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 with ease independent of the operating system being used.

Contents

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. These algorithms are broadly divided into two categories - Uninformed search searches that contain no domain knowledge about the search space, and Informed search searches that make estimations about the relative quality of a given state. 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 $N = width$, $M = height$ 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 the 12 section. 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 4.2.1. Furthermore, while this section contains a high-level discussion of the qualities and drawbacks of each search method used, an analysis of their optimality, completeness, time, and space complexity is described in-depth in section 4 as a component of this reports research initiative.

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.

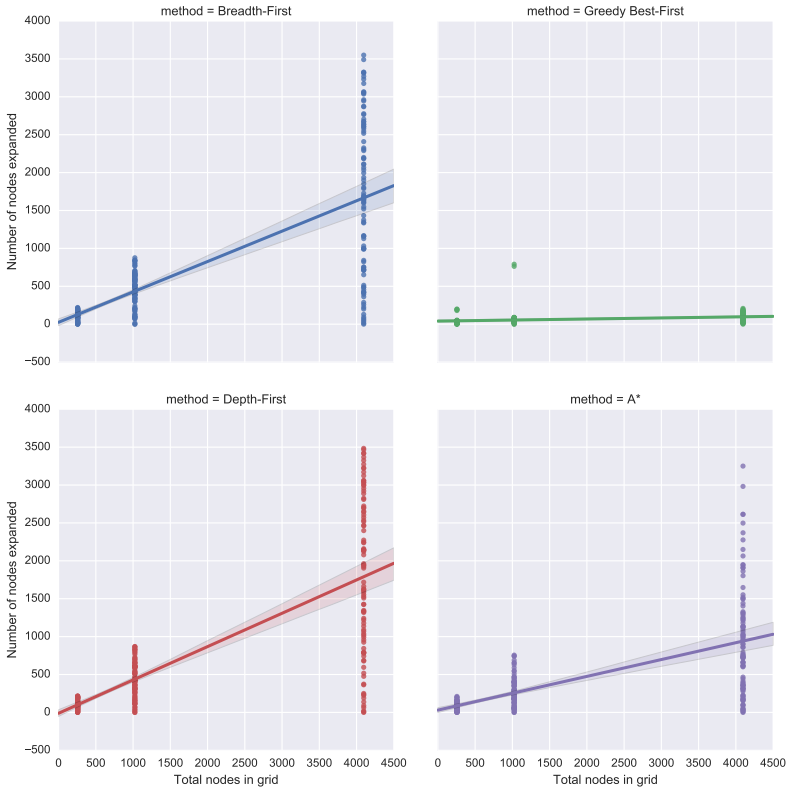
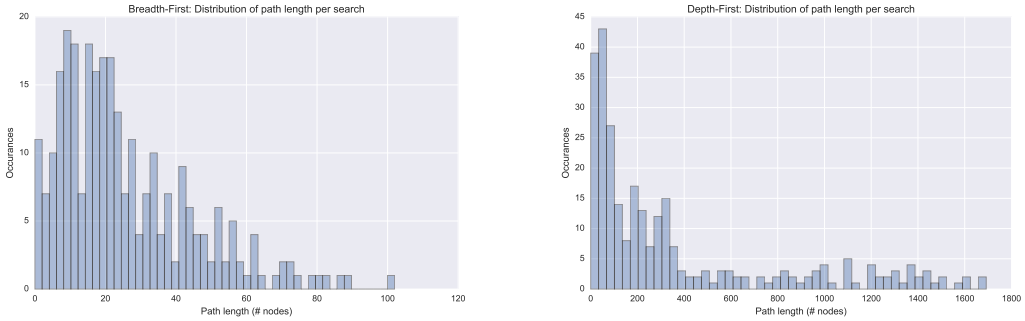


Figure 1: Linear relationship between size of RNP grid and the number of nodes expanded for each search algorithm

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 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 has many disadvantages. Firstly, it's non-optimal - for example, if DFS 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.



