**Homework 2** (15%)

CSE 5120 – Introduction to Artificial Intelligence – Fall 2025

*Submitted to*

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

*by*

Yasmin Darwish (007921609)

Ivan Bonola (007568269)

[Diego Montes-Guzman](mailto:008408524@coyote.csusb.edu)(008408524)

Benjamin Huante (007864370)

Date: *October 19, 2025*

*Email:*

· [*008408524@coyote.csusb.edu*](mailto:008408524@coyote.csusb.edu)

· [*007864370@coyote.csusb.edu*](mailto:007864370@coyote.csusb.edu)

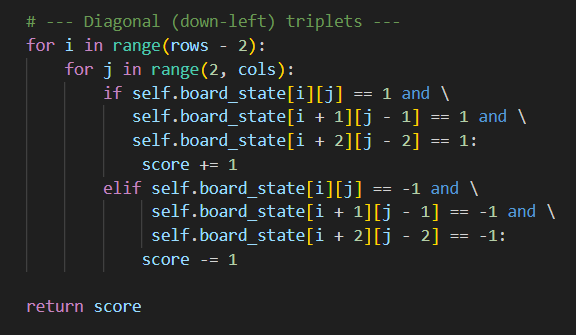
*.* [*007568269@coyote.csusb.edu*](mailto:007568269@coyote.csusb.edu)

*.* [*007921609@coyote.csusb.edu*](mailto:007921609@coyote.csusb.edu)

We all collaborated closely on every aspect of each problem, supporting one another through discussion, feedback, and verification of our solutions. Our teamwork involved constantly checking each other’s logic, sharing ideas, and troubleshooting errors together to ensure that our final results were accurate and well thought out. Although we all contributed to every section, we divided our main responsibilities to make our workflow more efficient. Yasmin and Benjamin focused primarily on the Constraint Satisfaction Problem (CSP), where they handled the formulation of constraints, variable domains, and solution algorithms. Meanwhile, Diego and Ivan concentrated on the Tic Tac Toe problem, taking the lead on implementing and refining the game logic and AI algorithms such as minimax and negamax. This division allowed each pair to develop a deeper understanding of their respective problems while still maintaining strong group collaboration through constant communication and shared problem-solving.

1. Minimax Algorithm with Alpha-Beta Pruning

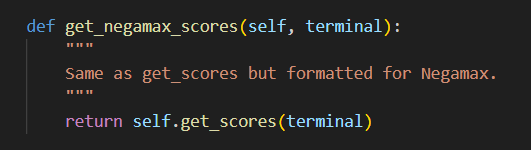
In our implementation, the minimax algorithm is used to make intelligent decisions in a game of Tic Tac Toe by simulating all possible future moves for both players. The function works recursively, exploring each potential move until it reaches a terminal state (a win, loss, or draw) or the specified search depth. At that point, it evaluates the game state using game\_state.get\_scores().



For the maximizing player (in our case, the human playing as O), the algorithm looks for the move that gives the highest possible score by recursively calling itself on all available moves. For the minimizing player (the AI, playing as X), it instead searches for the move with the lowest score, effectively trying to reduce the opponent’s chances of winning. By alternating between these maximizing and minimizing steps, our implementation allows the AI to anticipate the player’s responses and choose the most strategic move possible.

1. Negamax Algorithm

In our implementation, the negamax algorithm serves as a simplified and more elegant version of the traditional minimax approach. Instead of handling separate cases for the maximizing and minimizing players, negamax uses a single unified perspective by introducing a color parameter, which flips the sign of the score to represent the opposing player’s point of view. When the search reaches a terminal state or the maximum depth, the algorithm returns a score adjusted by the player’s color using game\_state.get\_negamax\_scores().



