Massachusetts Institute of Technology

6.034 Artificial Intelligence

Assignment # **4** Due: **10/6/98** 6.034 Item # **5**

Goals

- You will learn how to implement a variety of search routines.
- You will see that search is a quite general process that can be used to find a wide variety of things.

Background

Files mentioned in this problem set can be found in `/mit/6.034/ps4/` under Athena (remember to `setup 6.034`) or through our Web page.

Motivation

It seems that these days any new startup company with the even the most remote involvement with the Internet is a surefire way to get (temporarily) rich. We'd hate for you to be left out of such a promising (even if ephemeral) field.

So this week's homework will give you some experience with finding your way around the net.

In `netsearch.scm` you'll find the code that will get you started. You can load it in the usual way by doing `(load "/mit/6.034/ps4/load.scm")`.

The important top-level function in the code is `generalized-search`:

- `(generalized-search start finish finish? merge-paths-into-queue how-many successors)` – as the name suggests, this is a very general function capable of a wide variety of searches, if given the correct arguments.
  1. `start` and `finish` are the starting and ending points of the search.
  2. `finish?` is a function that determines whether we have reached the goal. Making `finish?` a function allows us to be more general than the standard `eq?` check.
  3. `merge-paths-into-queue` is a function indicating how to merge the newly generated paths with the queue of paths to be seached. You control a good deal of what `generalized-search` does by determining how it should do this merging.
  4. `how-many` is either `all`, meaning find them all, or an integer indicating how many paths to find.
  5. Finally, `successors` is a function capable of generating the successors of a given node. Typically in this problem set it is bound to `extend-path`.

There are three main data abstractions:

1. A host is composed of a host name, its latitude and longitude (in simple integer coordinates [geographical accuracy not guaranteed]), and the names of one or more hosts to which this host to forward packets. Example:
   `(mit 159 189 mt-holyoke amherst bu hahvud wellesley)`
   This indicates that MIT has the coordinates shown and can forward packets to the 5 other hosts shown.

   The variable `*data*` is bound to a list of these host abstractions.
2. A search `path` is simply a list of host names.
3. The `queue` is a list of search paths currently being explored.

To get you started there is test data describing small network. You may wish to draw the graph of this data as an aid to debugging your code.

Problem 1

There are some annoying but important details in searching a graph (rather than a tree). To give you a chance to explore the code and to encounter one of these difficulties, try evaluting
`(step-by-step '(mit))`
This function extends a search path one step at a time under your control so you can see how things are going. Answer `y` a few times until you see the problem, then answer `n` to have it quit.

Now fix the problem by modifying `extend-path` and run the example above to show that it works.

Hand in your modified code.

At this point you may wish to uncomment the line of code in `extend-path` shown below, to give you tracing information as you work on the rest of the problem set. You can re-comment it anytime you don't want to see the intermediate results.

```
;; (format #t " %Extending the path  a." (reverse-path path))
```

Problem 2

The code contains a skeleton function for depth-first search. Complete it by defining `combine-paths`.

Debug your code on the test data with

```
(depth-first 'dartmouth 'bc)
```

When you are sure you have it working, evaluate `(loadrealdata)` in order to get the entire network loaded into `*data*`.

Hand in your modified code and an example showing the system doing

```
(depth-first 'mit 'cmu)
```

(In what you turn in, make sure the "Extending the path" comment is turned on, so both you and we can see how your program works.)

Problem 3

Now create a `breadth-first` function. As before, debug your code on the test data with

```
(depth-first 'dartmouth 'bc)
```

When you believe it is working, use it to evaluate

```
(depth-first 'mit 'cmu)
```

Hand in the code and the results of the above test.

(In what you turn in, make sure the "Extending the path" comment is turned on, so both you and we can see how your program works.)

Problem 4

Now implement a branch and bound search called (what else?) `branch-and-bound` that always pushes forward from the shortest current path. You should do the simplest branch and bound (i.e., no need to worry about using dynamic programming to avoid exploring redundant paths).

We have supplied `straight-line-distance` and `sort` for your use (`sort` is in the file `qsort.scm`). At this point you may wish to make a separate copy of the code in netsearch.scm and replace some of the data abstraction definitions with new ones.

Show the results of your code working on the task

```
(branch-and-bound 'dartmouth 'wellesley)
```

on the test data (which is enough in this case to illustrate everything we need for this example).

Hand in the code and the results of the above test.

(In what you turn in, make sure the "Extending the path" comment is turned on, so both you and we can see how your program works.)

Show by reference to the "Extending the path ..." comments that the code is doing branch-and-bound and point out places where it becomes evident that the code is not using the dynamic programming principle.

Problem 5

Now, building what you have done above, implement an A* search (called `a-star`) using as `straight-line-distance` as the estimate of distance to the goal. Show the result of the following example:

```
(a-star 'dartmouth 'wellesley)
```

Once again, using the test data is good enough. Hand in the code and the results of the above test. How much did using A* save in this example?