Assignment 2

Artificial Intelligence

CISC 352, Winter, 2019

Due Friday, March 15, 2019 before Midnight

Pathfinding

Pathfinding is the plotting of the shortest route between two points. This field of research is based heavily on Dijkstra's algorithm for finding a shortest path on a weighted graph. At its core, a pathfinding method searches a graph by starting at one vertex and exploring adjacent nodes until the destination node (goal) is reached, generally with the intent of finding the shortest route. A core use for pathfinding in artificial intelligence is for determining routes for agents to follow. These could be digital agents, such as in video games, or real-world agents, as in the case of robotics.

Although graph searching methods such as a breadth-first search would find a route if given enough time, other methods would tend to reach the destination sooner. An analogy would be a person walking across a room; rather than examining every possible route in advance, the person would generally walk in the direction of the destination and only deviate from the path to avoid an obstruction, and make deviations as minor as possible.

Several algorithms exist for pathfinding. Dijkstra's algorithm (aka uniform-cost) was developed in 1959 by Edsger Dijkstra. It follows the cheapest path cost and is optimal, but it is inefficient when compared to other pathfinding algorithms. Greedy Best-First Search (or just Greedy Search) was described by Judea Pearl. It follows the cheapest heuristic cost and is generally efficient, but it is not optimal. Peter Hart, Nils Nilsson, and Bertram Raphael of Stanford Research Institute first described the A* algorithm in 1968 as an extension of Dijkstra's algorithm. A* is both optimal and efficient. It takes into consideration both the path cost and the heuristic cost.

Alpha-Beta Pruning

A minimax algorithm is a recursive algorithm for choosing the next move in an n-player game, usually a two-player tame. A value is associated with each position or state of the game. This value is computed by means of a position evaluation function and it indicates how good it would be for a player to reach that position. The player then makes the move that maximizes the minimum value of the position resulting from the opponent's possible following moves. If it is A's turn to move, A gives a value to each of his legal moves.

In the context of zero-sum games, the minimax solution is the same as the Nash equilibrium, described by John Forbes Nash, Jr. in his 1951 article *Non-Cooperative Games*. The minimax theorem, in the context of zero-sum games, is equivalent to:

> For every two-person, zero-sum game with finitely many strategies, there exists a value V and a mixed strategy for each player, such that
>
> (a) Given player 2's strategy, the best payoff possible for player 1 is V, and
> (b) Given player 1's strategy, the best payoff possible for player 2 is -V.

Equivalently, Player 1's strategy guarantees him a payoff of V regardless of Player 2's strategy, and similarly Player 2 can guarantee himself a payoff of -V. The name minimax arises because each player minimizes the maximum payoff possible for the other. Since the game is zero-sum, he/she also minimizes his/their own maximum loss (e.g., maximize his/her minimum payoff).

Alpha-beta pruning is a search algorithm that seeks to decrease the number of nodes that are evaluated by the minimax algorithm in its search tree. It is an adversarial search algorithm used commonly for machine playing of two-player games. It stops completely evaluating a move when at least one possibility has been found that proves the move to be worse than a previously examined move. Such moves need not be evaluated sooner. When applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision.

Interestingly, alpha-beta pruning was reinvented a number of times. Arthur Samuel had an early version around 1959, and Richards, Hart, Levine and/or Edwards invented alpha-beta independently in the United States in 1961. John McCarthy proposed similar ideas during the Dartmouth Conference in 1956 and suggested it to a group of his students at MIT in 1961. Alexander Brudno independently conceived the alpha-beta algorithm, publishing his results in 1963. Donald Knuth and Ronald W. Moore refined the algorithm in 1975, and Judea Pearl proved its optimality in 1982.

Solve the Following Problems.