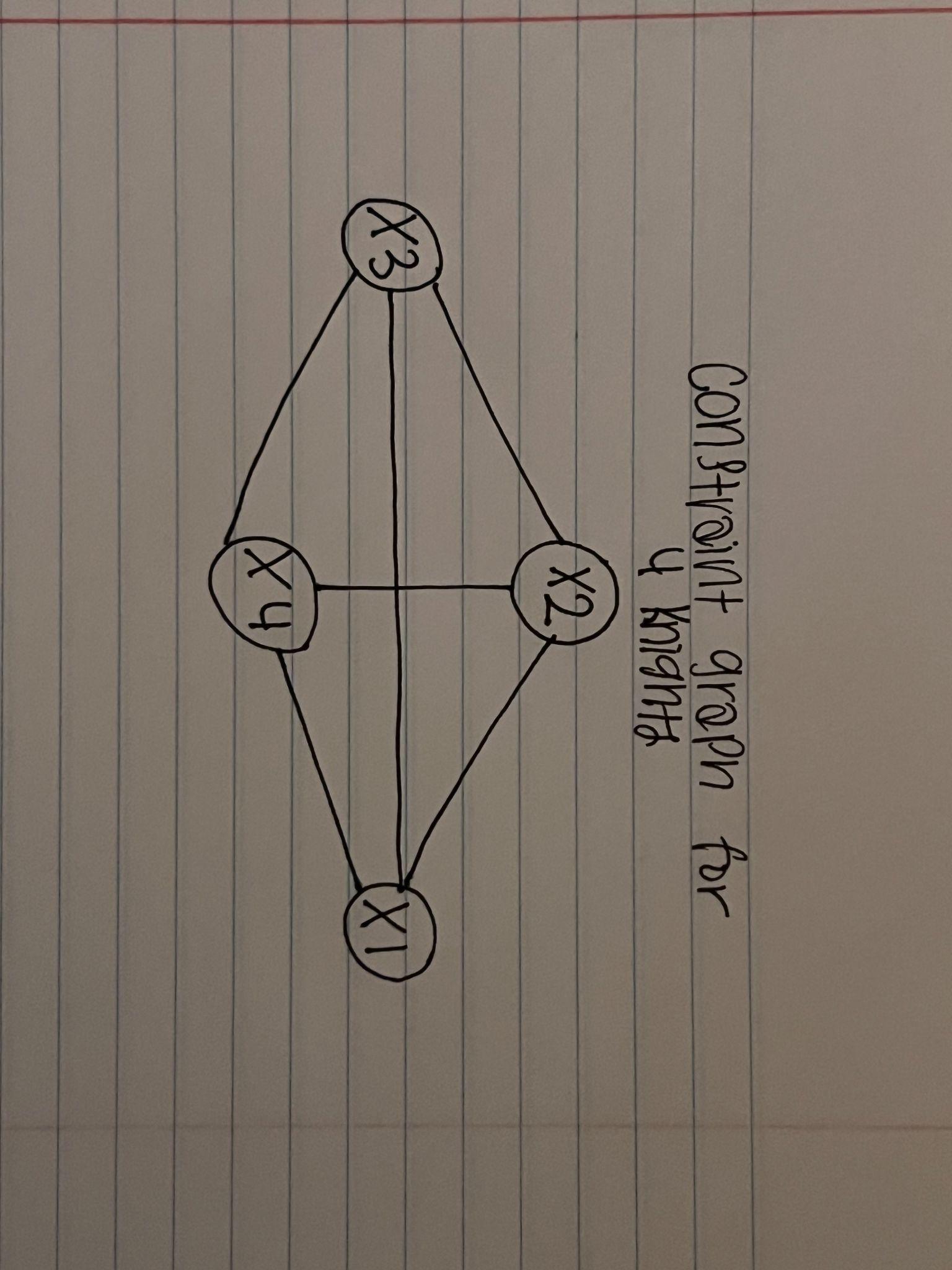
For each available move—determined by our get\_available\_moves() function, which scans the board for empty cells—the algorithm generates a new game state and recursively calls itself with the opposite color. The score from each recursive call is then negated, and the move with the highest resulting value is selected as the best move. This sign-flipping strategy allows us to streamline the logic of minimax into a more concise and efficient implementation while maintaining the same decision-making strength.

1. Constraint Satisfaction Problems

* **1:**
* **Variables**: Xi​ = position (row, column) of the i-th knight, for i=1,...,k
* **Domains:** D(Xi)={(r,c)|1<= r, c, <= n} : all squares of the n board.
* **Constraints)**

1. Xi is not equal to Xj (no two knights on the same square)
2. “|ri-rj| is not equal to 2 or |ci-cj| is not equal to 1, and |ri - rj| is not equal to 1 or |ci - cj| not equal to 2 ( no knight attacks another)

* **Constraint Graph:**



* **2:**

I listed the vehicles as one variable as per vehicle.

Vehicles : {A,B,C,D,E}

The two stops are:

Stops : {CGI, JB Hall}

We are only given 4 time slots:

Time slots : {1,2,3,4}

There only two actions the vehicles can do:

Move : {arrive,leave}

Constraints :

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

Constraints defined Formally :

1. B.stop = (CGI, B.time = 1, B.move = arrive)
2. A.stop = (CGI, A.time <= 2)
3. D.stop = (JB Hall, D.time >= 3)
4. C.stop = (CGI)
5. E.stop = (JB Hall)

Binary constraints :

1. Vehicle B has to be fixed, so (CGI, 1, arrive)
2. Vehicle D must be in Jack brown hall to make sure that it arrives before Vehicle C leaves. (JB Hall, 3, arrive)
3. Vehicle C has to leave at CGI at a different time. (CGI, 4 , leave)
4. Vehicle A has two options. (CGI, 2, arrive) or (CGI, 2, leave)
5. Vehicle E could take any time that hasn’t been taken at JB Hall at time slot 3. So they can either leave or arrive at 1, 2, or 4 time slot

Constraint Graph: