

Massachusetts Institute of Technology

# 16.410/13 Principles of Autonomy and Decision Making

## Problem Set #2

Problem set due Session 6

### Objective

You will implement in Java the following search algorithms:

1. Depth-first
2. Breadth-first
3. Depth-Limited
4. Iterative Deepening

And perform an empirical evaluation of three methods (depth-first-search, breadth-first-search, and iterative-deepening on a set of provided benchmark examples.

You will demonstrate good programming style including the appropriate use of exceptions, comments, interfaces, and classes.

You will use the implemented search algorithms to solve several problems. You will first demonstrate your algorithms on a simple test case (Problem 2) and then you will run your algorithms on a sequence of five different graphs and analyze the results (Problem 3).

In addition, the second objective of this problem set is to develop your skill at performing an asymptotic analysis of uninformed search.

### Background

Implement according to the lecture notes. Use AIMA chapter 3 for background reading but implement according to the lecture notes (not AIMA).

Uninformed search methods were developed in the notes for lectures three, four and five. Lecture three developed the basic concepts, pseudo-code and application to simple problems. Lecture four presented background on implementation within Java, while Lecture five introduced asymptotic performance analysis. Additional background on uninformed search is provided in Chapter 3 of AIMA.

Asymptotic analysis is a powerful method for analyzing computational systems. We recommend that those who are interested in learning more about asymptotic analysis read any of Chapters 1 - 4 of "Introduction to Algorithms," by Thomas Cormen, Charles Leiserson and Ronald Rivest, MIT Press.

## Problem 1 Implementation

Implement a general search capability with methods to the following search methods: breadth-first-search, depth-first-search, depth-limited-search, and iterative-deepening-search.

Write `JUnit` tests for each of these search algorithms. Demonstrate that your implementations of all four search algorithms work correctly and handle exceptional cases such as a data mal-formed data structures or failed searches gracefully.

Your search implementation must observe the following specifications in order for our automated testing and grading to work (and for you to get credit for your work).

Begin by implement the following search classes:

```
interface GeneralizedSearch {
    Path search (Object startstate, Object goalstate);
    List ExpandPath (Object astate);
    int getNumberOfVerticesExpanded ();
    int getMaximumQueueSize ();
    int getNumberOfVerticesVisited ();
}

class DepthFirstSearch implements GeneralizedSearch
{ /* Implement this class */ }
class BreadthFirstSearch implements GeneralizedSearch
{ /* Implement this class */ }
class DepthLimitedSearch implements GeneralizedSearch
{ /* Implement this class */ }
class IterativeDeepeningSearch implements GeneralizedSearch
{ /* Implement this class */ }
```

The specified methods do the following:

```
Path search (Object startstate, Object goalstate);
```

Searches starting with the `startstate` until it finds a simple path to the `goalstate`. It should return a path which is a list of the states starting at the `startstate` and ending with the `goalstate`. If no solution is found it should return `null`. The search should avoid cycles by using a visited list as described in the lectures.

```
List ExpandPath (Object astate);
```

Given a state returns a list of states reachable in one step from that state.

```
int getNumberOfVerticesExpanded ();
```

Upon completion of a search will return the number of vertices that were expanded during the search. This should be reset to zero at the start of each search.

```
int getNumberOfVerticesVisited ();
```

Upon completion of a search will return the number of vertices that were visited during the search. This should be reset to zero at the start of each search.

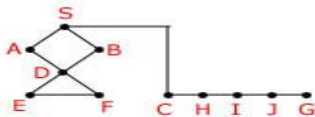
```
int getMaximumQueueSize ();
```

Upon completion of a search will return the maximum queue size reached during the search. This should be reset to zero at the start of each search.

Hints: You might find it useful to use the FIFO and LIFO queues that you implemented in Problem Set 1.

## Problem 2 Search

We are trying to find a path from S to G in the following graph:



Connections (all undirected):

S: A B C

A: S D

B: S D

C: S H

D: A B E F

E: D F

F: D E

G: J

H: C I

I: H J

J: I G

The graph has been provided in XML format (ps2data.xml) and a reader of the file has been provided.

