MINISTRY OF EDUCATION AND TRAINING

UNIVERSITY OF SCIENCE

INFORMATION TECHNOLOGY

HO CHI MINH UNIVERSITY OF SCIENCE

CSC14003 – Introduction to Artificial Intelligence

Project 1. Search

Topic: Rush Hour Solver

Class: 23CLC01 - Group 9

Members:	Trần Thọ - 23127264
	Trần Văn Huy - 23127380
	Phan Trung Hiếu - 23127529
	Nguyễn Tuấn An - 23127316
Instructor:	Nguyen Ngoc Thao, PhD
	Nguyen Hai Dang, MSc
	Nguyen Thanh Tinh
	Nguyen Tran Duy Minh, MSc
	Ho Thi Thanh Tuyen, MSc

Ho Chi Minh City, July 13rd, 2025

CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. PLANNING AND TASK DISTRIBUTION

Table 1. Planning and Taksk distribution for each members

Tasks	Name	Completion Rate
UCS implement	Trần Thọ	100%
UCS report desciption		
Report, experiments, references, appendix		
GUI idea, background, sound effect		
A* implement	Trần Văn Huy	200%
A* Report desciption		
GUI implement		
DFS implement	Phan Trung Hiếu	100%
DFS report desciption		
Video director		
Demo		
BFS implement	Nguyễn Tuấn An	100%
BFS report description		
Map design		
Map report description		

2. REQUIREMENT

All requirements are included in requirement.txt

3. ALGORITHM DESCRIPTION

3.1. Breadth-first Search

- BFS algorithm explores all neighboring states before proceeding to the next level of depth.
- A queue (FIFO) is used to store the frontier of unexplored states, and a set of visited states is maintained to prevent redoing the visted ones.

Implementation:

- Each game board is treated as a node. The initial board state is inserted into a queue along with an empty path. At every step, a new state is generated by applying a legal move, and the path is updated accordingly.
- A visited set is maintained to avoid revisiting already explored nodes and preventing infinite loops.

- Each time a new board state is explored, the path taken to reach it is recorded. When it sees a solution, it will record the path to the solutions lists.
- We can limit the depth the algorithm can go by changing the max_depth value to avoid infinite searches.

3.2. Depth-first Search:

dfs() function performs depth-first search, taking the initial game state (initial_state) and attempting to find a solution by expanding the deepest path first dfs() function uses a stack to store the current state and path

Algorithm

1. Store the initial state and empty path into the stack

2. The stack is not empty:

- Pop this state and path from the stack.
- If state is visited, continue, otherwise, mark as visited
- If this state is goal, append path to solutions - otherwise, generate all possible next states using moves() for each next state not in visited, push it and updated path to the stack

3. return all found solutions, visited states, and depth

3.3. Uniform Cost Search

Uniform Cost Search is a graph search algorithm that finds the optimal solution by always expanding the node with the lowest path cost first. It's essentially Dijkstra's algorithm applied to search problems and guarantees finding the minimum-cost path to the goal.

Cost Function:

- The cost of moving a vehicle is equal to its length
- This makes sense for Rush Hour because longer vehicles require more "effort" to move
- Total path cost = sum of lengths of all vehicles moved

Optimality:

- UCS guarantees finding the minimum-cost solution first
- Since it always expands the lowest-cost node, the first solution found is optimal

Completeness:

- Will find a solution if one exists (assuming finite search space)
- May find multiple solutions if they exist

Algorithm

1. Initialization:

- visited: A set to track already explored states (prevents cycles)
- solutions: List to store all found solution paths
- priority_queue: A min-heap that stores tuples of (cost, path, current_state)
- The initial state starts with cost 0

2. Node Selection:

- Always pops the state with the **lowest cost** from the priority queue
- Skips already visited states to avoid infinite loops
- Marks the current state as visited

3. Goal Testing & State Expansion:

- If current state is solved, add the path to solutions
- For each possible move from current state:
    - Calculate the cost of the move (length of the vehicle that moved)
    - Add the new cost to the cumulative path cost
    - Push the new state with updated cost and path to the priority queue

3.4. A* Search:

Overview:

- A* (A-Star) is an informed search algorithm that combines the actual cost to reach a node and a heuristic estimate of the cost to reach the goal from there.
- It uses a priority queue to explore the most promising states first, ensuring both optimality and efficiency in finding a solution.

