Coimbatore Institute of Technology

Department of Artificial Intelligence and Data Science

Assignment 1: N-Tile Puzzle

Course: Principles of Artificial Intelligence

Class: B.Tech AI&DS, 3rd semester

Team members:

Samuela Abigail Mathew , 71762108039

Nivetha S , 71762108030

Haripriya V , 71762108011

Table of Content

1. Introduction

The N-Tile puzzle is a game consisting of an empty tile ‘X’ and ‘N’ number of green tiles ‘G’ and red tiles ‘R’ each. So there’ll be 2N+1 tiles in total. The goal state is to bring all ‘G’ tiles to the left of ‘R’ tiles, and one ‘R’ tile should be in the rightmost position. So ‘X’ can be anywhere except the rightmost position. The program will print the total cost taken to reach goal state and total nodes expanded in the end.

The user input is taken via the main module which is named ‘main_game_Sam_Niv_Hari.py’ and user can select which search algorithm to apply to get goal state. In this main module, the input is validated and goal state is generated and stored in a list named GOAL if input is valid. Otherwise it’ll show error message. After generating goal states, move() function is called from the module which implements the search algorithm selected by user at runtime and the nodes are printed in the order they’re visited till goal state is reached.

Tiles are moved and successor nodes are generated by swapping ‘X’ with other tiles starting from position 1 till last tile, and cost is calculated by finding how many tiles ‘X’ has moved from it’s previous position. For example, GRXRG→XRGRG results in a cost of |3-1|=2 since ‘X’ has moved from position 3 to 1. The total number of nodes expanded is the total number of visited as well as unvisited nodes generated at the time when goal state is reached.

Note- All these programs are implemented in Python

2. Breadth First Search

In this program, BFS is implemented by importing BFS module to the main module. BFS is implemented using a queue named ‘OPEN’ and a list named ‘CLOSED’ in which unvisited and visited nodes are stored respectively. Initially, the cost and number of nodes expanded is 0 and OPEN, CLOSED are empty.

First, the input state is checked to see if it is goal state. If it is not goal state, then it is appended to the list ‘CLOSED’ to mark it as visited node and successor nodes are generated and are stored in the queue ‘OPEN’ to mark them as unvisited nodes.

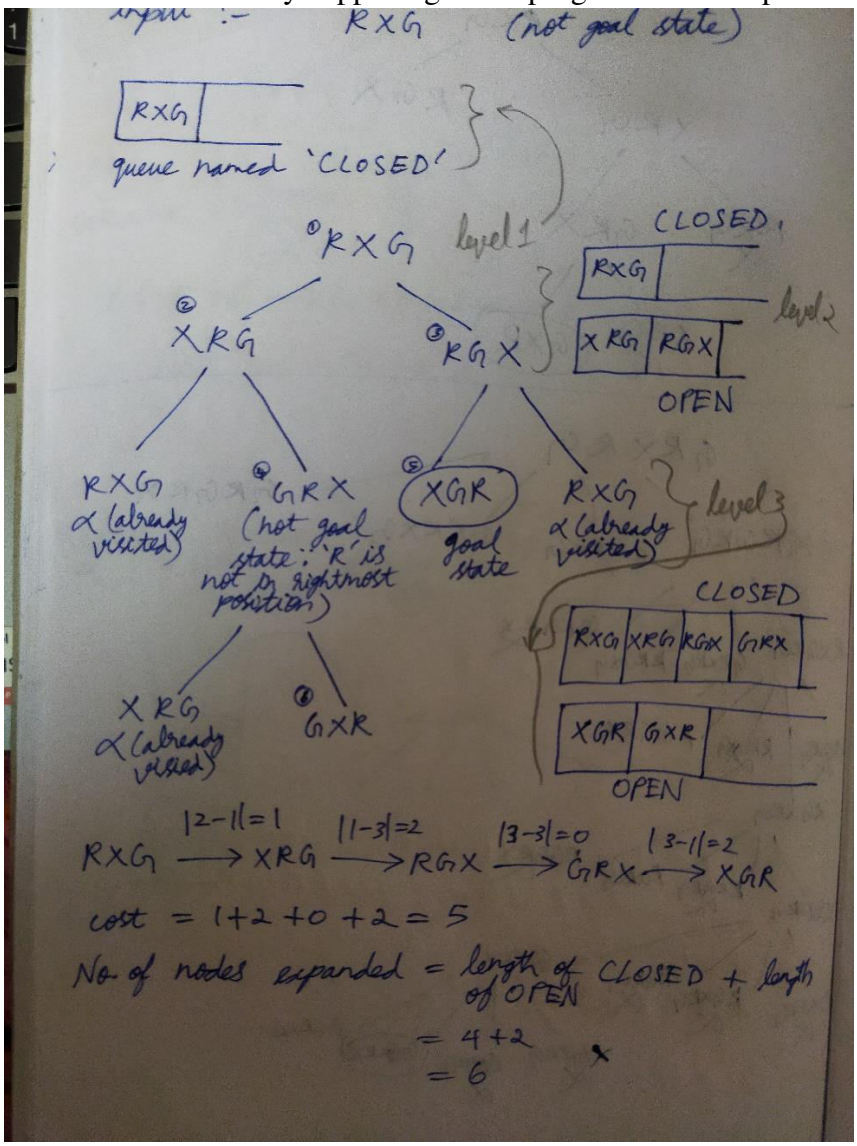
Every time a node from ‘OPEN’ is checked for goal state, it’s successors are generated and stored in queue according to First In First Out (FIFO) principle. If node is not goal state, it is popped from queue ‘OPEN’ from index 0 and appended to ‘CLOSED’ and next element of ‘OPEN’ (current element at index 0 after popping) is checked. This process is repeated till goal state is found or till OPEN is empty (i.e) there’s no solution(if no solution then it’ll return failure).