Here is how it works. The XML file for the above graph looks like this:

```
ps2data.xml
<GRAPH>
  <VERTEX name="S">
    <EDGE vertex="A"> </EDGE> <EDGE vertex="B"> </EDGE> <EDGE vertex="C"> </EDGE>
  </VERTEX>
  <VERTEX name="A">
    <EDGE vertex="S"> </EDGE> <EDGE vertex="D"> </EDGE>
  </VERTEX>
  <VERTEX name="B">
    <EDGE vertex="S"> </EDGE> <EDGE vertex="D"> </EDGE>
  </VERTEX>
  <VERTEX name="C">
    <EDGE vertex="S"> </EDGE> <EDGE vertex="H"> </EDGE>
  </VERTEX>
  <VERTEX name="D">
    <EDGE vertex="A"> </EDGE> <EDGE vertex="B"> </EDGE> <EDGE vertex="E"> </EDGE>
    <EDGE vertex="F"> </EDGE>
  </VERTEX>
  <VERTEX name="E">
    <EDGE vertex="D"> </EDGE> <EDGE vertex="F"> </EDGE>
  </VERTEX>
  <VERTEX name="F">
    <EDGE vertex="D"> </EDGE> <EDGE vertex="E"> </EDGE>
  </VERTEX>
  <VERTEX name="G">
    <EDGE vertex="J"> </EDGE>
  </VERTEX>
  <VERTEX name="H">
    <EDGE vertex="C"> </EDGE> <EDGE vertex="I"> </EDGE>
  </VERTEX>
  <VERTEX name="I">
    <EDGE vertex="H"> </EDGE> <EDGE vertex="J"> </EDGE>
  </VERTEX>
  <VERTEX name="J">
    <EDGE vertex="I"> </EDGE> <EDGE vertex="G"> </EDGE>
  </VERTEX>
</GRAPH>
```

As you can see, the XML consists of one <GRAPH> ... </GRAPH> which contains all of the vertices (<VERTEX></VERTEX>) each of which contains edges to other names vertices.

We have provided Java classes for the vertices and edges as follows:

```
Ps2Vertex.java
import java.util.LinkedList;
import java.util.List;

public class Ps2Vertex {
    protected LinkedList edges;
    protected String vertexname;

    Ps2Vertex (String vertexname) { this.vertexname=vertexname; edges=new LinkedList(); }
    String getName () { return vertexname; }
    void addEdge (Ps2Edge anEdge) { edgess.addLast(anEdge); }
    int numConnections () { return edges.size(); }
    Ps2Edge getEdge (int n) { return ((Ps2Edge)edges.get(n)); }
    LinkedList getConnections () { return edges; }
}

```

```
Ps2Edge.java

public class Ps2Edge {
    protected Ps2Vertex connectedto;

```

```
    Ps2Edge (Ps2Vertex vertex) { connectedto=vertex; }  
    Ps2Data getVertex () { return connectedto; }  
}
```

The provided XML reading code (Ps2XMLGraph.java and associated XMLTag.java, XMLAttribute.java, and XMLStream.java) can be used to read in the Graph data as follows:

```
// Read in the graph  
Ps2XMLGraph xmlreader=new Ps2XMLGraph();  
Dictionary vertices=xmlreader.readGraph("ps2data.xml"); // Read the Graph  
// Find the start and end vertices  
Ps2Vertex startVertex=(Ps2Vertex)vertices.get("S"); // Find the start node  
Ps2Vertex endVertex=(Ps2Vertex)vertices.get("G"); // Find the goal node
```

Run your four algorithms on this (ps2data.xml) data and present your results in the form of a table including the metric information for each of the four runs.



## **Preview of next assignment**

Next weeks assignment will include (but not be limited to) extending your implementation to include A\*. The unnumbered problem below will be part of next week's assignment. We provide this preview so that you can (in the case of A\*) keep in mind what you will have to do to generalize your software to work with informed search. Since the problem below depends mostly on the software that you develop in this assignment you might want to get a head start on next weeks assignment by working on this problem in advance. Please do NOT hand in this problem with this assignment.

## **Problem ? Empirical Evaluation**

We have provided a collection of graphs as XML files in files gs1.xml, gs2.xml, gs3.xml, gs4.xml, and gs5.xml. In each case search from node "S" to node "G" and analyze the metric information from each run using graphs and to aid in your analysis.

Construct an example for each algorithm in which it would perform poorly compared to the others.

Compare relative performance of BFS, DFS, and IDA in terms of memory and space as a function of graph size and connectivity.

Comment on why each algorithm performs well or poorly.