Proc. of the Second Intl. Conf. on Advances in Computer and Information Technology -- ACIT 2013
Copyright © Institute of Research Engineers and Doctors. All rights reserved.
ISBN: 978-981-07-6261-2

The Boundary Iterative-Deepening Depth-First Search Algorithm

Lim Kai Li, K.P. Seng, L.S. Yeong, S. I. Ch'ng
Department of Computer Science and Networked Systems
Sunway University
Petaling Jaya, Selangor, Malaysia

Ang Li-Minn
School of Engineering
Edith Cowan University
Joondalup, Western Australia

***Abstract*—Boundary searches were introduced in pathfinding aiming to find a middle-ground between memory intensive algorithms such as the A* search algorithm and the cycle redundancy of iterative-deepening algorithms such as the IDA*. Boundary search algorithms allocate a small memory footprint during runtime to store frontier nodes between each iteration to reduce redundancy, while expanding nodes in the same manner as iterative-deepening algorithms. The boundary search algorithm fringe search is an informed search algorithm derived from the IDA* for use in known environments. This paper proposes the boundary iterative-deepening depth-first search (BIDDFS) algorithm, which fills the gap made by the fringe search for uninformed search algorithms. The BIDDFS is optimised to perform blind searches in unknown environments, where simulation experiments found that it is up to more than 3 times faster than standard uninformed iterative-deepening algorithms.**

***Keywords*—pathfinding, uninformed search, iterative-deepening, boundary search**

I. Introduction

Pathfinding in computing is often described as the plotting by a computer application to find the best route between two points. In everyday life, a pathfinder generally finds routes between points in a physical environment such as a landscape, a map or a terrain. Pathfinding results are most often used in navigation. This often results in movement between the points on the route. Furthermore, due to the ubiquitous requirements for pathfinding, it is worth implying that pathfinding is supposed to generate an optimum route, although there are often factors that also prevent it from achieving so, such as computer limitations and terrain difficulties.

According to [1], the efficiency of a pathfinding algorithm can be classified to its completeness, where it is guaranteed to a route solution if it exists; its optimality, where it is able to provide the optimal solution; its time complexity, for how much time is taken for pathfinding; and its space complexity, for the amount of memory required to compute a route.

Pathfinding algorithms are classified into uninformed and informed search [1]. An informed search performs in an environment where its location information is known; this means that the algorithm is able to use this location information to identify the paths and obstacles of an environment (heuristics). On the other hand, uninformed searches are performed where environmental data is not known. The computing of the route uses forward planning, where it anticipates data from future iterations (aka blind search), Uninformed search algorithms often need to discover the environment before a proper path can be defined [2]. This is the cause of it being less efficient than the informed search since paths can be chosen that will not eventually lead to the ending point. An informed search will not encounter this problem.

With regards to the more well-known pathfinding algorithms, they are often considered to be flexible and efficient. Algorithms such as the A* (pronounced "A-star") search algorithm [3] and the Dijkstra's shortest path algorithm [4] are often used interchangeably in situations, depending on the requirements that the current situation holds, as both algorithms, along with other unmentioned algorithms are made to cater for different pathfinding scenarios.

Starting with the A* algorithm, it is considered to be one of the most established and well-known pathfinding algorithms around. Its flexibility and capability to compute the optimum route is highly favoured, this is also balanced with its high performance capability that is contributed by its improvements over older algorithms such as the greedy algorithm, and the Dijkstra's algorithm.

General pathfinding algorithms such as the Dijkstra's algorithm and the A* search algorithm performs well in many situations, but suffers from memory issues especially when faced with larger maps. From here, an iterative-deepening search was introduced to the algorithms, transforming them into the iterative-deepening depth first search (IDDFS) [1] and the Iterative-deepening A* (IDA*) [5] search algorithm. Iterative-deepening searches are intended to give the search algorithm a significantly smaller

Proc. of the Second Intl. Conf. on Advances in Computer and Information Technology -- ACIT 2013
Copyright © Institute of Research Engineers and Doctors. All rights reserved.
ISBN: 978-981-07-6261-2

memory footprint, often but not necessarily, at the expense of a longer runtime.

