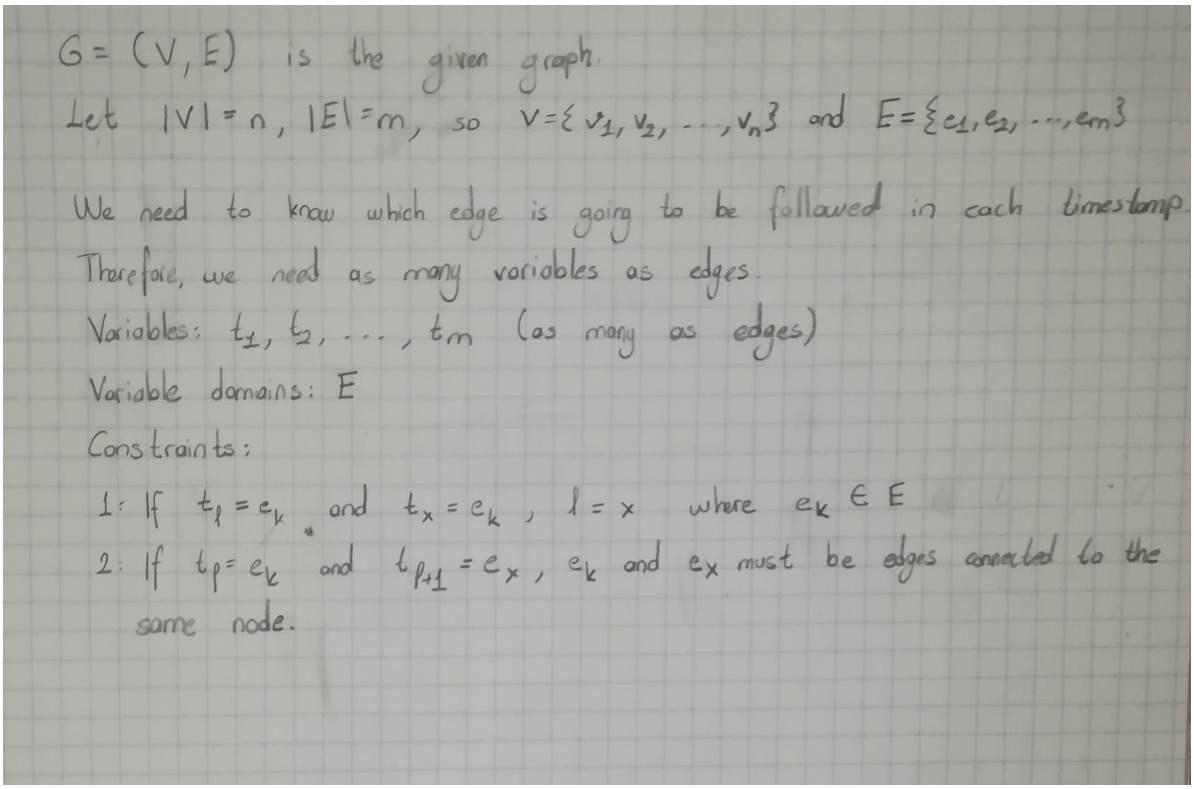
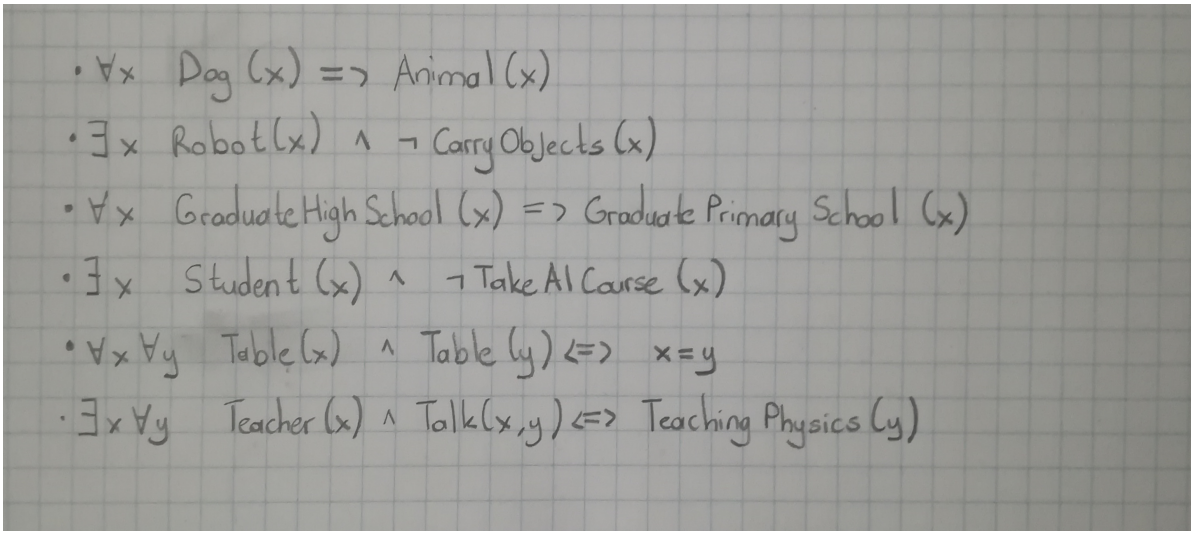
Artificial Intelligence Assignment No 2

