**ECE 457A**

### COOPERATIVE AND ADAPTIVE ALGORITHMS

**Assignment 1: Report**

**By:**

Shweta Aladi (saladi)

Bilal Arshad (barshad)

Pooja Sardesai (pmsardes)

**Due:**

May 31st, 2012

**Contributions**

Question 1: Shweta - Breadth First Search, Created and added methods to Node class  
         Bilal - Depth First Search, Display method  
 Pooja - A\* Search, Added methods to Node class  
  
Question 2: Shweta – Minimax Search, Gameboard class, Gamepiece class  
         Bilal – Alpha Beta Search, Minimax Search  
 Pooja - Display method  
  
Question 3: Shweta - Aspiration list  
 Pooja - Tabu Search, Created and added methods to Pair class

*Each team member also wrote the accompany documentation for their respective pieces of code. The design of each algorithm was discussed as a group, but implementation and testing was done independently.*

**Question 1 - Maze**

**Program Description**

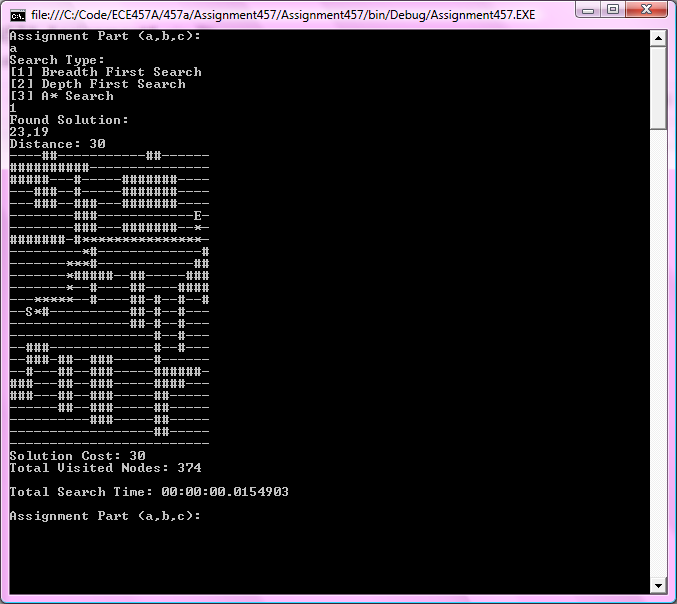
This program solves a maze by looking for the shortest path between the Start and End nodes, if a solution exists. The maze was mapped to a tree with each ‘square’ representing a node in the tree (implemented in the Node class), and the Start node acting as the root.

**Implementation Details**

**Breadth First Search**

The Breadth First Search searches all the nodes on one level of the tree before progressing to the next level. In this way, the search always guarantees that the solution found has the lowest cost, though it may not be unique. The disadvantage of this approach is that it requires more intensive traversal and requires all the nodes of the tree to be stored as it is being searched.

CREATE a queue (node\_queue) to store unvisited nodes  
CREATE a list (duplicate\_list) to track unique nodes  
SET the current node to the start node  
PUSH the current node onto node\_queue  
PUSH the current node onto the duplicate list  
  
WHILE(true)  
   IF (node\_queue is not empty) then  
   the next\_node to check is the first node in node\_queue  
   ELSE  
   break out of the while loop (no more nodes to check)  
   END  
  
   SET current\_node to next\_node  
   IF(current\_node is the end node) then  
   MARK the solution path by iterating through the current\_node's     parents  
   CALCULATE the distance (number of steps) from start to end  
   RETURN true because a solution has been found  
   ELSE  
   MARK current node as visited  
   FOR(each of the four directions - up, down, right and left)  
   GET x,y coordinates  
   CREATE new child nodes (child\_node)  
     
   IF (child\_node are not out-of-bounds or blocked)  
   ADD child\_node to parent (current\_node)  
   IF (child\_node is not in duplicate\_list)  
   ADD child\_node to duplicate list  
   ADD child\_node to node\_queue  
   END  
   END  
   END  
     END  
   END   
   RETURN false because no solution was found   



**Figure 1.1:** Sample Output for Breadth First Search: Start(2,11) and Exit(23,19)

**Depth First Search**

The pseudocode below is a regular implementation of depth first search in order to find the path to the end node. This done, by continuously checking if your child node is the solution until an end condition is met (node is the solution, already visited, blocked or out of bounds). If the node is already visited, blocked or out of bounds then the algorithm checks another child node. If there are no more child nodes, it either returns true if a solution was found to its parent or false if it wasn’t.  Once a solution is found, the search algorithm does not try to find another and hence not guaranteeing the optimal solution.

DepthFirstSearch ( Node , Reference Count, Reference NodeList )  
 IF Node is END then  
 return TRUE //Solution Found  
 END  
 MARK Node as VISITED in NodeList   
 FOR(INCREMENT i FROM 0 to 3)  
 CREATE new child node (child\_node) set to NULL  
 SWITCH (i)  
 CASE 0: // EAST  
 IF NOT out of bounds AND NOT blocked node

set child\_node to node EAST of Node

ADD child\_node as CHILD of node

END

CASE 1: // WEST  
 IF NOT out of bounds AND NOT blocked node

set child\_node to node WEST of Node

ADD child\_node as CHILD of node

END

CASE 2: // NORTH  
 IF NOT out of bounds AND NOT blocked node

set child\_node to node NORTH of Node

ADD child\_node as CHILD of node

END

CASE 3: // SOUTH  
 IF NOT out of bounds AND NOT blocked node

set child\_node to node SOUTH of Node

ADD child\_node as CHILD of node

END

END  
 IF child\_node IS NOT NULL  
 IF CHILDNODE is NOT marked as VISITED in NodeList  
 IF DepthFirstSearch ( child\_node , Count, NodeList )

MARK Node as SOLUTION in NodeList

increment Count

return true

ELSE

REMOVE  child\_node as CHILD of node

END

ELSE

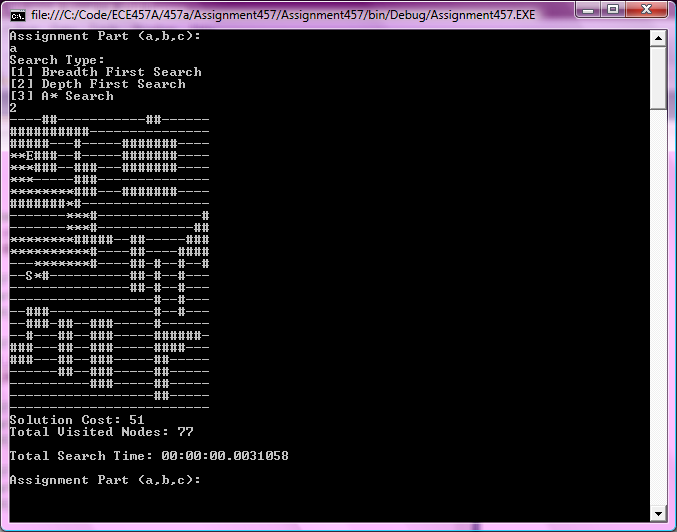
REMOVE  child\_node as CHILD of node

END

END  
 END

return false //Solution not found

**Figure 1.2:** Sample Output for Depth First Search: Start(2,11) and Exit(2,21)



**Optimized Depth First Search**

This search is a modified version of the depth first search described above. This solution attempts to find the optimal solution and consistently provides better results in terms of Path Cost when compared to the depth first search solution. This is done by not stopping after finding a solution (like in regular Depth First Search). As the algorithm returns the count value up the ‘tree’ or solution path, it checks whether there is a more optimal solution from the current node, than the one that was found.

