Lecture 13: Local Search

February 4th, 2020
https://www.cs.ubc.ca/~jordon/teaching/cpsc322/2019w2/lectures/lecture13.pdf

Local Search

Motivation: Scale

Many CSPs (scheduling, DNA computing, etc.) are simply too big for systematic approaches. If we have 10^5 variables, each with a domain size of 10^4, then systematic search will have a complexity of $O(b^m) = (10^4)^{(10^5)}$ and arc consistency will have a complexity of $O(n^2d^3) = (10^{10}) \times (10^{12})$. But if the solutions are densely distributed, then we can apply local search.

General method for Local search

For this we need to remember that the solution to a CSP is a possible world, and not a path. Thus we follow the following general stages:

- Start from a possible world
- Generate some neighbors ("similar" possible worlds)
- Move from the current node to a neighbor, selected according to a particular strategy

Selecting Neighbors

Usually this is simple: some small incremental change to the variable assignment.

(a) assignments that differ in one variable's value, by (for instance) a value difference of +1

(b) assignments that differ (by any amount) in one variable's value

(c) assignments that differ in two variables' values, etc.

Iterative Best improvement

In order to determine the neighbor node to be selected, we use the method of **Iterative Best Improvement**, which chooses neighbors that optimized some evaluation function.

In any case, the best strategy would be to choose the neighbor which has the minimal number of constraint violations.

Example: N-queen as a local search problem

The formulation is as follows:

- We have One variable per column, and the domains are $\{1, ..., N\}$ corresponding to row where the queen in the i^{th} column sits.
- Constraints: no two queens in the same row, column or diagonal

The Neighbor relation is that the value of a single column differs.
The Scoring function is the number of possible attacks

The algorithm therefore goes:

```
For each column:
```

```
    assign randomly each queen to a row (a number between 1 and N)

While not solved:

    For each column & each number:
       Evaluate how many constraint violations changing the \
             assignment would yield
    end for

    Choose the column and number that leads to the fewest \
          violated constraints;
    change the assignment
end While
```

Why this problem?

Lots of research in the 90's on local search for CSP was generated by the observation that the run-time of local search on n-queens problems is essentially independent of problem size!

Given random initial state, can solve n-queens in almost constant time for arbitrary n with high probability (e.g., $n = 10{,}000{,}000$)

Constrained Optimization

So far we have assumed that we just want to find a possible world that satisfies all the constraints, but sometimes solutions may have different values/costs...

We want to find the optimal solution that

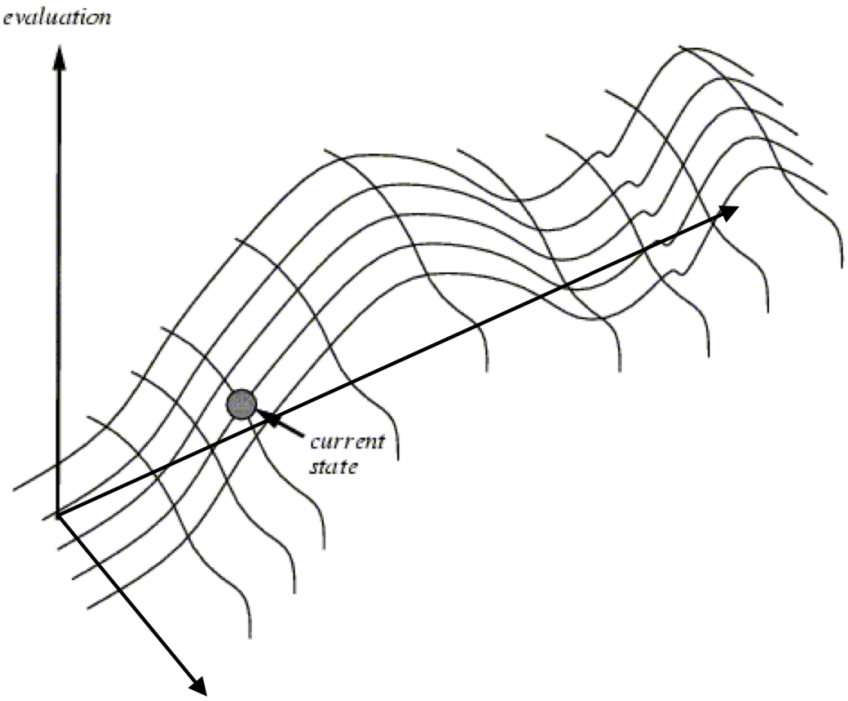
- **maximizes the value** *or*
- **minimizes the cost**

Two options:

1. **Hill Climbing** means selecting the neighbor which best improves a (value-based) scoring function.
2. **Greedy Descent** means selecting the neighbor which minimizes a (cost-based) scoring function.

