CS 486/686 Assignment 1 (100 marks)

Jesse Hoey & Josh Jung

Due Date: February 5, 2024; Time: AOE

Instructions

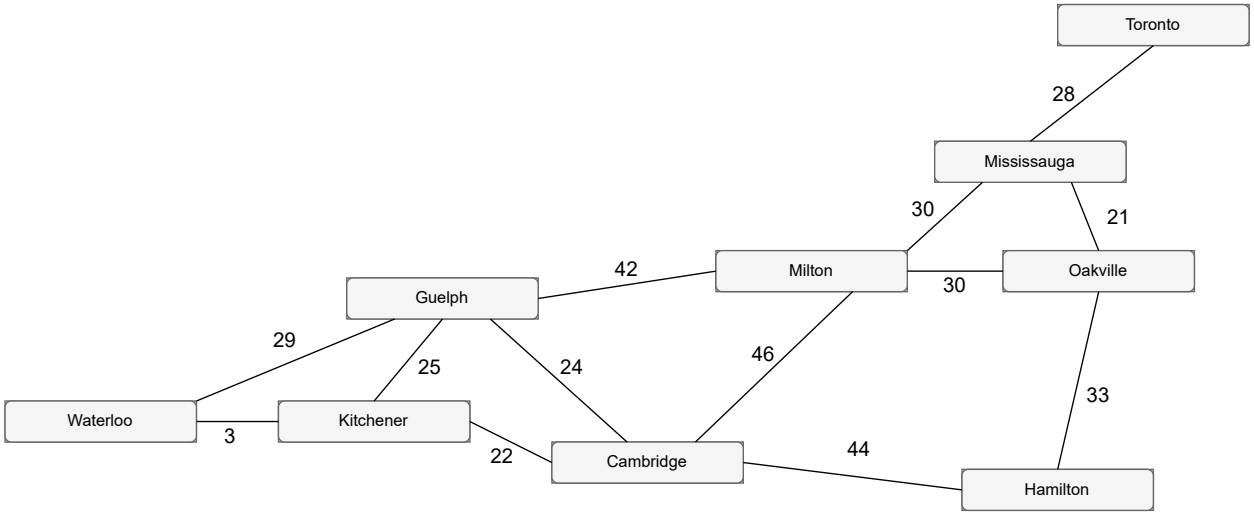
- Submit any written solutions in a file named `writeup.pdf` to the A1 Dropbox on `Learn`. If your written submission on Learn does not contain **one** file named `writeup.pdf`, we will deduct 5 marks from your final assignment mark.
- Submit any code to `Marmoset` at https://marmoset.student.cs.uwaterloo.ca/.
- Use the latest Python 3.12
- No late assignment will be accepted.
- This assignment is to be done individually.
- The Due Date is February 5, 2024, "Anywhere On Earth": if it is Februrary 6, 2024 wherever you are, the deadline has passed.
- Lead TAs:
  - Jess Gano (jgano)
  - Shuhui Zhu (s223zhu)

  The TAs' office hours will be scheduled on Piazza.

1 Shortest Route to Waterloo (30 points)

Suppose that you want to drive to Waterloo from Toronto. You are conscious of your carbon footprint; therefore, you are seeking the shortest path to Waterloo.