Heuristic Function:

- The custom heuristic estimates how far the red car 'X' is from escaping.
- Specifically, it counts the number of vehicles blocking the red car's path to the exit.
- This is done by checking each cell to the right of the red car's front bumper on the same row.
- The fewer the blockers, the closer we are to the solution.
- If 'X' is missing, the heuristic returns infinity to deprioritize invalid states.

Implementation Details:

- The algorithm starts by initializing:
    - A visited set to keep track of already-explored board states.
    - A solutions list to store successful paths.
    - A depth_states dictionary to record how many states were explored at each depth level.

Priority Queue:

- A priority queue (min-heap) is used, where each element is a tuple of:
    - f = g + h: total estimated cost,
    - g: the actual cost from the start to this state,
    - path: the sequence of states taken so far,
    - board: the current state of the puzzle.

Main Loop:

- While the priority queue is not empty:
    - The state with the lowest f value is removed.
    - If the state has already been visited, it's skipped.
    - Otherwise, it's marked as visited and counted in the depth histogram.

Goal Check:

- If the current board is a goal state — meaning the red car can reach the exit — the algorithm:
    - Adds the path to the solutions list,

- Returns the visited set, the solution paths, and the depth distribution.

Expanding Moves:

- For each legal move from the current board:
  - If the resulting state hasn't been visited:
    - It calculates the move cost using the length of the vehicle that moved.
    - It updates g and computes f = g + h.
    - The new state, along with its updated path and costs, is pushed into the queue.

Result Handling:

- If no solution is found after all paths are exhausted:
  - The function returns the visited states, an empty solutions list, and the depth stats for analysis or debugging.

4. MAP DESCRIPTION

- P1, P8: Moderately small maps with straightforward solution paths
- P2: This map contains many misleading or decoy paths that waste time.
- P3, P5: These maps are deep in tree nodes but ultimately unnecessary branches.
- P4: This is a relatively easy map and well-balanced in terms of widtth and depth of the tree nodes
- P6: This map has unnecessary nodes and costly paths
- P7: A small map with a very limited solution space.
- P9: One of the most expansive maps, it has massive solutions and misleading paths. Not hard in terms of difficulty.
- P10: Have many solutions and dead-end paths but not as many as P9
- P11: This map includes many misdirections and costly paths.
- P12: A complex map with longer solution paths.
- P13: This is a narrow-path puzzle where depth-based methods can easily explore.
- P14: A high difficulty map with high complexity.
- P15, P16: Claimed to be one of the hardest map exist, requiring the most moves (by this research)
- P17, P18, P19, P20: A complication of dead-end maps, with fast termination, minimal memory use, and low node visits except P19 with it having many promising paths to clear the puzzle but doesn't have the solution.

5. EXPERIMENTS

5.1. Search time

Table 2. Time usage of each algorithm with 20 maps

TIME USAGE (S)					
	BFS (max depth = 100)	DFS	DLS (max depth = 100)	UCS	A*
P1	0.2087	0.2186	11.3167	0.7504	0.745
P2	3.0928	3.2447	225.1218	19.7858	0.1365
P3	0.5444	0.5529	41.5331	2.6665	2.2304
P4	0.1051	0.1002	5.9308	0.3422	0.3172
P5	0.5212	0.5509	38.6853	2.3699	2.2422
P6	0.5956	0.614	43.8216	2.7078	1.3301
P7	0.1065	0.1026	7.6601	0.3486	0.3213
P8	0.1987	0.2065	11.7827	0.7803	0.7513
P9	5.859	14.3753	406.025	40.7852	4.5077
P10	3.7099	1.181	249.1539	23.4411	2.078
P11	1.1176	5.6772	85.2392	6.4833	0.6188
P12	1.5495	2.1183	126.1486	9.3705	4.7106
P13	0.2951	0.2877	21.3269	1.4216	0.256
P14	1.294	1.3797	44.2538	5.9603	5.1012
P15	1.3961	1.5332	69.9264	6.2073	4.5718
P16	1.0751	1.1237	68.4849	5.3421	4.4293
P17	0.0024	0.0022	0.0044	0.0054	0.0043
P18	0.0011	0.0001	0.0001	0.0001	0.0001
P19	0.1268	0.1348	11.0169	0.5467	0.4821
P20	0.001	0.001	0.0015	0.001	0.0015

