**Homework 2** (15%)

CSE 5120 (Section 01) – Introduction to Artificial Intelligence – Fall 2024

*Submitted to*

Department of Computer Science and Engineering  
California State University, San Bernardino, California

*by*

Cameron Hartling (007735059)

Mason Audet (007862784)

Dustin Santoso (007854906)

Joseph Godinez (007845624)

Date: October 20, 2024

*Email:*

* [007735059@coyote.csusb.edu](mailto:007735059@coyote.csusb.edu)
* [007862784@coyote.csusb.edu](mailto:007862784@coyote.csusb.edu)
* [007854906@coyote.csusb.edu](mailto:007854906@coyote.csusb.edu)
* [007845624@coyote.csusb.edu](mailto:007845624@coyote.csusb.edu)

**Report**

I formed a group with Mason Audet, Dustin Santoso, and Joseph Godinez. Mason worked on the Minimax algorithm, I worked on the Negamax algorithm, and we both worked on the GameStatus\_5120.py file. Dustin and Joseph both worked on getting the GUI in the large\_board\_tic\_tac\_toe.py file. We all helped on debugging and smoothing out the issues in the code.

1. **Minimax algorithm with Alpha-Beta pruning**

The minimax algorithm is used for a zero-sum game and the processing time is shortened with alpha-beta pruning. It calls itself recursively and switches between a maximizing and minimizing agent. At the beginning of the function it checks to see if the board is in a terminal state. If it is in the terminal state or at the max depth, it will return the end score immediately. After this the available moves are then found. The algorithm then determines which player it is calculating and goes into that branch. The maximizing agent calculates the greatest value between the possible moves. This value is taken from the recursive call that goes into the minimizing agent. It swaps between the minimizing and maximizing agent until the depth or terminal state is reached and then returns that final score. The algorithm will return the position of the next move in the best possible path.

To limit the number of times minimax recursively calls itself, alpha-beta pruning is used. This saves the values of the highest minimax and when checking the other branches it will prune the branches that can’t possibly have a higher value.

def minimax(game\_state: GameStatus, depth: int, maximizingPlayer: bool, alpha=float('-inf'), beta=float('inf')):

# Obtains the T/F boolean value from the is\_terminal() function

terminal = game\_state.is\_terminal()

# If we are at the end of our declared depth or the game is in a terminal state,

# collect the score of the board and return it to the parent caller

if (depth==0) or (terminal):

newScores = game\_state.get\_scores(terminal)

return newScores, None

#

availableMoves = []

for i in range(len(game\_state.board\_state)):

for j in range(len(game\_state.board\_state[i])):

# print('This at i and j→', game\_state.board\_state[i][j])

if game\_state.board\_state[i][j] == 0:

availableMoves.append((i,j))

print("Avail moves")

print(availableMoves)

# #input()

# This section of code is for our maximizing agent

if maximizingPlayer:

# The max evaluation is set to an arbitrary small value by default

# so that the first value compared with it will take it's place

maxEval = float('-inf')

# Conduct the recursive calls that switch between the Max and Min

# agents for each possible move on the board. The best move is checked

# at each step by evaluating the corresponding game score.

for position in availableMoves:

# Pass in (depth - 1) to move down a level and "False" to give control

# to the minimizing agent

print("game state before func get\_new\_state() →", game\_state.board\_state)

new\_game\_state=game\_state.get\_new\_state(position)

print("\tgame state before minimax →", game\_state.board\_state)

print("new\_game\_state before minimax →", game\_state.board\_state)

currEval,currMove = minimax(new\_game\_state, depth - 1, False, alpha, beta)

print("new\_game\_state after minimax →", game\_state.board\_state)

print("\tgame state after minimax →", game\_state.board\_state)

if currEval > maxEval:

bestMove = position

maxEval = currEval

"""elif currEval == maxEval:

decision = random.choice([1, 2])

if decision == 1:

bestMove = position"""

#bestMove=position

print("best move before alpha",bestMove)

#input()

# If the current eval has a better score than our highest known score,

# set the bestMove to the current position in the loop

#if currEval > alpha:

# print(maxEval)

#if currMove:

