Assignment #1
Artificial Intelligence - CSCE 523
Due: 8:00 AM, Wednesday January 16, 2019
Agents and Uninformed Search

Turnin: E-mail me a zip file containing your typed solution to questions 1 through 5, and your program and report for question 6.

1. (10 points) What is intelligence? There are several research views that can be taken to understand intelligence, for the purpose of this discussion, let us label them philosophical, physicalism, and psychological. Almost every AI algorithm we will discuss is related to one or more of these views. Read the Stanford Encyclopedia of Philosophy articles on the Theory of Mind topics of Qualia and Physicalism (http://plato.stanford.edu/entries/qualia/ http://plato.stanford.edu/entries/physicalism/), and the Sweller paper on Human Cognitive Architecture (http://www.csuchico.edu/~nschwartz/Sweller_2008.pdf) {note: this is the closest I have found to a good short discussion on cognitive psychology and the modeling of it}
    a. Which view do you feel is more correct? And why?
    b. Given your choice, is an artificially generated intelligence possible? Why or why not?

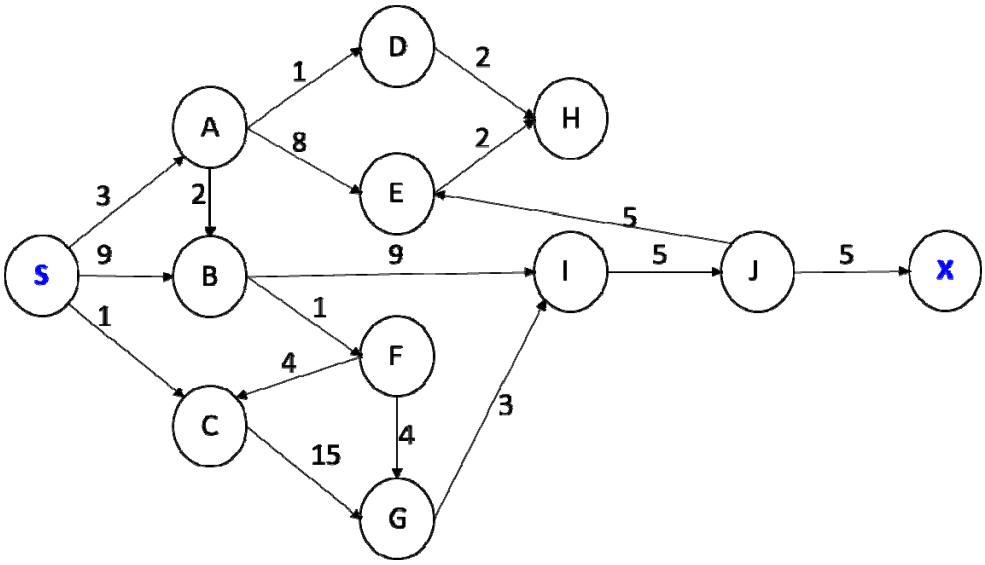
2. (15 points) Testing for intelligence: Read Turing's original paper (available on-line at: http://www.abelard.org/turpap/turpap.htm). In the paper, he discusses several potential objections to his proposed enterprise and his test for intelligence. Also, refer to the discussion on the Chinese Room Argument found in the text, slides and at: http://plato.stanford.edu/entries/chinese-room/ or video: http://www.open.edu/openlearn/history-the-arts/culture/philosophy/60-second-adventures-thought?track=04fb8b569.
    a. Which objections still carry some weight? Are his refutations valid?
    b. Can you think of new objections arising from developments since he wrote the paper?
    c. Do you think that there is a better test that could be proposed?

3. (5 points) Intelligence and computational limits: There are well-known classes of problems that are intractably difficult for computers and other classes that are provably undecideable by any computer. Does this mean that strong (human level) AI is impossible? Why or why not?
   This goes to the question of Turing computability if we must reduce intelligence to something Turing computable, then strong AI may not be possible, of course if we could determine at a fine enough level what the

brain actually does do, then we could potentially duplicate it as a Turing machine

4. (10 points) Consider a modified version of the vacuum-cleaner world (depicted in Figures 2.2 and 2.3 and specified on pages 35-36), in which the geography of the environment – its extent, boundaries, dirt locations, and obstacles is unknown; the agent can go Up and Down as well as Left and Right.
   If square dirty suck, if not randomly move (N, S, E, or W)
   a. Design a simple reflex agent for this domain (create some rules). Is this reflex agent perfectly rational for this environment? Explain.
      Given that perfectly rational agent would complete the task with some performance measure. In this case, the shortest number of moves -> No, without state, the reflex agent cannot identify rooms that it has or hasn't cleaned before and will therefore irrationally duplicate work.
   b. Can you design an environment in which your reflex agent will perform very poorly? If not explain, if yes provide an example.
      Yes, any maze will do.
   c. Can a reflex agent with state outperform a simple reflex agent? Why?
      Yes, see a.

