Search
Robot-Assisted Evacuation

Introduction

The assignment asks to implement and evaluate a number of AI search algorithms. They should be usable by a robot to find a safe path for evacuation from a building. I have completed all basic requirements, as well as two extensions. The basics are: Search1 which compares Breadth First Search and Depth First Search; Search2 which compares Best First Search and A* (A-Star) Search. The two extensions are: Search3 which examines Bidirectional Search; a graphical user interface (GUI) which makes comparing and contrasting the algorithms much easier.

It is worth noting that in order for these algorithms to be applicable for path finding during evacuation, a robot would need a live-map of the building. I.E., if some intersections are unreachable due to extreme conditions, the robot's map would need to be updated to reflect that.

Running

The provided jar files can be run from within the submission directory. The standard command "java -jar SeachX.jar" can be used. I have also provided a make file which can be used to compile and run the code. Commands include: "make compiled", "make SearchX", "make clean". Cleaning will remove only the files generated by make. It will not affect the jar files. In all cases, execution should happen from within the submission directory because the program tries to open "assets/config.props" and "assets/screen.props".

Design

In designing and developing the program for his practical, I have utilized Object-Oriented Programming. I have a couple of abstract classes that contain core functionality and about ten classes that inherit it. The whole program is divided into four packages ('gui', 'main', 'problem', 'searches').

The 'util' package contains a few classes with supporting functionality that I have written over the past couple of years. Some util classes have been written for previous practicals but none of them were created specifically for this practical.

All packages listed in the first paragraph of this section contain classes written for this assignment. The Problem classes define how the world of the task is represented in the system. The Searches classes implement the different AI search algorithms. The Main classes are used to run the different comparisons (Search1, Search2, Search3). Finally, the GUI classes describe how all of the above is displayed on the screen.

Problem

This package contains two classes. An Intersection is made up of x and y coordinates, an id, and a list of connected Intersections. The first three attributes are public because the are commonly needed across the system. They do not need to be encapsulated, as they are final and cannot be changed once defined in the constructor.
The Intersection class also acts a manager for its instances. A number of static methods are used to create, store, and access intersections. Two separate objects are used to reference the initial (‘init’) intersection and the goal. They are effectively final, as they are only set once per input file.
The array list of Intersections that are statically managed is effectively the state space. The ‘forEachConnected’ method is used to apply a method to all connected intersections. The constructor for that class is private, as new instances need to be put in the array for managing. The ‘createIntersection’ method is publicly available instead. It also advances the id every time an Intersection is created.

A Node is part of the graph search problem. It contains an intersection, variables that define the distances from this node, and a reference to the parent node. All of these fields are final. Once defined in the constructor, they do not need to be changed. The parent and intersection fields are public because they are commonly used but cannot be changed. Methods in this class allow accessing child nodes, and the distances depending on the currently selected heuristic.

Searches

This package contains three packages - ‘general’, ‘informed’, ‘uninformed’. The package ‘searches.general’ contains only one class. The abstract class ‘GraphSearch’ implements the general search algorithm described in the lectures. It keeps track of the visited intersections, the status (whether or not the search is complete), the last Node visited, the solution Node. It also contains a reference to the method used to update the interface to reflect the status of the search. The frontier is accessed only via abstract methods. This way implementing classes can use any Collection as a frontier. The general search algorithm needs to add elements to the frontier and retrieve them but it is not concerned with how they are stored.
The core of this class is the ‘nextStep’ method which moves the search forward by exactly one step. At each step, first - take the next node, second - check for search failure, third - check for goal reached, fourth – expand node, fifth – show progress, lastly – mark the intersection as visited. This method is called by the class that keeps a reference to the search algorithm (that is the MainSearch class from the main package).
This class also contains a backtrack method which lets it construct a list of intersections that make up the path from init to goal.

With this core, all search classes have minimal differences between them. Breadth First Search (BFS) uses a linked list as a queue for its frontier. Nodes are added to the end of the queue. They must also contain intersections that have not yet been visited. Nodes are retrieved from the front of the queue. This way intersections that are one connection away appear first in the queue. They are followed by nodes that are two connections away, and so on. This way BFS visits an expanding tree of nodes by covering the full breadth of the tree at each depth.

Depth First Search (DFS) uses a linked list as a stack for its frontier. Nodes are added to the top of the stack. Similar to BFS, new nodes can only contain intersections that have not been visited. Nodes a retrieved from the top of the stack. This way, nodes that have been added first are visited last. Newly added nodes are visited first. This way DFS visits nodes until it reaches one with no connections (the maximum depth of the graph tree). Removing nodes from the stack achieves returning to previous nodes.

