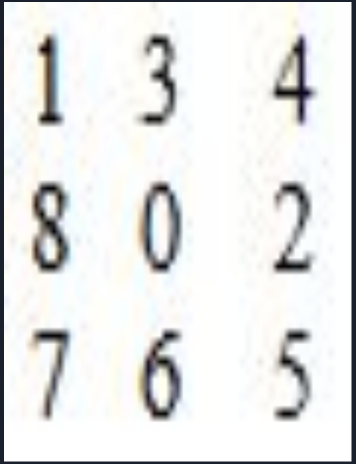
8-Puzzle Report

By: Jaylen Brown, Eric Kemmer, Randall Hunt

8-Puzzle

Initial State:
random order of tiles

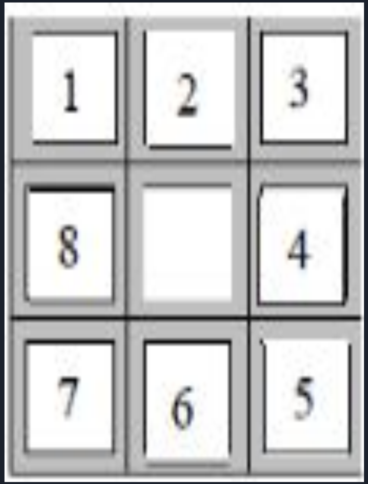


1	3	4
8	0	2
7	6	5

Goal:
Move the tiles adjacent to the blank space until the goal state is reached.

Note:
We only consider versions of the puzzle that are solvable

Goal State:
Made of seven tiles and one blank

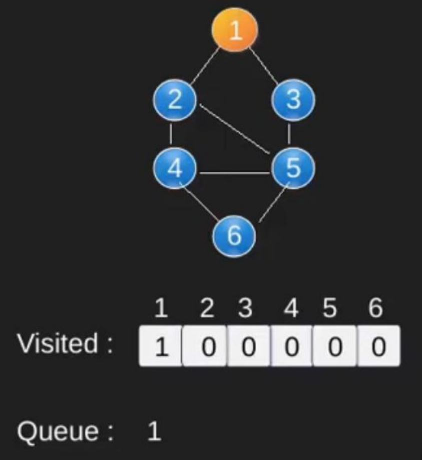


1	2	3
8		4
7	6	5

BFS - Breadth First Search

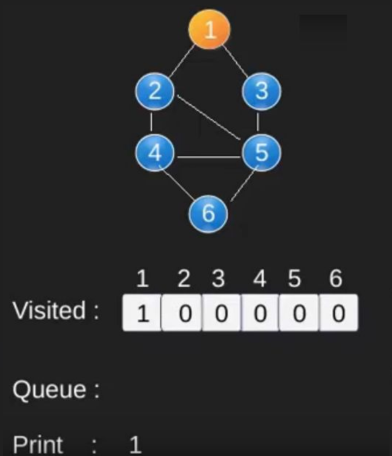
- Breadth first search uses a **visited array** and a **queue** to store it's traversal
- Each node *enqueues* itself and marks itself 1 in the visited queue
- Each node it visits is marked in visited and *enqued* as well
- Once it has visited all the nodes adjacent to it, it will *dequeue* itself and *print*

Time Complexity = O(V + E)



	1	2	3	4	5	6
Visited :	1	0	0	0	0	0

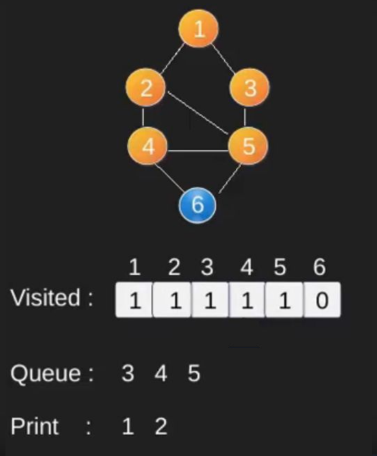
Queue : 1



	1	2	3	4	5	6
Visited :	1	0	0	0	0	0

Queue :

