

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**DEPARTMENT OF COMPUTING AND INFORMATION SYSTEMS**

**SCHOOL OF ENGINEERING AND TECHNOLOGY**

**COURSE : ARTIFICIAL INTELLIGENCE**

**COURSE CODE : CSC3206**

**ACADEMIC SESSION : April 2024**

**DEADLINE : 31st May by 11.59 pm**

**Assignment 1 – Group 8**

|  |  |  |
| --- | --- | --- |
|  | **NAME** | **STUDENT ID** |
| 1 | Tan Jun Rong | 21041967 |
| 2 | Yap Jay Ann | 21024765 |
| 3 | Yeong Meng Li | 21018429 |
| 4 | Lim Xiwei | 21045596 |

**Lecturer’s Remark** (Use additional sheet if required)

List down the names and the student ID here.

I **Tan Jun Rong, Yap Jay Ann, Yeong Meng Li, Lim Xiwei** (Student’s Name) **21041967, 21024765, 21018429, 21045596** (Student ID) received the assignment and read the comments.

Rong **12/05/2024**, A black background with a black square

Description automatically generated with medium confidence **12/05/2024**, mengli **12/05/2024**,  **Shape

Description automatically generated with low confidence****12/05/2024** (Signature/Date)

**Academic Honesty Acknowledgement**

“I **Tan Jun Rong, Yap Jay Ann, Yeong Meng Li, Lim Xiwei** (Student’s Name) verify that this paper contains entirely my own work. I have not consulted with any outside person or materials other than what was specified (an interviewee, for example) in the assignment or the syllabus requirements. Further, I have not copied or inadvertently copied ideas, sentences, or paragraphs from another student. I realize the penalties *(refer to page 16, 5.5, Appendix 2, page 44 of the student handbook diploma and undergraduate programme)* for any kind of copying or collaboration on any assignment.”

Rong **12/05/2024**, A black background with a black square

Description automatically generated with medium confidence **12/05/2024**, mengli **12/05/2024**,  **Shape

Description automatically generated with low confidence12/05/2024** (Student’s signature / Date)

# **Introduction**

The treasure hunt event taking place inside the virtual space calls for players to navigate through the traps, obstacles, and rewards in pursuit of obtaining the treasures scattered and laid across the world. To realize this objective, it is of great importance to analyse and comprehend the layout of the world before arranging the best possible outcome for this event. That is, for the players maximize their opportunities to collect the greatest number of treasures before realizing a game-over scenario in the most efficient way possible. For this assignment, we shall assume efficiency to take account of the least cumulative energy expenditure, which is directly proportional to the least number of steps taken. This will allow us to accurately pinpoint the fastest method towards securing our goal.

This virtual world presents itself in a honeycomb pattern with each cell (node) in a hexagon shape (Nazzi, 2016). As such, the arrangement of the cells is not collinear but staggered off each other. The traps, rewards, obstacles, and treasures are scattered across the world. Reward nodes are supporting elements for players to lessen their overall energy expenditure. Conversely, traps serve as the opposing factor to rewards that attempt to hinder players from reaching their objectives. While most traps hinder player progression, triggering trap 4 would be the worst-case scenario as it removes all uncollected traps that initiate game over for the player. Furthermore, obstacles are immutable nodes that block certain paths. This would prompt the player to navigate around the obstacles.

Additionally, it is observed that a reoccurring pattern can be visually seen for these elements. All the treasures can be seen to be placed in a pair with a type of trap. This arrangement can be assumed to be an attempt to slow down player progression. However, the rewards, which offset the negative effects of the traps, are placed not too far apart from the pairs, which makes them easily obtainable. However, it is noted that there are more traps than rewards located in the world and not all rewards may negate the negative effects brought on by the traps. Taking account of all these factors, the importance of employing a suitable search algorithm is heightened as to realize the best employable game-over scenario.

