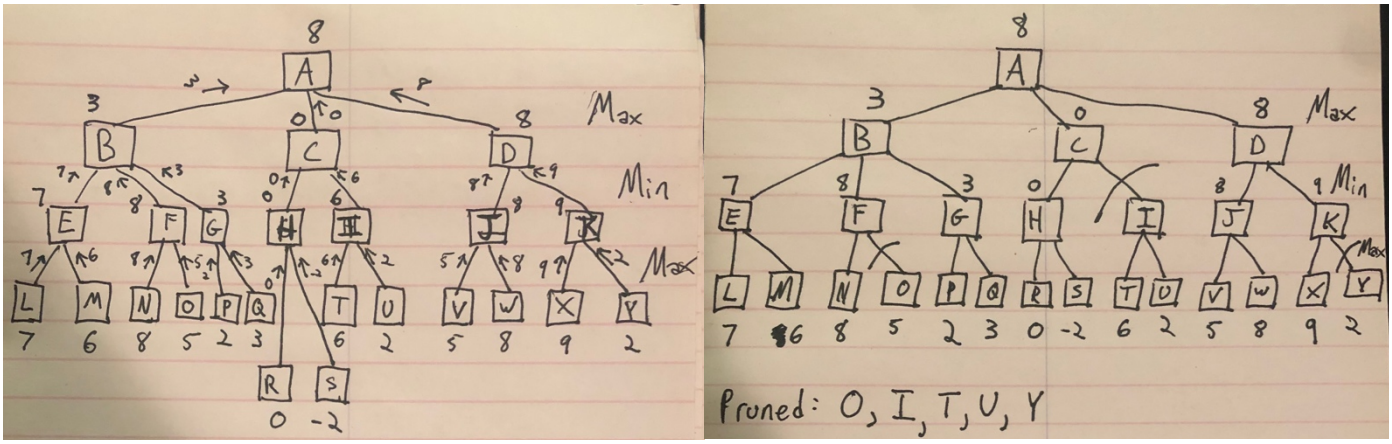
Caden Kroonenberg
Problem Set 5
EECS 649
2-23-22

5.1
Forward search is required because in most games, the player is not entirely in control of the environment and therefore the sequence of actions required to get from the initial state to the goal state requires actions that the player cannot choose. For example, in chess you couldn't start at the end state because you don't know what moves your opponent will make. Likewise, in Texas hold 'em you cannot start at the end state because you don't know what cards the dealer will place on the table.

5.2
Texas hold 'em:
State descriptions: States can be described by the number of remaining players, the amount of money in the pot, whose turn it is, the current bet, the player's hand, the community cards, and previous actions in the round of betting (if your bet has been raised, etc.).
Move generators: Possible moves are fold, check, bet, call, and raise.
Terminal tests: True if only one player hasn't folded or if all five community cards have been placed and betting has stopped.
Utility functions: The utility is 1 if the player hasn't folded and has the strongest hand of all remaining players. If the player either folds or doesn't have the strongest hand of all remaining players, the utility is 0. You could potentially make a utility function which takes the amount of money won into account as well. In this case, the utility would be the difference in money from the beginning of the game.
Evaluation functions: This function would return a value proportional to the strength of the player's hand (including the community cards).

