

Assigned: Tuesday 28 January

Due: Monday 3 February, 10:30 p.m.

Objectives:

- Understand details 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:

- Problem 1 (programming) will be completed in pairs assigned by the instructor.
- Problem 2 (analysis) must be completed individually.

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(problem, 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 ← []
current ← STATE(node)
for each action in ACTIONS(problem, current)
  next ← RESULT(problem, current, action)
  if not CONTAINS( PATH(node), next) then
    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 the ordering criterion to determine the precise search algorithm.

Note that the call to CONTAINS ensures no state will be (wastefully) repeated within 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
```

Do not change the starter code's `require` commands to use absolute paths, as this makes running your code more difficult for grading purposes. Do not modify the `problem.rkt` and `node.rkt` files provided.

2.1 General search

2.1.1 Search routines: `search.rkt`

The file `search.rkt` 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.rkt`

As our textbook explains, a generic search 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` value by passing in these three elements to a procedure `make-problem`, which encapsulates them all. Once these elements 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.rkt`

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 Section 3.3.1, a node encapsulates

- an action,
- the state resulting from that 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. We use Racket's built-in `keyed-sort` for these.

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, many 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. However, toggling a light twice is the same as never toggling it. We will “disallow” such repeated moves by additionally capturing whether a particular light has been toggled previously so that the number of allowable moves decreases

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

Goal test: This checks whether all lights are off.

Path cost: Each step has unit cost.

2.3 Binary Tree

As a simple means of directly testing your search algorithms’ solutions (and the number of nodes they expand), `bintree.rkt` defines a problem with a simple binary tree whose states are positive integers. The start state is always 1, actions a are 0 (left), or 1 (right), with the state $s' = 2s + a$ resulting from taking action a in state s . The problem has unit path cost, and you may define any state (or no states) to be a goal as well as the largest state.

2.4 Summary

The following table lists function equivalencies between the pseudo-code and the provided Scheme code.

Pseudo code	Scheme	File
NODE-INIT	<code>(node-init state heuristic)</code>	<code>node.rkt</code>
GOAL	<code>((problem-goal? problem) state)</code>	<code>problem.rkt</code>
EXPAND	<code>(problem-expand-node problem node heuristic)</code>	<code>problem.rkt</code>
SOLUTION	<code>(node-extract-actions node)</code>	<code>node.rkt</code>

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, depth-first and uniform-cost search to solve a 3×3 lights-out puzzle with 4 random moves.

```
(require "search.rkt") (require "lightsout.rkt") (require "problem.rkt")
(random-seed 42)
(define side 3)
(define problem (lights-out-problem side))
(define start (random-lights-out-state side 3))
(define bfs-sol (breadth-first-search start problem))
(define dfs-sol (depth-first-search start problem))
(define ucs-sol (uniform-cost-search start problem))
```

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

```
(display (list 'bfs (length (car bfs-sol)) (cadr bfs-sol))) (newline)
(display (list 'dfs (length (car dfs-sol)) (cadr dfs-sol))) (newline)
(display (list 'ucs (length (car ucs-sol)) (cadr ucs-sol))) (newline)
(display ((problem-goal? problem)
         (problem-apply-action-sequence problem start (car bfs-sol))))
```

This produces the following output:

```
(bfs 3 88)
(dfs 3 11745)
(ucs 3 88)
#t
```

In the event you do not complete Problem 1, or you wish to benchmark your implementations, a complete compiled version is provided, which you may copy to a new directory:

```
$ cp -R ~weinman/courses/CSC261/code/searchc ./complete
$ drracket complete/complete.rkt &
```

3 Lab assignment

Problem 1 - Writing search algorithms [40 points]

12 additional points will be allotted for testing and code formatting considerations.

Part A [12 points]

Implement the function `search`, as defined in `search.rkt`. This is the general tree search algorithm as given above. Note that the `problem` parameter may be used with the routines given in `problem.rkt`, 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 they use the always-zero heuristic given above.

Part B [4 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 [4 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.rkt` will be helpful. Your implementation will not distinguish between cutoff failures and standard search failure (i.e., exhausting the search tree without discovering a goal). However, both cases should still produce the total number of nodes expanded.

Part D [8 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 [40 points]

In this problem, you will do some comparative analysis of your search routines by writing a single, integrated essay. The Part A/B/C structure below is simply to help you organize your efforts.

Note that `random-eight-puzzle-state` calls `random` to produce states; you should make your results repeatable by using the procedure (`random-seed seed`) to initialize the random number generator.

Part A

Generate a fairly easy eight-puzzle state and a small lights out problem (e.g., $s = 3$). 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 details of each problem and/or how they were generated.

Part B

Generate the hardest eight-puzzle problem and most difficult lights out problem (i.e., $s = 5$, at least 4 moves and depth limit of at least 5) you feel like waiting for solutions to under most (though not necessarily all) search algorithms. Run your search algorithms again, adding rows to the tables you created above. Be sure to specify the details of each problem and/or how they were generated.

Part C

Using the data you have generated, answer the following questions:

- How do the number of nodes expanded compare among the search algorithms? How does the relative efficiency of search algorithms vary with problem difficulty (if at all)? (Note this means reranking/evaluation, not the expected growth with respect to problem size.)
- How do the solution costs compare among the search algorithms? How does this comparison vary with problem difficulty (if at all)?

Draw conclusions about the relative efficiency and effectiveness of these search algorithms on these problems.

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, as well as 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. Be sure to include a title (but not your name). Your analysis submission's `references.txt` should acknowledge any parties responsible for the search source code used for analysis.

What to turn in

Programming Assignment

In addition to the `references.txt`, your joint submission should include the following

- Your completed `search.rkt` file
- A short test program `driver.rkt` that demonstrates all *your* search algorithm implementations applied to simple problems (i.e., that do not take long to run) with evidence for correctness.
- A transcript `output.txt` of your driver program's output

Analysis

Your individual submission should be only a PDF `analysis.pdf` for Problem 2. Files in any other format will receive a zero. (You will also need to include the standard `references.txt`.)

Copyright ©2013, 2015, 2018 Jerod Weinman. This work is licensed under a [Creative Commons Attribution-Noncommercial-Share Alike 4.0 International License](https://creativecommons.org/licenses/by-nc-sa/4.0/).

