Search Algorithms

Tutorial and Implementation

by Kontopoulos Ioannis

Student of Harokopio University of Athens, Greece

This is a tutorial for the basic and most famous search algorithms in computer programming. It is accompanied by a Java project which implements them, as well as the library (*.jar) and Javadoc. Feel free to use it or expand it.

Tutorial

Depth-First Search

Depth-First is an unintelligent algorithm (i.e. no heuristic is used) which starts at the initial state and proceeds as follows :

1. Check if current node is a goal state.
2. If not, expand the node, choose a successor of the node, and repeat.
3. If a node is a terminal state (but not a goal state), or all successors of a node have been checked, return to that node's parent and try another successor. (The failed states need not stay in memory.)

The primary strength of the depth first search is its efficient use of memory—relatively few nodes need to be kept track of at any given time. However, the depth first search has a number of important weaknesses :

1. There is no guarantee that the solution this algorithm finds will be the "cheapest," i.e. the one that requires the least number of steps from the initial state. For example, in the above scenario, all possibilities using the initial state's left successor will be considered before any possibilities using the right successor. A solution starting with the left option and requiring 10, 100, or 1000 steps would be returned, even if the right option solved the problem in 1 step!
2. In some cases, a depth first search may get caught in an infinite loop and never find a viable solution. Take a maze, for instance; even if the search is programmed to never go back the way it came, the search could easily go around in a circle without realizing it is covering the same ground over and over again. Since a depth-first search evaluates possibilities in an arbitrary—but usually consistent—order (for instance, in navigating a maze, the algorithm might always attempt north, then east, then south, then west), if it arrives at a state it doesn't realize it's seen before, it will always make the same choice and keep looping.

Thus, depth first search is ideal when:

- Efficient memory use is important.
- Any solution to the problem will do.
- There is little chance of infinite looping

Breadth-First Search

Also an unintelligent searching algorithm, breadth-first searching expands a node and checks each of its successors for a goal state before expanding any of the original node's successors (unlike depth-first search).

This last observation is key in understanding the primary usefulness of the breadth first search—the most efficient solution (or, at least, a solution at least as efficient as any other solution) is guaranteed, since all possible states reached in less moves have already been checked, and are known not to be goal states. Related is the feature that a breadth-first search cannot get stuck in an infinite loop; since all possible states of a certain level are checked in order, the state(s) which break out of the loop will be evaluated and expanded (so will the ones in the loop, but unlike depth-first, there is no chance that the loop will be the only thing being checked—unless, of course, the only possible result from the initial state is an infinite loop).

Breadth-first's primary drawback is in its use of memory. Since all possible paths are being considered step-by-step in tandem, nodes are essentially never deleted from memory until a goal is found. Also, since a breath-first search goes "deeper" much more slowly than a depth-first search, a problem with many solutions of all approximately the same depth will take much longer to solve with a breadth-first approach.

Best-First Search

Best first search is an intelligent search algorithm which makes use of a heuristic to rank the nodes based on the estimated cost from that node to the goal. First, the initial node is placed in an open list, then it is checked for goal conditions. If it is not a goal state, it is removed from the open list (making the open list momentarily empty), and its child nodes are put in the open list. The heuristic is applied to these child nodes, and the node that is estimated to be the best is then taken out of the open list and evaluated. If it is not a goal, the node is placed in the closed list, its children are put in the open list, and the heuristic is used to select the node in the open list that now appears to be the best in the list. This continues until a goal is found or the open list is empty.

A simple case of the best first search is the greedy best first search, where the heuristic simply chooses the node that appears to be closest to the goal node from the current node. This will provide the best solution sometimes, but may cause problems in others-for instance, when plotting a trip on a map, two points may seem very close together when only distance is taken into account, but actually traveling between them may require additional time or may even be impossible. This heuristic is called greedy because, in a sense, it greedily grabs at the best solution without attempting to calculate the long term costs. Its salient characteristic is that in its attempt to get to the goal, it starts from the current node, not the start node, to choose a path towards the goal.

A* Search

Algorithm A* is a best-first search algorithm that relies on an open list and a closed list to find a path that is both optimal and complete towards the goal. It works by combining the benefits of the uniform-cost search and greedy search algorithms. A* makes use of both elements by including two separate path finding functions in its algorithm that take into account the cost from the root node to the current node and estimates the path cost from the current node to the goal node.

