CS 4100: Artificial Intelligence (Fall 2023) Lawson L.S. Wong
Northeastern University Due October 6, 2023

Exercise 2: Search and Adversarial Search

Please remember the following policies:

ˆ Exercises are due at 11:59 PM Boston time (ET).

ˆ Submissions should be made electronically on Canvas, as a single .pdf file. You can make as many submissions as you wish, but only the latest one will be considered, and late days will be computed based on the latest submission.

ˆ Each exercise may be handed in up to one day late (24-hour period), penalized by 10%. Submissions later than this will not be accepted. There is no limit on the total number of late days used over the course of the semester.

ˆ Solutions may be handwritten or typeset. For the former, please ensure handwriting is legible. If you write your answers on paper and submit images of them, that is fine, but please put and order them correctly in a single .pdf file. One way to do this is putting them in a Word document and saving as a PDF file.

ˆ You are welcome to discuss these problems with other students in the class, but you must understand and write up the solution yourself, *and* indicate who you discussed with (if any). If you are collaborating with a large language model (LLM), acknowledge this and include your entire interaction with this system. We strongly encourage you to formulate your own answers first before consulting other students / LLMs.

ˆ Contact the teaching staff if there are *extenuating* circumstances.

Collaborators: Roger Flores

1. 2 points. *This is a continuation of Ex1 Q1, applying greedy search and A* to the same graph.*

   Consider the graph below. The agent starts at node s and must reach node g. All edges are undirected, so they can be traversed in both directions (recall this is equivalent to a pair of directed edges). Apply greedy search and A* to find a path from s to g.
   Use the graph search version of these algorithms. The heuristic value of each state are given by the *h* value in the node. Assume that nodes with the same priority are *added* to the queue/stack in alphabetical order. (This means that nodes are removed from a stack in *reverse* alphabetical order; also, note that "s" is part of the alphabet.) For algorithms that use a priority queue, also use alphabetical order to break ties when removing nodes with the same priority (e.g., (a,10) before (c,10)). Show:

   The order in which nodes are expanded

   The contents of the frontier after each expansion, in the correct order / with priorities if applicable ˆ

   The contents of the explored set, if applicable

   The solution path that is returned, if a solution is found

   Hint: You do not have to draw out the search tree. Also recall that nodes technically store *paths* instead of states, but it is fine to just write the corresponding state to denote the search node. The solution is the path stored in the final search node, *not* the list of states expanded.

Greedy:

Current node removed/expanded	Fronter(queue)	Explored Set
None	[s]	[]
s	[(c,2), (b,3)	[s]
c	[(d,1), (e,1), (b,3)]	[s,c]
d	[(g,0), (e,1), (b,3)]	[s,c,d]
g(s -> c -> d -> g)		

A*:

Cost of path so far	Current node removed/expanded	Fronter(queue)	Explored Set
0	None	[(s, 3)]	[]
0	s	[(b, 4), (c,5)	[s]
1	b	[(c,4), (d, 6)]	[s,b]
2	c	[(d,5), (e,5)]	[s,b,c]
4	d	[(e,5), (g,7)]	[s,b,c,d]
4	e	[(g,5)]	[s,b,c,d,e]
5	g(s -> b -> c -> e -> g)		

2. 2 points. (AIMA 3.5) *This is a continuation of Ex1 Q2, considering heuristics for the two-friends problem.*

Suppose two friends live in different cities on a map, such as the Romania map shown in class. On every turn, we can simultaneously move each friend to a neighboring city on the map. The amount of time needed to move from city *i* to neighbor *j* is equal to the road distance *d*(*i, j*) between the cities, but on each turn the friend that arrives first must wait until the other one arrives (and calls the first on his/her cell phone) before the next turn can begin. We want the two friends to meet as quickly as possible.

Let SLD(*i, j*) be the straight-line distance between cities *i* and *j*. Which of the following heuristic functions are admissible? If admissible, provide a brief explanation; otherwise, provide a counterexample.

