CS 520 : Notes on Local Search

16:198:520

Search problems are generally formulated in terms of the following:

- A **state space** X
- An **action set** A that allows transitions between states in X (available actions may depend on the current state)
- A **start state** $s \in X$
- A **goal state** $g \in X$ or set of states $G \subset X$

Typically we are interested in finding an efficient path or sequence of actions through the state space from start to goal, that minimizes some notion of cost. *Global Search* algorithms like DFS, BFS, A^* systematically explore the entire space of paths through the state space to find the path of least cost. There are a couple of potential issues that might need facing, though:

- Exponential space costs
- Exponential time costs
- The specific path may not be needed or interesting itself, only the final state
- The goal may not be explicitly identified, but rather you 'know it when you see it'

As an example of the third, consider solving a Rubik's cube, where the only thing that really matters is that the cube be rendered solved - the precise sequence of moves (up to a point) to solve it is less interesting. (*An exception might be made here for speed solving.*) As an example of the fourth, consider chess, where there are many possible ways to be in checkmate, and it is difficult to know which one will occur - though it is easy to verify you are in checkmate when it happens. To try to get around one or all of these issues, it may be advisable to consider *local search algorithms.*

Local Search Algorithms

The general framework for local search is the same as before, except that instead of attempting to find the global optimal path towards the goal, we simply ask: *given the current state, are any of the 'nearby' states better?* To answer this, we need to refine what we mean by 'better', and 'nearby'.

- **nearby:** Given a state x, the *neighborhood of* x $N(x)$ is the set of states that are immediately reachable from x. Typically, this means states within one step of x or within a given distance of x.
- **worth:** We equip our state space with an *objective function* F such that if $F(x) > F(x')$, we treat x as preferable to x'. Goal states are taken to be the maxima (or minima) of the objective function.

One of the difficulties in implementing these algorithms is assessing what the objective function should be, i.e., how to assess if one state is better or closer to the goal than another. Examples include:

- The 8-Queens Problem: The objective would be to minimize the number of queens that can attack each other.

- The 8-Tile Problem: The objective would be to minimize the number of tiles in the wrong place.
- The Traveling Salesman Problem: The objective would be to minimize the total distance along any circuit through the cities.

Local search algorithms proceed by taking the current state x, looking at the local or nearby states x', and moving from the current state to a neighboring state if it is determined to be worth moving. The goal of these algorithms is to proceed, iteratively, along states of increasing value, to reach a state x^* that maximizes (or minimizes) the objective function F. Because the decisions are made on a local, state by state basis, the memory footprint of these algorithms is typically quite small, $O(1)$. The cost of this efficiency is that because these algorithms remember little more than where they currently are, the only assurance is that any state they return will be *locally* optimal, i.e., $F(x) > F(x')$ for all x' near x. In order to guarantee *global* optimality, the algorithm would have to explore the entire state space, and remember where it had explored to guarantee it traverses the entire state space. Since it is difficult to guarantee discovery of the true optimal state, time complexity guarantees for local search algorithms can be difficult to come by.

While this local optimality condition is a major shortcoming of these algorithms, it is worth considering if it is in fact a major drawback. Frequently, while the 'best' state or solution may be difficult to find, 'good enough' solutions are readily accessible. A workable solution generated in 30 seconds may be more valuable than the optimal solution generated in 30 days (*note: much will depend on the specific problem and the relative value of the best solution*). Additionally, if there is any kind of uncertainty, or the environment is changing or evolving over time, what is optimal today may not be optimal tomorrow, so that the 'value' of any optimal solution may decay quite quickly over time, while good enough solutions may be more stable. If optimality is vital, it may be worth devoting the additional resources to a complete global search. Local searches are best utilized when good enough is sufficient.

Random Restarts: It is worth observing here that all hope is not lost with regards to discovering a global optimum. In particular, local searches typically generate a sequence of states $x_0, x_1, x_2, \ldots$ of increasing value of the objective function. Because each state depends only on the previous one, much depends on the initial state x_0 that starts the whole process. If at any point the algorithm converges to a local optimum, or appears to be lost or meandering, the algorithm can execute a *random restart*, choosing a new state x_0, and proceeding from this new starting point. If the randomly chosen state is sampled over the entire space, random restarts can guarantee that if the algorithm is run long enough, it will eventually discover the true optimal state with probability 1. This requires no more additional memory cost than to store the best state seen so far. But much depends on the resources you are willing to devote to continuing the algorithm as long as is necessary.

The following sections outline some specific local search algorithms, each exploring the idea of how to proceed in such a way as to steadily improve the current state.

Hill Climbing

Hill climbing is in many ways a purely greedy approach to local search - whatever the current state is, take a neighboring state and if it is better, move there. This is frequently compared to trying to climb a mountain by always moving to a higher point within one step of where you currently are. Pseudocode for such an algorithm might look like:

```
x <- initial state
until termination:
    x' <- select a neighbor in N(x)
    if F(x') > F(x)
```

```
        x <- x'
```

This could be made even more strict, where in every step a search is made over $N(x)$ to find the *best* state x' to move to. In this case, the termination condition becomes more clear - terminate when $F(x) > F(x')$ for every neighbor $x' \in N(x)$.

Simulated Annealing

Hillclimbing's main weakness is that it can get trapped in local optima. Simulated annealing attempts to overcome this by introducing the possibility of mistakes - moves that actually take you to a worse state rather than a better one. This allows further exploration, and the opportunity to leave local optima traps. However, it is important to note that while these 'bad moves' can have value, ideally their frequency should diminish over time - if they did not, then the algorithm might constantly be wandering away from whatever best solution it had currently discovered.

