

Implementation List

Part one, The Depth-first search

The Breadth-first search

Part two, The Best first search

The A* search

Literature review

The search is the standard method for artificial intelligence to solve problems. In some little games like cuckoo, sudoku, crossword puzzles, the search algorithm can find the specific place in these games. The application of the search algorithm also can be found in daily life. For instance, many companies engaged in developing the pilot automobile. The search algorithm can address the vehicle routing problem ^[1]. The route search algorithm can divide into the static algorithm and dynamic algorithm. The static algorithm will find the shortest path by judging the geographic information, the traffic rules and other constraints. As for the actual traffic situation, the static algorithm may not useful. The dynamic algorithm is based on the static algorithm, it also combines real-time information, which can make it adjust the path to find the best path. The common search algorithms used in routing problems are Dijkstra search, A* search and bidirectional search. Following, there are four kinds of algorithm involved in this assignment, which are the depth-first search, the breadth-first search, the best first search and the A* search. The report will indicate the concepts about these algorithms and the implementation of them ^[2].

Design: Part One

The Alpha must find the way from the start node to the exit node by using the map which is an array. In this case, Alpha can only walk along the path that connected with any two nodes. There is also some path not available because those nodes do not connect. In the map, the whole nodes in the map are involved in the state space. As for the code, these nodes can be stored in the array like char ['a', 'b', 'c', 'd', 'e']. Moreover, among the nodes, there is a start node where Alpha starts to walk away, which is the initial state. The initial state can be found by using the loop that finds the number 2 (means start node) in the map array. Similarly, the exit node is the goal which means Alpha has found the way from the start node to the exit

node. The implementation way is the same as initial state. Due to some nodes cannot reach to the certain node, because there is no way between them. So, every time, Alpha arrives at a new node, the available nodes in the next step are changing. In this situation, Alpha needs a method to find which nodes can walk to when it arrives at a new node. This method can be described into Successor functions, which makes Alpha find the available nodes. Just like the way finding the initial state and the goal state, there will be a number 5 if two nodes connect with each other, the opposite is number 10 which means there is not the way between two nodes. Therefore, the available nodes can be found by looping the map array and judge if the number is 5, if it is 5 then add the node to the set. This function will always return to a dataset which contains all the available nodes. To ensure that Alpha does not walk to the repetitive nodes, the node that arrived before will not add into the frontier set. The order of the nodes in the data set also will be different by using diver's algorithm. In the case, the problem is to find the way. Thus, the action of Alpha is the movement between two connected nodes until arrive at the goal state. Alpha will stop movement when it judges the current node is the goal node. During the movement, Alpha will spend time, battery or something when it walks along the path, the cost like time, energy, the length of the path can be considered the path cost.

To implement the DFS and the BFS, the first thing is to distinguish the difference between them. The depth-first search begins with a node. After visiting this node, then it will visit the node that connects to the current node and it is not visited before. Then, it will find the new node connected to this node and it is not visited before. By this way, it will finish the process until visiting all the node or getting the goal node. The complexity of the DFS depends on the number of the path; it will not check the repetitive nodes^[3]. As for the breadth-first search, it also begins with a node; then it will find the all available nodes connected to the current node. Following, it will visit these nodes one by one. After this, it will go to visit all nodes that connected to the first available node. Also. It will loop this processing until all nodes have visited or found the goal^[4]. The complexity of the BFS depends on the numbers of nodes; it can check the repetitive nodes. The significant difference between two methods is that the former visits the first adjacent point of the vertex and then the second adjacent point of the vertex; the latter visits all the adjacent points of the vertex from the vertex and then goes

down one layer at a time ^[4]. Whatever the breadth-first search or the depth-first search, both can change from the general search method. In order to solve the problem, the first thing is to figure out the state space, initial state, the goal state and the rule. In the case that finds the way by the map, the initial state is the start node, and the goal is the exit node. The rule is Alpha can only move to the available node every time. So, in the actual java implementation, the first thing is to figure out how many nodes in the map through counting the length of the map array (x or y), the state space will store in a character array. Then, loop the map array to find the initial state and the goal state and stored into object character (char initial, char goal). Regarding DFS, the next step is initialising a stack as a frontier collector. Moreover, if it is BFS, it should use the queue as a frontier collector. The difference between stack and queue is the pull order of the elements. In the stack case, the first element pushed in will be the last one pulled out. For the queue, the first element added will be the first one pulled out ^[5]. Except for the frontier collector, it is also necessary to create an explored collector to collect the node has been visited by using the array list. Then put the initial state into the frontier. The following part will be put in a do-while block. In the beginning, the element pulled from the frontier will be regarded as the current node. To find out which nodes are connecting to the current node, it must get the array index of the current node in the character array. The index can help fixate the x coordinate or the y coordinate in the map array; then the connected nodes will be found by traversing the array. If the number equals 5 and this node is neither in the explored node list nor the frontier list, then add the node that the index in character array corresponds with the loop index into the frontier list. After that, put the visited node into explored node list. The loop will stop when the current node is the goal node, or the frontier is empty. If the frontier is empty, at the same time the explored node list does not have the goal node, that means there is no available path.