# **Search Algorithm**

For this assignment, our primary focus is comparing the algorithms' efficiency and completeness before reaching a game-over scenario. Both informed and uninformed search algorithms will be considered to gain insights into different applicable search strategies. The actions are defined as moving the player in six directions: up, down, up-right, up-left, down-right, and down-left. The states indicate the player's current position on the grid. We will assume a uniform initial energy expenditure of 1 Kj per cell traversal for all search algorithms. With that in mind, our focus algorithms are Breadth First Search, Uniform Cost Search, and A\* search.

# **Breadth First Search**

Breadth First Search (BFS) is a systematic graph traversal algorithm that explores nodes in layers. It begins with a given node and probes all its neighbours before proceeding to nodes at the next depth level. This method aims to identify paths based solely on the number of steps.

To apply BFS to the treasure hunt, we treat each hexagonal cell in the graph as a node, with edges signifying the possible moves between adjacent cells. Beginning at the entry point, the algorithm initializes a queue to track the cells that need to be explored and their current conditions, which include energy levels and speed that are influenced by rewards and traps encountered. A set is also maintained to keep track of visited cells, ensuring that each cell is processed only once to maintain the efficiency of the search. A cell is dequeued for examination from the front of the queue during the BFS process, and its unvisited nearby cells are enqueued.

Throughout the exploration, BFS will encounter various traps and rewards that may affect movement speed and cost. However, algorithms capable of factoring in the varying costs associated with traps or rewards, whether in terms of steps or energy, BFS does not alter its path based on these effects. Instead, it continues its systematic exploration, exploring each cell level by level without deviation. For example, if BFS encounters a node containing Trap 2, doubling the number of steps needed for neighbouring cell movement, it won't actively avoid it. Similarly, if it encounters Reward 1, halving the energy cost per step, BFS does not take advantage of this benefit unless it's part of the final solution path.

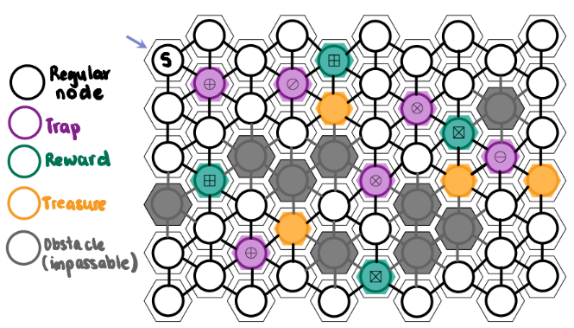
The advantage of using BFS is that it returns the shallowest goal node by exploring nodes layer by layer, ensuring each move is systematically examined. This thorough approach means no potential routes up to the depth level of the last goal node are missed, as all nodes are considered during the search. Additionally, BFS is straightforward to implement compared to other search algorithms, as it utilizes a simple queue to manage the exploration process. Furthermore, it is also a complete algorithm, meaning it will find a solution if one exists, ensuring all treasures can be collected if a valid path is available.

However, BFS has some drawbacks in this scenario with traps and rewards. It can consume significant memory, as it needs to store all nodes at the current level before proceeding to the next, which can be a substantial issue in a large virtual world, and it can be slow in large grids as it explores all nodes at the present depth before moving further. Although BFS determines the shortest path in terms of steps, it does not take into consideration for the varying costs due to traps and rewards, potentially leading to paths that are not ideal in terms of energy consumption or speed modifications. Regardless of whether the cell contains a trap or a reward, BFS explores all walkable neighbouring cells within the map's boundaries. For instance, during its exploration, since it does not have the capability to avoid traps, it will encounter Trap 4 which removes all treasures that have not been collected before encountering any treasures. The treasure hunt will instantly fail as a result of this flaw.