To achieve a balanced compensation between runtime speed and memory consumption, the fringe search was introduced, deriving from the IDA* search algorithm. Based on the specifications of the fringe search and the original IDDFS algorithm, a new algorithm was proposed to accommodate fringe searching in uninformed environments, dubbed the Boundary IDDFS (BIDDFS), for the word 'boundary' provides a more accurate representation of the search algorithm.

Six search algorithms were identified for this research with increasing depth and complexity. These algorithms are categorised into uninformed and informed searches, iterative deepening searches, and fringe searches [6], as illustrated in Table I.

TABLE I: PATHFINDING ALGORITHMS

	Uninformed	**Informed**
Pathfinding	Dijkstra's	A*, Greedy BFS
Iterative-Deepening	IDDFS	IDA*
Boundary	BIDDFS	Fringe Search

For the analysis of the algorithms, each algorithm will be run and compared on the same, randomly generated grid map environment. The differences between each algorithm are presented in each of their logical structure. Subsequently, their efficiencies and memory footprint will be analysed in the following section.

The term 'node' is used rather frequently in this paper. A node here is defined as a single grid box from the map illustrations of Section V. Nodes can be made as a starting node (green dot), an ending node (yellow dot), or an OPEN or CLOSED node, where the latter will be further explained in the next section.

Section II in this paper explains the iterative-deepening searches, and the general logic behind these algorithms. Section III describes boundary searches, a subset of iterative-deepening searches that stores boundary nodes before each new iteration for faster runtimes. Section IV proposes a new algorithm, the boundary iterative-deepening depth-first search algorithm, to introduce boundary searches in uninformed environments. Section V will involve simulation examples, where the BIDDFS will be compared against other algorithms in different map environments. Finally, Section VI presents the conclusion and future works.

II. ITERATIVE-DEEPENING SEARCHES

These algorithms were devised to address the issues of large memory footprints of their preceding algorithms, especially in large maps, where the memory requirement for each runtime could increase exponentially. For the implementation of this research, an iterative-deepening algorithm was identified for one each of the uninformed and informed search. As such, the iterative-deepening depth-first search (IDDFS) addresses the memory shortcomings of the Dijkstra's algorithm and likewise for the IDA* search algorithm addressing the memory issues of the A* search algorithm.

Here, the main contribution to the large memory footprint of the algorithms – the OPEN and CLOSED sets of each node, are removed, leaving only the heuristic and cost information of the neighbouring, OPEN nodes. Once a node has been expanded, where normally a Dijkstra's algorithm or A* search algorithm implementation will update these nodes to a CLOSED state, iterative-deepening algorithms will not store any information regarding these nodes, and hence "forgetting" these nodes entirely, hence saving significant amounts of memory.

In other words, iterative-deepening algorithms are implemented here as forward, progressive algorithms. By way of implementation, these algorithms rely on the existence of "thresholds". Since it is not possible for the algorithm to remember the nodes it has expanded, the "threshold" governs the expansion boundary of each runtime, for the lack of memory regarding expanded nodes will lead to overexpansion. If a goal is not found at a specific threshold, the pathfinding process will terminate, the threshold will be increase, and the process starts again. Hence with the implementation of these algorithms, a double loop will be used – one for the standard process of the search algorithm, and another to increase the threshold and restart the runtime for each run that fails to reach the goal.

Indeed, at each reset of the runtime with a threshold increment, the nodes expanded in the previous runtime must be expanded again. To ease explanations, the entire pathfinding runtime of the algorithm can be separated into different sub-runtimes at different thresholds. Therefore, the total runtime for an iterative-deepening algorithm may actually be greater than its predecessor algorithm, especially in smaller maps. Yet, when used in large environments, the small memory footprint of these algorithms will in fact result in a faster runtime than the predecessor algorithms.

