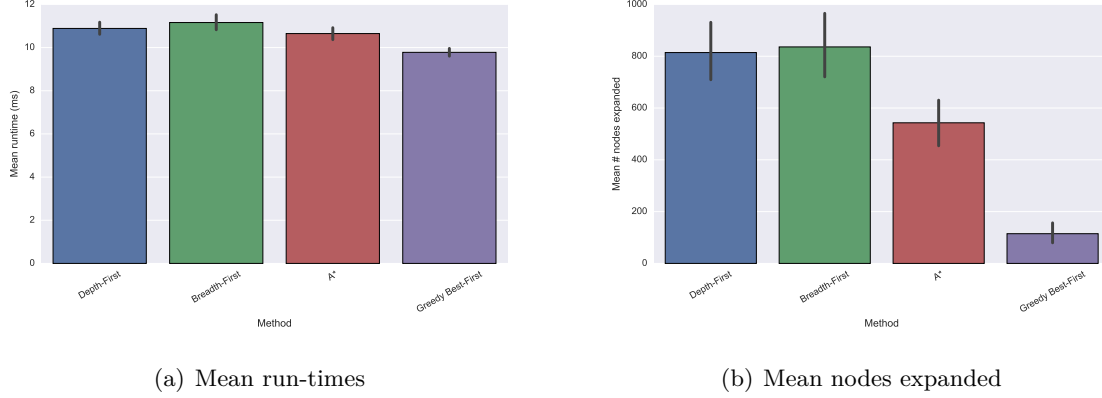


Figure 3: The mean number of nodes expanded and mean run-time for the non-custom search methods

## 2.5 Iterative-Deepening Depth-First Search (IDDFS)(cus1)

Iterative Deepening Depth-First search is an improved version of regular depth-first search uninformed search which executes DFS until either the goal is encountered or some depth-limit  $\ell$  is reached, signaling a cutoff upon which the search tree is cleared, increasing  $\ell$  by 1 and running the algorithm again, running iterations of DFS until the goal is found. This method aims to combine the best qualities of DFS and BFS, taking the limited space requirements of DFS and combining it with the completeness and optimality of BFS.

## 2.6 Iterative-Deepening A\* Search (IDA\*)(cus2)

Iterative Deepening A\* is a search method that attempt to improve the memory requirements of A\*. It is similar to IDDFS in its methodology, except for what is used as  $\ell$  for the current iteration. A cutoff is reached when the current nodes  $f$  cost exceeds the current  $\ell$  value, upon which  $\ell$  is increased to be equal to that nodes  $f$ , the search tree cleared and another iteration begun. In a sample of 300 randomly generated grids of differing sizes, as outlined in section 5.2, IDA\* cut down the mean amount of nodes expanded by approximately 50 nodes.

However, while memory requirements are improved, on graph-based searches such as the RNP IDA\*'s tendency to analyze the same state for different depths in a standard implementation showed an increase of approximately  $2ms$  per search in the same tests. However, it's important to note that the implementation described in section 3 does not make much effort to track repeated states in graph-based problems and IDA\* is technically as efficient as A\* given the right implementation (Korf 1985).

# 3 Implementation

## 3.1 Search Methods

Each search method is a derived form of the base `SearchMethod` class, each implementing the `search()` and `get_child()` methods.

**BFS** BFS was implemented iteratively, using a FIFO frontier, its pseudocode shown below.

---

**Algorithm 1** BFS

---

```

1: procedure SEARCH(env - a problem environment)
2:    $n \leftarrow$  starting state,  $explored \leftarrow \{\}$ ,  $frontier \leftarrow$  empty FIFO queue
3:   if  $n = goal$  then
4:     return solution
5:   end if
6:   add  $n$  to  $frontier$ 
7:   while  $frontier \neq empty$  do
8:      $n \leftarrow$  first node in frontier
9:     for all valid child  $c$  with action  $a$  in  $env$  do
10:      if not explored then
11:        add  $c$  to explored set
12:        if  $c = goal$  then
13:          return solution
14:        end if
15:        add to frontier
16:      end if
17:    end for
18:  end while
19:  return no solution found
20: end procedure

```

---

**DFS** DFS is implemented in a very similar way to the pseudocode outlined above for BFS, however it uses a LIFO queue as it's frontier instead of a FIFO queue.