(a) BFS (b) DFS

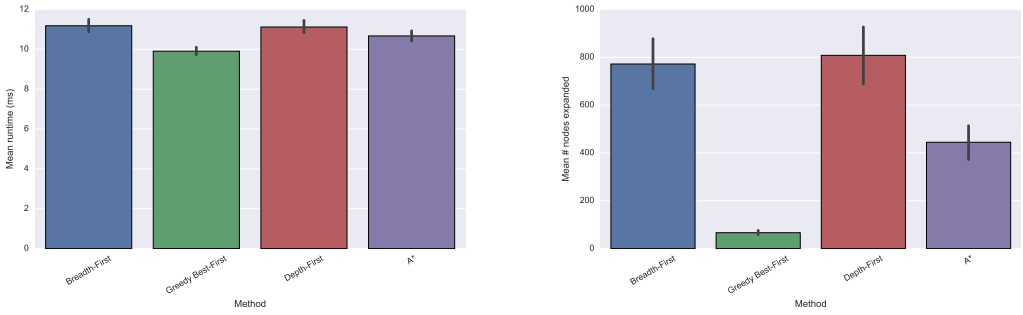
Figure 2: Comparison between the distributions of total path length for Breadth-First Search and Depth-First Search

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 runtime consequences, especially as the search space increases in size as demonstrated in figure

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 it's 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.



(a) Mean run-times

(b) Mean nodes expanded

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 ℓ is reached, signaling a cutoff upon which the search tree is cleared, increasing ℓ 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 ℓ for the current iteration. A cutoff is reached when the current nodes f cost exceeds the current ℓ value, upon which ℓ 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, IDA* cut down the mean amount of nodes expanded by approximately 50 nodes as demonstrated in figure 2.6.

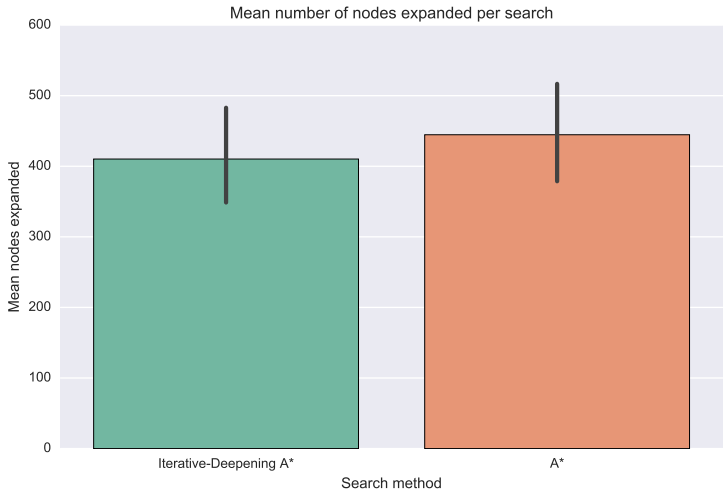


Figure 4: Comparison of mean number of nodes expanded per search between A* and IDA*

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 $800ms$ per search as shown in figure 2.6.

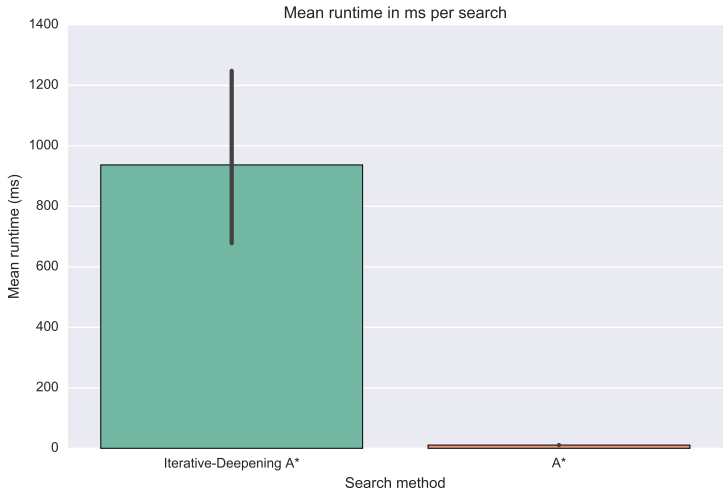


Figure 5: Comparison of mean run-time per search between A* and IDA*

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

At a

4. Research

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

4.1. 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 `-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.