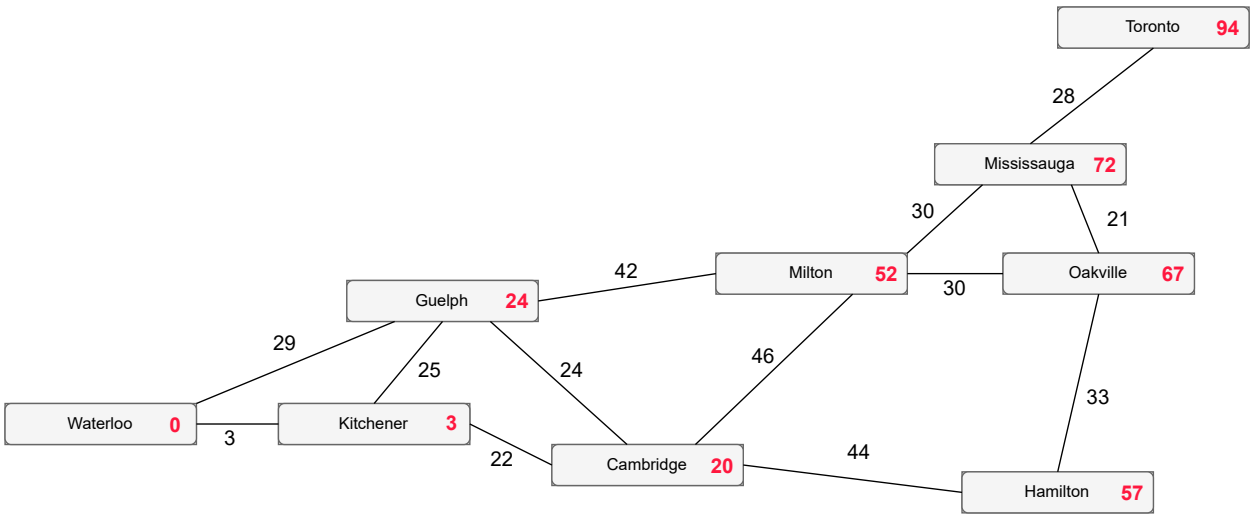
Below is a graph indicating the driving distances between various cities surrounding Toronto and Waterloo. All distances are given in kilometres.



You would like to apply A* search to identify the shortest route to Waterloo from Toronto. Below are the components of the search problem formulation:

- **States:** Each city is the state. We will identify each state by the first 3 letters of its name. For example, the "Guelph" node is denoted as `Gue`.
- **Initial state:** `Tor`
- **Goal state:** `Wat`
- **Successor function:** State B is a successor of state A if and only if there exists an edge on the above graph connecting city B and city A.
- **Cost function:** The cost of an edge connecting two states is the distance between the cities that the states correspond to.

Your decide to use Euclidean distance as a heuristic function. That is, h is the Euclidean distance from a city to Waterloo (in kilometres). That is, if (x_C, y_C) is city C's longitude and latitude coordinates, $h(C) = \sqrt{(x_C - x_{Wat})^2 + (y_C - y_{Wat})^2}$. On the diagram below, the red text number to each city indicates its Euclidean distance to Waterloo.



Please complete the following tasks.

(a) **[5 pts]** Describe why the Euclidean distance to the destination is an admissible heuristic function. Use no more than 4 sentences.

(b) **[5 pts]** Describe why Euclidean distance to the destination is a consistent heuristic function. Use no more than 4 sentences.

(c) **[20 pts]** Execute the A* search algorithm on the problem using the Euclidean distance heuristic function as described above. Do not perform any pruning. Add nodes to the frontier in alphabetical order. Remember to stop if you expand the goal state.

When drawing nodes, remember to abbreviate cities by writing the first 3 letters. For example, label a "Waterloo" node as `Wat`. Annotate each node in the following format: $C + H = F$ where C is the cost of the path, H is the heuristic value, and F is the sum of the cost and the heuristic values. Clearly indicate which nodes you expanded and in what order. You do not need to write out the frontier, but the tree must show all paths expanded after removing a node from the frontier.

Break any F-value ties using alphabetical order. For example, "Milton" precedes "Mississauga" and should be expanded first if the F values for both nodes are the same.

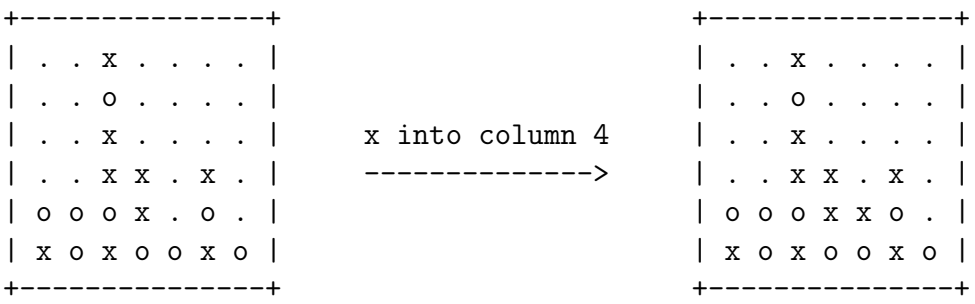
We recommend using https://app.diagrams.net/ to draw the search tree.