While generating successors, the nodes are added to ‘OPEN’ only if those nodes are not already in ‘OPEN’ and ‘CLOSED’. While checking each node, cost is simultaneously calculated. This program for BFS traverses horizontally at each level from left to right, and a variable is used to keep count of how many nodes to traverse in each level.

Below is an example-

```
IDLE Shell 3.10.4
File  Edit  Shell  Debug  Options  Window  Help
Python 3.10.4 (tags/v3.10.4:9d38120, Mar 23 2022, 23:13:41)
Type "help", "copyright", "credits" or "license()" for more
>>>
========== RESTART: D:\Samuela\CIT\Year 2\Semest
Enter tiles configuration: RXG
Which search do you want to perform?
1.Breadth First Search
2.Depth First Search
3.Best First Search
4.A* Search
5.Hill Clmibing Search
Enter choice: 1
MOVE 1   XRG
MOVE 3   RGX
MOVE 3   GRX
MOVE 1   XGR

Cost : 5
Total nodes expanded: 6
>>>
```

This is what is actually happening in the program for the input 'RXG'-



3. Depth First Search

In this program, DFS is implemented by importing DFS module to the main module. DFS is implemented using a stack named 'OPEN' and a list named 'CLOSED' in which unvisited and visited nodes are stored. Initially, the cost and number of nodes expanded is 1 and 0 respectively, and OPEN, CLOSED are empty.

First, the input state is checked to see if it is goal state. If it is not goal state, then it is appended to the list 'CLOSED' to mark it as visited node and is present as first element in stack 'OPEN', and successor nodes are generated and are stored in the stack 'OPEN'.

Every time the topmost node from 'OPEN' is checked for goal state, it is appended to list 'CLOSED' and it's successors are generated and stored in the stack 'OPEN' according to Last In First Out (LIFO) principle. If node is not goal state, then it's successors are generated and stored in 'OPEN'. If the topmost element of OPEN which is currently being checked for goal state has no unvisited children, it is popped from the stack 'OPEN' and next element which is the topmost element in OPEN is checked.

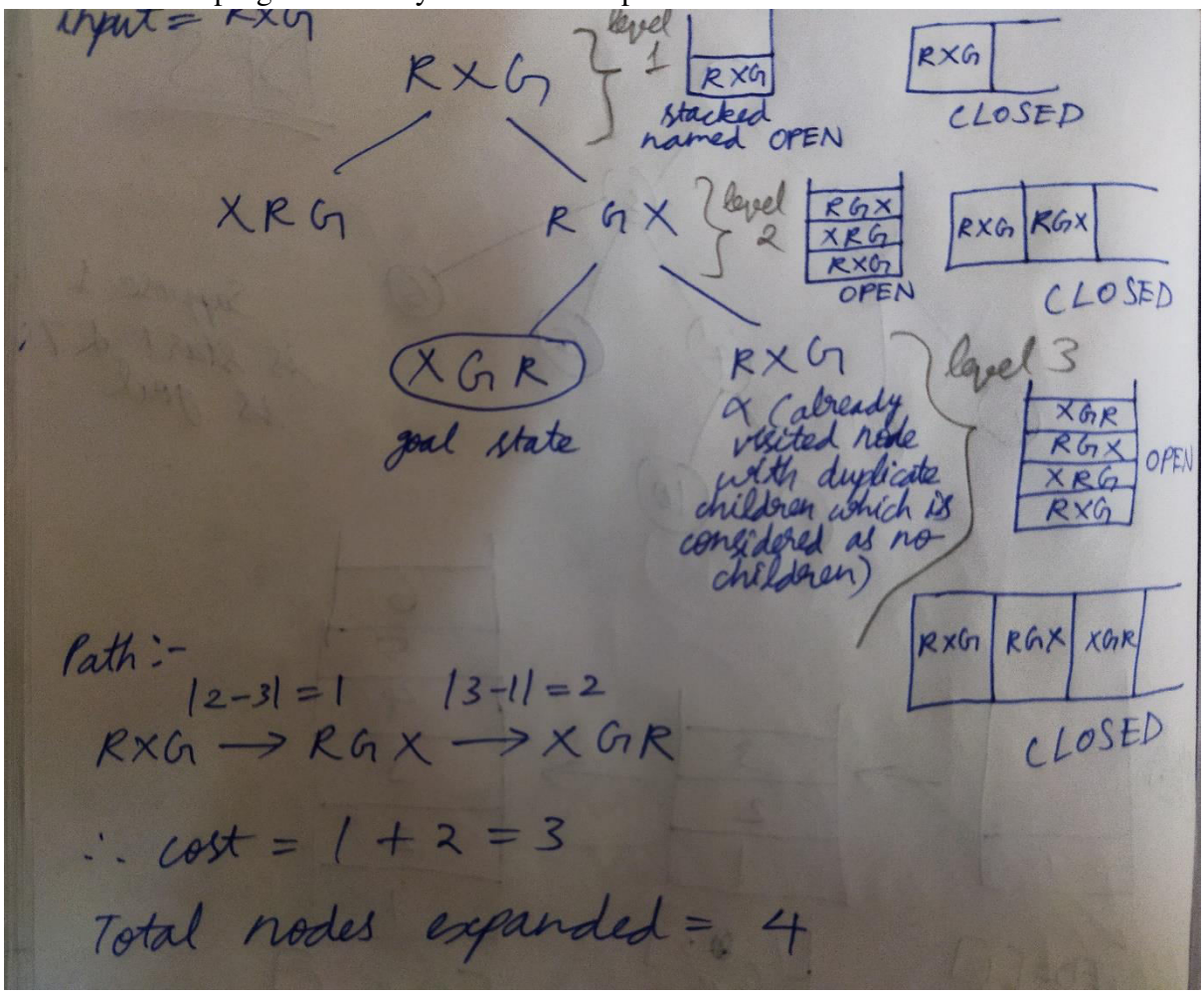
This process is repeated till goal state is found or till OPEN is empty (i.e) there's no solution (if no solution then it'll return failure). While generating successors, the nodes are added to 'OPEN' only if those nodes are not already in 'OPEN' and 'CLOSED'.