Best First Search (Best) uses a priority queue for its frontier. Nodes are ordered based on the selected heuristic (see config.props). The primary field by which nodes are compared is the distance to the goal. Ties are broken by the distance from the initial node (this is accumulative distance from the path so far). Second ties are not manually broken. The collection returns them in the order they were added. This is likely to yield good results, as nodes that are added earlier have better-cost parents and may result in better-cost children. Best allows multiple nodes in the frontier to be at the same intersection. It is possible that it finds a better path to the same intersection. To avoid visiting the same intersection twice, there is a mechanism to remove repeating intersections in the getNext method. When a node is retrieved from the front of the queue, all other nodes at the same intersection are removed. If they are not at the front, then they will not give better results.

A-Star Search (A*) also uses a priority queue for its frontier. Nodes are again ordered based on the selected heuristic (see config.props). A* takes the sum of the distance to the goal and the distance to the initial node. Ties are not manually broken. That is again left the priority queues ordering. A* also allows multiple nodes in the frontier to be at the same intersection. The reasoning is the same as for Best, and the removal mechanism is also the same.

Bidirectional Search (2Dir) is very similar to BFS. It is, in fact, the combination of two BFSs. Because of that, it overwrites the next step method of the abstract class. It expands the nextStep with a method to check if the two searches have crossed paths. Lets say that happens at intersection X. The method takes the nodes from goal to X and then constructs new nodes at the same intersections but with parents pointing in the opposite direction (to the initial intersection). The last reversed node is at the goal.

Main

This package contains files necessary for running the program. The InputReader class reads and parses files that define the intersection locations and connections. The MainSearch class is an abstract class that defines the behaviour of all Search1, Search2, and Search3 classes. MainSearch contains an array of GraphSearch instances (instances of the abstract search algorithm class). The only difference between Search1, 2, and 3 is how they fill the array. Search1 instantiates one BFS and one DFS; Search2 – one Best and one A*; Search3 – one 2Dir.

An enumeration is used to keep the state of MainSearch. State ‘Ready’ means that the program is ready to run its algorithms on the next file. ‘Searching’ means that the program is actively processing the current file, executing its algorithms. ‘Done’ means that the program has finished processing all files.

MainSearch contains several methods. The ‘boot’ method loads the properties from “assets/config.props” and “assets/screen.props”. It then instantiates a Display (see the GUI section)

and loads the first file. Search1, 2, and 3 all contain a ‘main’ method which creates an instance and calls ‘boot’ on it.
This class also contains a ‘nextStep’ method. It is called by the Display instance. Depending on the current state “Ready”, “Searching”, or “Done”, it will either load the next file, execute the next step of the active GraphSearch, or shut down respectively. The program keeps track of the index of the active GraphSearch and accesses the instance via the array. When the active algorithm is no longer searching for a solution, MainSearch calls backtrack on it and displays the path if no such is found. It then increments the index of the active algorithm.
The ‘checkProgress’ method determines whether all algorithms have been executed on the current file and changes the state accordingly.
The ‘loadNextFile’ method prints a progress bar to the console each time it’s called to show how many files have been processed. If all files are processed, it changes the state to Done. Otherwise, it sets the state to Searching.

InputReader contains half a dozen static methods which are used to open a given file, parse its contents, and construct Intersections. The expected format is the one on studres but without the array declaration. For full details, see the files in ‘assets/maps’. The methods here parse the files one line at a time. The file ‘config.props’ defines the structure of the lines. Intersections are created with the static method ‘createIntersection’.

GUI

To create a graphical user interface (GUI), I use a combination of java.swing and java.awt classes. The class Display extends JFrame and defines how the program is displayed on the screen. An instance of MainSearch creates an instance of Display and both objects have a reference to one another. Display exposes a few methods to update the view on the screen. All other methods are private as they define how the Display is configured. A display has three components: map, which displays the graph loaded from the file; button, which moves the program by one step when pressed; and state, which is a label that details the status of the active search algorithm. The configuration methods of these fields add them to the frame and set their positions and sizes. The public methods that update the status change the text in the label, the nodes on the map, and the text of the button.

The class Graph extends JComponent and defines how a graph of nodes is drawn in a component. The Display’s map field is an instance of Graph. Methods in this class do a variety of operations to support drawing. These include: transforming Intersection coordinates into on-screen pixel coordinates; drawing circles at these coordinates and filling them with letters; drawing lines between connected Intersections. The class keeps a reference to the frontier, visited intersections, and solution path of the current active GraphSearch algorithm. These are used to determine the colours of the circles and lines drawn on the screen. The default colour is black; grey is used for visited intersections; blue – for the frontier; cyan – for the initial node; red – for the goal; green – for the current intersection; and orange for the solution. The colours are listed in increasing priority. That is, orange covers other colours if a solution exists. Green replaces red if the current intersection is the goal. The goal stays red even when added to the frontier. The initial node stays cyan even after visited. The reason for all of these is to make visually tracking the algorithm easier. The solution in orange stands out (yellow makes the letters inside harder to read).