(a) SLD(*i, j*):

No, the heuristic is not admissible. While the minimum distance between two friends can be calculated by the SLD(i,j), imagine a case where all the cities are arranged in a line, and the the friends are on opposite sides of the same city A(i) →(cost of 2)→ B ←(cost of 2)← C(j). While the SLD(i,j) = 4, the

actual time needed for the friends to meet is actually 2, as they are both traveling a distance of 2. Therefore, this heuristic can overestimate the actual cost and therefore is not admissible.

(b) $2 \cdot \text{SLD}(i, j)$

No, this heuristic is not admissible, for the same reason as above: 2 * SLD(i,j) = 8, whereas the actual cost would be 2.

(c)SLD(i,j)/2:

This heuristic is admissible. The minimum amount of time needed for two friends to meet can never be less than SLD(i,j)/2. Case 1: The friends are equidistant from the city where they are meeting(the cities are still in a line): A(i) →(cost of 2)→B ←(cost of 2)← C(j). The time needed = 2 = SLD(i,j)/2. Case 2: The friends are not equidistant from the city where they are meeting: A(i)→(cost of 3)→ B←(cost of 1) ←C(j). Even though j is closer than SLD(i,j)/2 from B, j must still wait for i to arrive at the city, which takes time of 3. The best case would be if the two friends are equidistant, in which the cost would be equal to SLD(i,j)/2. Therefore this admissible never overestimates the actual cost.

3. 5 points. (AIMA 3.14.) Which of the following are true and which are false? Explain your answers.

(a) Depth-first search always expands at least as many nodes as A* search with an admissible heuristic.

False. DFS can expand fewer nodes if the solution lies quickly along a deep branch of the search tree. A*, even with an admissible heuristic, will prioritize paths based on their path costs, which will lead it to explore other branches and thus expand more nodes.

(b) $h(s) = 0$ is an admissible heuristic for the 8-puzzle.

True. An admissible heuristic is one that never overestimates the true cost to reach the goal from a given state. Assuming there are no negative costs, this heuristic will never overestimate the cost to reach the goal. However, this heuristic is useless since it provides no guidance to the search algorithm.

(c) A* is of no use in robotics because percepts, states, and actions are continuous.

False. Even though these states are continuous, a parallel discrete state space can be defined. Our world, although continuous in nature, is measured and acted upon by robots in discrete steps. Otherwise, how would Shakey have come to life if A* is useless in robotics?

(d) Breadth-first search is complete even if zero step costs are allowed.

True. BFS is complete, meaning it will find a solution if it exists in the finite search space. Even with zero step costs, BFS explores all nodes at the current depth before moving on to the next level, ensuring that it explores all possible paths and will eventually find the solution.

(e) Assume that a rook can move on a chessboard any number of squares in a straight line, vertically or horizontally, but cannot jump over other pieces. Manhattan distance is an admissible heuristic for the problem of moving the rook from square A to square B in the smallest number of moves.

False. Because a rook can move any number of squares vertically or horizontally, the Manhattan distance can overestimate the number of moves required. For instance, if the rook is in the bottom left corner and the goal is to get to the top right corner, the rook only needs to complete one move(go straight up). However, the Manhattan distance would calculate the sum of absolute differences in the row and column positions, returning a value of 8(assuming the chess board is 8x8).

4. 5 points. (AIMA 6.3.) Consider the problem of constructing (not solving) crossword puzzles for fitting words into a rectangular grid. The grid, which is given as part of the problem, specifies which squares are blank and which are shaded. Assume that a list of words (i.e., a dictionary) is provided and that the task is to

fill in the blank squares by using any subset of the list. Formulate this search problem precisely, using two different strategies for filling in blanks:

(a) Filling in blanks one letter at a time.