The first function is g(n), which calculates the path cost between the start node and the current node. The second function is h(n), which is a heuristic to calculate the estimated path cost from the current node to the goal node. F(n) = g(n) + h(n). It represents the path cost of the most efficient estimated path towards the goal. A* continues to re-evaluate both g(n) and h(n) throughout the search for all of the nodes that it encounters in order to arrive at the minimal cost path to the goal. This algorithm is extremely popular for pathfinding in strategy computer games.

The process for A* is basically this :

1. Create an open list and a closed list that are both empty. Put the start node in the open list.
2. Loop this until the goal is found or the open list is empty :
   i. Find the node with the lowest F cost in the open list and place it in the closed list.
   ii. Expand this node and for the adjacent nodes to this node :
      a. If they are on the closed list, ignore.
      b. If not on the open list, add to open list, store the current node as the parent for this adjacent node, and calculate the F,G, H costs of the adjacent node.
      c. If on the open list, compare the G costs of this path to the node and the old path to the node. If the G cost of using the current node to get to the node is the lower cost, change the parent node of the adjacent node to the current node. Recalculate F,G,H costs of the node.
3. If open list is empty, fail.

The terms locally finite, admissible, and monotonic all aid in the understanding of when A* can be expected to be complete, meaning that it finds a solution, and optimal, meaning that it finds the solution with the lowest path cost. A locally finite graph is one where none of the nodes on the graph have an infinite branching factor, thus none of the node paths branch forever. A branching factor of a node refers to the amount of new nodes that can be expanded from that node.

A heuristic is admissible if it is always optimistic; it either underestimates the path cost to the goal or provides a correct estimate for the path cost to the goal, but it never overestimates the path cost to the goal.

A heuristic is monotonic if in every path from the root to the goal the total estimated path cost does not decrease as the heuristic goes down a node tree. Non-monotonic heuristics can be made monotonic by the pathmax equation, which compares the estimated path cost of a node with the estimated path cost of its parent node. It then uses the higher path cost for estimation. Therefore, if the heuristic's estimated path cost decreases from one node to its child node, the pathmax equation uses the path cost of the parent node so that it is not evaluated as decreasing.

A* is complete and optimal on graphs that are locally finite where the heuristics are admissible and monotonic.

A* must be locally finite, because if there exist an infinite amount of nodes where the estimated path cost, f(n), is less than the actual goal path cost then the algorithm could continue to explore these nodes without end, and it will be neither complete nor optimal.

How does monotocity affect A*'s completeness? Because A* is monotonic, the path cost increases as the node gets further from the root. Contours can be drawn to show areas where the estimated path cost, the f(n), for the nodes inside the areas are lower than or equal to the path cost for the outer bounds of the contours. These contours can be drawn as larger and larger areas that increase outwards as the f(n) for the outer bound of these contours increases. The first solution found is optimal since it is the first band where the f(n) for the contour is equal to the path cost for the goal. All the contours outside of this solution will have a higher f cost.

A*'s optimality is proved by contradiction. First, it is assumed that g is an optimal goal state with a path cost of f(g), that s is a suboptimal goal state with a path cost of g(s) > f(g), and that n is a node on an optimal path to g. We are assuming that A* selects s (the suboptimal goal) instead of n (the node on the optimal path) from the open list.

Since h is admissible, (optimistic), f(g) >= f(n). (The actual path cost is greater than or equal to the path cost estimated by the heuristic at n.)

If n is not chosen over s for expansion by A*, f(n) >= f(s). (The heuristic chooses the node with the lowest estimated F path cost.)

Thus, f(g) >= f(s).

Since s is a goal state, h(s) = 0. (The estimation from the current node to the final node must be 0.)

So f(s) = g(s). (f(s) = g(s) + h(s).)

Thus, f(g) >= g(s). This contradicts the statement that S is suboptimal so it must be true that A* never chooses a suboptimal path. Since A* only can have as a solution a node that it has selected for expansion, it is optimal.

Branch And Bound Search

The branch and bound algorithm is a pruning technique that allows us to examine a given path, decide whether to keep considering it or to disregard it based on the evaluation of the path in comparison with others so far. As mentioned above, the branch and bound search algorithm considers the best path so far, while other algorithms look at the best successor step in a given path. The path cost for each route explored is stored and used to sort the choices.

