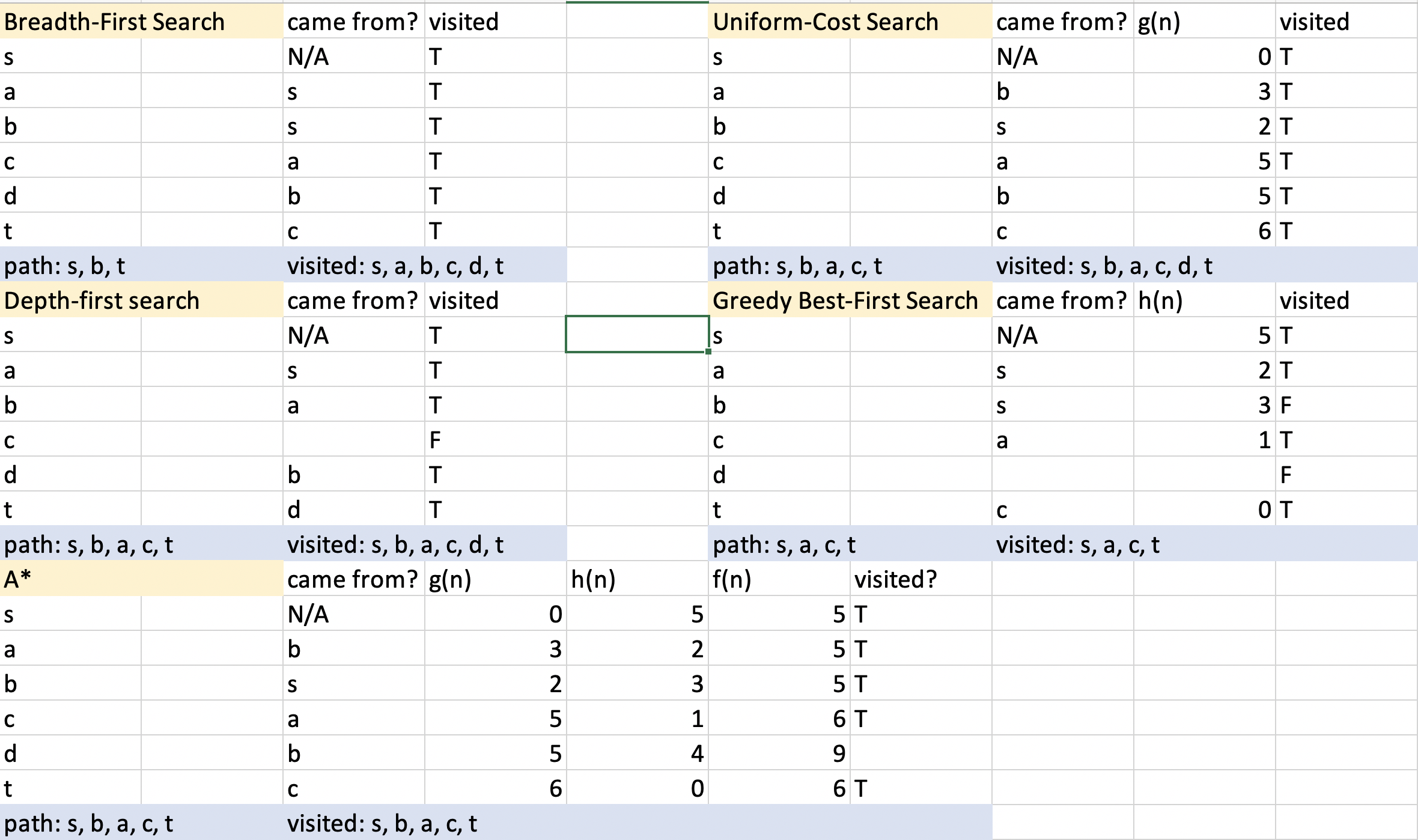
**Artificial Intelligence**

**Andrew Morrison**

**Homework 2 Part 1:**



**Homework 2 Part 2:**

Strategy:

Points will be used to track the value of each space. The main strategy was cutting down time-cost to allow for more generations to be checked. The values of the initial board are slightly higher as they increase vertically, as well as towards the middle. Whenever a move is made, I will calculate the change made to the board by that piece, and change the value of the board accordingly. This way avoids re-calculating the cost of every space in the board.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 6 | 7 | 8 | 9 | 8 | 7 | 6 |
| 5 | 6 | 7 | 8 | 7 | 6 | 5 |
| 4 | 5 | 6 | 7 | 6 | 5 | 4 |
| 3 | 4 | 5 | 6 | 5 | 4 | 3 |
| 2 | 3 | 4 | 5 | 4 | 3 | 2 |
| 1 | 2 | 3 | 4 | 3 | 2 | 1 |
| 0 | 1 | 2 | 3 | 2 | 1 | 0 |

The figure above shows my proposed initial configuration of the board scores. The main heuristic for each board will be the combination of all the values. Relevant spaces will be re-evaluated as pieces are placed. Spots near one friendly piece giving some points, two giving more, and three giving 10,000 points to signal the end of the game (these rules change based on the connect-N size). Enemy pieces will subtract points from the board in the respective situations.

Another possible heuristic will evaluate specifically for win-win conditions.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  | w |  |  |
|  | x | x | x | w |  |  |
|  | o | x | o | o |  |  |
| o | x | x | o | o |  |  |

The example board above has a win-win condition for x. If an o is placed in column 5, x will win the next turn by placing another piece in column 5. If an o is not placed in column 5, then x will win by placing a piece in column 5. The heuristic will raise the value of spots based on the ability to create a situation like this. It will basically check for possibilities to combine a horizontal and vertical (diagonal or not) row of friendly pieces in the same way as the example board.

Pseudocode for additional functions:

class spot: # this class represents individual spots on the board

score = 0

playerPiece = 0

def \_\_init\_\_ spot(s, p):

int score = s # score of the spot

int playerPiece = p # the piece in that spot (0 for none, 1 for player 1, 2 for player 2)

def \_\_init\_\_ node: # each node is a board, and this will be used to keep track of the

previous board (parent) Old boards will be deleted to save memory

spot[][] layout

node parent

node[] children

def initialHeuristics(board) # creates the initial point layout of the board

for r in board[r][c]

for c in columns[r][c]

check for surrounding pieces (n up, n horizontal, n down, n diagonal up, n diagonal down)

add score to board spot

score = calcScore(board)

return score

def calcScore(board)

sum = 0

for r in board[r][c]

for c in board[r][c]

sum += board[r][c].score

return sum

def mainHeuristics(board, column)

score = 0

check n in all directions of last move (given by get successors function)

change points accordingly

calcScore(board)

compare changed spots with old spot points from the previous board and add or subtract points from the total score

return score