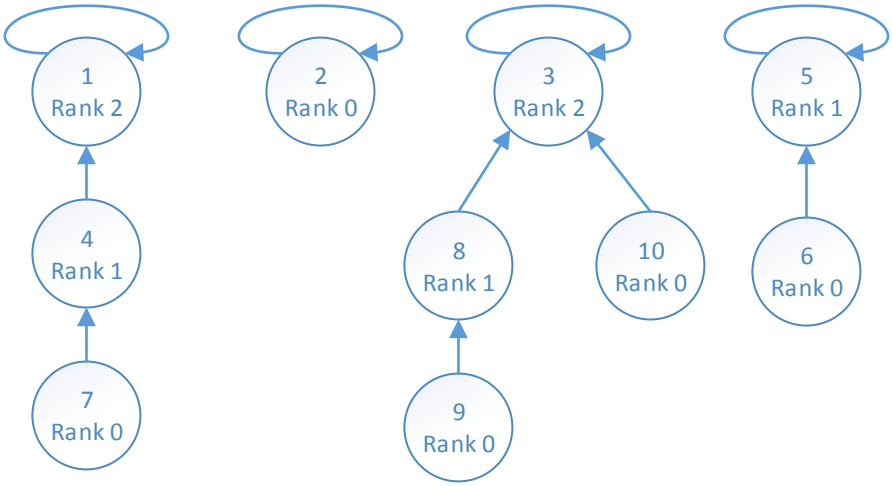
Problem #1

Initial Union Find Data Structure:

Index	1	2	3	4	5	6	7	8	9	10
Parent	1	2	3	1	5	5	4	3	8	3
Rank	2	0	2	1	1	0	0	1	0	0

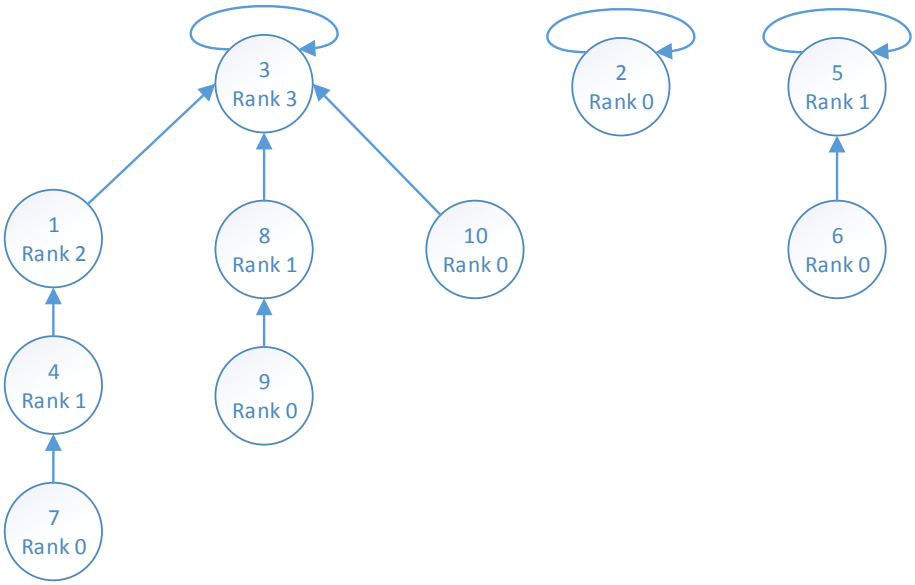
Part A: Draw the Tree Representation of the Set

Problem #1 – Initial Data Structure



Part B: Redraw the resulting trees following `Union`(7,8) using union by rank but without path compression. Provide the value of the *rank* and *parent* arrays after the `Union` operation.

Problem #1 – After Union(7,8)

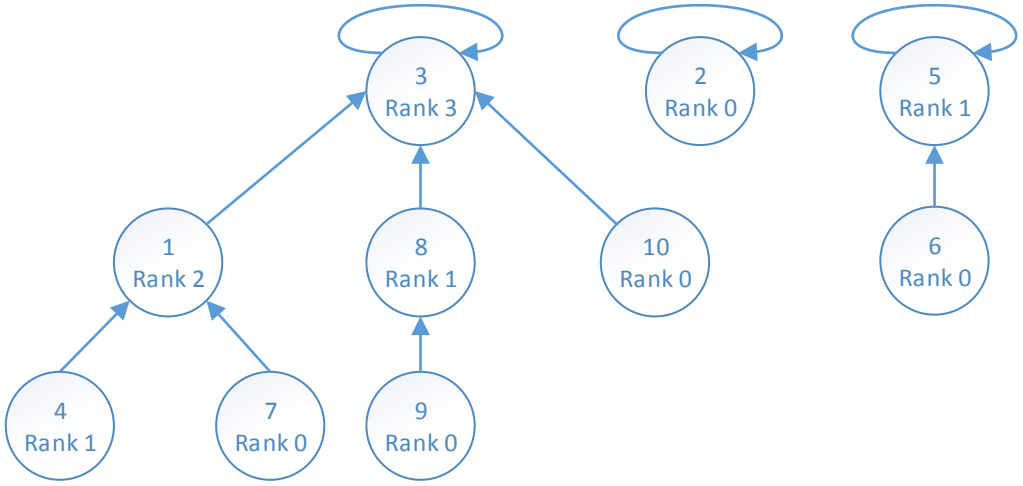


The updated *parent* and *rank* arrays after the `Union` operation are:

Index	1	2	3	4	5	6	7	8	9	10
Parent	3	2	3	1	5	5	4	3	8	3
Rank	2	0	3	1	1	0	0	1	0	0

Part C: Redraw the resulting trees following `Union`(7,8) using union by rank with path compression. Provide the value of the *rank* and *parent* arrays after the `Union` operation.

