**CIS 3700 (Winter 2020): Introduction to Intelligent Systems**

# ASSIGNMENT 1

Instructor: Y. Xiang
Assigned date: Thur., Jan. 16, 2020. Due Date: Mon., Feb. 3, 2020.

## Problems (Total marks: 40)

1. (2 marks) [Agent environment] AI researchers are working towards robot soccer teams that can compete with top human teams. Categorize the environment of robot soccer agents in terms of seven properties discussed in lectures. Justify each choice clearly with the reason.

2. (3 marks) [Environment representation] A medical diagnostic agent needs to determine whether a patient suffers from Tuberculosis. Develop a state based environment description.

   (a) Specify a set $V$ of variables (no more than 10 and no less than 5).

   (b) For each variable, describe what it represents briefly and specify its range space.

   (c) Specify a complete environment state.

   Specify your answer with set notation as shown in lectures. Write the set of variables as $V = \{var_1, var_2, ...\}$, where $var_i$ is a variable name. Write the space of variable $var$ as $S_{var} = \{value_1, value_2, ...\}$, where $value_j$ is a possible value. If the space is a real interval, write $S_{var} = [L, H]$, where $L$ and $H$ are bounds of the interval. Write environment state as a tuple $(var_1 = v_1, var_2 = v_2, ...)$.

   Hint: Google the disease and its diagnosis. This problem practices how to encode environment of an agent, not how to become a medical specialist.

3. (3 marks) [State space and search tree] An environment state space can be represented as a directed graph. Each node represents a distinct state. Each directed arc $s \rightarrow s'$ represents a legal action that, when performed at state $s$, results in state $s'$. A double arrow $s \leftrightarrow s'$ is a shortcut for $s \rightarrow s'$ and $s \leftarrow s'$. Answer the following where the search tree refers to a fully expanded search tree with an arbitrary initial state.

   (a) Does a finite state space (of a finite number of nodes) always lead to a finite search tree, and why?

   (b) Does a finite state space that is a tree without double arrows always lead to a finite search tree, and why?

   (c) What is the most general type of state spaces (characterized by topology of the directed graph) that always leads to finite search trees?

4. (4 marks) [Tree search] Consider tree search to solve this Sudoku puzzle. The initial state is the given configuration. For any state, to determine its successors, select an empty cell. The number of successors is the number of alternative digits to fill in the cell, while respecting Sudoku rules. Estimate the branching factor $b$ of the search tree, the depth $d$ of the shallowest goal node, and the max path length $m$.

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

(a) (2 marks) To estimate $b$, initialize a modified puzzle $M$ by copying the given puzzle $P$. For each empty cell in $M$, do the following independently according to $P$: If it can be filled with only 1 digit (from 1 to 9), then fill it with 'A' (to differ from 1). If it can be filled with only 2 digits, then fill it with 'B', and so on. Show the resultant puzzle $M$. What is the value $b$ estimated using $M$, and why?

(b) (2 marks) What are the values of $d$ and $m$ of the search tree, and why?

5. (6 marks) [BFS, DLS, and IDS] Consider state space where each state is labeled by an integer. From any state $n$, states reachable by one action are specified by function $Successor(n) = \{2n, 2n+1\}$. The goal state is 11.

(a) (1 mark) Draw the portion of the state space that contains only states labeled from 1 to 15. For each state $n$, draw successor states below $n$, and connect $n$ to each successor by a directed link. Draw state $2n$ to the left, and $2n + 1$ to the right.

(b) (1 marks) When a node is *visited*, it is first tested for goal. If the test fails, it is *expanded* so that each successor is *generated*. The next node to visit is selected by the search strategy. If multiple leaves tie by the strategy, break the tie in the order from left to right. List the order in which nodes are visited by BFS.

(c) (2 marks) List the order in which nodes are visited by DLS with limit 3.

(d) (2 marks) List the order in which nodes are visited by IDS. If a node is visited with limit $i$, and visited again with limit $i + 1$, then it appears in the order each time.

6. (4 marks) [BFS and A*] Solve the map routing problem discussed in lectures using both Breadth-First Search (BFS) and A* with straight-line distance heuristics.

(a) (2 marks) Show search tree produced by BFS. When expanding a node, generate child nodes in alphabetical order, and show them from left to right. How many nodes are generated by BFS? Is the solution optimal, and why?

(b) (2 marks) Show the search tree produced by A* with the same ordering above. For each node $n$, show the evaluation function value in the form $g(n) + h(n) = f(n)$, e.g., 118 + 329 = 447. How many nodes are generated by A*? Is the solution optimal? If so, justify **directly** (not by admissibility of $h(n)$).

7. (3 marks) [8-puzzle and A* with misplaced-tiles heuristics] Solve 8-puzzle by $A^*$ tree search for initial state $r$ (left) and goal state $t$ (right).