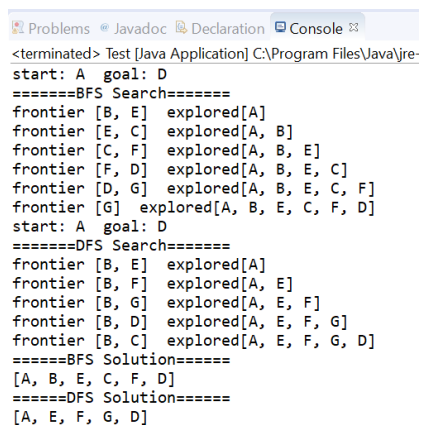
Design: Part Two

The best first search is a kind of heuristic algorithm. It can be considered as an improvement of the breadth-first search. Based on the BFS algorithm, it will estimate the value between the current node and the next node by estimate function, then it will choose the node that has minimum value to traverse until getting the goal node, or all nodes have been

visited ^[6]. The last one is the A* search. It also a kind of heuristic algorithm. The benefit of the A* search is it always can find the shortest path if there is an available way. The key point of the A* search is a formula, which is $F = G + H$. The G means the movement cost that from the initial state to every next node. Moreover, the H means the estimated length from the current node to the goal state. Therefore, it can find the best way by traversal and the selection of the minimum F ^[7]. As for the implementation; the general search functions are very similar to the DFS and the BFS search algorithm, the difference is how to control the order of the frontier. The order will affect the traversal sequence, in other words, it will change the future path directly. In the best first search and A* search algorithm, the priority queue will be a good choice as a frontier collector. The priority queue can store data just like stack and queue. Moreover, it can control the order of the elements in it. Moreover, the user can edit the rule of order to control the sequence ^[5]. In the best first search implementation, the sequence of the frontier collector is ordered by the length between every node and the goal node. In the priority queue, it will not order all elements in it, but it will put the smallest one in the top place of the queue. Therefore, the node that has the smallest length to the goal node will be polled firstly every time. In this programme, there is a node class; it has element x, element y, element length and element value, respectively represent the coordinate x and y, the length to the goal node and the character of the node. Except for the getter, setter and constructor method, there is a method called compareTo which is used to compare the node's length. At the beginning of the processing, it will create node objects depends on the map length and find the initial state and the goal state. At the same time, the elements of the node will initialise. Then, like the processing of BFS, adding the connected nodes into the frontier and polling the frontier to find a goal. In terms of A*, the only different thing is there is another element, the past cost. Thus, the sequence of the priority queue should order by the past cost plus the length between the current node and the goal state. Both the best first search and the A* search is the heuristic algorithm. The estimate function of the BestFS is h, and the estimate function of the A* is g+h. Therefore, the best first search can find the path quickly, but it may not the best path. The A* will always find the best path between the initial state and the goal state.

