Assignment Submission Instruction

CIS*3700 (W21)

Assignment marking provides important feedback in learning to individual students, but is also resource intensive. This Assignment Submission Instruction is designed to promote good time management and professionalism, to make submission convenient and efficient to the student, and to ensure efficient marking.

1. **[Requirement]**

   For each assignment, **each student should submit 2 files**, specified below:

   All short-answer problems in the assignment are to be submitted

   (a) as a **single** file in **pdf** or **MS Word** format that is **legible**

   (b) with clear **identification**

   (c) with the **same problem order** as the assignment and

   (d) by the **due date and time** to the **CourseLink Dropbox**.

   The Java programming problem in the assignment is to be submitted

   (e) as a **single** compressed **.zip** file including all **Java source files** and

   (f) a plaintext **readme.txt** file

   (g) by the **due date and time** to the **CourseLink Dropbox**.

   Use the above as a check list for completing submission. Details on each item are given below.

2. **[Order of solutions]** Short-answer solutions should follow the **same order** of problems as they appear in the assignment. Index each solution in the same way as in the assignment. That is, start with the solution of problem 1 and index it as so, followed by the solution of problem 2, and so on. If the solution on problem 1 is followed by the solution of problem 3, then problem 2 is marked as missing.

3. **[Identification]** For each submission, indicate student name, student ID, course number, and the assignment number. For Java program submission, include this info in readme.txt.

4. **[File readme.txt]** In readme.txt file, include identification info, instructions on how to run the program, and optional info (if any) to help marking the program.

5. **[Format of short-answer solutions]** Solutions can be written **manually**, by **word processors**, or by **combining both**. However, the submission should be in pdf or MS Word format. Submissions in other formats will receive a zero. For hand-written solutions, scan/photo pages into a single pdf file.

6. **[Generating a single pdf file]** No matter how many pages the solutions have, submit a single file. Submission of multiple files splitting the same assignment will receive a zero. Combine multiple pdf files, e.g., some by scanning and some saved by MS Word, into a single file as follows.

   In Windows, use File Explorer, highlight pdf files to be combined, right-click mouse to select "Combine supported files in Acrobat ...", and save into a single pdf file. The utility allows you to alter the order of files to be combined. Make sure that pages are combined in the correct order. Use a similar utility in Mac.

7. **[Legibility]** Scanned or photoed handwriting in submission must be legible. To do so, the writing should be **legible** and a **sufficiently high resolution** should be used in scanning or photo taking.

   An illegible solution will receive a zero with the reason noted. A submission with illegible solutions for multiple problems will **not be marked**. A zero will be assigned with the reason noted.

8. **[Java program compression]** It is suggested to compress Java source files into a .zip file from a Windows computer. Mac users should check to ensure their compressed files are Windows compatible, as problems have occurred in the past with Mac compressed files. Submissions not in .zip format will receive a zero.

9. **[Timeliness]** Timely submission is essential for timely marking and feedback to students. See Course Outline for the submission graceful period and deduction on late submissions.

10. **[Submission to Dropbox]** Submit to the CourseLink Dropbox. Do not email to the Instructor or TA. An assignment **receives a zero** if the submission is absent in the Dropbox after the graceful period.

11. **[Multiple Submissions]** Multiple submissions of an assignment are a source of confusion. Avoid multiple submissions by checking assignment carefully before submitting.

    (a) If the submission has a wrong format, a wrong problem order, or poor legibility, resubmit as soon as possible but no later than the graceful period.

    (b) The **last** resubmission will be marked. If it is after the due date, it will be marked as a late submission.

Your cooperation in assignment submission by following the above instruction is highly appreciated.

CIS 3700 (Winter 2021): Introduction to Intelligent Systems

ASSIGNMENT 1

Instructor: Y. Xiang Assigned: Thur., Jan. 21, 2021. Due: 11:30 PM, Thur., Feb. 11, 2021.

All assignments of this course are to be completed by individual students independently. No program code developed in whole or in part by someone else (other than those provided to the class) should be included in submission for programming problems. Please ensure academic integrity in your work.

Problems (Total marks: 40)

1. (2 marks) [Agent environment] Table tennis robots that are in the market deliver balls to a human player for training, but cannot engage the human player with competitive games. Consider the task environment of a truly competitive table tennis robot agent. Categorize the environment in terms of seven properties discussed in lectures. Justify each choice clearly with reasons.