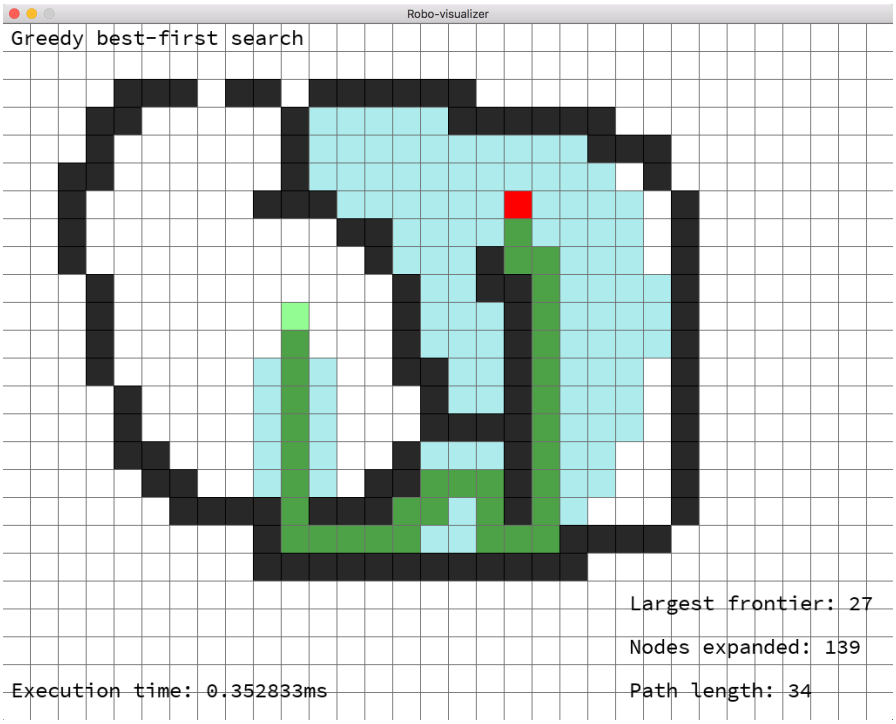


Figure 6: Screenshot of the GUI pathfinding visualizer

4.2. Search Method Analysis

While each search method has been described and analysed at a high level, this section analysis in-depth the time and space complexity of each method, and their effectiveness across different search environments. A method for random-generation of search problems with expected optimal paths for testing was developed for the purpose of testing a variety of environments.

4.2.1. Methodology

100 randomly generated search spaces were generated for grids of sizes 16×16, 32×32, and 64×64, 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 A method for generating search spaces and their optimal path, l_{opt}, was developed to generate data for the tests. Grids of sizes 16×16, 32×32, and 64×64 were 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 l_{opt} as a benchmark for all subsequent tests.

4.2.2. Results

As outlined in table 1, A* and Greedy Best-First had the lowest mean run-times ($\mu = 10.67$, $\mu = 9.90$ respectively), while both iterative deepening methods had the highest. Furthermore the higher run-times had the highest standard deviations ($\sigma = 2377.96$ and $\sigma = 99.89$).

Run-time (ms) ($N = 300$)			
method	Mean	Median	Std. Dev
A*	10.67	10.05	2.26
Breadth-First	11.18	10.44	2.68
Depth-First	11.12	10.25	2.60
Greedy Best-First	9.90	9.59	1.62
Iterative-Deepening A*	936.86	69.20	2377.96
Iterative-Deepening Depth-First	56.03	16.63	99.89