#bestMove = currMove

# maxEval = currEval

# alpha = currEval

# Need to see if this approach is allowed / works

# In the case that the two scores being compared are equal, the implication

# is that both options lead to the same utility at the end of the game.

# Rather than simply leaving the first node found as the chosen position,

alpha = max(alpha, currEval)

print('best moves',bestMove,'and new alpha is', alpha)

if beta <= alpha:

break

# alpha is set to the larger value between itself and the currEval.

# Then, if the beta is <= alpha, the branch is pruned

#print("1Check alpha > currEval?",alpha, currEval)

#alpha = max(alpha, currEval)

#print("2Check alpha > currEval?",alpha, currEval)

#if beta <= alpha:

# break

# Returns the best possible score for the maximizing agent and the corresponding move

return currEval,bestMove

# This section of code is for our minimizing agent

else:

# The min evaluation is set to an arbitrary large value by default

# so that the first value compared with it will take it's place

minEval = float('inf')

# We obtain a list of available moves by calling the get\_moves()

# function in the GameStatus file

# Conduct the recursive calls that switch between the Max and Min

# agents for each possible move on the board. The best move is checked

# at each step by evaluating the corresponding game score.

for position in availableMoves:

# Pass in (depth - 1) to move down a level and "True" to give control

# to the maximizing agent

new\_game\_state=game\_state.get\_new\_state(position)

currEval,currMove = minimax(new\_game\_state, depth - 1, True, alpha, beta)

if currEval < minEval:

bestMove = position

minEval = currEval

# If the current eval has a lower score than our lowest known score,

# set the bestMove to the current position in the loop

#input()

# Need to see if this approach is allowed / works

# In the case that the two scores being compared are equal, the implication

# is that both options lead to the same utility at the end of the game.

# Rather than simply leaving the first node found as the chosen position,

# the code below will randomly pick one of the two equal score positions

"""elif currEval == minEval:

decision = random.choice([1, 2])

if decision == 1:

bestMove = position"""

# beta is set to the smaller value between itself and the currEval.

# Then, if the beta is <= alpha, the branch is pruned

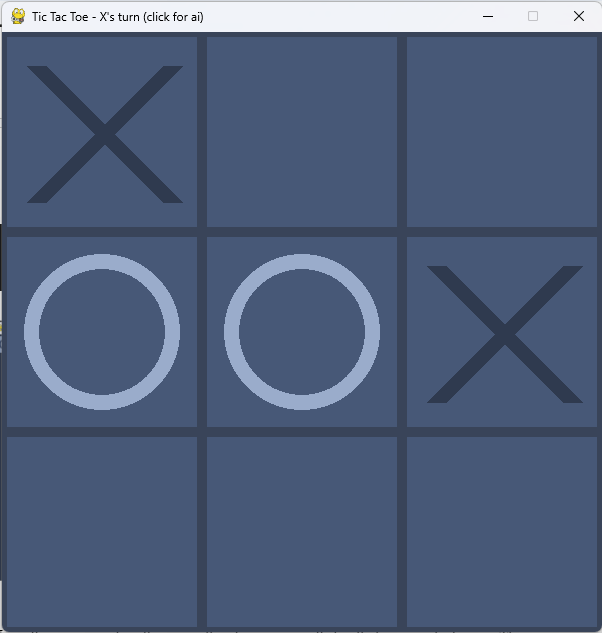
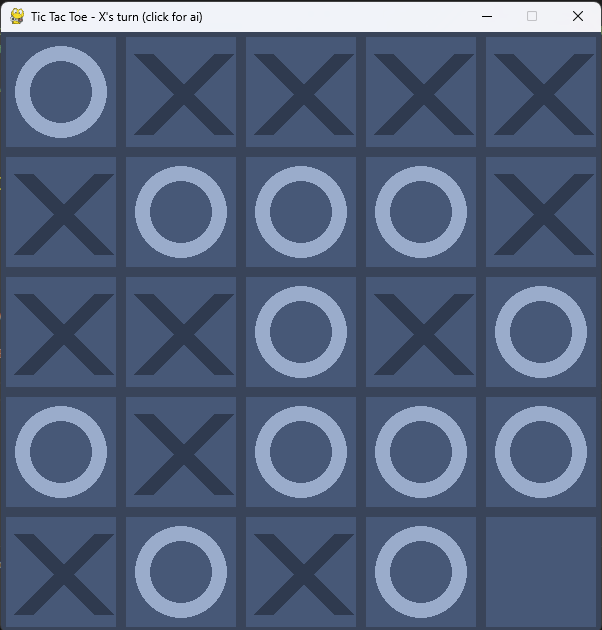
beta = min(beta, currEval)