OptDepthFirstSearch ( Node , Reference NodeList )  
 IF Node is END then  
 **return 1**//Solution Found  
 END  
 MARK Node as VISITED in NodeList  
 SET size to arbitrarily large number (signifies distance between node and endpoint)  
 CREATE new child node (chosen\_null) set to NULL  
 FOR(INCREMENT i FROM 0 to 3)  
 CREATE new child node (child\_node) set to NULL  
 SWITCH (i)  
 CASE 0: // EAST  
 IF NOT out of bounds AND NOT blocked node

set child\_node to node EAST of Node

END

CASE 1: // WEST  
 IF NOT out of bounds AND NOT blocked node

set child\_node to node WEST of Node

END

CASE 2: // NORTH  
 IF NOT out of bounds AND NOT blocked node

set child\_node to node NORTH of Node

END

CASE 3: // SOUTH  
 IF NOT out of bounds AND NOT blocked node

set child\_node to node SOUTH of Node

END

END  
 IF child\_node IS NOT NULL  
 IF CHILDNODE is NOT marked as VISITED in NodeList  
 IF ( DepthFirstSearch ( child\_node, NodeList ) is NOT equal to -1  
 IF Count is LESS THAN size  
 SET size to count  
 SET chose\_child to child\_node  
 ELSE IF  Count is EQUAL THAN size  
 MARK Node as VISITED POTENTIAL SOLUTION in NodeList

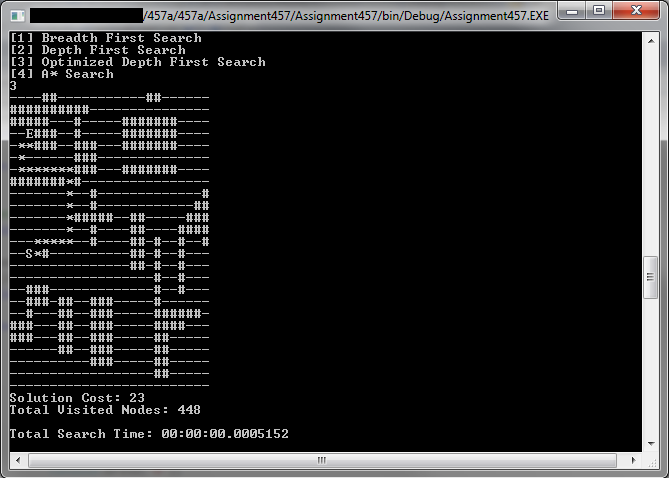
END

END

END

END  
 END  
 IF chosen\_child is not equal to Null  
 Set Node as SOLUTION in NodeList  
 Add chosen\_child as a child to current node  
 return size + 1  
 END

return -1//Solution not found



**Figure 1.3:** Sample Output for Optimized Depth First Search: Start (2,11) and End(2,21)

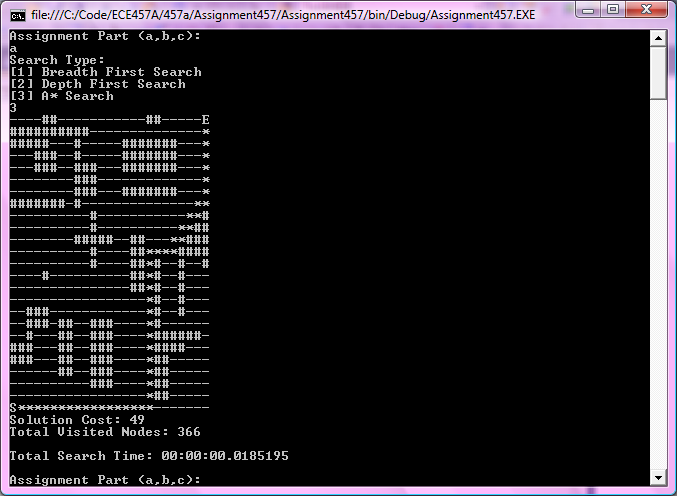
**A\* Search**

The A\* search algorithm is used to find the optimal path from the start node to the end node. Several lists were created to keep track of specific types of nodes, such as blocked nodes, “to be explored” nodes (stored in a list called openList) and fully explored nodes (stored in a list called closedList).

The algorithm begins by first exploring the start node. It evaluates all of its child nodes, which are located north, south, east and west of the start node. If the child node is invalid, it is ignored because it is either blocked, out of bounds or it has already been explored. If the node is valid, it is added to openList, so that the node can be explored in the future. If the node is already in the openList, the G cost of the new and old node is compared. The node will be updated with the lowest G cost and then put back into openList. Once all the child nodes have been added to openList, the start node is placed in the closedList, indicating that the node has been fully explored and cannot be explored further in the future. The openList is then sorted by the lowest F cost. Next, the first node from the openList (the node with the lowest F cost) is selected to be explored. This process is repeated until the end node is found. If the end node is not found or if openList is empty, the solution does not exist.

The function used for the A\* search is F = G + H, where F is the distance from the start node to the end node, G is the distance from the start node to the current node and H is the estimated distance from the current node to the end node. By following this heuristic function, the search algorithm was able to determine the optimal (shortest) path from the start node to the end node.

**Method:** AStarSearch   
Set number of nodes explored to 0  
Set the start node as the current node  
CREATE list (called openList) to store nodes to be explored  
Add current node to the openList so that it is explored  
  
WHILE (there are nodes to be explored)  
 Set current node to the first node in openList  
 Remove first node from openList because it is being explored  
 Mark current node to being explored  
 IF (current node is the end node) then  
   Calculate the path cost  
   Mark nodes that are on the solution path  
   Mark the end node  
   RETURN true because a solution has been found   
 ELSE  
   Get current node's coordinates  
   Create four child nodes: located north, south, east and west of the current node  
   Calculate the estimated path costs of all child nodes  
   Add child nodes to openList so that they can be explored  
   Order list by the lowest path cost (F)  
 END  
END  
  
RETURN false because a solution cannot be found  
  
**Method:** addToOpenList Method  
Get child node's coordinates  
IF (child node has already been explored or is blocked or is out of bounds)  
 Ignore  
END  
  
IF (child node is already in the openList)  
 Calculate new path cost from start node to current node  
 IF (new path cost is lower)  
   Update child node with new path cost  
ELSE   
 Add child node to openList so that it can be explored  
 Mark child node as explored  
 Increment number of nodes explored counter  
END  
  
**Method:** updateNodeInList  
Get node from list  
Remove node from list  
Update G and F values of the node   
Update parent of node  
Add updated node to list  
  
**Method:** markSolutionPath  
WHILE (current node is not null)  
 Mark node because it is on the solution path  
 Reset parent pointer to current nodes parent  
END



**Figure 1.4:** Sample Output For A\* Search: Start (0,0) and End (2,24)

**Complexity**

|  |  |  |
| --- | --- | --- |
|  | **Time Complexity** | **Space Complexity** |
| **Breadth First Search** | O(3^d) where d is depth, and breadth is 3 | O(3^d) where d is depth, and breadth is 3 |
| **Depth First Search** | O(3^d) where d is depth, and breadth is 3 | O(3+d) where d is depth, and breadth is 3 |
| **Optimized Depth First Search** | O(3^d) where d is depth and breadth is 3 | O(3+d) where d is depth and breadth is 3 |
| **A\* Search** | O(3^d) where d is depth, and breadth is 3 | O(3^d) where d is depth, and breadth is 3 |

