Title: Performance analysis of optimized A-Star and iterative deepening A-Star algorithm with distance and custom heuristics.

Author: Zakwan Ashfaq Zian

Honours in Computer Science

Department of Computer Science

Memorial University of Newfoundland

14 April 2023

A dissertation submitted to the Department of Computer Science in partial fulfilment of the requirements for the degree of Bachelor of Science (Honours.)

Abstract:

This study presents an examination of the A* and Iterative Deepening A* (IDA*) search algorithms, emphasizing their optimizations, performance, and memory efficiency when solving rush-hour game problems. A comparative analysis of A* and IDA* using distance and custom heuristics revealed that while A* consistently outperformed IDA* in terms of speed, IDA* proved more memory-efficient. Crucial to the performance optimization of A* is the custom sorting of priority queues, the efficient management of open and closed lists, and the use of admissible distance and custom heuristics. The study further explores a method of encoding states into unique strings for rapid lookup and reduced memory usage, improving performance. There is also a demonstration of why the first occurrence of a unique state in A* ensures the shortest path to that state due to depth-optimized priority queue. All these insights help us to understand the trade-offs between computational speed and memory usage in these algorithms, ultimately informing the choice of algorithm based on the specific problem and available resources.

Acknowledgements:

This exploration was inspired by similar experiments by Michael Fogleman and Dr. Dave Churchill.

Table of Contents

Abstract …….. 2

Acknowledgements …… 2

Table of Contents ……... 3

List of Tables …….. 6

List of Figures …… 7

Chapter 1: Introduction ……... 8

Chapter 2: Background ……... 10

2.1 Rush Hour Game:

- 2.1.1 Description of gameplay ……... 10
- 2.1.2. Properties of Rush Hour gameplay …….. 10

2.2 Algorithms:

- 2.2.1 Description of A* algorithm …… 11
- 2.2.2 Description of IDA* algorithm ……. 12

Chapter 3: Methodology …….. 14

3.1 Rush Hour C++ application …….. 14

3.2 Encoding states to strings for fast lookups and low memory usage ……. 16

3.3 Optimizations made to A*:

- 3.3.1 Priority queue sorting for A* and node selection …….. 17
- 3.3.2 Closed and open list optimizations …… 18

3.4 Heuristic:

- 3.4.1 Distance heuristic …….. 19
- 3.4.2 Custom heuristic …… 20

3.5 Briefs on tested combinations :

- 3.5.1 Solving with A* and Distance heuristic …… 21
- 3.5.2 Solving with IDA* and Distance heuristic …… 21
- 3.5.3 Solving with A* and Custom heuristic …….. 21
- 3.5.4 Solving with IDA* and Custom heuristic ……. 22

Chapter 4: Results and Discussion …….. 23

4.1 Run time comparisons:

- 4.1.1 A* vs IDA* with distance heuristics ……. 23
- 4.1.2 A* vs IDA* with custom heuristics ……. 24

4.2 Analysing run time performance of A* vs IDA* with distance and custom heuristics ……. 25

4.3 Number of nodes searched comparisons:

- 4.3.1 A* vs IDA* with distance heuristics ……. 26
- 4.3.2 A* vs IDA* with custom heuristics …….. 27

4.4 Analysing number of nodes searched for A* and IDA* with distance and custom heuristics …… 27

4.5 Proof on why the first occurance of a unique state is the shortest path to that state in the optimized A* approach ……. 29

Chapter 5: Conclusion ……. 31

5.1 The trade-offs between memory usage and computational efficiency in A* and IDA* algorithm …….. 31

5.2 Which algorithm is best suited for this problem …….. 32

5.3 Further work:

- 5.3.1 Explore training a ML model and benchmark against A* and IDA* ….. 32
- 5.3.2 Exploring better custom heuristic functions …….. 33
- 5.3.3 Exploring more IDA* optimizations ……. 34

Bibliography ……... 35

List of tables

Run times of A* vs IDA* with distance heuristics ……... 23

Run times of A* vs IDA* with custom heuristics …… 24

Number of nodes searched of A* vs IDA* with distance heuristics ……. 26

Number of nodes searched of A* vs IDA* with custom heuristics ……... 27

List of figures

Pseudo-code for A* algorithm ……..... 12

Pseudo-code for IDA* algorithm ……... 13

Example of a puzzle defined in the “input.txt” file …….…... 14

Printed output of the grid-state ……... 16

Manhattan and Euclidean distance visualization …….……... 19

Iterative deepening steps of the Rush Hour Puzzle Solver ……... 30

Chapter 1:
Introduction

This study is being conducted to investigate the performance of two popular search algorithms, A* and IDA*, by comparing their run times and the number of nodes searched. The objective here is to determine which algorithm is more efficient in solving the complex rush hour puzzles. Rush hour puzzles involve navigating a congested grid of vehicles to free a specific target vehicle, and they are known to have a vast state space(Michael Fogleman) due to the numerous possible configurations of vehicles and their potential movements.

