CSC584 : Assignment-2 Solutions

Instructor: Dr. Chris Martens

Student Name: **Vinay Kumar** UnityID: **vkumar24@ncsu.edu**

I'm using Processing **3.5.4** with **Python** mode.

PART-2 (PATH-FINDING):

Various path-finding algorithms exit, such as A*, DFS, RBFS, BFS, IDA* etc. In my implementation, I have chosen **A***. Any path-finding algorithm requires a tree or graph representation of the navigation space. In our scenario, the navigation space is the 2D world map of the game. A very realistic, easy and efficient representation of the game world is a **grid** representation. The whole $640*480$ units^2 view of the game is divided into **rectangular** cells of $10*10$ units^2. Each cell functions as a node in the graph representation of the game world and the player (Knight) moves from one cell to another cell only. All other NPCs are fixed in their positions as specified by the JSON input file. There are two main distinction of the cells that can be accessed by the player (obstacles-region and non-obstacle-region). The obstacle-regions includes all the cells that are inside the obstacle-boundaries specified in the JSON file; rest everywhere else is accessible by the player.
One of the **hyperparameters** in implementing the A* algorithm is the **resolution** of the grid. Should I make the cells to be high-resolution ($10 * 10$ units) or low-resolution ($50 * 50$ units) decides the speed of the A* pathfinding. This hyperparameter affects both the speed and the precision of the movement. If we take high resolution then the path looks more natural but A* takes longer time to find the best path to a destination. I have experimented with various grid-resolutions and the results of the pathfinding A* are shown in fig-2. In A* algorithm, we have two sets (open_set and closed_set). The cells in open_set is represented with *green* stroke while the cells in closed_set are represented in *red* stroke. The cells in best path is represented in *blue* stroke.
Following are some details about my implementation:

- **Heuristic Function:** I have implemented two heuristic function (manhattan-distance and eucledian-distance).
- The allowed movements are **Left, Right, Up, Down**. I have not allowed movement in diagonal direction. Hence, when selecting neighbors for an active node (cell), the algorithm looks only in these 4 cells (Left, Right, Up, Down).

Are the paths taken by the agent optimal?
Yes, the path taken by the agent is optimal, but it depends on the time when the person selects the goal location. Because, one of the conditions of the assignment was that the person should be able to manually select the goal location by mouse-clicking on the game-view, the optimal path is dependent on the time of the selection of the goal. It happens because of the following reasons:

- The A* search algorithm starts as soon as the program is run. Because I have set a default **frameRate** of 600, the program runs very quick. I suggest the TAs to reduce the frameRate to be able to slowly monitor the path-finding, openset, closedset and select the **goal** by mouse-click. Due to high frame rate the TAs might have less time to select the goal before the cell has already been explored by the A* algo. The default goal is set to be at $(0, 0)$ (which is an obstacle in the given JSON file, so the A* will explore the whole map before returning NO_SOLUTION if a goal is not selected by clicking the mouse)
- Because the default goal is set at $(0, 0)$, the A* expands nodes/cells with that heuristic and in that process some of the nodes may not be explored if the mouse-selected goal-node and the default goal-node are in opposite directions of the heuristic function. In this situation, the A* may not produce optimal path (compared to had the mouse-selected goal-node been the default goal-node at the very beginning). This is because the **heuristic values** of the cells (nodes) changes mid-way while the A* is finding a solution, this may lead to sub-optimal path. This is shown in fig-(2).

Do the paths taken by the agent look natural?
Yes, the path taken by the agent looks very much natural if the grid-resolution very high. If the grid-resolution is very low (i.e. big cells make the grid) then the path looks un-natural and seems more like jumping than moving.

What are some of the other improvements that could be made to your algorithm??
Some of the improvements that could be made in my algorithm & implementation are:

- Getting better representation of the world map. Currently I am using grid-representation with moderate resolution ($10*10$), but I could have used better representation such as navigation-meshes. Navigation meshes provide better looking results when the Knight moves from one position to another.
- Making more efficient memory management by proper class organization.

