Problem Solving using Search

The breadth-first algorithm spreads out in a uniform manner from the start node. From the start, it looks at each node one edge away. Then it moves out from those nodes to all nodes two edges away from the start. This continues until either the goal node is found or the entire tree is searched.

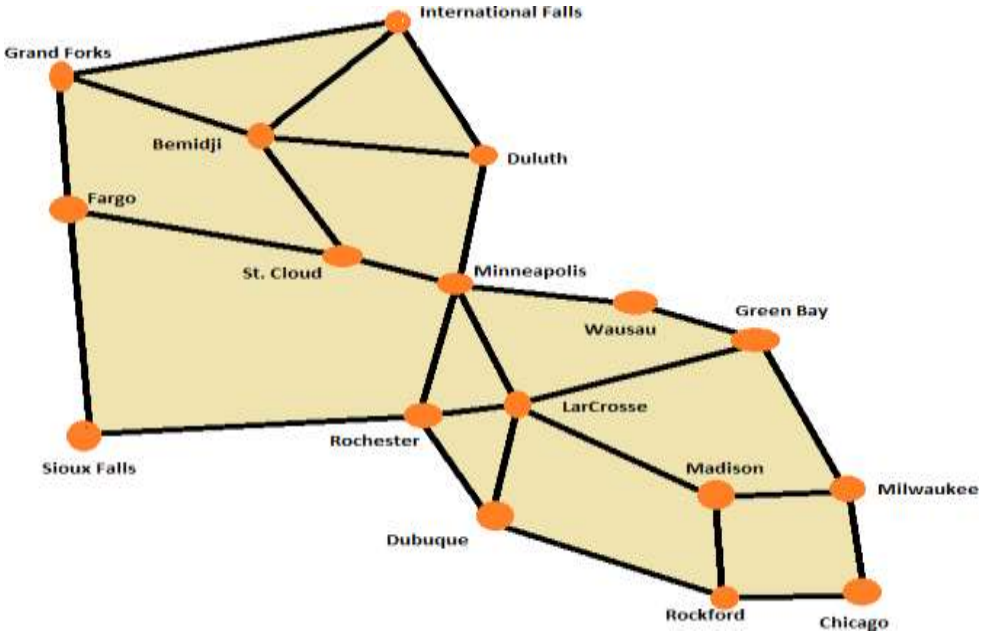
Characteristics of breadth-first algorithm

- Breadth-first search is ***complete***; It will find a solution if one exists.
- But it is neither ***optimal*** in the general case (it won't find the best solution, just the first one that matches the goal state),
- It doesn't have ***good time or space complexity*** (it grows exponentially in time and memory consumption).

Example

A map like the one in Figure below can be naturally represented by *a graph data structure*, where the cities names are the nodes, and the major roadways between cities are the *links or edges* of the graph. So, from a programming perspective, our problem is to traverse a graph data structure in a systematic way until we either find the goal city or exhaust all possibilities. Hopefully having the entire state-space shown on a map will make understanding the operations of the search algorithms easier. In more complex problems, *all we have is the single start state and a set of operators which are used to generate more and more new states*. The search algorithms work the same way, but conceptually, *we are growing or expanding the graph, instead of having it specified at the start.* Map of midwestern U.S. cities is illustrated below:-

The breadth-first algorithm spreads out in a uniform manner from the start node. From the start, it looks at each node one edge away. Then it moves out from those nodes to all nodes two edges away from the start. This continues until either the goal node is found or the entire tree is searched.



Let's walk through an example to see how breadth-first search could find a city on our map.

1- Our search begins in **Rochester (Start State)**, and we want to know if we can get to **Wausau (Goal State)** from there.

2- The **Rochester** node is placed on the queue in step 1 in previous algorithm. Next we enter our search loop at step 2. Queue=[**Rochester]**