Properties

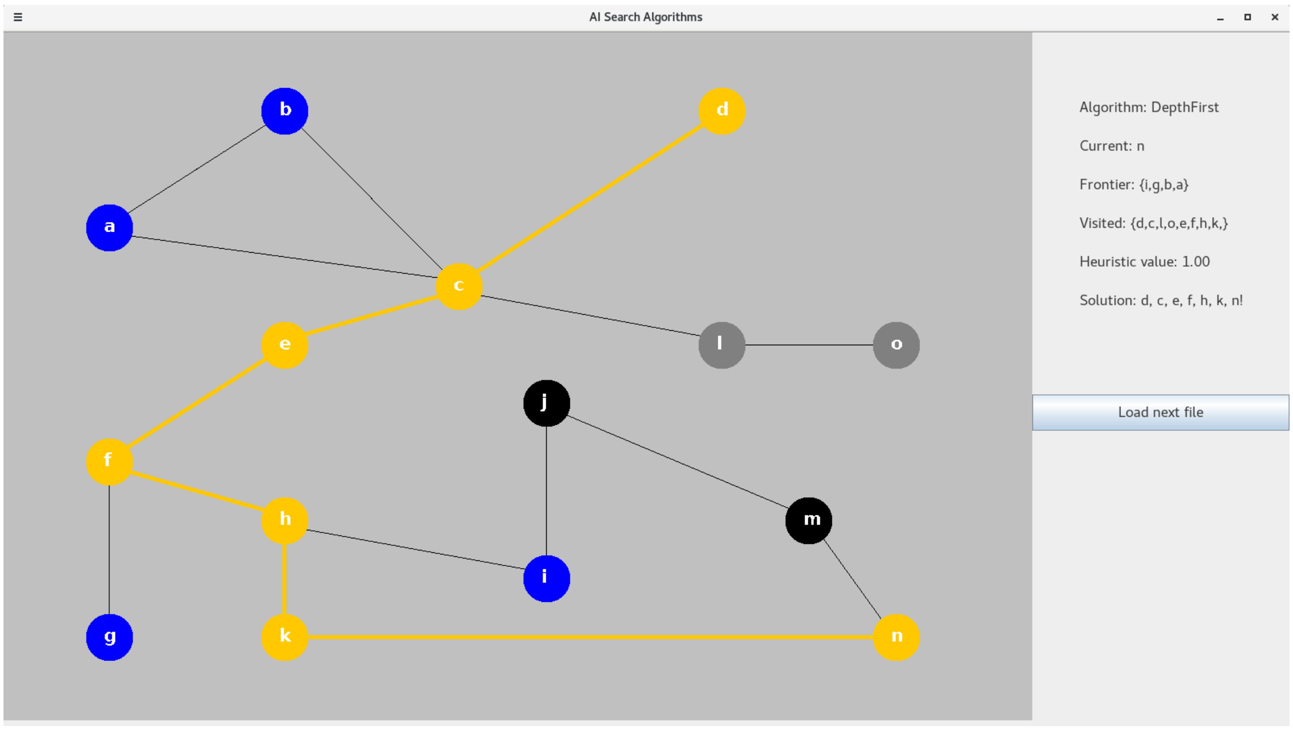
The system has a number of configurable parameters that can be changed without the need to recompile the code. These are referred to as ‘properties’ and are stored in two text files: “config.props” and “screen.props”. Each property is written as a key-value pair separated by an ‘=’ (equals) sign. The program expects to find the keys as they are written. The values may be changes, as long as the new values are of the same type (longs, doubles, or strings). Note that ‘1’ is parsed as a Long value, whereas ‘1.0’ is parsed as a Double.

The properties in ‘config.props’ define how an input file is structured: how are the numbers separated, how do lines start and begin, what do the different number codes mean. It also defines the names of the input files that need to be processed, and their indexing. MainSearch iterates the indexes from the property ‘first index’ to ‘last index’ and opens files whose names start with the value of the property ‘coordinates filename’, followed by the current index of iteration, and end with the value of the property ‘file extension’. The submitted configuration will open all files from “assets/maps/loc1.txt” to “assets/maps/loc9.txt” and then the same with the connections in “assets/maps/con1.txt” to “assets/maps/con9.txt”. The final property is ‘heuristic to use’ which can be assigned to either ‘manhattan’ or ‘chebyshev’. If any other value is provided, the program will default to ‘euclidean’.