PART-3 (DECISION-MAKING):

Behavior Trees and Goal-Oriented-Action-Planning (GOAP) are one of the most widely used decision making algorithm due to their versatility to adapt to new condition/rules rather than making a lookup table for all possible rules/actions/reactions. I am familiar with both the algorithms abd I am using GOAP for my decision making implementation. In my GOAP implementation, I have deployed A* algorithm for searching the best path (sequence of actions) to reach the goal state (i.e. Dead_Rameses or Fight_Rameses or Dead_Knight). A* is a general purpose algorithm for finding the best path in any state-space representation. Here, in this scenario, the state-space consists of states which represents the conditions of the Knight and various NPCs. I have created different base classes for **GameWorld(), Planner(), Actions()**. The GameWorld() class represents the whole game which contains different Planner() classes. Each Planner() class has a set of actions and reaction pair entries that determines the allowed actions and resulting states after that action has been taken.

Some of the challenges faced during implementation of GOAP schedule are:

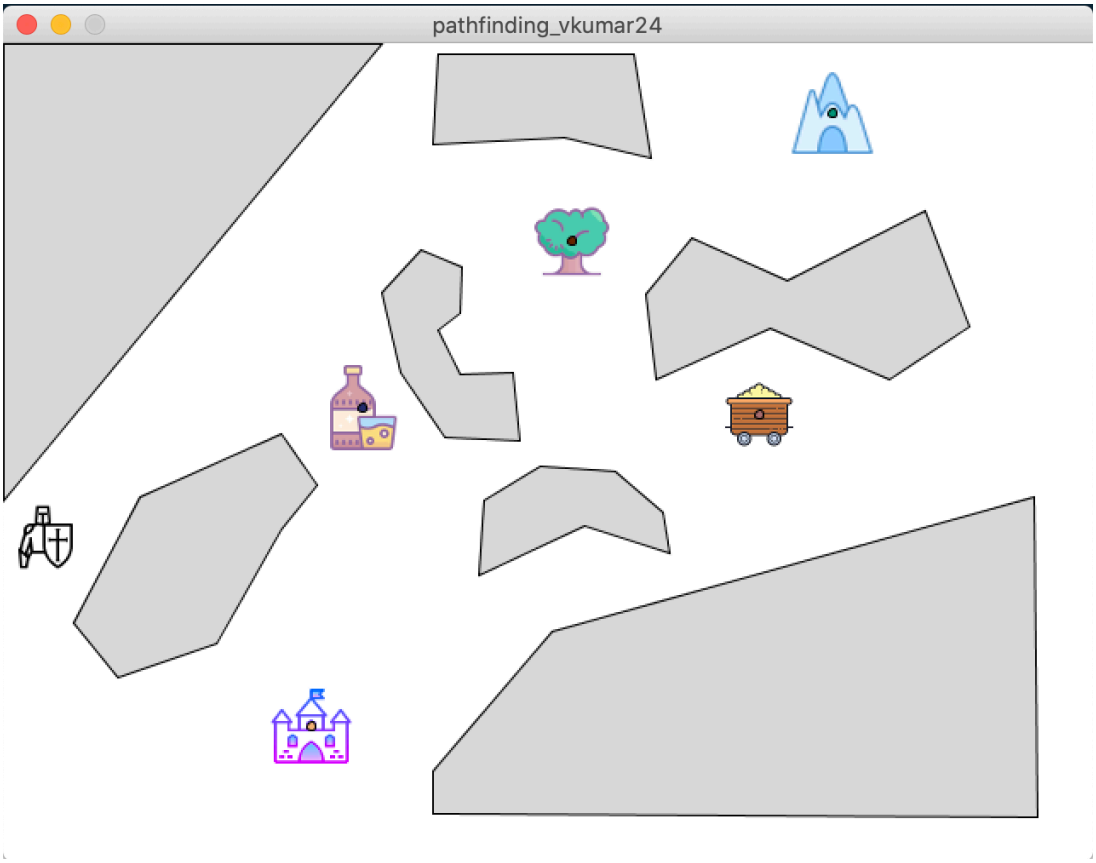
- **Handling the count of number of gold-coins in each state:** Except for the number of gold-coins all other commodities like water, Fenrir, Ale, Wood, etc are either one or infinite; and hence such commodities can be represented with BOOLEAN variables. The challenge I faced was in representing the number of gold coins which can vary depending on the JSON file. I have tried to implement is as integer variable and assigned num_gold attribute to every state. This is not efficient implementation but should work fine.
- **Deciding on the details of the Planner:** I felt it to be tricky to decide on what should be the Planner class instances. Apart from the Knight I could not think of another planner. I have used only one planner here.
- Another approach I tried (without completion) was to design a Decision Tree and use search algorithm (specifically DFS) to find the best sequence of states. The challenge, due to which I left it and adopted GOAP, was the slowness of the algorithm and handcrafting all possible states.