By comparing the run times and the number of nodes searched, I will find out the effectiveness and efficiency of A* and IDA* in exploring the state space of rush hour puzzles. A* is a heuristic search algorithm that uses an admissible heuristic function to estimate the cost of reaching the goal from each explored node. On the other hand, IDA* is an iterative version of A* that aims to reduce memory usage by performing a depth-first search with increasing depth limits.

Understanding which algorithm performs better in terms of run times and the number of nodes searched is crucial for improving rush hour puzzle-solving techniques. By identifying the strengths and weaknesses of A* and IDA*, I can develop more efficient and effective search algorithms or optimize this one further. Additionally, the study of rush hour puzzles with their

immense state space provides an excellent start for exploring various search strategies and heuristics, which can be applied to other puzzle problems as well.

This research serves as a valuable starting point for further studies in the field of search algorithms, puzzle-solving and artificial intelligence. It not only contributes to the understanding of the comparative performance of A* and IDA* in rush hour puzzles but also lays the foundation for future advancements in algorithmic approaches for solving complex problems. By building upon the findings of this study, I plan to delve deeper into improving search algorithms, refining heuristics, and exploring novel techniques that can have broader applications beyond rush hour puzzles.

Chapter 2:
Background

2.1 Rush Hour Game

2.1.1 Description of gameplay

Rush Hour is a sliding block puzzle invented by Nob Yoshigahara in the 1970s. It challenges the player to create a clear path for a player's car(typically red car) to exit a grid full of other vehicles. The player can move the vehicles horizontally or vertically, depending on their orientation, but they cannot lift them off the grid or rotate them. The vehicles can only move within the boundaries of the grid and they cannot overlap with each other. Vehicles can either be cars (which occupy two grid cells) or trucks (which occupy three grid cells). The goal is to move the red car to the exit area on the right side of the grid with as few moves as possible.

2.1.2 Properties of Rush Hour gameplay

Rush Hour game is characterized by the following properties:

- Puzzle: The game is a sliding block puzzle, requiring strategic thinking to move the vehicles in such a way that a clear path can be created for the target vehicle to exit.
- Deterministic: There's no element of chance or randomness in the game. The starting configuration of vehicles determines the potential solutions to the puzzle.
- Fully Observable: The entire game state is visible to the player. There's no hidden information, and the player can see all the cars and trucks on the grid.

- Single Agent Environment: The game involves only one player, and the 'opponent' is the game itself or, more specifically, the initial configuration of vehicles.
- Discrete and Finite State Space: The game is played in turns, with each move changing the game state discreetly. Additionally, the game has a finite number of possible configurations(Michael Fogleman), given the fixed grid size and the limited number of vehicles.

2.2 Algorithms

2.2.1 Description of A* algorithm

The A* algorithm is a widely used pathfinding and graph traversal algorithm, which is an instance of the best-first search algorithm. The A* algorithm introduces a heuristic into the standard Dijkstra's algorithm(a generalized form of BFS) to guide the pathfinding process by providing an estimate of the cost to reach the goal from a particular node.

The algorithm maintains a priority queue of nodes, where each node has a cost associated with it equal to `f(n) = g(n) + h(n)`. Here, `g(n)` is the actual cost from the start node to the current node `n`, and `h(n)` is the heuristic estimated cost from node `n` to the goal node. The heuristic function `h(n)` is problem-specific and should never overestimate the cost to reach the goal for the algorithm to guarantee the shortest path.

At each step, the A* algorithm selects the node with the smallest `f(n)` value, explores all its neighbours, updates their costs, and adds them to the queue. This process continues until the goal node is reached or if there are no more nodes to explore.

```
1  Put node_start in the OPEN list with f(node_start) = h(node_start) (initialization)
2  while the OPEN list is not empty {
3    Take from the open list the node node_current with the lowest
4         f(node_current) = g(node_current) + h(node_current)
5    if node_current is node_goal we have found the solution; break
6    Generate each state node_successor that come after node_current
7    for each node_successor of node_current {
8      Set successor_current_cost = g(node_current) + w(node_current, node_successor)
9      if node_successor is in the OPEN list {
10       if g(node_successor) ≤ successor_current_cost continue (to line 20)
11     } else if node_successor is in the CLOSED list {
12       if g(node_successor) ≤ successor_current_cost continue (to line 20)
13       Move node_successor from the CLOSED list to the OPEN list
14     } else {
15       Add node_successor to the OPEN list
16       Set h(node_successor) to be the heuristic distance to node_goal
17    }
18     Set g(node_successor) = successor_current_cost
19     Set the parent of node_successor to node_current
20   }
21   Add node_current to the CLOSED list
22 }
23 if(node_current != node_goal) exit with error (the OPEN list is empty)
```

Figure: Pseudo-code for A* algorithm

2.2.2 Description of IDA* algorithm

Iterative-Deepening A* (IDA*) is an algorithm used in the field of artificial intelligence for traversing graphs and tree structures. It is a variant of the A* search algorithm and combines the strengths of iterative deepening depth-first search and the A* algorithm.