State space:

A state, (grid, #blank, square_to_fill)

s, represents the current crossword grid where some squares have letters and others are still blank

Initial state: An empty crossword grid with all blank squares(grid, size of cross-word, top_left_square)

Action, a, where a letter chosen from the dictionary is placed into the square_to_fill

The state space is all permutations of the crossword grid filled with different letters

Successor function:

Successor(state) => The set of grid configurations where:

- a letter is chosen from the dictionary and placed in the square_to_fill such that all words or partially formed words in the grid are "valid"(can be found in the dictionary)
- A new square_to_fill is chosen such that it is adjacent to the initial square_to_fill, or a new random blank square if there are no more blank adjacent squares
- #next_blank = #blank - 1

Goal Test: #blank = 0(all words should be valid from the successor function)

Path cost: Apply a uniform cost to each step

(b) Filling in blanks one word at a time

State space:

A state, (grid, #blank_word_space, word_space_to_fill)

s, represents the current crossword grid that is broken up into a series of "word-spaces"(I.E. a single set of horizontal squares such that the leftmost square has no adjacent squares to the left && same thing for the rightmost square)

Initial state: An empty crossword grid with all blank word_spaces(grid, #blank_word_spaces, top_left_word_space)

Action, a, where a word chosen from the dictionary is placed into the word_space_to_fill

The state space is all permutations of the crossword grid filled with different words

Successor function:

Successor(state) => The set of grid configurations where:

- a word is chosen from the dictionary and placed in the word_space_to_fill such that all words or partially formed words in the grid are "valid"(can be found in the dictionary)
- A new word_to_fill is chosen such that it is overlaps or is adjacent to the initial word_to_fill or a new random blank word space if there are no more blank adjacent word spaces
- #next_blank = #blank - 1

Goal Test: #blank_word_spaces = 0(all words should be valid from the successor function)

Path cost: Apply a uniform cost to each step

(c) Which strategy for filling in blanks is better? Be technical in your explanation, e.g., by commenting on the expected differences in time/space complexity.

The approach using words to fill in blanks is much more time efficient than the strategy using just letters. The branching factor for letter by letter is 26*4, as in the greatest branching case, any letter can be placed in the square_to_fill, and all four adjacent squares are blank. The depth of the search is equal to the size of the grid, n. Therefore the letter by letter approach is at worst $O(104^n)$.

The approach using words, on the other hand, has a branching factor = the number of words that fit into word_space_to_fill*number of intersections/possible next word_space_to_fill. This branching factor is likely dominated by the size of the dictionary, which we will denote as s. The depth on the other hand, is equal to the # of word_spaces in the total grid, n'. An average crossword will have an n' smaller than n by a factor of 5(average length of words ~ 5).

Letter by letter: O(104^n)

Word by word: O(s^(n/5)

If s = 100,000 words and n = 50(~10 words)

Even if the dictionary is quite large, $100{,}000^{10}$ is much smaller than 104^50.

This type of problem can actually be formulated as a *constraint satisfaction problem* (CSP), which is a special type of search problem. Because these problems have extra structure (variables, domains, constraints), they can be solved using more specialized (less general purpose), but much more efficient, algorithms such as *backtracking search*, *forward checking*, and *arc-consistency* (AC-3). Due to time constraints, we will not cover CSPs in this semester's version of the course, but if you are interested in the topic, feel free to look at Ch. 6 of AIMA (the textbook).

5. 5 points. (AIMA 3.31.) Recall that we came up with two different heuristics for the 8-puzzle by relaxing problem constraints. In an 8-puzzle, the set of valid actions are described by the following statement:

A tile can move from square A to square B if A is adjacent to B and B is blank.

We can generate three relaxed problems (leading to three admissible heuristic functions) by removing one or both of the above conditions:

- A tile can move from square A to square B if A is adjacent to B
- A tile can move from square A to square B if B is blank
- A tile can move from square A to square B

Recall that the first relaxation gives us the sum-of-Manhattan-distances heuristic, and the third relaxation gives us the number-of-misplaced-tiles heuristic. The second relaxation leads to a relaxed problem, whose (optimal) solution is known as *Gaschnig's heuristic* (Gaschnig, 1979).