What are some of the other improvements that could be made to your algorithm??
Some of the improvements that could be done in my implementation are:

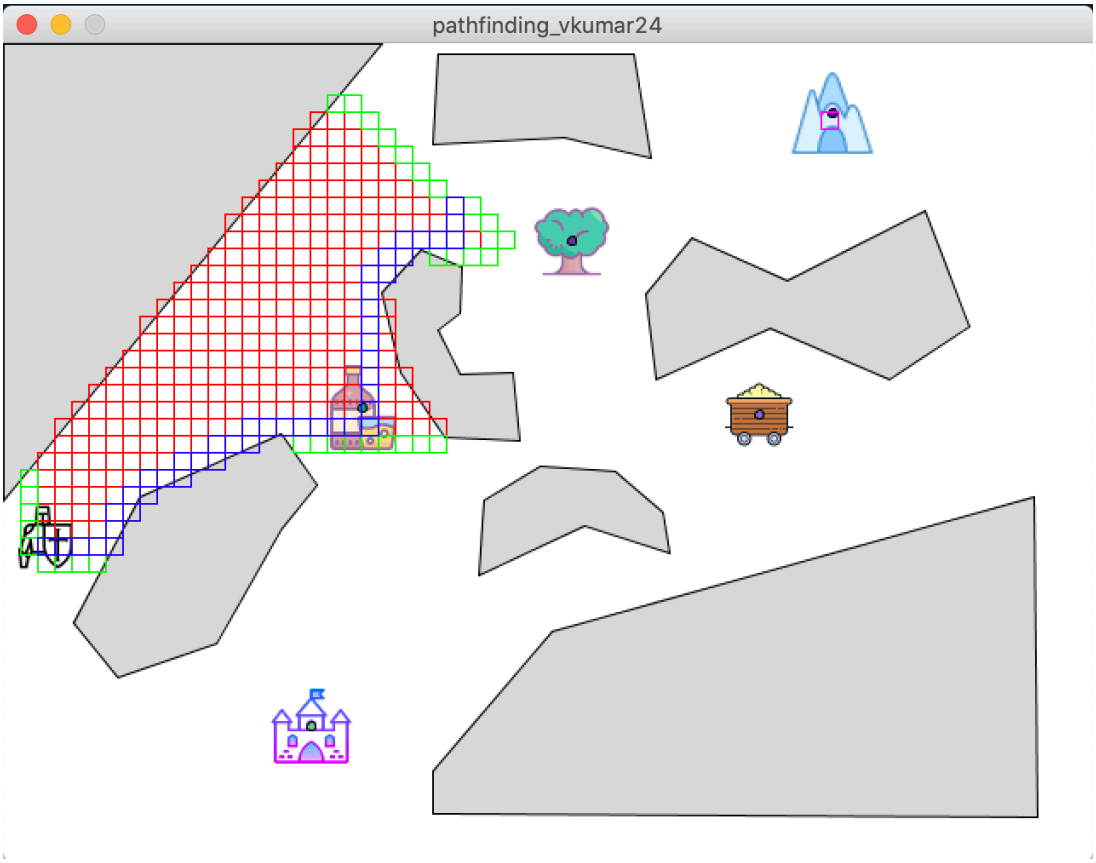
- In the current implementation I have used **A*** for navigation-pathfinding (part2) as well as state-pathfinding (part3). Though I have implemented both these parts separately, a better approach would be implement a single A* search algorithm for both the scenarios. This will help maintain the code-base much better.