Hill-climbing

***NOTE**: Everything that will be said for Hill Climbing is also true for Greedy Descent*



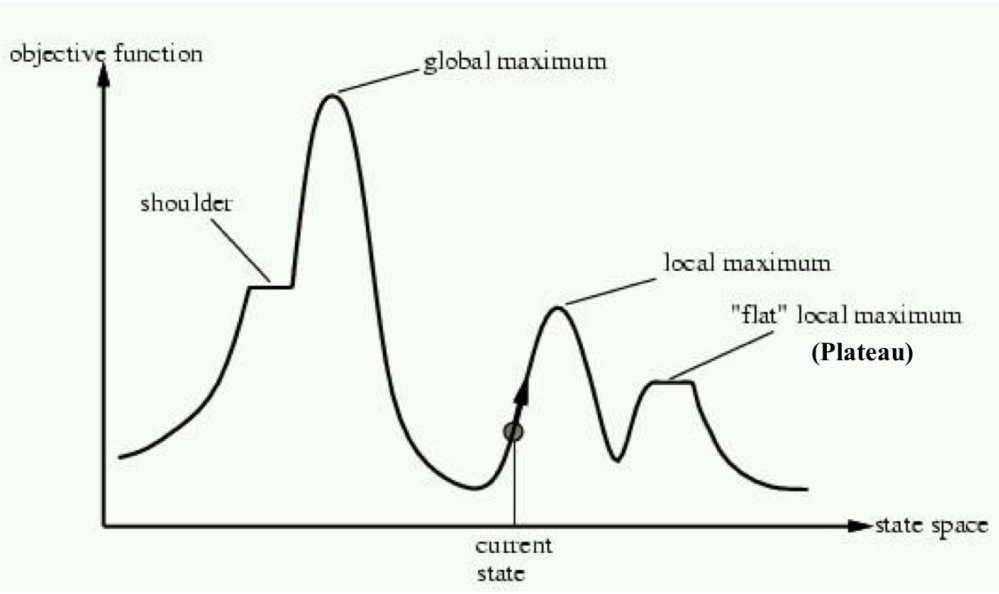
Example: A, B, C same domain $\{1, 2, 3\}$, $(A = B, A > 1, C \neq 3)$

- Value $= (C + A)$ so we want a solution that maximizes that
- The scoring function we'd like to maximize might be: $f(n) = (C + A)+$ number-of-satisfied-const

If we're doing Greedy Descent, then what we want to minimize is $cost + number - of - conflicts$

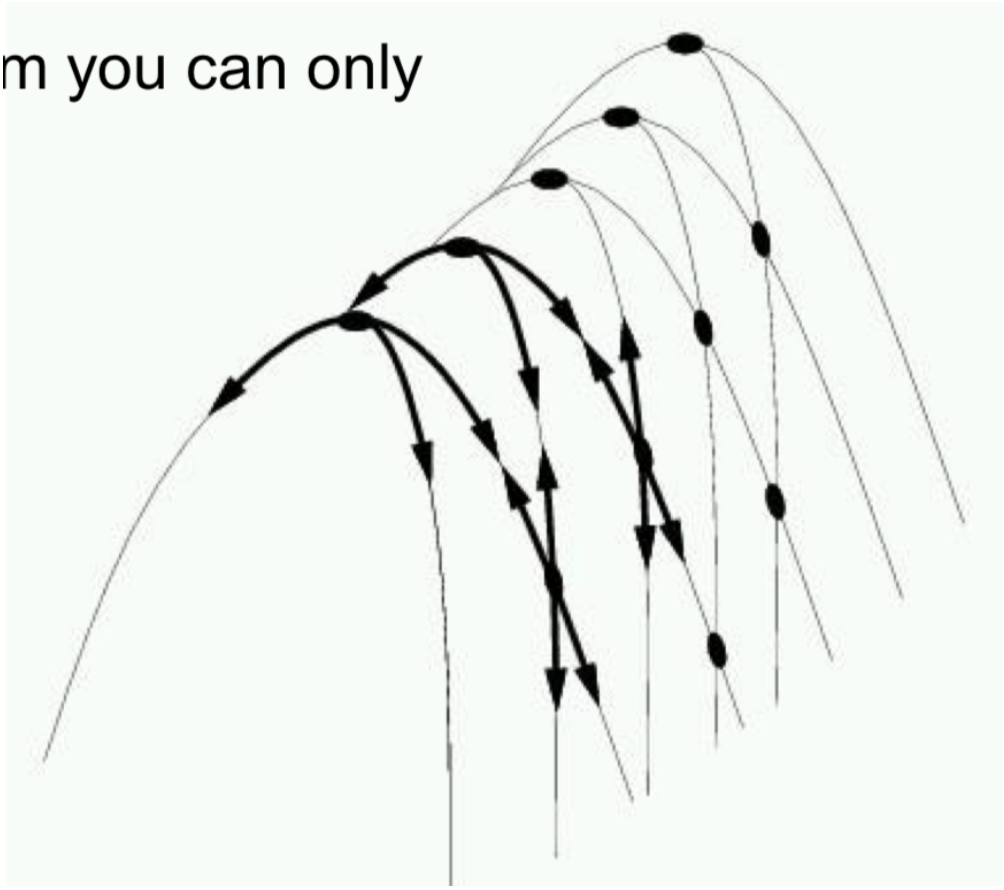
Problems with Greedy Descent/Hill Climbing