**GBFS and A\*** Once again, GBFS and A\* are both implemented similarly to BFS and DFS but using a priority queue as their frontier. Furthermore, when getting a valid child from the environment, both add their heuristic cost to the child node before adding it to the frontier, to do this they override `SearchMethods`'s `get_child()` method. GBFS will simply assign as its  $f$  cost the distance to the goal using one of the distance functions outlined in section 2.1, while A\* will sum both the distance and the child nodes parent  $f$  value.

**IDDFS** IDDFS is implemented recursively in the following manner using a specialized results struct containing a cutoff value and the solution if it exists.

---

**Algorithm 2** IDDFS implementation pseudocode

---

```

1: procedure SEARCH(env - a problem environment)
2:    $results \leftarrow$  empty results set,  $\ell \leftarrow 1$ 
3:   while  $results.cutoff$  do
4:      $results \leftarrow depth\_limited\_search(env, \ell)$ 
5:      $\ell \leftarrow \ell + 1$ 
6:   end while
7:   return  $results$ 
8: end procedure
9:
10: procedure DEPTH_LIMITED_SEARCH(env - a problem environment,  $\ell$ )
11:    $n \leftarrow$  starting node in  $env$ 
12:   return  $recursive\_dls(n, env, \ell)$ 
13: end procedure

```

---

```

14:
15: procedure RECURSIVE_DLS(env - a problem environment, n - a node. d - the current
    depth)
16:   Add n to explored set
17:   if n = goal then return solution
18:   end if
19:   if d < 1 then return cutoff
20:   end if
21:   cutoff_occurred  $\leftarrow$  false
22:   results  $\leftarrow$  empty results set
23:   for all children c of n with valid actions a in env do
24:     if c not in explored set then
25:       results  $\leftarrow$  recursive_dls(c, env, d - 1)
26:       if results.cutoff then cutoff_occurred  $\leftarrow$  true
27:       else if solution found in results then return results
28:       end if
29:     end if
30:   end for
31:   return cutoff_occurred
32: end procedure

```

---

**IDA\*** IDA\* is implemented almost identically to IDDFS, however it keeps a temporary map of the current path to ensure that nodes in the same path aren't repeated, alongside using the last encountered *f* value higher than the current limit as the cutoff.

### 3.2 Frontier

Each search method utilized a different frontier container type. The interface for this is provided via the `Frontier<T>` template type which has FIFO (`Frontier<std::queue>`), LIFO (`Frontier<std::vector>`), and Priority Queue (`Frontier<std::priority_queue>`) specializations whilst providing the same interface for all types.

### 3.3 Explored Set

The `ExploredSet` class provides an interface for adding and querying the nodes so far expanded for each search algorithm, this is implemented using an `std::unordered_map`. The explored set also contains an `std::vector` of all the operations that occurred during the search algorithm for use in the visualizer (see 5.1) for displaying the search tree as it was operated on in order.

### 3.4 Environment

Finally, the environment class defines valid actions and generates new child nodes based on the current *state* - (a point in the grid). It also defines a *goal test* for determining whether a given state is the goal.

## 4 Features and Known Bugs

### 4.1 CLI application

The CLI reads in test files from a relative path from the executables directory. Alongside the positional arguments required by the assignment spec, the CLI application also defines the following option flags:



- `-h | --help` - Displays a help message describing available commands
- `-v | --visualizer` - Opens the GUI visualizer
- `-s | --stats` - Displays extra stats about the algorithm such as largest frontier, must be placed after all positional arguments

The following search methods are available in both the CLI and the GUI visualizer:

- `BFS` - Breadth-First Search (BFS)
- `DFS` - Depth-First Search (DFS)
- `GBFS` - Greedy Best-First Search (GBFS)
- `AS` - A\* Search (A\*)
- `CUS1` - Iterative-Deepening Depth-First Search (IDDFS)
- `CUS2` - Iterative-Deepening A\* Search (IDA\*)

## 4.2 Known bugs

- **A\*** - A known bug exists where every so often A\* will expand a few nodes more than it should, however this doesn't affect the completeness or optimality of the implementation
- **IDA\*** - While the implementation here expands less nodes than A\*, it does so by keeping track of explored nodes. This shouldn't be necessary for IDA\* to work, however without that feature the algorithm seemed to take far too long to complete.
- **IDDFS** - IDDFS sometimes takes a very long time on large grids - an issue I was unable to resolve.

## 5 Research

The research component can be split into two components - the GUI **visualizer** and the search method **analysis** component.

### 5.1 Visualizer

To be able to both verify the behavior of each search algorithm and to demonstrate to others how each method affects its search tree, a GUI visualizer was developed as a research component. It can be opened via the same CLI application as the assignment component using the `./robonav -v` option. This will open the visualizer app - a grid with cells representing states in the search space - black cells are walls, the red cell is the start state, and the green cell is the goal.

**Usage** Both the start and end state can be moved by clicking and dragging, while walls can be placed and removed by clicking in empty spaces. The keys **1** - **6** are used to choose each search algorithm, **space** clears the current path and **enter** begins a search. To quit press the **escape** key.

**Visuals** The visualization of each search algorithm is not in real-time but in a time proportional to the amount of operations each one made. The visualizer displays each operation as it happened in order as denoted by the moving orange cell (the current node in the search tree), explored nodes are denoted as light blue which are added as the search tree expands. Finally, once the goal is found, the path taken to reach it is highlighted green.

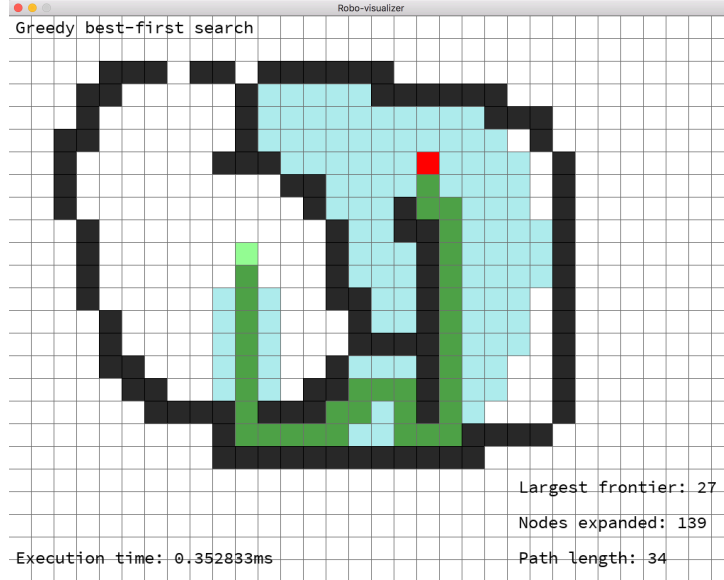


Figure 4: Screenshot of the GUI pathfinding visualizer

## 5.2 Search Method Analysis

While each search method has been discussed at a high level, this section aims to find the most effective search algorithm for the RNP. A method for random-generation of search problems with expected optimal paths was developed for the purpose of testing each search method across a variety of environments, with their time and memory usage recorded alongside the optimality of each solution.

### 5.2.1 Methodology

100 randomly generated search spaces were generated for grids of sizes  $16 \times 16$ ,  $32 \times 32$ , and  $64 \times 64$ , 300 samples in total. Each search method was made to solve the sample recording their average run-time, number of nodes expanded, and other data of interest. Run-time and nodes expanded were chosen as data points for analysis as they demonstrate the time and space complexity of each search method, while path length was chosen to demonstrate optimality.