5. (20 points) For the following tree, show the lists of open and visited nodes for each cycle of the listed search algorithms. Expand the nodes in a *top to bottom* ordering. The start node is **S** and the goal node is **X**. The numbers next to the edges indicate the associated cost. Note: follow the format from class.
   a. Breadth-first search
   b. Depth-first search
   c. Uniform cost search



a. BFS (queue)

Open	Visited
S	
ABC	S
BCDE	SA
CDEFI	SAB
DEFIG	SABC
EFIGH	SABCD
FIGH	SABCDE
IGH	SABCDEF
GHJ	SABCDEFI
HJ	SABCDEFIG
J	SABCDEFIGH
X	SABCDEFIGHJ

GOAL FOUND!

b. DFS (stack)

Open	Visited
S	
CBA	S
GBA	SC
IBA	SCG
JBA	SCGI
XBA	SCGIJ
BA	SCGIJ**X** <= GOAL FOUND!

c. UCS (priority-queue g(n))

Open	Visited
S_0	

$C_1A_3B_9$
$A_3B_9G_{16}$
$D_4B_5E_{11}G_{16}$
$B_5H_6E_{11}G_{16}$
$F_6H_6E_{11}I_{14}G_{16}$
$H_6G_{10}E_{11}I_{14}$
$G_{10}E_{11}I_{14}$
$E_{11}I_{14}$
I_{13}
J_{18}
X_{23}

FOUND!

S_0
S_0C_1
$S_0C_1A_3$
$S_0C_1A_3D_4$
$S_0C_1A_3D_4B_5$
$S_0C_1A_3D_4B_5F_6$
$S_0C_1A_3D_4B_5F_6H_6$
$S_0C_1A_3D_4B_5F_6H_6G_{10}$
$S_0C_1A_3D_4B_5F_6H_6G_{10}E_{11}$
$S_0C_1A_3D_4B_5F_6H_6G_{10}E_{11}I_{13}$
$S_0C_1A_3D_4B_5F_6H_6G_{10}E_{11}I_{13}J_{18}$
$S_0C_1A_3D_4B_5F_6H_6G_{10}E_{11}I_{13}J_{18}$ $\mathbf{X_{23}}$ <=GOAL

6. (40 points) Ah, Christmas is over. Now, that horrible mall traffic should let up… what is this, a traffic jam?

   Seems that everyone is stuck and you can't get through. You must direct the trapped shoppers out of your way and maneuver your trusty vehicle through the maze to get back home in one piece.

   For this problem, implement a solver for the Rush Hour puzzle. Rush Hour is a sliding block puzzle containing 4 trucks, 11 cars, and your vehicle placed on a 6x6 grid, with impenetrable walls surrounding its perimeter except for a one cell exit edge. The trucks occupy an area of one by three cells, and the cars occupy an area of or one by two adjacent grid cells. All vehicles can only move forwards or backwards, never overlapping. The goal is to move the vehicles one at a time so that your car may exit.

   The implementation should read a set of puzzles from a text file (format shown in the example below). Where A1 to K1 are cars, O1 to R1 are trucks, and X0 is your vehicle. The exit is always at the third position down on the right, and each empty grid location is (..). The file begins with the number of puzzles found in the file.

   Expected turn-in is the 1) source code, 2) compilation and execution instructions, and 3) a short paper describing your solution, results and any search enhancements. A set of 5 problems are provided for testing your search algorithm during development (simple.txt), and 5 that must be tested against in the report (hard.txt), be sure to include your timing and best path results for each of the problems your search can solve. Your implementation must be your work only. As for what enhancements to try – take a look at the publications of Andreas Junghanns on Sokoban (http://www.cs.ualberta.ca/~games/Sokoban/papers.html).