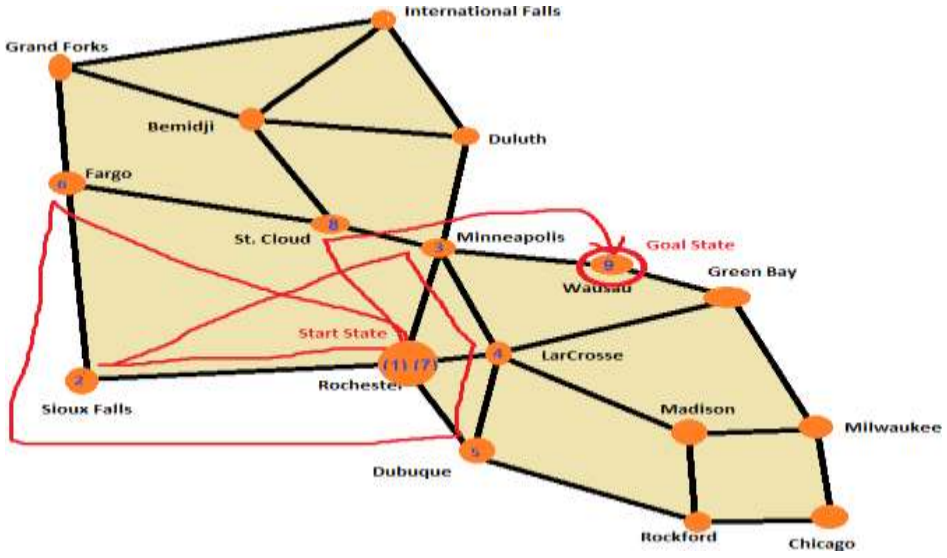
3- We remove Rochester, the first node from the queue. **Rochester** does not contain our goal state (**Wausau**) so we expand it by taking each child node in **Rochester**, *and adding them to the back of the queue.* Queue= **[Sioux Falls, Minneapolis, LaCrosse, and Dubuque].**

4- We remove the first node from the queue (**Sioux Falls**) and test it to see if it is our goal state. It is not, so we expand it, adding ***Fargo*** and ***Rochester*** to the end of our queue, which now contains **[Minneapolis, LaCrosse, Dubuque, Fargo, and Rochester]**.

5- We remove **Minneapolis**, the goal test fails, and we expand that node, adding **St.Cloud, Wausau, Duluth, LaCrosse**, and ***Rochester*** to the search queue, now holding **[LaCrosse, Dubuque, Fargo, Rochester, St.Cloud, Wausau, Duluth, LaCrosse, and Rochester]**.

6- We test **LaCrosse** and then expand it, adding **Minneapolis, GreenBay, Madison, Dubuque**, and **Rochester** to the list, which has now grown to **[Dubuque, Fargo, Rochester, St.Cloud, Wausau, Duluth, LaCrosse, Rochester, Minneapolis, GreenBay, Madison, Dubuque, and Rochester]**. We remove **Dubuque** and add **Rochester, LaCrosse**, and **Rockford** to the search queue.

7- At this point, we have tested every node which is one level in the tree away from the start node (**Rochester**). Our search queue contains the following nodes: **[Fargo, Rochester, St.Cloud, Wausau, Duluth, LaCrosse, Rochester, Minneapolis, GreenBay, Madison, Dubuque, Rochester, Rochester, LaCrosse, and Rockford]**.
8- We remove **Fargo**, which is two levels away from **Rochester**, and add **Grand Forks, St. Cloud**, and **Sioux Falls**.
9- Then we test and expand **Rochester** (Rochester to Minneapolis to Rochester is two levels away from our start). Next is **St. Cloud**; again we expand that node.
10- Finally, we get to **Wausau**; our goal test succeeds and we declare success.
11- Our search order was Rochester, Sioux Falls, Minneapolis, LaCrosse, Dubuque, Fargo, Rochester, St. Cloud, and Wausau as shown below:-



Note that this trace could have been greatly simplified by ***keeping track of nodes which had been tested and expanded***. This would have cut down on our time and space complexity.