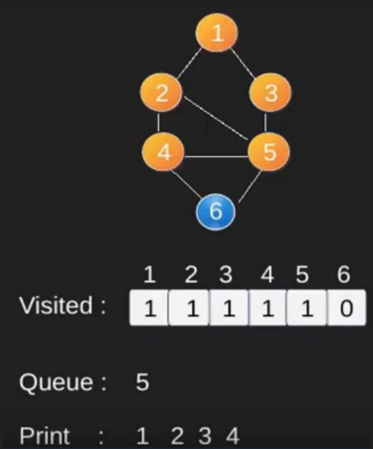
Print : 1



	1	2	3	4	5	6
Visited :	1	1	1	1	1	0

Queue : 3 4 5

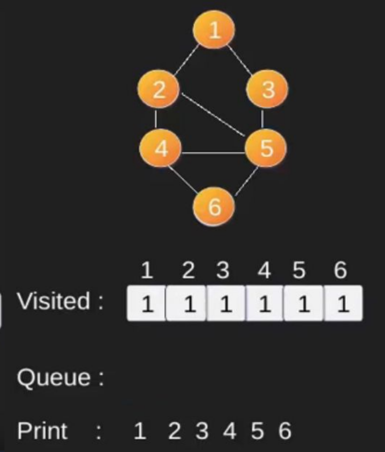
Print : 1 2



	1	2	3	4	5	6
Visited :	1	1	1	1	1	0

Queue : 5

Print : 1 2 3 4



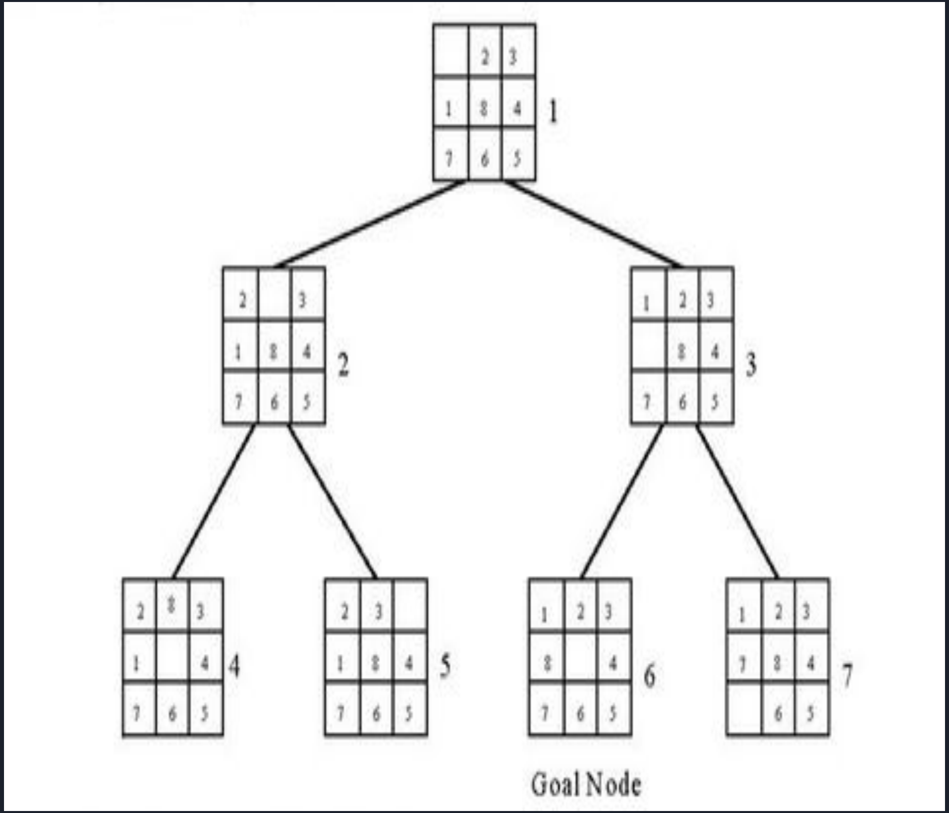
	1	2	3	4	5	6
Visited :	1	1	1	1	1	1

Queue :

