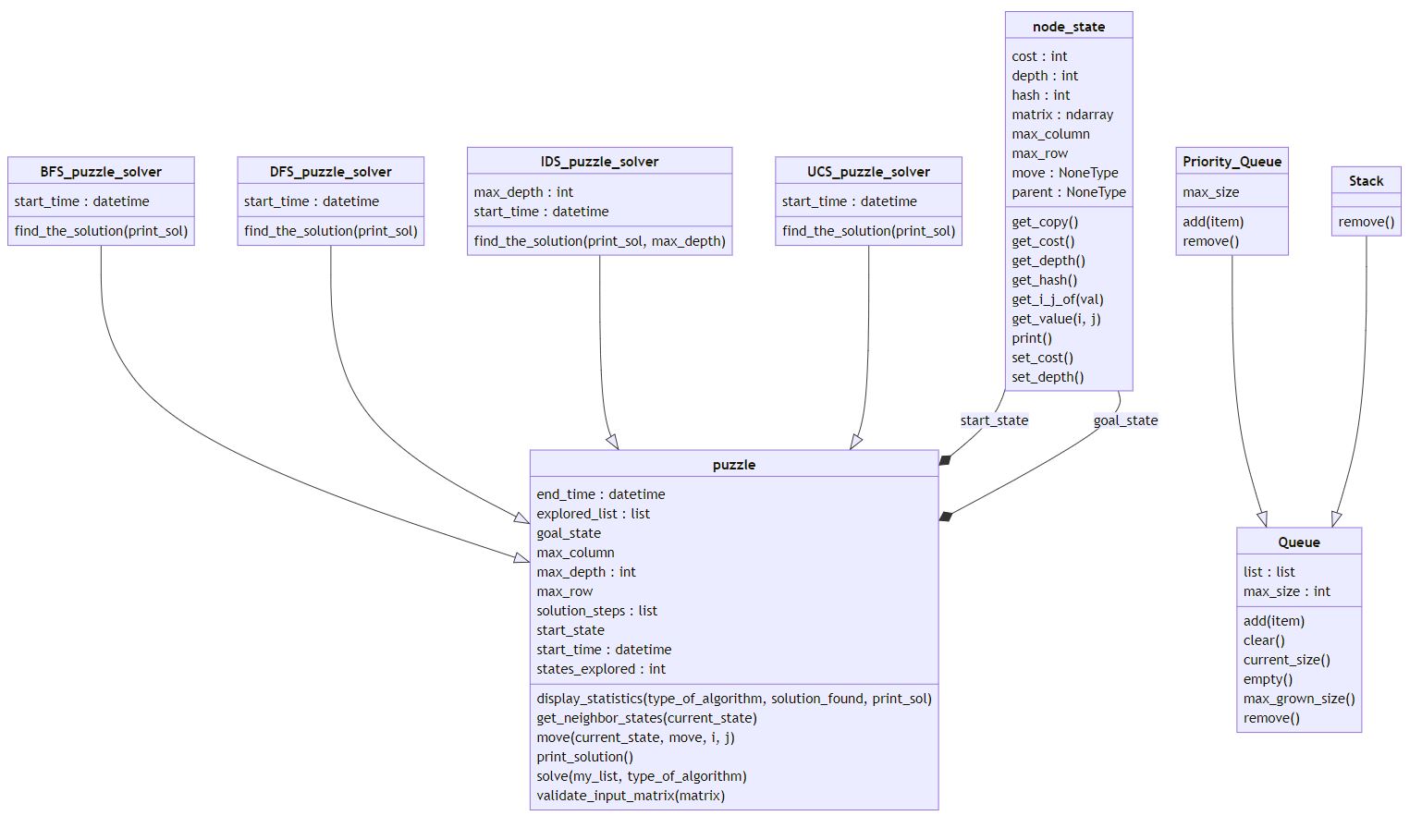
**Explanation of the Code:**

For the code we have created multiple classes as shown below:  
  


Below is the explanation of each class.

**node\_state:**

This class represents any state of the puzzle. A state of the puzzle can have some properties like

1. Matrix: this is the matrix of the values e.g. 123456780
2. Parent: except the start state, all other states will have a parent i.e. the state from which we have moved the blank space (up/down/left/right). This will be helpful in backtracking.
3. Move: The move (up/down/left/right) that was taken on the parent to reach this state.
4. Max\_row: Maximum number of rows of the matrix. (This is 3 for our problem but can also be used to extend the present 3x3 matrix to 4x4 matrix etc.)
5. Max\_column: Maximum number of columns in the matrix.
6. Hash: This is the string representation of the matrix. We make the matrix’s hash in the visited list.
7. Depth: while solving the puzzle we keep on moving to level. This variable is used to keep track of the depth.
8. Cost: This contains the cost of the state w.r.t the root. This is used in multiple algorithms like UCS.

The above properties are maintained to solve algorithm.

**Queue:**

This class is used to implement Queue and is used by algorithms like BFS. This class is also used as base class for the implementation of the Stack & Priority\_Queue classes.

**Stack**

This class is used to implement the DFS. “Queue” class is used as base to this class so that the function can be re-used.

**Priority\_Queue**

This class is used to implement the UCS. “Queue” class is used as base to this class so that the function can be re-used. The add & remove function is implemented differently. In this we use heapq to add the items in the list. We also use the cost of the state while adding into the list so that while removing the items we will remove the item that have the lowest cost.

**Puzzle**

This class is used to implement the puzzle itself. This class will be used as base class for the classes of the algorithms.

