Assigned: Tuesday 6 September 2011

Due: Monday 12 September 2011, 11:59 p.m.

Objectives:

- Understand details of of uninformed search algorithm implementations
- Explore performance implications of various search algorithms
- Practice reading and programming to specified interfaces
- Reinforce programming with higher-order procedures in Scheme
- Apply principles of good science writing

Collaboration: This laboratory will be completed in pairs assigned by the instructor.

1 Introduction

In this assignment, you will complete a general framework for implementing a variety of uninformed search algorithms, measuring their efficiency and solution quality. While the textbook gives a graph search version of most specific search algorithm implementations, we will be using a general tree search algorithm, which may be found below.

Algorithm 1 General tree-search algorithm requiring a starting state for a problem as well as a method for organizing nodes in the frontier.

```
function TREE-SEARCH(start, problem, ENQUEUE) returns a solution, or failure
    node ← NODE-INIT(start)
    if GOAL(start) then return SOLUTION(node)
    frontier ← QUEUE(node)
    do
        if EMPTY( frontier) then return failure
        node ← POP(frontier)
        if GOAL(problem, STATE(node)) then return SOLUTION(node)
        frontier ← ENQUEUE( EXPAND(problem, node), frontier)
```

```
function EXPAND( problem, node) returns a set of nodes
    successors ← []
    for each action in ACTIONS( problem, STATE(node) )
        if CONTAINS( PATH(node), RESULT(action,node)) then continue
        successors ← [ CHILD-NODE(problem, node, action) | successors ]
```

Many search algorithms can be implemented using this general structure. The only difference between various search algorithms is in how they order (enqueue) nodes on the frontier. Thus, the ENQUEUE parameter is actually a procedure that implements a particular ordering criterion, which determines the precise search algorithm.

Note that the call to CONTAINS ensures no state will be repeated in any given solution. Unfortunately, this is *not* the same as making sure no state is examined more than once anywhere in the search tree.

2 Code and environment

For this assignment, you will need to copy some starter code from the MathLAN directory:

```
~weinman/courses/CSC261/code/search
```

Note that Scheme is not all that intelligent about dealing with relative paths. Thus, any Scheme `load` commands that are issued will be relative to the working directory. You should not change the code's `load` commands to use absolute paths, as this makes running your code more difficult for grading purposes.

2.1 General search

2.1.1 Search routines: `search.scm`

The file `search.scm` contains skeletons and specifications for many of the procedures you are to write. In order to promote code reuse, the search routines take a generic kind of value we're calling a `problem` and operates on a `node` type. Both of these are described below.

2.1.2 Generic search problems: `problem.scm`

Like the text, a problem requires the ability to define a goal state, a method for generating successors (states that result from the actions available at a given state), and a cost for taking an action in a given state. Thus, we can create a `problem` by passing in these three procedures to a procedure `make-problem` that encapsulates them all. Once these are tied together, the general procedure `problem-expand-node` (akin to EXPAND above and already written for you) has everything needed to generate the list of potential actions and their resulting states. The accompanying 6-P documentation has further detail.

2.1.3 Representing search tree nodes: `node.scm`

To find the solution to a problem, it is important to keep track of various aspects of our search. We do this by means of a `node` type. As in AIMA 3.3.1, a node encapsulates a state, the state resulting from that action, action, the "parent" node preceding it in the search tree, and a path cost for taking the action from the parent. We also add to this structure the (estimated) total cost of the solution through this node (to be used in the next assignment), and the depth of the node in the search tree. All of this information is packaged together in type we call a `node`.

In addition, we will need to create a start node, the root of our search tree, using an initial state and any heuristic procedure for the problem; this is what the procedure `node-init` does. Note that the parent of the initial node is `null`.

Some of the searches (including the `uniform-cost-search` provided) may require you to sort the successor nodes and/or queue by some criterion. Thus, a simple insertion sort has been provided for you in `sort.scm`.

2.2 Lights Out

In addition to the 8-puzzle sliding block problem described in AIMA 3.2.1, you will apply your search routines to the so-called lights out puzzle.[1] The puzzle consists of a square grid of lights, which may be toggled on or off.

States: The grid has side-length s, meaning there are $\ell = s^2$ lights. A state description specifies the status of all lights.

Initial state: Any state can be designated as the initial state. However, some initial states have no solution, so generating a solvable model is best done by beginning with the all-off state and applying a sequence of toggle actions.

Actions: Whereas each light may be toggled, each state has ℓ actions.

[1] Barile, Margherita. "Lights Out Puzzle." From MathWorld–A Wolfram Web Resource, created by Eric W. Weisstein. http://mathworld.wolfram.com/LightsOutPuzzle.html

Transition model: When a specified light is toggled, all of its horizontal and vertically adjacent neighbors in the grid are toggled as well.

Goal test: This checks whether all lights are off.

Path cost: Each step has unit cost.