The branch and bound is a depth first search based algorithm that considers the best path so far, backtracking whenever it encounters a node whose value is larger than its current upper bound. The algorithm terminates when all paths have been explored or pruned. The algorithm is both optimal and complete, as in it will search all the possible solutions and find the best one. The advantages of the search are low memory use and the availability of the best partial path so far.

In short, branch and bound is an algorithm technique to find the optimal solution by keeping the best solution found so far. If a partial solution examined at a given point cannot do better than the best, work on it is abandoned.

Complexities

Assuming that b is the branching factor, d the route length to the solution and m the max search tree depth, we have the following table :

	Depth-First	**Breadth-First**	**Best-First**	**A***	**Branch and Bound**
Time complexity	$O(b^m)$	$O(b^{d+1})$	$O(b^m)$	$O(b^m)$	*
Space Complexity	$O(bm)$	$O(b^{d+1})$	$O(b^m)$	$O(b^m)$	*
Optimal	No	Yes	No	Yes	Yes
Complete	No	Yes	No	Yes	Yes

*In the worst case, a Branch and Bound search algorithm must explore every node in a state space. Thus, the worst-cast complexity of Branch and Bound is the size of a state space. On the other hand, if the lower-bound cost function used is exact, the complexity is linear in the length of the path from the initial node to the goal node.

Implementation

Why Java ?

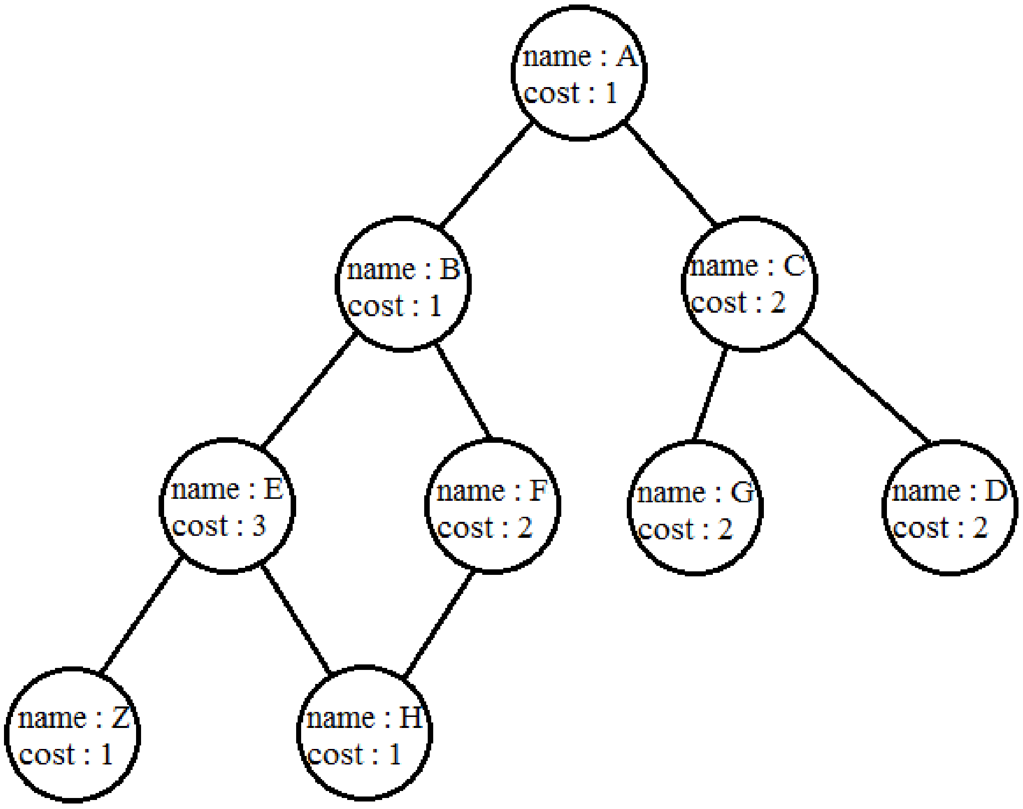
The implementation of the search algorithms is in the Java programming language. The reason is, that Java is one of the easiest languages to learn, especially for beginners and is one of the most famous. So, for a tutorial like this, I think it` s the best choice.