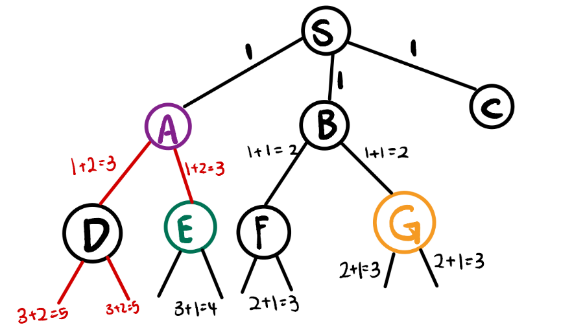
# **Uniform Cost Search**

The Uniform Cost Search (UCS) algorithm can be used to find the optimal path with the least cumulative energy expenditure. In a weighted graph, UCS ensures that the path with the lowest cumulative cost is always explored first. By expanding the least cost node first, UCS prioritizes exploring nodes that require less energy. This means the algorithm will naturally avoid paths with high energy costs or traps that increase energy expenditure unless they are unavoidable. Non-walkable obstacles are detected and avoided, ensuring they are never expanded or traversed. UCS does not consider the optimal number of steps the player must take, and it only evaluates the path based on the energy spent.

To implement UCS, we must first visualize the entire state space as a weighted graph, where each hexagon represents a node and the connections between neighbouring hexagons represent the edges.



In the given example, regular nodes are connected by edges with a default weight of 1 kJ, but this weight can change if the player steps on a trap or reward, affecting the energy cost or steps required for subsequent traversals. Trap 4, designated with a high cost, is typically avoided by the algorithm, as it leads to a game over. Meanwhile, rewards are likely explored as they reduce traversal costs. Treasures are essential nodes that must be collected, with UCS aiming to find the least cost path to gather all treasures. Obstacles are visualized but not included as nodes, effectively blocking paths by not creating edges between them and other nodes. The priority queue ensures the next node expanded has the least total cost from the starting node S, leading to an optimal path.



In a simplified example, the player starts at node S with cumulative energy expenditure of 0 and priority queue [(S,0)]. The first expansion is to its neighbours A (cost 1kJ), B (cost 1kJ)and C (cost 1kJ). The priority queue is updated to [(A,1),(B,1),(C,1)]. Since A, B and C have the same cost, they can be expanded in any order. In this case we expand A first and add its neighbours D (cost 1+2=3kJ) and E (cost 1+2=3 kJ) to the queue. As A is a trap that doubles the energy expenditure, travelling to either D or E from there costs 2 kJ. The priority queue is updated to [(B,1),(C,1), (D,3), (E,3)]. Next, node B is expanded because it has the same cost as C, and its neighbours F (cost 1+1=2 kJ) and G (cost 1+1=2 kJ) are added to the queue. G is one of the treasure nodes, so it is marked as collected. The priority queue is now [(C,1),(F,2),(G,2),(D,3),(E,3)]. From here, continue expanding nodes from the priority queue based on the lowest cost, while collecting treasures and updating the costs accordingly. The algorithm terminates when the least cost path to collect all treasures from the starting node is found. This ensures that the player finds the optimal path with the least cumulative energy expenditure while avoiding high-cost traps and non-passable obstacles.

UCS has several advantages, such as it guarantees an energy-efficient solution because it finds the path with the lowest cumulative cost. It also explores all possible paths the player can take, so it will always find a solution if one exists. Additionally, by expanding the least cost node first, UCS naturally avoids paths with high energy costs or traps that increase energy expenditure.

The possible drawback is UCS requires storing all nodes along with their cost and path in memory, leading to high memory consumption, especially for large graphs. Moreover, because UCS explores all possible paths to ensure the least cost, it can be slower than other algorithms, particularly in such a large and complex state space. Finally, UCS lacks heuristic guidance, meaning it might explore more paths than necessary compared to algorithms like A\*, which use heuristics to find solutions more quickly.

# **A\* Search**

To find the shortest path using the A\* algorithm, we must first assign coordinates to all cells on the map. Unlike square grids, hexagonal grids cannot be accurately represented using x and y coordinates only because of the half-length shift in every second row. Therefore, introducing a third dimension is necessary to describe the hexagonal grid in cube coordinates.