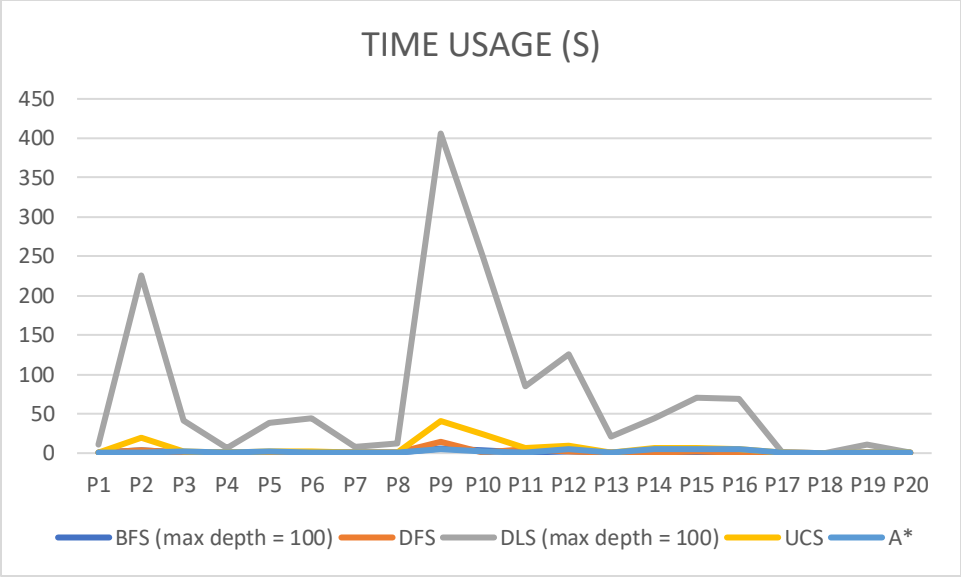

Figure 1. Search time of each algorithms

- BFS (max depth = 100): Time usage is generally moderate, but can be high for complex problems (e.g., P9: 5.859 S, P10: 3.7099 S).
- DFS: Similar to memory, DFS often has high time usage, especially for problems where it explores deep, unproductive paths (e.g., P9: 14.3753 S, P11: 5.6772 S).
- DLS (max depth = 100): Exhibits the highest time usage among all algorithms for most problems, often by a large margin (e.g., P2: 225.1218 S, P9: 406.025 S, P10: 249.1539 S). This reinforces the observation that the depth limit of 100 is often too large, leading to extensive exploration.
- UCS: Generally performs better than BFS, DFS, and DLS in terms of time usage, but is not as fast as A* (e.g., P9: 40.7852 S, P10: 23.4411 S).
- A*: Consistently shows the fastest search times for almost all problems, often by a significant margin (e.g., P2: 0.1365 S, P9: 4.5077 S, P10: 2.078 S). This highlights the efficiency gained from using a heuristic.

5.2. Memory usage

Table 3. Memory usage of each algorithms with 20 maps

MEMORY USAGE (KB)					
	BFS (max depth = 100)	DFS	DLS (max depth = 100)	UCS	A*
P1	903.38	4130.55	2622.37	725.23	733.7
P2	15033.02	28660.98	3908.09	10274.58	814.9
P3	2473.81	15769.42	210878.88	1913.78	1917.23
P4	479.67	1331.69	1486.96	374.6	370.95

P5	2474.8	15769.42	3908.09	1910.52	1929.12
P6	2625.34	15381.25	25511.24	2141.96	1176.34
P7	479.46	1642.02	1369.78	375.28	370.16
P8	903.38	4130.73	2622.64	730.38	733.8
P9	22319.8	579155.53	272483.24	17779.92	4823.7
P10	13170.92	40897.95	236025.15	11697.74	2762.45
P11	4761.77	268519.63	53486.01	3790.99	1042.49
P12	6073.34	61767.86	22142.73	4291.2	4180.74
P13	1506.89	3560.24	20920.9	1175.7	583.09
P14	4207.09	42893.48	6125.11	3709.25	3707.43
P15	5209.38	33323.8	8802.04	4841.03	4412.27
P16	3365.62	21852.13	5846.92	3133.33	3144.27
P17	22.05	21.53	20.87	24.41	19.03
P18	6.45	4.8	5.26	5.27	5.49
P19	822.38	2566.59	736.45	553.97	543.56
P20	12.91	13.21	12.81	11.25	11.59