**Test Case 1:** Start Node (2,11) -> End Node (23,19)  
**Test Case 2:** Start Node (2,11) -> End Node (2,21)  
**Test Case 3:** Start Node (0,0) -> End Node (24,24)  
  
Visited Nodes (Nodes)

|  |  |  |  |
| --- | --- | --- | --- |
|  | Test Case 1 | Test Case 2 | Test Case 3 |
| Breadth First Search | 374 | 260 | 448 |
| Depth First Search | 100 | 77 | 105 |
| Optimized Depth First Search | 448 | 448 | 448 |
| A\* Search | 113 | 96 | 336 |

Timing Analysis (s)

|  |  |  |  |
| --- | --- | --- | --- |
|  | Test Case 1 | Test Case 2 | Test Case 3 |
| Breadth First Search | 0.0774985 | 0.0058160 | 0.0103001 |
| Depth First Search | 0.0059427 | 0.0015360 | 0.0013491 |
| Optimized Depth First Search | 0.0022011 | 0.0018756 | 0.0019362 |
| A\* Search | 0.4161943 | 0.0060766 | 0.0088966 |

Cost (Nodes)

|  |  |  |  |
| --- | --- | --- | --- |
|  | Test Case 1 | Test Case 2 | Test Case 3 |
| Breadth First Search | 30 | 21 | 49 |
| Depth First Search | 54 | 51 | 103 |
| Optimized Depth First Search | 30 | 23 | 57 |
| A\* Search | 30 | 21 | 49 |

**Question 2 - Conga Game Play**

**Program Description**

The program simulates a two-player game between a Random Agent (Player 1) and the Computer (Player 2). The computer tries to win against the random agent (used to simulate a real-life player) using the Minimax and Alpha-Beta Pruning algorithms.

For the purpose of this program, the random agent is using the BLACK pieces (///10////), and the computer is using the WHITE pieces (...10...). The output of each move is shown on the console, along with the current move number. The program allows a maximum of 5000 moves before declaring a stalemate.

The evaluation function used the number of possible moves each player can make based on the current gameboard. The Minimax algorithm allows Player 2 to choose the best move that will lead to Player 1 not being able to make any more moves (the condition for Player 1 to win). The Max moves take next potential moves that give the most number of possible moves for the next turn. Conversely, the Min moves take next potential moves that give the least number of possible moves for the next turn. This version of Minimax is the Iterative-Deepening Minimax. It increments the depth of the tree of possible moves until a leaf node is reached - a player no longer has any moves (a win condition). Player 2 will take the next move on the path that will lead to Player 1 having no moves. Player 1 makes random moves out of the list of possible moves.   
  
The Alpha-Beta Pruning algorithm allows Player 2 to choose the best move based on an evaluation of the gameboard. The evaluation is based on how many children the gameboard has. The algorithm is currently limited to a certain depth. Upon reaching the depth or reaching a node in which the game ends, the algorithm returns the evaluation of the current node down the stack (recursively). This allows for alpha/beta comparisons between various min and max nodes and pruning of the game tree. The removal of a child node due to a more optimal child having being found allows for a more memory efficient search over minimax, while still providing similar results.

**Description of Implementation of the Random Search Agent**

The opponent, Player 1’s moves were generated randomly. First, a list of valid moves was generated from the current Gameboard. Then a random number is generated using the System.Random class. The random number generates an integer between 0 and the number of children -1. The child node corresponding to the random number is chosen from the list of children and implemented as the next move.

**Analysis of Evaluation Functions**

Evaluation Function 1: Consider number of squares covered by each player

This function was not implemented because each player can cover a maximum of 10 squares each (out of a combined total 16 squares). This will not work because a player can make moves onto the squares containing stones of the same colour - therefore the game can go on indefinitely with players just moving stones between squares that have the same colours. This implementation did not lead to either player winning, and therefore consistently resulted in a stalemate.  
  
Evaluation Function 2: Consider the number of possible moves each player can make based on the current game

Each gameboard generates all possible moves as child nodes. Therefore, the number of possible moves is the number of child nodes of the current move. For min nodes, we use the child node that gives the least possible number of moves. Conversely, for max node, we use the child node that gives the maximum possible next moves. Ideally the Minimax should try to get the min node to return zero moves - this is the case where Player 1 (the random agent) loses to the Player 2 (the Max player using the Minimax algorithm).

Evaluation Function 3: Consider the number of moves not blocked by the opponent

This evaluation function would consider the total number of moves available to the player making the move, and then subtract the number of moves that blocked due to the opponent or the boundaries of the Gameboard. This evaluation function was not used because it was more complicated to implement correctly, and took longer to run without yielding any better results than those returned by the second evaluation function.

Finally, evaluation function was implemented because it returned good results within a reasonable amount of time and was not difficult to implement.

**Key Moves**

**Minimax**

Refer to Figure 2.1 for the following section.

**Key Move #331:** Player 2 has completed their turn

Player 1’s pieces at (2, 3) and (3, 3) can either move to each other’s square or the 4 stones at (3, 3) can move one square down to the square at (3, 2). This move is seen by Player 2 and blocked in this move by moving 2 white pieces from (3, 1) to (3, 2). This effectively traps the remaining white pieces.

**Key Move #332:** Player 1 has completed their turn

Player 2 has only 2 options – either move the 6 pieces from (2, 3) to (3, 3) or move the 4 pieces from (3, 3) to (2, 3). Either move results in all 10 black pieces situated on a single square – a perfect chance for Player 2 to win.

K**ey Move #333:** Player 2 has completed their turn; Player 2 wins

The only option for Player 1 is to move all 10 pieces from (3, 3) to (2, 3). So, to block this move, Player 2 moves the 1 black piece from (1, 3) to (2, 3). Note that this block could also have been accomplished by moving the 1 black piece from (1, 2) diagonally to the right into the square (2, 3), or by distributing the 2 black pieces at (2, 1) up. Either way, Player 1 is blocked and loses the game.

**Alpha Beta Pruning**

Refer to Figure 2.2 for the following section.

**Key Move #31:** Player 1 has completed their move

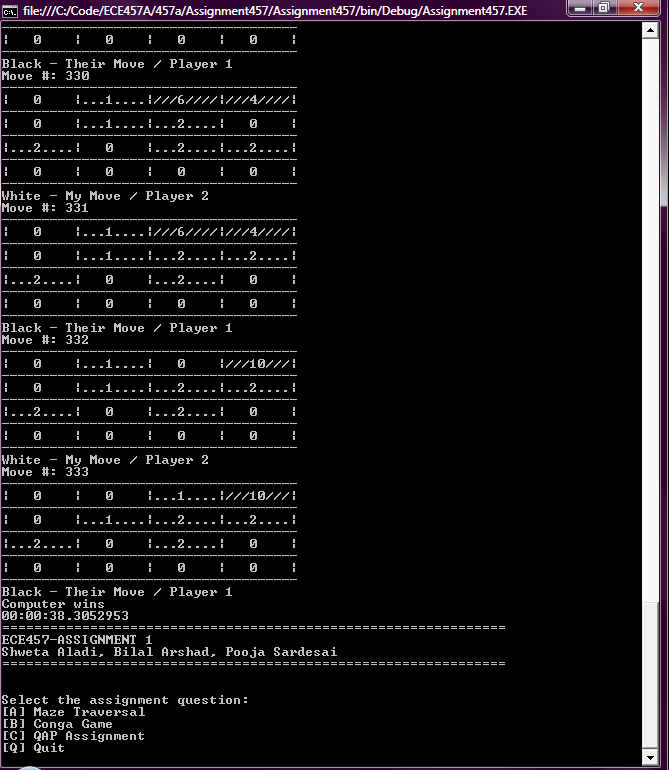
The 9 black pieces at (3, 3) are trapped by the surrounding white pieces. The only black piece left to trap is the 1 piece at (0, 1). Though moving the 2 white stones from (1, 1) to (0, 1) does not decrease the number of moves the white piece can make, the black pieces are 'herding' the white piece into the corner at (0, 3).