The results are shown in fig-3.

********** APPENDIX **********

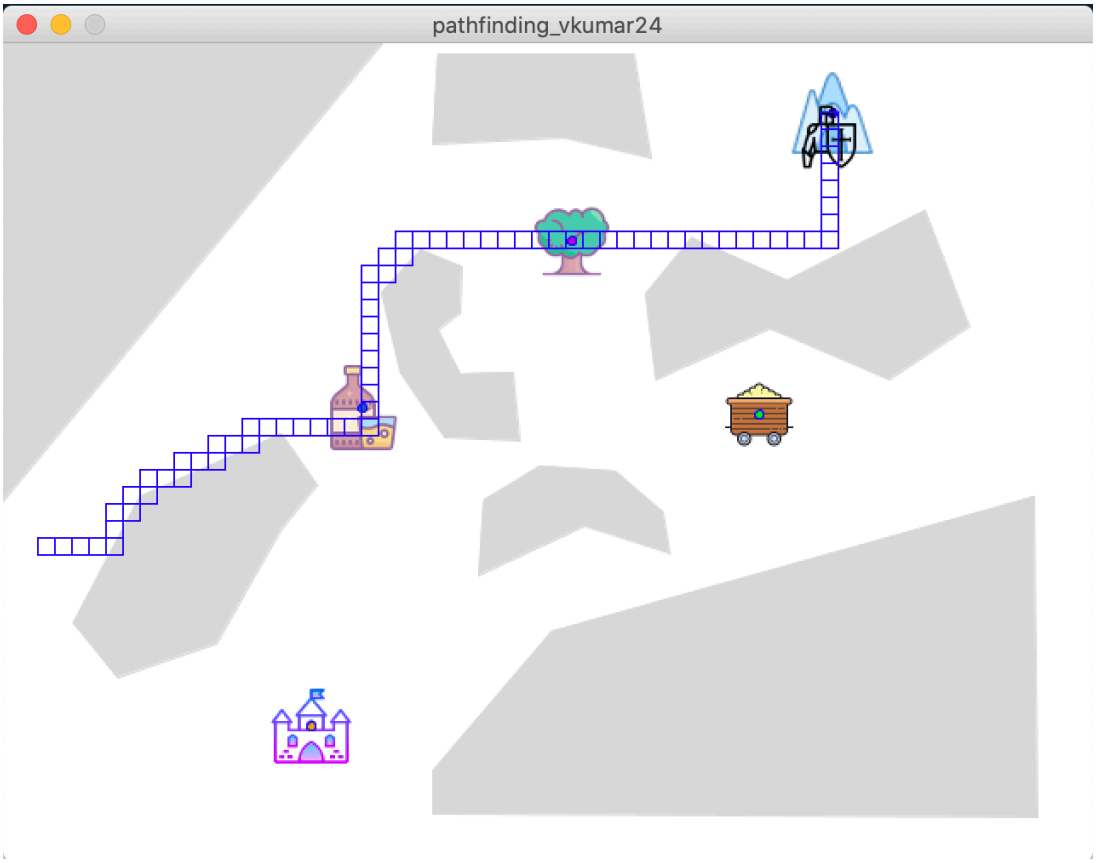


(a) Start Position. Only start_node is specified from JSON file; goal_node is not yet specified. The goal_node is defined when the user selects a location by mouse-click.

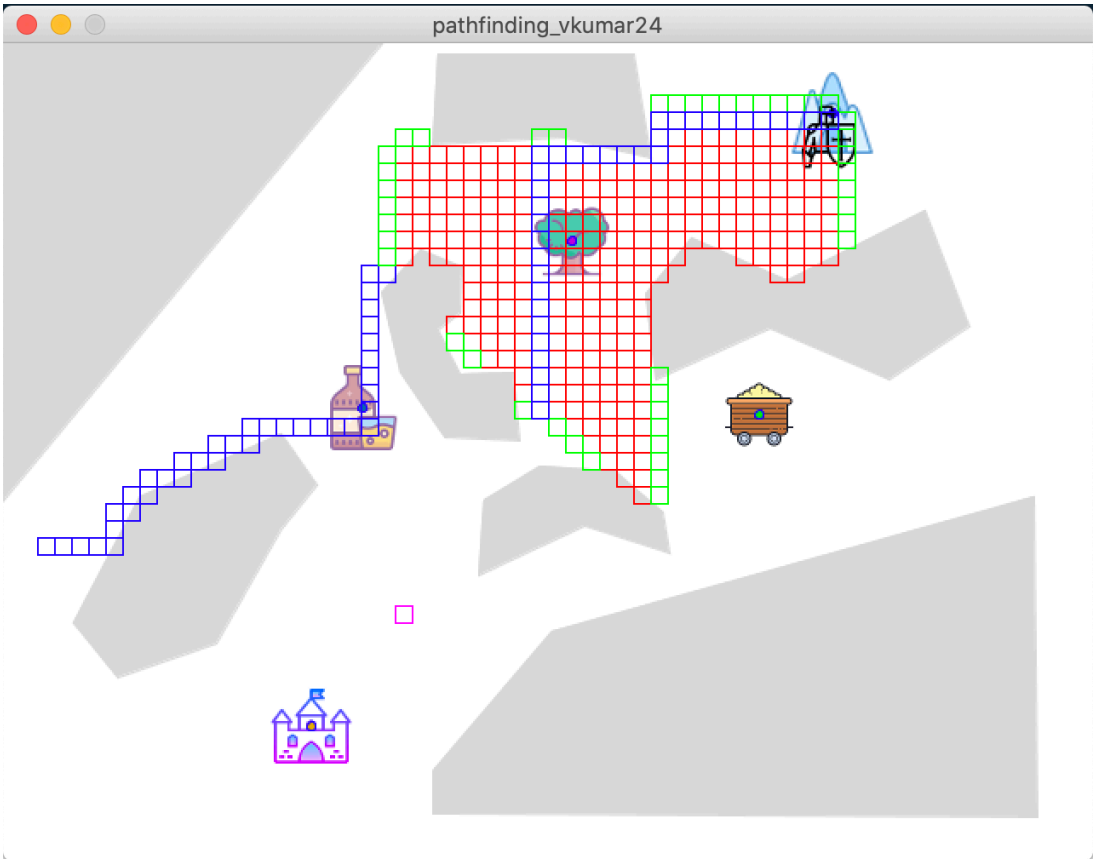


(b) The user selcts the goal_node by mouse-click. The knightAI agent searches for path from start_node to goal_node using **A*** algorithm. The red cells represent the cells(nodes) in **closed_set**. The green cells represent the **open_set**. The blue cells represent the best_path found till now (the search is still going on).

Figure 1: Part-2 (Pathfinding): The basic structure of implementation of **A*** pathfinding algorithm.