Programming Language Requirement: Python 3

Max Group Size: 6

1. **Pathfinding.**
    a. Your next task is to take a m x n (where 8<=m<=1024 and 8<=n<=1024) input grid containing a start position for an agent, a goal position, obstacles, and open areas, and find a path between the start position and the goal position using the Greedy algorithm and the A* algorithm. Assume that the agent can move up, down, left, and right, but not diagonally. Further assume that the cost of moving up, down, left, or right is 1. Use the most informed distance heuristic for your solution.

    The input will consist of several m x n grids, each separated by a blank line between them. The input will not contain the size of the grid.

    The first line in your output file will be the algorithm used (Greedy or A*). The next part of your output should consist of the same input m x n grid with the path between the start and goal displayed using an uppercase "P".
    An input grid will have a capital "X" represent walls and obstacles, a capital "S" represent the start, a capital "G" represent the goal, and an underscore "_" represent open spaces. Your input file for part a is named "pathfinding_a.txt". And your output file for part a should be named "pathfinding_a_out.txt".

    b. Repeat part a, but now the agent can move up, down, left, right, and diagonally. The cost of moving up, down, left, right, and diagonally is 1. Use the most informed distance heuristic for your solution. Your input file for part b is named "pathfinding_b.txt". And your output file for part b should be named "pathfinding_b_out.txt".
    c. Name the program **pathfinding.py**.

For instance, an input “pathfinding_a.txt” could contain:

```
XXXXXXXXXX
X___XX_X_X
X_X__X___X
XSXX___X_X
X_X__X___X
X___XX_X_X
X_X__X_X_X
X__G_X___X
XXXXXXXXXX
```

Your output “pathfinding_a_out.txt” would resemble something like:

```
Greedy
XXXXXXXXXX
X___XX_X_X
X_X__X___X
XSXX___X_X
XPX__X___X
XP__XX_X_X
XPX__X_X_X
XPPG_X___X
XXXXXXXXXX
A*
XXXXXXXXXX
X___XX_X_X
X_X__X___X
XSXX___X_X
XPX__X___X
XP__XX_X_X
XPX__X_X_X
XPPG_X___X
XXXXXXXXXX
```

Here, Greedy and A* came up with the same solution. This will not always be the case.

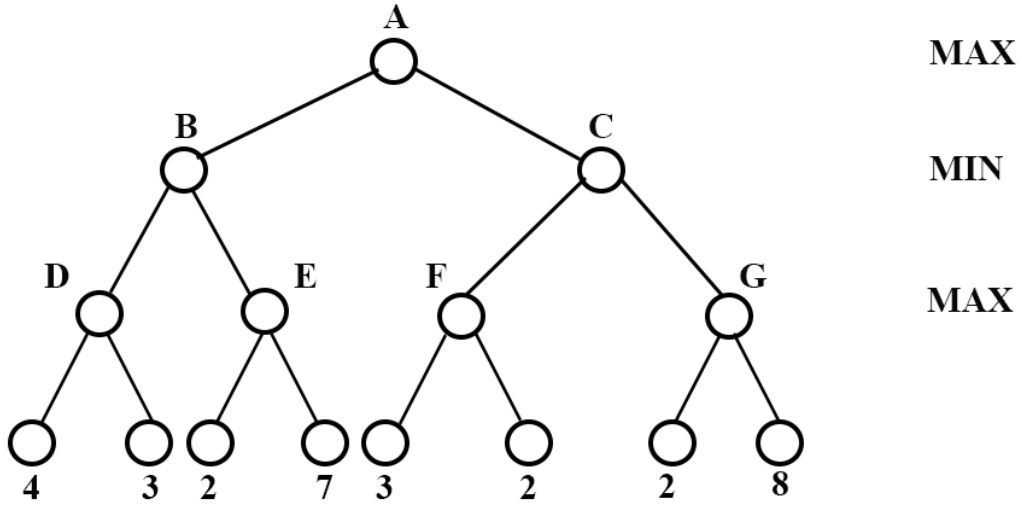
Note that the input text file can contain several grids. For readability, place a blank line between successive solutions to different input grids (but not between the two algorithms used on a single grid).

