Robot Navigation

By: Aleksandar Manev (100586500)
Introduction to Artificial Intelligence
Swinburne University of Technology, 2017

INTRODUCTION

The Robot Navigation, this is a path finding system that uses advanced search algorithms to find the path between one point and another point on a pre-defined grid. The purpose of this system is to showcase my knowledge of understanding the difference between each search and to compare the effectiveness of each algorithm. My system does this by visual representation of the grid and the search process, In my program you are able to slow down the system or speed it up only by adding one extra argument, a value that will represent delay in milliseconds. This lets you analyses the search in more detail, in the Informed searches there is a label in each tile indicating.

Glossary	
State	A Combination that exists at a particular time.
Node	Object that holds state and record parent node
Graph	A direct network of nodes.
Tree	A connected graph with no cycles is called a tree.
Binary Search	Search a sorted array by repeatedly dividing the search interval in half
Frontier	An open list
BFS (Breadth-first search)	Uninformed search algorithm with open list functioning as a queue.
DFS (Depth-first search)	Uninformed search algorithm with open list functioning as a stack.
h(n) - Heuristic Cost	Estimated cost of the cheapest path from the state at node n to a goal state
g(n) - Path Cost	The cost to reach the n node.
A* (A Star) Search	Informed search algorithm with open list sorted by the value f(n)=h(n) + g(n)
GBFS (Greedy-best-first search)	Informed search algorithm with open list sorted by the value h(n)
Bidirectional Search	Bidirectional search means that the search performs forward and backward searching in the same time.
Iterative Deepening Search	A search that limits the depth of the DFS and corrects this bound in every layer of the search tree.
Iterative Deepening A* Search	A search that limits the f-cost of the A* Search and corrects this bound in every layer of the search tree.

How to Use My program:

Java -jar search.jar [file name] [algorithm]
Java -jar search.jar [file name] [algorithm] [delay time]

[file name] is your file name (E.g. input.txt)

[algorithm] is one of the algorithms listed below:

- **BFS** Breath First Search
- **DFS** Depth First Search
- **GBFS** Greedy Best First Search
- **AS** A* Search
- **BDBFS** Bidirectional Breath First Search
- **BDDFS** Bidirectional Depth First Search
- **BDGBFS** Bidirectional Greedy Best First Search
- **BDAS** Bidirectional A* Search
- **IDS** Iterative Deepening Search
- **IDA** Iterative Deepening A* Search

[delay time] by default is 200ms but optionally u can add argument of what speed you want to cover.

Search Algorithms

Comparison

Test Case		Node		Moves	
	Search Type	Single Directional	Bi Directional	Single Directional	Bi Directional
[15,11] (0,1) (10,3) (2,0,2,2) (5,0,1,14) (8,0,1,2) (10,0,1,1) (2,3,1,2) (3,4,3,1) (9,3,1,1) (8,4,2,1) (6,8,1,1) (8,8,3,1) (6,12,4,1) (0,8,4,2) (1,13,4,1)	BFS	110	116	48	48
	DFS	94	85	56	48
	GBFS	92	116	48	52
	AS	94	140	48	48
	IDS	270	N/A	48	N/A
	IDA	91	N/A	48	N/A

Test Case		Node		Moves	
	Search Type	Single Directional	Bi Directional	Single Directional	Bi Directional
[16,25] (1,1) (16,12) (2,2,2,8) (0,9,24,1) (19,10,1,3)	BFS	255	357	44	44
	DFS	322	97	186	44
	GBFS	165	265	44	78
	AS	322	456	44	44
	IDS	2093	N/A	44	N/A
	IDA	316	N/A	44	N/A

Test Case		Node		Moves	
	Search Type	Single Directional	Bi Directional	Single Directional	Bi Directional
[7,17] (1,1) (16,1) (3,5,13,1) (3,1,1,4) (5,0,1,4) (7,1,1,4) (9,0,1,4) (11,1,1,4) (13,0,2,4)	BFS	64	64	25	25
	DFS	45	38	39	25
	GBFS	47	49	39	39
	AS	57	85	25	25
	IDS	100	N/A	25	N/A
	IDA	53	N/A	25	N/A

BREATH-FIRST SEARCH

Breadth-first search or BFS is a simple uninformed search that check every combination until a goal is found. In this strategy, the parent node is expanded first, then all the nodes generated by the parent node are expanded next, and then their successors, and so on. In general, all the nodes at depth d in the search tree are expanded before the nodes at depth d + 1. Breadth-first search can be implemented by calling the GENERAL-SEARCH algorithm with a queuing function that puts the newly generated states at the end of the queue, after all the previously generated states.

DEPTH-FIRST SEARCH

Depth-first search always expands one of the nodes at the deepest level of the tree. Only when the search hits a dead end (a nongoal node with no expansion) does the search go back and expand nodes at shallower levels. This strategy can be implemented by GENERAL-SEARCH with a queuing function that always puts the newly generated states at the front of the queue.

GREEDY-BEST-FIRST SEARCH

According to Peter Norvig and Stuart J Russle, greedy best first search tries to expend the node that is closest to the goal, on the grounds that this is likely to lead to a solution quickly. Thus, it evaluates

nodes by using just the heuristic function that is f(n) = h(n), where the h(n) is estimated cost of the cheapest path from the state at node n to a goal state. (2010, p93)

A* SEARCH

According to Peter Norvig and Stuart J Russle, The most widely known form of best first search is called A* Search. It evaluates nodes by combination g(n), the cost to reach the node and h(n), the cost to get form the node to the goal. (2010, p94)

f(n)=g(n)+h(n)

IMPLEMENTATION

My implementation of the Grid Structure

- I look at this grid as a sliding puzzle, where the blank tile represents the robot and the walls are fixed position tiles. The only difference is that I am not trying to solve the whole puzzle by moving the blank but I am trying to move the blank to the goal position. In the same time changing the Heuristic function to worry only about the path for the blank tile and its goal position and not worry about how misplaced is the rest of the grid.

My Implementation of Blind-Search (Uninformed Search):

- By blind search I mean DFS and BFS, they are both implemented on the same file. Firstly, I set the defaults, such as search type (The search type can be DFS or BFS), the starting node with the given grid form the input file and I add that node to the frontier. When the search starts if firstly check if the start grind is same with the final grid if not then while it doesn't find solution it loops through the BFS algorithm. Firstly, checks if the frontier is empty this indicates that there are no more possible nodes to be examined and the solution is not found yet, so it outposts that solution is impossible and brakes out of the search loop. Knowing that the frontier contains one or more nodes inside it assigns the first node as a current search node to be examined. Then the current search node looks for possible ways to move and returns this in a form of list of children nodes each individual clone from the current node, in case the search algorithm is DFS it preforms operation to reverse the children's order in the list because when they are being inserted in the frontier later they all go on position 0 on by one posing the previous 0 back for one, a reversed children list will end up with original frontier list. Now of each child in children list I make sure that this node is caring a state that haven't been discovered before (check for a repetitive state), if the child node doesn't appear as repetitive state the search will check if this child is our final node, if that's the case the result output will be printed on the screen and the program will break the search loop. But if that's not the case this is the major difference between these two blind searches, in case of BFS the child will be stored on the back of the frontier, where in case of DFS the child will be stored in the front of the frontier. This is what makes the difference between the DFS and BFS search algorithm,

the DFS has a stack like frontier and the BFS has a queue like frontier. Finally, when the exploration of the current node is finished this node is being entered previous nodes list that holds every explored node, this is list is used for the respective state check. When that's done, the node is removed from the frontier and the search loops back to the beginning of the loop repeating until solution is fond or every position is tested.

My implementation of Best-Frist-Search (Informed Search):

- By Best-First-Search algorithms I mean Greedy-Best-First-Search (GBFS) and A* Search (AS). By default, this algorithm need to have pre-defined start state that is being read from the input file and frontier that have only the start state inside. When the search commences the first step is to check if the initial state is the goal state in the same time. If this is the case the system outputs result of 0 steps taken. But if this is not the case the search begins by entering a loop, a loop that will repeat itself until solution has been found. Inside this loop, the first check is checking is the frontier empty, if this is the case this means that the program has come to a state where it can't explore any more nodes, having explored all possible nodes and still haven't found a solution the system will output that solution is impossible to reach. If there are one or more items in the frontier I assign the first item laying on index 0 as a current search node, this is the node that's about to be examined. Then the program will check if this node contains the goal state if it does, the result will be outputted on the screen and system will break the search loop. But if it doesn't satisfy the condition for the goal state the current node will see what's the possibility of children nodes, all possible children nodes will be returned in a form of a list, I will use this list to loop through each child. Picking child by child I will check if this child has been discovered before if the child has been not discovered it enters it in the sorted list (frontier) using binary search to improve efficiency. This is where the main differences come between GBFS and AS, where GBFS has a frontier sorted by the heuristic cost h(n) that shows an estimated cost of the path with lowest cost from the n node to the goal. And AS has a frontier sorted by the heuristic cost summed with the cost to reach the node, we call this the f cost f(n) which represents the estimated cost of the cheapest solution through node n, f(n) = h(n)+g(n). When I am done populating the frontier with new nodes to be examined I store the current node (the node that we just finished examining) in a sorted list called previous nodes. This list keeps record of all previously examined nodes for repetitive state check purposes. Finally, I remove the node that was just examined out of the frontier and the search loop continues this process all over again until solution is found or all other possible states have been examined.