IDA* uses a depth-first search to traverse the graph, but it also uses a bound on the cost of the path it is currently exploring. The bound for the search is the estimated cost `f(n) = g(n) + h(n)` of a node `n`, as used in the A*.

Algorithm

```
root=initial node;
Goal=final node;
function IDA*()                                  //Driver function
{
      threshold=heuristic(Start);
      while(1)          //run for infinity
      {
            integer temp=search(Start,0,threshold); //function search(node,g score,threshold)
            if(temp==FOUND)                        //if goal found
                  return FOUND;
            if(temp== ∞)                     //Threshold larger than maximum possible f value
                  return;                    //or set Time limit exceeded
            threshold=temp;
      }
}
function Search(node, g, threshold)          //recursive function
{
      f=g+heuristic(node);
      if(f>threshold)          //greater f encountered
            return f;
      if(node==Goal)           //Goal node found
            return FOUND;
      integer min=MAX_INT;    //min= Minimum integer
      foreach(tempnode in nextnodes(node))
      {
            //recursive call with next node as current node for depth search
            integer temp=search(tempnode,g+cost(node,tempnode),threshold);
            if(temp==FOUND)          //if goal found
            return FOUND;
            if(temp<min)    //find the minimum of all 'f' greater than threshold encountered
            min=temp;
      }
      return min;  //return the minimum 'f' encountered greater than threshold
}
function nextnodes(node)
{
        return list of all possible next nodes from node;
}
```

Figure: Pseudo-code for IDA* algorithm

Chapter 3: Methodology

3.1 Rush Hour C++ application

This Rush Hour C++ application is a well-structured program designed to solve the sliding block puzzle of the Rush Hour game. It starts by reading the puzzle configuration from an input file that provides the grid size, the goal position, the player's size and location, as well as the sizes, orientations, and locations of pawns on the grid. This input file uses a clear, easy to understand format that describes every game piece concisely, which aids in the accurate initialization of the game state.

Grid 7 7 (grid size width, grid size height)
Goal 5 2 (goal location x, goal location y)
Player 2 2 2 (player size, then player x location, player y location)
Pawn 3 HORIZONTAL 2 1 (pawn size, orientation, x coordinate, y coordinate)
Pawn 3 VERTICAL 5 1
Pawn 3 VERTICAL 3 4
Pawn 2 HORIZONTAL 4 4
Pawn 2 VERTICAL 6 4
Pawn 3 HORIZONTAL 4 6

Figure: Example of a puzzle defined in the “input.txt” file

In terms of its architecture, the application comprises several essential components. Two core classes, "aStar" and "IDAStar", handle the implementation of the A* and IDA* algorithms, respectively. The "IDAStar" class is a subclass of "aStar", exhibiting the principle of inheritance by modifying the A* class to suit the IDA* algorithm. Both classes receive the filename of the input file in a string format.

The application also includes "structs.cpp", which encapsulates all the enums and structs used throughout the codebase. This file is crucial for organizing the game's data structures and ensuring that all game elements interact appropriately within the software's logic. Makes the codebase easy to read and modify.

Another valuable component of the Rush Hour C++ application is "utils.cpp", which houses all functional elements of the codebase that can be easily tested and universally utilized. These utilities promote loose coupling within the system, making the application more adaptable and easily modifiable, thereby facilitating enhancements and adjustments to the game's logic and performance. This approach contributes to the overall maintainability and scalability of the application, enhancing its long-term value and usefulness. This is crucial as I will be working further on this project as a platform to learn more and experiment to expand my knowledge even further.

3.2 Encoding states to strings for fast lookups and low memory usage

Encoding the grids in nodes to a unique string value and storing them in a hashmap, makes it a lot faster to check if a similar state already exists. If a node exists in the hashmap, ignore the node, but, if it does not include it in the open_list. The node is being encoded by the positions, orientation, size and ids of the pawns. For example let us take the node below to convert to a unique string.