**Search space generation** Each cell in a grid was iterated over in a random order, with walls having a 15% chance of being placed in each cell. Starting and ending positions were placed randomly, ensuring that no cells collided. As A\* has been demonstrated to be both *optimal* and *complete*, with the implementations correctness having been proven through static tests as outlined in on several pathfinding benchmarks, A\* was used to calculate the optimal path as a benchmark for all subsequent tests.

### 5.2.2 Results

As outlined in table 1, A\* and Greedy Best-First had the lowest mean run-times ( $\mu = 10.65$ ,  $\mu = 9.78$  respectively), while both iterative deepening methods had the highest. Furthermore

the higher run-times had the highest standard deviations ( $\sigma = 142.93$  and  $\sigma = 6.22$ ).

| method                          | Run-time (ms) |        |          |
|---------------------------------|---------------|--------|----------|
|                                 | Mean          | Median | Std. Dev |
| A*                              | 10.65         | 9.97   | 2.46     |
| Breadth-First                   | 11.16         | 10.14  | 2.91     |
| Depth-First                     | 10.89         | 10.20  | 2.42     |
| Greedy Best-First               | 9.78          | 9.56   | 1.54     |
| Iterative-Deepening A*          | 13.18         | 11.04  | 6.22     |
| Iterative-Deepening Depth-First | 74.41         | 16.37  | 142.93   |

Table 1: Averages and standard deviation for the run-time of each search method on all grids

Table 2 describes the average memory usage for each method, where GBFS and IDA\* expanded the lowest number of nodes on average ( $\mu = 114.76$  and  $\mu = 511.21$ ), while IDDFS expanded the highest number ( $\mu = 19493.45$ ), with the highest standard deviation ( $\sigma = 36364.81$ ).

| method                          | Number of nodes expanded |        |          |
|---------------------------------|--------------------------|--------|----------|
|                                 | Mean                     | Median | Std. Dev |
| A*                              | 542.93                   | 200.5  | 751.61   |
| Breadth-First                   | 835.93                   | 397.0  | 1020.19  |
| Depth-First                     | 814.13                   | 397.0  | 950.98   |
| Greedy Best-First               | 114.76                   | 49.0   | 336.36   |
| Iterative-Deepening A*          | 511.21                   | 177.0  | 736.04   |
| Iterative-Deepening Depth-First | 19493.45                 | 2910.0 | 36364.81 |

Table 2: Averages and standard deviation for the amount of nodes expanded for each search method on all grids

Figure 5 shows the distribution of run-times for different grid sizes for both IDA\* and IDDFS, demonstrating the locations of outliers for each method.

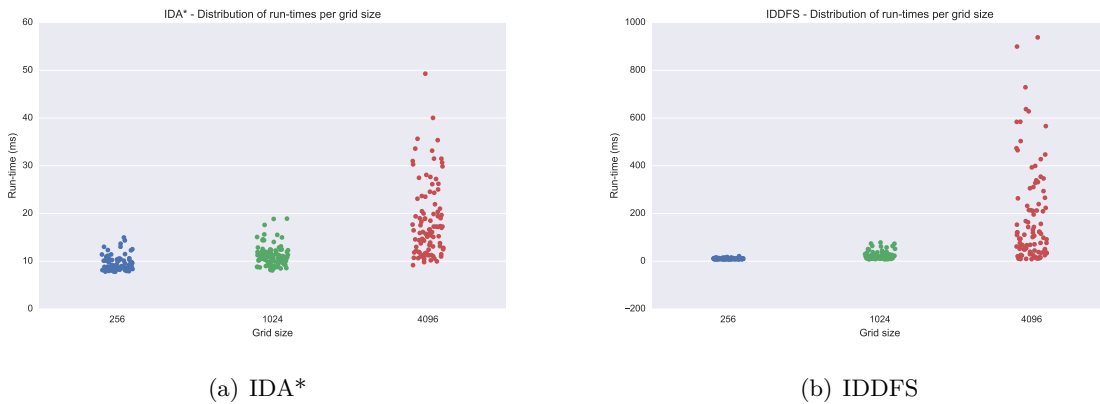


Figure 5: Distribution of run-times for each grid size for both IDA\* and IDDFS

Each search method had the same average path length ( $\mu = 27.94$ ), aside from GBFS which was slightly higher ( $\mu = 30.05$ ). DFS is excluded from the results here as its average path