The properties in ‘screen.props’ define how components are displayed in the window for the GUI. The first two properties set the dimensions of the window (in pixels). All other values set the various positions and sizes of the components. These are measured as percentages of the window dimensions. So “button x=0.80” means that the x coordinate of the button should be 80% from the left end of the window. Using percentages means that the window dimensions can be changed without displacing all of the components. Only the graph properties (coord multiplier to font size) are measured as percentages of the component size, rather than the window size.

Examples & Testing

I tested the system by running all three searches (Search1, Search2, Search3) on all 9 provided configuration files. I have taken screenshots of all 45 searches (5 algorithms by 9 inputs). They are included in the ‘runs’ directory. Note that for A* and Best, I have used euclidean distance to generate the paths visible on the images. In this and in the next sections, I will only include some of the more relevant runs.



Algorithm: DepthFirst

Current: n

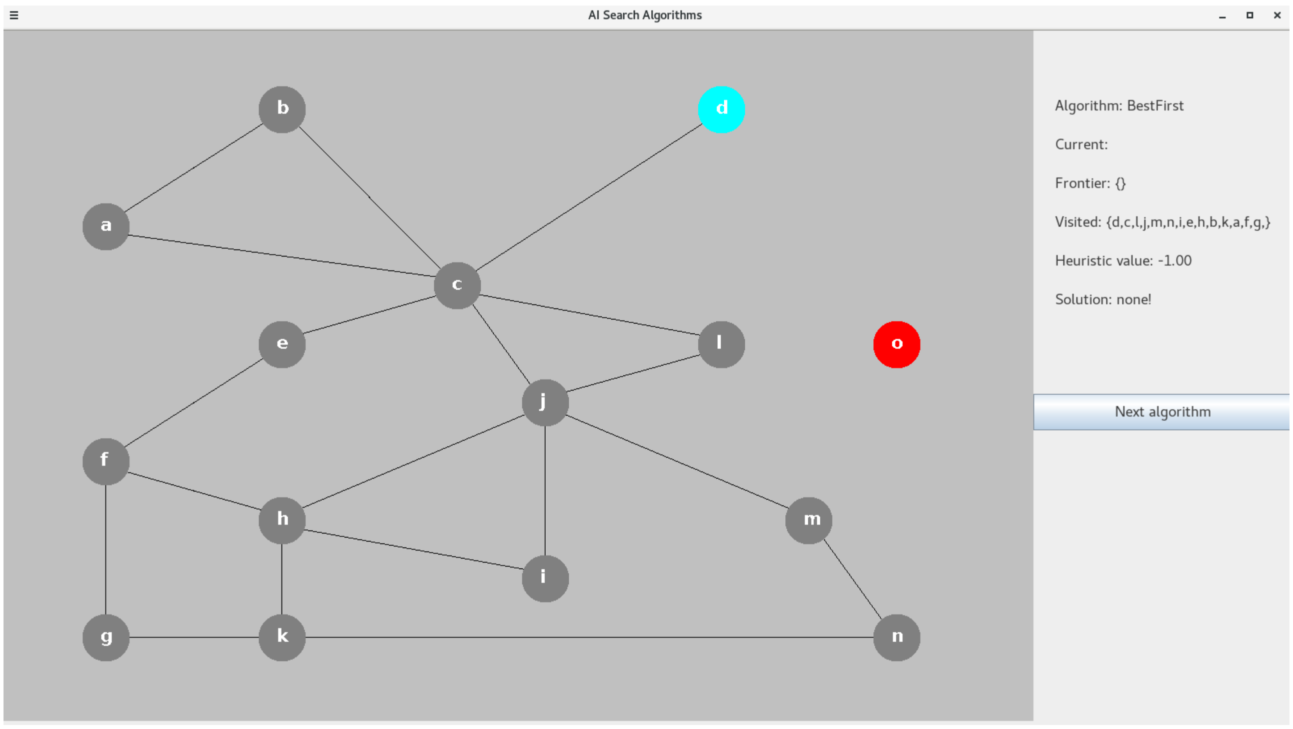
Frontier: {i,g,b,a}

Visited: {d,c,l,o,e,f,h,k,}

Heuristic value: 1.00

Solution: d, c, e, f, h, k, n!

Example of successful path finding. Map 5, from d to n.