Basic Code Concept

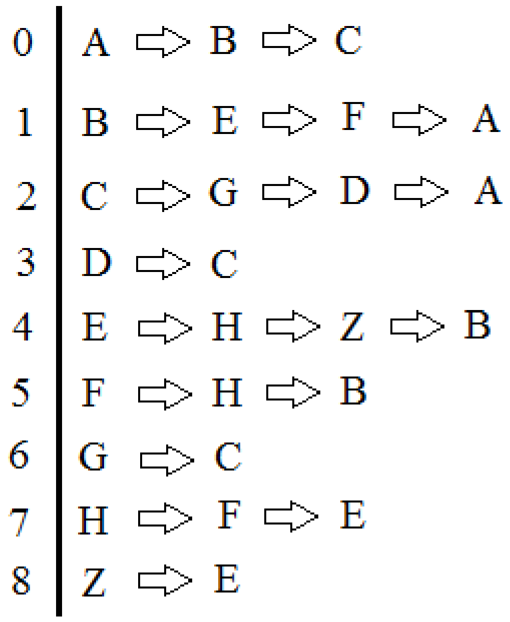
For starters, in order to comprehend the code, some explanation upon it will be given shortly. Each vertex must have some specific fields for the implemented algorithms: name, adjList, cost, costSoFar. The name and the adjacency list is necessary for every algorithm (adjList is of type Neighbor). The cost is needed for the Best-First, A* and Branch and Bound algorithms. The costSoFar is used by the A* algorithm. So, I created a class named Vertex which contains those fields.

To make the adjacency list possible we need another class called Neighbor which contains a number (integer) and a "next" variable (of type Neighbor).

To make the graph, I created a class called Graph which contains an ArrayList that holds all the vertices of the graph. The number in the Neighbor class mentioned earlier is an index for the position of the vertex in the ArrayList of the graph. In the Graph class I have already implemented a default graph, just used for testing purposes. You can see it below and is created in the initializeDefault() method :



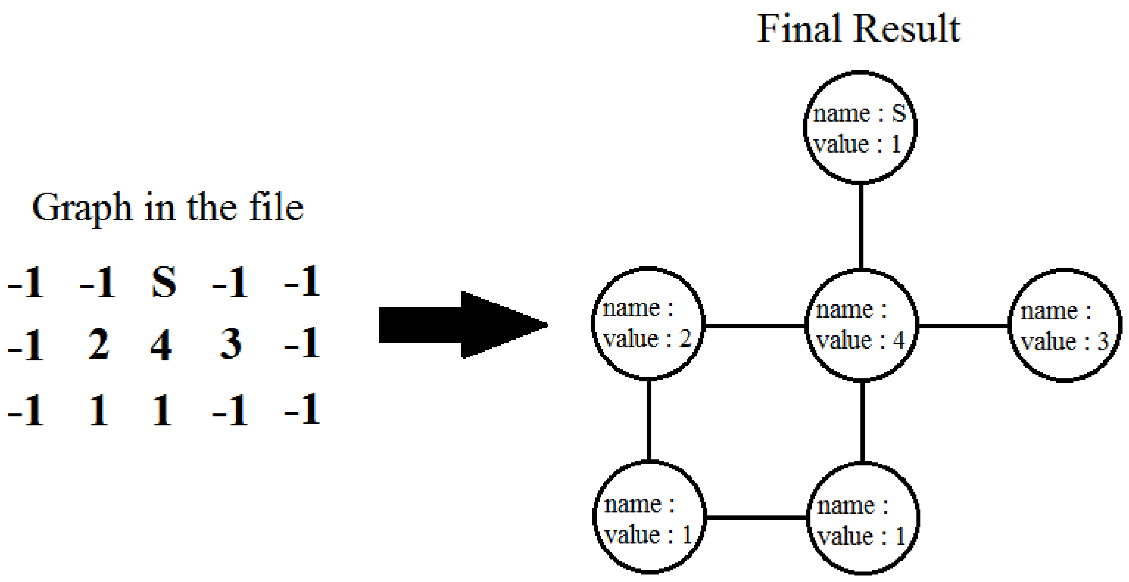
Based on the default graph contained in the Graph class and based on the variables mentioned earlier according to the adjacency list, you can imagine the ArrayList of the class as shown below :