While checking each node, cost is simultaneously calculated in the same way as in BFS. This program for DFS traverses vertically at each level from right to left, and number of total nodes expanded is incremented by 1 every time a node is pushed to 'OPEN'.

This is an example-

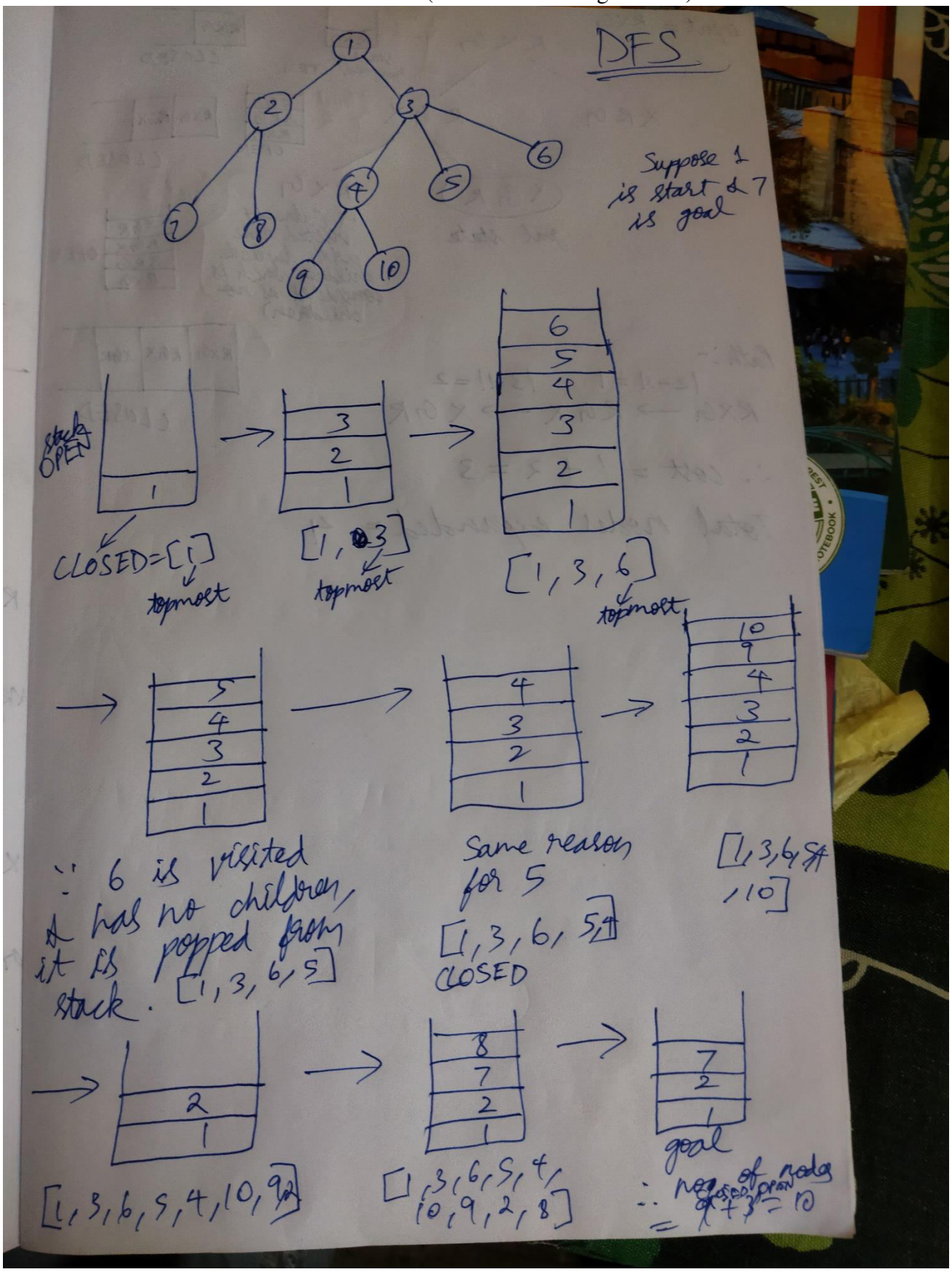
```
========== RESTART: D:\Samuela\CIT\Year 2\Semester 3\Assignments'
Enter tiles configuration: RXG
Which search do you want to perform?
1.Breadth First Search
2.Depth First Search
3.Best First Search
4.A* Search
5.Hill Clmibing Search
Enter choice: 2
MOVE 3   RGX
MOVE 1   XGR

Cost : 3
Total nodes expanded: 4
```