(a) Explain why Gaschnig's heuristic is at least as accurate as the number-of-misplaced-tiles heuristic. Hint: Observe that since both are admissible heuristics, it is equivalent to showing that:

$$0 \leq \text{Number of misplaced tiles} \leq \text{Gaschnig's heuristic} \leq h^*$$

In the above, $h^*$is the true cost-to-go function. In particular, explain why the number of misplaced tiles is always an underestimate of Gaschnig's heuristic.

The 3rd relaxation allows tiles to move to any spot, even if it is not blank. Therefore, to the heuristic evaluation of a single tile out of place would be equal to 1. The second relaxation, however, enforces that B must be blank. Therefore, in order to move a single tile into the correct space, there are two cases:

1. B is blank: A moves into B, taking 1 move and is not in the correct space
2. B is not blank: B must move to a blank space, then A moves into the newly vacated space, taking two moves.

Therefore, the number of misplaced tiles will either always be equal to or less than Gaschnig's heuristic, which accounts for B not being blank => more moves need to be taken => higher heuristic cost.

(b) For Gaschnig's heuristic and the sum-of-Manhattan-distances heuristic, show that neither is strictly better than the other. To show this, find an 8-puzzle instance where the sum-of-Manhattan-distances heuristic is more accurate than Gaschnig's heuristic, and find an 8-puzzle instance where the opposite is true. Hint: The latter case may be harder to find. Consider an instance that is very close to the goal state.

More "accurate" = closer to true cost

Goal:

1	2	3
4	5	6
7	8	

8-puzzle instance where Manhattan is better:

7	2	3
4	5	6
1		8

Manhattan Distance: 5

Gaschnig's: 4

Where Gaschnig is better:

1	2	3
7	5	6
4		8

Manhattan: 3

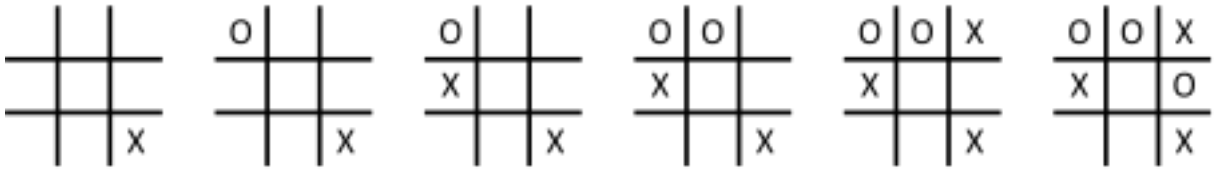
Gaschnig's: 4

(c) Describe a method to calculate Gaschnig's heuristic efficiently. You do not have to prove its

correctness, but you should convince yourself that your method is correct.

1. If a blank if in one of the tile positions(not in the bottom right corner), move any mismatched tile into the blank(+1)
2. Find the tile that should go into the new blank's location, and move it into the blank(+1)
3. Repeat until reached goal state