Print : 1 2 3 4 5 6

Breadth First Search 8-puzzle Implementation

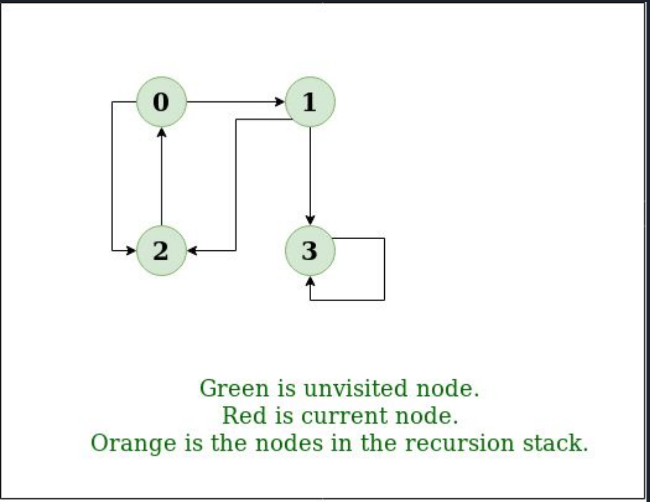
- Functionality
  - **Add** the first node to the open list
  - **Create** all it's possible children by calling moves up, down, left, right
  - Each child created is **added to the open list** *if it is not the goal state and is not already in the list*
  - Each child that is being explored is **added to the closed list**
  - **Continue** until a goal state is reached



DFS - Depth First Search

The algorithm starts at the root node and explores each branch as far as it can before backtracking and exploring any other branches that would have been missed. The vertices are marked as they are visited to indicated if there are any unexplored branches.

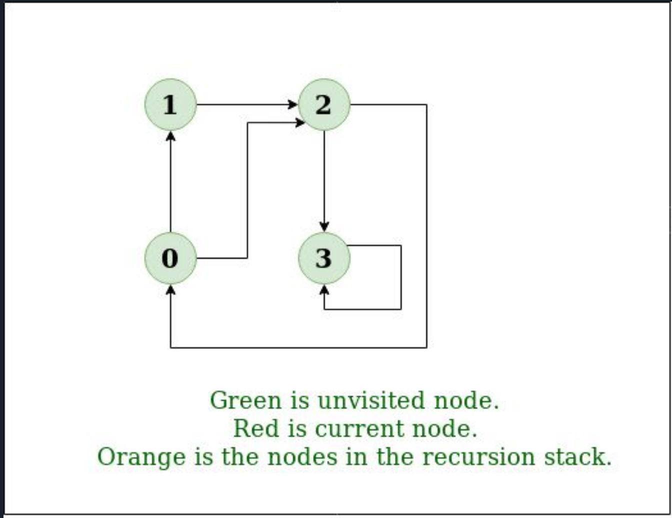
Time Complexity = O(V + E)



Steps:

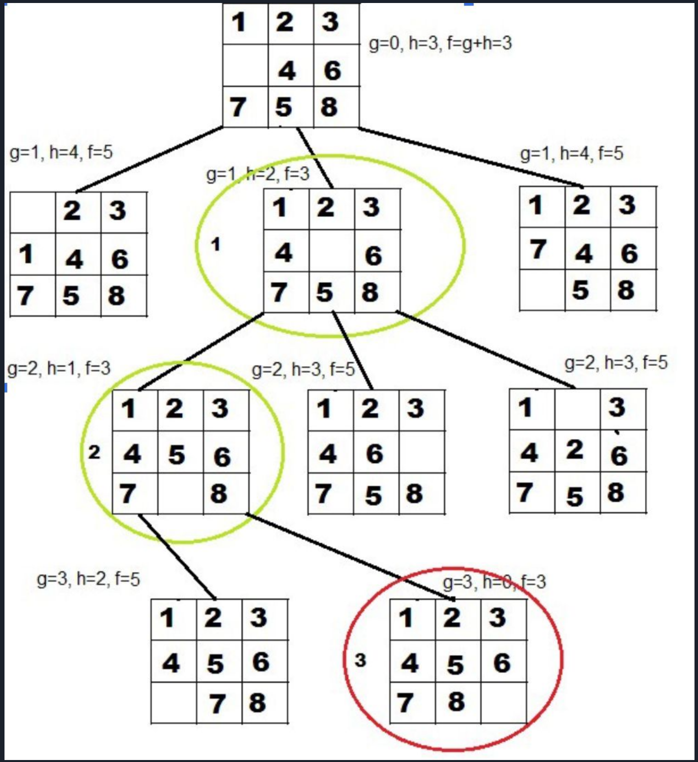
- Create a recursive function that is passed the root node.
- The current node is marked.
- Traverse all the unmarked nodes recursively.