III. BOUNDARY SEARCHES

Boundary searches were conceptualised due to the need of reducing the search redundancy of iterative-deepening search algorithms, while keeping the memory requirements of the algorithm low. In fact, the only known boundary search algorithm in literature that aims to address these issues is the Fringe Search algorithm. In this section, the word "boundary" is used instead of "fringe" to give a more accurate representation of the algorithm, and the minimise confusion between the Fringe Search and the forthcoming, new algorithm.

The main idea behinds boundary searches is rather simple – to prevent the need of re-expanding the entire search map every time the algorithm resets for a threshold increment. Since it was mentioned that iterative-deepening searches do not store information regarding CLOSED nodes, and it simply forgets them, the proposed solution was to save the OPEN nodes of the last iteration into the memory before the algorithm resets and increase its threshold, these nodes are called the leaf nodes, the boundary nodes, or the frontier nodes. Hence, like the iterative-deepening algorithms, the

Proc. of the Second Intl. Conf. on Advances in Computer and Information Technology -- ACIT 2013
Copyright © Institute of Research Engineers and Doctors. All rights reserved.
ISBN: 978-981-07-6261-2

usage of an incremental threshold and the lack of data regarding expanded nodes, is still present in these algorithms. Furthermore, the need of storing the last OPEN nodes in the memory also means that boundary searches does indeed require a greater memory footprint than that of iterative-deepening algorithms. Yet, since only information regarding the "boundary" is stored in the memory, (unlike the Dijkstra's or A* search algorithm, which stores information about all expanded nodes in the memory), its memory footprint is still significantly lower than the A* and the Dijkstra's search algorithm.

Performance wise, boundary searches are suitable for use in medium to large-scaled maps. While a single sub-runtime, boundary searches still performs much faster than the standard A* or Dijkstra's search algorithms. However, boundary searches, at a single threshold, is still slower in node expansion as compared to iterative-deepening searches, since the algorithm has to load its boundaries into the memory.

Shifting perspectives onto the entire runtime however, boundary searches do not re-expand nodes from the beginning, and hence in many cases it is able to perform faster than iterative-deepening searches, especially in medium to large map environments. This means that instead of re-expanding the whole map at every threshold increment, only the boundary nodes are re-expanded. While this does not completely eliminate the redundancy issues in iterative deepening searches, these algorithms are able to keep it at a minimum.

IV. Boundary Iterative-Deepening Depth-First Search

The BIDDFS is a newly proposed algorithm aiming to address the same issues the fringe search did with the IDA*, this time with the IDDFS algorithm. Its main concept utilises the fringe search described in the previous section and it is modified to address the redundancy issues of the IDDFS algorithm. Hence, while the fringe search explores a middle-ground between the A* and the IDA* search algorithms; the BIDDFS now explores the middle-ground between the IDDFS and the Dijkstra's algorithm.

That said, the BIDDFS is an uninformed search algorithm, and similar to the application above, it maintains only the "now" list to store information of the frontier nodes. Essentially, while its main search procedure is a boundary search algorithm, its other pathfinding procedures were derived from the IDDFS and hence the Dijkstra's algorithm. In other words, it should be noted that first, the expansion pattern for this algorithm, along with the IDDFS and the Dijkstra's algorithm, should be the same, so long as the map and cost environment remains the same. Second, since this algorithm is indirectly based off the Dijkstra's algorithm, this algorithm, along with all other algorithms discussed in this section (except the greedy BFS) should return the same, shortest path back to the starting node from the goal node.

