HO CHI MINH UNIVERSITY OF SCIENCE - FACULTY

Lab01
Search Strategies

18127022 – Pham Ngoc Thuy Trang

Course: Introduction to AI - 18CLC2

PROBLEM OF LAB01

Problem: A robot has been sent to a maze of size $N \times N$ and it must navigate the maze to reach the exit. Given that the robot has the map of the maze and it knows that its initial location is at square 0 and needs to reach some square E (from 0 to $N2 - 1$) to go out of the maze. Figures 1 demonstrates a maze of size 8×8 and the exit E is at square 61.

Input: The maze is represented in a text file as follows.

- The first line includes a positive integer N only, which indicates the size of the maze.

- Each of the next $N \times N$ lines contains an adjacency list of node i. Nodes in the list are unordered and separated by white spaces.

- The last line includes a non-negative integer E (from 0 to $N2 - 1$), which indicates the exit.

Output: Print to the console the following information

- The time to escape the maze

- The list of explored nodes (in correct order)

- The list of nodes on the path found (in correct order).

REFERENCES

[1]: a 5x5 maze from friends

[2]: Artificial Intelligence: A Modern Approach 3th edition of Russell and Peter Norvig

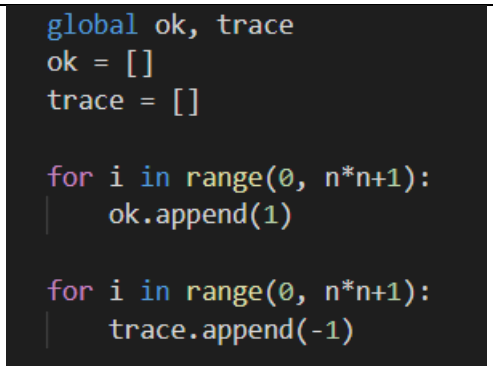
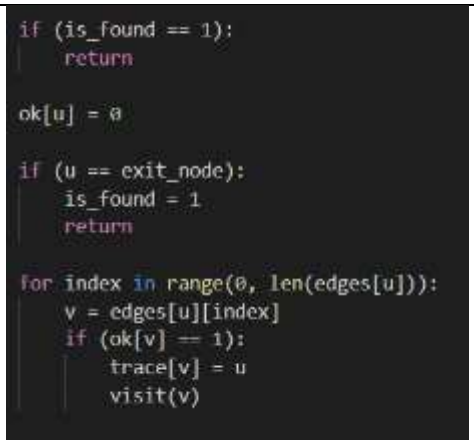
BRIEF DESCRIPTION

Name of (Function/Variable…)	Description/Meaning	Main Piece of Code

`edges = []`
`exit_node = 0`
`n = 0`
`ok = []`
`trace = []`
`queue = []`
`times = 0`
`is_found = 0`
	Initialize the required attributes of the maze like:
	• Time to escape from the maze (based on how many steps we use to escape the maze)
	• List of explored nodes
	• List of nodes on the founded path
	• A variable that check whether the exist node is found
	• List of the maze's edges
	• Size of maze	

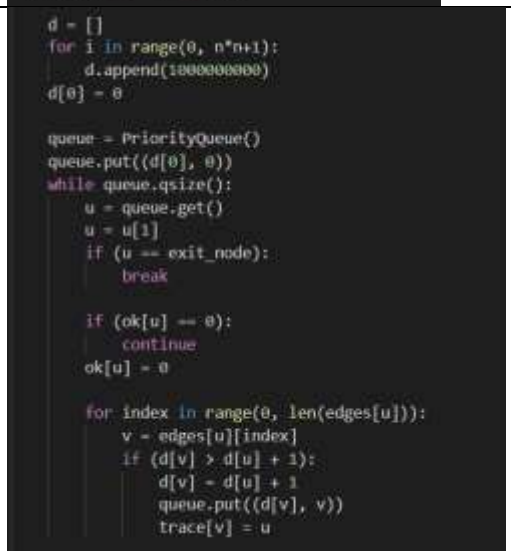
`def distance(src, dst):`	This function is used to calculate the Manhattan distance as heuristic by calculating the total length of the projection of the line connecting these two points in the Cartesian coordinate system.	`global n`
		`x1 = src % n`
		`y1 = int(src/n)`
		`x2 = dst % n`
		`y2 = int(dst/n)`
		`return abs(x1-x2) + abs(y1 - y2)`