**Key Move #32:** Player 2 has completed their move

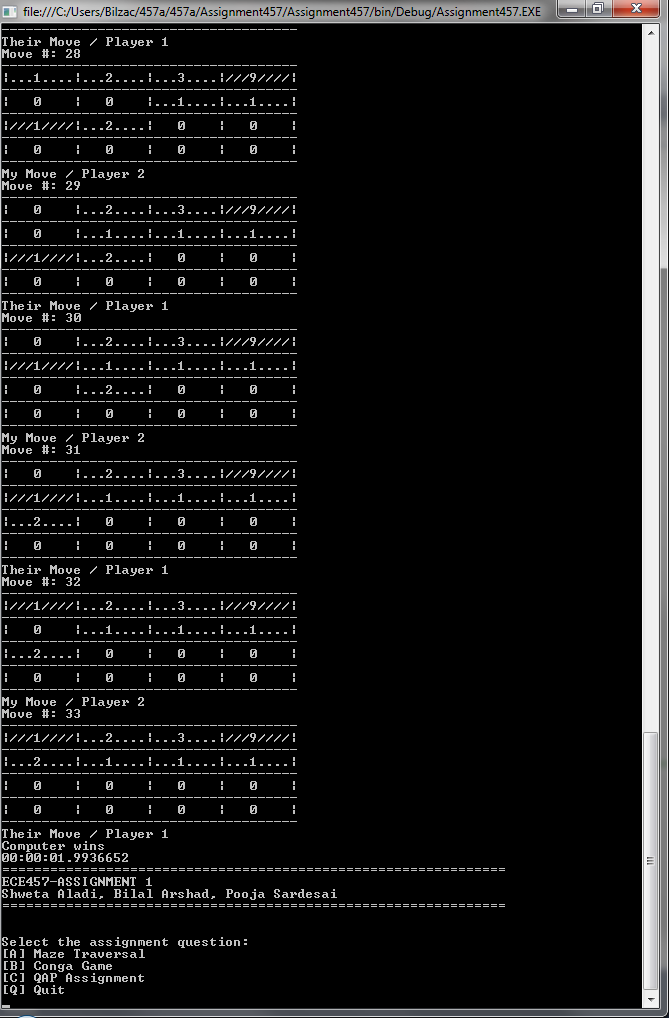
The random agent could have chosen to go diagonally from (0, 2) to (1, 1) (which would open up more moves) or go up to (0, 3). Luckily for Player 1, the white piece is moved up - it is trapped in the corner.

**Key Move #33:** Player 1 has completed their move; Player 2 wins

The obvious move is to block the white piece by moving the 2 black stones from (0, 1) to (0, 2). Player 2 wins the game.



**Figure 2.1:** Sample Output Against Random Agent of Best Evaluation Function using Minimax



**Figure 2.2:** Sample Output Against Random Agent of Best Evaluation Function using Alpha Beta Pruning

**Implementation Details**

**Minimax Search**

MINIMAXSEARCH  
Initialize the starting gameboard (parent)  
Starting game is Min - Player 1 (black, random agent)  
Set turn to false so that player 1 goes first  
Initialize counter (total\_moves) to track total number of moves  
Display the starting move called 'parent' on console  
  
WHILE(total\_moves is less than 5000)  
  IF (turn is false) then - player 1's turn, black  
        
      Get number of moves for parent node  
      IF( moves is 0) THEN  
       CalculateMoves of parent node  
      END  
  
      IF (moves is 0) Then  
       Break - player 2 wins  
      END  
  
      Create new object from random class  
      Calculate all possible moves for black to make from parent gameboard  
      Choose one random move from list of possible moves and set to next\_move  
      Set turn to true - player 2's turn  
  ELSE  
      Calculate best move for parent using CalculateMaxMove function and set to start\_move  
      Set next\_move = parent  
      Set counter = 0 (counts depth of tree of child moves being generated)  
      Set iterative counter internal\_counter to 1 - track iterative deepening search  
        
      WHILE(counter < internal\_counter)  
      Get number of moves in next\_move  
     
      IF( number of moves is greater than 0) Then  
          Set moves to number of children in next\_move  
      ELSE  
       Calculate number of moves for next\_move  
      END  
  
      IF(moves is 0) Then  
       break; - player 1 wins  
      END  
  
      IF(parent node is a max node) Then  
          IF(next\_move doesn't have any children) Then  
              break;  
          END  
          Set next\_move to the Max child  
      END  
      Increment counter  
      IF( counter is equal to internal\_counter and internal\_counter is less than 50) Then  
          increment internal counter  
      END  
     
      WHILE(next\_move's parent's parent is not the start move)  
       Set next\_move to its parent  
      END  
  
     Set turn to false  
         
  END  
  Display move and move #  
  Set parent to next\_move  
  Increment total\_moves  
  Remove next\_move's parent  
  Call Garbage collect  
END  
  
IF(next\_move has children) Then  
  It is a stalemate  
END  
  
IF(next\_move is of type Max)  
  Player 1/Random Agent wins  
ELSE  
  Computer/Player 2 wins  
END

CALCULATEMAXMOVE(parent)  
  
IF(parent has children) Then  
   Store the first child in variable best\_child  
   FOREACH(child of parent)  
   IF(child's move >= parent's alpha value) Then  
   Set child as new best\_child  
   Set parent's alpha value to number of best\_child's moves  
   END  
   END  
   Return best\_child  
END  
Return null  
  
CALCULATEMINMOVE(parent)  
  
IF(parent has children) Then  
   Store the first child in variable worst\_child  
   FOREACH(child of parent)  
   IF(child's move <= parent's beta value) Then  
   Set child as new worst\_child  
   Set parent's beta value to number of worst\_child's moves  
   END  
   END  
   Return worst\_child  
END  
Return null  
  
CALCULATEMOVES(board)  
  
Set moves to 0  
Set player colour to none  
Set player type to null  
  
IF(board is a Max node) Then  
   Set player colour to black  
   Set child types to Min  
ELSE  
   Set player colour to white  
   set child types to Max  
END  
  
FOR(x < 3)  
   FOR(y < 3)  
   IF(board has stones in position x,y of same as player colour,  
   or is not empty) Then  
   IF(There are 1 or 2 stones in position x,y) Then  
   Check 1 square North of current position x,y  
   Check 1 square South of current position x,y  
   Check 1 square East of current position x,y  
   Check 1 square West of current position x,y  
   Check 1 square NorthEast of current position x,y  
   Check 1 square SouthEast of current position x,y  
   Check 1 square NorthWest of current position x,y  
   Check 1 square SouthWest of current position x,y  
   ELSEIF(There are 3 stones in position x,y) Then  
   Check 2 squares North of current position x,y  
   Check 2 squares South of current position x,y  
   Check 2 squares East of current position x,y  
   Check 2 squares West of current position x,y  
   Check 2 squares NorthEast of current position x,y  
   Check 2 squares SouthEast of current position x,y  
   Check 2 squares NorthWest of current position x,y  
   Check 2 squares SouthWest of current position x,y  
   ELSE  
   Check 3 squares North of current position x,y  
   Check 3 squares South of current position x,y  
   Check 3 squares East of current position x,y  
   Check 3 squares West of current position x,y  
   Check 3 squares NorthEast of current position x,y  
   Check 3 squares SouthEast of current position x,y  
   Check 3 squares NorthWest of current position x,y  
   Check 3 squares SouthWest of current position x,y  
   END  
   END  
   END  
END  
  