Cite: https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/



DFS - 8 Puzzle Implementation

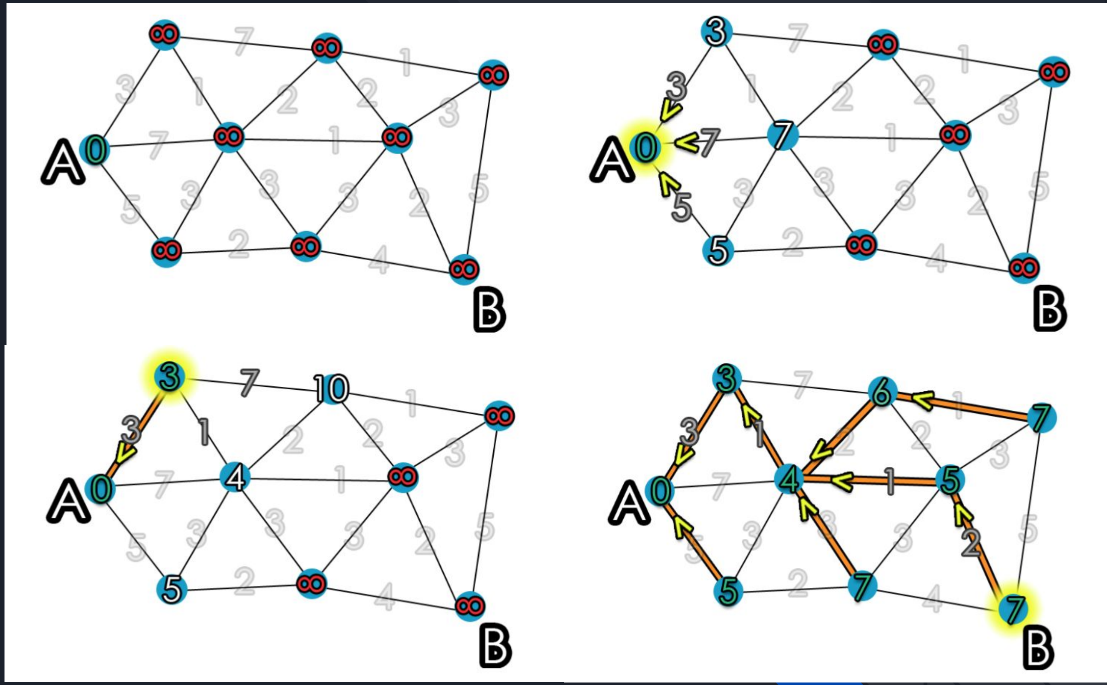
- DFS recursively calls itself taking in a node as the parameter
- We check if the node == goal state
  - If it has we print the path from the goal state → initial state and return false
- We check if the node already exists in the visited list
  - If it does we return false
- Otherwise we add the node to the visited list and call each move possible on it up, down , left, right
- We then iterate through the newly created children
  - For each child we add one to it’s cost and then send it through the DFS function again
  - *Note: Each move has a cost of 1 for DFS*



Dijkstra's

Time Complexity = O(V + ElogV)