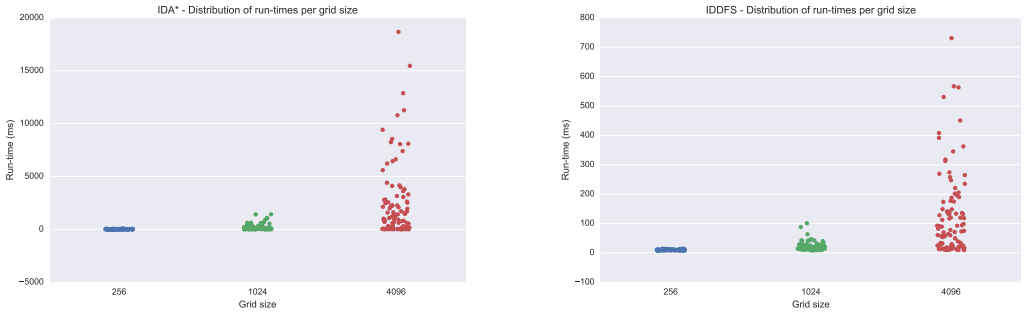
Table 1: Averages and standard deviation for the run-time of each search method on all grids

Table 2 describes the average memory usage for each method, where GBFS and IDA* expanded the lowest number of nodes on average ($\mu = 66.40$ and $\mu = 410.22$), while ID-DFS expanded the highest number ($\mu = 15035.24$), with the highest standard deviation ($\sigma = 27521.75$).

Number of nodes expanded ($N = 300$)			
method	Mean	Median	Std. Dev
A*	444.63	209	582.05
Breadth-First	771.89	422	915.50
Depth-First	808.05	387	964.84
Greedy Best-First	66.40	49	75.20
Iterative-Deepening A*	410.22	179	562.19
Iterative-Deepening Depth-First	15035.24	3320	27521.75

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

Figure 7 shows the distribution of run-times for different grid sizes for both IDA* and IDDFS, demonstrating the locations of outliers for each method.



(a) IDA*

(b) IDDFS

Figure 7: Distribution of run-times for each grid size for both IDA* and IDDFS

Each search method had the same average path length ($\mu = 25.59$), aside from GBFS which was slightly higher ($\mu = 26.15$), shown in figure 8. DFS is excluded from the results here as its average path length ($\mu = 369.70$) was, as expected, too large of an outlier to be of much use for comparisons.

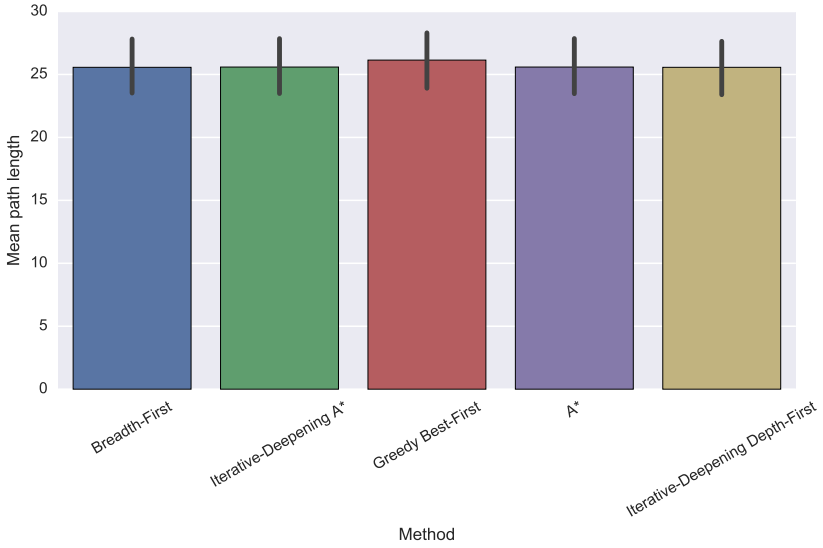


Figure 8: Average path length for all search methods excluding DFS

Figures 9 and 10 show the linear relationship between the number of walls present in a grid and the run-time and number of nodes expanded per search. Both IDA* and IDDF

are excluded from these results as neither method had any sign of relationship between these variables.