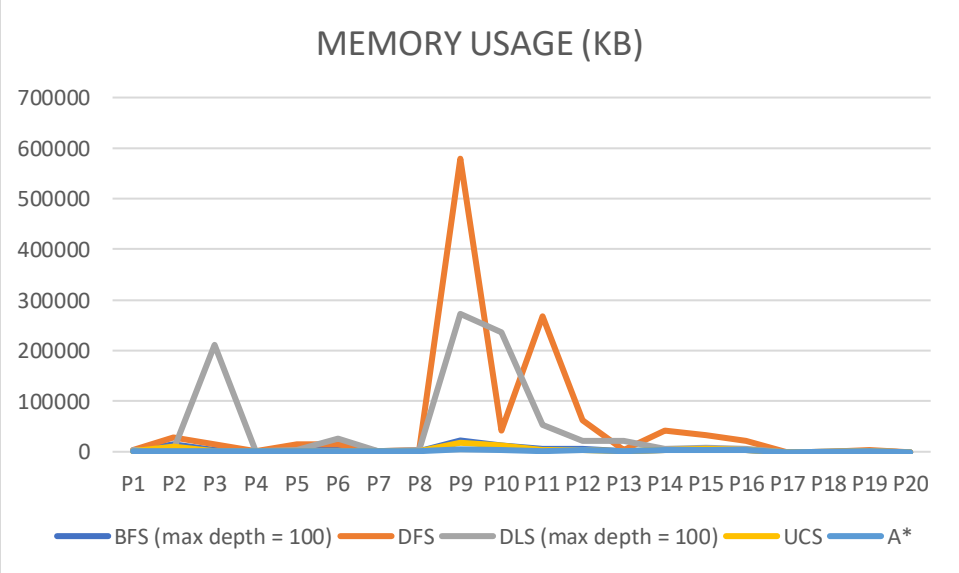

Figure 2. Memory usage of each algorithms

- BFS (max depth = 100): Generally has moderate memory usage, but can be high for some problems (e.g., P2: 15033.02 KB, P9: 22319.8 KB).
- DFS: Shows very high memory usage for many problems, often significantly higher than other algorithms (e.g., P2: 28660.98 KB, P9: 579155.53 KB, P11: 268519.63 KB). This is expected due to its depth-first nature exploring deep paths.
- DLS (max depth = 100): Memory usage varies widely. For some problems, it's relatively low (e.g., P1: 2622.37 KB), but for others, it can be extremely high, even

exceeding DFS in some cases (e.g., P3: 210878.88 KB, P9: 272483.24 KB, P10: 236025.15 KB). This indicates that the depth limit of 100 might be too large for some problems, causing it to explore a large state space.

- UCS: Generally has lower memory usage compared to BFS, DFS, and DLS for most problems (e.g., P1: 725.23 KB, P9: 17779.92 KB).
- A*: Consistently demonstrates the lowest peak memory usage among all algorithms for almost all problems (e.g., P1: 733.7 KB, P9: 4823.7 KB). This is a significant advantage of A* due to its informed search.

5.3. Node visited

Table 4. Number of visited nodes of each algorithm with 20 maps

NODES VISITED					
	BFS (max depth = 100)	DFS	DLS (max depth = 100)	UCS	A*
P1	838	838	838	838	824
P2	9139	8926	12544	8446	106
P3	1646	1646	1646	1646	1587
P4	457	457	457	457	428
P5	1646	1646	1646	1646	1586
P6	1834	1834	1834	1834	982
P7	457	457	457	457	429
P8	838	838	838	838	824
P9	15784	15784	15784	15784	2484
P10	10934	10934	10934	10934	1430
P11	3382	3382	3382	3382	520
P12	4116	4116	4116	4116	2336
P13	1084	1084	1084	1084	346
P14	3539	3540	3540	3540	3074
P15	4643	4643	4643	4643	3898
P16	2968	2968	2968	2968	2958
P17	13	12	12	16	12
P18	3	3	3	3	3
P19	488	488	488	488	488
P20	8	8	8	8	8