0 0 0 0 0 0 0

2 2 2 0 0 0 0

0 0 0 0 0 1 1

0 0 0 0 0 1 1

5 5 0 4 0 3 6

0 0 0 4 0 3 6

7 7 7 4 0 3 0

Figure: Printed output of the grid-state

To encode this node's grid to a string we first declare a mutable string variable. Then we look at the player pawn which has an id of 1 and add "player:5-4$" to the mutable string variable. Here 5 indicates the x-coordinate of the node and 4 indicates the y-coordinates of the node. The "$" sign is used to separate pawns in the string. After we add the player encoding to the string we loop through all the pawns and add their encoding to the string. The format for the pawns is "p(2,0):0-1$". Here "p" represents a pawn, the number "2" represents the id of that

pawn and “0” represents the orientation. The id can be any real number but the orientation can only be 0 or 1 to represent horizontal and vertical. The next two numbers in “0-1$” indicate the x and y location of the pawn and the “$” sign separates the pawn encodings to be more readable for debugging. The final encoding will be this for the above grid figure:

"player:5-2$p(2, 1):0-1$p(3, 0):5-4$p(4, 0):3-4$p(5, 1):0-4$p(6, 0):6-4$p(7, 1):0-6$"

This combination creates a string that has a one to one relation with a given grid-state, this is it is unique to the given grid state. If another string matches it the grid would look like the above figure and we can safely ignore that state. By this technique we do not have to loop through the entire open and closed list to check for unique states and we do not have to store the entire states themselves, saving us memory.

3.3 Optimizations made to A*

3.3.1 Priority queue sorting for A* and node selection

The custom sorting of the priority queue in this code is essential for the node selection strategies used in this A* algorithm. Without this, the search for a solution in the state space would be significantly slower, as the search state space is quite large. The priority queue is sorted based on two criteria: the cost to reach the state and the state evaluation value. The cost is the number of steps taken to reach a particular node, and the state evaluation value is calculated by the `evaluateState` or `evaluateStateCustom` function, which estimates how close the player is to the goal state.

In the custom sorting function, nodes with higher costs are ranked lower. If the costs are equal, the node with the higher state evaluation value is ranked higher. For example, given two nodes with costs of 5 and 7 respectively, the node with cost 5 will be given more priority. If there are

two nodes with the same cost of 5 but with state evaluation values of 1200 and 900 respectively, the node with a state evaluation value of 1200 is prioritized, as it is closer to the goal state. This ranking system ensures that nodes are prioritized based on their proximity to the goal, allowing for more efficient and effective node selection in the A* algorithm. By sorting the priority queue based on both cost and state evaluation value, the A* algorithm can quickly identify the most promising nodes to explore, thus efficiently traversing the state space.

3.3.2 Closed and open list optimizations

A typical A* algorithm maintains two lists: the open list and the closed list. The open list stores the nodes that need to be evaluated, while the closed list contains nodes that have already been evaluated. In this A* algorithm, to maximize the performance of looking through the openlist and closed list, two hashes are used along with a priority queue (the openlist). The hashes allows for incredibly fast lookup to check if a given state is present in either of the open or closed list, and reduces the time complexity of the algorithm by a huge margin.

Instead of looping through all the nodes in the closed list to check if a given state exists, now it is a matter of a quick hash encoding lookup (encoding explained later in this document). So instead of looking and comparing thousands of nodes in each iteration, and there might be thousands of iterations as well, we are doing a single check. The time taken to execute for this block went down from taking around 10,000 microseconds to 10-14 microseconds on average. Similar performance gains were seen for closed list lookups as well.

We still maintain another openlist in the form of a priority queue, but it does not affect the overall performance for the lookups as it is only used when selecting the next node to expand.

3.4 Heuristic

3.4.1 Distance heuristic

The function `evaluateState` within the `aStar` class computes the distance from the player's current position to the goal. It accepts a shared pointer to a `stateNode` object as its argument, from which it extracts the player's coordinates. The function then calculates the Manhattan distance, which is the sum of the absolute differences of the x and y coordinates between the player's position and the goal. This effectively measures the total number of steps needed to reach from the player's position to the goal if only horizontal and vertical movements are allowed. The Manhattan distance used in the function `evaluateState` is considered an admissible heuristic because it never overestimates the cost of moving from the current node (the player's position) to the goal.

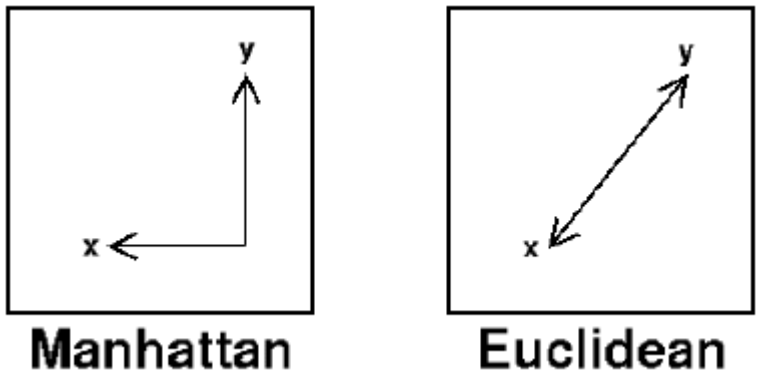

Figure: Manhattan and Euclidean distance visualization.