Also, we have discussed how to visit nodes using Greedy and A* but not how to reconstruct the path found by the algorithms. This is detailed in the “Notes” section of this document.

2. **Alpha-Beta Pruning.** Your task is to output the score and number of leaf nodes examined by using the alpha-beta pruning algorithm on an input graph. The input will be contained in "alphabeta.txt". Your output should go to "alphabeta_out.txt".

   The input "alphabeta.txt" will contain multiple graphs to be examined. These will be in the form of a set of nodes along with its type ("MAX" or "MIN") followed by a **space** and then a set of edges between the nodes. Nodes can contain multiple letters, be uppercase, or lowercase. Aside from leaf nodes, nodes will not contain numeric values anywhere in their name. The maximum branching factor of the input graphs will be 2<=b<=1024.

   The input for the following graph in "alphabeta.txt" would be:
   {(A,MAX),(B,MIN),(C,MIN),(D,MAX),(E,MAX),(F,MAX),(G,MAX)} {(A,B),(A,C),(B,D),(B,E),
   (C,F),(C,G),(D,4),(D,3),(E,2),(E,7),(F,3),(F,2),(G,2),(G,8)}



   Note that those sets are on the same line and separated by a space.
   Your output in "alphabeta_out.txt" for this example input would be:
   Graph 1: Score: 4; Leaf Nodes Examined: 6

   Leaf nodes are recognizable by the fact that they are the only nodes that contain numbers. They will not be listed in the first set. "alphabeta.txt" may contain multiple input graphs on separate lines. There will be a blank line between separate inputs. Name the program **alphabeta.py**.
3. There will be no order given to the first input set, except that it will start with the root node (A in this case). In the second set, edges are listed in left-to-right top-to-bottom order.
4. Turn in the programs, along with a technical document. Zip the submission and name it "**assignment2.zip**".
5. The technical document(s) must also provide details about the algorithms you implemented, along with each function, their purpose, and the code for each function. The key things to this document are that you explain your algorithm well and present it in the best manner possible. The document is required because I allow any high-level language.

Grading

Part 1: **Pathfinding.**

a. Your pathfinding program part a will be run with several input grids. You will be graded on the number of correct solutions.

$$\frac{\#correctSolutions}{\#totalInputStatements} \times 20$$

b. Your pathfinding program part b will be run with several input grids. You will be graded on the number of correct solutions.

$$\frac{\#correctSolutions}{\#totalInputStatements} \times 20$$

b. Part 2 is thus worth a maximum of **40 points**.

Part 2: **Alpha-Beta Pruning.**

a. Your alpha-beta pruning program will be run with several input graphs. You will be graded on the number of correct solutions.

$$\frac{\#correctSolutions}{\#totalInputGraphs} \times 40$$

b. Part 3 is thus worth a maximum of **40 points**.

Part 3: **Technical Document**

a. The third part is a grade on your technical document describing your algorithms. This is worth **20 points**. The grade will include grammar, spelling, presentation quality, and the description of your algorithms. I should be able to know how your algorithms work from reading the document. Regarding presentation quality, **do not submit a .txt file**. The document should be presented as nicely as possible, with headings where appropriate. **Put the name and student ids of all group members on this document. Convert the file to a PDF and name it assignment2.pdf.**

You won't be graded on efficiency in this assignment. However, programs will not be run for more than **15 minutes** and will be considered incorrect after that point.

The total points you can earn for this assignment is **100**.

This assignment is **11%** of your total grade for the course.

Submit the program and technical document, one per group, compressed together in a zip file named **assignment2.zip** to the onQ Assignment 2 Dropbox.

Notes

Pathfinding

Graph search algorithm:

```
frontier = queue()
frontier.put(start )
visited = {}
visited[start] = true

while not frontier.empty():
  current = frontier.get()
  for next in graph.neighbors(current):
    if next not in visited:
      frontier.put(next)
      visited[next] = true
```

This loop is the essence of the graph search algorithms, including A*. But how do we find the shortest path? The loop doesn't actually construct the paths; it only tells us how to visit everything on the map.

We want to use the algorithm for finding paths, so let's modify the loop to keep track of *where we came from* for every location that's visited, and rename `visited` to `came_from`:

```
frontier = priorityqueue()
frontier.put(start, 0)
came_from = {}
came_from[start] = none

while not frontier.empty():
  current = frontier.get()

  if current == goal:
    break

  for next in graph.neighbors(current):
    if next not in came_from:
      priority = heuristic(goal, next)
      frontier.put(next, priority)
      came_from[next] = current
```

Now **came_from** for each location points to the place where we came from. These are like "breadcrumbs". They're enough to reconstruct the entire path.

The code to reconstruct paths is simple: *follow the arrows backwards **from** the goal **to** the start*. A path is a *sequence of edges*, but often it's easier to store just the nodes:

```
current = goal
path = [current]
while current != start:
  current = came_from[current]
  path.append(current)
path.append(start)
path.reverse()
```

That's the simplest pathfinding algorithm. It works not only on grids but on any sort of graph structure.

For Greedy search, the code is:

```
frontier = priorityqueue()
frontier.put(start, 0)
came_from = {}
came_from[start] = none

while not frontier.empty():
  current = frontier.get()

  if current == goal:
    break

  for next in graph.neighbors(current):
    if next not in came_from:
      priority = heuristic(goal, next)
      frontier.put(next, priority)
      came_from[next] = current
```

For A*, the graph search code is very similar:

```
FRONTIER = PRIORITYQUEUE()
FRONTIER.PUT(START, 0)
CAME_FROM = {}
COST_SO_FAR = {}
CAME_FROM[START] = NONE
COST_SO_FAR[START] = 0

WHILE NOT FRONTIER.EMPTY():
  CURRENT = FRONTIER.GET()

  IF CURRENT == GOAL:
    BREAK

  FOR NEXT IN GRAPH.NEIGHBORS(CURRENT):
    NEW_COST = COST_SO_FAR[CURRENT] + GRAPH.COST(CURRENT, NEXT)
    IF NEXT NOT IN COST_SO_FAR OR NEW_COST < COST_SO_FAR[NEXT]:
      COST_SO_FAR[NEXT] = NEW_COST
      PRIORITY = NEW_COST + HEURISTIC(GOAL, NEXT)
      FRONTIER.PUT(NEXT, PRIORITY)
      CAME_FROM[NEXT] = CURRENT
```

Note, we use a heuristic function.

For Euclidean distance, this can be defined as

```
DEF HEURISTIC(A, B):
  # EUCLIDEAN DISTANCE ON A GRID
  RETURN SQRT(POW(B.X-A.X,2) + POW(B.Y - A.Y,2))
```

For Manhattan distance, this can be defined as:

```
DEF HEURISTIC(A, B):
  # MANHATTAN DISTANCE ON A GRID
  RETURN ABS(A.X - B.X) + ABS(A.Y - B.Y)
```

For Chebyshev distance, this can be defined as

```
DEF HEURISTIC(A, B):
  # CHEBYSHEV DISTANCE ON A GRID
  RETURN MAX(ABS(B.X-A.X),ABS(B.Y-A.Y))
```

Alpha-Beta Pruning

```
Function alpha_beta (current_node, alpha, beta)
{
     if is_root_node (current_node)
     then
     {
          alpha = -infinity
          beta = infinity
     }
     if is_leaf (current_node)
     then return static_evaluation (current_node);
     if is_max_node (current_node)
     then
     {
          alpha = max (alpha, alpha_beta (children, alpha, beta));
          if alpha >= beta
          then cut_off_search_below (current_node);
     }
     if is_min_node (current_node)
     then
     {
          beta = min (beta, alpha_beta (children, alpha, beta));
          if beta <= alpha
          then cut_off_search_below (current_node);
     }
}
```