This is what the program actually does for the input ‘RXG’-



Since this example is too short and small to understand the logic we’ve used, below is another example-

You can see nodes at each stage being popped and moved into a list with square brackets named CLOSED. The stack is named OPEN (1 is start and 7 is goal state)-



4. Best First Search

In this program, best first search is implemented by importing BEST module to the main module. This is implemented using a priority queue named 'OPEN' nodes are stored. Initially, the number of nodes expanded is 0 and OPEN is empty. Heuristic cost is taken as the position of 'X' in each node, and hence cost is initialized to position of 'X' in start state.

First, the input state is checked to see if it is goal state. If it is not goal state, then it's successors are generated and inserted at position 1 of 'OPEN' after sorting nodes cost-wise in ascending order such that the node at first position of 'OPEN' has the least heuristic cost.

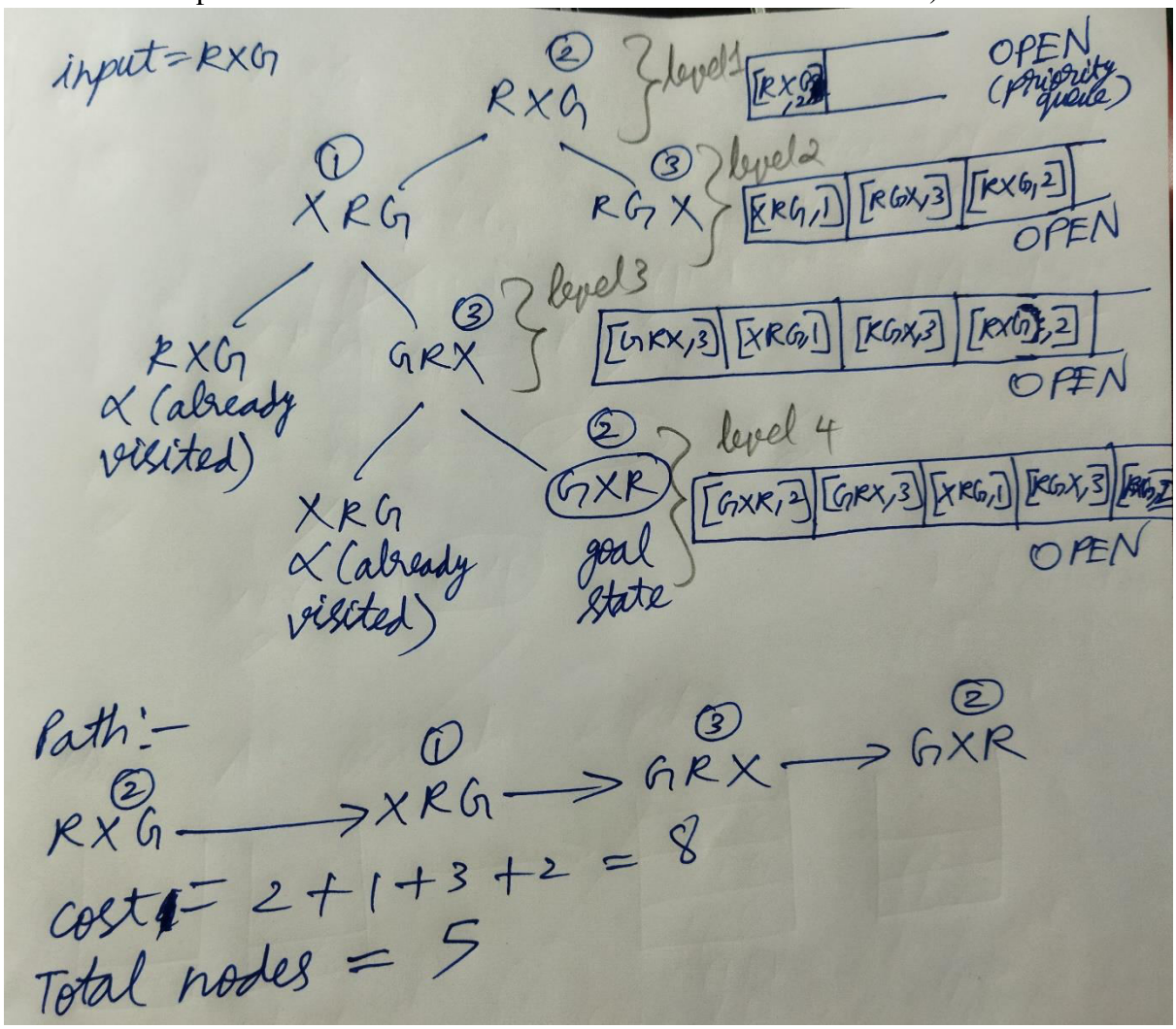
Then again the node in first position of 'OPEN' is checked for goal state and this process is repeated till goal state is found or till OPEN is empty (i.e) there's no solution (if no solution then it'll return failure). While generating successors, the nodes are added to 'OPEN' only if those nodes are not already in 'OPEN'. And if the first node being checked doesn't have any children or if all of it's children are already present in OPEN, then that node is popped and successors are generated for the next node, thereby having the feature of backtracking.