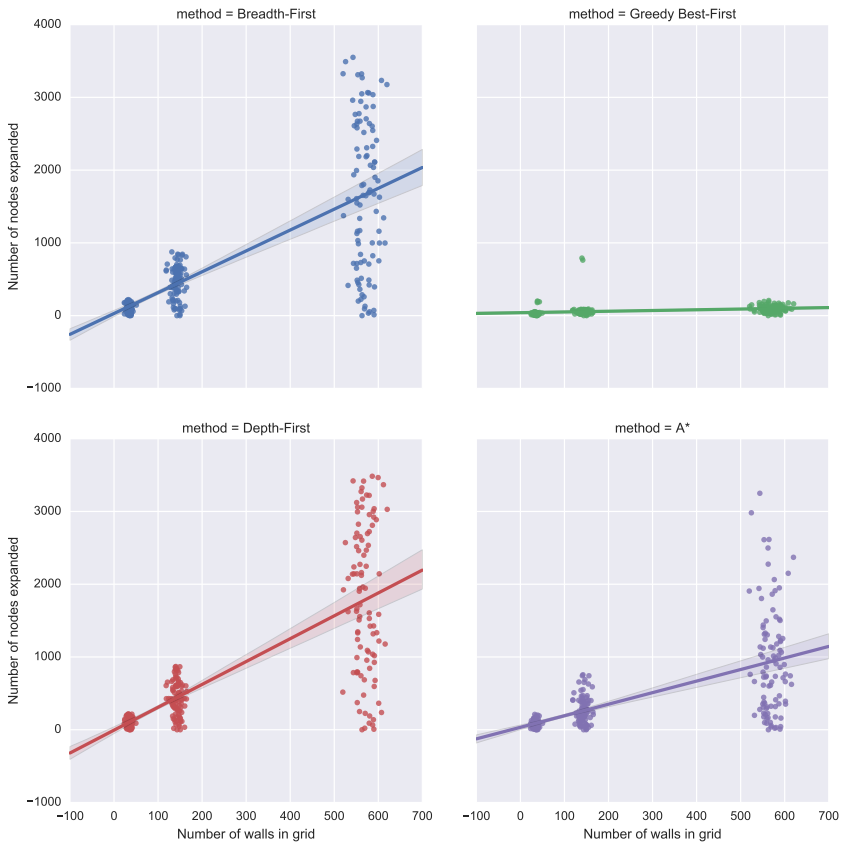


Figure 9: Linear relationship between number of walls in a grid and the number of nodes expanded