The main problems with those methods are local maxima, plateaus and shoulders. Those are shown in the figure below:

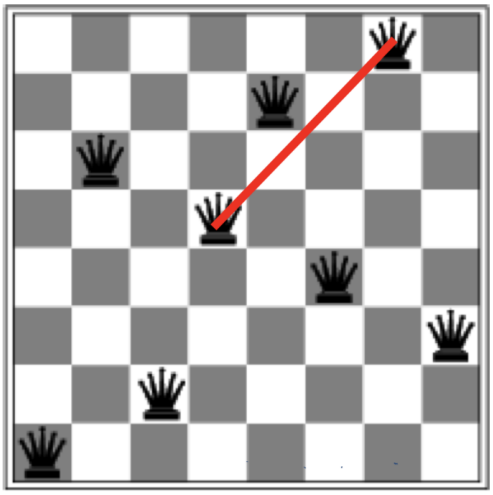


There are also other problems associated with these methods, notably in higher dimensions. Those involve:

- Ridges – sequence of local maxima not directly connected to each other
- From each local maximum you can only go downhill



An example of this in the n-queens problem is the following state:



In the state above we currently have a local minimum with $h = 1$, however each neighboring state will have $h > 1$

Lecture 14

February 4&6th, 2020
`https://www.cs.ubc.ca/~jordon/teaching/cpsc322/2019w2/lectures/lecture14.pdf`

Stochastic Local Search

Goal: We want our local search

- To be guided by the scoring function
- Not to get stuck in local maxima/minima, plateaus etc.

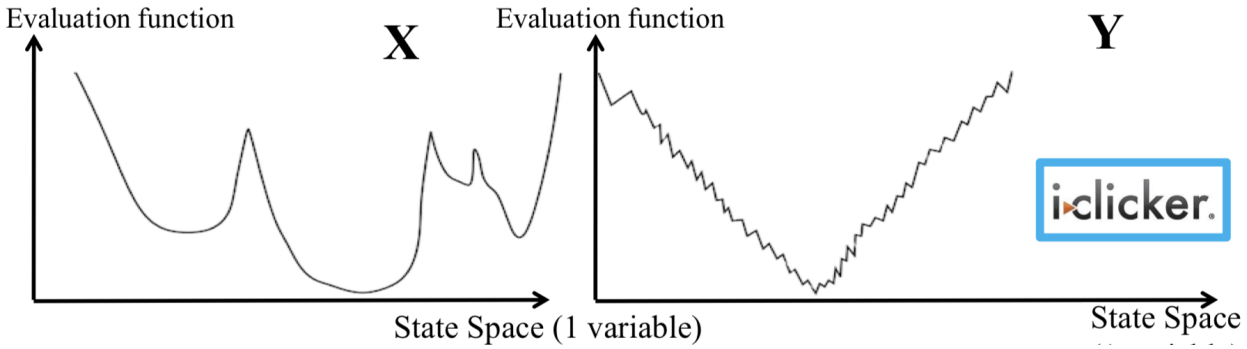
Solution: We can alternate

(a) Hill-climbing steps

(b) Random steps: move to a random neighbor.