To get you started, I have provided Java and MATLAB code that will read the files and handle the state updating (both are in the class directory). For the Java, just implement the search interface in each of your own search(es). For MATLAB, fill in the search function. Note on implementation language – you may find that MATLAB is quicker to write than Java, however you will take a significant performance hit. Using a BFS, the runtime comparison between the implementations on the simple.txt problems are:

Problem	Java	Matlab
1	0.003s	0.03s
2	0.08s	0.13s
3	0.50s	4.30s
4	0.44s	755.89s (13 min)
5	0.92s	(7hr 36min)

```
1
O1..Q1Q1Q1B1
O1....C1..B1
O1X0X0C1.... <= Exit is here, this comment and the arrow
......P1....              does not appear in the file.
......P1....
......P1....
```

Breadth First Search:

```
public class BFSwVisitedSearch implements Search {
    public long node_count;
    private Board board;
    private boolean done;    // Have we found the goal?
    private Queue<Board> Q = new LinkedList<Board>();
    private Hashtable<String, Integer> visited = new
Hashtable<String, Integer>();

 public BFSwVisitedSearch( Board b ){
  board = b;
  Q.offer(b);
 }

 public Move findMoves(){
   // while not done
   while (!done){
    node_count++;
```

```
      // Pull node off of queue
     board = Q.poll();

        // determine if goal found
     if ( board.isGoal() ) {
       done = true;
       return board.move_list; //found goal return the
move_list that got us here
     }

     //get moves for this node
     Move possible_moves = board.genMoves();

     //expand moves from this node
     for (Move another_move = possible_moves; another_move
!= null; another_move = another_move.next ) {
       // Test for a duplicate board
       board.makeMove(another_move);
       if (!visited.containsKey(board.hashKey())) {
         // We haven't seen this move before so add it to
the queue
         Board new_board = new Board(board);
      visited.put(new_board.hashKey(), new Integer(1));
       // insert new board node onto queue (BFS)
   Q.offer(new_board);
       }
       board.reverseMove(another_move);
     }
   }

   return null;  //return null (should never hit this)
 }

 public long nodeCount(){
  return node_count;
 }
}
```

Iterative Deepening Search:

```
public class IDSwVisitedSearch implements Search {
 public long node_count;
 private Board board;
 HashMap<String, Integer> visitedBoards;

 /*
  * IDSSearch constructor
  *
```

```
  * @param b Board
  */
 public IDSwVisitedSearch( Board b ){
  board = b;
 }

 /*
  * @see Search#findMoves()
  *
  * For IDS, the search iteratively increases the search
depth and performs a
  * depth first search
  *
  * @return Move if no goal found null, else the path to
the goal node
  */
    public Move findMoves() {
      Move move_list = null;
      int depth = 1;

      // search to the give depth, if we find a goal then
stop, else increase depth and
      // search again.
      while (move_list == null ) {
     System.out.println("Depth: " + depth);
     // we have to reinitialize the visited list each
iteration
     visitedBoards = new HashMap<String, Integer>(100000);
     move_list = IDS(depth);
     depth++;
      }

   return move_list;
    }

    /*
     * Recursive implementation of IDS
     * Note that the depth value decreases to the leaf
rather than increase
     *
     * @param depth int the current search depth
     * @return Move if no goal found null, else the path to
the goal node
     */
    private Move IDS( int depth ) {
      Move move_list = null;
```

```
   // if the current depth exceeded, then return no goal
found
   if ( depth < 0)
  return null;

   // visiting this node, count it
   node_count++;

   //determine if goal found
   if ( board.isGoal() ) {
     return board.move_list; //found goal return the
move_list that got us here
   }

   //get moves for this node
   Move possible_moves = board.genMoves();

   //expand moves from this node
   for (Move another_move = possible_moves; another_move !=
null; another_move = another_move.next ) {
  board.makeMove(another_move); //try_move

  String boardString = board.hashKey();       //string
representation of board after move

  if (visitedBoards.containsKey(boardString)) {    //if
already visited this space
    if (visitedBoards.get(boardString) >= depth) { // and
the previous is closer to the start
      board.reverseMove(another_move);            // undo
the last move
      continue;                                   //
continue iterating
    }
    else {                                        // this
node is closer, remove the node, the new
   visitedBoards.remove(boardString);           // depth
will be added as part of the regular process.
    }
  }
  visitedBoards.put(boardString, new Integer(depth));

  // search this node
  move_list = IDS(depth-1);
  if ( move_list != null) // if we found a goal, return the
move_list
    return move_list;
```

```
  board.reverseMove(another_move); // undo the move
   }

   return null;
    }

    /*
     * (non-Javadoc)
     * @see Search#nodeCount()
     */
    public long nodeCount() {
   return node_count;
    }
}
```