Artificial Intelligence Project#1

Diagonal 15-Puzzle

Project Design Document

Rıdvan Gülcü – 150117508

Berre Ergün – 150117507

Node Class

There ara 6 different fields in the Node class. In “state" we keep the current snapshot of puzzle with a 2d integer array. The “parentNode” keeps the information about which node created this node. “realCost” keeps the total path cost for this state. “estimatedCost" keeps the estimated cost for remaining to final state. We initialize zero by default because some algorithms dost work with heuristic strategies such as UCS, ILS. “move” keeps the direction of movement to come the current state. We created the state Text field for easier comparison between statuses. "stateText" keeps the integer array of 2d as a string. Node also implements the Comparable interface, which allows you to sort according to the cost of the nodes.

Utility Class

Utility class has the general functions that more than one class needs and uses. Their description is given below.

stateCompare: State compare has 2 parameters: state1 and state2 with types String. This method simply compares these 2 states and if they are equal returns true, if they are not false.

getAllowableMoves: This method has 1 parameter named emptyTileCoordinate which is an integer array and store coordinate of zero. In our board empty tile can move up, down, right, left, upright, upleft, downright, downleft. But if the empty tile is at corners or edges it can’t move to certain areas in order not to leave the board. This method is for that purpose, It returns the directions that tile can go according to current coordinate.

detectTileCoordinate: This method gets the tile and the current state and returns tile’s coordinates according to state.

moveState: moveState has 2 parameters: state and direction. With detectTileCoordinate method it gets empty tiles coordinates. According to given direction, method updates the state with swap method. This method returns move cost in integer type.

swap: Swap method gets coordinates of the 2 tiles which will be swapped, and the state. And with swapping It updates state.

copyState: Copy state is a method to get a shallow copy.

printState: Prints the given state.

printPath: It takes the final node as a parameter and prints the solution path.

printSolution: After the solution is found, it prints the cost of the solution, the total number of expanded nodes and the maximum number of nodes kept in memory.

GraphSearch Class

GraphSearch class contains methods about our search methods(UCS,ILS,A*).

graphSearch: grapSearch is our main algorithm method. It has 3 parameters: initialState, strategy and heuristic function. According to the strategy(UCS,ILS,A*) It calls lengLimithedSearch with different parameters. For A* and UCS lengthLimithedSearch's length limit is infinite. But for ILS length limit increases in a while loop and lengthLimitedSearch is being called again and again until the solution is found or time limit is exceeded. This method returns a solution array with the values expanded node number and max stored node number.

lengthLimitedSearch: This method has 4 parameters: initialState, strategy, heuristic function and length limit. According to strategy initial cost is calculated. If strategy is A*, our heuristic function 1, 2 and 3 calculates the cost. For UCS and ILS initial cost is 0. After that initial node, frontier list and explored set list is created. In a while loop nodes in the frontier list are expanded with ascending order by their cost. After every expand with our expandNode method, expanded node counter increases. This while loop continues until frontier list is empty, goal node is found or time limit is expanded. This method returns the node if the solutions is found, or null if solution is not found.

goalStateCompare: It compares the given node's state with the goal state using our stateCompare method.

move: Move method has 2 parameters: current node which type is Node(our Node class) and direction. With our moveState method, It gets move cost and creates new node with changed state and cost. After that returns that newNode.

expandNode: This method gets 6 parameters: current node, frontier list, explored set, length limit, strategy and heuristic function. First, allowable moves calculated with our allowableMoves method. For every move, according to strategy estimated move cost is calculated. New node is created after the move, if this node is not in the explored set or in frontier list or the cost is not more than our length limit we add the node to our frontier list.And at the end we check frontier list size, if it is grater than our max stored node number we update it.

isExplored: This method checks if the new node is in explored set. If our new node's state same and child node's path cost is greater than or equal to the path cost of a node in explored set, method returns true. If the node is not explored it returns false. We don't need to check parent node because path cost check include parent check and algorithm is still consistent.

isInForntier: This method checks if the node is in frontier list just like the isExplored method.

heuristicFuntion1: This method calculates miss-placed tiles number except empty tile and returns it.

heuristicFunction2: This method calculates city block distance for all tiles except zero and returns it.

checkTime: check time calculates this run's time and compares it with the time limit, in our program our time limit is 5 minutes. If that time is expanded method returns true.