2. (3 marks) [Environment representation] A medical diagnostic agent needs to determine whether a patient suffers from Covid-19. 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. At least one variable must be discrete and at least one must be continuous. At least one discrete variable must have a range space of 3 or more values.

   (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 set $\{var_1 = v_1, var_2 = v_2, ...\}$.

   Hint: This problem practices how to encode environment states, not how to become a medical specialist.

3. (4 marks) [Measuring size of problem]

   Consider tree search to solve Sudoku puzzle. The initial state of a puzzle is shown. Successors of a state depends on selection of an empty cell. Once the cell is selected, the number of successors is the number of alternative digits to fill in the cell, while respecting Sudoku rules. Estimate branching factor b of the search tree, depth d of the shallowest goal node, and max path length m.

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

   (a) (1 marks) Denote cell at ith row and jth column as cell (i, j). In the above puzzle, cell $(1, 3)$ is 5. Let the above state be the root node of a search tree. What are the successors of the root node if the empty cell selected is cell $(3, 8)$?

   (b) (2 marks) To estimate b, copy the given puzzle P to puzzle M. For each empty cell in M, do the following 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?

   (c) (1 mark) What are the values of d and m of the search tree, and why?

4. (6 marks) [Tree search with BFS, DLS, and IDS] Consider a fully expanded search tree, where the root is labeled by integer 0. Successors of node n are specified by function $Successor(n) = \{2n+1, 2(n+1)\}$. When $2n+1 > 14$, the node n has no successor. The goal state is 11.

   (a) (1 mark) Show the fully expanded search tree. For node n, draw successors below n, with successor $2n+1$ to the left, and $2(n+1)$ to the right.

   (b) (1 marks) During tree search, the 1st node *generated* is the root. When a node is *visited*, it is 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 tie in order from left to right.

   List the order in which nodes are generated by tree search with BFS, and order in which nodes are visited.

   (c) (2 marks) DLS allows removal of nodes from memory as DFS. List the order in which nodes are generated by tree search with DLS using depth limit 3, the order in which nodes are visited, as well as the order in which nodes are removed from memory.

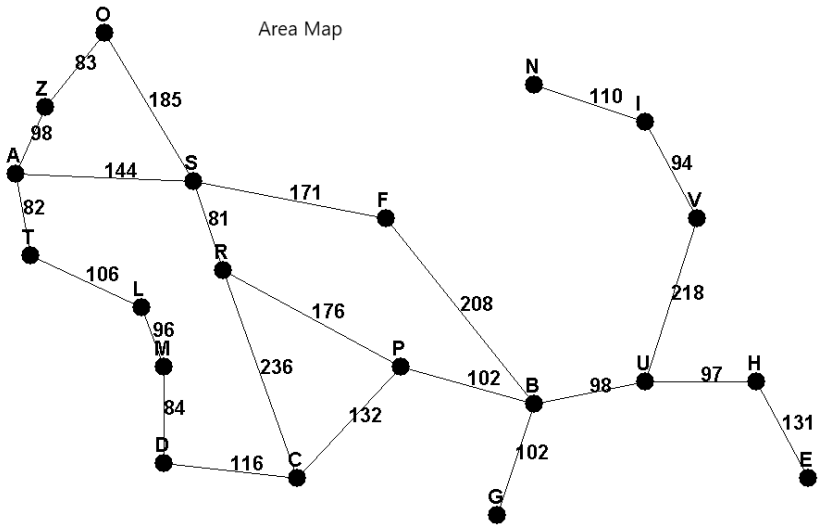
   (d) (2 marks) List the order in which nodes are generated by tree search with IDS, the order in which nodes are visited, as well as the order in which nodes are removed from memory. If a node is visited with depth limit i, and visited again with limit $i+1$, then it appears in the order each time.

5. (3 marks) [Routing by BFS tree search]

   An area map of multiple cities is shown. Solve the routing problem from city O to city B using Breadth-First Search (BFS).

   (a) (2 marks) Show search tree produced. When expanding a node, generate child nodes in alphabetical order, and draw from left to right. Show number of nodes generated. Show the solution as a sequence of env states.

   (b) (1 mark) Is the solution optimal, and why?