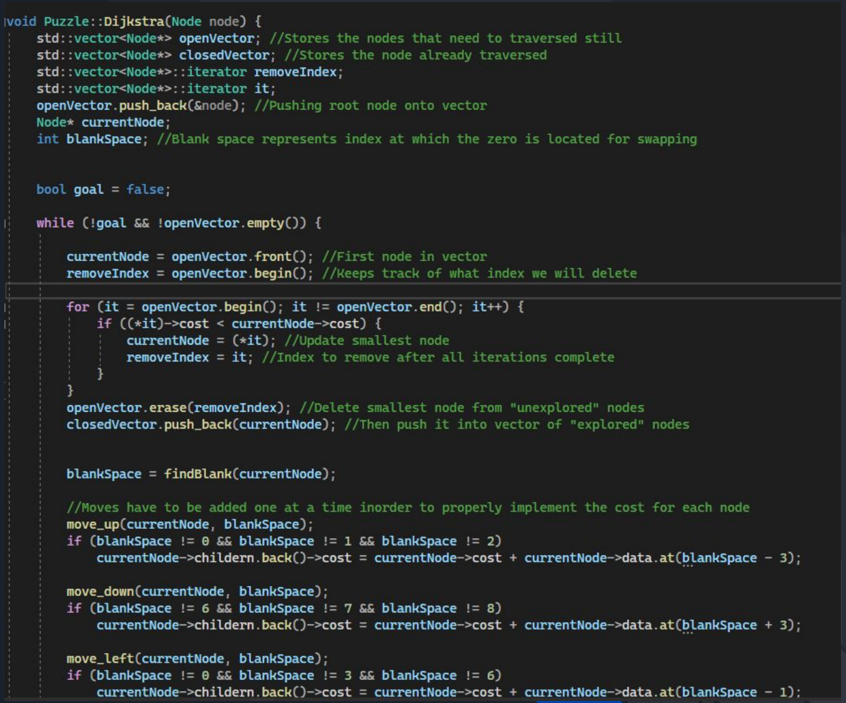
- Mark the root vertex as 0 and the rest as infinity.
- Find all the vertices that lead to the targeted vertex and add the distances together to identify the shortest path.
- We traverse through the paths to identify the shortest path, if a distance is larger than a previously recorded distance then we will ignore it.
- This will continue until we reach the targeted vertex via the shortest path.



Cite: https://brilliant.org/wiki/dijkstras-short-path-finder/

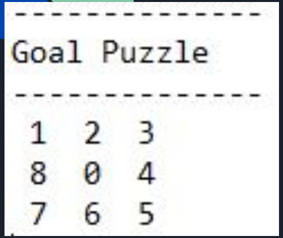
Dijkstra’s 8-puzzle implementation

- Using vectors we represent an open and closed list
- The initial state is pushed onto the open vector
- A while loop iterates under the condition that we have not met the goal state and the open vector is not empty
    - A for loop will sort the open vector list and remove the smallest element from the open vector and move it to the closed vector
    - We call functions for moving up, down, left, right and update their cost if the move is possible
    - A final for loop will iterate through the children just created for the node to find the goal state or add to the open vector



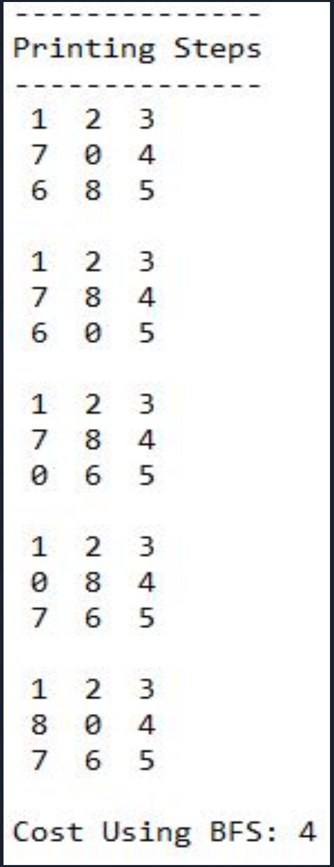
```
void Puzzle::Dijkstra(Node node) {
    std::vector<Node*> openVector; //Stores the nodes that need to traversed still
    std::vector<Node*> closedVector; //Stores the node already traversed
    std::vector<Node*>::iterator removeIndex;
    std::vector<Node*>::iterator it;
    openVector.push_back(&node); //Pushing root node onto vector
    Node* currentNode;
    int blankSpace; //Blank space represents index at which the zero is located for swapping


    bool goal = false;

    while (!goal && !openVector.empty()) {

        currentNode = openVector.front(); //First node in vector
        removeIndex = openVector.begin(); //Keeps track of what index we will delete

        for (it = openVector.begin(); it != openVector.end(); it++) {
            if ((*it)->cost < currentNode->cost) {
                currentNode = (*it); //Update smallest node
                removeIndex = it; //Index to remove after all iterations complete
            }
        }
        openVector.erase(removeIndex); //Delete smallest node from "unexplored" nodes
        closedVector.push_back(currentNode); //Then push it into vector of "explored" nodes


        blankSpace = findBlank(currentNode);

        //Moves have to be added one at a time inorder to properly implement the cost for each node
        move_up(currentNode, blankSpace);
        if (blankSpace != 0 && blankSpace != 1 && blankSpace != 2)
            currentNode->childern.back()->cost = currentNode->cost + currentNode->data.at(blankSpace - 3);

        move_down(currentNode, blankSpace);
        if (blankSpace != 6 && blankSpace != 7 && blankSpace != 8)
            currentNode->childern.back()->cost = currentNode->cost + currentNode->data.at(blankSpace + 3);

        move_left(currentNode, blankSpace);
        if (blankSpace != 0 && blankSpace != 3 && blankSpace != 6)
            currentNode->childern.back()->cost = currentNode->cost + currentNode->data.at(blankSpace - 1);
```