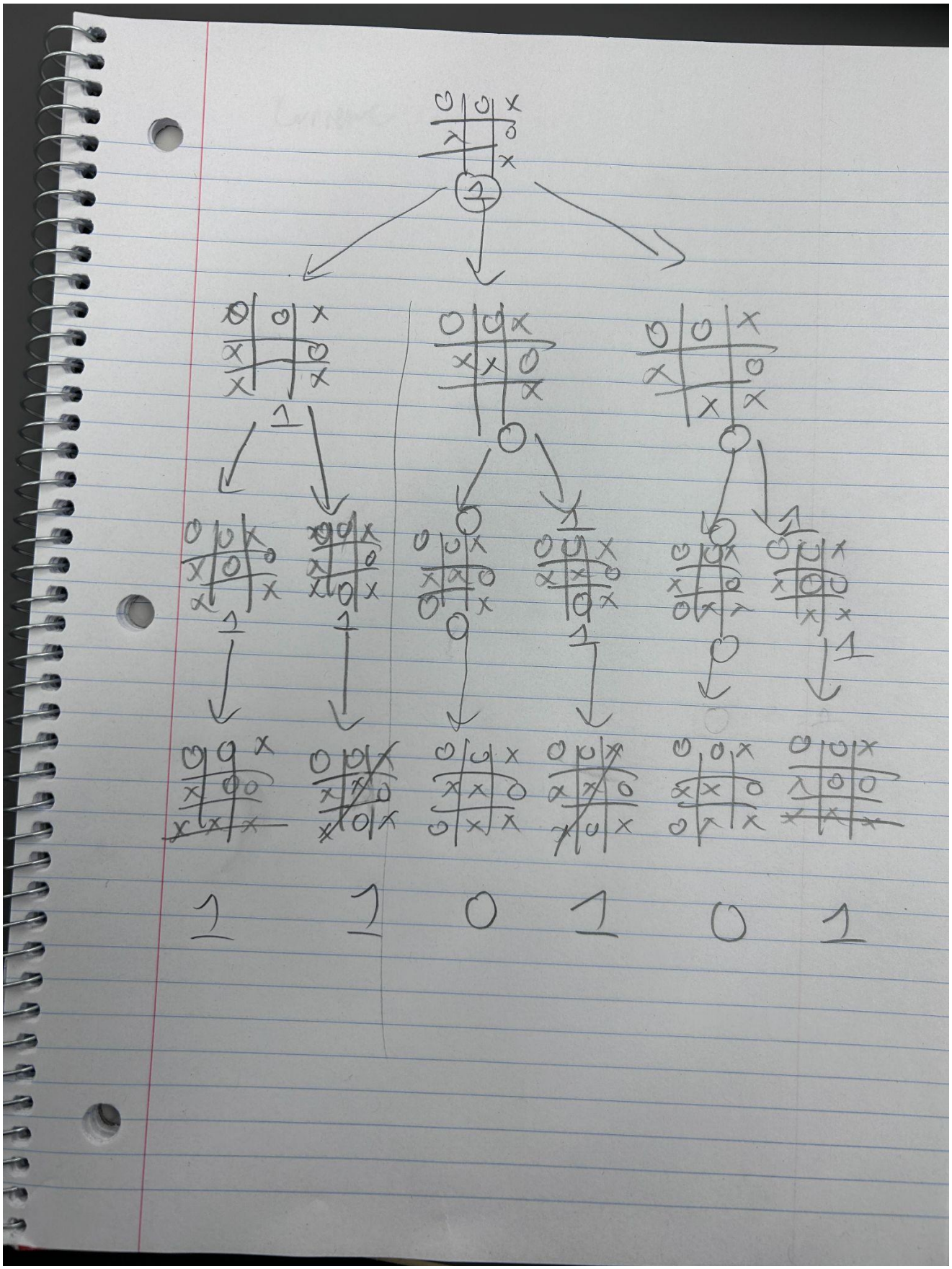
6. 5 points.



$s_1\ s_2\ s_3\ s_4\ s_5\ s_6$

The first six moves of a tic-tac-toe (noughts and crosses) game is shown above (left to right). It is now X's turn, starting at the right-most state.

(a) Draw the game tree starting from the final position shown above. The game tree should include all terminal states as well. Apply minimax search to this tree, showing the backed-up value of each node.



(b) Does X have a winning strategy at the current (right-most) state? What should X play next?

X does have a winning strategy: X should play a move at the bottom left corner(1,3). From the mini-max

tree, we see that X can win 100%of the time if it plays this move.

(c) Recall that a game is "solved" if the outcome of every game state is known, assuming each player plays optimally (in a minimax fashion) from that state onward. Tic-tac-toe is a solved game; the optimal outcome from the all-blank state is a draw. In the above game, did any player play suboptimally? If so, which move(s) do you think was suboptimal?
(Just take an educated guess. You do *not* have to do minimax search for this part, or do too much reasoning. We will continue this problem in PA2 . . .)

The first move and the second move of O is suboptimal. From the first move that O takes, the game is already lost(they can the top right corner). O must block. X then goes to the bottom left corner, creating a double threat, and thus the game is won.

In this game, X's second move is also suboptimal, but O then blunders (forcing X to take the top right corner => O must block on the side => X can take the bottom left corner = double threat).