5.3

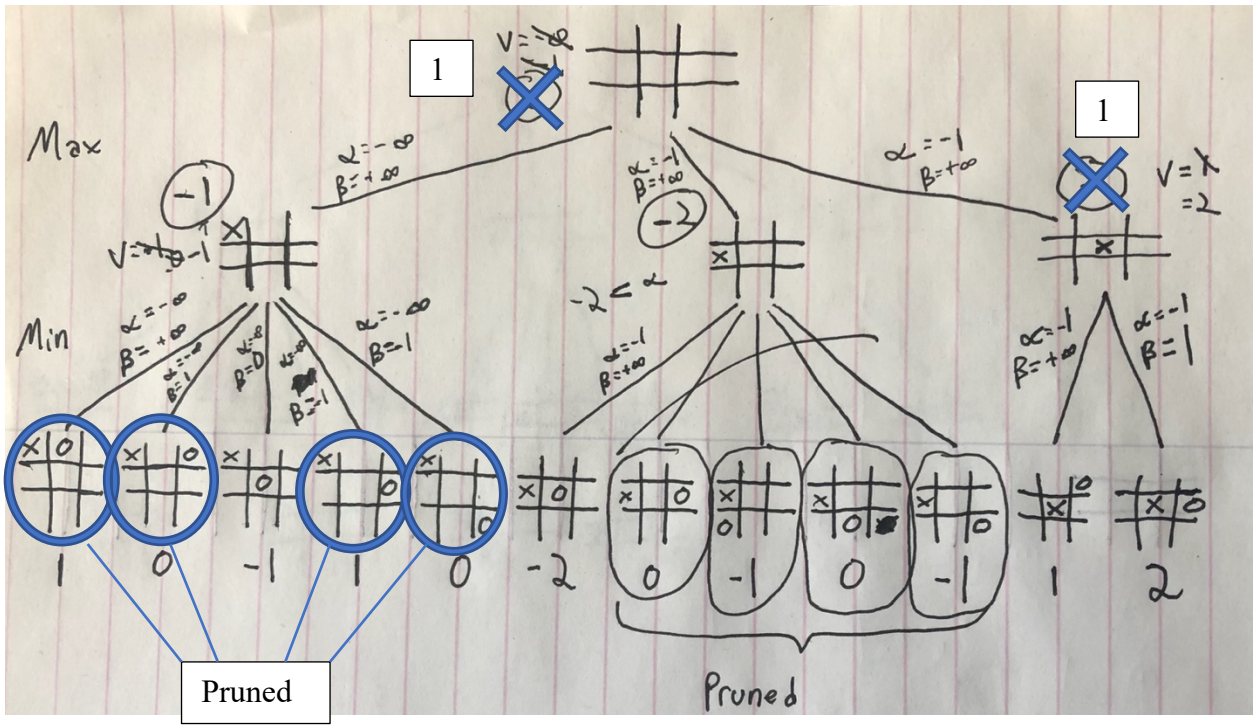


a. The player should move to state D. The minimum payoff the player is assured from this action is 8.
b. Pruned Nodes: O, I, T, U, Y

5.4

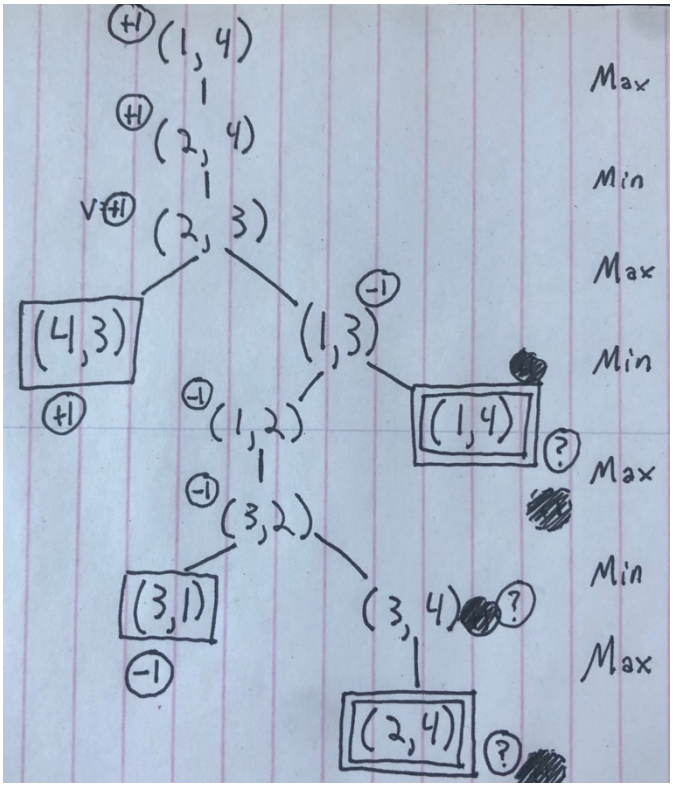
a. ~9!

b – e:



5.5

a.



b. I treated (3,4)'s value as "?" because it had no other successors. Then the only nodes which had a "?" were min nodes. In these cases, you can use the rule min(-1, ?) = -1.

c. The algorithm would loop infinitely because it's depth-first search. If the value of loop states is "?" with the rule that min(-1,?) = -1 and max(+1,?) = +1, no looping will occur. This would give optimal decisions because given winning and looping, the algorithm will choose to win. For zero-sum games, this should be the optimal solution as winning is prioritized and losing is avoided.

d. For n > 2, if n is even, A will win if it always moves forward because when A and B meet, A will have the next move, so it can jump over B. The opposite holds for B winning when n is odd.

5.6

a.

```
# Author: Caden Kroonenberg
# Date: 2-18-22

import random

def fitness(queens, N):
    c = 0 # Number of conflicts
    # For each queen
    for q in range(len(queens)):
        for q_y in range(len(queens)):
            q_x = queens[q_y]
            if q_y != q: # Check all other queens
                if q_x == queens[q]: # If queens are in same column
                    c += 1
                if abs(q - q_y) == abs(queens[q] - q_x): # If queens share a diagonal
                    c += 1
    return (N*(N-1))/2 - c

# choose successor based on min-conflict method
def successor(queens, N):
    global rand_choice
    global last_q
    if rand_choice: # Random choice queen
        q = random.choice(queens)
    else:           # Cyclic choice queen
        last_q = (last_q + 1) % N
        q = last_q

    successors = []     # All successor states for queen q
    for i in range(N):  # Each possible positon for q
        qc = queens.copy()
        qc[q] = i
        successors.append(qc)
    best_f = -1         # Fitness of best successor
    for s in successors:
        f = fitness(s,N)
        if f > best_f:
            best_s = s  # Update best successor
            best_f = f  # Update best fitness
    return best_s

rand_choice = False # Choose random queen to minimize conflict; cyclic if False
last_q = -1 # Last evaluated queen (for cyclic queen selection)
N = 8       # Grid size, number of queens
```

```
# Random initial state
queens = []
for n in range(N):
    queens.append(random.randint(0,N-1))

for i in range(25): # Cutoff after 25 epochs – likely to fail for i > 25
    queens = successor(queens,N) # Update queen positions
    if fitness(queens,N) == (N*(N-1))/2: # Test for success
        print('{} epochs\t{}'.format(i, queens))
        break
```

c.

Implementation	Average # of epochs	Average # of evaluations
Random variable selection (including failed attempts)	188.84	1699.56
Random variable selection (excluding failed attempts)	13.59	131.31
Cyclic variable selection (including failed attempts)	100.89	908.01
Cyclic variable selection (excluding failed attempts)	12.39	120.51