Problem #1 – After Union(7,8) with Path Compression

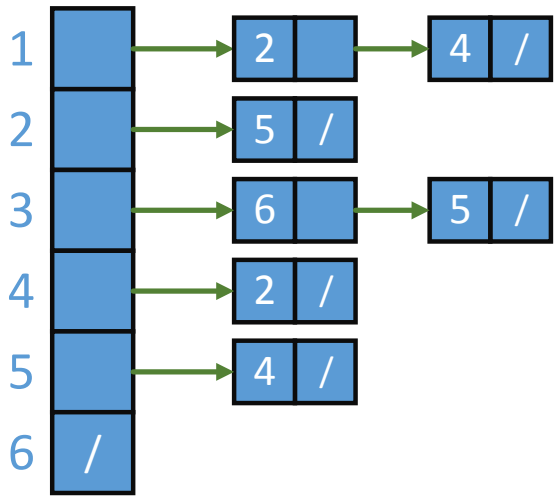


Index	1	2	3	4	5	6	7	8	9	10
Parent	3	2	3	1	5	5	1	3	8	3
Rank	2	0	3	1	1	0	0	1	0	0

Problem #2 – Exercise 22.1-1 (Page 592)

Given an adjacency list representation of a directed graph, A) how long does it take to compute the out-degree of every vertex. B) How long does it take to compute the in-degrees.

Problem #2 – Adjacency List Example for a Directed Graph



Part A: Out Degree

A vertex's out-degree in a directed graph is defined as the number of outward edges that begin from that given vertex. In an adjacency list, a vertex's out-degree is determined by iterating through the linked list associated with that vertex and incrementing a counter for each vertex in the list. Hence, for a given vertex *v,* the running time is $\theta(degree(v) + 1)$. For example in the figure above, the procedure to find the out-degree for vertex 2 would be start at the head of the linked list at 2, go to 5 in the linked list while incrementing the counter by 1, and then terminating since the NULL pointer was reached. However, for vertex *6*, which has an out-degree of 0, its running time is $\theta(1)$ since the head pointer must still be checked.

Therefore, the running time to find the out-degree for all vertices requires that all of the linked lists for all the vertices be traversed. Hence it is:

$$\theta\left(\sum_{v \in V}(degree(v) + 1)\right)$$

This simplifies to: $\theta(|V| + |E|)$.

Part B: In-Degree

The running time for in-degree is also $\theta(|V| + |E|)$, but the algorithm is different. In contrast to the out-degree algorithm, which kept a single counter as it iterated through each linked list, the in-degree algorithm requires that $|V|$ counters be kept, with the individual counters representing the in-degree of each vertex. The algorithm iterates through all $|V|$ linked lists. For a given node, its data/payload is examined; the counter associated with the data/payload is then incremented by 1 since another inward edge was found.

Problem #3 – Exercise 22.1-3 (Page 592)

The ***transpose*** of a directed graph G = (V,E) is the graph $G^T = (V,E^T)$, where

$$E^T = \{(v,u) \in V\ x\ V : (u,v) \in E\}$$

Thus, G^T is G with all edge directions reversed. Describe an efficient algorithm for computing G^T from G, for both the adjacency-list and adjacency matrix representations of G. Analyze the running times of your algorithms.

Part A: Below is an algorithm written in Java that transposes an adjacency matrix. The algorithm has two nested for loops, which iterate through the input adjacency matrix (named `adj_matrix`). It then uses the standard transpose formula of:

$$A_{i,j} = A^T_{j,i}$$

to transpose the matrix.

```
static int[][] Transpose_Adjacency_Matrix(int[][] adj_matrix){
	int i, j;
	int v = adj_matrix.length; //---- Get the size of the array
	int[][] transpose_adj_matrix = new int[v][v];

	//---- Iterate through the matrix and swap where Aij = ATji
	for(i=0; i < v; i++){
		for(j=0; j<v; j++){ //---- Can start at i+1 since no need to transpose diagonal

			//---- Add cell to the matrix
			transpose_adj_matrix[j][i] = adj_matrix[i][j];
		}
	}

	System.out.println("Transposed matrix is:");
	Print_Random_Adjacency_Matrix(transpose_adj_matrix);
	System.out.println("\n\n");

	//----- Return the adjacency matrix
	return transpose_adj_matrix;
}
```

The running time of this algorithm is bounded by the two for loops, which each run $|V|$ times, where *V* is the set of vertices. As such, the running time is: $\theta(|V|^2)$.

Part B: Below is an algorithm written in Java that transposes an adjacency list.

```
static LinkedList<Integer>[] Transpose_Adjacency_List(LinkedList<Integer>[] adj_list){
	int i, j;
	Iterator<Integer> itr;
	int v = adj_list.length;
	LinkedList<Integer>[] transpose_adj_list = new LinkedList[v]; //--- Create array of linked lists

	//---- Initialize the adjacency list
	for(i=0; i < v; i++){
		transpose_adj_list[i] = new LinkedList<Integer>();
	}

	//---- Iterate through the matrix and swap where Aij = ATji
	for(i=0; i < v; i++){
		itr = adj_list[i].iterator();
		while(itr.hasNext()){
			j = itr.next();
			//---- Add cell to transposed list
			transpose_adj_list[j].add(i);
		}
```

```
            }

            System.out.println("Transposed adjacency list as an adjacency matrix is:");
            Print_Adjacency_List(transpose_adj_list);
            System.out.println("\n\n");

            //----- Return the adjacency matrix
            return transpose_adj_list;
        }
```

The first for loop simply initializes the transposed adjacency list; this is more a Java requirement than it is an algorithm running time requirement. The procedure of the algorithm is to iterate through each vertex's linked list. It then adds each retrieved edge to a new linked list but swaps the starting and end points of the directed edge. This is done in the line:

```
transpose_adj_list[j].add(i);
```

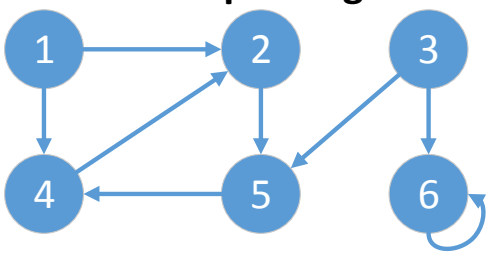
Note how the edge that previously started at the i^{th} vertex is being added to *j*'s adjacency list in the transposed linked list data structure.

This algorithm has a running time of $\theta(|V| + |E|)$. This is because the main for loop iterates through all vertices making its running time $\theta(|V|)$; it also checks all adjacency lists for all vertices which has a running time of $\theta(|E|)$. Hence, the combined running time is $\theta(|V| + |E|)$.

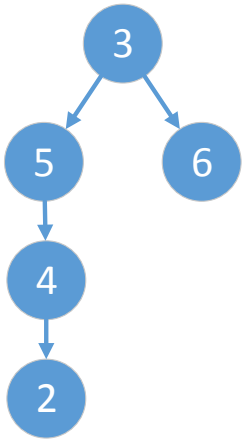
Problem #4 – Exercise 22.2-1 (Page 601)

Show the *d* and π values that result from running the breadth-first search on the directed graph of Figure 22.2(a), using vertex 3 as the source.