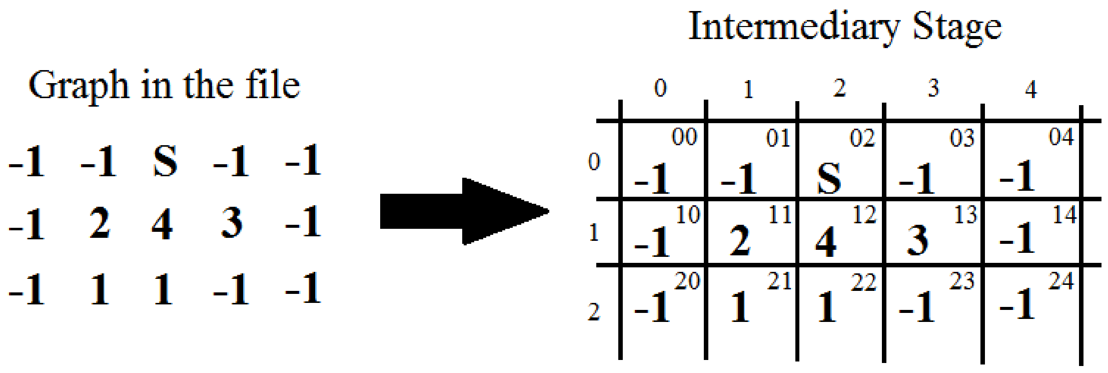
Where the numbers on the left are the indexes for the position of each vertex in the ArrayList and the letters right next to them are the vertices with their neighbors. For example, our graph contains 9 vertices in total, where vertex B (in position 1 of the ArrayList) has the vertices E, F, A as neighbors. You can see in the graph shown earlier that vertex B is connected to those vertices.

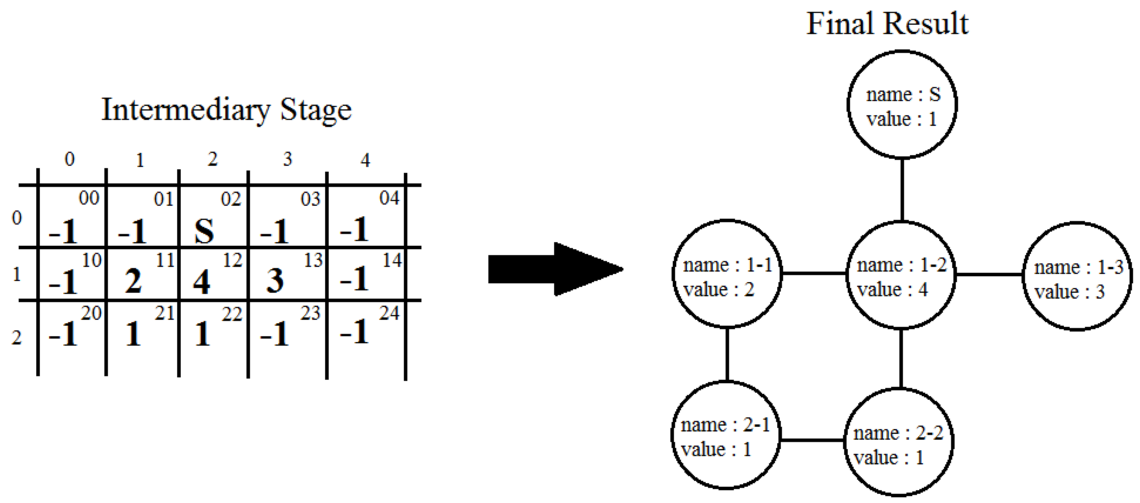
Reading from Files

The library has the ability to read a graph from file. The next picture shows the form of the file and the final graph which will be created based on that file. On the left we can see the file and on the right we can see the final result. The -1 number means that there is no vertex. When a vertex has a name like S, G its cost will be 1 by default, otherwise it will be given the cost indicated by the file. In order to read the file, the graph is going through some stages first, which are explained later :



So, in order to read from the file I create an imaginary grid with coordinates. These coordinates are used for the adjacencies, but also for naming the vertices with no name. Below you can see the intermediary stage as well as the final result :





Now, the adjacency goes like this. By observing the grid we can see that vertices on the same line of the file which are neighbors, have a difference equal to 1 on the right coordinates and no difference on the left ones. Vertices on the same column of the file and which are of course neighbors, have a difference of 1 on the left coordinates and no difference on the right ones. From that we can create the edges and extract the adjacencies.

