School of Computer Science and Software Engineering

Sample Questions 2008

CITS4211 AI

This paper contains: 7 pages (including the title page)
Time allowed: 2 hours 10 minutes

Rubric from the paper:
The paper contains two sections. Each section contains 3 questions.
Candidates should attempt TWO (2) questions from EACH section.
Marks for this paper total 60.

Instructions

The rubric for the actual exam paper is as shown on the previous page.

The paper will contain two sections. Each section contains 3 questions.
Candidates should attempt TWO (2) questions from EACH section (4 questions in total).
Each question is worth 15 marks. Marks for this paper total 60.

If more than two questions have been attempted in any section, only the *first* two as submitted will be marked. If you change your mind after starting a question you should put a cross through the whole question and write "Not Attempted".

Two sample questions for the first section are provided overleaf. (See the help4211 noticeboard for more information about exam content.) These are provided to give you an idea of the size and flavour of the questions — obviously the real exam will include additional material.

Note also that the appropriate chapters of the text contain a wealth of problems that you can use to test your understanding of the material if you wish. You should also review your lab work.

1.

(a) Briefly compare *iterative deepening* search with *breadth-first* and *depth-first* search. Your answer should consider the four main criteria on which search algorithms are usually compared. You may assume unit step cost (and finite branching).

You may answer in point form. For each point, you should state how the search performs under that criterion, and briefly why. For example:

Breadth-first

- **Completeness**: yes — since the children of a visited node are added to the end of a finite queue, all nodes will eventually be visited
-

(6)

(b) A *general search* algorithm can be described as follows:

General Search

Given a *start state* $s_0 \in S$, a *goal function* $g(s) \rightarrow \{true, false\}$, and a *terminal condition* $t(s) \rightarrow \{true, false\}$:

Initialise a set $U = \{s_0\}$ of *unvisited* nodes containing just the start state and an empty set $V = \{\}$ of *visited* nodes.

(i) If U is empty halt and return `null` (no goal found).
(ii) Select, according to some (as yet undefined) *strategy*, a node s from U.
(iii) If $g(s) = true$ halt and return the goal node.
(iv) If $t(s) = true$ discard s and repeat from 1.
(v) Otherwise move s to the set V, and add to U all the nodes reachable from s. Repeat from (i).

Complete the method `search` in the following class to implement the general search. Recall that `Actions` is an `ArrayList`. Assume `Node` contains a method `update(Action a)` that applies an action to the node to obtain its successor.

```
package search;
import agent.*;
import java.util.*;

public abstract class GeneralSearch {

  Environment environment;
  UtilityFunction utilityFunction;
  ArrayList unvisited, visited;

  public GeneralSearch (Environment environment,
                        UtilityFunction utilityFunction) {
    this.environment = environment;
    this.utilityFunction = utilityFunction;
    unvisited = new ArrayList();
    unvisited.add(new Node(environment, new Actions()));
    visited = new ArrayList();
  }

  public Node search () {
    Actions arcs;
    Action arc;
    Node visit, successor;
    while ___________________ {
      visit = select();
      if (utilityFunction.goal(visit.environment)) return visit;
      else if (!_________________________) {
        arcs = visit.environment.getActions();
        ________________________________
        ________________________________
        ________________________________
        ________________________________
        ________________________________
        ________________________________
      }
      visited.add(visit);
    }
    return null;
  }
```

```
    public abstract Node select ();
    // removes a node from the unvisited list

    public abstract void insert (Node node);
    // inserts a node in the unvisited list

    public abstract boolean terminalCondition (Node node);
    // returns true if node satisfies the terminal condition
  }
```

(5)

(c) Describe how the above class can be extended or supplemented to provide a *depth-first* search, a *breadth-first* search, and an *iterative deepening* search.

(4)

2.

(a) What do we mean by a heuristic? How is a heuristic used in the A* algorithm? What property of a heuristic ensures A* is optimal?

(3)

(b) What problem with A* is *iterative deepening A** (IDA*) designed to overcome? Briefly describe, using diagrams if necessary, how it achieves this while at the same time preserving optimality.

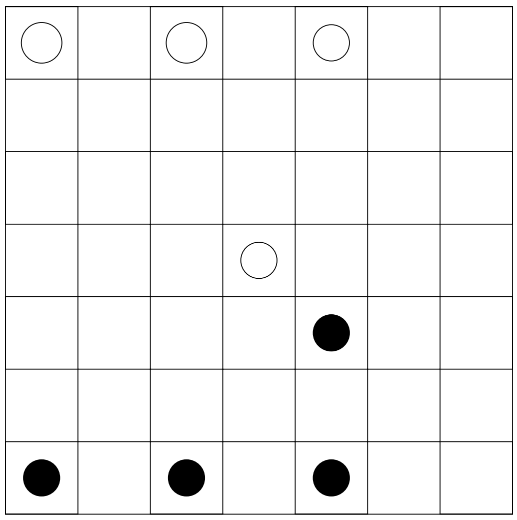
(3)

(c) Under what situation does IDA* reach a worst-case time performance and why? If N nodes are expanded by A*, give a "big O" expression for the number of nodes expanded by IDA* in the worst case, showing how this expression is derived.

Briefly explain how this performance can be improved by trading accuracy for time performance.

(5)

(d) The following picture shows a game of Finish First (a draughts-like game in which the draughts are moved one square diagonally, and the aim of each player is to get all of their pieces to the last row of the board). White is moving up the board, and it is white's turn to move.



Note that (although black *should* win) either player can win from this position. Applying a standard minimax algorithm, however, white may be equally likely to choose a move which prevents it from having any chance of winning. Why is this?

(2)

(e) Briefly describe the *horizon problem* brought about by depth-limited searches.

(2)