Student: Uğur Ali Kaplan
Student ID: 150170042

1)



2)



FOL and Resolution

University(A) ∧ StudentAt(Arda, A) ∧ StudentAt(Cihan, A) ∧ StudentAt(Gamze, A) c_1

∧ Language(English) ∧ Language(French) ∧ Language(Russian) ∧ Language(Turkish) c_2

∧ Speak(Arda, Turkish) ∧ Speak(Cihan, Turkish) ∧ Speak(Gamze, Turkish) c_3

∧ (Speak(Arda, English) ∨ Speak(Arda, French) ∨ Speak(Arda, Russian)) c_4

∧ (Speak(Cihan, English) ∨ Speak(Cihan, French) ∨ Speak(Cihan, Russian)) c_5

∧ (Speak(Gamze, English) ∨ Speak(Gamze, French) ∨ Speak(Gamze, Russian)) c_6

∧ Food(Fish) ∧ Food(Hamburger) ∧ Music(Classical) ∧ Music(Jazz) ∧ Music(Rock) c_7

∧ (¬ Student(x_1) ∨ ¬ Speak(x_1, French) ∨ ¬ (¬ Like(x_1, Jazz) ∨ ¬ Dislike(x_1, Rock))) c_8

∧ (¬ Student(x_2) ∨ ¬ Speak(x_2, Russian) ∨ Like(x_2, Rock)) c_9

∧ (¬ Student(x_3) ∨ ¬ Like(x_3, Hamburger) ∨ Speak(x_3, English)) c_{10}

∧ (¬ Student(x_4) ∨ Like(x_4, Hamburger) ∨ ¬ Speak(x_4, English)) c_{11}

∧ Like(Arda, Jazz) ∧ Like(Arda, Fish) ∧ Dislike(Arda, Classical) ∧ Dislike(Arda, Rock) c_{12}

∧ Dislike(Arda, Hamburger) ∧ Dislike(Cihan, Jazz) ∧ Like(Cihan, Classical) ∧ Like(Cihan, Rock) c_{13}

∧ Like(Cihan, Fish) ∧ Dislike(Cihan, Hamburger) ∧ Like(Gamze, Fish) ∧ Like(Gamze, Hamburger) c_{14}

∧ Like(Gamze, Classical) ∧ Dislike(Gamze, Jazz) ∧ Dislike(Gamze, Rock). c_{15}

Conversions to build the KB:

- $\forall x_1$ Student(x_1) ∧ Speak(x_1, French) ⇒ Like(x_1, Jazz) ∧ Dislike(x_1, Rock)

