Local Search (Ch. 4-4.1)
Announcements

First homework due today at 11:55 pm

First writing assignment posted (soon) (due in a week)
Review: tree vs. graph search

Tree search = sequence of actions matter

Grid search = end state matters
Local search

Before we tried to find a path from the start state to a goal state

Now we will look at algorithms that do not care about the path, just try to find the goal

Some problems, may not have a clear “best” goal, yet we have some way of evaluating the state (how “good” is a state)
Local search

Today we will talk about 4 (more) algorithms:

1. Hill climbing
2. Simulated annealing
3. Beam search
4. Genetic algorithms

All of these will only consider neighbors while looking for a goal
Local search

These algorithms will also only consider the actions from their current state (neighbors).

They all have a greedy component, along with typically a random component.

In general, they can efficiently find a good solution, but have difficulty finding the best.
Hill climbing

Remember greedy best-first search?
1. Pick child with best heuristic
2. Repeat 1...

Hill climbing is only a slight variation:
1. Pick best between: yourself and child
2. Repeat 1...

This avoids the looping issue...
Hill climbing

This actually works surprisingly well, if getting “close” to the goal is sufficient (and actions are not too restrictive)

Newton's method:
Hill climbing
Hill climbing

For the 8-puzzles we had 2 (consistent) heuristics:

- **h1** - number of mismatched pieces
- **h2** - $\sum$ Manhattan distance from number's current to goal position

Let's try hill climbing this problem!
Hill climbing

Can get stuck in:
- Local maximum
- Plateau/shoulder

Local maximum will have a range of attraction around it

Can get an infinite loop in a plateau if not careful (step count)
Hill climbing

To avoid these pitfalls, most local searches incorporate some form of randomness

Hill search variants:
Stochastic hill climbing - choose random move and take that if better than current

Random-restart hill search - run hill search until maximum found (or looping), then start at another random spot and repeat
Simulated annealing

The idea behind simulated annealing is we act more random at the start (to “explore”), then take greedy choices later.

https://www.youtube.com/watch?v=qfD3cmQbn28

An analogy might be a hard boiled egg:
1. To crack the shell you hit rather hard (not too hard!)
2. You then hit lightly to create a cracked area around first
3. Carefully peal the rest
Simulated annealing

The process is:
1. Pick random action and evaluation result
2. If result better than current, take it
3. If result worse accept probabilistically
4. Decrease acceptance chance in step 3
5. Repeat...

(see: SAacceptance.cpp)

Specifically, we track some “temperature” $T$:
3. Accept with probability: $e^{\frac{\text{result- current}}{T}}$
4. Decrease $T$ (linear? hard to find best...)
Simulated annealing

Let's try SA on 8-puzzle:
Simulated annealing

Let's try SA on 8-puzzle:

This example did not work well, but probably due to the temperature handling.

We want the temperature to be fairly high at the start (to move around the graph).

The hard part is slowly decreasing it over time.
Simulated annealing

SA does work well on the traveling salesperson problem

(see: tsp.zip)
Local beam search

Beam search is similar to hill climbing, except we track multiple states simultaneously.

Initialize: start with K random nodes
1. Find all children of the K nodes
2. Select best K children from whole pool
3. Repeat...

Unlike previous approaches, this uses more memory to better search “hopeful” options.
Local beam search

However, the basic version of beam search can get stuck in local maximum as well.

To help avoid this, stochastic beam search picks children with probability relative to their values.

This is different from hill climbing with K restarts as better options get more consideration than worse ones.
Local beam search
Genetic algorithms are based on how life has evolved over time.

They (in general) have 3 (or 5) parts:
1. Select/generate children
   1a. Select 2 random parents
   1b. Mutate/crossover
2. Test fitness of children to see if they survive
3. Repeat until convergence
Genetic algorithms

Selection/survival:
Typically children have a probabilistic survival rate (randomness ensures genetic diversity)

Crossover:
Split the parent's information into two parts, then take part 1 from parent A and 2 from B

Mutation:
Change a random part to a random value
Genetic algorithms are very good at optimizing the fitness evaluation function. While there are a fair amount of parameters to choose from, they are not very sensitive. The downside is that it typically takes a while to converge to the optimal solution (i.e. many generations have to be created).