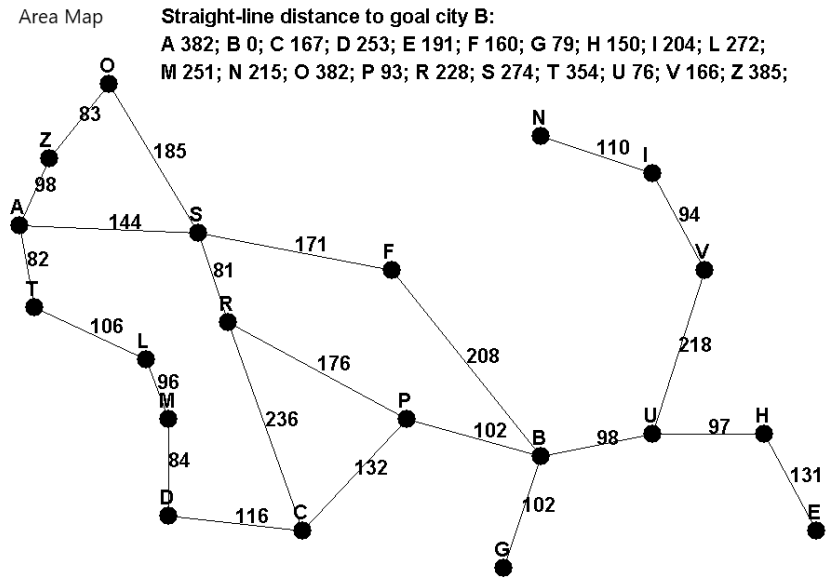


6. (3 marks) [Routing by A* tree search]

   An area map is shown. Solve the routing problem from city O to city B using A* tree search with straight-line distance heuristics.

   (a) (2.5 marks) Show search tree with alphabetical order on child nodes from left to right. For each node n, show evaluation function value in the form $g(n)+h(n) = f(n)$, e.g., 118 + 329 = 447.

   (b) (0.5 marks) Show number of nodes generated. Show the solution as a sequence of env states.



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

2	8	3
	6	4
1	7	5

1	2	3
8		4
7	6	5

Use Manhattan heuristic function defined by

$$h(n) = \text{sum of distances of tiles (not hole) from goal positions at state n.}$$

For instance, $h(r) = 7$, where tile 8 contributes 2 to the sum. 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) (2.5 marks) 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) (0.5 marks) Show the solution as a sequence of states. Show also the corresponding sequence of actions.

8. (3 marks) [Monotonicity and admissibility] Analyze the above Manhattan heuristic function $h(n)$.

   (a) (2 marks) Show that $h(n)$ is monotonic.

   (b) (1 marks) Show that $h(n)$ is admissible.

   (Hint: An example for a specific state n is insufficient, as it cannot ensure the condition for another state.)

9. (13 marks) [A* with misplaced-tiles heuristics]

   [Task] Implement A* tree search in Java that solves 8-puzzle with the misplaced-tiles heuristics.

   [Heuristics] The misplaced-tiles heuristic function is defined as

$$h(n) = \text{number of misplaced tiles (not hole) at state n.}$$

   For the initial state r and goal state t in Problem 7, we have $h(r) = 5$.

   [Generic Java classes] Organize Java classes in 2 sets: The 1st is generic to tree search, and is independent of search strategy as well as 8-puzzle task. They are given as file $search.jar$ (at CourseLink website) with source code in Appendix.

   (a) ObjectPlus: An object of this abstract class represents a generic env state. It can return step cost for the action leading to this state. It can show itself in console.

   (b) Problem: An object of this abstract class represents a problem definition. It maintains initial state and goal state of the problem. Abstract methods for successor function and goal test must be implemented by subclass. It maintains search strategy, which influences info kept at each state, such as $h(n)$ value (needed by A* but not by BFS).

   (c) Node: An object of this class represents a node in search tree (see lecture slide 7). The env state for the node is an object of class ObjectPlus.

   (d) SearchAgent: An object of this class is a tree search agent. It maintains problem definition, a search tree, and its fringe. Abstract methods for fringe insertion, search tree plotting, and solution printing must be implemented by subclass.

   [8-puzzle related Java classes] The 2nd set includes classes specific to 8-puzzle. You should implement them as below, and submit Java source code.

(a) Game.java: An object of this class describes a 8-puzzle state. It maintains tile and hold locations, the move leading to this state, and heuristic function value of the state. Implement as subclass of ObjectPlus.