StateMixer Class

mixState: mixState function takes goalState and depth.It mixes the puzzle by making random moves as many as the depth. We do not allow the movement in the opposite direction of the previous move for prevent cycles.

Outputs

We ran each algorithm 10 times for each depth level and got average values. We set a time limit of 5 minutes for the algorithm to work, and we said no solution for algorithms that exceed this period. While taking the average, we only considered the solutions found. For this reason, we added another table which shows us the number of solved problems.

The number of solved problems within time limits				
d:depth of the solution	UCS	ILS	A* -H1	A*-H2
2	10	10	10	10
4	10	9	10	10
6	9	3	10	10
8	7	—	10	10
10	4	—	9	10
12	—	—	6	10
16	—	—	4	10
20	—	—	0	9
24	—	—	0	9
28	—	—	0	4

Table 1

The Total number of expanded nodes				
d:depth of the solution	UCS	ILS	A* -H1	A*-H2
2	60	135	4	2
4	1052	5703	22	12
6	5633	20228	376	26
8	6530	—	1227	55
10	10185	—	4135	165
12	—	—	3524	318
16	—	—	10594	947
20	—	—	—	1025
24	—	—	—	5884
28	—	—	—	9412

Table 2

The maximum number of nodes stored in memory				
d:depth of the solution	UCS	ILS	A* -H1	A*-H2
2	196	61	18	12
4	3358	2228	85	49
6	17334	9004	1276	104
8	20839	—	3993	203
10	32099	—	12707	586
12	—	—	11040	1061
16	—	—	32222	3144
20	—	—	—	3198
24	—	—	—	17200
28	—	—	—	28031

Table 3

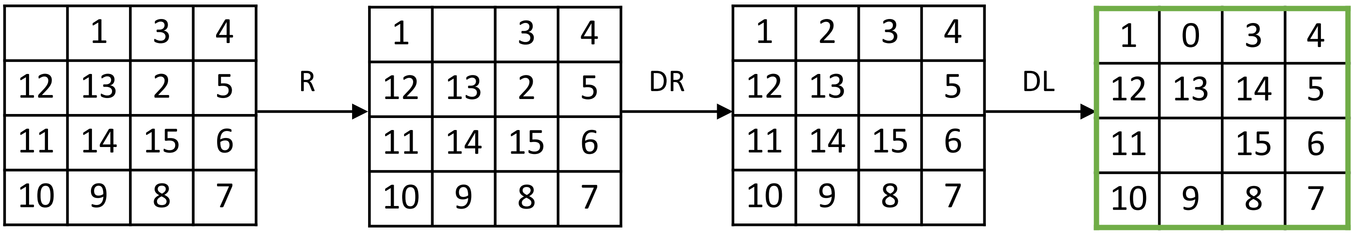
Since there is a randomness in the state mixing algorithm, time complexity may decrease in some cases where depth increases. In this case, it is necessary to consider Table 1.

Solution of a,b,c

A)

Cost of solution found = 7 (Cost of solution is same for all algorithms)

Solution path: (Solution path is same for all algorithms)



	1	3	4
12	13	2	5
11	14	15	6
10	9	8	7

R

1		3	4
12	13	2	5
11	14	15	6
10	9	8	7

DR

1	2	3	4
12	13		5
11	14	15	6
10	9	8	7

DL

1	0	3	4
12	13	14	5
11		15	6
10	9	8	7

Number of expanded nodes :