(c) Random restart: reassign random values to all variables.

Using the following two graphs as examples, we have that random restarts would work best for example X and random steps would work best on example Y:



But these examples are simplified extreme cases for illustration. In practice, you don't know what your search space looks like, thus we usually integrate both kinds of randomization, and see that this works best.

Random Steps (Walk): one-step

Let's assume that neighbors are generated as assignments that differ in one variable's value. Then given n variables with domains of size d, a state will have $n(d-1)$ neighbors.

One strategy to add randomness to the selection of the variable-value pair. Thus we sometimes choose the pair:

1. According to the scoring function
2. A random one

For the n-queens problem, this would translate to:

1. Choose one of the best neighbors
2. Choose a random neighbor

Random Steps (Walk): two-step

Another strategy: select a variable first, then a value:

- Sometimes select variable:
  1. that participates in the largest number of conflicts.
  2. at random, any variable that participates in some conflict
  3. at random
- Sometimes choose value
  a) That minimizes number of conflicts
  b) at random

Stochastic Local search (SLS) advantages

Online setting

For instance we find SLS very useful in situation where the problem can change (particularly important in scheduling). For instance in the case of airline scheduling: thousands of flights and thousands of personnel assignments, and a storm can render the schedule infeasible...

The **goal** is to repair with minimum number of changes:

- This can be easily done with a local search starting form the current schedule
- Othertechniquesusually:
  - require more time
  - might find solution requiring many more changes

Anytime algorithms

When should the algorithm be stopped?

- When a solution is found (e.g. no constraint violations)
- Or when we are out of time: you have to act NOW
- Anytime algorithm:
  - maintain the node with best h found so far (the "incumbent")
  - given more time, can improve its incumbent

Stochastic Local search (SLS) limitations

Typically there is no guarantee that a solution will be found even if one exists. This is because SLS algorithms can sometimes **stagnate** (get caught in one region of the search space and never terminate). This is very hard to analyze theoretically.

Sometimes we are not able to show that no solution exists, because SLS simply won't terminate. This means that either:

1. You don't know whether the problem is infeasible
2. The algorithm has stagnated

Evaluating SLS algorithms

SLS algorithms are randomized, meaning that the time taken until they solve a problem is a random variable. It is entirely normal to have runtime variations of 2 orders of magnitude in repeated runs:

- E.g. 0.1 seconds in one run, 10 seconds in the next one
- On the same problem instance (only difference: random seed)
- Sometimes SLS algorithm doesn't even terminate at all: stagnation

If an SLS algorithm sometimes stagnates its mean runtime becomes **infinity**... Thus in practice, one often counts timeouts as some fixed large value X. In either case however, summary statistics, such as mean run time or median run time, don't tell the whole story (e.g.: would penalize an algorithm that often finds a solution quickly but sometime stagnates).

Consequently the **key idea** behind SLS is really combining greedily improving moves with randomization. This means that as well as improving steps we can allow a "small probability" of:

- Random steps: move to random neighbor
- Random restarts: reassign random values to all variables.

Then we always keep best solution found so far, and we stop when either a solution is found (in vanilla CSP, all constraints satisfied) or we run out of time (in which case we return best solution so far).

Lecture 15

February 6th, 2020
`https://www.cs.ubc.ca/~jordon/teaching/cpsc322/2019w2/lectures/lecture15.pdf`

SLS Variants: Tabu Lists

This variant aims at reducing problematic behaviors, in particular returning to recently visited nodes and cycling. In this case we maintain a **tabu** list with the last k visited nodes. We then keep away from worlds that at in the *tabu* list.

SLS Variants: Stimulated Annealing

Key Idea: Change the degree of randomness.

In *metallurgy* annealing is a process where metals are heated and then *slowly* cooled:

- *Analogy:* start with a high "temperature": a high tendency to take random steps
- Over time, cool down: more likely to follow the scoring function

Temperature reduces over time, according to an annealing schedule.

Stimulated Annealing: Algorithm

Here is how it works, for maximizing $h(n)$:

→ You are in a node n. Pick a variable at random, and a new value at random. You generate n'.