  → ¬ Student(x_1) ∨ ¬ Speak(x_1, French) ∨ ¬ (¬ Like(x_1, Jazz) ∨ ¬ Dislike(x_1, Rock))
- $\forall x_2$ Student(x_2) ∧ Speak(x_2, Russian) ⇒ Like(x_2, Rock)

  → ¬ Student(x_2) ∨ ¬ Speak(x_2, Russian) ∨ Like(x_2, Rock)
- $\forall x_3$ Student(x_3) ∧ Like(x_3, Hamburger) ⇒ Speak(x_3, English)

  → ¬ Student(x_3) ∨ ¬ Like(x_3, Hamburger) ∨ Speak(x_3, English)

$\forall x_4$ Student(x_4) ∧ ¬ Like(x_4, Hamburger) ⇒ ¬ Speak(x_4, English)

¬ Student(x_4) ∨ Like(x_4, Hamburger) ∨ ¬ Speak(x_4, English)

b)

1) $C_4 \wedge C_9 \wedge C_{11} \wedge C_{12} \wedge C_{13} \wedge \neg Speak(Ardo, French)$

↳ Unsatisfiable because Ardo cannot Speak Russian or English. He has to speak at least 1 foreign language and if he doesn't speak French, we have contradiction. → Return true.

2) $C_5 \wedge C_8 \wedge C_{11} \wedge C_{13} \wedge C_{14} \wedge \neg Speak(Cihan, Russian)$

↳ Unsatisfiable because Cihan cannot speak French or English. But he needs to speak at least 1 foreign language. → Return true.

3) $KB \wedge \neg Speak(Gamze, Russian)$

↳ Satisfied because Gamze dislikes rock and is a student, therefore she cannot speak Russian (C_9). → Return false.

For these queries, I have assumed $\neg Dislike \equiv Like$

3 - Optimal Decision in Games

Running the Program

To run the program, user needs to provide the name of the file with the sudoku grid.

```
python minimax.py input.txt # run minimax
python minimaxprune.py input.txt # run minimax with alpha beta pruning
```

To run the program, following additional packages are required:

- `numpy`

Possible Problem in ITU SSH Server

Number of nodes to be generated might be too much in the ITU SSH server. Process is killed after a while in my own PC. I had to use Google Colab to run the code.

---

a) I have created a `Board` class to store the game state. This class also has a `read_world` method to read the game grid. Method is defined as follows:

```
def read_world(self, file_name):
    with open(file_name, "r") as f:
        for i, line in enumerate(f):
            vals = np.array(line.split())
            self.grid[i] = vals
```

b) I have created a `MaxPlayer` class to implement the minimax search algorithm. Class definition is as follows:

```
class MaxPlayer:
    def __init__(self):
        self.node_count = 1

    def eval(self, level):
        # If level is an even number, MAX Player wins (Last move was made by
the MAX Player)
        eval_v = -1
        if level % 2 == 0:
            eval_v = 1

        return eval_v

    def search(self, board, board_level=1):
        legal_moves = board.get_legal()  # Number of legal moves left

        if len(legal_moves) == 0:
            # If no more legal moves (game ended)
            return self.eval(board_level)
        else:
            # For empty cells that we can write legal numbers in
```

```
            for x, y in legal_moves:
                # For each legal number we can write, create a new state
                for val in legal_moves[x, y]:
                    b = Board(grid=board.grid)
                    self.node_count += 1
                    b.grid[x, y] = val
                    board.eval_v = max(board.eval_v, self.search(b, board_level
+ 1))
            return board.eval_v

    def does_win(self, board):
        board.eval_v = max(board.eval_v, self.search(board))
        return board.eval_v
```

Minimax algorithm is implemented with the `search` method. It requires an object of type `Board` and `board_level` which can be defined as "distance to the root node + 1".

Since we assume max player starts the game and the first state has `board_level = 1`, this means MAX player wins if board_level is even. This can be seen in the `eval` method.