Being an uninformed search algorithm, the BIDDFS is essentially a fringe search algorithm without the heuristic data that makes it an informed search algorithm. Like the fringe search, the BIDDFS also operates using a threshold to compensate for the lack of memory, when a sub-runtime reaches its boundary, the frontier nodes are saved into the memory and accessed when the threshold increases and the search process restarts. Essentially, this algorithm follows the procedure below:

1. Increasing threshold by 1
    a. Calculating the cost of surrounding nodes from the location node.
    b. Update surrounding OPEN nodes' cost.
    c. Assign costs and pointers for neighbouring nodes.
    d. OPEN a new, neighbouring node as the location node, if available. Otherwise check for better route to it.
2. Save frontier nodes in memory and define starting position.

Hence, the pseudocode below summarises the logic behind the BIDDFS:

```
init
    boundary B = s
    cache C[beginning] = (0, null)
    for n in C, n != beginning
        C[n] = null
    threshold = h(start)
    reachedgoal = false

    while NOT goal=true AND B NOT empty
        fmin = ∞
        for n in F, from left to right
            (g, parent) = C[n]
            if g > threshold
                fmin = min(g, fmin)
                continue
            if n = goal
                reachedgoal = true
                break
            for s in children(n), from right to left
                g(s) = g + cost(n, s)
                if C[s] != null
                    (g', parent) = C[s]
                    if g(s) >= g'
                        continue
                if s in B
                    remove s from B
                insert s in F past n
                C[s] = (g(s), n)
            remove n from B
        threshold = fmin

    if reachedgoal = true
        make path from cache
```
Figure 1: BIDDFS Pseudocode

Performance wise, the BIDDFS is able to show significant improvements in efficiency over its predecessor, the IDDFS algorithm, especially in small to medium-large maps. This is credited to the usage of the small memory allocation to store information for the frontier nodes before the threshold increment reset process, which will be illustrated in section 4 below. That said, like the IDA*, the BIDDFS is designed to be a situational algorithm; it depending on the method of implementation, or may or may not yield better results than the algorithm it is compared to.

Proc. of the Second Intl. Conf. on Advances in Computer and Information Technology -- ACIT 2013
Copyright © Institute of Research Engineers and Doctors. All rights reserved.
ISBN: 978-981-07-6261-2

For example, if a map is small enough, a typical Dijkstra's search algorithm will be able to solve the map faster than the BIDDFS, due to its requirement for threshold resets, even on small maps. On the other hand, if a map is too large, the IDDFS would perform faster than the BIDDFS. This is because in large maps, the number of frontier nodes will be greater, and hence more time would be needed to store the large amounts of data in the memory. Though, this algorithm will still be faster than the Dijkstra's algorithm as it needs to save even more data into its memory.

Then again, the method of simulation would also contribute rather greatly to the eventual performance of the algorithm. For instance, the simulation method described in section 4 below requires the figure to be closed and reopened for to clear the map. This alone undermines the ability of the algorithm to perform ideally as the opening and closing of the figure drawing the map, coupled with the need for the process to redraw the map, causes significant delays to the algorithm, and also other similar algorithms.

All in all, the theoretical argument and purpose of this algorithm is to find an efficient and less redundant uninformed search for use in unknown environments, a middle-ground between the Dijkstra's algorithm and the IDDFS algorithm. Furthermore, algorithms like these are able to cater for dynamically changing algorithms by including an adjustment function to adjust the map properties in between thresholds, and so long as that adjusted node has not been expanded, it will be expanded properly as soon as it becomes the frontier node. Once a goal has been found, the algorithm will follow the same methods used in the Dijkstra's algorithm to route the resultant route from the goal back to the starting node.

V. Simulation Examples

The algorithms listed in Table I were analysed for their time and memory efficiencies. These algorithms are tested on the same grid map environment of varying size and number of obstacles. All analyses are programmed on MATLAB. Mapping illustrations are based on the source codes by Bob L. Sturm [7], which also performs route calculations. The IDA*, IDDFS, fringe search, and BIDDFS algorithms were programmed individually and independently to run on Strum's mapping framework.