Heuristic Functions

The heuristic functions that were used for the algorithms are the ones that are being used most often.

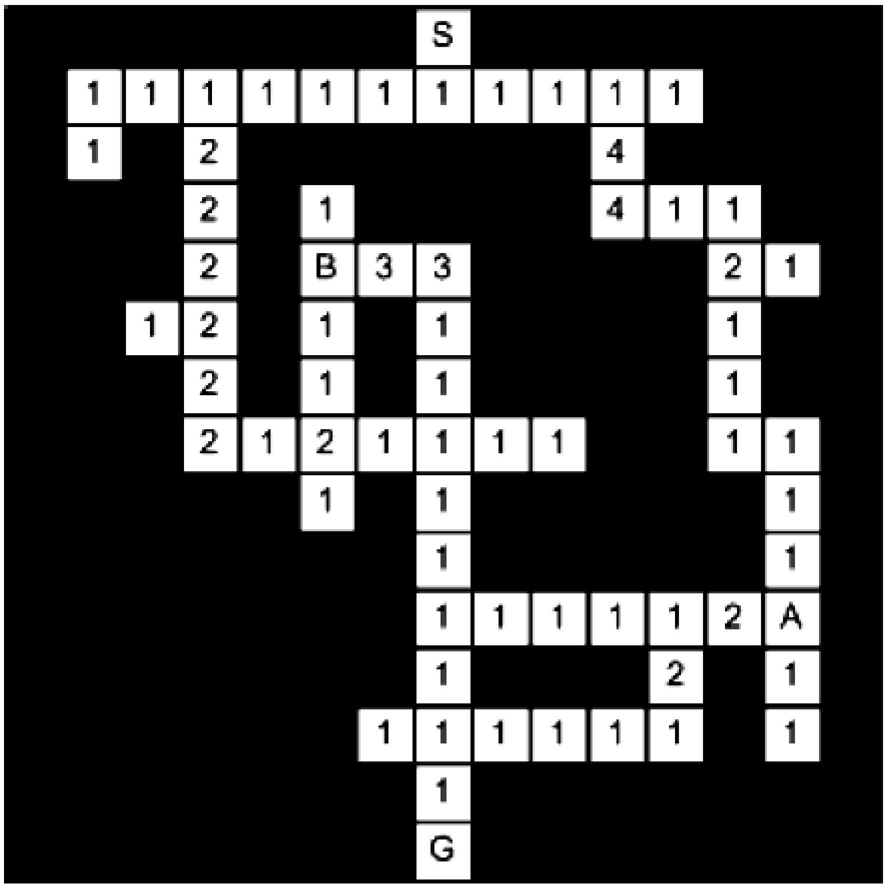
Specifically, for the **Best-First** algorithm we examine the vertices based on the cost of the vertex and we choose the one with the lowest cost, h(n). For instance, if we have the vertices A with cost 1, B with cost 3 and C with cost 2, the algorithm will choose vertex A, because he has the lowest cost, h(n) = 1.

For the **A*** algorithm, we examine the vertices based on their cost h(n), and the total cost from the beginning of the route up to that point, g(n). For example, if we have the vertices A with cost 1 and cost so far 5, B with cost 3 and cost so far 2 and C with cost 2 and cost so far 2, the algorithm will choose vertex C, because the sum cost + cost so far = 4 (f(n) = h(n) + g(n) = 4), is the lowest.