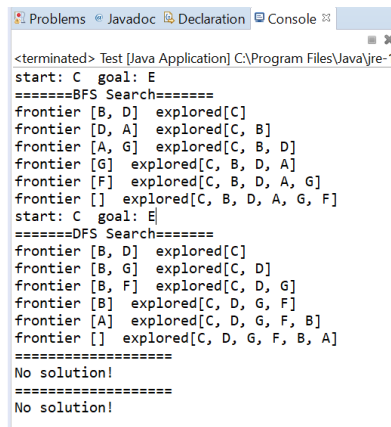
Examples and Testing

The program was compiled by eclipse, so the test example is also made in eclipse. There are three class files in the package, Search1, Search2 and Test. The implementation of the DFS and the BFS is in search1 file. The other two algorithms are in Search2 file. The test executes in the Test file. The result shows in following picture1 is for the map6, it shows the frontier, explored list and the path. If there is no available path from the initial state to the goal state, the final result will alert there is no solution like the picture2 shows. To examine if the result is correct, I draw the search tree by myself on the paper, then compare the node list to the result. However, I can just do this way if there are not too many nodes like in map6 and map7. Although I did not test the case that has 15 nodes in the map by myself, I compared the result to other people's result. Compared the DFS and the BFS, through the search of nine maps, the number of the DFS visited nodes always less than the number of the BFS visited nodes. Therefore, the DFS search will be better than the BFS search if every path cost is the same. The order of the node that added into the frontier is decided by the alphabetical order cause this order is the traversal order of the node list which is from the first alphabet(A) to the last alphabet of the list.



```
<terminated> Test [Java Application] C:\Program Files\Java\jre-
start: A goal: D
=====BFS Search=====
frontier [B, E] explored[A]
frontier [E, C] explored[A, B]
frontier [C, F] explored[A, B, E]
frontier [F, D] explored[A, B, E, C]
frontier [D, G] explored[A, B, E, C, F]
frontier [G] explored[A, B, E, C, F, D]
start: A goal: D
=====DFS Search=====
frontier [B, E] explored[A]
frontier [B, F] explored[A, E]
frontier [B, G] explored[A, E, F]
frontier [B, D] explored[A, E, F, G]
frontier [B, C] explored[A, E, F, G, D]
=====BFS Solution=====
[A, B, E, C, F, D]
=====DFS Solution=====
[A, E, F, G, D]
```

Picture1



```
<terminated> Test [Java Application] C:\Program Files\Java\jre-1
start: C goal: E
=====BFS Search=====
frontier [B, D] explored[C]
frontier [D, A] explored[C, B]
frontier [A, G] explored[C, B, D]
frontier [G] explored[C, B, D, A]
frontier [F] explored[C, B, D, A, G]
frontier [] explored[C, B, D, A, G, F]
start: C goal: E
=====DFS Search=====
frontier [B, D] explored[C]
frontier [B, G] explored[C, D]
frontier [B, F] explored[C, D, G]
frontier [B] explored[C, D, G, F]
frontier [A] explored[C, D, G, F, B]
frontier [] explored[C, D, G, F, B, A]
=====
No solution!
=====
No solution!
```

Picture2

As for the test of the best first search, in the test processing, I changed the toString method to show out the length between every node in the frontier and the goal state. In this way, I can see if the frontier order is right or not. As long as the top of the frontier is the node that has the shortest length, the way will be the correct path. Besides, I also figured out the length from the node to goal state by myself and draw the search tree to compare with the result.

The test for A* is similar to the way mentioned above. I changed the toString method to check the f value and the order of the frontier, cause the f is the key point of the A* search algorithm.

Running

There two jar files and other files in the folder. The program can run by the jar file. The two jar files named Search1 and Search2. The main method in Search1 is the Search1 that executes the DFS and the BFS algorithms. By the command "java -jar Search1.jar [any param]" to run the Search1 jar file. The maps have been initialised in the file. There are nine maps, so the user can input the map to the param place of the method through using param name like map1, map2. If the user inputs the wrong param or do not input any param, it will alert that please input the right param. The Search2 jar file includes the implementation of the BestFS and the A* algorithm. Due to the BestFS and the A* need two maps at the same time. It should provide two params when calling the method. Just like the first one, by the command "java -jar Search2.jar [any param]" to run the program. But user have to input two params for two maps. Another map is called loc and it also has been initialised in the file. There are nine maps in the loc file as well. It will show alerts when the user input the wrong params. The result will show the frontier and the explored node list whenever the new node is visited. Moreover, it will print out the two solutions together in order to find the difference between algorithm.

Bibliography

1. Zhang Q, Qiu H. A dynamic path search algorithm for tractor automatic navigation. Transactions of the ASAE. 2004;47(2):639.
2. Automobile navigation system using real time spoken driving instructions. United States patent US 5,177,685. 1993 Jan 5.
3. Tarjan R. Depth-first search and linear graph algorithms. SIAM journal on computing. 1972 Jun;1(2):146-60.
4. Beamer S, Asanović K, Patterson D. Direction-optimizing breadth-first search. Scientific Programming. 2013;21(3-4):137-48.
5. Budd T, Budd TA. Classic data structures in Java. Addison-Wesley; 2001.
6. Korf RE. Linear-space best-first search. Artificial Intelligence. 1993 Jul 1;62(1):41-78.

7. Russell SJ, Norvig P. Artificial intelligence: a modern approach.