Store number of moves in board using SetAlphaBeta value  
Return number of moves  
  
**AlphaBeta Search**

AlphaBetaSearch()

CREATE new GameBoard (parent)

SET Node Type of parent as Min

SET total\_moves to 0

CREATE flag/boolean (turn) and SET to False //False player’s turn, True computer turn

CREATE new GameBoard (next\_move) and SET to Null

WHILE parent has children AND total\_moves less than 5000

IF turn is False //Player, Black, Min turn

Generate random number between 0 and number of parent children

SET next\_move to random child of parent (based on random index generated)

SET turn to false

ELSE

set best beta (best) value to 10,000

CREATE Collection (potentialMoves)

FOREACH( child of parent )

CalculateAlphaBeta(child, depth, alpha, best, Min Node);

//Default Values of depth = 5, alpha = -10000 & beta = 100000

IF ( Child Beta value is LESS THAN to best )

SET best to child beta value

CLEAR potentialMoves

ADD child to potentialMoves

ELSE IF ( Child Beta value is EQUAL to best)

ADD child to potentialMoves

END

END

Generate random number between 0 and number of potentialMoves size

SET next\_move to random element of potentialMoves (based on random index  generated)

set parent beta value to best

set turn to true

END

SET parent to next\_move

increment total\_moves

END

IF next\_move is node type Max

END GAME, PLAYER 1 wins

ELSE

END GAME, COMPUTER wins

END

CalculateAlphaBeta(GameBoard parent, int depth, int alpha, int beta, NodeType player)

SET moves to Number of Children of parent

if moves IS 0 OR depth IS 0

return 10000

END

IF player type MAX

FOREACH ( child in parent )

SET alpha to Maximum of alpha and CalculateAlphaBeta(child, depth-1, alpha,  beta, child NodeType);

SET Alpha value of child node to alpha

IF beta is LESS THAN OR EQUAL to alpha

break

ENDIF

ENDFOREACH

return alpha

ELSE //MIN NODE

FOREACH ( child in parent )

SET beta to Minimum of beta and CalculateAlphaBeta(child, depth-1, alpha,  beta, child NodeType);

SET Beta value of child node to beta

IF beta is LESS THAN OR EQUAL to alpha

break

ENDIF

ENDFOREACH

return beta

END

RETURN 0

**Complexity**

|  |  |  |
| --- | --- | --- |
|  | **Time Complexity** | **Space Complexity** |
| **Min Max Search** | O(b^d) where b is the breadth and d is the depth passed in as a parameter | O(b^d) where b is the breadth and d is the depth passed in as a parameter. |
| **Alpha Beta Pruning** | O(b^3d/4) where b is the breadth and d is the depth passed in as a parameter. | O(b+d) where b is the breadth and d is the depth passed in as a parameter. |

**Question 3 - QAP Assignment**

**Program Details**

The general idea behind this tabu search is:

1) Start with an initial solution

2) Obtain all solutions in the current neighbourhood by applying a series of valid moves, such as moves that have not been applied for more than 100 times, and moves that are not tabu moves

3) The best solution from the neighbourhood is selected and it becomes the new best current move

4) Repeat steps 2-3 for many iterations until a stopping condition is met

**Simple Tabu Search**

The initial starting solution used for this search is:  
1   2   3   4  
5   6   7   8  
9   10  11  12  
13  14  15  16s  
17  18  19  20  
The results obtained below were based on this starting solution.

Because the distance and flow matrix is a 20x20 matrix, the initial tabu list size was set to 201/2, which is approximately 5. This is an ideal starting tabu list size, as presented in the Week 2 slides.

The aspiration criteria used for this search are:

1) Always choosing the solution that is the best in the neighbourhood, and

2) If a tabu move provides a solution that is better than the best solution, the move is allowed.

The stopping condition for the search is:

1) To stop the search at 1000 iterations, even if the optimal cost is not found, or

2) To stop the search when the optimal cost is found.

Once it stops, the cost and solution is displayed on the console.

**Changing the initial starting point**

If the initial solution is changed, the optimal solution occurs at different iterations. Because of this, the solution hits

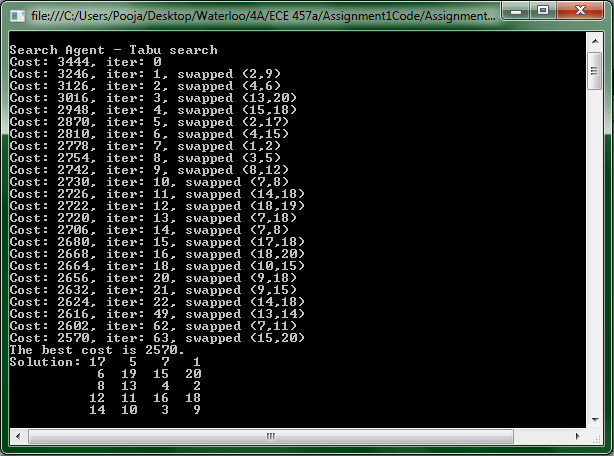
**Initial/Increase/Decrease tabu list**

The initial change to the simple tabu search was setting a fixed tabu list size, at different values. The data obtained can be seen in Table 3.1.

**Table 3.1:** Cost obtained at different tabu list size after 1000 iterations. The optimal cost occurs at different iterations of the search.

|  |  |  |
| --- | --- | --- |
| **Tabu List Size** | **Iterations** | **Cost** |
| 2 | 29 | 2620 |
| 5 | 53 | 2620 |
| 9 | 505 | 2570 |
| 12 | 85 | 2570 |
| 14 | 267 | 2570 |
| 19 | 125 | 2570 |
| 20 | 63 | 2570 |
| 24 | 557 | 2570 |
| 40 | 22 | 2624 |
| 60 | 855 | 2614 |
| 80 | 22 | 2624 |
| 100 | 22 | 2624 |

As seen above, a tabu list of length 5 does not yield the optimal cost. It yields a maximum cost in iteration 53 of 2620.  As you decrease the list size, the cost does not improve. If the list size is increased, the solution gets better, first obtained at high iterations and then finally obtaining the best cost with a tabu list size of 20. Note that if the list size is too big, it diminishes the search results since it forces the search to explore unexplored solutions in the search space, hence penalizing solutions that are close to the current best. Similarly, a smaller list size explores a small portion of the search space and remains local, hence penalizing solutions that are far away from the current solution. Figure 3.1 shows a sample console output.



**Figure 3.1**: Sample console output of the search with tabu list size 20

**Dynamic tabu list size**

The dynamic tabu list size range is between 1 and 10. When a dynamic tabu list size is applied to the search, the optimal cost is achieved approximately 50% of the time, some occurring quicker than the rest. Table 3.2 shows the results for the first 10 searches.