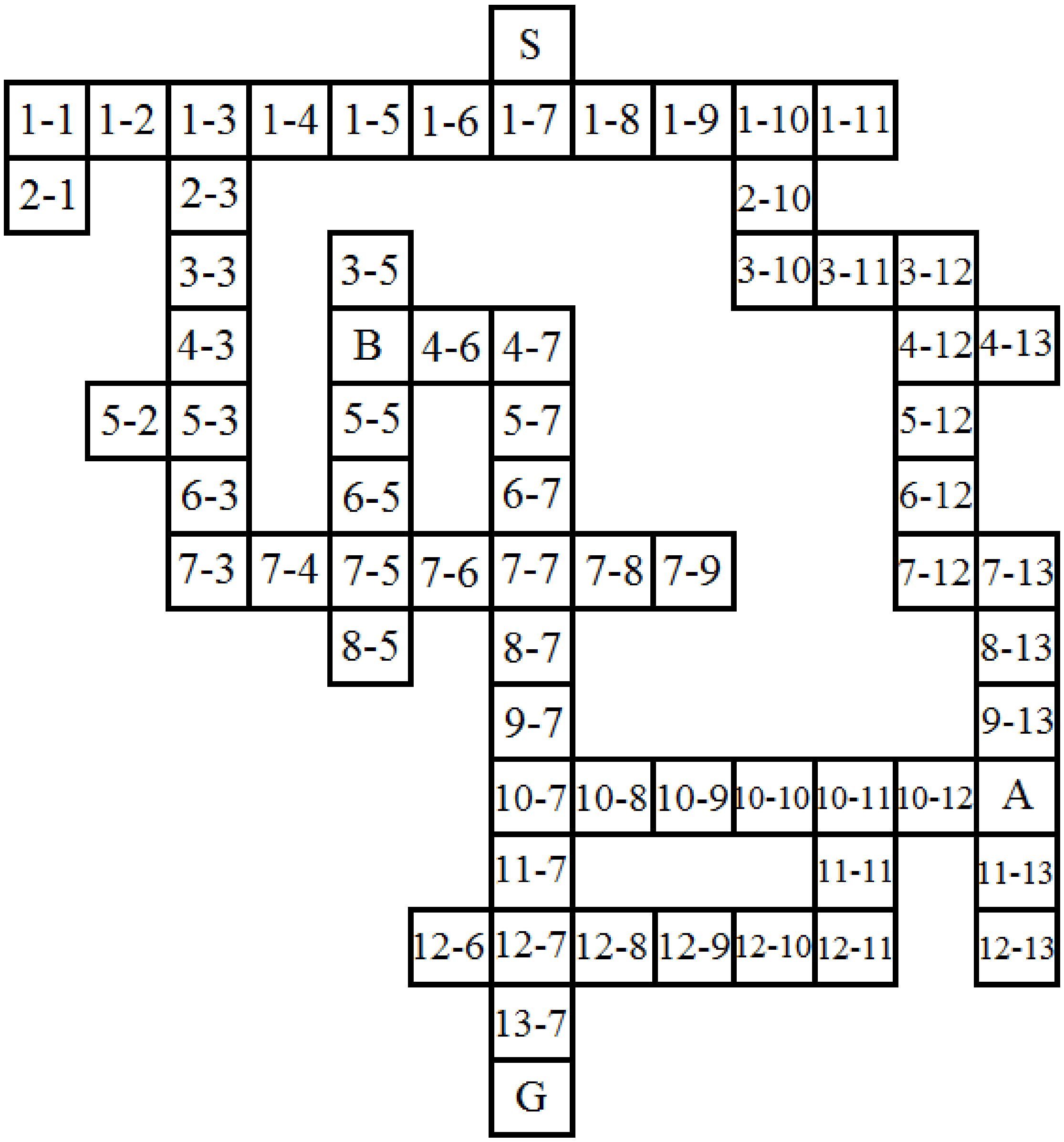
For **Branch and Bound** algorithm, the sorting in the priority queue is based on total route cost, meaning the sum of the vertices` costs. There is another way that is implemented but not used, and it does the sorting based on the route length, i.e. the number of the vertices in the route.

Maze

There is a file inside the project called maze.txt. This file has a quite large maze and it is shown below :



And you can see in the picture below that this maze will be converted by the library to this :


S
1-1
1-2
1-3
1-4
1-5
1-6
1-7
1-8
1-9
1-10
1-11
2-1
2-3
2-10
3-3
3-5
3-10
3-11
3-12
4-3
B
4-6
4-7
4-12
4-13
5-2
5-3
5-5
5-7
5-12
6-3
6-5
6-7
6-12
7-3
7-4
7-5
7-6
7-7
7-8
7-9
7-12
7-13
8-5
8-7
8-13
9-7
9-13
10-7
10-8
10-9
10-10
10-11
10-12
A
11-7
11-11
11-13
12-6
12-7
12-8
12-9
12-10
12-11
12-13
13-7
G