My implementation of Bi-Directional Searches:

- All four searches algorithms can be implemented as bi-directional search. I came across the Bidirectional search while reading the book Artificial Intelligence, A modern approach by Stuart J. Russel and Peter Nerving. According to them the idea behind bidirectional search is to run two simultaneous searches hoping that the two searches meet in the middle (Figure 3.1). So that what I did, I am running two simultaneous searches one starting from the goal position searching backwards and another starting from the initial state searching forward. I replaced the goal state check with checking if each child new

child if it is contained in the frontier of the opposite search. When the two searches connect, the system outputs the result.

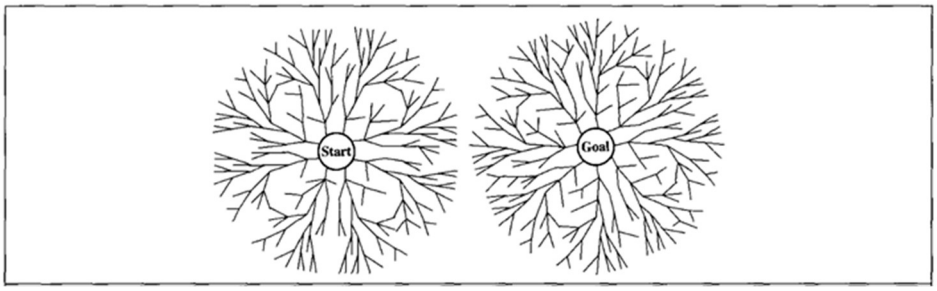


Figure 3.1 – Source: Artificial Intelligence: A Modern Approach 3rd Edition

FEATURES/BUGS/MISSING

Features

- **Visual Representation** – For the visual representation I used java's swing library to build a JPanel grid of CellPane positioned on m n coordinates each stored in a list that I later sent and assigned to each tile in the puzzle, so if a tile is a wall it will be colored dark gray if a tile is in the frontier it will be colored green, if a tile is blue it means this tile have already need examined, when solution is found the path will be marked as yellow tiles.
- **Bi Directional Searching –** I implemented a bidirectional search for BFS, DFS, GBFS and AS, to start it you just need to add BD in front of the search u want to see preform as Bidirectional (e.g. use BDBFS to see Bidirectional BFS)
- **Making the search 600% faster** – I realized that my search was getting slower over time on bigger sized grids. This was caused from oversized previous nodes list, because for each new child I was checking every single past node individually. So, I implemented a binary search that returns an index where this node needs to be. I use the binary search to create sorted list for the frontier and the previous nodes. Sorted lists are very efficient when the list gets bigger because we don't need to do linear sorting. Also, using the binary search I could find the index where the cost starts and search for repetitive states form this point until the cost is lover then the current cost. This gave a great result a search that was running for 3 seconds afterwards finished in 500 ms

Bugs

- One bug that I have is in the IDS search the program will never say "solution impossible". This is due my implementation when the frontier is empty I look at it as there is nothing more to discover at this depth. But the problem is that when it examines everything the frontier is full as well.

Missing

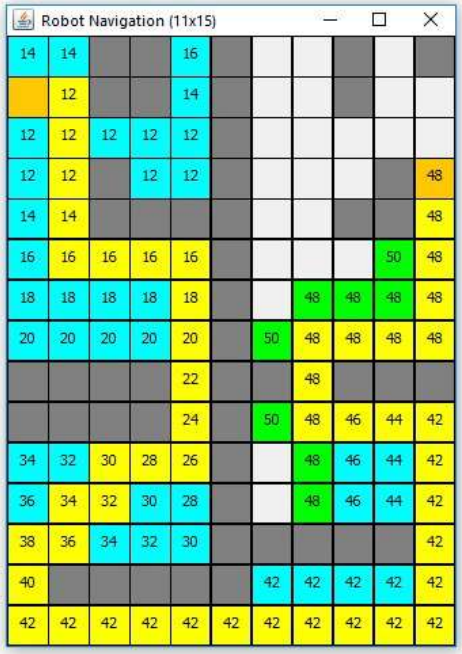
- I am happy to report that my program works as expected in every required and every additional feature I implemented.

