Lecture 4: Paths and Experiments

1 General problem

Specify a path search problem by:

- Set of *states*, S
- Set of *actions*, A
- Initial state $s_0 \in S$
- *Action function* from S to set of possible A
- *successor function*: from $S \times A$ into S
- *goal test*: from S to $\{T, F\}$
- *cost*: from $S \times A$ into $\mathcal{R}$; non-negative

Graph defined implicitly (usually in AI) vs explicitly (usually in graph theory)

Trying to find a path $s_0, a_0, \ldots, s_n$ such that:

- $s_{i+1} = \mathit{successor}(s_i, a_i)$
- $\mathit{goalTest}(s_n) = T$
- $\sum_{i=0}^{n-1} c(s_i, a_i)$ is minimized

1.1 Example problems

- 8 puzzle
- driving a car; need gas, food, etc
- using a robot with a 10-lb payload to move a lot of boxes

2 Unform cost search

Guaranteed to find a shortest path. Time is $O(b^d)$ where b is branching factor and d is the depth of the solution.

```
Search(successor, goalTest, actions, cost, s0):
    agenda = PriorityQueue()
    explored = Set()
    agenda.add(Node(s0, 0, None), 0)
    while not agenda.empty():
        node = agenda.extractMin()
        if goalTest(node.state):
            return node.path()
        explored.add(node.state)
        for a in actions(node.state):
            newState = successor(node.state, a)
            newNode = Node(newState, node.cost + cost(node.state, a), node)
            if newState not in explored:
                if newState in agenda with higher priority:
                    remove that node from the agenda
                agenda.add(newNode, newNode.cost)
    return None
```

3 Informed search

Use a heuristic to guide search. $h(s)$ is the estimated cost of the cheapest path from state s to some state satisfying the goal test.

All we have to do is change this line of code:

```
agenda.add(newNode, newNode.cost)
```

3.1 Greedy best-first search

Use h as the priority for inserting nodes into the agenda. So, we have

```
agenda.add(newNode, h(newState))
```

Drives quickly to a goal state. Doesn't guarantee shortest path.

3.2 A*

Use $f(n) = \mathit{pathCost}(n) + h(n.\mathit{state})$ as priority. So, we have

```
agenda.add(newNode, newNode.cost + h(newState))
```

Think of $f(n)$ as estimated cost of cheapest path to goal that goes through node n.

A* is optimal if:

- h is *admissible*: that is, $h(s) \leqslant$ true cheapest path cost to goal.
- h is *consistent*: that is, $h(s) \leqslant c(s, a, s') + h(s')$

Our algorithm above is correct iff h is consistent. Can modify to work with heuristics that are admissible but not consistent.

Consistent actually implies admissible.

Runtime is still $O(b^d)$, but the tighter the heurstic, the smaller the effective branching factor.

Get heuristics by solving a *relaxed* version of the problem. Need to be reasonably efficient to compute.

4 Designing an experiment

What kind of claim do we want to make?

1. I got the following great result on a particular problem with algorithm A!
2. Algorithm A is **better** than algorithm B in class X of problems.
3. The performance of algorithm A varies in some interesting way as a function of a parameter in the algorithm or the problem class.

4.1 Particular problem

Anything goes. Generally, this is only interesting if people really care about the problem (it's an important molecular bonding problem, or it solves a puzzle with a very large prize or it beats Kasparov).

4.2 Comparing two algorithms

Better: Decide what you mean by *better*. Could be: finds higher scoring solutions, runs more quickly, has lower variance in the running time or in the solution quality.

Algorithm: When actually comparing algorithms, they need to be *black boxes*. That is, they take a problem instance as input and they terminate, and either generate a solution, assert there is no solution (in the case of a 0-1 problem) or fail. If they are going to do multiple restarts, or adapt parameter values, they do that internally to the algorithm.

Problem class: Could be either

- A fixed set of problem instances, or
- A distribution over problems from which instances can be randomly and independently drawn.

Making claims: Let's consider 4 cases, which require different kinds of analysis. (You can figure out on your own how to deal with one deterministic and one randomized algorithm).

A. Deterministic algorithms, fixed problem class

B. Deterministic algorithm, problems drawn from distribution

C. Randomized algorithms, fixed problem class

D. Randomized algorithms, problems drawn from distribution

Deterministic algs, fixed problem class It's reasonably clear that you need to run each algorithm on each problem and get values of whatever measure(s) you care about (see the 'Better' topic above). The only interesting question is how you should compare them. For instance, what if one problem instance is intrinsically really easy (has a very short run time or small score) and another is really hard (has a run time or score that is orders of magnitude bigger than the other one)? It would not be a good idea to average the scores from algorithm A, average the scores from algorithm B, and then compare them.

Better is to *pair* the tests: for each problem instance, compute the difference in scores between the two algorithms, and report the average difference. This still might not be scaled well. Even better would be to do something like

$$\frac{1}{n}\sum_{i=1}^{n}\frac{a_i - b_i}{max(a_i, b_i)}$$

where the a_i and b_i are the scores of algorithms A and B on problem i. This is an average relative improvement.

Deterministic algs, problems drawn from distribution Now we have to do some statistics. Drawn N problems from the distribution. Run each algorithm on each problem. Do a *paired T-test* (if you think that the statistics might be normally distributed) or a *paired non-parametric test, like Wilcoxon signed rank* or some kind of a *Bayesian hypothesis test* on the difference between the algorithms. If you don't find a significant difference, you could try increasing N. Or you could decide that if it's that hard to tell the difference, then we shouldn't care anyway.

Randomized algs, fixed problem class We're probably trying to answer the question: if I were to get a new problem, and run one algorithm on it, *once*, is A likely to better then B? What I would do is:

- Run A and B each M times on *each problem* in the problem class.
- Do a statistical test on *each problem* (this is not a paired-sample test), deciding whether you can conclude that A is better, that B is better, or that there is no significant difference.
- Report the percentage of problems on which A performs significantly better and the percentage of problems on which B performs significantly better.

Randomized algs, problems drawn from distribution This is a classic problem in experiment design. The easy approach is to basically pretend we're in case **B**, and ignore the fact that our algorithm is randomized. We can just draw N problems, run both algorithms on each one, and do a paired test. We might need to use a bigger N to compensate for the variation due to the random performance of the algorithm.

Alternatively, run each algorithm M times on each of the N problems. Now, use a more sophisticated analysis of variance (ANOVA) to decide which algorithm is better. Depending on the

relative variances in the process, you might be able to design an experiment by picking a ratio of N to M that gives the most statistical power for a total number of runs of the algorithms.

Choosing parameters for your algorithm

If the algorithm you want to make a claim about runs with a fixed parameter value, such as a learning rate or an annealing schedule (and doesn't try to select it automatically), then you have to be careful about how you select the parameter. In particular, it is *not okay* to use the same set of problems to evaluate different choices of the parameter that you will use later to evaluate the performance of the algorithm. You are very likely to positively bias your assessment of how well your algorithm works on problems in general; when somebody else buys it and applies it to their problem, they won't have the benefit of hand-tuned parameter values.

4.3 Performance curves

This is basically a variation on comparing two algorithms: except you're comparing a whole family of algorithms, by varying a parameter. Sometimes it's tempting (and sufficient) to plot curves and show error bars, and not bother with the significance testing. That's okay, but if you're going to do that, and there is randomization in your algorithm or choice of problems, then *show confidence intervals not standard deviations*. The size of the confidence interval on the mean depends on the number of samples you've used, and can give useful information visually. The standard deviation can be harder to interpret. And, in any case, state quite clearly what you're plotting and how you derived the numbers.