UCS = 530

ILS = 1278

A* H1 = 18

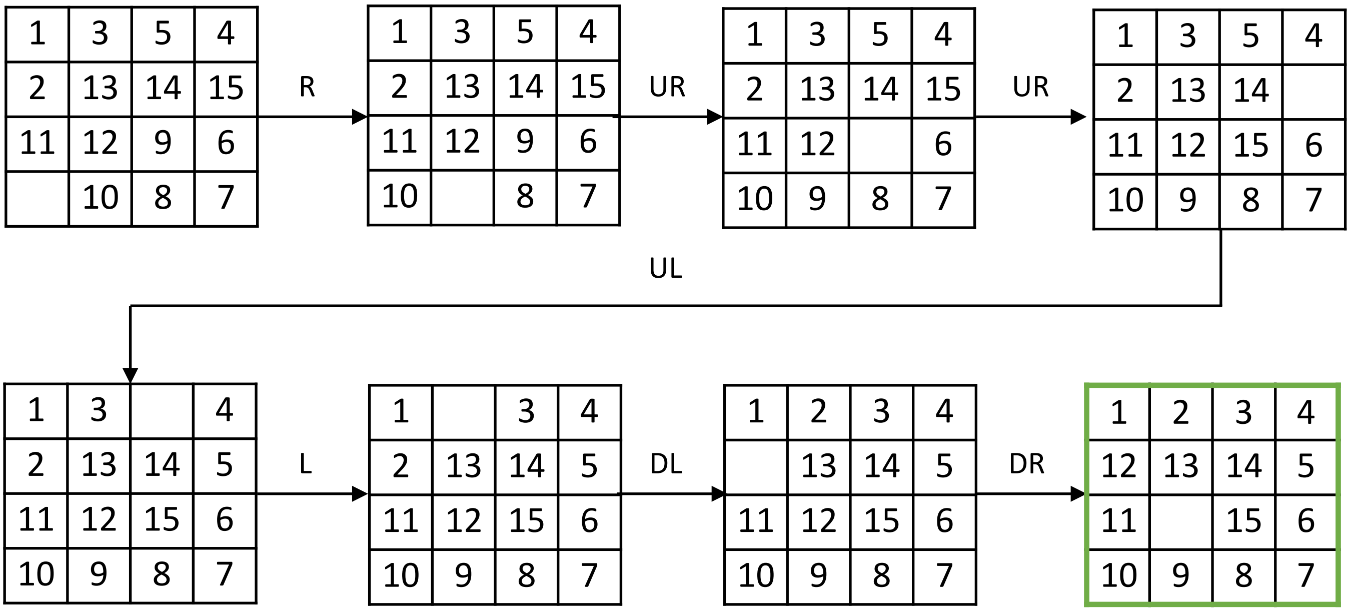
A* H2 = 5

B)

UCS and ILS didn’t solve this problem within 20 minutes.

Cost of solution found = 17 (Same result for h1 and h2)

Solution path: (same path for h1 and h2)



1	3	5	4
2	13	14	15
11	12	9	6
	10	8	7

R

1	3	5	4
2	13	14	15
11	12	9	6
10		8	7

UR

1	3	5	4
2	13	14	15
11	12		6
10	9	8	7

UR

1	3	5	4
2	13	14	
11	12	15	6
10	9	8	7

UL

1	3		4
2	13	14	5
11	12	15	6
10	9	8	7

L

1		3	4
2	13	14	5
11	12	15	6
10	9	8	7

DL

1	2	3	4
	13	14	5
11	12	15	6
10	9	8	7

DR

1	2	3	4
12	13	14	5
11		15	6
10	9	8	7

Number of expanded nodes :