Algorithm: BestFirst

Current:

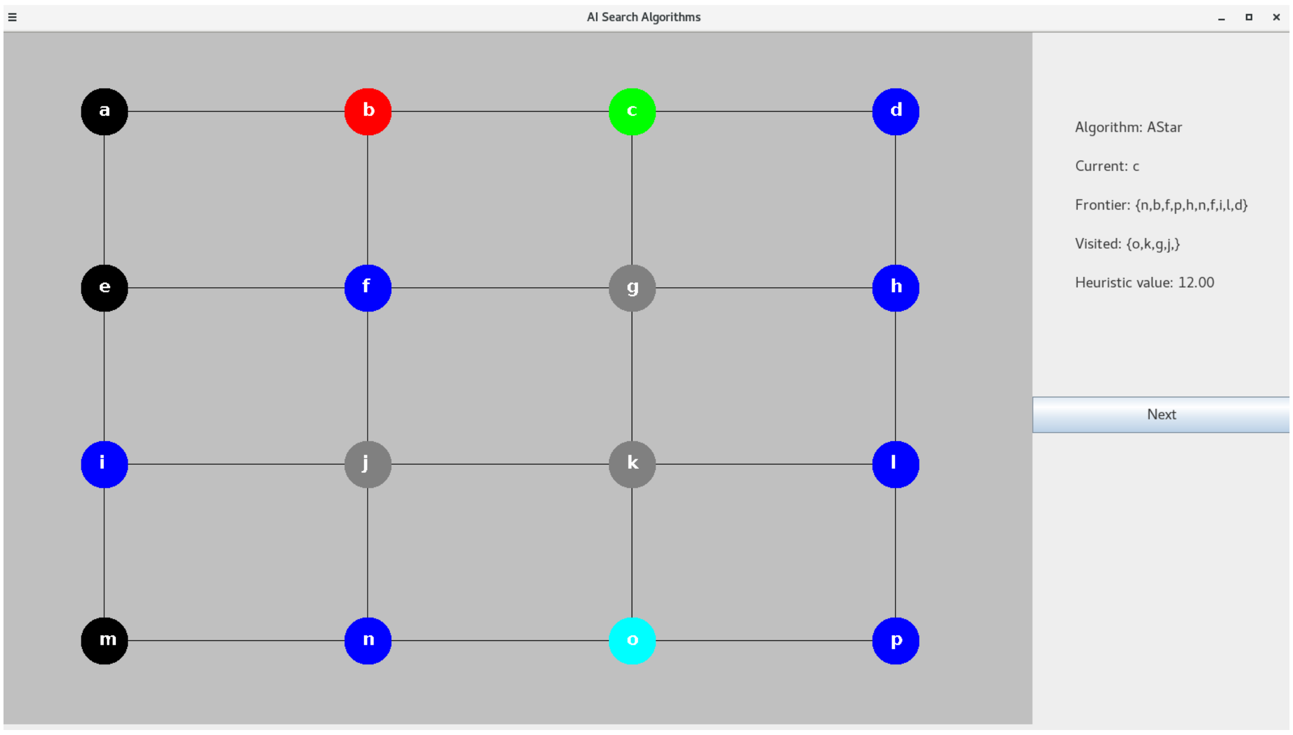
Frontier: {}

Visited: {d,c,l,j,m,n,i,e,h,b,k,a,f,g,}

Heuristic value: -1.00

Solution: none!

Example of unsuccessful search. Map 3, from d to o.



Algorithm: AStar

Current: c

Frontier: {n,b,f,p,h,n,f,i,l,d}

Visited: {o,k,g,j,}

Heuristic value: 12.00

Example of search in progress. Map 8, from o to b.

Evaluation

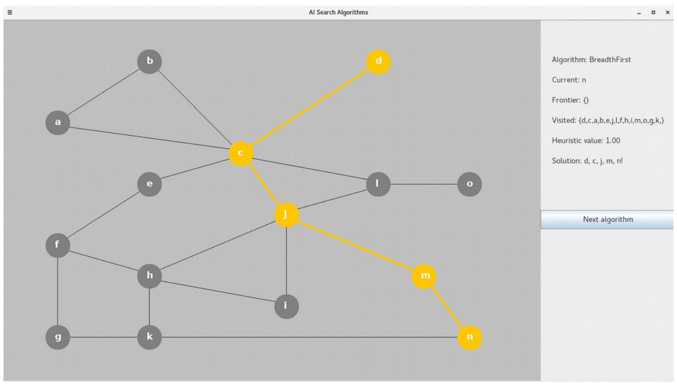
Overall, my implementation achieves all the required basic parts, as well as two valuable extensions. The GUI supports comparing the different search algorithms and gaining insight on them. The bidirectional search furthers the analysis by showing an algorithm that quickly determines if there is no path.

My program can be further improved by extending it to support multiple robots searching for a path at the same time (and not interfering with one another).