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

(b) PuzzleSpec.java: An object of this class defines an instance of 8-puzzle problem. It should be a subclass of Problem class, and must implement methods for successor generation and goal test.

Generate successors in 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) Do8Puzzle.java: An object of this class is a 8-puzzle agent. The class should be subclass of SearchAgent. It loads initial state and goal state from a file. It must implement methods for fringe insertion, search tree plotting, and solution printing as required by SearchAgent.

Echo 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.

[Compilation and run] To facilitate marking, all java source files and the search.jar file must be placed in a directory named $3700A1$ followed by your first name, e.g., $3700A1Yang$. No "package" statement should be used in any source code.

All java classes must be compilable by the following command issued at the above directory, e.g.,

```
C:\Users\3700A1Yang> javac -cp .;search.jar *.java
```

Your implementation must be executable by the following command issued at the above directory, e.g.,

```
C:\Users\3700A1Yang> java -cp .;search.jar Do8Puzzle InitGoalFile
```

[Input file] The input InitGoalFile (at CourseLink website) has the format below, where 0 indicates the hole:

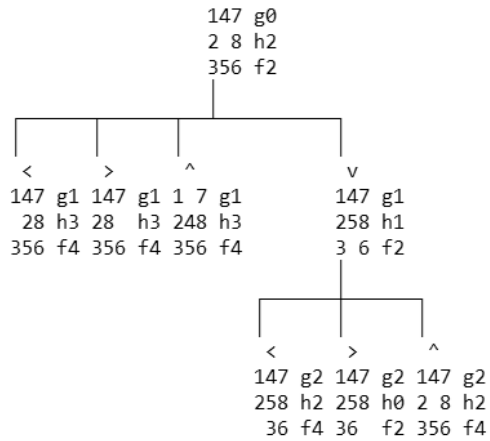
```
Initial state:
1 2 0
4 5 3
7 8 6

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:

```
  Fringe size = 1
Visit node at depth 0 of state:
  147
  208
  356
  Fringe size = 4
Visit node at depth 1 of state:
  147 v
  258
  306
  Fringe size = 6
Visit node at depth 2 of state:
  147 >
  258
  360

Search tree size = 8 nodes
 -Hole movements are shown as <, >, ^ and v.
 -g(), h() and f() values are shown for each node.
```




```
Solution to 8-puzzle:
 node at depth 0 of state:
  147
  208
  356
 node at depth 1 of state:
  147 v
  258
  306
 node at depth 2 of state:
  147 >
  258
  360

Action sequence for solution:
  (Down,Right)
```

[Submission] Submit electronically Java source files Game.java, PuzzleSpec.java and Do8Puzzle.java by following Assignment Submission Instruction.

Appendix: Source code for classes in search.jar

```
// ObjectPlus.java: A generic env state.

abstract class ObjectPlus {
  // Get step cost from parent env state.
  abstract int getStepCost();

  // Show the state on screen.
  abstract void show();

  // For complex state, show part by index.
  abstract void showPart(int index);
}

// Problem.java: Problem for tree search.

import java.util.LinkedList;

abstract class Problem {

  ObjectPlus initialState, goalState;
  String strategy; // search strategy

  ObjectPlus getInitialState() {
    return initialState;
  }

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

  ObjectPlus getGoalState() {
    return goalState;
  }

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

  String getStrategy() {
    return new String(strategy);
  }

  void setStrategy(String stra) {
    strategy = new String(stra);
  }

// 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 search tree node.

import java.util.LinkedList;

class Node {
  Node parent;
  ObjectPlus state;
  int depth;
  int pathCost; // path cost from root
  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;
    pathCost = p.getPathCost() + s.getStepCost();
  }

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

  Node getParent() {
    return parent;
  }

  ObjectPlus getState() {
    return state;
  }

  int getDepth() {
    return depth;
  }

  int getPathCost() {
    return pathCost;
```

```
  }

  // 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 succeesor states of this node, add child nodes to tree.
  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 at depth "+depth+" of state:");
    state.show();
  }

  // Show part of the node by index.
  void showPart(int index) {
    state.showPart(index);
  }
}

// SearchAgent.java: Tree search agent.

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: value of solution is set.
  // Note: Search tree and fringe are maintained separately.
  //       Each node has 1 copy and is referenced by both.
  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);

    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();  // show node
      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);
}
```