While checking each node, cost is simultaneously calculated. This program traverses vertically at each level starting from node having least heuristic cost, and number of total nodes expanded is the total number of nodes 'OPEN' when the goal state is found. Basically, best first search finds goal state via a path which has low cost i.e seems best at that moment.

This is an example-

```
========== RESTART: D:\Samuela\CIT\Year 2\Semester 3\Assignments\Princip
Enter tiles configuration: RXG
Which search do you want to perform?
1.Breadth First Search
2.Depth First Search
3.Best First Search
4.A* Search
5.Hill Clmibing Search
Enter choice: 3
MOVE 1   XRG
MOVE 3   GRX
MOVE 2   GXR

Cost : 8
Total nodes expanded: 5
```

This is what is actually happening in the program for the input ‘RXG’ (the square brackets i.e list inside each position of OPEN contains the node and it’s heuristics cost) -



5. A* Search

In this program, A* search is implemented by importing ASEARCH module to the main module. A* search is implemented using 2 stacks named ‘OPEN’ and ‘temp_sol’ in which nodes and explored paths are stored respectively. Initially, the cost and number of nodes expanded is 1 and 0 respectively, and OPEN, ‘temp_sol’ are empty.

First, the input state is checked to see if it is goal state. If it is not goal state, then it is appended to both the stacks, and successor nodes are generated and are stored in the stack ‘OPEN’ if those nodes are not already in OPEN. Then topmost element in ‘OPEN’ is pushed to ‘temp_sol’.

Every time the topmost node from ‘temp_sol’ is checked for goal state, successors of topmost node of ‘OPEN’ are generated and stored in the stack ‘OPEN’ if they are not already present in OPEN according to Last In First Out (LIFO) principle. If node is not goal state, then topmost node of ‘OPEN’ is pushed to ‘temp_sol’ and the process continues.

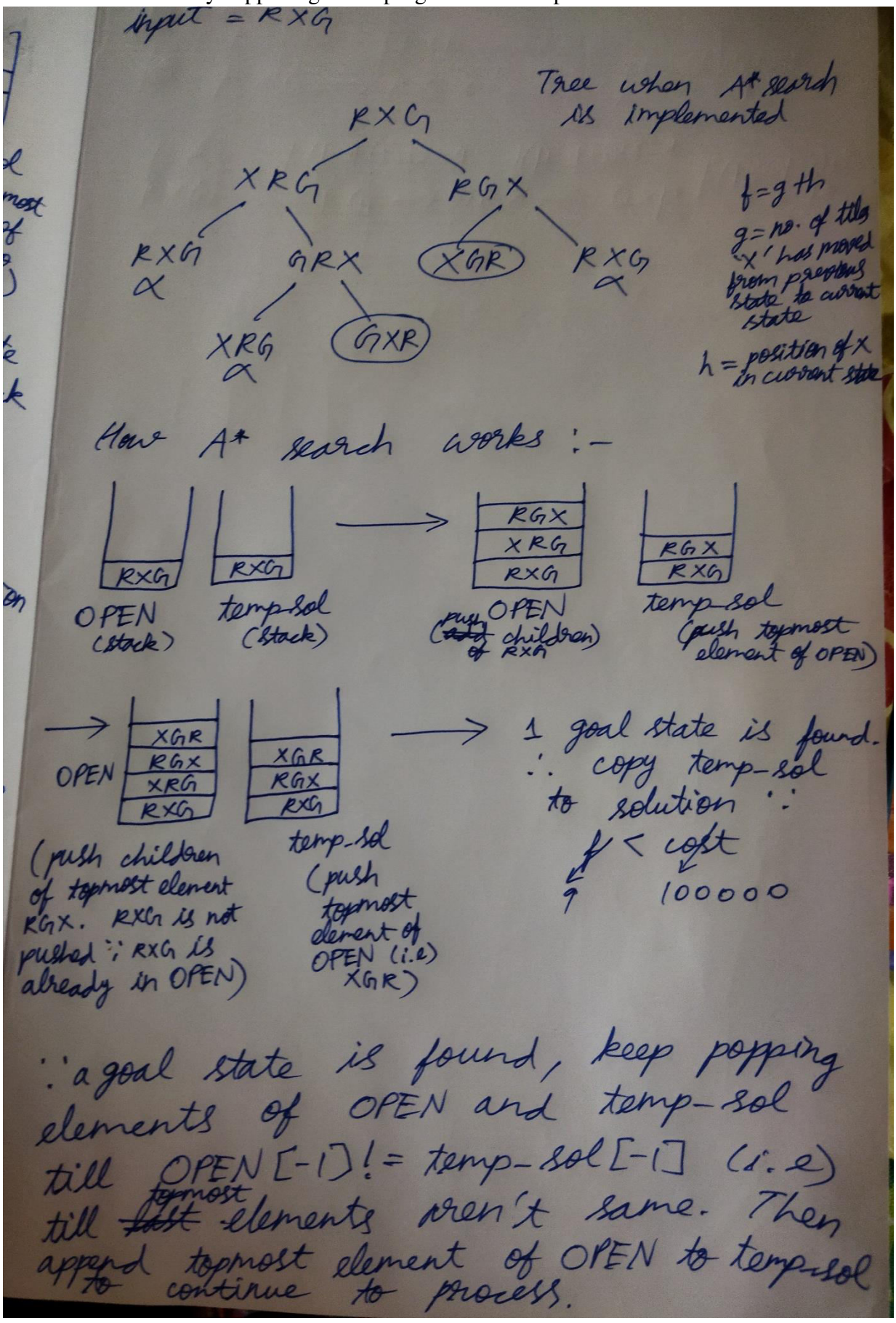
If the topmost element of ‘temp_sol’ is goal state, then the cost calculated till now is compared with solution cost, and if the current cost is less, then ‘temp_sol’ is copied to a list called ‘solution’ and cost is updated. Whether ‘solution’ is updated or not, after a goal state is encountered the elements of OPEN and temp_sol are popped as long as their topmost nodes are equal. Then after popping, the topmost element of ‘OPEN’ is appended to ‘temp_sol’ and the A* search continues till ‘OPEN’ becomes empty.