Undirected Graph – Figure 22.2a



Breadth-First Tree



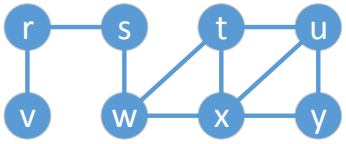
Above is the breadth-first tree for the graph. Below is a table of the distances and parent/predecessor for each vertex.

Vertex#	1	2	3	4	5	6
d	∞	3	0	2	1	1
π	Nil	4	Nil	5	3	3

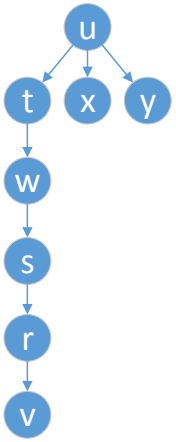
Problem #5 – Exercise 22.2-2 (Page 601)

Show the *d* and π values that result from running the breadth-first search on the directed graph of Figure 22.3, using vertex *u* as the source.

Breadth First Search – Figure 22.3



Breadth First Tree



Above is the breadth-first tree for the graph. Below is a table of the distances and parent/predecessor for each vertex.

Vertex#	r	s	t	u	v	w	x	y
d	4	3	1	0	5	2	1	1
π	s	w	u	Nil	r	t	u	u

Problem #6

Problem Description: Given a dictionary of *n* five letter words, provide an algorithm that allows you to determine the minimum number of one-character changes to proceed from a starting word *s* to a target word *t*.

Graph Description: Assuming a 26 letter alphabet that is case insensitive, each word is one character away from 125 different five letter strings (25*5 = 125). None, some, or all of these related words may appear in the dictionary.

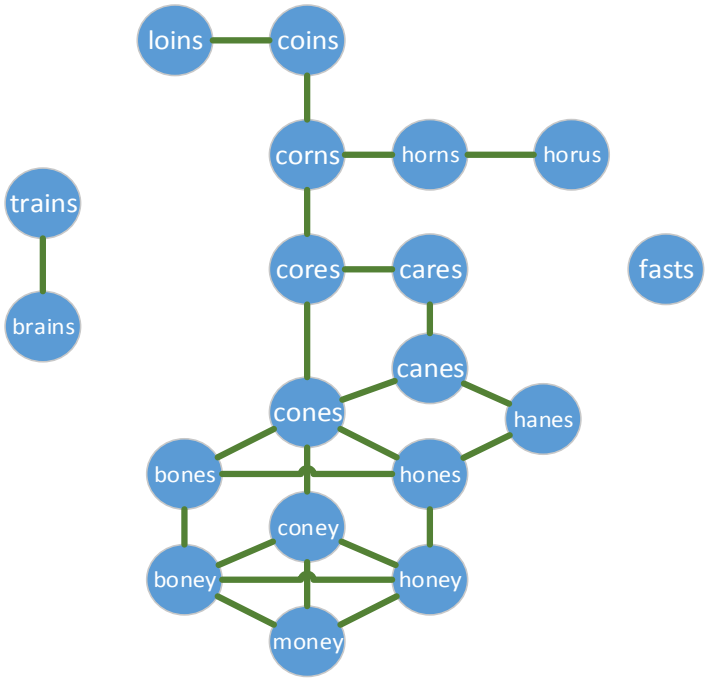
1. Consider each word, *w*, a vertex in an undirected graph[1]. The graph has *n* total vertices because of the *n* words in the dictionary.

2. A vertex representing word *w*' is connected to the vertex representing *w*'' if and only if *w*' and *w*'' are different by only one character.

3. All edges in the graph have weight 1, which represents the one character change.

Below is an graph with 19 (*n*) words in the dictionary. Note that for a given starting word *s*, some words may be unreachable.

Problem #6 – Undirected Graph for *n* Dictionary Words of Length 5



Part A Algorithm Description: Breadth-first search (BFS) is an appropriate algorithm to solve this problem. First, by definition of breadth first search, the algorithm starts at some source vertex *s*. In this problem, the source vertex is the one that contains/represents the start string (also referred to as *s*). Second, breadth-first search correctly computes the shortest path distances between two vertices. As such, given how the graph has been setup and defined, using

1. The graph is undirected because if word *w'* is one character away from word *w''*, then w'' is also one character away from word w'. The makes all edges bidirectional which is simply represented by an undirected graph.

breadth-first search will allow for us to find the minimum number of steps (i.e. one character changes) to go from vertex/string *s* to vertex/string *t*.

Below is a modified version of the BFS pseudo code in the textbook with the modifications in **bold**.

```
One_Character_BFS(G, s, t){
        for each vertex u ∈ G.V – { s }
                u.color = white
                u.d = ∞
                u.π = NIL
        s.color = GRAY
        s.d = 0
        s.π = NIL
        Q = ∅
        while(Q ≠ ∅ and t.d = ∞) //----- Continue running until t is discovered or all reachable vertices checked
                u = DEQUEUE(Q)
                for each v ∈ G.Adj[u]
                        if v.color == WHITE
                                v.color = GRAY
                                v.d = u.d + 1
                                v.π = u.π
                                ENQUEUE(Q, v)
                u.color = BLACK
        return t.d
}
```

The three primary modifications are:

1. Pass the target vertex/string *t* into the modified BFS function.

2. Change the primary while loop to terminate either when the queue *Q* is empty or when vertex *t* is reached (noted by a change in *d* which is the distance from the source vertex). This change makes the algorithm more efficient but does not change its worst-case asymptotic running time.

3. Return *t.d* which is the number of one character changes. If the two words are not reachable based off the graph's dictionary, infinity is returned.

Part B Algorithm Description: Breadth-first search has a running time of:

$$\theta(|V| + |E|)$$

For this modified case, the number of vertices is *n* so this simplifies to:

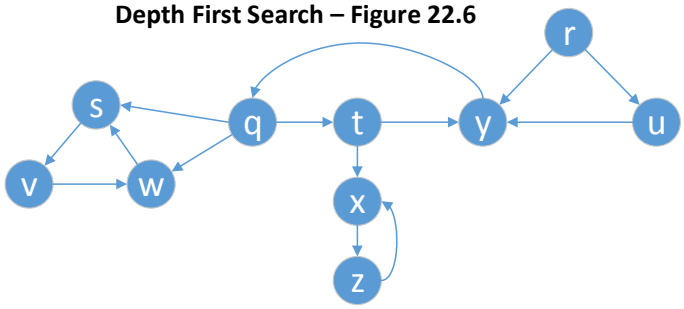
$$O(n + |E|)$$

The reason it is Big-Oh and not Big-Theta is because the BFS can terminate early if *t* is found. The algorithm's running time cannot be simplified further since the number of edges is not known.