2 Minimax and Connect Four (50 points)

For this question, you will be programming Minimax agents to play Connect Four.

Connect Four is an adversarial game in which two players, one 'x' and one 'o', take turns dropping their pieces into the top of a vertical grid with 6 rows and 7 columns. To play a move, a player must select one of the 7 columns into which to drop a piece. The piece then falls downward through the grid, staying in the same column and stopping when it reaches the bottom, or when the square below it is already occupied by another piece. If a column is completely full, it may no longer be selected. In this way, pieces may be stacked on top of each other until either one player is able to get four of their pieces in a row (vertically, horizontally, or diagonally), or the grid becomes completely full. If one player achieves four-in-a-row, that player wins, but if the board fills up before either player has done so, the game is a draw.

In Python, we represent the board as a two-dimensional list, where the first index references the column, and the second references the row. Both are indexed from 0, where (0,0) is the bottom-leftmost square. In the example below, the 'x' player plays into column 4, it descends to row 1, and 'x' wins the game via a diagonal four-in-a-row.



Information on the Provided Code

We have provided two files: `connect_four.py` and `agents.py` on Learn. You will need to complete the `minimax` and `my_heuristic` functions in `agents.py`, and then submit `agents.py` on Marmoset. You may submit `agents.py` as many times as you wish until the deadline. Marmoset will evaluate your program for its correctness and efficiency. Written answers should be submitted on Learn.

`connect_four.py` contains the Connect Four game implementation, including the `State` class that contains game state information and provides functions for querying and advancing the game state. You will need to call some of these functions in your own code.

`agents.py` contains a few example heuristics, as well as the `MinimaxNode` class, which you must use to build a tree in the `minimax` function.

`agents.py` also implements several agents that extend the `Player` abstract class used for games of Connect Four. Each agent implements an `initialize` function that is called once at the start of a game and a `play` function that is called each time it is their turn to play. `MinimaxPlayer` requires your implementation of `minimax` to function.

You may run `agents.py` to set up games between agents and test your code. The lines beneath `if __name__ == "__main__":` may be edited freely for this purpose. See the comments in that section for examples.

Otherwise, you should not alter any existing code outside of `minimax` and `my_heuristic`, or alter the signatures of those functions. Doing so may cause the automatic tests to fail. You may add your own helper functions, if you wish.

(a) **[30 pts]**Complete the `minimax` function in `agents.py`, which implements minimax with heuristic evaluation at a given depth. Initially, `minimax` is given a root `MinimaxNode` that contains a valid `state` attribute, but only default `value` and `successors` attributes. Your `minimax` function will need to set these attributes and act recursively on the successor nodes.

You may use the minimax pseudocode from Lecture 3b as a starting point. (Be careful with the recursive calls; they are not exactly the same!) Be sure to read the documentation associated with `State` (and its functions), `MinimaxNode`, and `minimax` to find useful helper functions, as well as the types of all parameters and return values. In addition, you may find the `deepcopy` method useful for making copies of a `State` that can be altered without changing the original.

(b) **[5 pts]** Try running a game of a `RandomPlayer` vs. a `MinimaxHeuristicPlayer` at depth 2. Use the provided heuristic `three_line_heur` for `MinimaxHeuristicPlayer`, which evaluates states based on the difference in the number of three-in-a-rows that each player has. Does the game terminate in less than 10 seconds? Increase the depth until the game does not terminate within 10 seconds. What is the first depth at which this occurs?

(c) **[5 pts]** Fill out `my_heuristic` by creating your own heuristic function that you expect to outperform `three_line_heur`. In actual terminal states, `my_heuristic` should produce a value of 100 if the maximizing player won, -100 if the minimizing player won, or 0 if it was a draw.

Describe your heuristic function as concisely as possible. Why should it outperform `three_line_heur`?