In particular, simulated annealing introduces a probability of acceptance for bad moves, and has this probability diminish over time. Pseudocode for this might look like the following:

```
x <- initial state
t <- 1
until termination:
    temperature <- 1/t
    x' <- select a neighbor in N(x)
    if F(x') > F(x)
        x <- x'
    else
        if rand() <= P( x, x', temperature )
            x <- x'
```

In this case, 'worsening' moves are accepted only with a probability defined with the function P. Typically, these threshold probabilities need to go down as the temperature goes to 0, and need to go down the worse the move would be. A common choice is

$$P(x, x', T) = \exp\left(-k\frac{F(x) - F(x')}{T}\right) \tag{1}$$

In this case, as $T \to 0$, $P \to 0$, and the larger $F(x)$ is compared to $F(x')$, the smaller P is. The choice of constant k determines the rate of diminishing.

A Note on Origins: *The use of the term temperature here is deliberate - an analogy is drawn to physical systems that are cooling, and settling into a stable state or arrangement. The higher the temperature, the more the molecules are moving, and the easier it is for the system to slip out of local optima. This allows the whole system to settle into more stable states over all in the long run.*

Beam Search

Beam search is a natural extension of hillclimbing in the following way: suppose that instead of one hill climber, there were an entire team. However, it is not simply hill climbing run in parallel (and hill climbing is an extremely parallelizable algorithm). Rather, imagine a team of climbers that are additionally in contact with one another so that they can coordinate their searches. In particular, imagine that each of m-many climbers identifies the k-best neighbors to their current positions, generating a set of $m * k$ possible candidate positions. Of those $m * k$ candidates, the best m are then chosen as the next locations for the climbers. This proceeds, in a coordinated fashion, working the climbers up to better states with steady improvement. Note additionally, that if any climber determines that it is at a local optimum, that location can be noted, and that climber can join the search in some other region of the state space.

A word of caution: In these kinds of 'ensemble' search algorithms, where a population of potential solutions is maintained and improved, it is no cliche to say that there is strength in diversity. Beam search, in the ideal case, searches broadly over a wide swath of the potential search space. One risk however is that if one climber generates a lot of very good neighbors, all the climbers may converge to that one region of state space. In this way, the algorithm converges to something much closer to hill climbing, and loses much of the advantages of searching with a population.

Genetic Algorithms

Genetic algorithms in many ways represent a tremendous departure from the previous frameworks. Evolution in a biological context can be thought of as an algorithm for generating solutions to the problem of survival. Genetic organisms are evaluated by nature and the environment: those that are 'fit' are able to pass their genes on to generate offspring, while those that are not fit die before passing on their genes. Note further, the offspring are typically similar or 'close to' their parents, incorporating aspects of their parents while recombination and mutation generate new variations and properties that may not have been seen before. As the process continues, over time a population of organisms adapts to its environment by becoming more fit (over all).

We can adapt this idea to our problem of local search by imagining a population of states or solutions as a population of organisms. The 'fitness' of a given state or solution is determined by the relative value of the objective function for that state - the larger the value of F, compared to the other states in the population, the more likely that state is to be chosen to 'breed'. When two (or more) candidates are chosen to reproduce, they are combined to produce new states that combine properties of the original states. An easy example of this would be if any state could be specified by a 100-bit binary string, an 'offspring' of two bitstrings could be generated by taking the first 60-bits of one string and combining them with the last 40-bits of the other, to construct a new 100-bit string that shared properties with both parents, but was itself new. In general, the amount taken from each parent (the point of *crossover*) is chosen at random. After recombining the original population to produce a new population of children candidates, the best overall candidates will be kept, i.e., survive, to be recombined and generate the next generation.

Additionally, in keeping with biological evolution, we frequently introduce an element of *mutation*: when two parent candidates are combined to produce an offspring candidate, it may be modified at random (bits flipped, swapped, inserted, deleted, etc) to introduce new variation into the population. In general, this mechanism both promotes diversity over all, and introduces new opportunities to explore and develop new features. This is important for a variety of reasons: one, much like in beam search, the algorithm benefits over all from exploring a diverse range of possible states, rather than concentrating on a potential local optimum; two, mutation introduces new features for recombination that may not have been present in the original population; three, populations that converge to a single solution (genetically identical populations or monocultures) are historically catastrophically susceptible to small changes in the environment. Organisms optimized perfectly for living at a particular temperature may suffer from a massive die off if the temperature rises or falls a single degree. Maintaining diversity, and introducing new features, can make sure that the population is dynamic, and able to adapt to changing environments over time.

The typical outline for a genetic algorithm solution looks something like the following:

- generate a population of N solutions

- rank each solution according to its objective function value ('fitness')
- select solutions for recombination, with frequency determined by its fitness
- generate N children solutions recombining the selected parents
- with some probability, introduce mutations into each child
- of the $2N$ solutions (parents + children), select the best N
- repeat

Ideally, over time, the surviving population converges toward some optimum of the objective function.

Notes on Implementation: Perhaps moreso than for any other algorithm proposed so far, genetic algorithms hinge on how the underlying problem is represented. In order for genetic algorithms to be successful, the states or solutions x must be represented in such a way that recombinations preserve properties of the originals. For instance, imagine that the state space was taken to be the natural numbers, $1, 2, 3, 4, 5, \ldots$. In analogy with DNA we might represent each integer with its binary representation out to some number of bits, 5 as 000101, 17 as 010001, which could be recombined to [010][101] or 21 *which may or may not have any relation at all to* 5 *and* 17 *within the context of the problem.* However, taking the left half of a chessboard with 8 queens arranged on it, and the right half of a chessboard with 8 queens arranged on it, would potentially yield another chessboard with 8 queens on it, sharing properties of both parents while combining their elements in new ways.