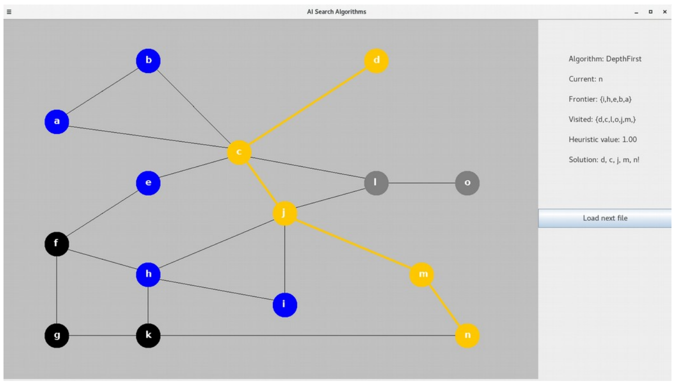
The following subsections go into depth on the benefits and drawbacks of using the different algorithms. All conclusions are derived from the 45 executions mentioned earlier (see 'runs' directory). Conclusions also draw upon the material covered in the lectures.

Search1 – BFS vs DFS

BFS creates shorter paths than DFS. This is due to visiting nodes closer to the start before visiting ones further ahead. DFS, on the other hand, goes through nodes until it can't get further from the start, then returns back to closer nodes. The drawback of BFS is that it visits more nodes. If we look at BFS1.png and DFS1.png we will see that BFS has visited all nodes, where as DFS has visited only 2 nodes outside the same solution.



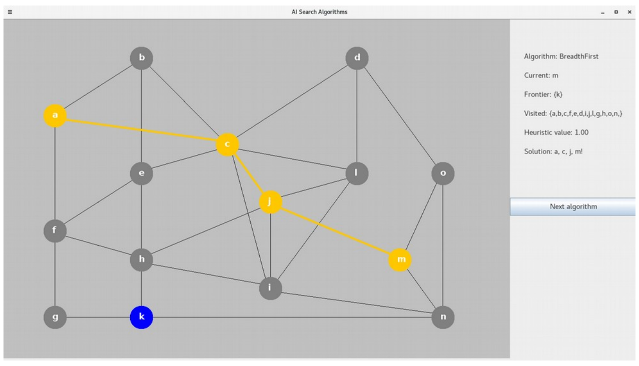
Algorithm: BreadthFirst
Current: n
Frontier: {}
Visited: {d,c,a,b,e,j,l,f,h,i,m,o,g,k,}
Heuristic value: 1.00
Solution: d, c, j, m, n!



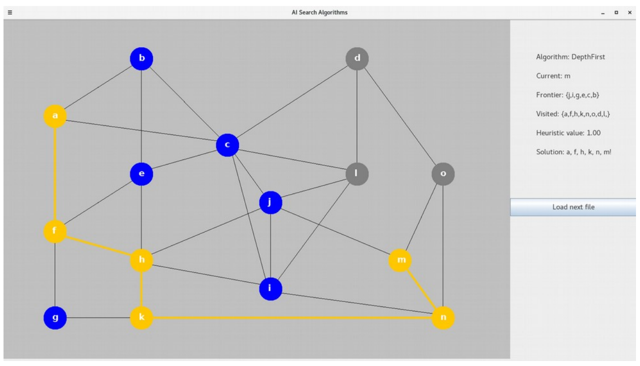
Algorithm: DepthFirst
Current: n
Frontier: {i,h,e,b,a}
Visited: {d,c,l,o,j,m,}
Heuristic value: 1.00
Solution: d, c, j, m, n!

BFS on the left, and DFS on the right for Map 1.

Map 4, however, shows that BFS finds shorter paths than DFS:



Algorithm: BreadthFirst
Current: m
Frontier: {k}
Visited: {a,b,c,f,e,d,i,j,l,g,h,o,n,}
Heuristic value: 1.00
Solution: a, c, j, m!

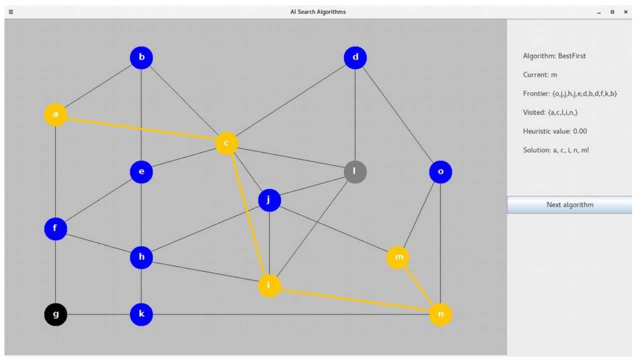


Algorithm: DepthFirst
Current: m
Frontier: {j,i,g,e,c,b}
Visited: {a,f,h,k,n,o,d,l,}
Heuristic value: 1.00
Solution: a, f, h, k, n, m!