We validate the states i.e. start state and goal state before stating. This is done to make sure that there is no duplicate entry in the start/goal state.

The input matrix and the goal matrix are converted to the Start state & Goal state.

All the classes that will be using this class as a base class, can use the solve() function. This function need the instance of the list (Queue/Stack/Priority Queue) based on the need of algorithm. In this function we solve the problem in following way

* Check if the list is empty or not. (The called has the add the root in the list before calling this function).
  + If the list is empty then it means that we are not able to find the solution
* Remove a state from the list and mark it as current state (as the called has given the instance of list so it will be Queue/Stack/Priority\_Queue based on the callers requirement).
* Check if the current states hash is equal to the Goal state hash.
  + If yes then it means we have found the solution.
    - Back-tract to the state state i.e. root and fill each state to the list solution\_steps.
    - Return True
* Find the neighbors states using get\_neighbor\_states() function.
  + The get\_neighbor\_states() function check which moves are possible i.e. up/down/right/left. The states that can be possible are returned to the caller of get\_neighbor\_states().
* Check each neighbor state in the explored\_list.
  + If any neighbor state is in the explored\_list then we do not need to process it.
  + If the neighbor state is not in the explored\_list then we can add it to the list i.e. Queue/Stack/Priority\_Queue.

In this way the caller will be able to find the solution.

There are other function in this class that are helpful e.g. display\_ statistics() is used to print the statistics e.g. time take in solving the puzzle by the algorithm. The number of steps required to solve the puzzle using the algorithm. The number of states explored to solve the puzzle etc.

print\_solution() function is used to print the solution.

**BFS\_puzzle\_solver**

This class uses “puzzle“ class as a base. The BFS uses Queue. So, we take an instance of Queue class in “my\_list”. We pass the instance of Queue to the solve() function. The solve() of “puzzle” call is used as this class has inherited it from the base class. If the solve() function return true then it means we have found the solution.

We call display\_ statistics() to display the statistics of the result.

**DFS\_puzzle\_solver**

This class uses “puzzle“ class as a base. The DFS uses Stack. So, we take an instance of Stack class in “my\_list”. We pass the instance of Stack to the solve() function. The solve() of “puzzle” is used as this class has inherited it from the base class. If the solve() function return true then it means we have found the solution.

We call display\_ statistics() to display the statistics of the result.

**UCS\_puzzle\_solver**

This class uses “puzzle“ class as a base. The UCS uses Priority\_Queue. So, we take an instance of Priority\_Queue class in “my\_list”. We pass the instance of Priority\_Queue to the solve() function. The solve() of “puzzle” is used as this class has inherited it from the base class. If the solve() function return true then it means we have found the solution.

We call display\_ statistics() to display the statistics of the result.

**IDS\_puzzle\_solver**

This class uses “puzzle“ class as a base. The IDS uses Stack. So, we take an instance of Stack class in “my\_list”. We pass the instance of Stack to the solve() function. The solve() of “puzzle” is used as this class has inherited it from the base class. We set the max\_depth to 1 and call solve(). If the solve() return true then it means we found the solution. If not then we increment the depth and call it again. The get\_neighbor\_states() function of “puzzle” class uses the depth. It checks if the depth of the parent state is less then the configured depth then it returns the next possible neighbors. If not then it do not returns any neighbors.

We call display\_ statistics() to display the statistics of the result.

**Comparison of all 4 algorithms**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **BFS** | **DFS** | **UCS** | **IDS** |
|  | BFS explores all nodes at a given depth level before moving to the next depth level. Therefore, if a goal node is reachable, BFS is guaranteed to find the shortest path to that node, assuming uniform edge costs. | DFS is not guaranteed to find the optimal solution, especially when edge costs are not uniform. It may explore very deep branches before exploring shallower ones, potentially leading to a suboptimal path. | UCS explores paths with lower costs first, ensuring that it finds the optimal path to the goal node. | IDS combines the advantages of BFS and DFS. It performs multiple depth limited DFS searches, gradually increasing the depth limit with each iteration |
| **Space Complexity** | O(bs+1)  b: Branching Factor  s: Depth of shallowest solution | O(bm)  b: Branching Factor  m: max depth of tree | O(bC\*/e)  b: Branching Factor  c: cost of least cost sol. | O(bs)  b: Branching Factor  s: Depth of shallowest solution |
| **Time Complexity** | O(bs+1)  b: Branching Factor  s: Depth of shallowest solution | O(bm+1)  b: Branching Factor  m: max depth of tree | O(bC\*/e)  b: Branching Factor  c: cost of least cost sol. | O(bs)  b: Branching Factor  s: Depth of shallowest solution |
| **Advantages** | It guarantees the shortest path | It is memory efficient algorithm | UCS guarantees finding the shortest path and is better then the BFS. | It has properties of both BFS & DFS. It is memory efficient as compared to BFS. |
| **Disadvantages** | It consumes lot of memory as it has to explore all the states at a level before moving to the next one. | It may not reach goal if the depth is infinite. | It is not good solution if we have high number of branching. It can fail if action are arbitrarily cheaper | It could take more time to reach goal specially when the branching factor is high. |
| **Example: For [[3 2 1]**  **[4 5 6]**  **[8 7 0]]** | States explored before reaching goal: 130052  Solution can be reached in steps: 24 | States explored before reaching goal: 158318  Solution can be reached in steps: 27816 | States explored before reaching goal: 132241  Solution can be reached in steps: 24 | States explored before reaching goal: 338144  Solution can be reached in steps: 30 |