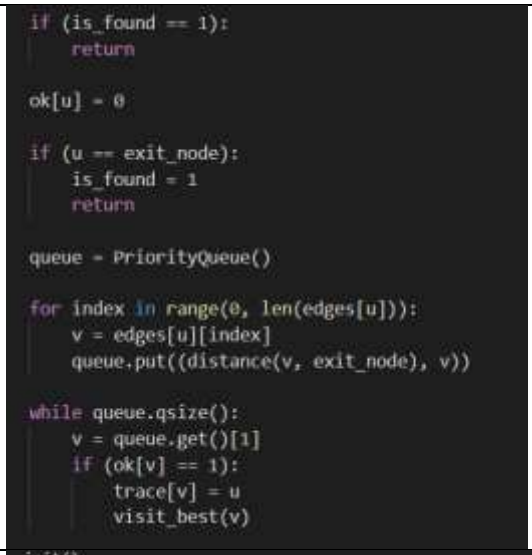
`def read_input():`	This function reads the input value from maze input file and includes:
	• The first line is a positive
		`for i in range(0, n*n):`
		`  edges.append([])`
		`  edges[i] = []`
		`  line = inp[cnt]`
		`  cnt = cnt + 1`
		`  list_neary_by = line.split(' ')`
		`  for j in range(0, len(list_neary_by)):`
		`  v = int(list_neary_by[j])`
		`  edges[i].append(v)`

	interger n (nxn is the maze's size) • Put an edge of the current node in list edges • Determinate the exit node of the maze	
`def init():`	This function is used to initialize the value (prepare data before searching) for the list ok by assigning each node a value of 1 *(this list will be used for storing expanded nodes)* and list trace *(this list is used for saving track)* by assigning a value of -1. Those above lists will have the same size of nxn	
`def visit(u):`	This function is used to browse the graph and search for the goal node for the IDS algorithm. If the target node is found, then return, otherwise continue to browse the vertices and put them in the expanded node list.	

`def IDS():`	This function is based on the way DFS searches with max-depth being extremely positive. However, we do not use the queue as the BFS algorithm. It starts from any vertex to any of its adjacent vertices and then saves these vertices. Then it continues taking the latest peak which has been just saved and goes until there is no longer able to go. Because the number of vertices is finite, this algorithm certainly finds the result This function also returns the time to exit the maze based on the number of moves and the real time system	 ``` init() start = time.time() global is_found is_found = 0 visit(0) global times print("IDS") g.write("IDS\n") times = 0 print_result(exit_node) print() g.write("\n\n") end = time.time() g.write("Time measured: " + str(end-start) + "\n\n") print("Time measured:", end - start) ```

`def bfs():`	By using the queue from python's queue library. Instead of passing parameters, we initialize the value (prepare the data) by calling the init function.
	
	This function is used to find the exit node (node that exits the maze) using the BFS algorithm. Starting from any vertex, we go to all its adjacent vertices, saving the vertices. Continue to bring another vertex from the set of saved vertices to consider and go until there are no other vertices that can go. In the process of going from vertex to vertex, proceed to save the parent vertices of the adjacent vertex, so that when going back from the ending vertex to the starting vertex, we get the shortest path and find the node closest to The goal node is based on this heuristic
	
	The function also shows the way out of the maze, the time to exit the maze and the list of nodes has been
		
		`init()`
		`start = time.time()`
		`global trace, times, ok, is_found`
		`is_found = 0`
		`queue.append(0)`
		`trace[0] = 0`
		