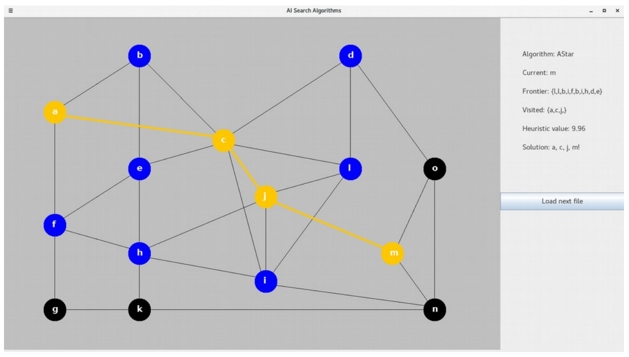
Note that BFS and DFS do not use heuristics, so the displayed value defaults to 1.
When comparing the path length, the number of nodes on the path is a good indicator. BFS paths are shorter in those terms.
The euclidean length of paths grows with the number of nodes between the start and end. If we have a straight line between two points, introducing another point, outside of that line, will create a longer path.

Search2 – Best vs A*

The relationship between Best and A* is similar to that between DFS and BFS. Best tends to find paths that are a little longer than those found by A*. But the benefit of Best is that it visits a lower amount of nodes. Thanks to the heuristics, Best always directs itself to the node, closest to the goal. A*, on the other hand, always finds the shortest path, because it takes into account the distance travelled. The heuristic for A* shows the distance from init to goal, where as the heuristic for Best show the distance from the current node to the goal.



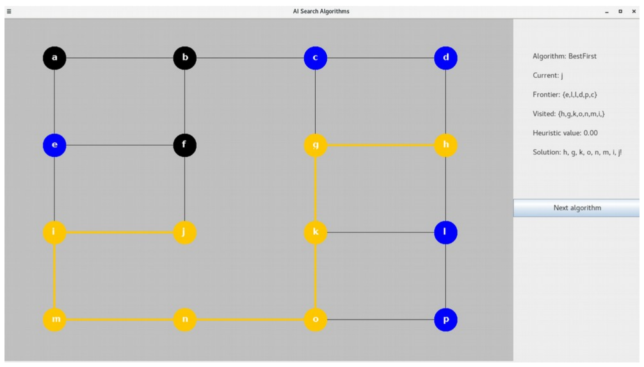
Algorithm: BestFirst
Current: m
Frontier: {o,j,j,h,j,e,d,b,d,f,k,b}
Visited: {a,c,l,i,n,}
Heuristic value: 0.00
Solution: a, c, i, n, m!



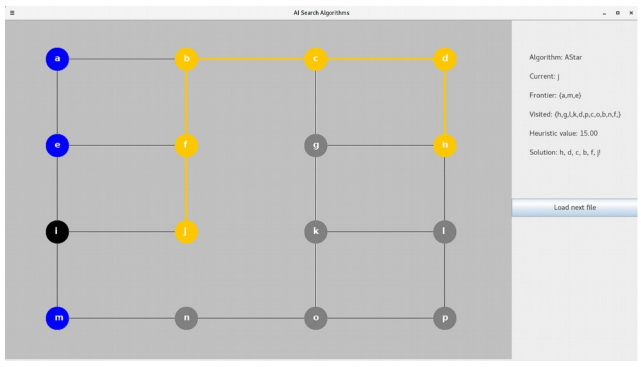
Algorithm: AStar
Current: m
Frontier: {l,l,b,i,f,b,i,h,d,e}
Visited: {a,c,j,}
Heuristic value: 9.96
Solution: a, c, j, m!

Map 4:
Best, on the left image, determines that node i (lower middle) is closer to m, than j is.
A*, on the right image, determines that the path to j is better than the one to i.

The results on Map 9 offer a good illustration of A* better path at the cost of more nodes visited.



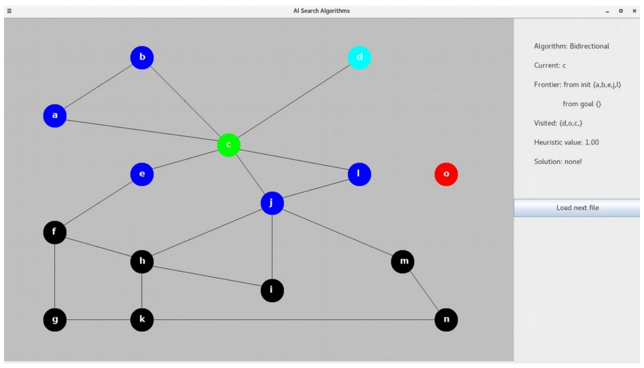
Algorithm: BestFirst
Current: j
Frontier: {e,l,l,d,p,c}
Visited: {h,g,k,o,n,m,i,}
Heuristic value: 0.00
Solution: h, g, k, o, n, m, i, j!