length ( $\mu = 372.64$ ) was, as expected, too large of an outlier to be of much use for comparisons. Figure 6 shows the linear relationship between the number of walls present in a grid and the number of nodes expanded per search. Both IDA\* and IDDF are excluded from these results as neither method showed a significant relationship between these variables. Finally, while the number of solutions was recorded for each search method, the mean number of solutions found was the same for all methods ( $\mu = 92.47$ ).

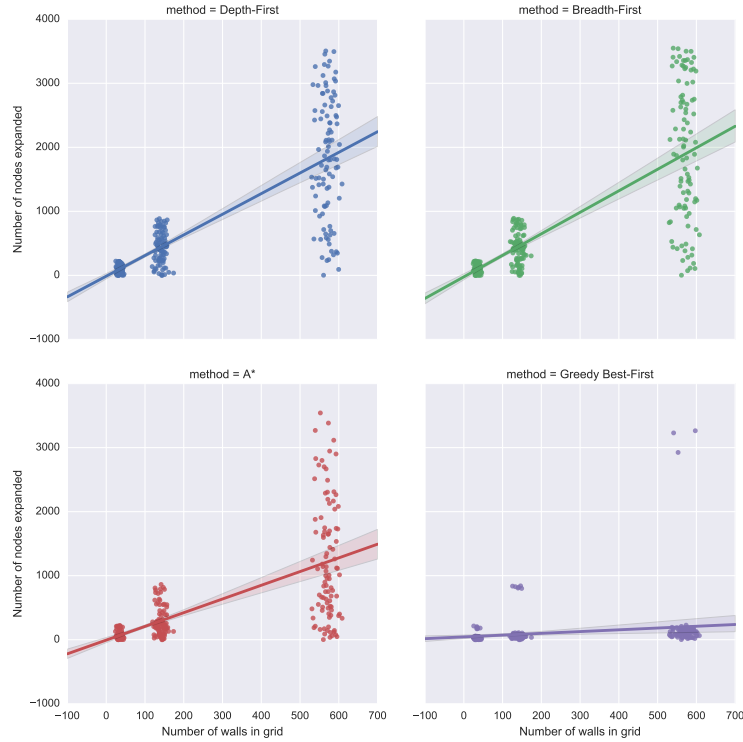


Figure 6: Linear relationship between number of walls in a grid and the number of nodes expanded

### 5.2.3 Discussion

As expected (discussed in section 2) each search method excluding DFS and GBFS proved *optimal* in all problems in which a solution existed. GBFS, while certainly non-optimal, had an average path length that proved not much longer than the optimal one with a lower standard deviation and proved to find solutions faster while expanding far fewer nodes than any other method, which indicates that although it may not always find the optimal solution for the RNP, it finds a *good-enough* solution reliably while using much less resources in doing so - possibly suited for problems where some kind of path, not necessarily the optimal one, is required in a short period of time.

Of the **optimal** search methods, A\* proved to have the most reliably efficient run-time whilst still remaining modest in terms of memory usage, demonstrating its usefulness as a general purpose search algorithm. Of note, is that IDA\* expanded less nodes on average while still maintaining a modest run-time, this could be of use for computing environments with limited memory, however for environments where memory is plentiful the trade off in run-time may be too great to make IDA\* worth utilizing. Finally, of the custom and iterative methods, IDDFS proved to be an outlier with both the amount of nodes expanded and its average run-time proved quite high. However, as highlighted in figure 5, IDDFS showed a few very high outliers for larger grids with the distribution of run-times existing at the lower end for other grid sizes,

which may indicate that while IDDFS is certainly optimal and can be efficient it may prove too unreliable to be of use for problems such as the RNP.