2.3 Summary

The following gives a table of function/procedure equivalencies between the pseudo-code and the provided Scheme code.

Pseudo code	Scheme	File
NODE-INIT	`(node-init` *`state heuristic`*`)`	`node.scm`
GOAL	`((problem-goal?` *`problem`*`)` *`state`*`)`	`problem.scm`
EXPAND	`(problem-expand-node` *`problem node heuristic`*`)`	`problem.scm`

For now, we will ignore the `heuristic` parameter, which is used on the next assignment. For now, you can simply use

```
(lambda (state) 0)
```

as a basic state evaluation function that always returns zero.

Here is an example that uses breadth-first and uniform-cost search to solve a 3×3 lights-out puzzle with 4 random moves.

```
(load "search.scm") (load "lightsout.scm")
(random-seed 42)
(define side 3)
(define start (random-lights-out-state side 4))
(define bfs-sol (breadth-first-search start (lights-out-problem side)))
(define ucs-sol   (uniform-cost-search start (lights-out-problem side)))
(display (list 'bfs (length (car bfs-sol)) (cadr bfs-solution))) (newline)
(display (list 'ucs (length (car ucs-sol)) (cadr ucs-solution)))
```

This produces the following output:

```
(bfs 2 33)
(ucs 2 419)
```

In the event you do not complete Problem 1 below, or you wish to benchmark your implementations, a complete compiled version is provided. Copy the directory (use `cp -R` to recursively copy the entire contents)

```
/home/weinman/courses/CSC261/code/search/compiled
```

to the location of your other Scheme files and add

```
(require "search.scm")
```

rather than a load of your own Scheme file. Note that you will have to rename *your* `search.scm` file to something else to avoid any conflicts.

3 Lab assignment

Problem 1 - Writing search algorithms [50 points]

Part A [25 points]

Implement the routine `search`, as defined in `search.scm`. This is the general tree search algorithm as given above. Note that the `problem` parameter may be used with the routines given in `problem.scm`, in particular `problem-goal?` and `problem-expand-node`.

Per the specification, you will need to track `expansions`, a count of the number of times you call the problem's successor function `problem-expand-node`. When you have found a node that is a solution, `search` should return

`(list (node-extract-actions node) expansions)`

As an example, the procedures `depth-first-search` and `uniform-cost-search` both *call* `search` with a specific enqueueing procedure. Both of these searches are uninformed, so we use the always-zero heuristic given above.

Part B [5 points]

Write the procedure `breadth-first-search` by calling your `search` routine with an appropriate enqueueing procedure. *(Hint: Follow the example of `depth-first-search`.)*

Part C [10 points]

Write the procedure `depth-limited-search` by calling your `search` routine with an appropriate enqueueing procedure. *(Hint: Do not enqueue a node whose depth exceeds the given limit.)* The procedure `(node-depth` *`node`*`)` from `node.scm` will be helpful. Your implementation need not distinguish between cutoff and standard failures.

Part D [10 points]

Write and document the procedure `iterative-deepening-search` by repeatedly calling your `search` routine with an appropriate enqueueing procedure. *(Hint: Use `depth-limited-search`.)*

Problem 2 - Analysis [50 points]

In this problem, you will do some comparative analysis of your search routines.

Note that, while `random-eight-puzzle-state` and `random-lights-out-state` call `random` to produce their states, you can make your results repeatable by using the procedure `(random-seed` *`seed`*`)` to set the seed of the random number generator.

Part A

Generate a fairly easy eight-puzzle state and an easy lights-out state. Run each search algorithm on both, creating two tables (one for each problem) listing the number of nodes expanded to find a solution, and the total number of actions in the solution. Be sure to specify the specifics of each problem or how they were generated.

Part B

Generate the hardest problems you feel like waiting for solutions to under most search algorithms. Run your search algorithms again, adding rows to the tables you created above. Be sure to specify the specifics of each problem or how they were generated.

Part C

Using the data you have generated, draw some conclusions about the relative efficiency and effectiveness of these search algorithms on these problems.

- How do the number of nodes expanded compare among the search algorithms? How, if at all, does this comparison vary with problem difficulty?
- How do the solution costs compare among the search algorithms? How, if at all, does this comparison vary with problem difficulty?

Note that a complete analysis will feature coherent paragraphs, a brief introduction stating the purpose and context, as well as your overall conclusions. It should be nicely formatted and feature a logical organization, complete sentences, and proper grammar, spelling, and punctuation. The audience is your peers in this class; they do not know *a priori* what problems you examined, nor what your results or conclusions may be.

What to turn in

Your submission should include the following

- Your completed `search.scm` file
- A short driver program that demonstrates *your* search algorithm implementations applied to a simple problem (i.e., one that does not take long to run).
- A single PDF containing (merged)
  - Your Scheme files
  - A transcript of your driver program's output
  - Your analysis essay