**Table 3.2:** Dynamic tabu list size results for the first 10 searches

|  |  |
| --- | --- |
| **Cost** | **Iterations** |
| 2570 | 929 |
| 2570 | 491 |
| 2570 | 107 |
| 2588 | 411 |
| 2574 | 236 |
| 2574 | 324 |
| 2574 | 306 |
| 2570 | 187 |
| 2570 | 219 |
| 2574 | 694 |

**Use less than the whole neighbourhood to select the next solution**

For this search, a fixed tabu size was used. When the search only explores some candidates in the neighbourhood, the search finishes quicker than the other changes. However, the cost lies within 2600 and 2700, and the optimal cost is almost never achieved. The results are the same as the fixed tabu list size is varied or if a dynamic tabu list size is used. Table 3.3 shows the results obtained when a fixed tabu list size of 5 is used.

**Table 3.3:** Results when some candidates are selected from the neighbourhood

|  |  |
| --- | --- |
| **Cost** | **Iterations** |
| 2598 | 920 |
| 2704 | 414 |
| 2640 | 213 |
| 2640 | 411 |
| 2656 | 830 |
| 2640 | 397 |
| 2618 | 443 |
| 2668 | 999 |
| 2634 | 344 |
| 2620 | 987 |

**Add frequency based tabu list**

Adding a frequency based tabu list helps improve the search. The optimal cost can be found in fewer iterations, based on the tabu list size.

**Implementation Details**

**Method:** doTabuSearch

Create initial solution  
Set initial solution as the best solution  
Create tabu list  
Create a list to store a frequency of moves  
  
Create variable to keep track of the number of iterations  
WHILE (the number of iterations exceed a certain value)  
 Increase iteration count  
 Reduce tenure value in tabu list  
 Get valid candidates from the neighbourhood  
 IF (there are valid candidates)  
   Get the first element since it has the best solution in the neighbourhood  
   Print solution if it is the current best  
   Swap the values that makes the solution better  
   Add move to Tabu list  
   Increase the frequency of move  
 END  
 Print solution  
END  
  
**Method:** getSomeCandidatesFromNeighbourhood

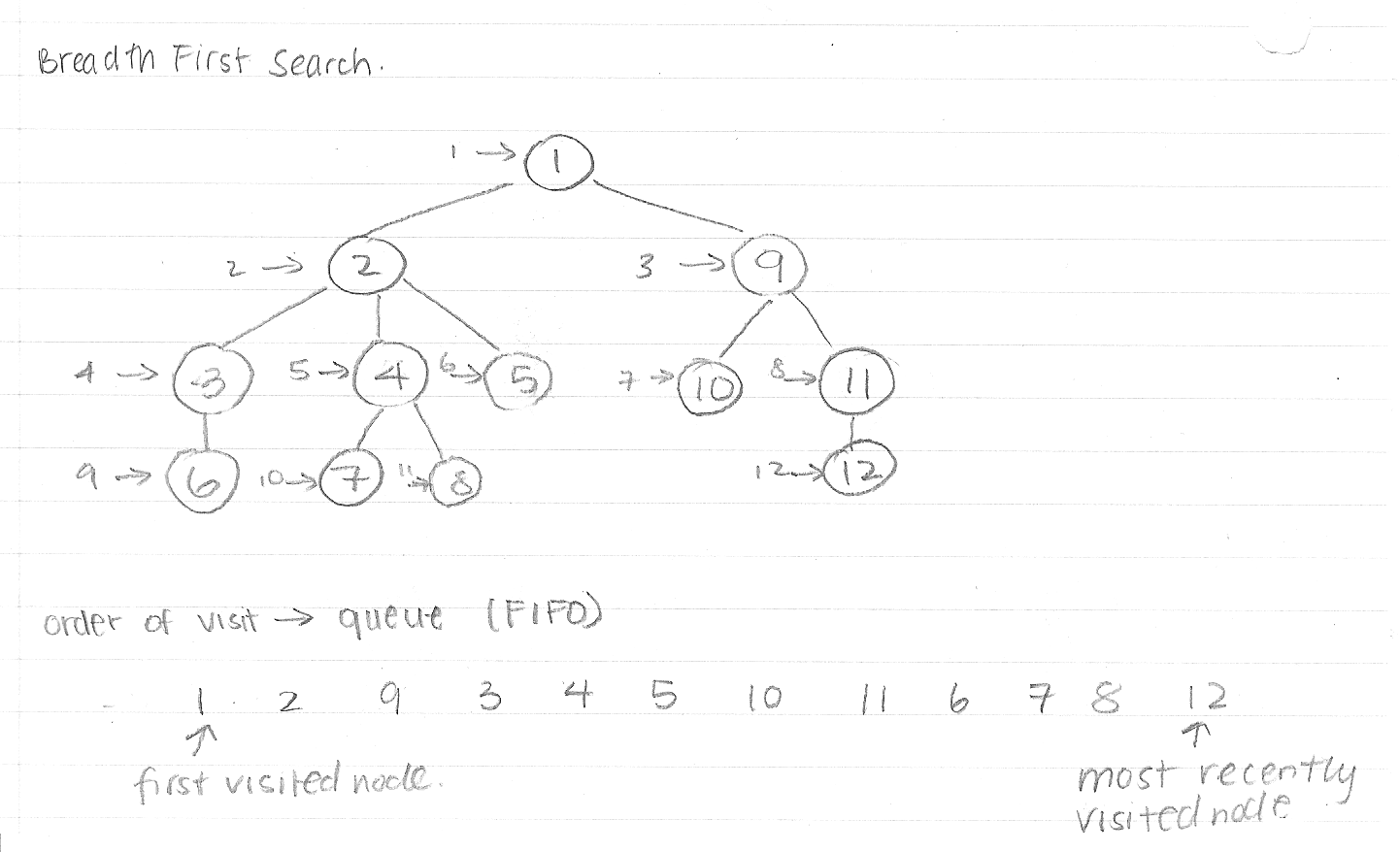
CREATE random variable  
GENERATE random position values  
CHECK if position values are valid  
  
CREATE a list to store valid candidate pairs  
FOR (each row)  
 FOR (each column)  
   IF (move has not been used many times)  
     Temporarily swap values and get the new arrangements  
     Get new solution  
     IF (the swap is in the tabu list)  
       IF (solution is better than the best solution)  
         ADD solution to candidate solutions  
       END  
     ELSE  
       ADD solution to candidate solutions  
     END   
   END  
 END  
END  
  
SORT list in order from lowest to highest solution value

**METHOD:** getAllCandidatesFromNeighbourhood  
  
CREATE a list to store valid candidate pairs  
FOR (each row)  
 FOR (each column)  
   IF (move has not been used many times)  
     Temporarily swap values and get the new arrangements  
     Get new solution  
     IF (the swap is in the tabu list)  
       IF (solution is better than the best solution)  
         ADD solution to candidate solutions  
       END  
     ELSE  
       ADD solution to candidate solutions  
     END   
   END  
 END  
END  
  