If both OPEN and ‘solution’ are empty, then there’s no solution (if no solution then it’ll return failure). A* search traverses vertically from right to left as in DFS, but in A* search all paths are explored and the path with least cost is printed. Here, the cost is calculated as f=g+h where g is the number of tiles ‘X’ moved from previous position to current position and h is the position of ‘X’ in current node.

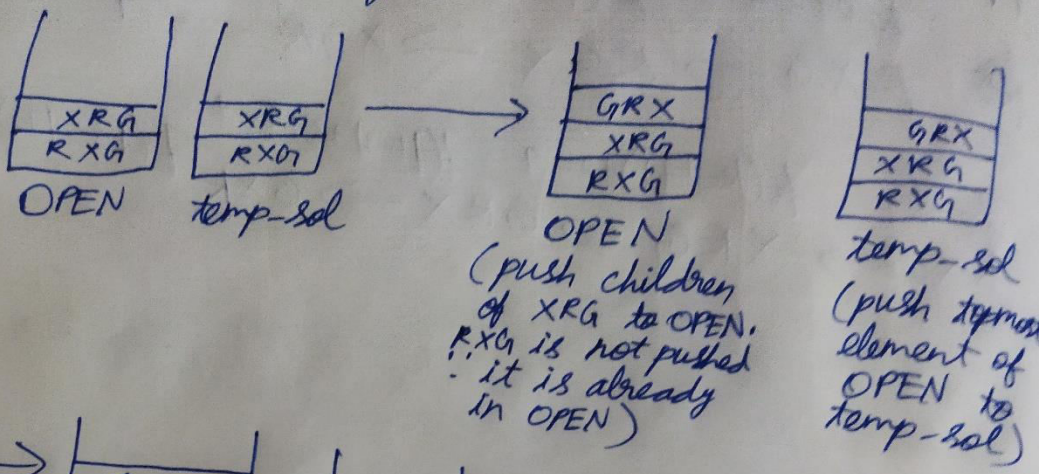
This is an example-

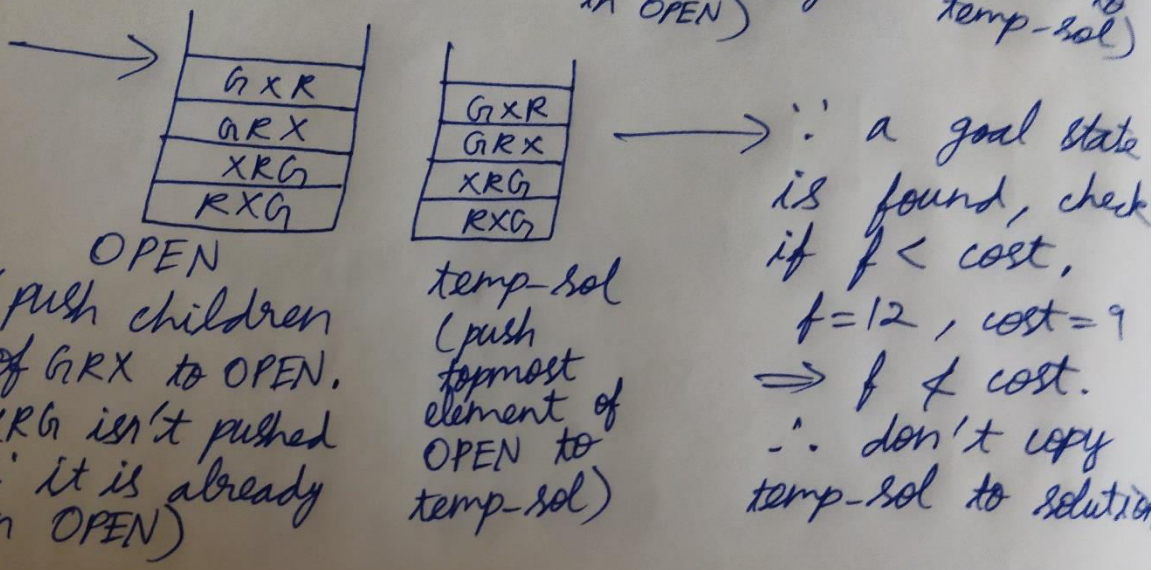
```
>>
========== RESTART: D:\Samuela\CIT\Year 2\Semester 3\Assignments\Princ
Enter tiles configuration: RXG
Which search do you want to perform?
1.Breadth First Search
2.Depth First Search
3.Best First Search
4.A* Search
5.Hill Clmibing Search
Enter choice: 4
MOVE 3   RGX
MOVE 1   XGR

Cost : 9
Total nodes expanded: 6
>>
```