if beta <= alpha:

break

# Returns the best possible score for the minimizing agent and the corresponding move

return currEval, currMove



1. **Negamax algorithm**

Using the negamax algorithm we will make a program that will play against a human. The negamax algorithm works similarly to the minimax function, but it does not require knowing which player is making their move. When the negamax algorithm recursively calls itself it gets the opposite value of the recursive call. By getting the best move for the opponent the algorithm determines which move is the worst for the opponent and returns that. This simplifies the minimax function as it does not require to keep track of which players turn it is.

def negamax(game\_state: GameStatus, depth: int, turn\_multiplier: int, alpha=float('-inf'), beta=float('inf')):

"""

YOUR CODE HERE TO CALL NEGAMAX FUNCTION. REMEMBER THE RETURN OF THE NEGAMAX SHOULD BE THE OPPOSITE OF THE CALLING

PLAYER WHICH CAN BE DONE USING -NEGAMAX(). THE REST OF YOUR CODE SHOULD BE THE SAME AS MINIMAX FUNCTION.

YOU ALSO DO NOT NEED TO TRACK WHICH PLAYER HAS CALLED THE FUNCTION AND SHOULD NOT CHECK IF THE CURRENT MOVE

IS FOR MINIMAX PLAYER OR NEGAMAX PLAYER

RETURN THE FOLLOWING TWO ITEMS

1. VALUE

2. BEST\_MOVE

THE LINE TO RETURN THESE TWO IS COMMENTED BELOW WHICH YOU CAN USE

"""

# Obtains the T/F boolean value from the is\_terminal() function

terminal = game\_state.is\_terminal()

# If we are at the end of our declared depth or the game is in a terminal state,

# collect the score of the board and return it to the parent caller

if (depth==0) or (terminal):

newScores = turn\_multiplier \* game\_state.get\_negamax\_scores(terminal)

return newScores, None

# We obtain a list of available moves by calling the get\_moves()

# function in the GameStatus file

availableMoves = []

for i in range(len(game\_state.board\_state)):

for j in range(len(game\_state.board\_state[i])):

#print('This at i and j→', game\_state.board\_state[i][j])

if game\_state.board\_state[i][j] == 0:

availableMoves.append((i,j))

# Set value to minimum and initialize bestMove

value = alpha

bestMove = None

# Loop through possible moves

for position in availableMoves:

# Get new game status of the new position

new\_game\_status = game\_state.get\_new\_state(position)

# Call the negamax function to find best path

negaValue, negaMove = negamax(new\_game\_status, depth - 1, -turn\_multiplier, -beta, -alpha)

# If the negamax value has a better score than our highest known score,

# set the bestMove to the current position in the loop

if -negaValue > value:

value = -negaValue

bestMove = position

# Find greatest value between alpha and value

alpha = max(alpha, value)

