CS 486 — Lecture 5: Local Search

1 Intro to Local Search

- We've discussed a few search strategies — but they have some problems.
- So far, search algorithms explore the space systematically and keep track of one or more paths.
- What happens if the search space is too big?
- What if we don't care about the path to the goal — just the goal node itself? For example, nobody cares about the *order* of the solution to the 4Q problem. Just the result.
- We remove two things we care about. One is that we no longer systematically search. Furthermore, we no longer remember the path to the goal node — just the current state we are visiting.
- This means we need less memory and only explore a portion of the search space. It can find solutions quickly on average and works on CSPs and general and general optimization problems.
- However, there is no guarantee that a solution will be found if one exists, nor can we prove that no solution exists with just a local search.
- A LSP contains of:
  - A state — a complete assignment to all variables.
  - A neighbour relation — which states to explore next?
  - A cost function — how good is each state?
- Let us show the 4Q problem as a LSP:
  - Variables and domains are the same as before.
  - The initial state is 4 queens on the board in random row positions.
  - The goal state is 4 queens on the board that are not attacking each other.
  - Note, we have no constraints — this is as there are many possible positions that we check that will not meet the constraints.
  - The neighbour relation can either be moving a single queen to a different row or swap the row positions of 2 queens. Discussed later.
  - The cost function is that we want to minimize the number of pairs of queens attacking each other, directly or indirectly (so if you had 3 queens on the diagonal, then the first and third queen are technically indirectly attacking each other).

2 Local Search Algorithms

2.1 Greedy Descent

- Start with a random state.
- Move to a neighbour with the lowest cost if it is better than the current state.
- Stop when no neighbour has a lower cost than the current state. It will be one of the best states in the neighbourhood.
- Greedy as it only sees and considers the current node (a la greedy algorithm).

- This performs well and will often progress rapidly towards a solution.
- However, greedy descent can get stuck at local optimums (think of the gradient descent pits).
- GD can get stuck in actual local minimums or flat local minimums — for example, a long flat section that isn't a pit (shoulder), or an actual flat pit. The latter is much harder to escape, the former can get escaped if we travel long enough.
- Ways we can improve is allowing moving to states with the same cost, and keeping some track of recent neighbours to avoid getting stuck in loops.
- One way to deal with getting stuck is random restarts (restart randomly in a new location) and random walks (don't always move to a state with a lower cost).

2.2 Simulated Annealing

- GD focuses on optimization/exploitation, whereas randomly moving allows us to explore the search space.
- Can we combine these two properties into one a algorithm?
- Annealing is a process where we cool down molten metals to make them stronger.
- As per CS tradition, we stole this term to describe what we want to do.
- At each step, choose a random neighbour. If it is an improvement, move to it. If it is not, then move to the neighbour probabilistically depending on both the "current temperature", T, and how much worse the neighbour is compared to the compared state.
- Define A as the current state and A' as the worst neighbour. Then let $\Delta C = cost(A') - cost(A)$. The current temperature is T.
- Then, our probability of moving to neighbour A' is $e^{-\frac{\Delta C}{T}}$.
- We can write the simulated annealing function as such:

```
current = initial_state
T = some large positive value
while T > 0:
    next = a random neighbour of current
    delta_c = cost(next) - cost(current)
    if delta_c < 0:
        current = next
    else:
        current = next with prob. p = e**(-delta_c / T)
    decrease T
return current
```

- In theory, we want to decrease T slowly to find the global optimum. If it is slow enough, we are guaranteed to find the global optimum with prob. approaching 1.
- An "annealing schedule" that's commonly used is called geometric cooling (ie: multiply T by 0.99 each step).

2.3 Genetic Algorithms

- Local search algorithms so far only remember one state — what if we remember multiple states?
- One idea is beam search.
  - Remember k states, and choose the k best states out of all neighbours.

  - If $k = 1$, then this is identical to a normal local search.
  - How is beam search different from k random restarts in parallel? The idea behind beam searching is that it is aware of other states — restarting randomly would not have this ability, and are independent!
  - But this communication adds a problem — now you have a good region in the search space, then we cluster our results around there, losing some diversity.

- We next look at stochastic beam search. Beam search chooses k states deterministically; stochastic BS chooses via probabilities.
  - The probability of choosing a neighbour is prop. to its fitness.
  - This maintains diversity in the population of states.
  - This also mimics natural selection.

- Lastly, we have the genetic algorithm. Similar to the stochastic BS, but it differs in that we pick two states to make a new child state, and there is a random probability for the child state to mutate.