Beam Search and Local Search

In this lecture, we look at a number of other informed search strategies. These strategies use heuristics. But, in an effort to increase space and time efficiency, they sacrifice completeness and optimality. Nevertheless, they have proved very useful for certain kinds of problems, and we look at these problems.

1 Beam Search

One of the exercises for a previous lecture described *beam search*. Beam search is a form of heuristic search using $f(n) = g(n) + h(n)$. However, it is parameterised by a positive integer k (much as depth-bounded search is parameterised by a positive integer l). Having computed the successors of a node, it only places onto the agenda the best k of those children. (The k with the lowest $f(n)$ values.)

As you will have demonstrated in your answer to the exercise, this strategy is not complete nor is it optimal. But, it is efficient in both time and space.

Why am I mentioning it again here? We are about to look at local search, and then we look at something called local beam search, which combines ideas from beam search and local search.

2 Local Search

2.1 The Basic Algorithm

In *local search*, we take beam search to its extreme. We set k (the maximum agenda size) to 1.

Of course, this means we don't really need an agenda. All we need to keep track of is the current state. When we expand the current state, we pick one of the successors and make it the new current state. We discard all other successor states. They are not placed on an agenda. We have no intention of visiting these unexplored states later.

In effect, one path is pursued relentlessly, to the exclusion of all other paths.

Obviously, the lack of systematicity means that local search is not complete nor optimal in general. Its success in practice depends crucially on the function used to pick the most promising successor.

Here's the algorithm for a very basic form of local search. It assumes that we are, as before, seeking a path of actions that leads from an initial state to a goal state:

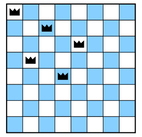
```
currentState = initial state
while currentState does not satisfy goal test and time limit
   is not exceeded
{   successors = the successors of the currentState
    currentState = the member of successors with highest
       promise (i.e. its heuristic value is lowest)
}
if currentState satisfies goal test
{   return path of actions that led to the currentState
}
else
{   return failure
}
```

2.2 Who Cares About Paths?

In the search algorithms that we have looked at so far, what was important was finding a *solution path*, i.e. a sequence of actions that transforms the initial state to the goal state. However, sometimes we are not interested in this path of actions. Sometimes we are interested only in finding the goal state itself. Local search is often used for such problems.

Let's start with a toy example, as usual. In the n-Queens Problem, the task is to place n Queens ($n > 0$) on an n by n chessboard in such a way that no Queen can attack any other. (Queens can attack pieces in the same row, the same column or the same diagonal.)

Here's the 8-Queens Problem after 5 Queens have been placed in the first 5 rows. Unfortunately, it is now impossible to add a Queen to the 6th row. (Finding a legal solution is left as a challenge to the reader.)



In this puzzle, all that matters is finding the goal configuration. The sequence of actions that led from an empty board to a goal board is irrelevant.

Some real-world problems can have this character too. For example, consider the problem of factory floor layout. The initial state is an empty factory floor and a list of machines that are to be installed on this factory floor. The goal is to position each machine on the factory floor to both satisfy safety regulations and minimise work-flow costs. Assume that the only output we desire is a diagram showing where the machines should be located. The order in which the machines were added to this diagram (the solution path) is not of interest.

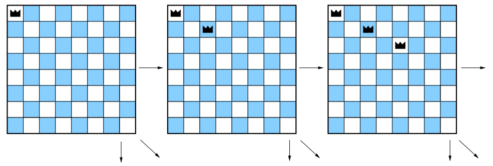
Other examples may include integrated-circuit design, job-shop scheduling, timetabling, and vehicle routing. Contrast these with the 8-tiles puzzle, the water jugs puzzle, vehicle assembly and cargo loading, in which we would want to be told what actions to perform and the order in which to perform them.

Local search is often used for these kinds of problems. It doesn't matter too much if the search meanders around a bit because the path of actions itself (and hence its length/cost) are not relevant. All that matters is eventually getting to an optimal (or near-optimal) state.

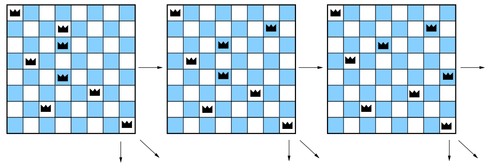
2.3 Why Not Use Complete States?

In problems where paths don't matter, we often get another important choice about how to represent the states. We'll illustrate this choice using the n-Queens Problem.

One approach (implicit in the description above) is that we start with the empty board. Each time we generate the successor, we add one more Queen to the board. We keep doing this until all 8 Queens have been added to the board.



But an alternative is to start with a board in which all 8 Queens have been randomly assigned to squares on the board. (We can help the search along a bit by assigning them to distinct rows.) Then a successor is generated by moving a single Queen to another square in the same row. This is sometimes called a *complete-state formulation*.



The same choices face us in our factory layout example. We could start with an empty factory floor and add machines one by one. Or we could start with a factory floor in which all machines have been given random locations and then shift them about.

Local search is often used with complete-state formulations.

3 Optimisation Problems, Objective Functions and Hill-Climbing

Local search algorithms are often used to solve optimisation problems. In these problems, not only might the solution path be irrelevant, there might not even be a goal state or goal test. Instead, there is an *objective function* and the aim is to find a state with a very high value for this function, preferably one of the states with the *highest* value for this function. (Note the difference: previously we wanted to minimise the cost of the path; here we want to maximise the value of the state.)