This is what is actually happening in the program for the input 'RXG'-



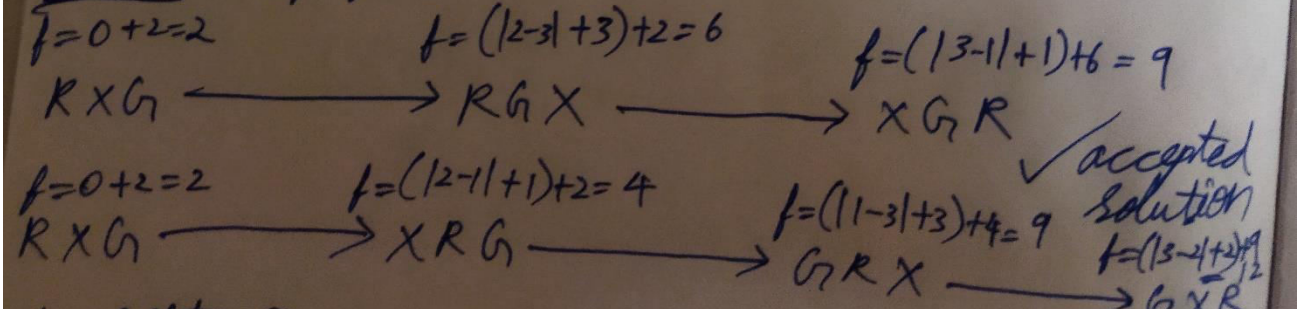
After exploring path 1 :-





∵ another goal state is found, keep popping elements of OPEN and temp_sol till OPEN[-1] == temp_sol[-1]. ⇒ both stacks have become empty. ∴ stop A* search.

Paths :- (f = g + h)



∴ cost = 9

Total nodes expanded = 6

6. Hill Climbing Search

In this program, hill climbing search is implemented by importing HC module to the main module. This is implemented using a priority queue named 'OPEN' and a list named 'CLOSED' in which unvisited and visited nodes are stored. Initially, the number of nodes expanded is 0, and OPEN, CLOSED are empty. Heuristic cost is taken as the position of 'X' in each node, and hence cost is initialized to position of 'X' in start state.

First, the input state is checked to see if it is goal state. If it is not goal state, then it is removed from 'OPEN' and appended to the list 'CLOSED' to mark it as visited node, and successor nodes are generated and are inserted in the first position of priority queue 'OPEN' after sorting nodes cost-wise in ascending order such that the node at first position of 'OPEN' has the least heuristic cost.

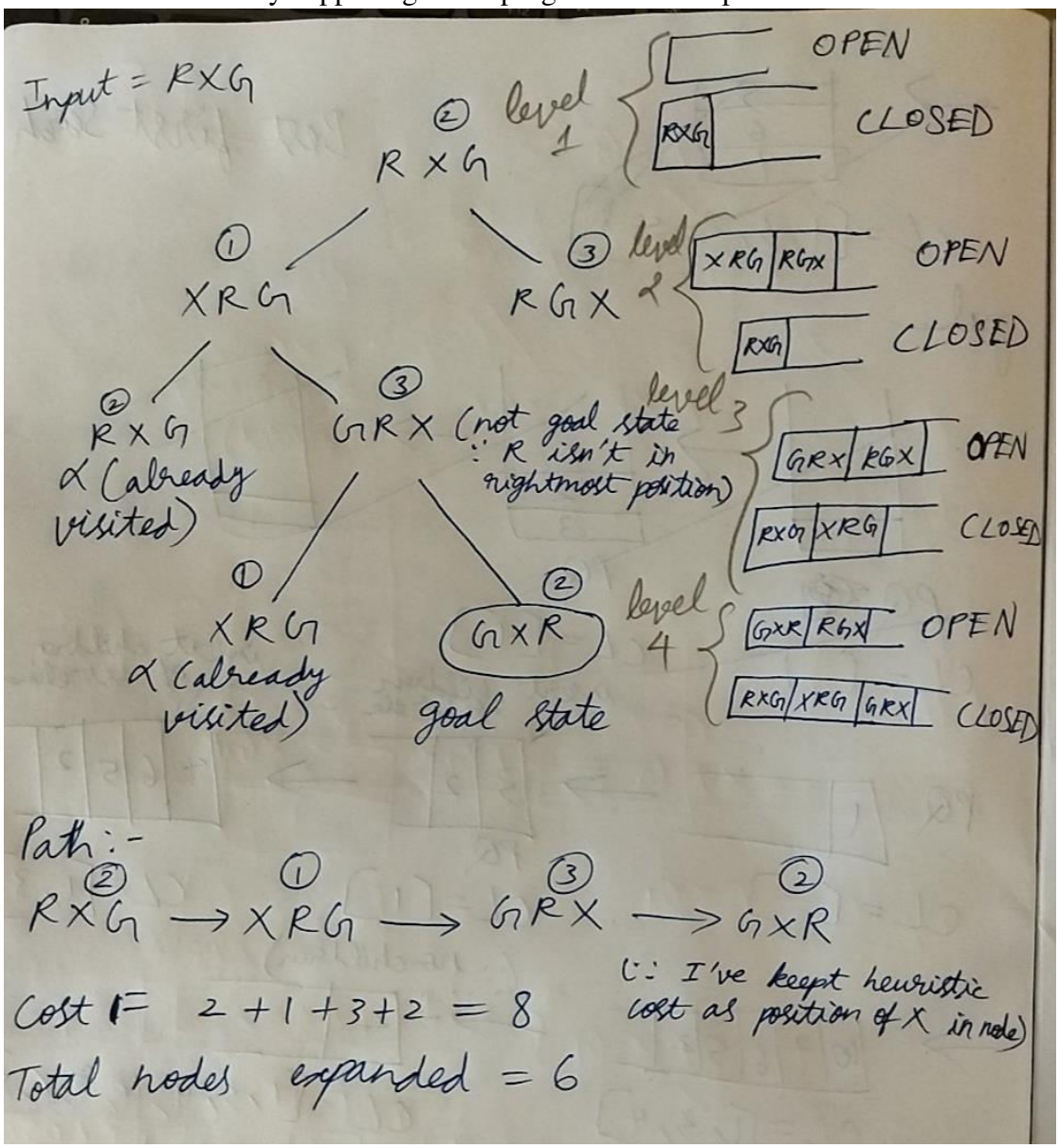
Then again the node in first position of 'OPEN' is checked for goal state and this process is repeated till goal state is found or till OPEN is empty (i.e) there's no solution (if no solution then it'll return failure). While generating successors, the nodes are added to 'OPEN' only if those nodes are not already in 'OPEN' and 'CLOSED'.