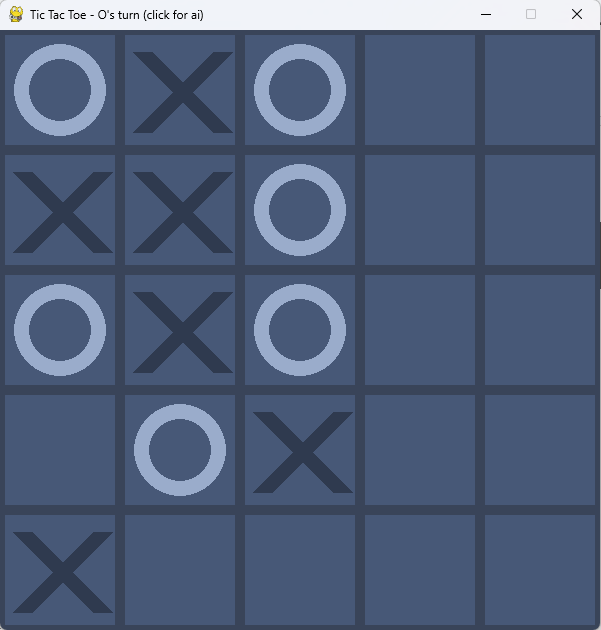
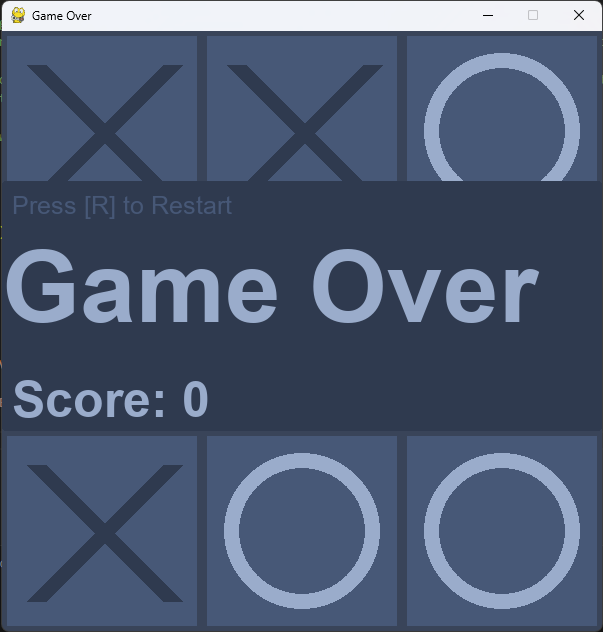
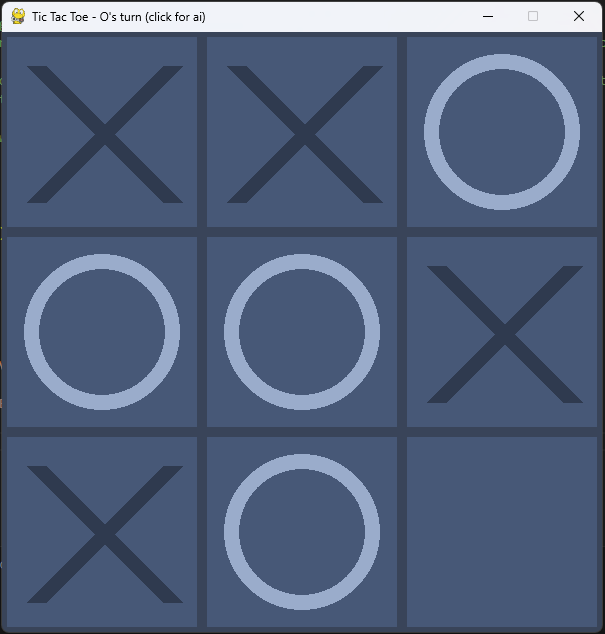
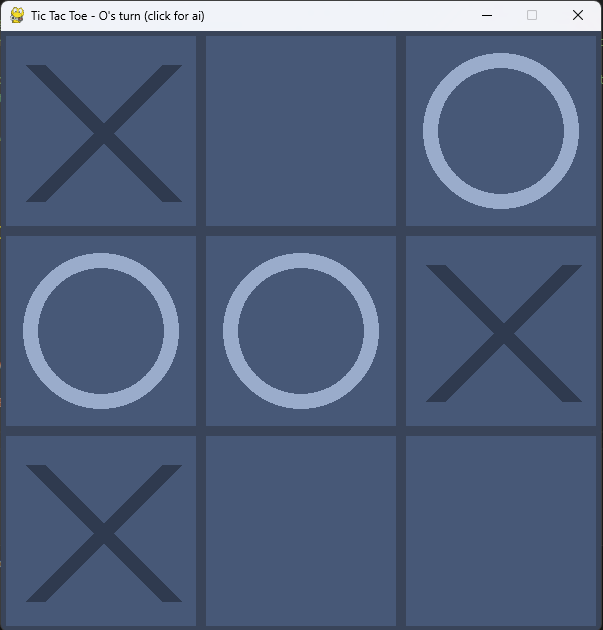