Resulting Costs: BFS, DFS and Dijkstra's

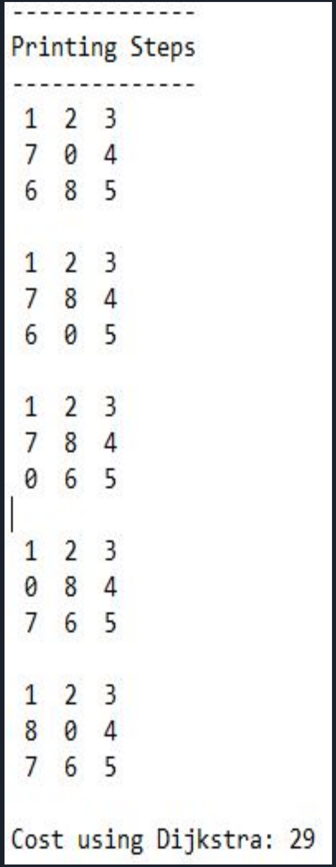


```
--------------
Goal Puzzle
--------------
 1  2  3
 8  0  4
 7  6  5
```

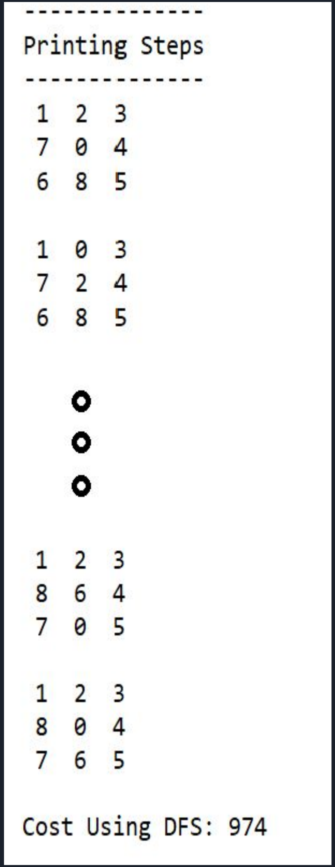
As we can see from the following results; BFS has the lowest cost with 4, Dijkstra has the 2nd lowest being 29, and DFS has a horrendous cost being 974.



```
--------------
Printing Steps
--------------
 1  2  3
 7  0  4
 6  8  5

 1  2  3
 7  8  4
 6  0  5

 1  2  3
 7  8  4
 0  6  5

 1  2  3
 0  8  4
 7  6  5

 1  2  3
 8  0  4
 7  6  5

Cost Using BFS: 4
```



```
-------------
Printing Steps
-------------
 1  2  3
 7  0  4
 6  8  5

 1  2  3
 7  8  4
 6  0  5

 1  2  3
 7  8  4
 0  6  5

 1  2  3
 0  8  4
 7  6  5

 1  2  3
 8  0  4
 7  6  5

Cost using Dijkstra: 29
```