3.4.2 Custom heuristic

The function `evaluateStateCustom` from the `aStar` class computes the evaluation score for a given game state, encapsulated in a `stateNode` object. It starts by extracting the player's coordinates from the node and then calculates the euclidean distance (pythagoras) between the player's current position and the goal. The function computes an initial score by dividing a constant, `MAX_EVAL_VALUE`, by the distance. The score is inversely proportional to the distance, therefore, states where the player is closer to the goal will receive a higher initial score than states further away from the goal location. Afterwards, the function adjusts the score by deducting a value that is the product of the node's cost and a depth factor, `DEPTH_FACTOR`. This penalizes the score for nodes that have a higher cost or are deeper in the search tree. Finally, the function returns the resulting score. Both `MAX_EVAL_VALUE` and `DEPTH_FACTOR` are integer constants that can be altered to explore performance improvements.

The Manhattan distance component of the heuristic is admissible, as it never overestimates the cost of reaching the goal as described earlier for the distance heuristic function. The second part of the heuristic is the cost of the node modified by two constant factors(`MAX_EVAL_VALUE` and `DEPTH_FACTOR`). The node's cost, factored by `DEPTH_FACTOR`, and subtracted from the score, represents the real cost incurred to reach the current node. As this value represents an exact cost rather than an estimate of future cost, it doesn't infringe on admissibility. The `MAX_EVAL_VALUE` is a constant that scales the score based on the Manhattan distance, enhancing differentiation between states. As it's

divided by the distance (which is itself an admissible heuristic), it doesn't lead to an overestimation of the cost, preserving the overall admissibility of the heuristic.

3.5 Briefs on tested combinations :

3.5.1 Solving with A* and Distance heuristic

This combination should be most effective in problem-solving. As the A* algorithm effectively navigates through the state space and the Distance heuristic (Manhattan distance) provides a clear, admissible estimate of cost from the current node to the goal. This method prioritized nodes with the least cost and closest proximity to the goal, resulting in low run times and minimal total number of nodes searched.

3.5.2 Solving with IDA* and Distance heuristic

IDA* (Iterative-Deepening A*) with the Distance heuristic should perform similarly. The primary difference would be that IDA* employs a depth-first search that gradually increases its depth limit and has to re-search the entire tree after crossing each threshold. Nothing is stored in R.A.M., as we do not keep a record of visited nodes. This should make the search more memory efficient than A*, while the Distance heuristic should still provide a reliable cost estimate for each node.

3.5.3 Solving with A* and Custom heuristic

The combination of A* and the Custom heuristic could introduce an additional layer of effectiveness in problem-solving scenarios. A* navigates efficiently through the state space, and the Custom heuristic provides a nuanced estimation of the cost from the current node to

the goal, taking into account both distance and node cost. This method should result in prioritizing nodes that balance the considerations of cost and proximity to the goal. Consequently, this setup could lead to a greater level of efficiency compared to using the Distance heuristic, potentially reducing run times and the total number of nodes searched.

3.5.4 Solving with IDA* and Custom heuristic

Employing the Custom heuristic with the IDA* algorithm might further optimize the depth-first search approach. The Custom heuristic's careful evaluation of each node's cost and proximity to the goal could significantly enhance the accuracy of the IDA* search process. While still maintaining memory efficiency due to not storing previously visited nodes, this combination could potentially increase the performance of IDA* in certain scenarios by further reducing the total number of nodes searched. However, it should be noted that since IDA* re-searches the entire tree after crossing each threshold, it may still require considerable run times depending on the problem set.

Chapter 4:
Results and Discussion

4.1 Run time comparisons:

To measure the run-time performance of A* and iterative deepening A* algorithm, I ran the same puzzle with each of these algorithms on the same computer to limit outside factors on my results. To validate my findings and check for anomalies, I ran experiments three times using different puzzles each time.