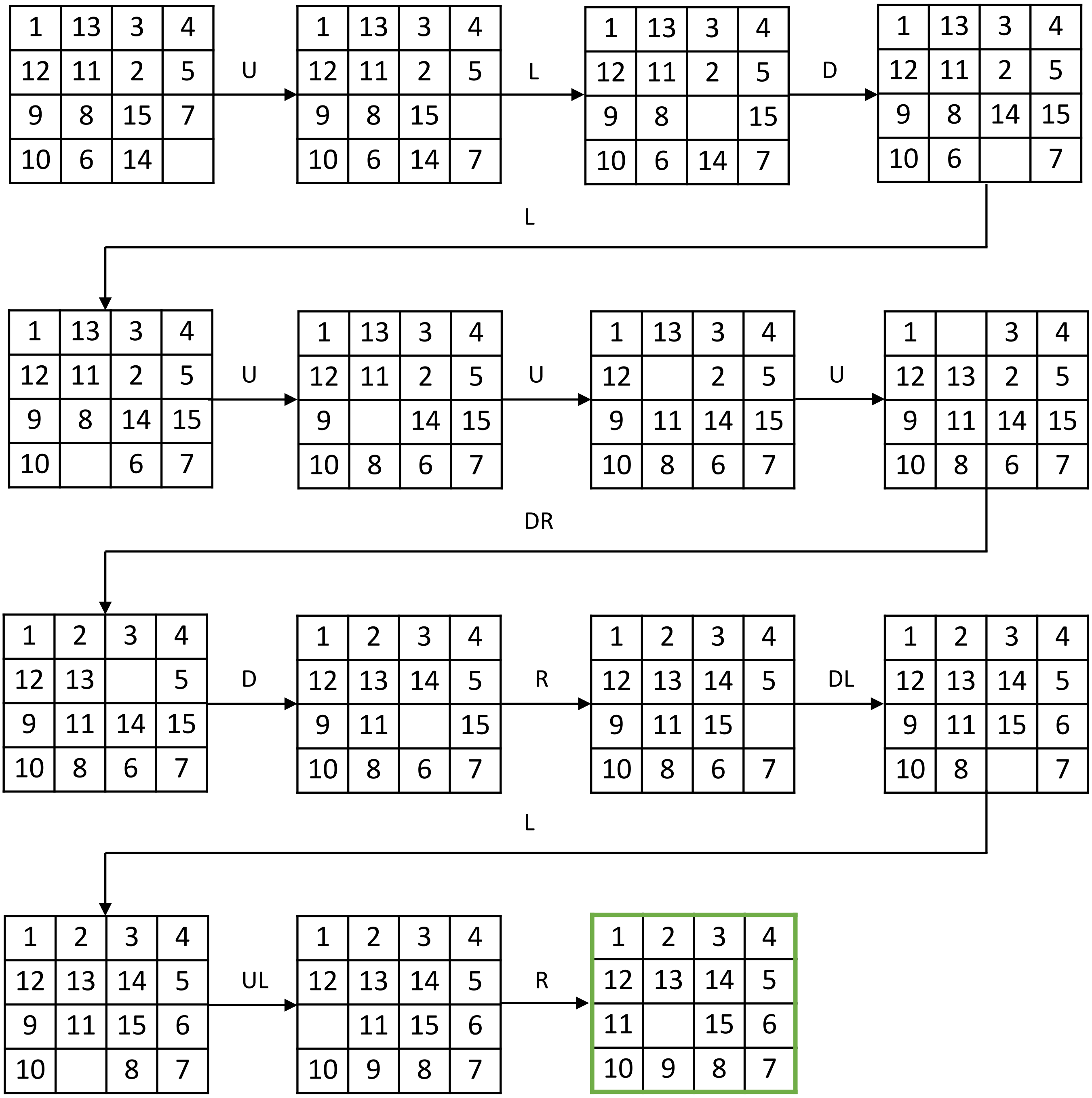
A* H1 = 2251 , A* H2 = 176

C)

UCS and ILS didn’t solve this problem within 20 minutes.

Cost of solution found = 20 (Same result for h1 and h2)

Solution path: (same path for h1 and h2)



1	13	3	4
12	11	2	5
9	8	15	7
10	6	14	

U

1	13	3	4
12	11	2	5
9	8	15	
10	6	14	7

L

1	13	3	4
12	11	2	5
9	8		15
10	6	14	7

D

1	13	3	4
12	11	2	5
9	8	14	15
10	6		7

L

1	13	3	4
12	11	2	5
9	8	14	15
10		6	7

U

1	13	3	4
12	11	2	5
9		14	15
10	8	6	7

U

1	13	3	4
12		2	5
9	11	14	15
10	8	6	7

U

1		3	4
12	13	2	5
9	11	14	15
10	8	6	7

DR

1	2	3	4
12	13		5
9	11	14	15
10	8	6	7

D

1	2	3	4
12	13	14	5
9	11		15
10	8	6	7

R

1	2	3	4
12	13	14	5
9	11	15	
10	8	6	7

DL

1	2	3	4
12	13	14	5
9	11	15	6
10	8		7

L

1	2	3	4
12	13	14	5
9	11	15	6
10		8	7

UL

1	2	3	4
12	13	14	5
	11	15	6
10	9	8	7

R

1	2	3	4
12	13	14	5
11		15	6
10	9	8	7

Number of expanded nodes :

A* H1 = 16594

A* H2 = 431

As a result, no algorithm except A * h2, didn't solve all the depth levels within the given time. Difference of A*H1 and A*H2 shows how effective a good heuristic strategy. As long as the cost of the diagonal movement is above 2, we think there is no better admissible heuristic than city-block distance. We thought about it for a long time, but we found a case that would make the heuristic non admissible, although the heuristics we found gave optimal results in a very short time.

When we analyzed UCS and ILS, we decided that it makes more sense to use UCS for this problem. Although ILS provides an advantage in memory usage, it could only give us a solution up to the 6.depth level. But we were able to reach a solution up to the 10th depth level with UCS.

Since all algorithms are consistent, we reached optimal results with all of them, but as it is clear according to the outputs, A * h2 gave better results than other algorithms.