Characteristics of Depth First Search

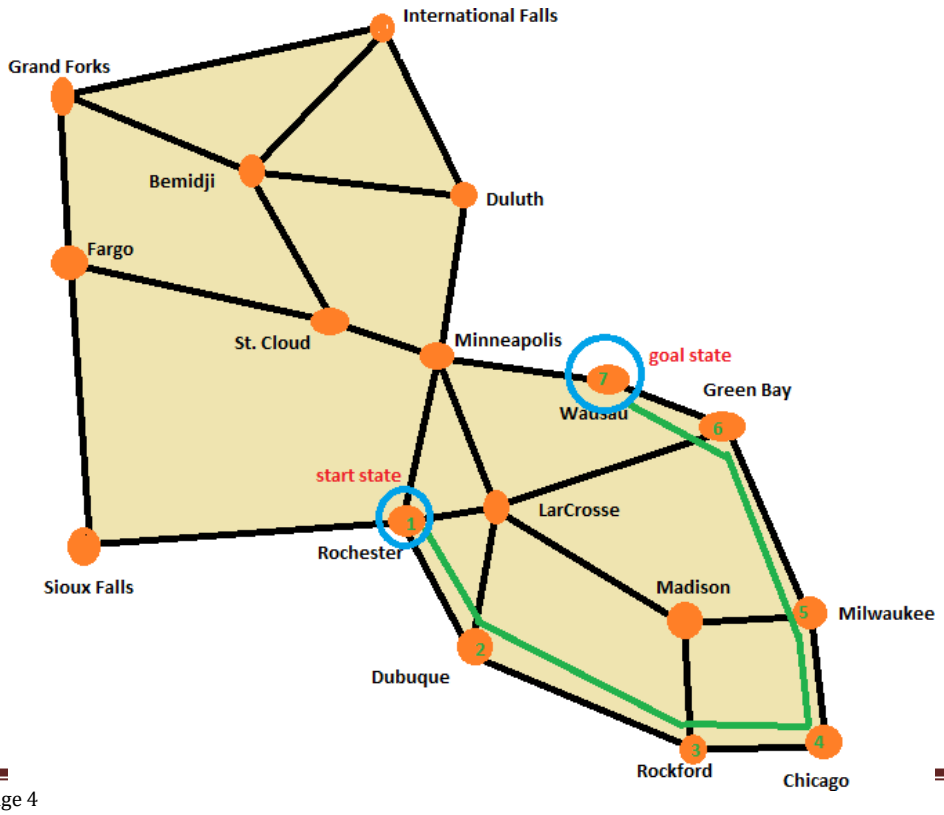
- The depth-first algorithm searches from the start or root node all the way down to a leaf node. If it does not find the goal node, ***it backtracks up the tree and searches down the next untested path until it reaches the next leaf***.
- If you imagine a large tree, the depth-first algorithm may spend a large amount of time searching the paths on the lower left when the answer ***is really in the lower right***.
- Depth-first search is a brute-force method, it will blindly follow this search pattern until it comes across a node containing the goal state, or it searches the entire tree.
- Depth-first search has *lower* memory requirements than breadth-first search,
- It is neither complete nor optimal.

Example

Let's walk through a simple example of how depth-first search would work if we started in **Rochester** and wanted to see if we could get to **Wausau**.

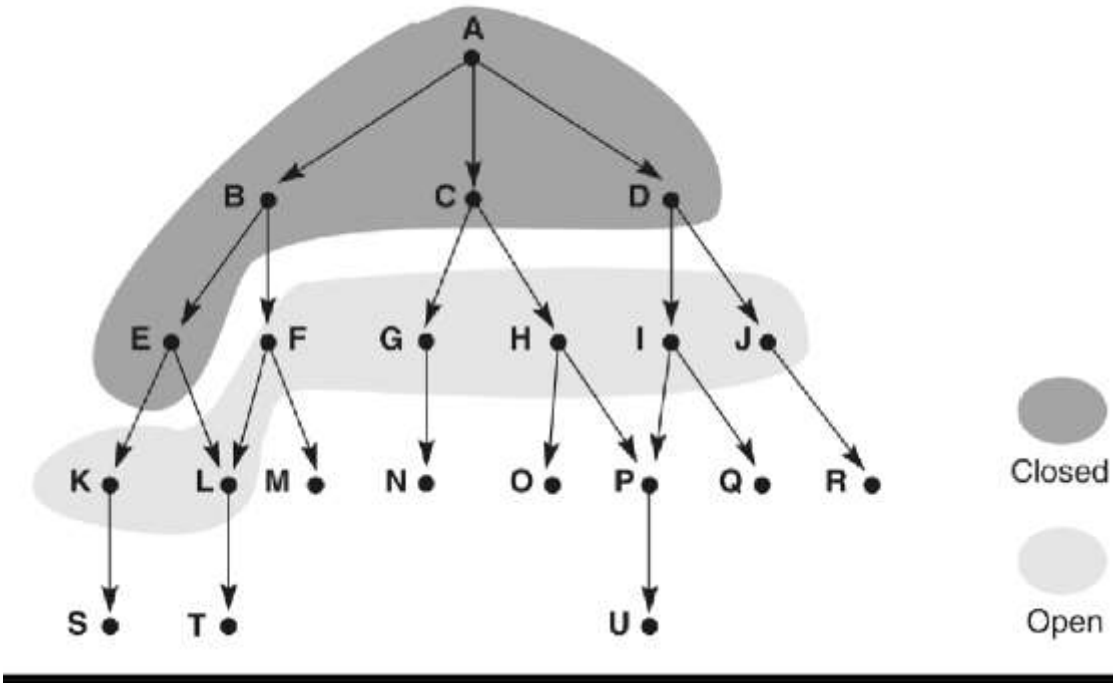
Initial State= (Rochester) and Final-State (Wausau)