		`while (len(queue) > 0):`
		`u = queue.pop(0)`
		
		`for index in range(0, len(edges[u])):`
		`if (is_found == 1):`
		`break`
		
		`v = edges[u][index]`
		`if (ok[v] == 1):`
		`queue.append(v)`
		`ok[v] = 0`
		`trace[v] = u`
		`if (v == exit_node):`
		`is_found = 1`
		`break`

	expanded. Also write that result to the file. This function also returns the time to exit the maze based on the number of moves and the real time system	
`def print_result(u):`	This function is used to record the nodes that have been expanded from the list ok, the time to exit the maze and the path of each algorithm from list trace to file and screen.	
`def uniform_cost_search():`	This algorithm uses the priority queue to store the list of pending nodes (node chờ duyệt), with the cost calculated from the start node to the current node and without using the evaluation function. This function also returns the time to exit the maze based on the number of moves and the real time system	

`def visit_best(u):`	This function is used for greedy best first search algorithm. It uses a priority queue instead of a queue like BFS. Order visit according to Manhattan distance as required	
`def greedy_best_fs():`	This function is used to implement the GBFS algorithm based on the distance calculation function Manhattan is distance (src, dst). This function also returns the time to exit the maze based on the number of moves and the real – time system	
`def a_star():`	This function is used to implement the A* algorithm, which is a combination of UCS and GBFS (during the installation mention when we use Manhattan distance and the installation). There is another new rule that is nodes that have already been opened will not be opened again.	

	This function also returns the time to exit the maze based on the number of moves and the real – time system **Notes:** A* search: it evaluates nodes by combining g(n), the cost to reach the node, and h(n), the cost to get from the node to the goal: ***f(n) = g(n) + h(n)*** with g(n): path cost from the start node to n h(n): estimated cost of the cheapest path from n to the goal. f(n): estimated cost of the cheapest solution through n.	

`def main():`	This function acts as the main function of the problem, including:
	• Call the function name which reads the input value
	• Execute functions that use algorithms to search and print the results screen
		`read_input()`
		`bfs()`
		`print()`
		