A diagram of different angles of hexagons

Description automatically generated

In this scenario, the entry cell is set at the coordinate (0,0,0), and coordinates for the remaining hexagons are assigned accordingly. The following figure demonstrates how the grid will appear with the provided (x, y, z) coordinates:

A hexagons with numbers and letters

Description automatically generated

To effectively implement the A\* algorithm, the cell expansion sequence is determined by the three parameters: F, G, and H. Parameter F is the sum of G and H, where G represents the current path cost (step) from the starting point to the currently considered cell, and H indicates the estimated lowest path cost from the currently considered cell to the target cell. To determine H, we will use the Manhattan distance adapted for hexagonal grids as the heuristic function. With the cell coordinates, the distance between a cell and the target cell is computed using the formula However, we will adjust our heuristics by assigning higher estimated costs to trap cells and lower ones to reward cells for prioritizing them accordingly.

Initially, with four treasures (goal cells) to collect, the heuristic is calculated from the starting point to each goal cell. The closest goal cell (lowest H value) is chosen as the current target, which in this case is the cell (4, -3, -1) with F = 4.

A screenshot of a game

Description automatically generated

The algorithm starts from cell (0, 0, 0) with G = 0. H is calculated as Max(4, 3, 1) = 4, yielding F = 0 + 4 = 4. Among the adjacent cells, the top right cell (1, 0, -1) with the lowest F value is chosen. Continue selecting the first available cells with the lowest F value, as determined by the implementation, until reaching the goal, omitting cells with traps.

A screenshot of a game

Description automatically generated

The reached goal becomes the new starting point, and heuristics are recalculated for each remaining goal. If goals have the same heuristic value, selection depends on implementation. Assuming cell (3, -5, 2) is the next goal, the path proceeds to the bottom left cell with the lowest F value, avoiding obstacles. Only one walkable cell remains, but its F is higher than the F value of the adjacent cell (5, -4, -1) of the previous cell. Hence, the path switches to cell (5, -4, -1) and continues to the first available cell with the lowest F value. At cell (5, -4, -1), the reward cell is prioritized over the goal cell because rewards facilitate further exploration. Eventually, another goal (7, -6, -1) is reached. Even though the initially intended goal (3, -5, 2) was not reached, the current goal becomes the new starting point, and the steps are repeated for the remaining two goals, thus completing the treasure hunt.

Overall, A\* search is a complete and optimal algorithm when using an admissible heuristic. This informed search algorithm estimates the cost to reach the goal, ensuring it always finds the shortest path if one exists, making it ideal for efficient navigation in virtual worlds. A\* is flexible, allowing its heuristic function to be adjusted for specific characteristics of a treasure hunt, such as higher costs for traps and lower costs for rewards. This adjustment helps balance between avoiding traps and collecting rewards, optimizing player efficiency, and minimizing energy expenditure. However, A\*'s performance heavily depends on the heuristic. If the heuristic is poorly designed or inadmissible (i.e., overestimates the cost of reaching the goal), the algorithm may not find the optimal path or become inefficient.

# **Compare & Contrast Search Algorithms**

After detailing the considered search algorithms, we may now analyse the resultant output of the algorithms and compare the solutions to determine suitability in defining the problem at hand.