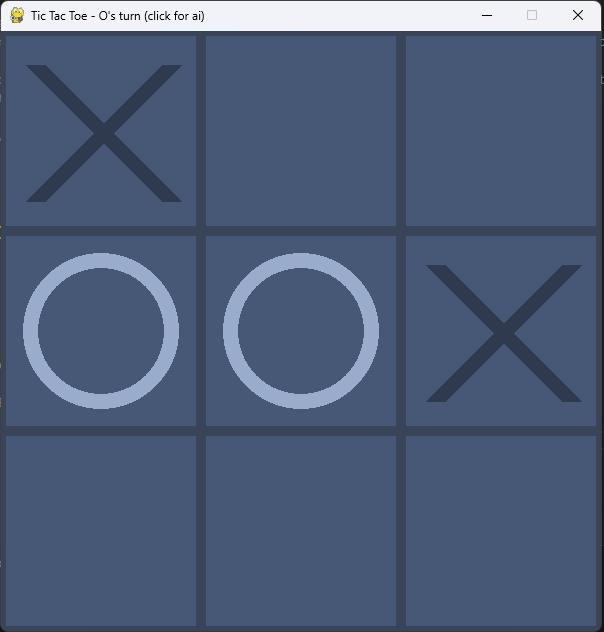
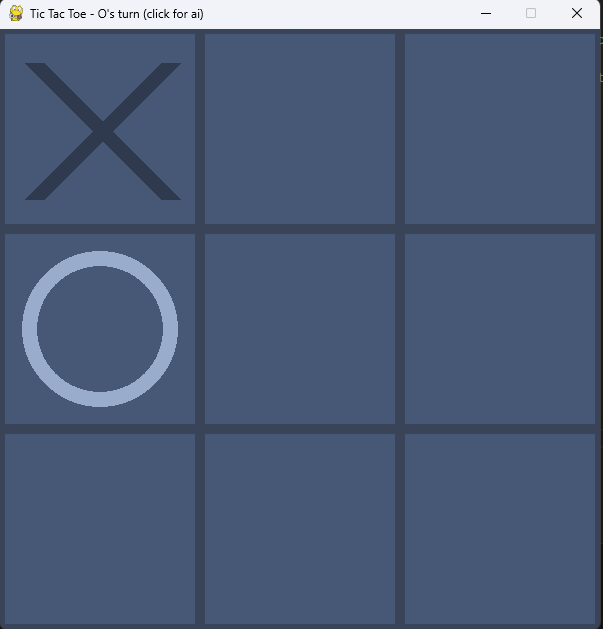
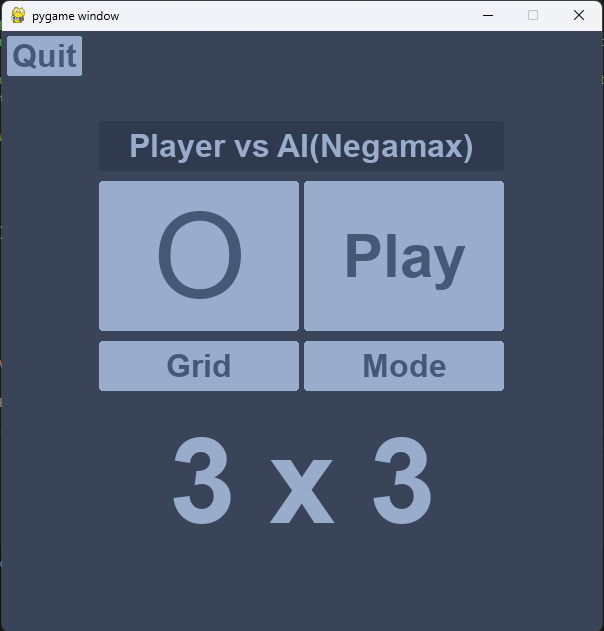
# Prune the branch if alpha is greater than or equal to beta