For each simulation, the map size will be determined by a square field of size *n* by *n* blocks, and walls will be placed at random blocks until it constitutes to a percentage of a map as determined by the wall percentage. Then, simulation results will be posted, starting from the original pathfinding algorithms, the iterative-deepening algorithms, and then the boundary algorithms.

For each result set, a screen capture of the map and its simulation time will be recorded, noted by the algorithms' self-time feature in the MATLAB profiler. With reference to the screen capture of the maps, blocks of increasing blue hues represent the block's proximity to the starting node whilst blocks of increasing red hues represent the block's proximity to the goal node. The starting node is marked with a green dot and the goal node is marked with a yellow dot. The grey tracing connecting both the starting and goal node is the resulting path returned by each algorithm. Simulation times are obtained by clicking the "Run and Time" button on respective code editor windows. Each figure in the following examples are screen captures of the map representing the search pattern of an uninformed search algorithm, e.g. the BIDDFS.

With regards to the threshold of each iterative-deepening algorithm, the threshold is set to display during runtime where it is incremented after each sub-runtime that fails to find the goal node. This value will then be tabulated in the results once the runtime ends. A greater threshold means more nodes are expanded for pathfinding, which can also indicate a greater distance between the starting node and the goal node.

All simulations are performed using the 64-bit version of Mathworks MATLAB R2012b running on Microsoft Windows 8 Pro x64. The CPU used is an Intel Core i7 3770K processor at 3.50 GHz, with 16GB of DDR3 RAM, graphics are handled by the NVidia GeForce GTX 670 GPU.

A. Example 1

- Field size: 10 by 10
- Wall Percentage: 40%

The first simulation example begins with a small-scaled map, demonstrating the algorithms performance in a 10 by 10 block map. Here, the direct path to the goal from the starting node is blocked and the algorithm has to route across it to map a path. It should be noted that the simulations here were performed in extremely short amounts of time.

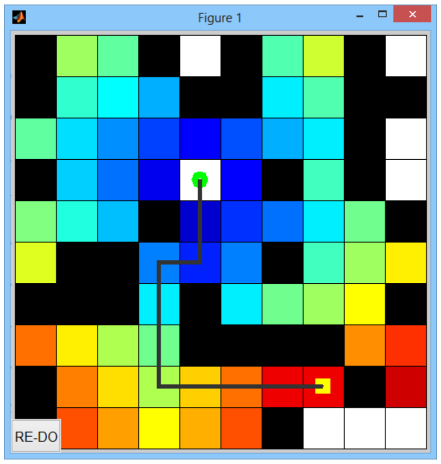

Figure 2: Search pattern

TABLE II: EXAMPLE 1 RESULTS

Algorithm	**Time Taken /s**	**Threshold**
IDDFS	0.216	60
IDA*	0.114	37
Fringe Search	0.099	37
BIDDFS	0.137	60

Proc. of the Second Intl. Conf. on Advances in Computer and Information Technology -- ACIT 2013
Copyright © Institute of Research Engineers and Doctors. All rights reserved.
ISBN: 978-981-07-6261-2

Based on the results, it can be seen that the IDDFS, being an uninformed algorithm, runs longer compared to the IDA*, which is an informed search algorithm. The IDDFS has a threshold of 60, which means that the starting node needs to be expanded 60 times for it to reach the goal. It is evident that boundary searches are faster than iterative-deepening searches, thanks to the usage of memory to store frontier nodes between threshold increments. However, the nature of an informed search of the BIDDFS still makes it slower compared to the fringe search, which is an informed search algorithm.

B. *Example 2*

- Field size: 70 by 70
- Wall percentage: 40%

To show the radial node expansion of uninformed search algorithms, a medium-large map will be used for this example. To ensure consistency, the walls here are still kept at 40%, but the size of the map is now increased to 70 by 70 blocks. Then again, a 40% wall ratio in this map translates to an entirely different scenario for the algorithms. This wall ratio means that the obstacles will appear more scarce than the 10 by 10 grid above; yet, is it enough to prevent a diagonally straight path from connecting the starting node to the goal.