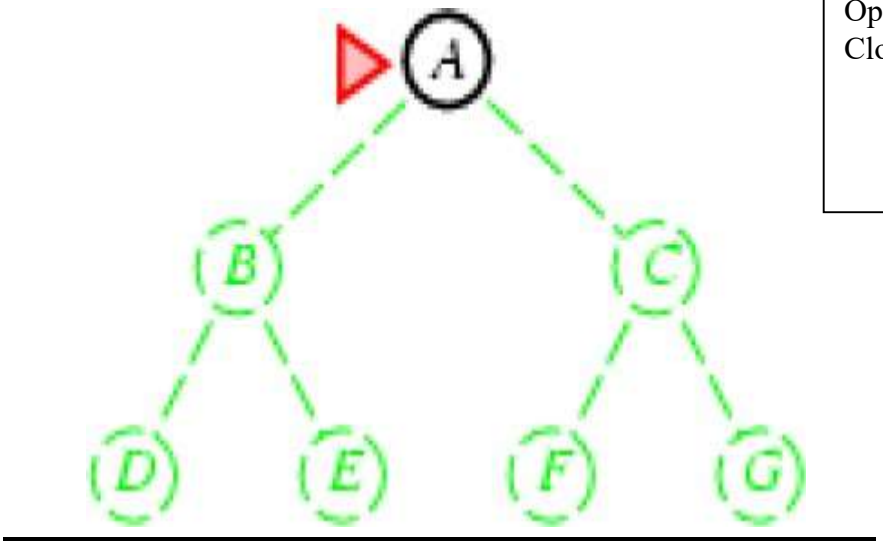
1. Starting with **Rochester**, we test and expand it, placing **Sioux Falls**, then **Minneapolis**, then **LaCrosse**, then **Dubuque** at the front of the search queue ***Queue*=[Dubuque, LaCrosse, Minneapolis, Sioux Falls]**.
2. We remove **Dubuque** and test it; it fails, so we expand it adding **Rochester** to the front, then **LaCrosse**, then **Rockford**. Our search queue now looks like **[Rockford, LaCrosse, Rochester, LaCrosse, Minneapolis, Sioux Falls]**.
3. We remove **Rockford**, and add **Dubuque, Madison**, and **Chicago** to the front of the queue in that order, yielding ***Queue*=[Chicago, Madison, Dubuque, LaCrosse, Rochester, LaCrosse, Minneapolis, Sioux Falls]**.
4. We test **Chicago**, and place **Rockford**, and **Milwaukee** on the queue.
5. We take **Milwaukee** from the front and add **Chicago, Madison**, and **Green Bay** to the search queue. It is now ***Queue*=[Green Bay, Madison, Chicago, Rockford, Chicago, Madison, Dubuque, LaCrosse, Rochester, LaCrosse, Minneapolis, Sioux Falls]**.
6. We remove **Green Bay** and add **Milwaukee, LaCrosse**, and **Wausau** to the queue in that order. Finally, **Wausau** is at the front of the queue and our goal test succeeds and our search ends. Our search order was Rochester, Dubuque, Rockford, Chicago, Milwaukee, Green Bay, and Wausau.