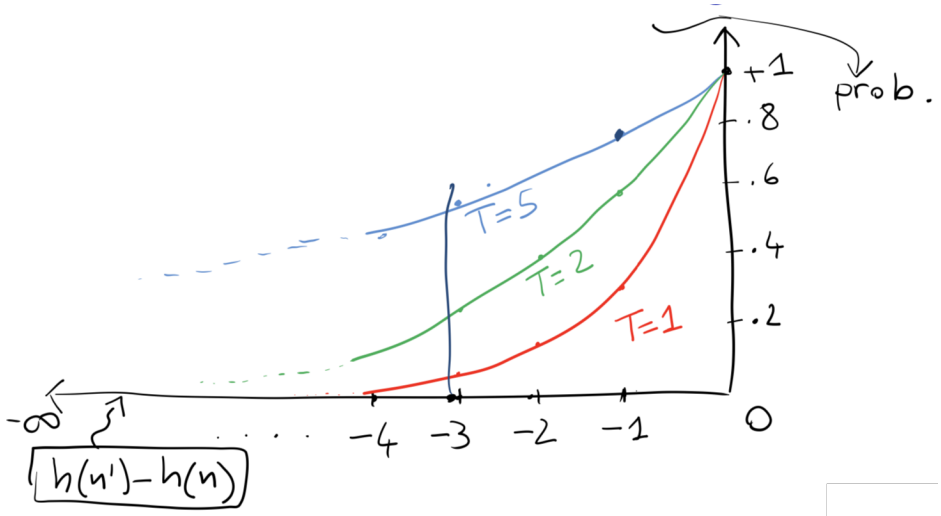
→ If it ***is*** an improvement (i.e.: $h(n') \geq h(n)$), then adopt it.

→ If it ***isn't*** an improvement, adopt it probabilistically, depending on the difference and a temperature parameter, T:

- We move to n' with probability: $e^{\frac{h(n')-h(n)}{T}}$

Notes:

- If it isn't an improvement, adopt it probabilistically depending on the difference and a temperature parameter, T.
- Therefore having a **higher** temperature T means that the algorithm is **more** likely to move to n' if it is "**worse**" than n:



Properties of simulated annealing search

One can prove:
If T ***decreases slowly enough***, then stimulated annealing search will **find a global optimum with probability approaching 1**.

Finding the ideal cooling schedule is unique to each class of problem:

- Want to move off of local extrema but not global extrema
- Want to minimize computation time while taking enough time to ensure we reach a good minimum

SLS Variants: Population Based SLS

Often we have more memory than the one required for current node (+ best so far + tabu list).

Key Idea: maintain a population of k individuals:

- At every stage, update your population.
- Whenever one individual is a solution, report it.

Simplest strategy: Parallel Search

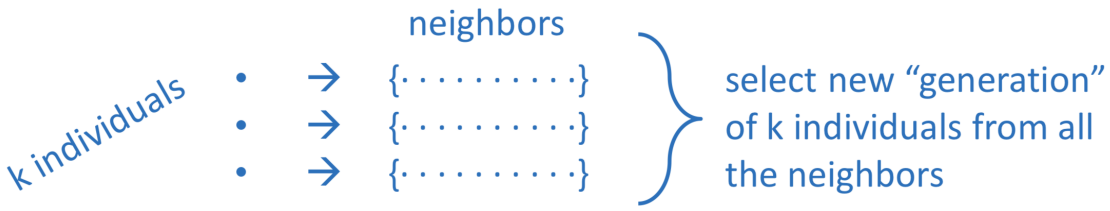
- All searches are independent
- No information shared
- Essentially running k random restarts in parallel rather than in sequence
- But we can take this idea...

Population Based SLS: Beam Search

This is a **non stochastic** method.

→ It works like parallel search, with k individuals, but you choose the k best out of all of the neighbors.

→ Useful information is passed among the k parallel search thread:



→ **Troublesome case:** If one individual generates several good neighbors, and the other $k-1$ individuals all generate bad successors, then the next generation will consist of very similar individuals...

Population Based SLS: Stochastic Beam Search

Non-stochastic beam search may suffer from lack of diversity among the k individual (just a more expensive hill climbing). **Stochastic version** alleviates this problem:

→ Selects the k individuals at random (from neighbors n_1 to n_m)

→ But probability of selection proportional to their value (according to scoring function $h(n)$)

Consequently, the advantages of using stochastic beam search is that it **maintains diversity** in the population. The **biological metaphor** would be (*asexual reproduction*):

- each individual generates "mutated" copies of itself (its neighbors)
- The scoring function value reflects the fitness of the individual
- the higher the fitness the more likely the individual will survive (i.e., the neighbor will be in the next generation)

Population Based SLS: Genetic Algorithms