SORT list in order from lowest to highest solution value  
  
  
**Complexity**

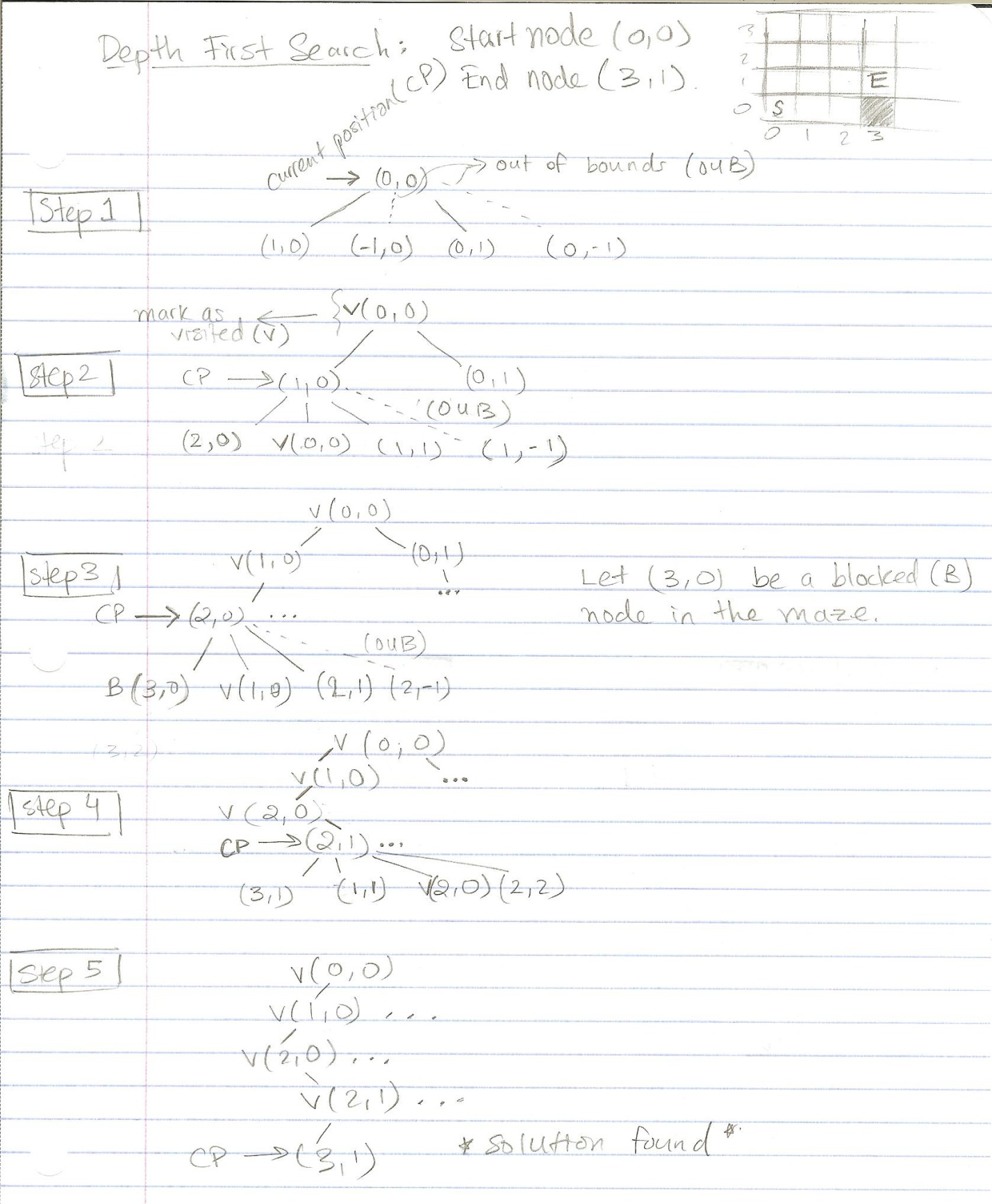
|  |  |  |
| --- | --- | --- |
|  | **Time Complexity** | **Space Complexity** |
| **Tabu Search** | O(n^2) | O(n^2) |