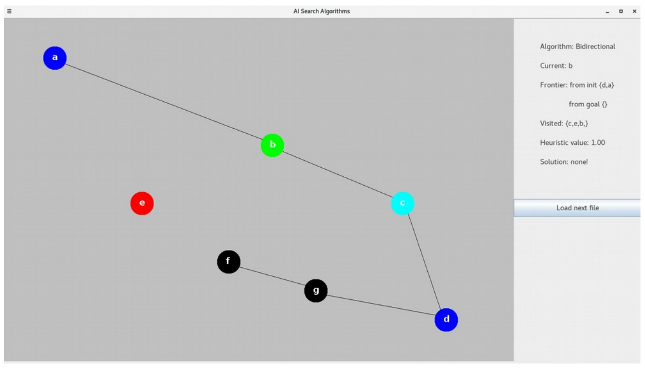
Algorithm: AStar
Current: j
Frontier: {a,m,e}
Visited: {h,g,l,k,d,p,c,o,b,n,f,}
Heuristic value: 15.00
Solution: h, d, c, b, f, j!

Search3 – 2Dir

All four of the algorithms described so far need to visit all nodes of a graph to determine if the goal is not connected. This is best illustrated in maps 3 and 7. This is where Bidirectional search does much better. Since 2Dir is double execution of BFS, it inherits the benefit of short path and the drawback of visiting lots of nodes. However, the following two images show the added benefit of 2Dir.



Algorithm: Bidirectional
Current: c
Frontier: from init {a,b,e,j,l}
from goal {}
Visited: {d,o,c,}
Heuristic value: 1.00
Solution: none!



Algorithm: Bidirectional
Current: b
Frontier: from init {d,a}
from goal {}
Visited: {c,e,b,}
Heuristic value: 1.00
Solution: none!

In both cases 2Dir has already determined that there is no solution because the frontier from the goal is empty. It can be seen that a lot of the nodes are black (not seen), as opposed to grey (visited).

Literature Review

AI Search algorithms have a variety of real-world usages across different fields. In most cases, the algorithms applied are modifications of the ones we study in this course. Adjustments are made to better suit the needs of the specific problem at hand.

Search algorithms are often used by GPS programs. Although no company has disclosed the specifics of its route finding algorithm, it is speculated that A* or Dijkstra is used, and that the heuristic values larger roads more highly.
A variety of search algorithms is used for the layout of Very-Large-Scale Integration (VLSI layout). Such algorithms can find an optimal placement for all transistors on a single chip.
Automatic assembly sequencing uses search algorithms based on a combination of Breadth and Depth searches to find a way to construct complex geometric shapes using a variety of parts.
Researches often take advantage of informed search algorithms to design different types of protein. Because of the huge amount of possible proteins, search algorithms are often supported by a branch-and-bound algorithm to reduce the search space.

Word count without Bibliography: 3355.

Bibliography

[1] Which search algorithm does Google maps use to find paths and why?
https://stackoverflow.com/questions/41325566/which-search-algorithm-does-google-maps-use-to-find-paths-and-why

[2] What are some real-life examples of Breadth and Depth First Search?
https://www.quora.com/What-are-some-real-life-examples-of-Breadth-and-Depth-First-Search

[3] What algorithm does your car's GPS use?
https://www.quora.com/What-algorithm-does-your-cars-GPS-use-Is-it-the-shortest-path-algorithm

[4] Efficient maze-running and line-search algorithms for VLSI layout
S.-Q. Zheng, J.S. Lim, S.S. Iyengar
https://ieeexplore.ieee.org/document/465749/metrics#metrics

[5] VLSI layout algorithms
Andrea S. LaPaugh
https://dl.acm.org/citation.cfm?id=1882731

[6] Automatic assembly sequence exploration without precedence definition
Roberto Vigano, Gilberto Osorio Gomez
https://link.springer.com/article/10.1007/s12008-012-0165-9

[7] Protein Design by Algorithm
Mark A. Hallen, Bruce R. Donald
https://arxiv.org/abs/1806.06064

[8] Algorithms for protein design
Pablo Gainza, Hunter M Nisonoff, Bruce R Donald
https://www.sciencedirect.com/science/article/pii/S0959440X1630015X