RESEARCH

If you managed to do some additional research to improve the program in some ways (see a number of ideas for research initiatives), please report it here.

I Improved this program in few ways:

- For this program, I developed **General User Interface** using java's Swing library to give good visual representation over the searches. The GUI works in real time as the search processes showing f(n) value in each cell on informed searches, it gives the user better understanding how each algorithm discovers nodes until it reaches solution. By default, there is 200ms delay on the search so the user can visualize the GUI, if you want to slow down the process even more you could add extra argument when calling the search. For example, if you want to see the search process on every 2 seconds then you should call the file as "java -jar search.jar input.txt AS 2000" the last argument represents the delay time. To see the program in full speed set the delay on 0.
    - Screenshot:



- I realized that the searches are getting slower after the rise of the discovered nodes, so I **improved the efficiency** of all searches by using binary search and priority queue to search thought past states much faster. Also, later I taught of a new Idea to improve my searches and this was by modifying my comparison function, the function the compares two nodes. This function was checking every tile of the gird if it is on the same position to the other node' tile on the same positon. How I improved it now is first the functions checks if the h cost of these two tiles is the same. If the H-cost is not the same then this indicates that blank tile is on different position and returns false not even trying to check any tile on the grid. This resulted in huge improvement in search speed, in fact it made my speed 600% improved.
- Another way I improved my searched is by implementing **Bidirectional Search** on the applicable searches (BFS, DFS, GBFS, AS). I got the idea for this from the textbook while I was reading Chapter 3, in section 3.4.6 page 90 the authors talk about bidirectional search. The motivation is that

$b^{d/2} + b^{d/2}$ is much less then b^d. This search makes the BFS time complexity O($b^{d/2}$) in both directions. The space complexity is also O($b^{d/2}$).

- Looking in improving search algorithms I came across **Iterative Deepening Search (IDS)**. DFS has very good space complexity but the down side is it is not complete and it is not optimal. The IDS is the best way of improving the depth first search and to make it complete and optimal with keeping the same space complexity. Because it gives us the same time complexity of DFS O(b^d) and the space complexity of DFS O(*bd*). The approach for IDS is by simulating Breath First Search but without the space cost, and this is done by adding depth bound to the Depth Frist Search. Having this brilliant idea of IDS we can use it to combine it with the idea of A star search making **Iterative Deepening A* (IDA)**. The IDA search functions same as IDS we just use the f cost as a bound. So firstly we approach the lowest f cost and then we bump it up to the next lowest f cost.

CONCLUSION

To conclude I think the A* Search is the most efficient for this consistent type of problem, because it has the benefit of working as Best-First Search but also gives the most optimal solution. How I improved this algorithm is using binary search and priority queue to improve the efficiency of searching through substantial number of nodes in the past states list and the frontier. Also, I realized that the frontier can be sorted not only by F cost but by H cost as well. For example, if the frontier has multiple nodes with f-cost of 33 they might have different h-cost. So by sorting the f-cost firstly and h-cost secondary, the function used for comparing two nodes can identify if the h costs is different than don't bother checking if this two nodes are the same return false automatically , because this indicates that the grind is in different combination of movements.

ACKNOWLEDGEMENTS/RESOURCES

I mostly used the book Artificial Intelligence, A Modern Approach 3rd Edition by Stuart J. Russel and Peter Norvig to assist me with the explanation of the algorithms. And my program was developed based on my understanding and logic on the field of Artificial Intelligence and Data Structures and Patterns.

For my research of IDS and IDA I watched a lecture given by Alan Mackworth from UBC University.

For my research

REFERENCES

Stuart J, Russel and Peter, Norvig. 2010. Artificial Intelligence, A Modern Approach 3rd Edition. Prentice Hall.

Alan Mackworth. 2013. Lecture 9 | Search 6: Iterative Deepening (IDS) and IDA*. Online video. Accessed on 5th of April 2017 https://www.youtube.com/watch?v=5LMXQ1NGHwU&t=2226s