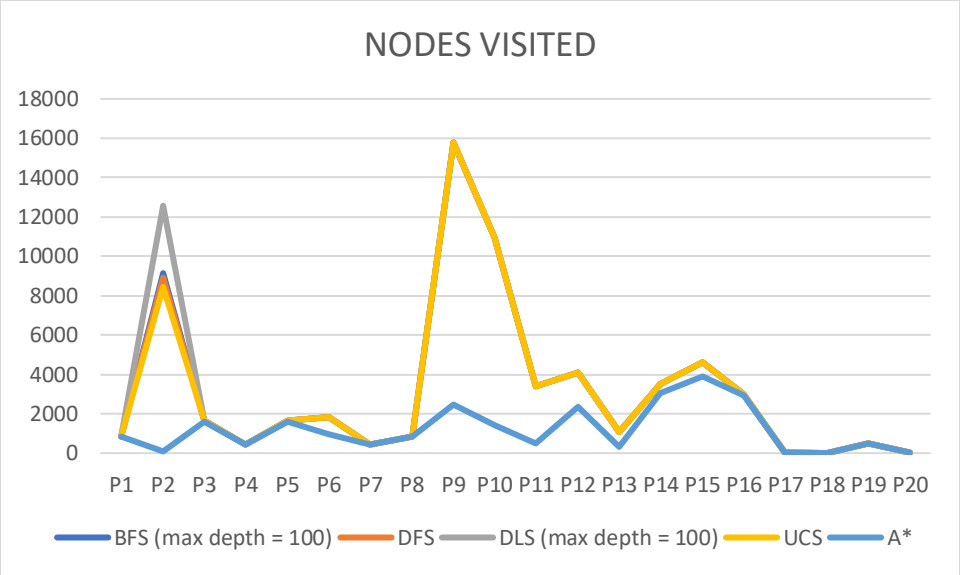

Figure 3. Number of visited node of each algorithms

- BFS (max depth = 100): Visits a considerable number of nodes, especially for larger problems (e.g., P2: 9139, P9: 15784).
- DFS: Similar to BFS, it visits a large number of nodes, often comparable to BFS or slightly higher (e.g., P2: 8926, P9: 15784).
- DLS (max depth = 100): Visits a very high number of nodes, often the highest among all algorithms, indicating extensive exploration within the depth limit (e.g., P2: 12544, P9: 15784).
- UCS: Visits a similar number of nodes to BFS and DFS for many problems (e.g., P2: 8446, P9: 15784).
- A*: Consistently visits the fewest nodes across almost all problems (e.g., P2: 106, P9: 2484). This is a direct consequence of its informed search, which prunes the search space effectively.

Overall:

- A* appears to demonstrate superior efficiency in terms of both time efficiency (fastest search times) and space efficiency (lowest memory usage and fewest nodes visited). Its use of a heuristic allows it to explore the search space much more intelligently.
- DLS (with max depth = 100) performs poorly in both time and memory for many problems, suggesting that a fixed, large depth limit can be detrimental if the optimal solution is found at a much shallower depth or if the search space within that depth is vast.

- BFS, DFS, and UCS generally perform similarly in terms of nodes visited, but their time and memory usage can vary. DFS can be particularly memory-intensive. UCS, while better than BFS and DFS in some cases, doesn't match the performance of A*.

6. VIDEO DEMO

Demo video link

REFERENCES

[1] P. Community, "Pygame Tutorial," [Online]. Available: https://www.pygame.org/.

[2] GeeksforGeeks, "Breadth First Search (BFS) for Artificial Intelligence," [Online]. Available: https://www.geeksforgeeks.org/artificial-intelligence/breadth-first-search-bfs-for-artificial-intelligence/. [Accessed 23 6 2025].

[3] GeeksforGeeks, "Depth First Search (DFS) for Artificial Intelligence," [Online]. Available: https://www.geeksforgeeks.org/artificial-intelligence/depth-first-search-dfs-for-artificial-intelligence/.

[4] GeeksforGeeks, "Depth Limited Search for AI," [Online]. Available: https://www.geeksforgeeks.org/artificial-intelligence/depth-limited-search-for-ai/.

[5] GeeksforGeeks, "Uniform Cost Search (UCS) in AI," [Online]. Available: https://www.geeksforgeeks.org/artificial-intelligence/uniform-cost-search-ucs-in-ai/.

[6] R. Wilson-Perkin, "Github," 2015. [Online]. Available: https://github.com/ryanwilsonperkin/rushhour. [Accessed 5 7 2025].

[7] GeeksforGeeks, "A* Search Algorithm in Artificial Intelligent," [Online]. Available: https://www.geeksforgeeks.org/dsa/a-search-algorithm/.