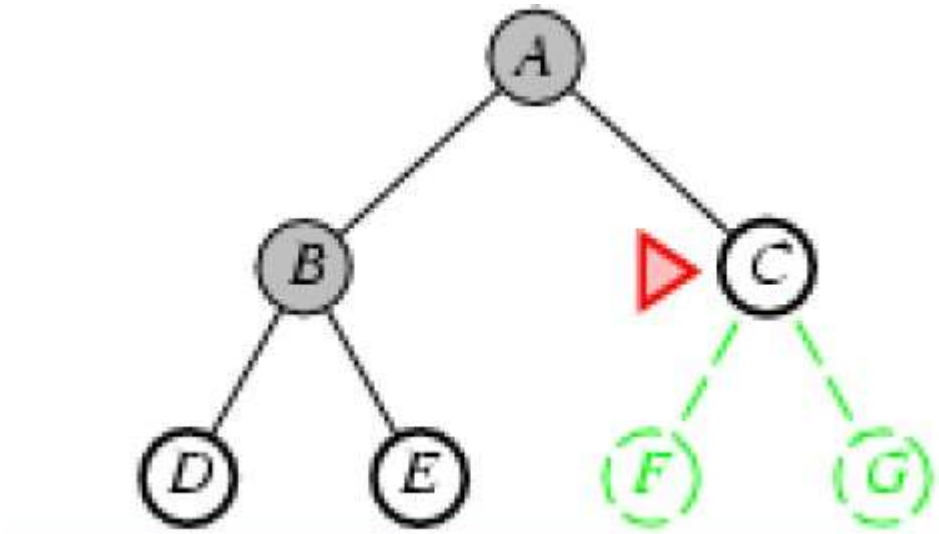
In this example, *we again did not prevent tested nodes from being added to the search queue*. As a result, we had duplicate nodes on the queue. In the depth-first case, this could have been disastrous. *We could have easily had a cycle or loop where we tested one city, then a second, then the first again, ad infinitum*.

Example of Breadth First Search

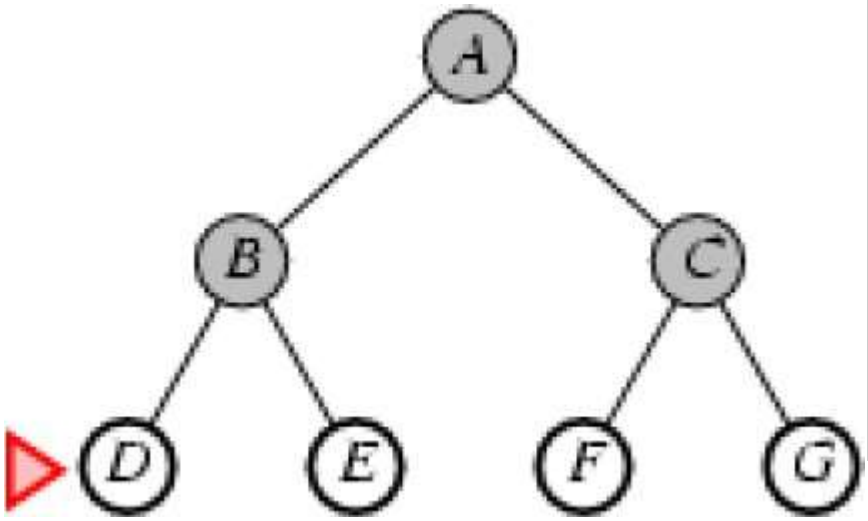




Open: [A]
Closed[ ]



Open: [B,C]
Closed:[A]



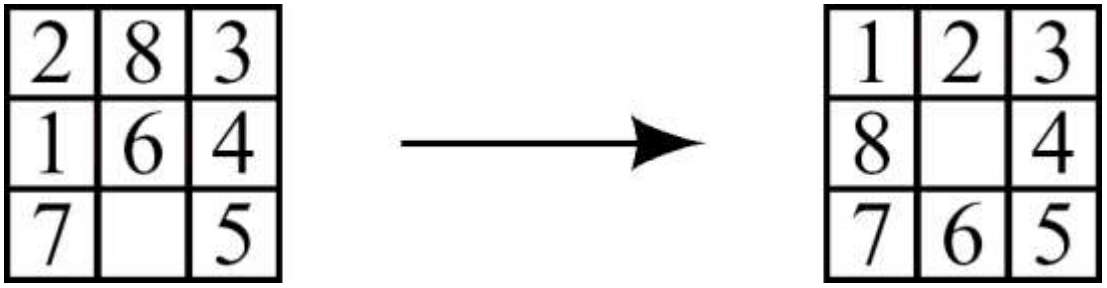
Open:[C,D,E]
Closed:[A,B]

We continue in the same procedure until we reach the Goal [G]