4.1.1 A* vs IDA* with distance heuristics

	A* Search with distance heuristic	IDA* Search with distance heuristic
	Format: RUN 1/ RUN 2/ RUN 3 in milliseconds	
Problem 1 (J2)	602/629/596	697/688/733
Problem 2 (J1)	12002/9578/10120	13739/15600/14539

Problem 3 (J10)	3201/3223/3230	143122/145762/144188

Figure: Run times of A* vs IDA* with distance heuristics

4.1.2 A* vs IDA* with custom heuristics

	A* Search with custom heuristic	IDA* Search with custom heuristic
	Format: RUN 1/ RUN 2/ RUN 3 in milliseconds	
Problem 1 (J2)	1366/1331/1319	25729/25469/25383
Problem 2 (J1)	25064/25817/26504	101442/101062/101062
Problem 3 (J10)	5797/5941/6972	261518/259549/253672

Figure: Run times of A* vs IDA* with custom heuristics

4.2 Analysing run time performance of A* vs IDA* with distance and custom heuristics

Upon analyzing the A* and Iterative Deepening A* (IDA*) algorithms performance on three problems using both distance and custom heuristics, it's very clear that the algorithms perform differently depending on the heuristic applied and the complexity of the problem. When using a distance heuristic, A* algorithm consistently outperforms IDA* in terms of run time across all three configurations and problems. This implies that A* is able to use the distance heuristic to effectively reduce the state search space, thus decreasing the time complexity.

On the other hand, when a custom heuristic is applied, the time taken to solve for A* increases for all three problems, yet still remains faster than IDA*. This suggests that the custom heuristic is not as effective as the simple distance heuristic at pruning the state search space.

For IDA*, run times are substantially higher than A* regardless of the heuristic used. This might be due to IDA* having to iterate multiple times over the same nodes/states, especially when the threshold needs to be increased for more complex problems. As the threshold gets increased IDA* have to recheck all the nodes it checked from the very beginning and explore more options from there.

In conclusion, this data suggests that while the choice of heuristic can affect the performance of both A* and IDA*, A* tends to be more efficient in terms of run time for this specific problem.

4.3 Number of nodes searched comparisons:

To measure the number of nodes searched for A* and iterative deepening A* algorithm, I ran the same puzzles that I used for the run-time comparisons.

4.3.1 A* vs IDA* with distance heuristics

	A* Search with distance heuristic	IDA* Search with distance heuristic
Problem 1 (J2)	493	1761
Problem 2 (J1)	6627	13463
Problem 3 (J10)	2123	323482

Figure: Number of nodes searched of A* vs IDA* with distance heuristics

4.3.2 A* vs IDA* with custom heuristics

	A* Search with custom heuristic	IDA* Search with custom heuristic
Problem 1 (J2)	493	1748
Problem 2 (J1)	6627	13428
Problem 3 (J10)	2123	323476

Figure: Number of nodes searched of A* vs IDA* with custom heuristics

4.4 Analysing number of nodes searched for A* and IDA* with distance and custom heuristics

The analysis of the number of nodes searched using different combinations of search algorithms and heuristics shows interesting trends. For A* Search, the number of nodes searched is identical whether the Distance or Custom heuristic is used. Specifically, for Problem 1, the algorithm searched through 493 nodes; for Problem 2, 6627 nodes were investigated, and for Problem 3, a total of 2123 nodes. This suggests that the choice of heuristic does not impact the breadth of the search using the A* algorithm.

In contrast, the IDA* Search shows a minor difference in the number of nodes searched when comparing the use of Distance versus Custom heuristic. For Problem 1, IDA* searched 1761 nodes with the Distance heuristic and 1748 nodes with the Custom heuristic. Problem 2 required the searching of 13463 nodes with the Distance heuristic and 13428 nodes with the Custom heuristic, while for Problem 3, the numbers were 323482 and 323476 respectively. This slight decrease in nodes searched when using the Custom heuristic suggests that it may provide a slightly more efficient search pattern in the IDA* algorithm, but not a significant amount.

However comparing A* vs IDA* (generally across heuristics), IDA* searches a lot more nodes than A* does. This comes down to the behaviour of IDA*, where it needs to research a bunch of nodes it already searched upon crossing the threshold. Also, for both algorithms, the number of nodes searched varied significantly across different problems, highlighting the fact that the problem's complexity influences the search's breadth more than the choice of heuristic. For example, less pawns and smaller grid size results in less complexity.

4.5 Proof on why the first occurance of a unique state is the shortest path to that state in the optimized A* approach

In this modified A* algorithm, the search is performed in a series of depth-limited searches like an IDA* which remembers its last state before hitting the threshold value. At each iteration, the algorithm explores all possible states up to a certain depth limit. If a goal state is not found at that limit, the search is expanded to the next deeper level. During this process, the algorithm keeps track of the states that have been visited in the hashes mentioned earlier and the cost of the path that led to each state. Since multiple grid states can generate a given unique state (many to one relation) through iterations, the algorithm may encounter a state multiple times at different depths. However, due to the way the algorithm iteratively explores the possible states, the first occurrence of a state in the search tree is guaranteed to have the lowest cost. This is because the algorithm explores the states in increasing order of their depth, and when a state is encountered for the first time, its cost is guaranteed to be the lowest possible for that state. There are other ways to get to that state but it would take more steps/actions to get to that. Therefore, the first occurrence of a unique state in the search tree is always the shortest path to that state, which is why it is chosen for the final path. This property is important for the efficiency of the algorithm, as it reduces the number of nodes that need to be explored and thus reduces the time complexity and space complexity of the search to find the shortest path to the goal state.