In fact, our factory layout example was already of this form. The task was to lay out the machines so as to minimise work-flow costs. Hence, this is an optimisation problem.

The simplest local search algorithm for such problems is the *hill-climbing* algorithm:

```
currentState = initial state
while true
{   successors = the successors of the currentState
    chosen =  the member of successors with highest
       promise (i.e. its value for the objective function
       is highest)
    if value(chosen) <= value(currentState)
    {   return currentState
    }
    currentState = chosen
}
```

You can see that this keeps searching until none of the successors improves the objective function.

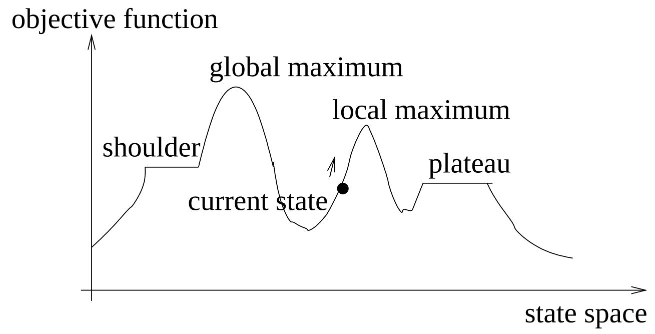
(Here we are looking to maximise an objective function that represents the value of a state. Authors sometimes describe the algorithm the other way around, where the goal is to *minimise* the cost of the state. To change the algorithm, simply switch from $<=$ to $>=$. Then hill-*climbing* is a misnomer and *gradient descent*, discussed in the neural nets lectures, is a more appropriate name.)

The basic hill-climbing algorithm computes all the successors and chooses the best of them. There are many variants of this. Here are two:

Stochastic hill-climbing: Compute all successors. Take those that improve the value. Choose one of these at random.

First-choice hill-climbing: Compute successors randomly one-by-one. Choose the first that improves the value. This is a useful approach when a state can have many successors because, except in the worst-cases, it avoids generating all of them.

Hill-climbing is a greedy strategy. The upside is it may make rapid progress towards the best state. The downside is that it can halt at a *local maximum*. A local maximum is a state that is better than all its successors but worse than the global maximum.



One workaround is to allow *sideways moves*, i.e. to allow the algorithm to choose a successor that has the same value as the current state. (In the algorithm, $<=$ becomes $<$.) This can help a search get off a shoulder (why?), but the search becomes infinite on a plateau (why?). So a limit must be placed on the number of consecutive sideways moves.

The most sophisticated versions of this idea go under the name *tabu search*. Tabu search is hill-climbing search, often with complete-state formulations, in which sideways moves are allowed. However, a fixed size memory is maintained of the most recently visited states. This is called the *tabu list*. The current state may not be replaced by any state that is currently on the tabu list.

Allowing sideways moves still doesn't solve the problem of other kinds of local maxima.

One possible solution is *random re-start hill-climbing*. In random re-start hill-climbing, a series of hill-climbing searches are executed from randomly-chosen initial states. In the limit (if enough random re-starts are done), this should find an optimal solution.

Another possible solution is to even allow *downwards moves*, i.e. if none of the successors is better than the current state, but the time limit is not yet reached, allow the algorithm to choose one of these successors that worsens the value. If you do this, you should store the current state before making the downwards move. Next time you reach a maximum, you can see whether the new maximum is better or worse than the one you stored earlier, and work with the better of the two. (This is not the same as having an agenda. Here we are remembering just one state: the best (local) maximum seen so far.)

Simulated annealing search is a variant of hill-climbing that allows downwards moves, and which has met with some success in practice. In simulated annealing, we generate the successors of the current state. But we do not simply pick the best of these, or the best of those that improve the function. Instead, we pick any one of them at random. If this randomly-chosen successor has a higher value than the current state, it becomes the new current state. If the randomly-chosen successor is of equal or lower value, it will become the new current state with probability less than 1. The more severe the downwards move, the lower the probability it will be accepted. And, as the search progresses, all probabilities decrease. So 'bad' moves are more likely to be chosen early on in the search, and they become less likely as the search progresses. By this means, we encourage 'wider' exploration early on, but we prohibit it later on when, we hope, the current state is near the global maximum.

(Why is this called simulated annealing? Annealing is the process of hardening metals by heating them to a high temperature and then gradually cooling them so that they coalesce into a low energy crystalline state.)

4 Local Beam Search

Local beam search is a cross between beam search and local search. It is, again, most used for problems where there is an objective function to be maximised.

In local beam search, we keep hold of k current states. We generate all the successors of each of the k states, and put them into a big pool. We evaluate each of them using the objective function. We throw away from the pool all but the top k. And we repeat.

Question. Some people think this is the same as doing k local searches in parallel. But it is not. These people have not read the description carefully enough. Why is it different?

One potential problem with local beam search is that the k states may soon lack diversity: the search focuses too early on a narrow part of the state space. One way of countering this is to change the way that the k are selected from the pool. Instead of selecting the top k, we would select at random, using probabilities that reflect the value of the objective function (i.e. the higher a state's value, the more likely it is to be selected).

Anyone with their brains switched on will realise that Genetic Algorithms carry out local beam searches. The initial k states/individuals are generated randomly. Then, based on the fitness/objective function, the next k states are generated using copying, crossover and mutation. Various methods are used to maintain diversity especially of the earliest generations of the population (e.g. using mutation, using tournament selection, etc.)