Time Complexity
Space Complexity

Example 2: BFS (Breadth First Search)

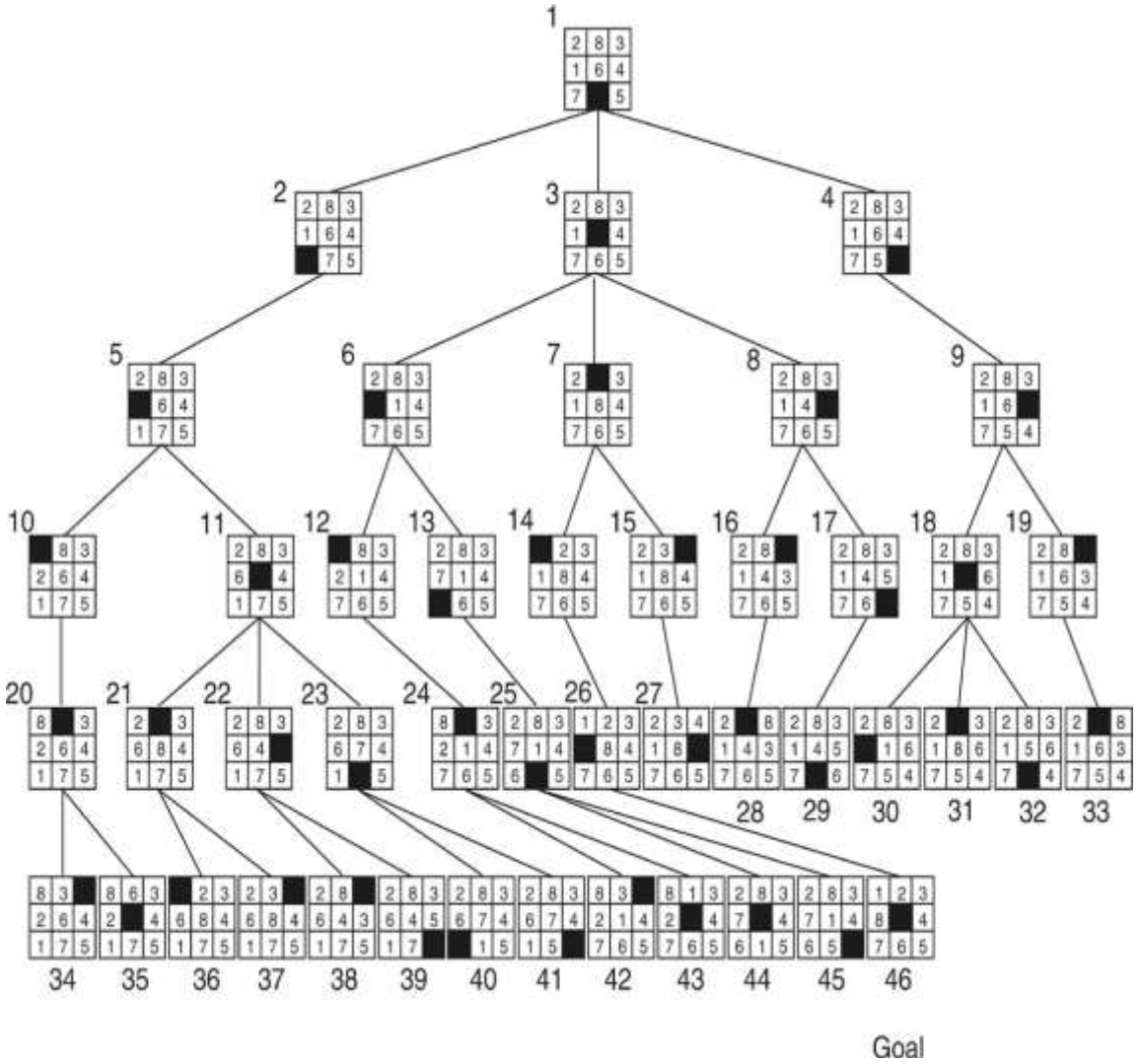
Taken from http://iis.kaist.ac.kr/es/



How can we solve this problem?

1. **We need to initiate the start (initial State) .**
2. **We need to use the control strategy.**
3. **Apply the initial state and control strategy to reach the Goal.**

The solving of this problem is to consider the initial state as above:


1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
Goal

Example3: Apply this example using Depth First Search with goal [N]?