		`IDS()`
		`print()`
		
		`uniform_cost_search()`
		`print()`
		
		`a_star()`
		`print()`
		
		`greedy_best_fs()`
		`print()`

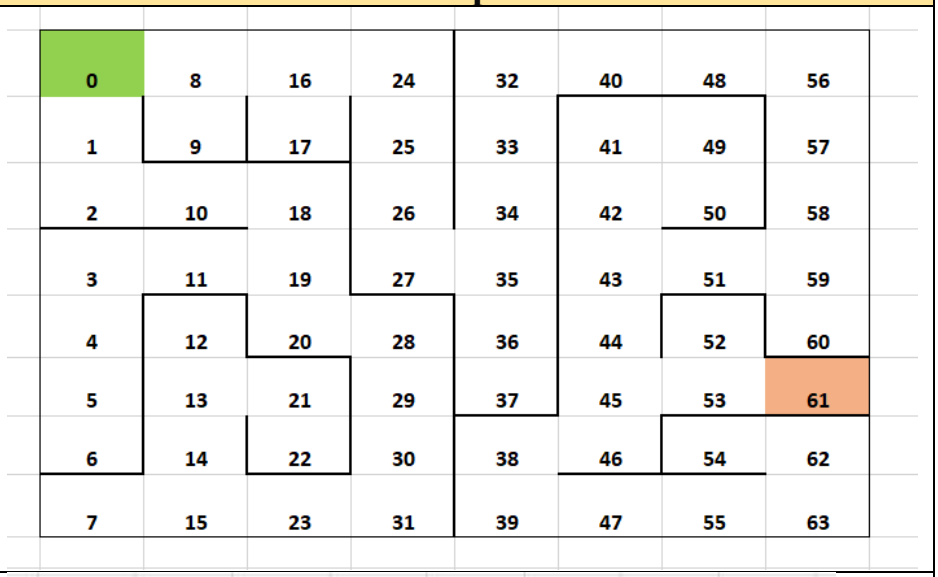
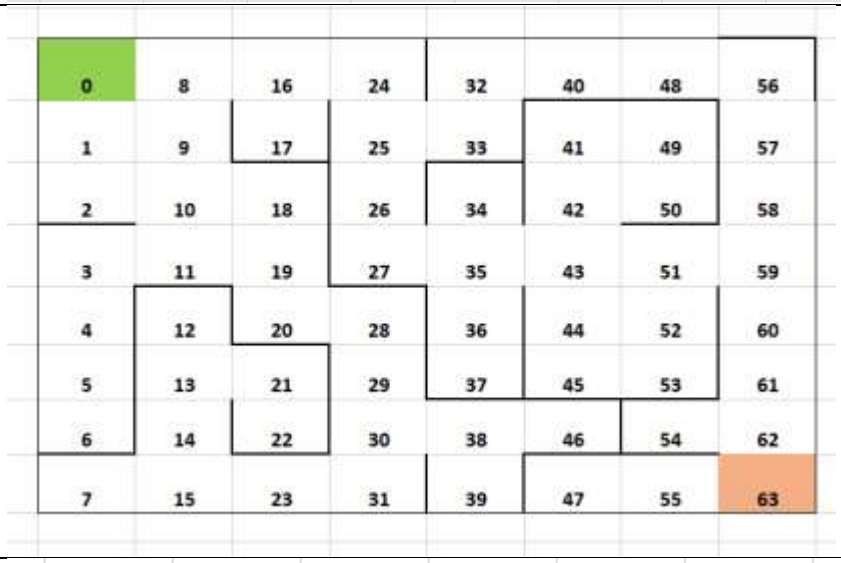
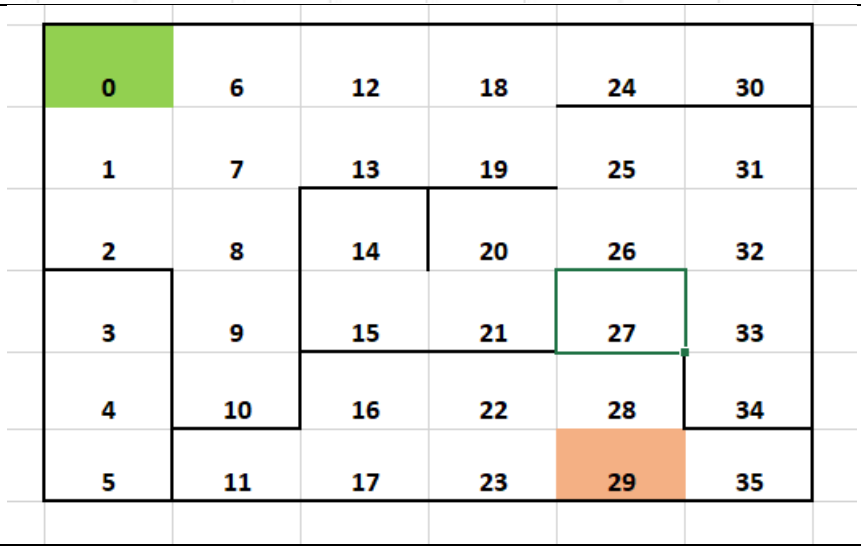
INDIVIDUAL ASSESSMENT

What I have done	Level Complete
Breadth – first search	100%
Uniform-cost search	100%
Iterative deepening search (uses DFS as a core component and avoids loops by checking a new node against the current path)	80%
Greedy-best first search	100%
Graph-search A*	100%

What I haven't done yet	Issues
IDS	List of explored nodes is not complete as required

Start node

Goal node (exist node)

Mazes	Size	Description
1	8x8	
2	8x8	
3	6x6	

Overall Level Complete: 4.5/5