Problem #7 – Exercise 22.3-2 (Page 610)

Show how depth first search works on the graph of Figure 22.6. Assume that the for loop of lines 5-7 of the DFS procedure considers the vertices in alphabetical order, and assume that each adjacency list is ordered alphabetically. Show the discovery and finishing times for each vertex, and show the classification of each edge.

Depth First Search – Figure 22.6

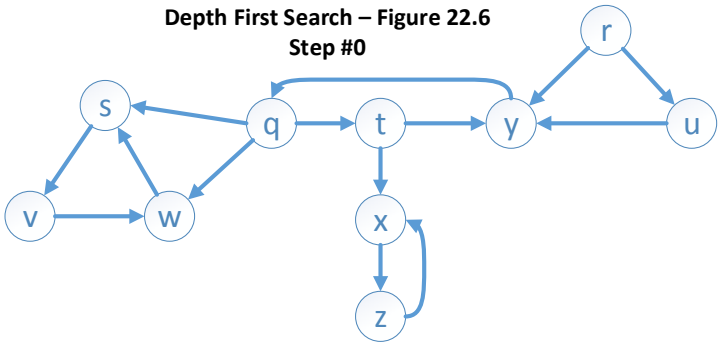


Below is a table of the order that each vertex is discovered and finished.

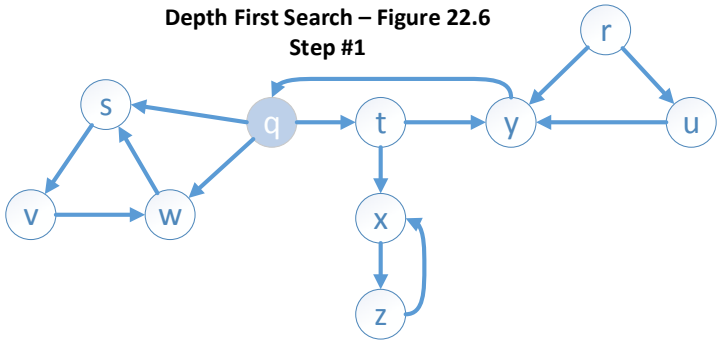
Vertex ID	q	r	s	t	u	v	w	x	y	z
Discovery Time	1	17	2	8	18	3	4	9	13	10
Finish Time	16	20	7	15	19	6	5	12	14	11

In the subsequent progressions, white vertices are undiscovered. A vertex is turned gray once discovered and dark blue once finished. Edges that are tree edges are marked black. All other edges are marked green and given a letter to identify them (A through F). The edge classifications are at the end of the graph progressions.

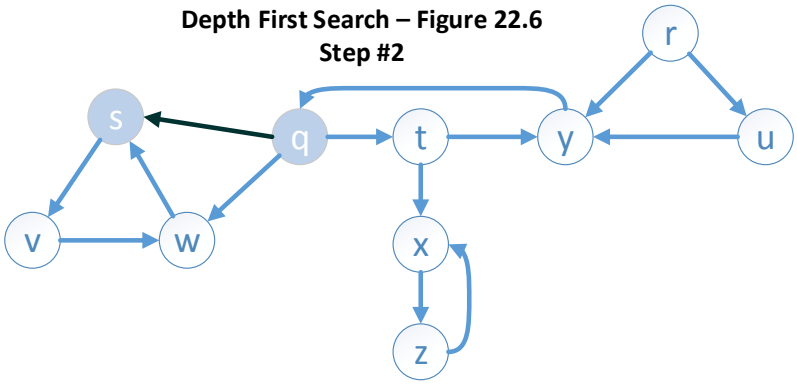
Depth First Search – Figure 22.6
Step #0



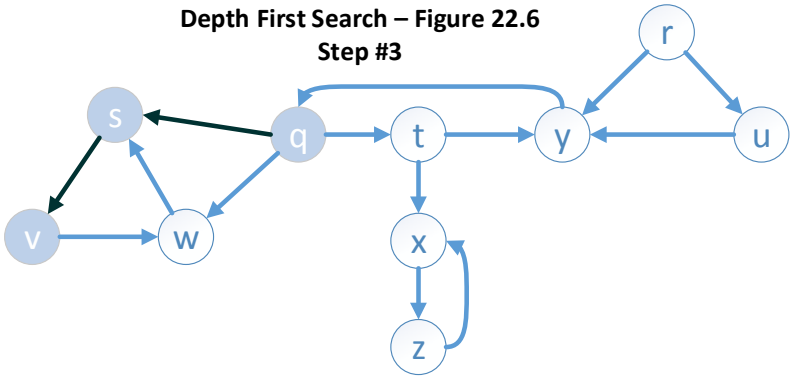
Depth First Search – Figure 22.6
Step #1



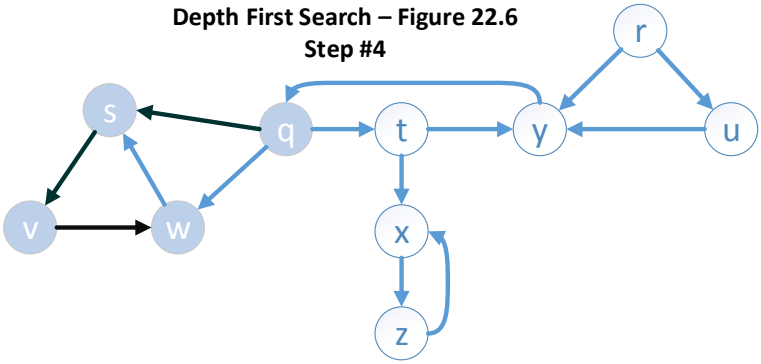
Depth First Search – Figure 22.6
Step #2