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

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

Use misplaced-tiles heuristics defined by $h(n)$ = number of misplaced tiles at state $n$. Do not count the misplaced hole. For instance, $h(r) = 5$ (not 6) for state $r$ above.

To expand a node, generate successors in the order of hole movement (Left, Right, Up, Down). If two nodes in fringe tie with evaluation function values ($f(n)$), they are visited in a first-in-first-out order. (Hint: This determines how nodes are inserted into fringe.)

(a) Show the search tree with successors of each node drawn from left to right in order of generation. For each node, show its state in the same format as $r$ and $t$ above.

(b) Show explicitly the solution obtained, as well as the corresponding sequence of actions.

8. (2 marks) [Monotonicity and admissibility] Answer the following regarding the above misplaced-tiles heuristics.

(a) Is $h(n)$ monotonic, and why?

(b) Is $h(n)$ admissible, and why?

9. (13 marks) [A* with Manhattan heuristics] Implement A* search in Java that solves 8-puzzle problem using treeSearch algorithm, A* strategy, and Manhattan heuristics. Manhattan heuristic function $h(n)$ is sum of distances of tiles (not hole) at state $n$ from their goal positions. In Problem 7, tile 8 in state $r$ contributes 2 to the sum.

   Organize Java classes in two sets: The 1st set is general to problem solving by treeSearch with A*, and is independent of 8-puzzle task. The set is given at course website as search.jar with source code in Appendix.

   (a) ObjectPlus: This abstract class enhances Java Object class with the ability to show itself in a console.

   (b) Problem: This abstract class specifies components in problem definition. It has an abstract method getSuccessor() that must be implemented by subclasses.

   (c) Node: This class implements a generic node in search tree. It has an instance variable *state* that is an object of class ObjectPlus.

   (d) SearchAgent: This class implements a generic agent that performs treeSearch.

   The 2nd set contains problem definition for 8-puzzle. You should implement this set of classes as described below, and submit the Java source code.

   (a) Board.java: It describes a state of 8-puzzle. It maintains the 8-puzzle configuration, the move that leads to this state, and heuristic function value of the state. As Board is an object whose configuration needs to be shown, it should be implemented as a subclass of ObjectPlus.

   Board should include methods that test whether the state is the goal, test whether a move is legal, determine the new state from a legal move, and determine the heuristic function value of a state.

   (b) PuzzleProblem.java: This class contains problem definition for 8-puzzle. It should be a subclass of Problem class above. Key methods include goal test and successor generation.

   Generate successors in the order of hole movement (Left, Right, Up, Down). If two nodes in fringe tie with evaluation function values ($f(n)$), they are visited in a first-in-first-out order. This determines how nodes are inserted into fringe.

   (c) PuzzleAgent.java: This class launches agent application, and should be a subclass of SearchAgent class above. It loads initial state and goal state from a file. It must implement showSolution(), showTree(), and insertFringe() as required by SearchAgent.

   Echo the search tree with successors of each node drawn from left to right in order of generation (see example below).

   No other Java classes should be submitted. All Java source codes should be placed in the same directory such as `\A1`, and should not use "package" statement. Your implementation must be compilable and executable as

```
javac -cp .;search.jar *.java
java -cp .;search.jar PuzzleAgent InitGoalFile
```

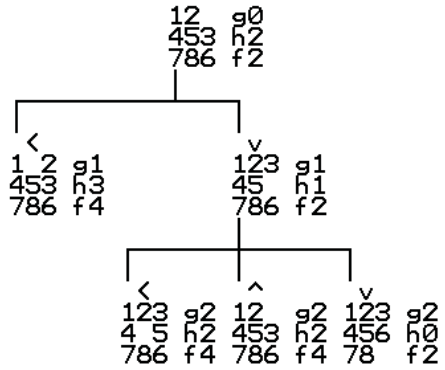
   where InitGoalFile is in the format below.

```
Initial state:
1 2 0
4 5 3
7 8 6

Goal state:
1 2 3
4 5 6
7 8 0
```

To demonstrate functioning, your program should display (1) nodes that are visited in sequence and the state of each node, (2) search tree produced in a visually intuitive format, (3) the number of nodes in the search tree (the tree size), and (4) the solution found. An example echo is shown below.

Submit electronically Java source files Board.java, PuzzleProblem.java and PuzzleAgent.java by following the submission instruction at the course website.

```
C:\cis3700\asgn\A1\8puzzle>java PuzzleAgent InitGoal1.txt
Fringe size = 1
Visit node: dep=0 state>>
120
453
786
Fringe size = 2
Visit node: dep=1 state>>
123v
450
786
Fringe size = 4
Visit node: dep=2 state>>
123v
456
780

Search tree size = 6 nodes
 -Hole movements are shown as <, >, ^ and v.
 -g(), h() and f() values are shown for each node.
```




```
Solution to 8-puzzle:
 node: dep=0 state>>
120
453
786
 node: dep=1 state>>
123v
450
786
 node: dep=2 state>>
123v
456
780
```