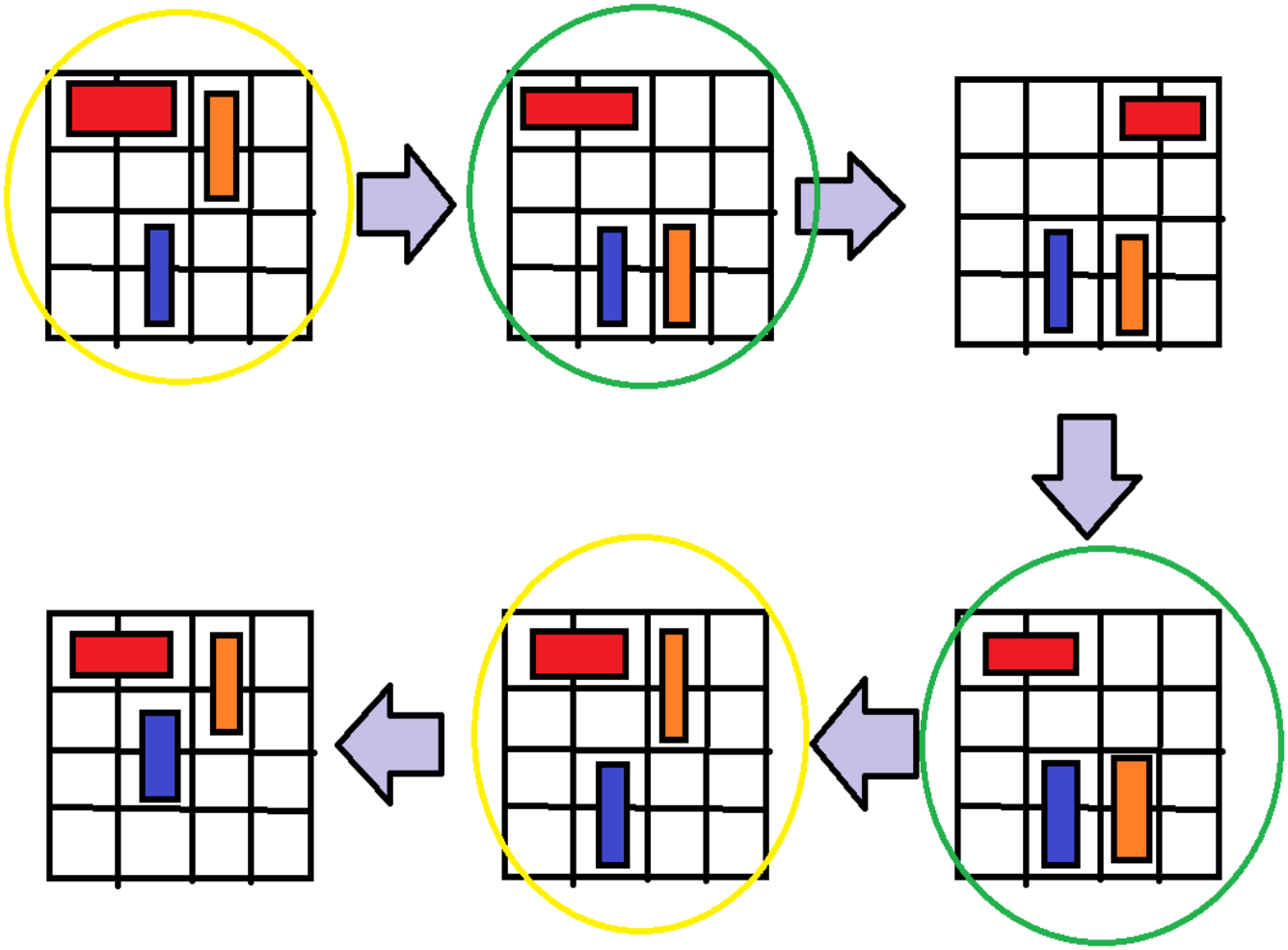

Figure: Iterative deepening steps of the the Rush Hour Puzzle Solver

As we can see in the diagram above, the rings with the same colors have the same states. But the one that occurred first has the lowest number of steps/cost to get to that particular state. This makes that state the lowest cost path to that state.