## 6 Conclusion

### 6.1 Improvements

The slower run-time performance of IDDFS could see some improvements by storing a list of the nodes currently existing in the given path, allowing it to avoid circular paths. One particular area of interest for the performance of each informed search method would be to investigate different heuristics - in the implementation outlined here, manhattan distance was used for calculating each methods heuristic, however other methods exist that could affect how each one expands the nodes in their search tree and in what order.

### 6.2 Final Thoughts

Overall, for the RNP, A\* is demonstrably the most complete, optimal, and efficient general purpose search algorithm, proving to have the lowest mean run-time and number of nodes expanded for optimal methods. For memory-limited environments, however IDA\* may be of use where run-time is perhaps less important. Finally, in terms of *uninformed* search methods, BFS showed to be the most overall efficient.

## References

- Cormen, Thomas H. et al. (2009). *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press. ISBN: 0262033844, 9780262033848.
- Hart, P. E., N. J. Nilsson, and B. Raphael (1968). "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". In: *IEEE Transactions on Systems Science and Cybernetics* 4.2, pp. 100–107. ISSN: 0536-1567. DOI: [10.1109/TSSC.1968.300136](https://doi.org/10.1109/TSSC.1968.300136).
- Jones, Tim (2008). *Artificial Intelligence: A Systems Approach*. 1st. USA: Jones and Bartlett Publishers, Inc. ISBN: 0763773379, 9780763773373.
- Korf, Richard E. (1985). "Depth-first Iterative-deepening: An Optimal Admissible Tree Search". In: *Artif. Intell.* 27.1, pp. 97–109. ISSN: 0004-3702. DOI: [10.1016/0004-3702\(85\)90084-0](https://doi.org/10.1016/0004-3702(85)90084-0). URL: [http://dx.doi.org/10.1016/0004-3702\(85\)90084-0](http://dx.doi.org/10.1016/0004-3702(85)90084-0).
- Patel, Amit (2014). *Introduction To A\**. URL: <http://www.redblobgames.com/pathfinding/a-star/introduction.html>.
- Russell, Stuart and Peter Norvig (2010). *Artificial Intelligence: A Modern Approach*. 3rd Edition. Edinburgh Gate, Harlow, Essex, England: Pearson Education Limited.

## Acronyms

**A\*** A\* Search.

**BFS** Breadth-First Search.

**DFS** Depth-First Search.

**FIFO** First-In, First-Out queue structure.

**GBFS** Greedy Best-First Search.

**IDA\*** Iterative-Deepening A\* Search.

**IDDFS** Iterative-Deepening Depth-First Search.

**LIFO** Last-In, First-Out queue structure.

## Glossary

**admissible** An admissible heuristic is one that never *over-estimates* the cost to get to the goal, meaning it's estimation is **never** higher than the **lowest** possible cost to get to the goal.

**complete** A search algorithm is called **complete** when it's guaranteed to find a solution as long as one exists.

**heuristic function** A function, denoted as  $h(n)$  which returns an *estimation* of the quality of a specific node  $n$ .

**informed search** Synonymous with *heuristic search*. A search algorithm that has domain-specific knowledge of the search space, such as the distance from the start or end states, that define a **heuristic function** function  $f(n)$  for evaluating the quality of a given node  $n$ .

**node** A generalized element of different types data structures that represents a single value or object of interest. It will usually contain a reference or pointer to its parent or adjacent nodes.

**optimal** A search algorithm is called **optimal** when it is guaranteed to find the solution with the lowest path cost that exists each time it's executed.

**RNP** The **Robot Navigation Problem (RNP)** is a **Task environment** in which a simulated robot attempts to find an optimal route to a given *goal node* in an  $N \times M$  grid of nodes, where  $N = \text{width}$ ,  $M = \text{height}$  and both are non-zero.

**task environment** In the context of a search problem, the *task environment* is the space and problem for which an agent attempts to find a solution.

**uninformed search** A search algorithm that has no information of the search space other than the ability to detect the type of node encountered (i.e. start, end, wall).