Legal moves are returned by the `Board` class and it is as follows:

```
    def get_legal(self):
        if self.legal_run:
            return self.legal_moves
        self.legal_run = True
        legal_pos_x, legal_pos_y = np.where(self.grid == 0)
        num_pos = legal_pos_x.shape[0]
        legal_moves = {(legal_pos_x[i], legal_pos_y[i]): [1, 2, 3, 4, 5, 6, 7,
8, 9] for i in range(num_pos)}
        for i in range(num_pos):
            x, y = legal_pos_x[i], legal_pos_y[i]
            x0, x1 = find_box(x)
            y0, y1 = find_box(y)
            box = self.grid[x0:x1, y0:y1]
            for num in range(1, 10):
                if (num in self.grid[x]) or (num in self.grid[:, y] or (num in
box)):
                    legal_moves[x, y].remove(num)
            if len(legal_moves[x, y]) == 0:
                del legal_moves[x, y]

        self.legal_moves = legal_moves
        return self.legal_moves
```

To find out the borders of the boxes, I have also defined the following static function:

```
def find_box(x):
    coor = []

    if x % 3 == 0:
        coor.append(x)
        coor.append(x + 3)
    if x % 3 == 1:
        coor.append(x - 1)
        coor.append(x + 2)
    else:
        coor.append(x - 2)
        coor.append(x + 1)
    return coor[0], coor[1]
```

Using these methods, board object returns possible legal moves as a dictionary. Then, we create new boards for each possible next state. If there are no legal moves left, one of the players has won and we can run the eval method. Value is propagated to the root node with recursion.

c) I did not want to change my original `MaxPlayer` implementation, therefore I have created a new class called `MaxPlayerPrune`. Class definition:

```
class MaxPlayerPrune:
    def __init__(self):
        self.node_count = 1

    def eval(self, level):
        # If level is an even number, MAX Player wins (Last move was made by
the MAX Player)
        eval_v = -1
        if level % 2 == 0:
            eval_v = 1

        return eval_v

    def search(self, board, board_level=1):
        legal_moves = board.get_legal()  # Number of legal moves left

        if len(legal_moves) == 0:
            # If no more legal moves (game ended)
            return self.eval(board_level)
        else:
            # For empty cells that we can write legal numbers in
            for x, y in legal_moves:
                # For each legal number we can write, create a new state
                for val in legal_moves[x, y]:
                    b = Board(grid=board.grid)
                    self.node_count += 1
                    b.grid[x, y] = val
                    board.eval_v = max(board.eval_v, self.search(b, board_level
+ 1))
                    if board.eval_v == 1 and board_level % 2 != 0:
                        return board.eval_v
                    elif board.eval_v == -1 and board_level % 2 == 0:
                        return board.eval_v
            return board.eval_v
```

```
    def does_win(self, board):
        board.eval_v = max(board.eval_v, self.search(board))
        return board.eval_v
```

For this one, I have simply changed the for loop in the `search` method and made the loop stop if a player has found a winning path (concept of winning path changes depending on if the deciding player is the MAX player or the MIN player):

```
for val in legal_moves[x, y]:
    b = Board(grid=board.grid)
    self.node_count += 1
    b.grid[x, y] = val
    board.eval_v = max(board.eval_v, self.search(b, board_level + 1))
    if board.eval_v == 1 and board_level % 2 != 0:
        return board.eval_v
    elif board.eval_v == -1 and board_level % 2 == 0:
        return board.eval_v
```

d) I probably should have used C++, but I have used Python and paid for it with time. These results are calculated on the `input_2.txt`, as it was the only one given to us and it had 12 empty cells. If I run the minimax algorithm on my own machine, process gets killed after a while. I have run it in Colab but got a timeout. I had to stop it early to see the number of created nodes.

- Created Number of Nodes
  - Minimax explores all possible paths. Therefore, it should $12! = 479001600$ nodes. However, I was only able to see up to 192943045 nodes.
  - Minimax with Alpha Beta Pruning is easier on the computer, it created 105211 nodes with my code.
- Runtime
  - Minimax which created 192943045 nodes runs in about 3-4 hours. I would assume if I was able to run it for all the nodes, it would take around 9-10 hours.
  - Minimax with Alpha Beta Pruning runs in about 10-15 seconds in Colab.