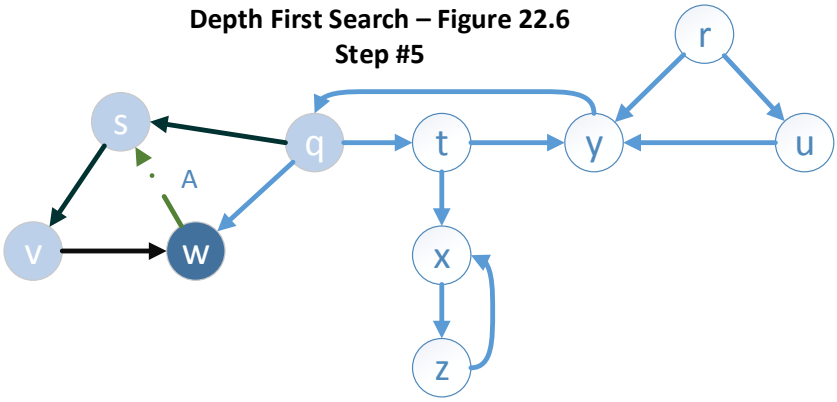
Depth First Search – Figure 22.6
Step #3



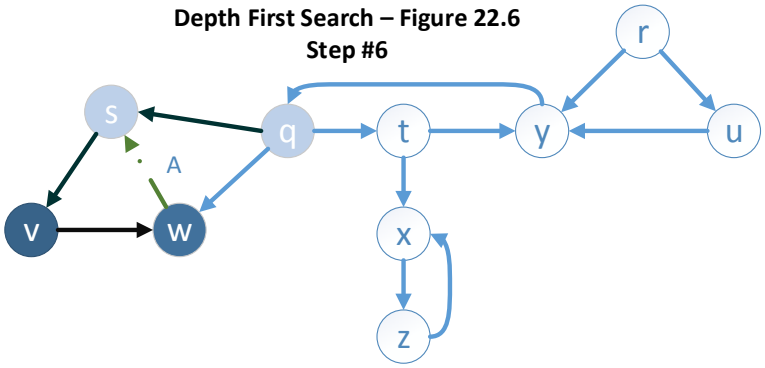
Depth First Search – Figure 22.6
Step #4



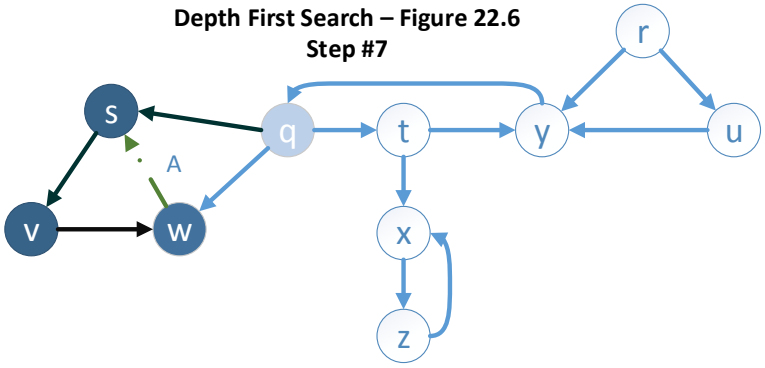
Depth First Search – Figure 22.6
Step #5



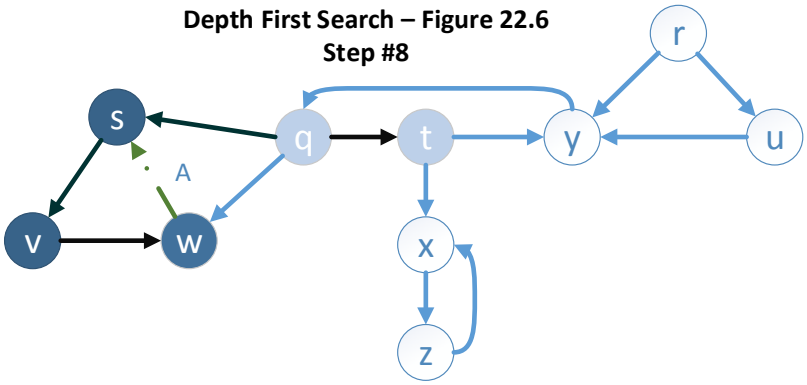
Depth First Search – Figure 22.6
Step #6



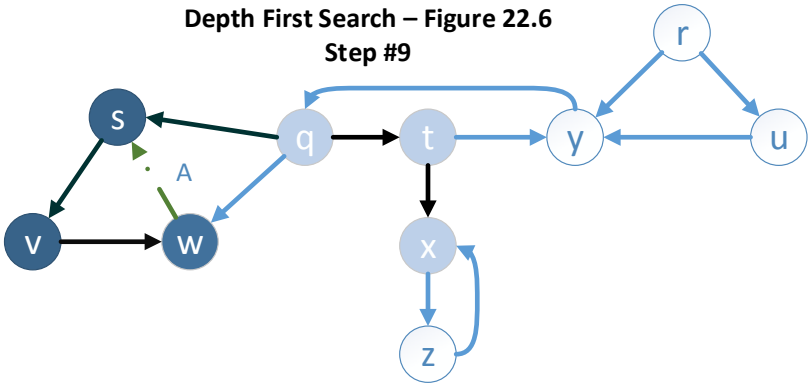
Depth First Search – Figure 22.6
Step #7



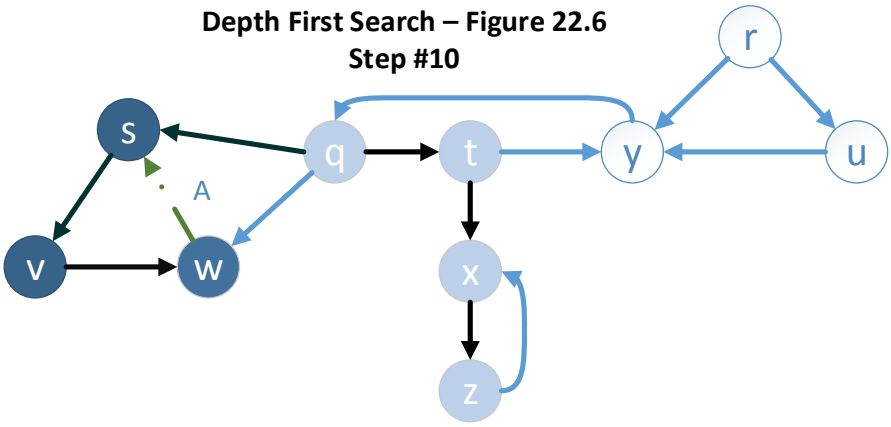
Depth First Search – Figure 22.6
Step #8



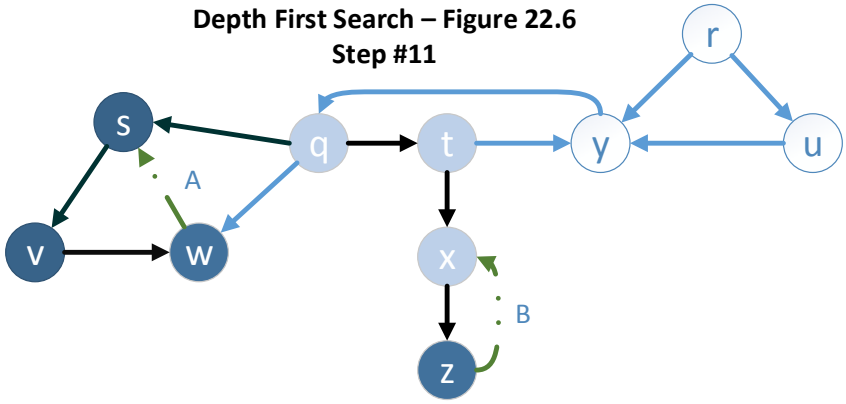
Depth First Search – Figure 22.6
Step #9



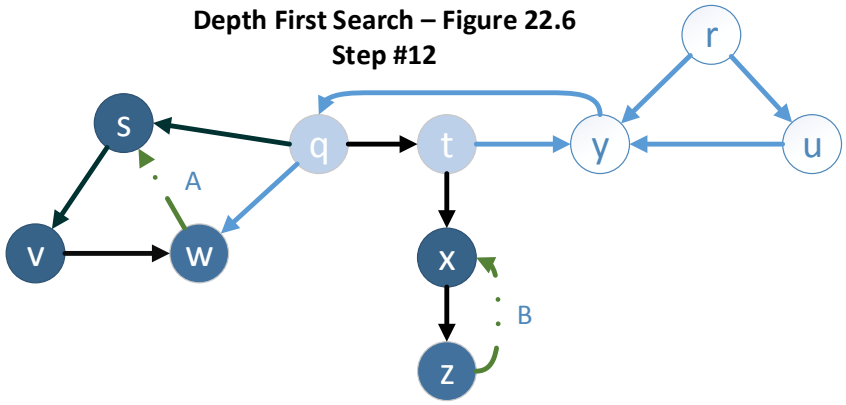
Depth First Search – Figure 22.6
Step #10



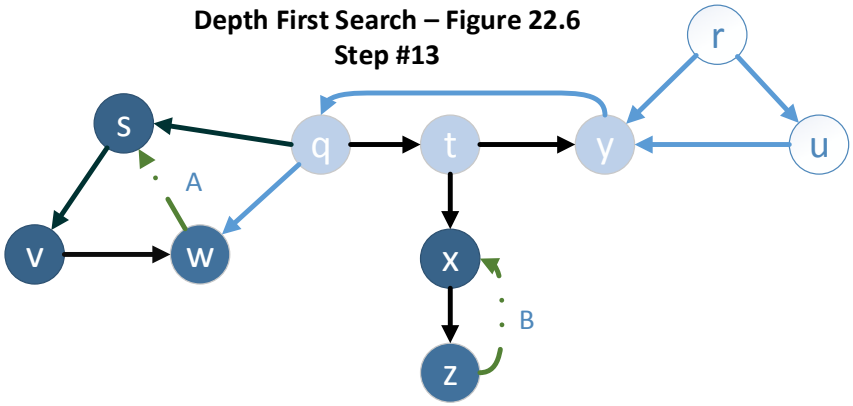
Depth First Search – Figure 22.6
Step #11



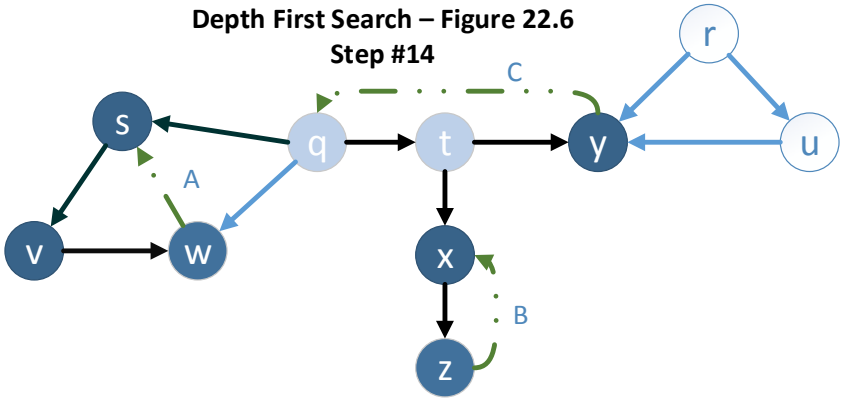
Depth First Search – Figure 22.6
Step #12



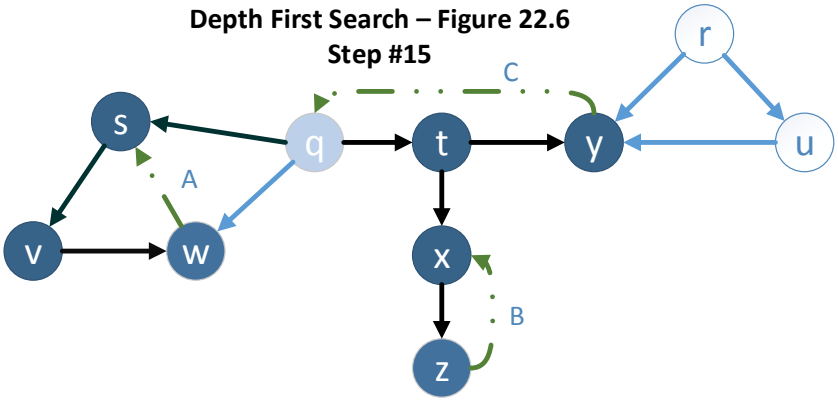
Depth First Search – Figure 22.6
Step #13



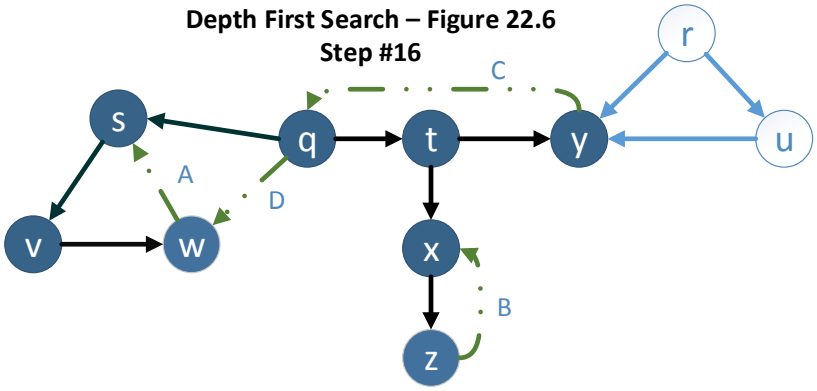
Depth First Search – Figure 22.6
Step #14



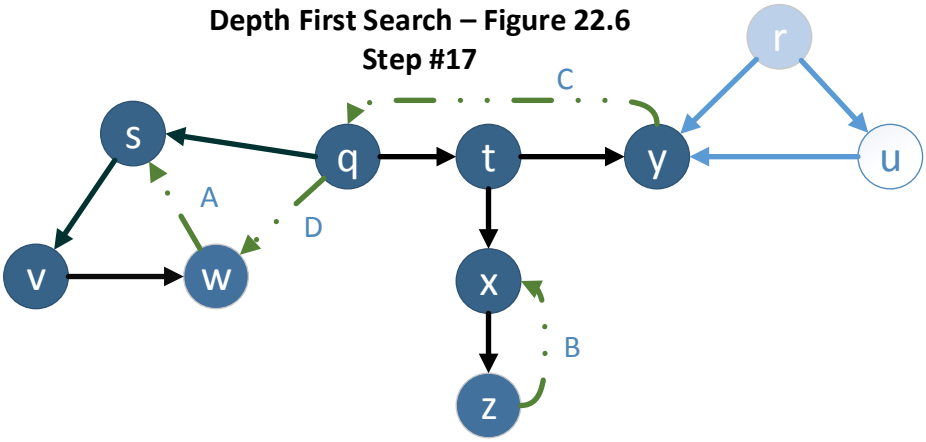
Depth First Search – Figure 22.6
Step #15

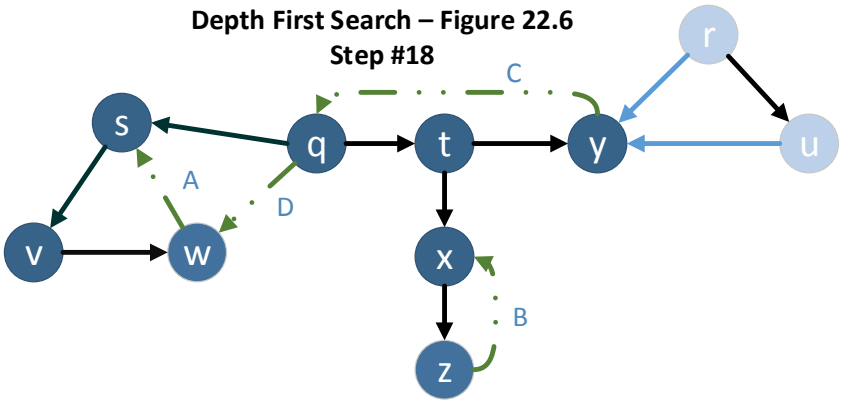


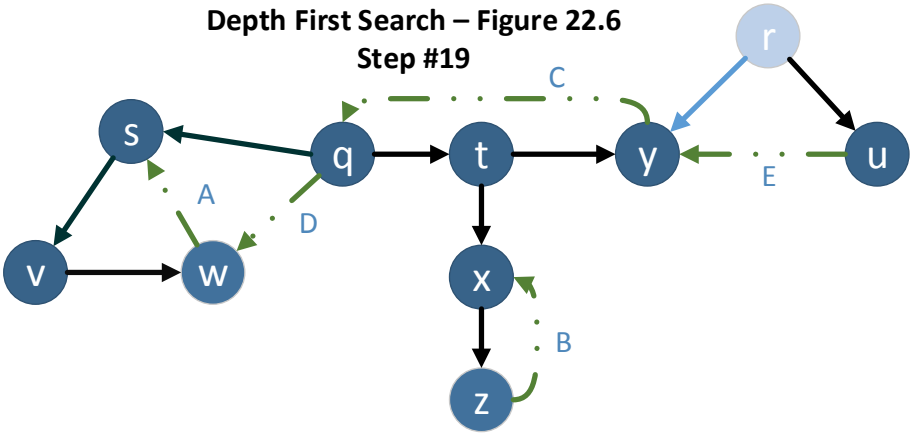
Depth First Search – Figure 22.6
Step #16