## Appendix: Source code for classes in search.jar

```
// ObjectPlus.java: An object that can show itself on screen.
// It is task independent.

abstract class ObjectPlus {
  // Show the object on screen.
  abstract void show();

  // For a complex object, show the part with the given index.
  abstract void showPart(int index);
}

// Problem.java: Abstract problem definition for tree search.
// It is task independent and supports both blind and heruistic search.

import java.util.LinkedList;

abstract class Problem {

  ObjectPlus initialState, goalState;
  String strategyHeuristics; // strategy heuristics combination

  ObjectPlus getInitialState() {
    return initialState;
  }

  void setInitialState(ObjectPlus s) {
    this.initialState = s;
  }

  ObjectPlus getGoalState() {
    return goalState;
  }

  void setGoalState(ObjectPlus s) {
    this.goalState = s;
  }

  String getStrategyHeur() {
    return new String(strategyHeuristics);
  }

  void setStrategyHeur(String straheur) {
    strategyHeuristics = new String(straheur);
  }

// Abstract methods

  // Get successors given state s.
  abstract LinkedList getSuccessor(ObjectPlus s);

  // Test if state s is a goal.
```

```
  abstract boolean isGoalState(ObjectPlus s);
}

// Node.java: A node in a search tree.
// It is task independent and supports both blind and heruistic search.

import java.util.LinkedList;

class Node {
  Node parent;
  ObjectPlus state;
  int depth; // depth is equivalent to uniform step cost
  Node child[] = null; // useful for tree printing

  // Create root node with state s.
  Node(ObjectPlus s) {
    parent = null;
    state = s;
    depth = 0;
  }
  // Create a node with state s and parent p.
  Node(ObjectPlus s, Node p) {
    parent = p;
    state = s;
    depth = p.getDepth() + 1;
  }

  boolean isRootNode() {
    if (parent == null) return true;
    return false;
  }

  Node getParent() {
    return parent;
  }

  void setParent(Node p) {
    this.parent = p;
  }

  ObjectPlus getState() {
    return state;
  }

  void setState(ObjectPlus s) {
    this.state = s;
  }

  int getDepth() {
    return depth;
  }

  void setDepth(int d) {
```

```
    this.depth = d;
  }

  // Get path from root to this node in a list with the root as the head.
  LinkedList getPathFromRoot() {
    LinkedList<Node> ll = new LinkedList<Node>();
    Node current = this;
    while(!(current.isRootNode())) {
      ll.addFirst(current);
      current = current.getParent();
    }
    ll.addFirst(current); // take care of root node
    return ll;
  }

  // Given a set ss of succeesor states of this node, construct
  // a list of child nodes.
  LinkedList stateToNode(LinkedList ss) {
    LinkedList<Node> nds = new LinkedList<Node>();
    int scnt = ss.size(); // # successor of this node
    child = new Node[scnt]; // successor pointers
    for (int i=0;i<scnt;i++) {
      Node n = new Node((ObjectPlus) ss.get(i), this);// new succeesor node
      nds.add(n);

      child[i] = n; // point to successor
    }
    return nds;
  }

  void show() {
    System.out.println(" node: dep="+depth+" state>>");
    state.show();
  }

  // Depending the index, show part of the node.
  void showPart(int index) {
    state.showPart(index);
  }
}

// SearchAgent.java: Agent who solves a problem by tree search.
// It is task independent and supports both blind and heruistic search.

import java.util.LinkedList;

abstract class SearchAgent {
  Problem problem;
  LinkedList<Node> tree;
  LinkedList<Node> fringe;
  LinkedList solution;

  void setProblem(Problem p) {
```

```
    this.problem = p;
  }

  // Search for solution.
  // Post: solution variable is set.
  // Note:
  //   Search tree and fringe are separately maintained using LinkedList.
  //   One copy of each node is stored and it is referenced by both lists.
  void search() {
    Node root = new Node(problem.getInitialState());
    tree = new LinkedList<Node>(); // search tree
    fringe = new LinkedList<Node>(); // fringe
    tree.add(root);
    insertFringe(root, fringe); // one copy of root is referenced in both lists

    while (fringe.size() != 0) {
      System.out.println("Fringe size = " + fringe.size());
      Node n = (Node) fringe.removeFirst(); // node to visit
      System.out.print("Visit"); n.show();
      ObjectPlus nodeState = n.getState();
      if (problem.isGoalState(nodeState)) {
        solution = n.getPathFromRoot();
        return;
      }
      LinkedList successors = problem.getSuccessor(nodeState); // child states
      LinkedList childnodes = n.stateToNode(successors); // child nodes

      for (int i=0;i<childnodes.size();i++) {
        tree.add(((Node) childnodes.get(i)));
        insertFringe((Node) childnodes.get(i), fringe);
      }
    }
    return;
  }

  abstract void showSolution();
  abstract void showTree();
  abstract void insertFringe(Node nd, LinkedList<Node> ll);
}
```