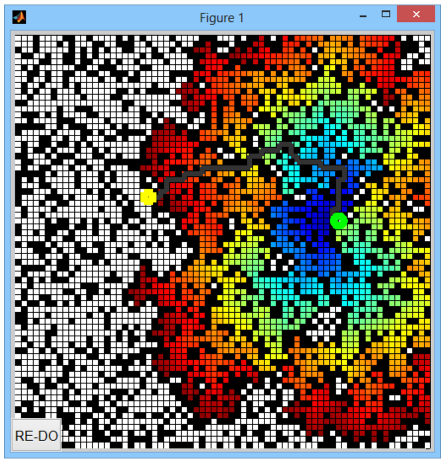

Figure 3: Search pattern

TABLE III: EXAMPLE 2 RESULTS

Algorithm	**Time Taken /s**	**Threshold**
IDDFS	377.165	1815
IDA*	15.552	545
Fringe Search	14.975	545
BIDDFS	103.622	1815

With a larger map, iterative-deepening algorithms now take significantly longer times to run than example 1. Still, the distance between the starting and ending node is only approximately half of the map's length, which would suggest that more time would be needed for such a runtime, considering that in this example, approximately 40% of the map has not been expanded. Nonetheless, the expansion pattern observed here shows that the nodes are expanded radially, since no heuristic data was relied on during an uninformed search.

To offer a fair comparison, the boundary algorithms are compared against the standard iterative-deepening algorithms. When boundary searches are used instead of the iterative-deepening searches, the number of thresholds remain the same. However, because the frontier nodes are saved in the memory between each threshold increment, the total runtime of each algorithm is vastly faster than their iterative-deepening predecessors.

Here, the time taken for the simulation for the BIDDFS is 3 times faster than the IDDFS. As mentioned previously, this effect of a larger map brings exponential runtime differences, as many more nodes will be expanded for pathfinding.

C. *Example 3*

- Field size: 25 by 25
- Wall percentage: 20%

By decreasing the wall percentage of the map, the results should produce a more linear path from the starting node to the goal. To allow a more reasonable simulation time, the map size has been reduced to a small-medium size of 25 by 25 blocks.

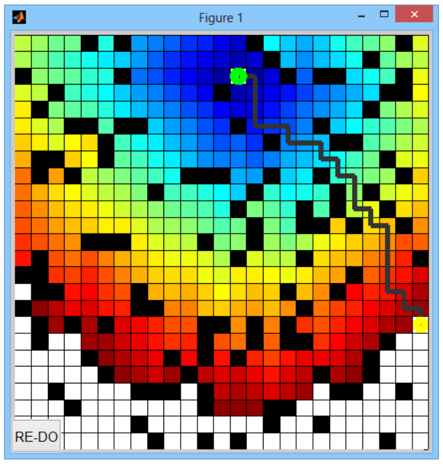

Figure 4: Search pattern

TABLE IV: EXAMPLE 3 RESULTS

Algorithm	**Time Taken /s**	**Threshold**
IDDFS	7.701	404
IDA*	1.700	177
Fringe Search	1.679	177
BIDDFS	5.092	404

A smaller map from example 2 means a smaller simulation time and hence a smaller threshold. Still, the simulation times for these algorithm is still quite long compared to the first example, though it has indeed seen significant improvements from a larger map.

Proc. of the Second Intl. Conf. on Advances in Computer and Information Technology -- ACIT 2013
Copyright © Institute of Research Engineers and Doctors. All rights reserved.
ISBN: 978-981-07-6261-2

Like the previous example, the boundary searches here shows significant time improvements over their predecessors in the iterative-deepening searches. The threshold remains the same and the time improvements is credited to the small amount of memory used to save the frontier nodes before every threshold increment.