While checking each node, cost is simultaneously calculated. This program traverses vertically at each level starting from node having least heuristic cost, and number of total nodes expanded is the total number of nodes in 'CLOSED' and 'OPEN' when the goal state is found. *The only difference between hill climbing search and best first search is that the former can't backtrack whereas the latter can backtrack.*

This is an example-

```
=============== RESTART: D:\Samuela\CIT\Year 2\Semester 3\Assignments\Princi
Enter tiles configuration: RXG
Which search do you want to perform?
1.Breadth First Search
2.Depth First Search
3.Best First Search
4.A* Search
5.Hill Clmibing Search
Enter choice: 5
MOVE 1   XRG
MOVE 3   GRX
MOVE 2   GXR

Cost : 8
Total nodes expanded: 5
>
```

This is what is actually happening in the program for the input ‘RXG’-



7. Conclusion and Thoughts

Although A* search finds solution path having least cost, it is time consuming and occupies a lot of space. Hill climbing can get stuck at local maxima and doesn’t have backtracking feature, and best first search gives the path which is seemingly best at that moment. BFS and DFS also expand many nodes to get solution.

So best first search is the best algorithm here since it can backtrack, finds solution which is optimal compared to others since it uses heuristics, and it’s worst case time complexity is

O(n * log n) . The performance of these programs can be improved by writing efficient codes and optimized heuristic values.

8. Team Contributions

Coding Part	Contributor
main_game_Sam_Niv_Hari.py	Samuela Abigail- coded everything (getting input, input validation, goal state generation)
BFS.py	Samuela Abigail- coded everything (implementation of queue, generation of nodes, implementation of breadth-wise traversal of nodes, validation, etc.)
DFS.py	Samuela Abigail- coded everything (implementation of stack, generation of nodes, implementation of depth-wise traversal of nodes, validation, etc.)
BEST.py	Haripriya- wrote basic skeleton of algorithm Samuela Abigail- implemented priority queue, implemented DFS-like traversing logic, applied heuristics and backtracking, implemented node generation, validation, bug fixing, etc.
ASEARCH.py	Nivetha- wrote basic skeleton of algorithm Samuela Abigail- implemented backtracking and stacks, implemented DFS-like traversing logic, applied heuristics, implemented finding of solution path with lowest cost, implemented node generation, validation, bug fixing, etc.
HC.py	Samuela Abigail- implemented priority queue, implemented DFS-like traversing logic, applied heuristics, implemented node generation, validation, bug fixing, etc.

9. References

BFS

For getting basic idea of how BFS works and why queue is required for BFS implementation-

https://www.interviewkickstart.com/learn/breadth-first-search-algorithm

https://www.tutorialspoint.com/data_structures_algorithms/breadth_first_traversal.htm

DFS

For getting basic idea of how DFS works and why stack is required for DFS implementation-

https://www.tutorialspoint.com/data_structures_algorithms/depth_first_traversal.htm

https://www.youtube.com/watch?v=ccyWjW4Fo34

Best First Search

For getting basic idea of how this search algorithm works and why priority queue is required to implement it-

https://www.cet.edu.in/noticefiles/271_AI%20Lect%20Notes.pdf (page 42)

A* Search

For getting basic idea of how A* search algorithm works and how DFS is required to implement it-

https://www.youtube.com/watch?v=PzEWHH2v3TE

Hill Climbing Search

For getting basic idea of how A* search algorithm works and how DFS is required to implement it-

http://www.nou.ac.in/Online%20Resourses/27-4/MCA%20part%203%20paper%2021.pdf (page 10)