Chapter 5:
Conclusion

5.1 The trade-offs between memory usage and computational efficiency in A* and IDA* algorithm

The A* and Iterative Deepening A* (IDA*) algorithms balance memory usage and computational efficiency differently. A* tends to be more computationally efficient as it finds the shortest path more quickly, but this comes at the expense of higher memory usage. Higher in the case of this A* algorithm as it needs three lists instead of two. A* keeps track of all the nodes it has explored, which can take up a lot of memory, especially for complex problems. On the other hand, IDA* is more memory-friendly. It doesn't keep a record of all nodes it has visited. Instead, it repeatedly explores the problem space, going a little deeper each time until it finds the solution. However, this iterative approach can make IDA* slower as it may need to revisit the same nodes many times, increasing its computational effort as we have seen through the experiments done earlier.

In conclusion, regardless of heuristics used A* is always faster but needs more memory, while IDA* uses less memory but might take longer. The choice between the two depends on the specific problem and what resources are available to solve the given problem. If memory-space(R.A.M.) is not an issue, A* would be the better option but if it is IDA* is the next best thing.

5.2 Which algorithm is best suited for this problem

The most suitable algorithm for solving this problem depends on the available system resources and the nature of the problem space. If the system is equipped with a large amount of RAM memory and the problem space is not overly large, the A* algorithm would likely be the optimal choice. The A* algorithm's ability to efficiently navigate the state space and prioritise nodes based on cost and proximity to the goal often results in low run times and fewer total nodes searched, which is particularly beneficial when memory is plentiful and problem complexity is relatively low. This algorithm is suitable for cases where we need fast response times and resources are not an issue.

However, if the problem needs to be solved over an extended period and memory space is limited, the IDA* algorithm would be a more suitable option. The IDA* algorithm operates by employing a depth-first search that incrementally increases its depth limit, resulting in a more memory-efficient approach than A*. Although IDA* often necessitates searching through a greater number of nodes and may require longer run times, especially when the problem space is large, its minimal memory requirements make it ideal for systems with scarce memory resources. Consequently, the choice between A* and IDA* should be made based on the specific constraints and requirements of the problem and system at hand.

5.3 Further work:

5.3.1 Explore training a ML model and benchmark against A* and IDA*

An interesting direction for future exploration could be the development and training of a

machine learning (ML) model to solve the problem, and benchmarking its performance against the A* and IDA* algorithms. Machine learning has seen successful applications in numerous problem-solving scenarios and could offer a novel approach to this problem. The training of a ML model would involve providing it with a large amount of data from successful problem-solving instances, from which the model can learn to identify patterns and make accurate predictions about the best nodes to explore. Once trained, the model could be benchmarked against A* and IDA*, comparing key metrics such as the number of nodes searched, runtime, and solution quality. This approach could potentially offer more efficient and effective problem-solving methods, especially in larger or more complex state spaces. This also would be a good opportunity to learn more on data science and ML techniques.

5.3.2 Exploring better custom heuristic functions

Another topic worth exploring is the development and testing of more sophisticated custom heuristic functions. The current custom heuristic provides a primitive cost estimation by taking into account both distance and node cost, but further refinements could potentially enhance its performance. For instance, additional factors such as number of cars in the way, how many steps it would be required to move the cars out of the way and other distance calculation techniques could achieve higher performance. It's essential, though, to maintain the heuristic's admissibility, ensuring that it doesn't overestimate costs. This exploration could result in a more precise heuristic that further improves the efficiency of the A* and IDA* algorithms.

5.3.3 Exploring more IDA* optimizations

Future work could also focus on further optimizing the IDA* algorithm. While IDA* is more memory efficient than A*, it tends to require longer run times and searches through a greater number of nodes. Therefore, introducing optimizations that reduce the number of nodes searched or enhance the algorithm's overall efficiency could significantly improve its performance. Techniques such as advanced pruning methods or effective node ordering strategies could be explored to accelerate the search process. In addition, finding ways to more efficiently handle the increased depth limit could also lead to performance improvements. This research could result in a more optimized IDA* algorithm that provides better problem-solving capabilities, especially in scenarios where memory is scarce.

Bibliography

1. Michael Folgerman: (n.d.). *Rush*. Retrieved from https://www.michaelfogleman.com/rush/

2. Alsedà, Lluís (n.d.). *A* Algorithm pseudocode*. Retrieved from https://mat.uab.cat/~alseda/MasterOpt/AStar-Algorithm.pdf

3. *Ida-Star(IDA*) algorithm in general*. Insight into programming algorithms. (2016, April 19). https://algorithmsinsight.wordpress.com/graph-theory-2/ida-star-algorithm-in-general/

4. *Manhattan*. Manhattan Distance Metric. (n.d.). http://www.improvedoutcomes.com/docs/WebSiteDocs/Clustering/Clustering_Parameters/Manhattan_Distance_Metric.htm