if alpha >= beta:

break

# Returns the best possible score for the maximizing agent and the corresponding move

return value, bestMove



1. **Constraint satisfaction problems**
2. **(2.5%) Consider the problem of placing k knights on an n×n chess board such that no two knights are attacking each other, where k is given and *k ≤ n2*.**

* **Choose a CSP formulation. What are the variables in your formulation?**

There will be a variable for each of the n2 squares on the board.

* **What are the possible values of each variable in your formulation?**

The possible values for each variable will be occupied or empty.

* **What sets of variables are constrained, and how?**

Every pair of squares separated by the possible knight's move will be constrained so that both cannot be occupied. The entire set of squares is constrained so that only k amount of squares are occupied.

1. **(2.5%) At CSUSB, we have 5 vehicles to take transfer students to a trip to the campus: A, B, C, D, and E and two stops: CGI building and JB Hall. Our job would be to schedule a time slot and a stop for each vehicle to either arrive at or leave the stop. The department gave us four possible time slots: {1, 2, 3, 4} for each stop, during which we can schedule a vehicle to arrive or leave.**

**Constraints:**

* **Vehicle B has lost its battery and must arrive in time slot 1.**
* **Vehicle D can only arrive or leave during or after time slot 3.**
* **Vehicle A is running low on fuel but can last until at most time slot 2.**
* **Vehicle D must arrive before Vehicle C leaves, because some students must transfer from D to C.**
* **Vehicles A, B, and C cater to students from CGI and can only use the CGI stop.**
* **Vehicles D and E cater to students from JB Hall and can only use the JB Hall stop.**
* **No two vehicles can reserve the same time slot for the same stop.**

1. **Formulate this problem as a CSP where there is one variable per vehicle, reporting the domains and constraints (e.g., the time slots are {1, 2, 3, 4} and stop are {CGI, JB Hall}. Also, list binary constraints on the classes. Your constraints should be specified formally, which should be implicit rather than explicit with words.**

Variables = {A, B, C, D, E}

Domains = {1, 2, 3, 4}, {CGI, JB Hall}

Constraints = {

B = 1,

D >= 3,

A <= 2,

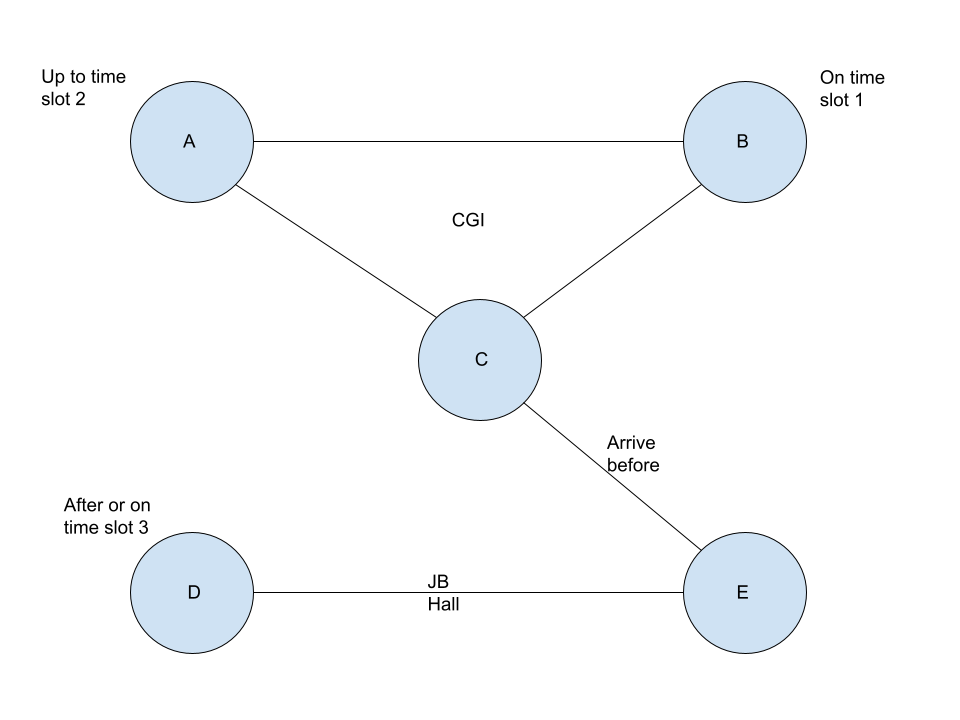
C = D,

A, B, C = CGI

D, E = JB Hall

No two vehicles = timeslot and stop}

1. **Draw the constraint graph for your problem in item 1.**

****