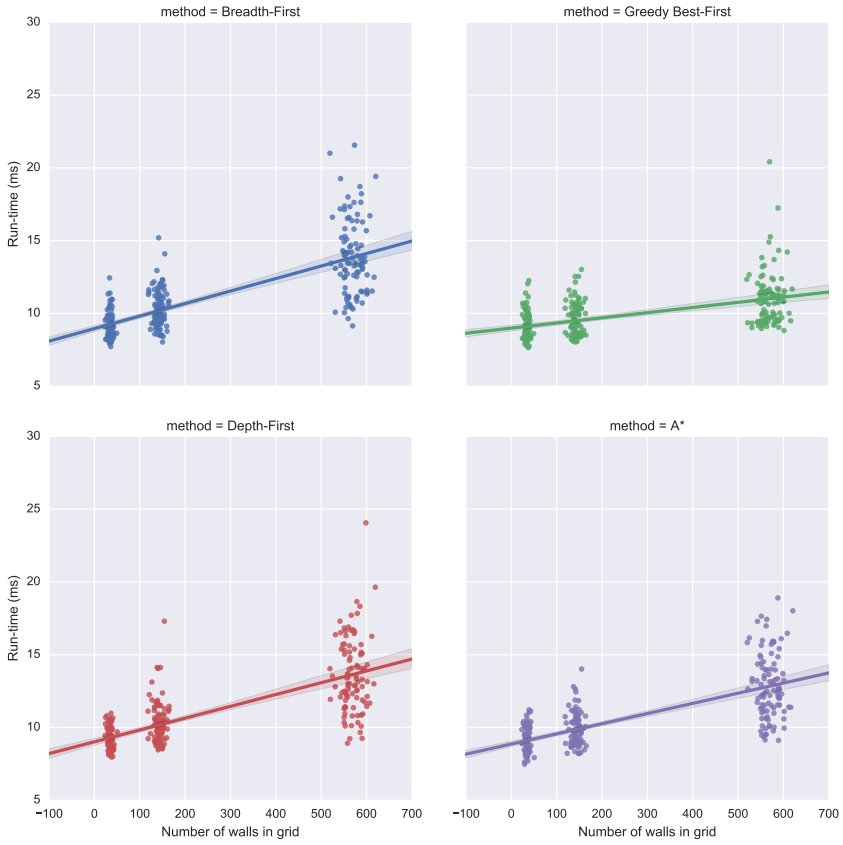


Figure 10: Linear relationship between number of walls in a grid and each search methods run-time

Finally, while the number of solutions was recorded for each search method, the mean number of solutions found was the same for all methods ($\mu = 275.65$).

4.2.3. Discussion

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.

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. URL: 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 that 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.

leaf A node in a tree data structure that has no children, i.e. a terminal state.

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 it's 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 $N = width$, $M = height$ and both are non-zero.

root node The node at the start of a tree data structure from which all children extend, usually denoted by removing its parent reference, i.e. making it equivalent to null.

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.

tree An abstract data type that represents a hierarchy of nodes extending from a single root node node.

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).

Appendix A Static tests

Static test environments

```
# test1.txt

[5,11]
(0,1)
```

```
(10,3)
(2,0,2,2)
(8,0,1,2)
(10,0,1,1)
(2,3,1,2)
(3,4,3,1)
(9,3,1,1)
(8,4,2,1)

 # test2.txt

[25,32]
(0,1)
(30,24)
(2,0,2,2)
(8,0,1,2)
(10,0,1,1)
(2,3,1,2)
(3,4,3,1)
(9,3,1,1)
(8,4,2,1)
(10,10,3,6)
(7,3,3,6)
(17,13,2,2)
(13,16,2,2)
(25,20,2,5)
(25,18,5,2)

 # test3.txt

[1000,1000]
(0,1)
(800,800)
(30,20,2,2)
(10,0,20,20)
(5,0,5,5)

 # test4.txt

[1000,1000]
(0,0)
(234,800)
```

```
 # test5.txt

[1000,1000]
(0,0)
(543,345)
(2,0,2,2)
(8,0,1,2)
(10,0,1,1)
(2,3,1,2)
(3,4,3,1)
(9,3,1,1)
(8,4,2,1)
(10,10,3,6)
(7,3,3,6)
(17,13,2,2)
(13,16,2,2)
(25,20,2,5)
(25,18,5,2)
(42,0,2,2)
(48,0,1,2)
(410,0,1,1)
(42,3,1,2)
(43,4,3,1)
(49,3,1,1)
(48,4,2,1)
(410,10,3,6)
(47,3,3,6)
(417,13,2,2)
(413,16,2,2)
(425,20,2,5)
(425,18,5,2)
(540,340,10,1)
(540,340,2,50)
(544,340,2,50)
(62,0,2,2)
(68,0,1,2)
(610,0,1,1)
(62,3,1,2)
(63,4,3,1)
(69,3,1,1)
(68,4,2,1)
```

```
(610,10,3,6)
(67,3,3,6)
(617,13,2,2)
(613,16,2,2)
(625,20,2,5)
(625,18,5,2)

 # test6.txt

[25,32]
(0,1)
(10,9)
(2,0,2,2)
(8,0,1,2)
(10,0,1,1)
(2,3,1,2)
(3,4,3,1)
(9,3,1,1)
```