D. *Example 4*

- Field size: 25 by 25
- Wall percentage: 60%

Increasing the wall percentage from example 3 yields a map that is more linear and more similar to a proper hedge maze. Maps like these also restricts the expansion pattern of the pathfinding algorithms to a more linear manner and hence, the radial expansions of uninformed searches will not be observed here.

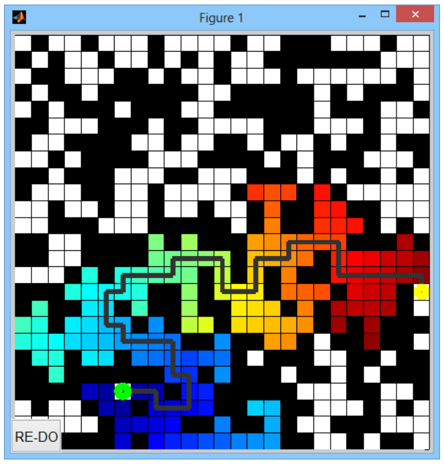

Figure 5: Uninformed search pattern

TABLE V: EXAMPLE 4 RESULTS

Algorithm	**Time Taken /s**	**Threshold**
IDDFS	1.183	152
IDA*	0.973	140
Fringe Search	0.949	140
BIDDFS	0.689	152

In this example, the BIDDFS algorithm really shines – the linearity of the map limits the node expansion number during pathfinding, and being a boundary uninformed search algorithm, node calculations are simplified since there is no involvement of heuristics, which is further enhanced by the boundary-node oriented expansion of the algorithm. This example clearly illustrates that among all the iterative-deepening algorithms (threshold related), the BIDDFS has the fastest node expansion rate. Also note that the difference in threshold between informed and uninformed searches is just 12, indicating that the number of nodes expanded by each algorithm is almost the same.

VI. CONCLUSION

Comparing with another uninformed search algorithm, the BIDDFS is found to be superior to the standard IDDFS in all simulation examples. This is credited to the small memory used to store the boundary nodes before the threshold is increased, and hence the next sub-runtime can start immediately from the leaf nodes instead of the starting node. When used in a more linear map, the BIDDFS is shown to be more superior to the informed search algorithms IDA* and fringe search, for a simpler calculation process means each node expansion can be done faster than that of the informed search algorithm. Though, being an uninformed search algorithm, the BIDDFS is still inferior to the informed search algorithms when exposed to an open map.

As to the future works of this algorithm, it is hypothesised that iterative-deepening algorithms will greatly benefit from parallel computing. During the runtime for each simulation, it was noted that MATLAB was not fully utilising the capabilities of the computer, which may explain the time difference between these algorithms with the non-iterative-deepening algorithms such as the Dijkstra's algorithm and the A* search algorithm, even with a lower memory footprint.

REFERENCES

[1] S. J. Russell and P. Norvig, Artificial Intelligence: A Modern Approach. New Jersey, USA: Prentice Hall, 2010.
[2] (2005, 6 October). Uninformed vs informed search. Available: http://monash.mindyoursite.com/index.php?title=Uninformed_vs_informed_search
[3] P. E. Hart, N. J. Nilsson, and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," Systems Science and Cybernetics, IEEE Transactions on, vol. 4, pp. 100-107, 1968.
[4] E. W. Dijkstra, "A Note on Two Problems in Connexion with Graphs," NUMERISCHE MATHEMATIK, vol. 1, pp. 269-271, 1959.
[5] R. E. Korf, "Depth-first iterative-deepening: an optimal admissible tree search," Artif. Intell., vol. 27, pp. 97-109, 1985.
[6] Y. Björnsson, M. Enzenberger, R. C. Holte, and J. Schaeffer, "Fringe search: beating A* at pathfinding on game maps," in Proceedings of IEEE Symposium on Computational Intelligence and Games, 2005, pp. 125-132.
[7] B. L. Sturm. (2012, 12/12). Bob L. Strum. Available: http://imi.aau.dk/~bst/