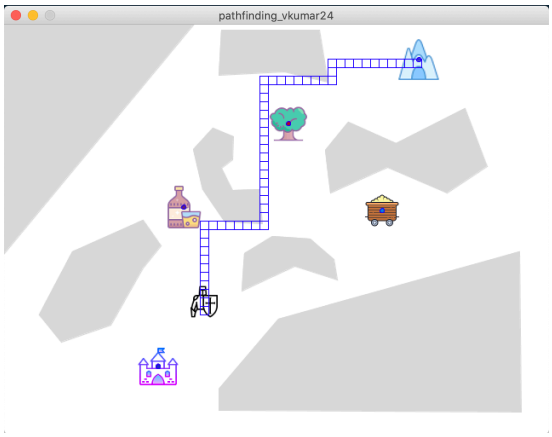


(a) The knightAI agent has found the best shortest path from the start_node to goal_node using A*. The **best_path** is represented in blue cells. After finding the best path, the knight moves from start_node to goal_node. In this figure, state is shown after the knight has moved to the goal_node using best_path trajectory.

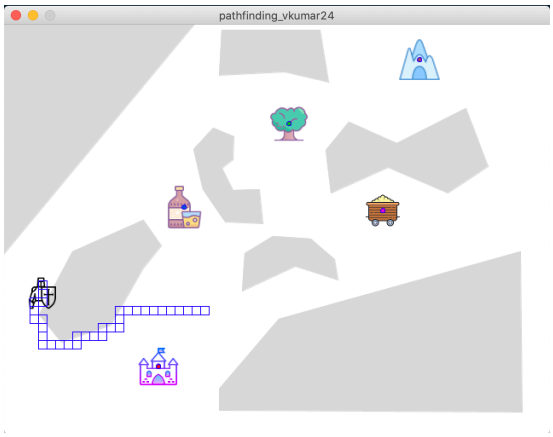


(b) Next, now the user presses the mouse and selects a new goal_node. The knighAI updates its start_node as its previous goal_node. The search for the best path begins. This images shows the closed_set and open_set of currect A* search run. (The previous best path is also shown here, but their color get updates according to the current A* search.

Figure 2: Part-2 (Pathfinding): The basic structure of implementation of **A*** pathfinding algorithm.

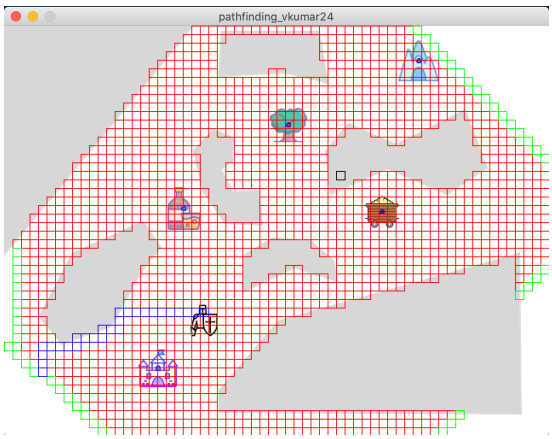


(a) Start Position



(b) Start Position

Figure 3: Part-2 (Pathfinding): These images show the results of the previous step in fig-2 and subsequent similar steps.



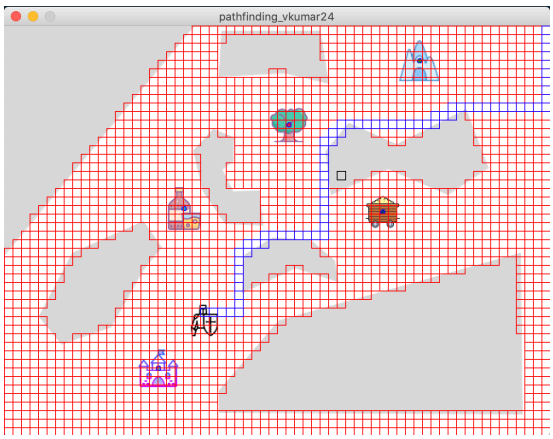


Figure 4: Part-2 (Pathfinding): This shows a condition where the user has selected a goal_node which lies within(or is) an obstacle (can be seen as black-colored cells inside the X-shaped obstacle above tar_pit. Here the A* algorithm exhausts itself to search for a path but fails to find any path and returns. The blue cells represents the best path the algorithm could find. But because it could not reach the goal_node, the knight does not move in this path and stays at its start_node.

Figure 5: (Q3): Decision Making