Source code for static test cases

```
#include "Search/Core/SearchMethod.hpp"
#include "Parsers/FileParser.hpp"

#include "Search/Methods/BreadthFirst.hpp"
#include "Search/Methods/DepthFirst.hpp"
#include "Search/Methods/GreedyBestFirst.hpp"
#include "Search/Methods/AStar.hpp"

#define CATCH_CONFIG_RUNNER
#include <catch.hpp>
#include <Path.hpp>

#include <random>

struct SearchTestCase {
    std::string file;
    int expected_length;
};
```

```
class SearchTestsFixture {
public:
    static sky::Path root_;

    SearchTestsFixture()
        : methods_(robo::generate_method_map()),
          parser_("SearchTests")
    {}

protected:
    robo::MethodMap methods_;
    robo::FileParser parser_;

    std::vector<SearchTestCase> test_cases_;
};

sky::Path SearchTestsFixture::root_ = "";

int main(int argc, char** argv)
{
    SearchTestsFixture::root_.assign(sky::Path::bin_path(argv));
    SearchTestsFixture::root_.append("../Tests");



    int result = Catch::Session().run(argc, argv);

    return ( result < 0xff ? result : 0xff );
}

TEST_CASE_METHOD(SearchTestsFixture,
                 "Search methods fail when goal can't be found", "[failure]")
{
    auto env = parser_.parse(root_.get_relative("failure.txt"));
    robo::Solution solution;

    for ( auto& m : methods_ ) {
        solution = m.second->search(env);
        REQUIRE_FALSE(solution.success);
    }
}
```

```
TEST_CASE_METHOD(SearchTestsFixture, "A* is optimal",
                 "[as]")
{
    test_cases_ = {
        { root_.get_relative("test1.txt"), 12 },
        { root_.get_relative("test2.txt"), 53 },
        { root_.get_relative("test3.txt"), 1599 },
        { root_.get_relative("test4.txt"), 1034 },
        { root_.get_relative("test5.txt"), 978 },
        { root_.get_relative("test6.txt"), 18 },
    };

    auto env = parser_.parse(test_cases_[4].file);

    robo::Solution solution;
    for ( auto& t : test_cases_ ) {
        auto env = parser_.parse(t.file);
        REQUIRE( (env.size().x > 0 && env.size().y > 0) );
        solution = methods_["AS"]->search(env);
        REQUIRE(solution.success);
        REQUIRE(solution.path.size() == t.expected_length);
    }
}

TEST_CASE_METHOD(SearchTestsFixture, "IDS Search", "[ids]")
{
    std::random_device seeder;
    std::mt19937 engine(seeder());
    std::uniform_int_distribution<int> dist(0, 63);
    for ( int i = 0; i < 10; ++i ) {
        robo::Environment env(64, 64);
        env.start = robo::Point(dist(engine), dist(engine));
        env.goal = robo::Point(dist(engine), dist(engine));
        auto lim = dist(engine);

        for ( int j = 0; j < lim; ++j ) {
            auto point = robo::Point(dist(engine), dist(engine));
            if ( env[point.x][point.y] == robo::Cell::empty ) {
                env.set_cell(point.x, point.y, robo::Cell::wall);
            }
        }
```

```
        auto expected = methods_["AS"]->search(env);
        auto actual = methods_["CUS1"]->search(env);

        SECTION("IDS is complete")
        {
            REQUIRE(expected.success == actual.success);
        }

        SECTION("IDS is optimal")
        {
            if ( expected.success )
                REQUIRE(expected.path.size() == actual.path.size());
        }
    }
}

TEST_CASE_METHOD(SearchTestsFixture, "IDA* Search", "[ida]")
{
    std::random_device seeder;
    std::mt19937 engine(seeder());
    std::uniform_int_distribution<int> dist(0, 31);
    for ( int i = 0; i < 10; ++i ) {
        robo::Environment env(32, 32);
        env.start = robo::Point(dist(engine), dist(engine));
        env.goal = robo::Point(dist(engine), dist(engine));
        auto lim = dist(engine);

        for ( int j = 0; j < lim; ++j ) {
            auto point = robo::Point(dist(engine), dist(engine));
            if ( env[point.x][point.y] == robo::Cell::empty ) {
                env.set_cell(point.x, point.y, robo::Cell::wall);
            }
        }

        auto expected = methods_["AS"]->search(env);
        auto actual = methods_["CUS2"]->search(env);

        SECTION("IDA* is complete")
        {
            REQUIRE(expected.success == actual.success);
        }
```

```
        SECTION("IDA* is optimal")
        {
            if ( expected.success )
                REQUIRE(expected.path.size() == actual.path.size());
        }
    }
}
```