**Hand Worked Samples**

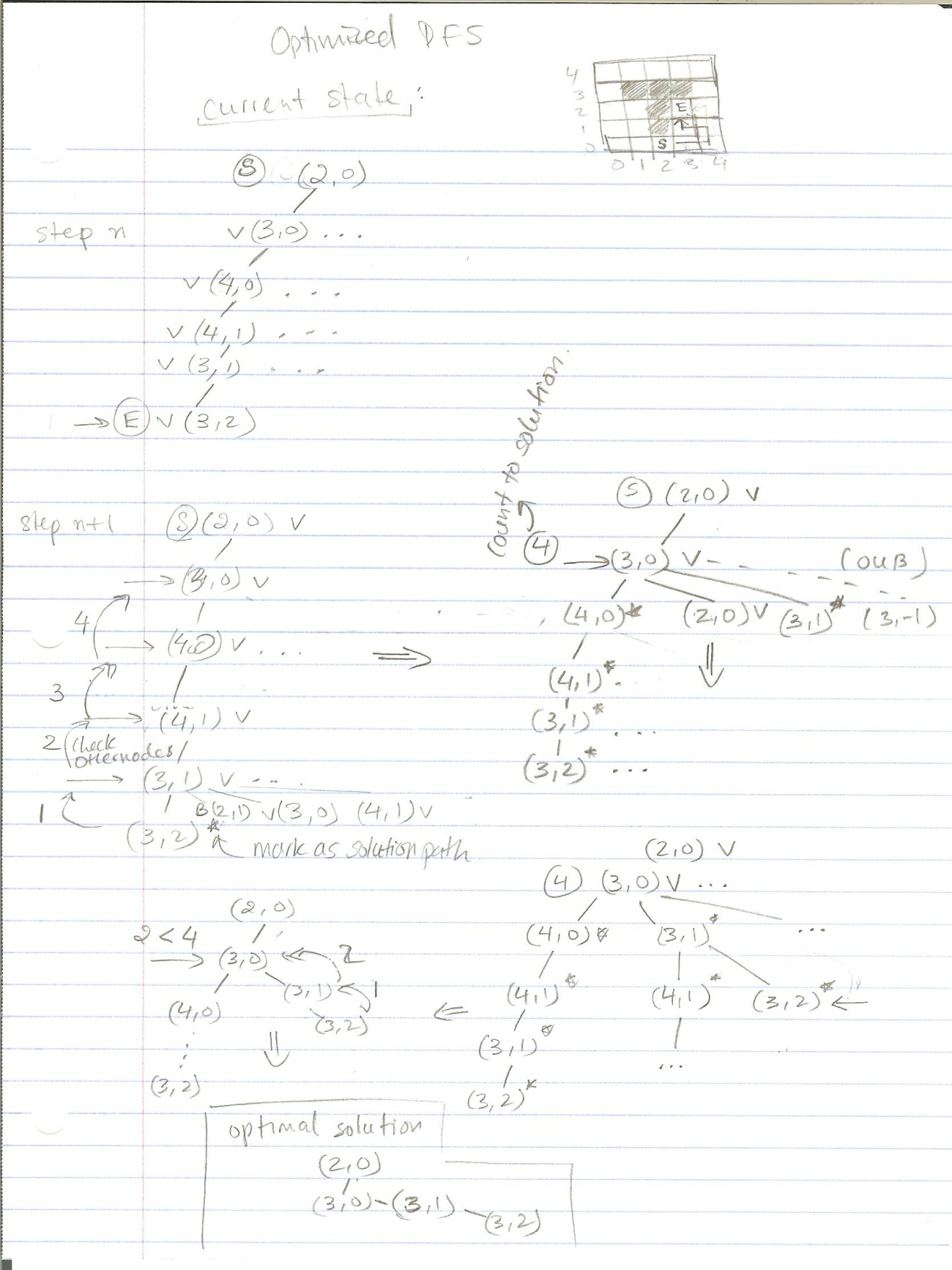
1. Breadth First Search



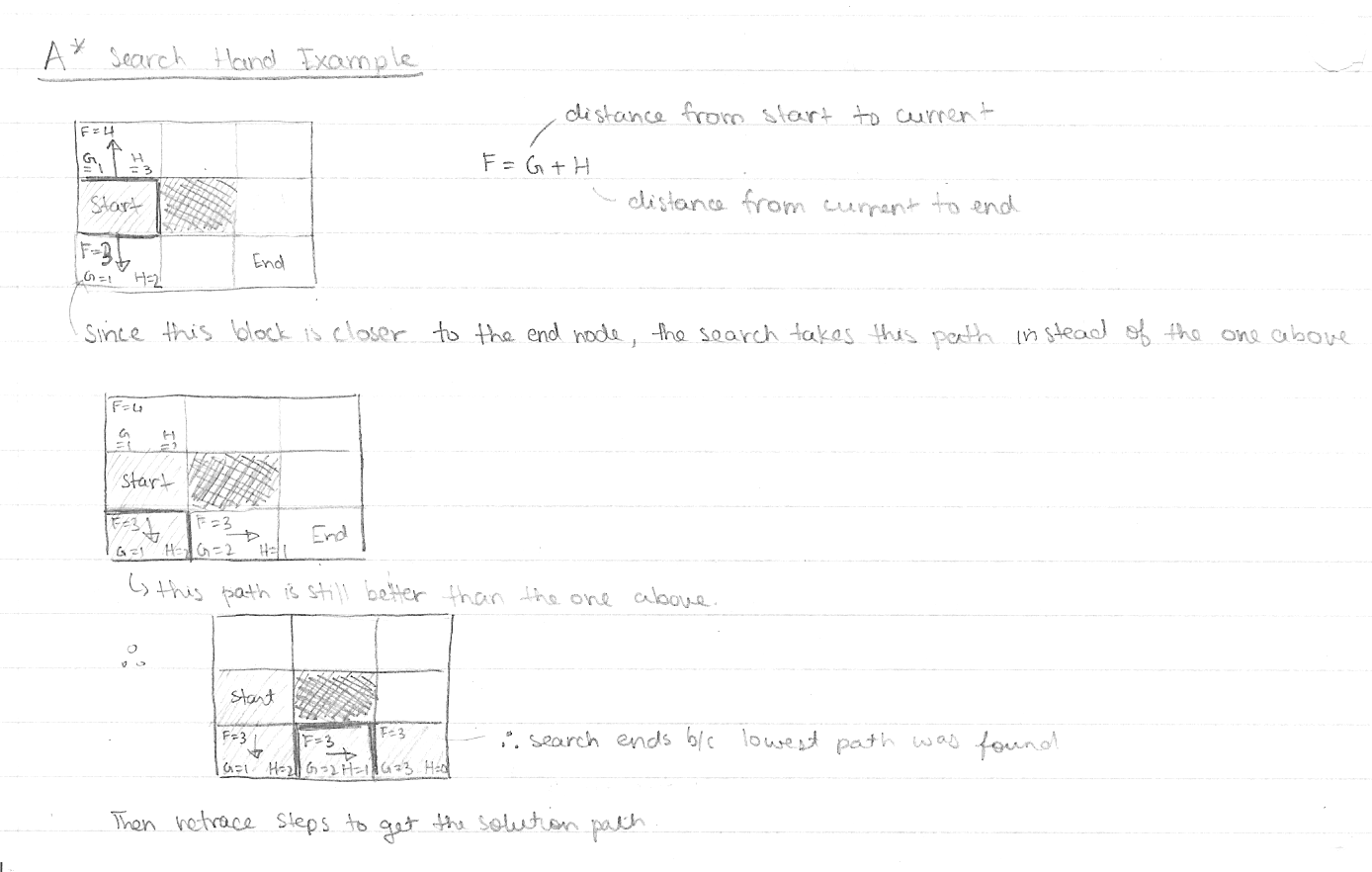
1. Depth First Search



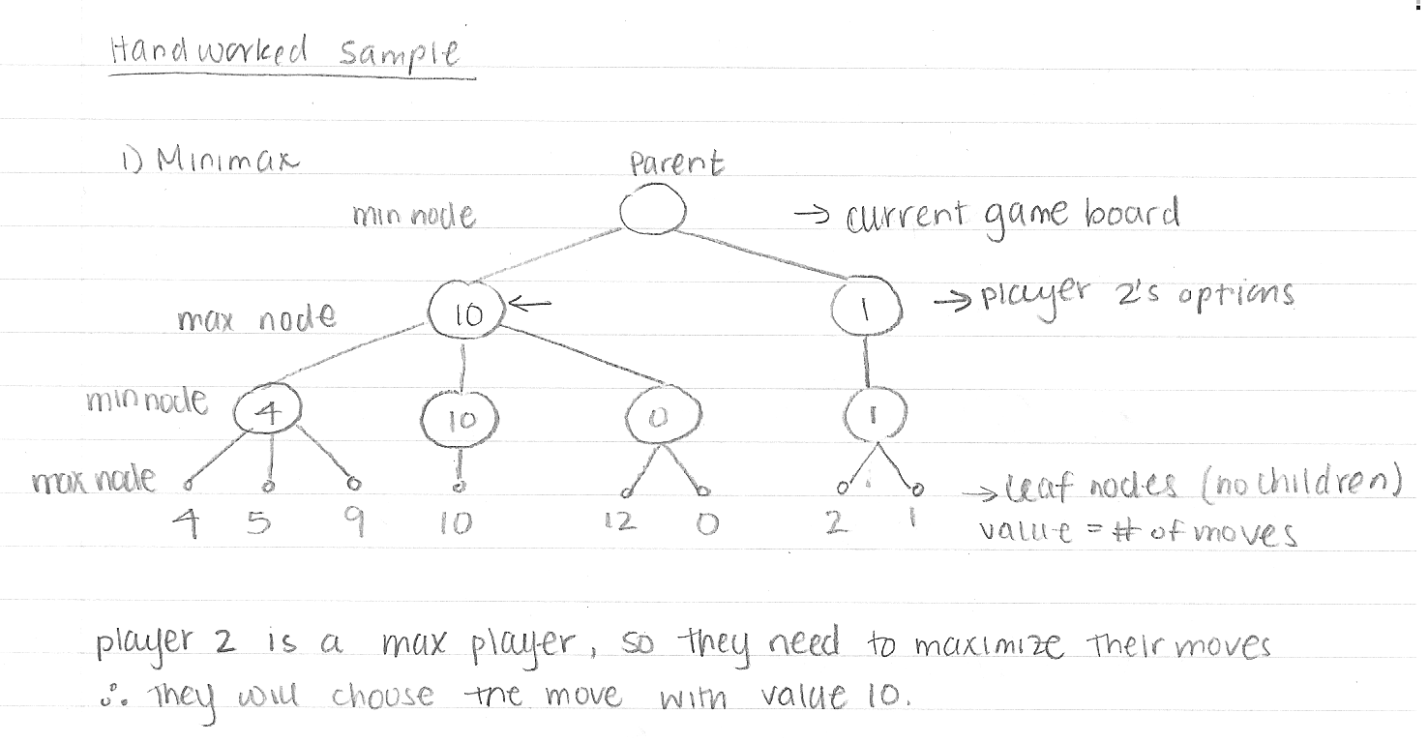
3) Optimized Depth First Search



4) A\* Search



5) Minimax



6) Alpha-Beta Pruning  
  
  
  
7) Tabu Search