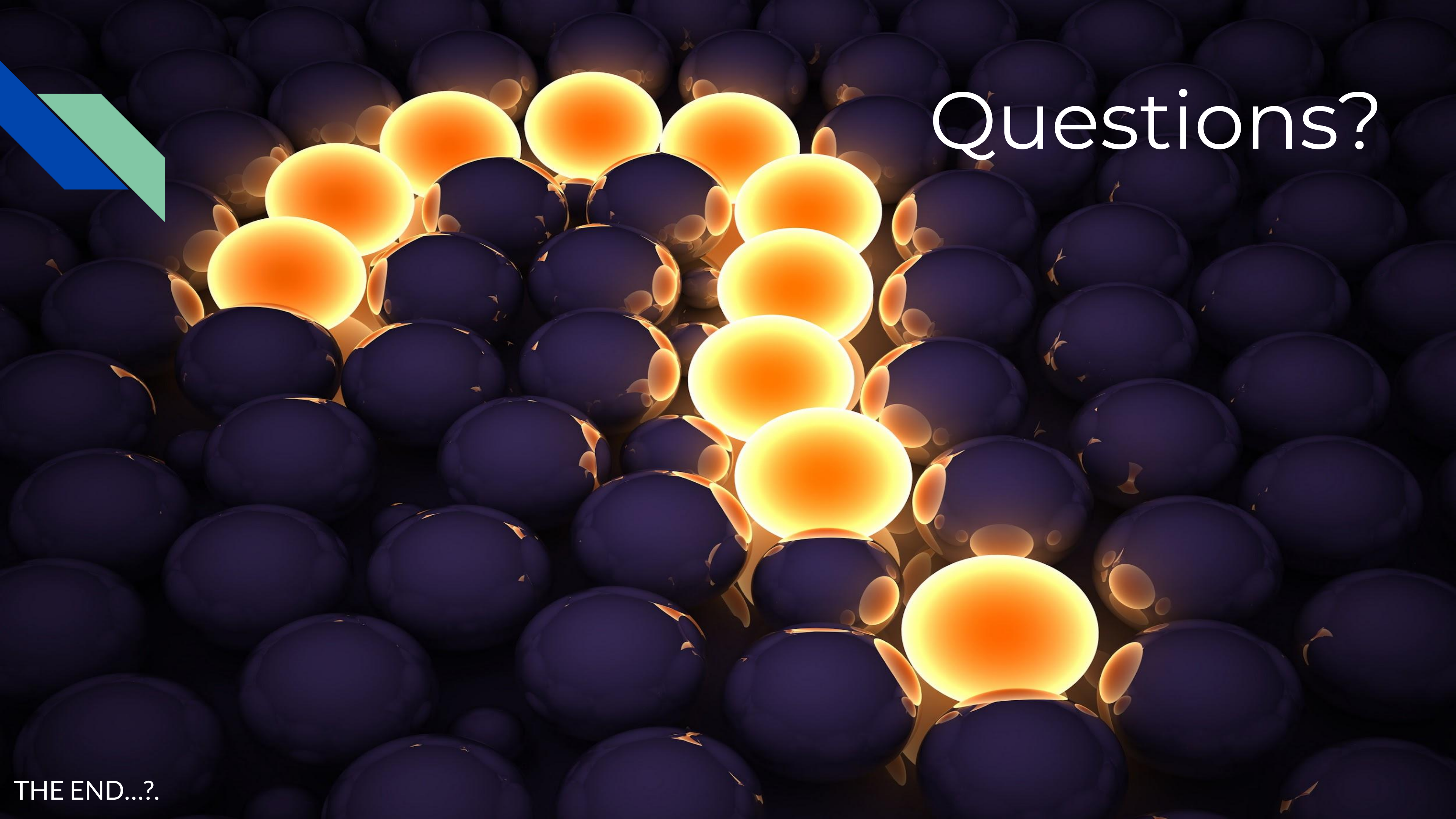


```
-------------
Printing Steps
-------------
 1  2  3
 7  0  4
 6  8  5

 1  0  3
 7  2  4
 6  8  5

    o
    o
    o

 1  2  3
 8  6  4
 7  0  5

 1  2  3
 8  0  4
 7  6  5

Cost Using DFS: 974
```

Conclusion

We can conclude from the costs of each algorithm running that BFS runs more efficiently than the others in terms of the 8 Puzzle.

While DFS runs in a reasonable cost size, but Dijkstra’s algorithm in regards to the 8 Puzzle has a very large cost.



Questions?

THE END…?.