(d) **[10 pts]** Fill out the table below by recording the results of games between the agents specified. For each table row, run at least 10 games. Record wins and losses from the perspective of Agent 1. (For games that specify a depth of 5, you may decrease to a depth of 4 if games are taking more than 10 seconds to terminate.) You can write code to automate this process and add it to agents.py if you wish.

Agent 1	Agent 2	Wins	Draws	Losses
MinimaxPlayer, depth 3, three_line_heur	RandomPlayer			
MinimaxPlayer, depth 3, my_heuristic	RandomPlayer			
MinimaxPlayer, depth 5, three_line_heur	MinimaxPlayer, depth 2, three_line_heur			
MinimaxPlayer, depth 5, my_heuristic	MinimaxPlayer, depth 2, my_heuristic			
MinimaxPlayer, depth 5, my_heuristic	MinimaxPlayer, depth 5, three_line_heur			

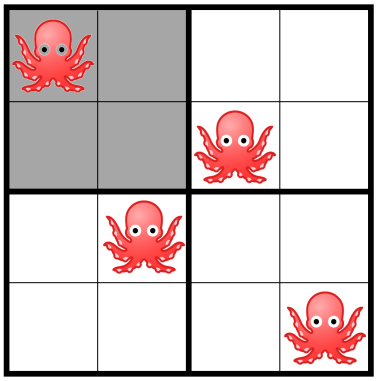
(e) **Bonus (10% on assignment grade)** On Marmoset, we will run `MinimaxPlayer` using your version of `my_heuristic` vs. another `MinimaxPlayer` using our own secret heuristics. There are five trials consisting of different heuristics. Each is a public test (meaning that you will be shown the results for each of your submissions). For each trial, your agent must win a best of five (i.e. win more games than it loses). For each trial passed, you will receive a 2% bonus on this assignment. Trials will be run at depth 4, and will time out (fail) after 10 seconds. You may assume that your agent can use half of that time. If you think you got unlucky, you can always resubmit.

3 Constraint Satisfaction (20 points)

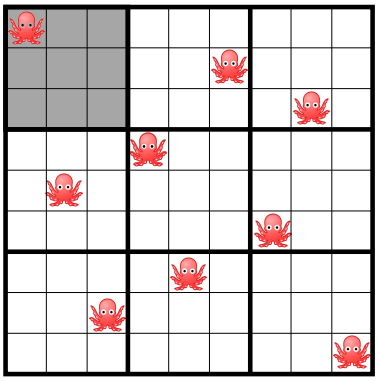
The N-Queens problem is to place N Queens on an $N \times N$ chessboard (grid) so that no Queen can attack any other Queen. Queens can attack one another if they are on the same row, same column, or same diagonal. Thus, a solution to the N-Queens problem has N Queens on an $N \times N$ chessboard with no two Queens on the same row, same column, or same diagonal.

1. **[10 pts]** Formulate the N-Queens problem as a constraint satisfaction problem (CSP) by giving the variables, their domains, and the constraints. Formulate the constraints mathematically so they are as precise as possible.

2. **[10 pts]** Like the N-Queens problem, the N-Octopi problem is to place N Octopi on an $N \times N$ chessboard so that no Octopus can attack any other Octopus. However, Octopi attack in a slightly different way than Queens. Octopi *cannot* attack on a diagonal, but can attack if they are on the same row or column. Furthermore, Octopi can attack each other if they are within one of the $M \times M$ *blocks* that make up the $N \times N$ grid, where M is integer, $M \geq 2$, $N = M^2$, and the M^2 blocks completely fill the $N \times N$ grid (so they are all aligned and adjacent). This is shown below for $M = 2$ (a 4×4 grid) and $M = 3$ (a 9×9 grid): the darker lines delineate the *blocks*, the top left block is shaded, and Octopi are shown in positions from which they cannot attack each other. If two Octopi were in the same block, or on the same row or on the same column, they could attack one another. Formulate the N-Octopi problem as a constraint satisfaction problem (CSP) by giving the variables, their domains, and the constraints.



$M = 2, N = 4$



$M = 3, N = 9$