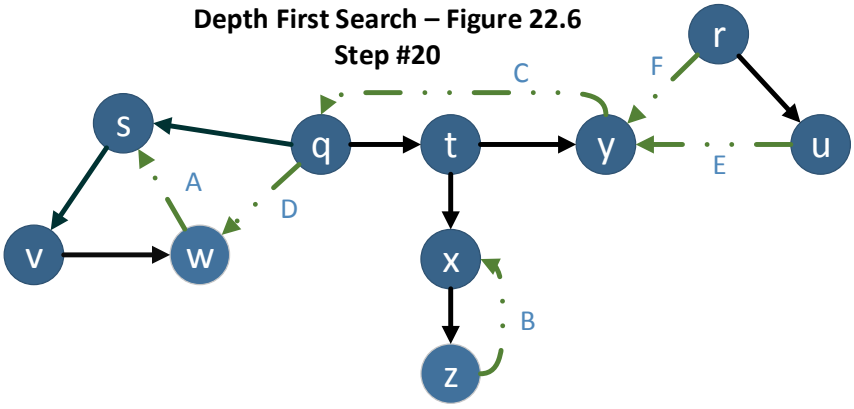


Depth First Search – Figure 22.6
Step #17







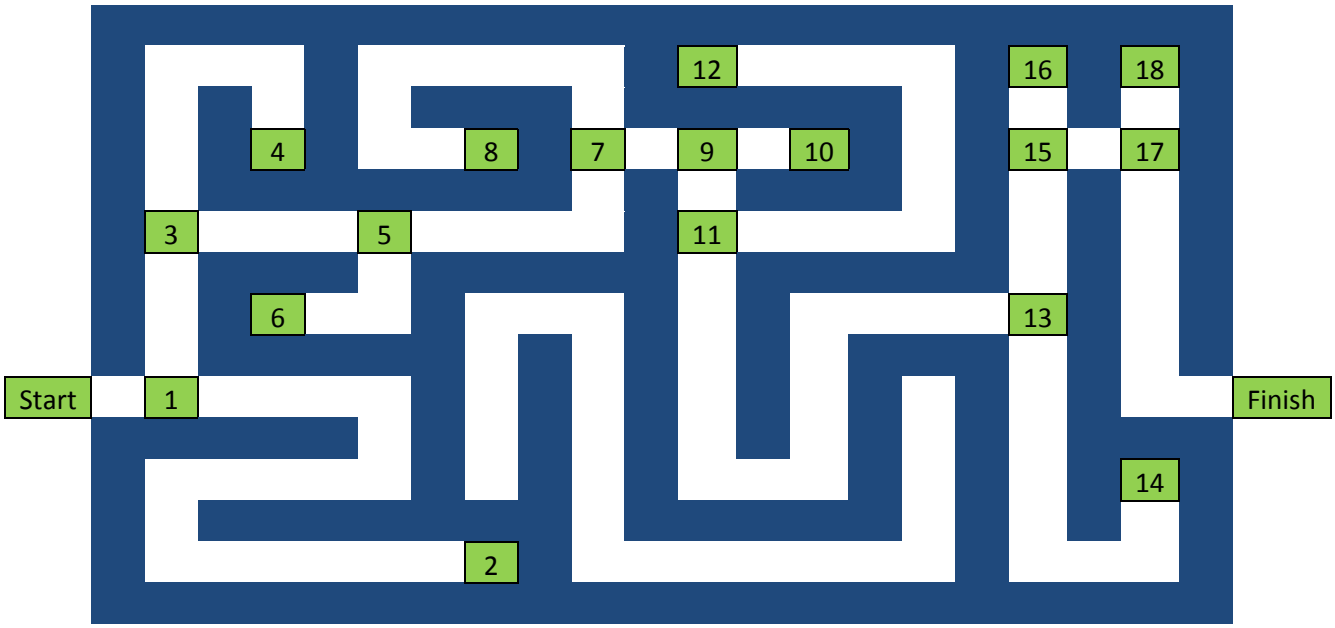


Edge Classification:

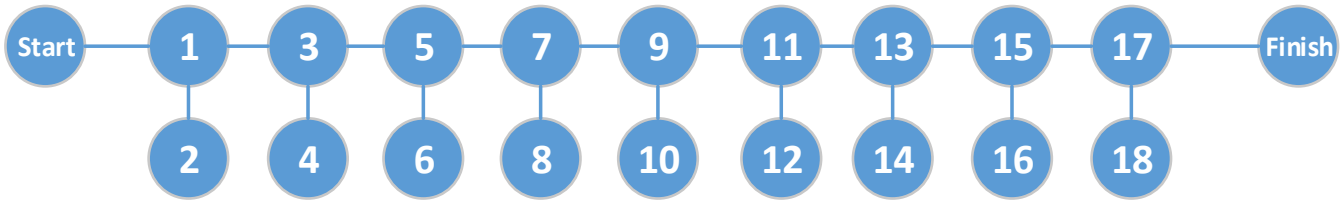
Tree Edges	(q, s) (s, v) (v, w) (q, t) (t, x) (x, z) (t, y) (r, u)
Back Edges	A: (w, s) B: (z, x) C: (y, q)
Forward Edges	D: (q, w)
Cross Edges	E: (u, y) F: (r, y)

Problem #8

Part A: Below is a picture of the maze redrawn. Each of the dead ends and forks in the maze are labeled as green blocks.



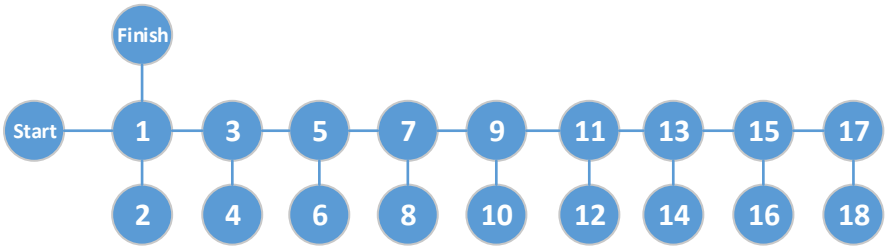
Below is the maze as an undirected graph.



Part B: Depth first search is generally more appropriate for searching a maze (it is the far better algorithm for the example in the homework). Generally (although not necessarily), the connection from the *Start* vertex to the *Finish* vertex is one of the longest (if not the absolute longest) paths in the graph. By using a depth first search, the algorithm is preferring to search for the longest path as opposed doing a wider breadth search. If a breadth-first algorithm was to be used in the above maze, every dead end (e.g. vertices 2, 4, 6, 8, 10, 12, 14, 16, with the exception of possibly 18) would be visited by the algorithm. In contrast, for a depth first search on the above graph, when the algorithm reaches a node with two possible paths out of it, there is only a 50% chance it chooses the dead end path. If it does not choose a dead end path in the above graph, it never needs to waste time traversing a set of vertices that do not lead to a finish state.

Moreover, the uniform expansion of a graph's frontier is relatively straightforward in a computer and does not require significant re-traversing of already visited edges. In contrast, for a human in a maze to uniformly expand the frontier would be extremely time consuming because s/he would need to work back and forth through the maze reaching each of the frontier vertices in order to expand the frontier. As the frontier grew larger and larger, the burden to walk between the frontier vertices would become extremely onerous as the effort expended traversing between frontier vertices would become much larger than the relative frontier expansion. In such a case, the breadth-first search algorithm would become extremely impractical.

However, depth-first search is not preferable in all cases. Below is a modified version of the sample graph provided as part of the problem. In this case, the shortest path from start to finish is two steps (Start → 1 → Finish). In this case, breadth-first search has a substantially better worst case execution than depth-first search.



In the end, both breadth and depth first search will eventually find a path through the maze. If you know some information about the graph in advance (e.g. that it has the form of the modified graph I proposed), then breadth first search may be superior. However, such cases are rarer making depth first search the generally preferred option.