|  |  |  |  |
| --- | --- | --- | --- |
| Search Algorithm | Breadth First Search(BFS) | Uniform Cost Search(UCS) | A\* Search |
| Similarities | **Node Expansion:**  All algorithms expand on nodes to explore possible paths in the search space.  **Completeness:**  All algorithms are complete, meaning they will find a solution if one exists.  **Optimal Path:**  All algorithms will find the optimal path to the goal, whether in terms of the fewest steps or the lowest energy expend. | | |
| Approach | * Expands nodes level by level. * Finds the shortest path in terms of steps, not cost. | * Expands nodes based on cumulative path cost measured in energy. * Finds the least-cost path considering varying traversal costs. | * Expands nodes based on total estimated cost (actual cost + heuristic) measured in steps. * Finds the least-cost path efficiently using heuristic guidance. |
| Trap Avoidance | No trap avoidance mechanism.   * Traps are explored along the way in the search process. | Traps are somewhat avoided.   * Traps are avoided indirectly if the cost of passing through them is higher than alternative paths. | Actively avoid trap nodes.   * Traps are avoided by incorporating trap costs in the heuristic function. |
| Reward Collection | No reward collection mechanism.   * The presence of rewards is not considered in the search process. | Rewards are somewhat collected.   * Indirectly prioritize rewards if they lead to lower path costs. | Actively collect rewards if they are adjacent.   * Rewards are collected by incorporating reward costs in the heuristic function. |
| Energy Expenditure | Largest   * Traps trigger regularly without a clear goal to offset their negative impact. * Nodes are prioritized based solely on the number of steps required to reach them. | Medium   * May avoid high-energy traps and collect rewards that reduce energy. * Does not prioritize finding the shortest path. | Smallest   * Balances between avoiding traps, collecting rewards, and finding the shortest path. |
| Efficiency | * Explores all nodes at a certain depth level before moving on leads to excessive exploration of unnecessary nodes. | * Explores unnecessary high-cost nodes before finding the optimal path. | * Uses a heuristic function to guide the search towards the goal, minimizing node exploration. |
| Treasure Collection Progress | Poor   * High risk of triggering Trap 4 leading to game over before obtaining much treasure. | Good   * Less risk of triggering Trap 4 * May indirectly avoid it if reaching it costs more than alternative paths. | Excellent   * Trap 4 can be avoided. |

# **Conclusion**

In conclusion, the implementation of all potential search algorithms mentioned above possesses similarities and differences in their approach to defining the problems. All have demonstrated their suitability and effectiveness in solving the issues at hand. In a scenario featuring traps and rewards, the BFS algorithm, traversing all neighbouring nodes depth by depth, has proven to be a detrimental attempt at solving the issue, despite its ability to return the shallowest goal nodes. With the absence of trap avoidance or reward targeting systems, BFS results in significantly higher overall energy expenditures due to more nodes being explored. This blind search through a significant number of empty nodes wastes energy, especially as traps increase the energy cost per cell traversal. Upon implementing BFS in this trap-filled world, the game-over scenario may be realised before any treasures are collected, as the first encounter with the detrimental Trap 4 happens early on, further highlighting its unsuitability for this environment.

Moving on to the UCS algorithm, it improves upon BFS by prioritizing the least cost path, thereby enhancing energy expenditure management and treasure collection efficiency. However, UCS does not avoid traps that do not affect energy expenditure, resulting in more steps needed to complete the treasure hunt.

Lastly, the A\* algorithm employs a heuristic function that estimates the cost from the current node to the target node. This function ensures that the player reaches the destination without blindly exploring unnecessary empty nodes, thereby both energy expenditure and the number of steps taken. Traps are effectively avoided, while treasure and reward nodes are efficiently reached.

The repeated emphasis on energy expenditure and the total steps required for all the algorithms directly correlate to their efficiency, as stated at the start of this assignment. Among them, the BFS algorithm fails to meet requirements in all aspects of efficiency, treasure collection, and energy expenditure, making it the least suitable of the three search algorithms. The UCS algorithm manages well in securing treasures while minimizing the triggering of traps that affect energy costs. However, its management of energy expenditure cannot surpass the A\* algorithm's minimal energy usage, which adeptly balances between avoiding traps, collecting rewards, and finding the shortest path. Hence, while the UCS algorithm demonstrates notable results, it still falls short of the A\* algorithm, which excels in all required factors. With all the evidence at hand, we determine the ranking of the algorithms to be A\*, UCS, and BFS, from most efficient to least efficient.

# **References**

Magdziarz, M. (2021, September 6). *Pathfinding on a hexagonal grid – A\* Algorithm – The Knights of Unity*. The Knights of Unity - Blog of Knowledge. <https://blog.theknightsofunity.com/pathfinding-on-a-hexagonal-grid-a-algorithm/>

Nazzi, F. (2016). The hexagonal shape of the honeycomb cells depends on the construction behavior of bees. *Scientific Reports*, *6*(1). <https://doi.org/10.1038/srep28341>