Problem Solving using Search- (Single agent search)

Debaditya Barman

Local Search

- Local search methods work on complete state formulations.
- They keep only a small number of nodes in memory.
- Local search is useful for solving optimization problems, where:
  - Often it is easy to find a solution
  - But hard to find the best solution
- Algorithm goal:
  - find optimal configuration (e.g., TSP),
- Algorithms:
  - Hill climbing
  - Gradient descent
  - Simulated annealing

Local search…

- Consider the traveling salesperson problem in a fully connected graph.
- It is easy to get one solution to the problem because any permutation of the city is a solution but it is difficult to get the best solution because there are n! such permutations and we do not know one efficient way of finding the best permutation in polynomial time.
- So, in TSP the objective is to find the optimum configuration which is difficult to find.
- We do not know to find a good algorithm but if we just want to find a solution then it is easy.

Local search: Iterative improvement

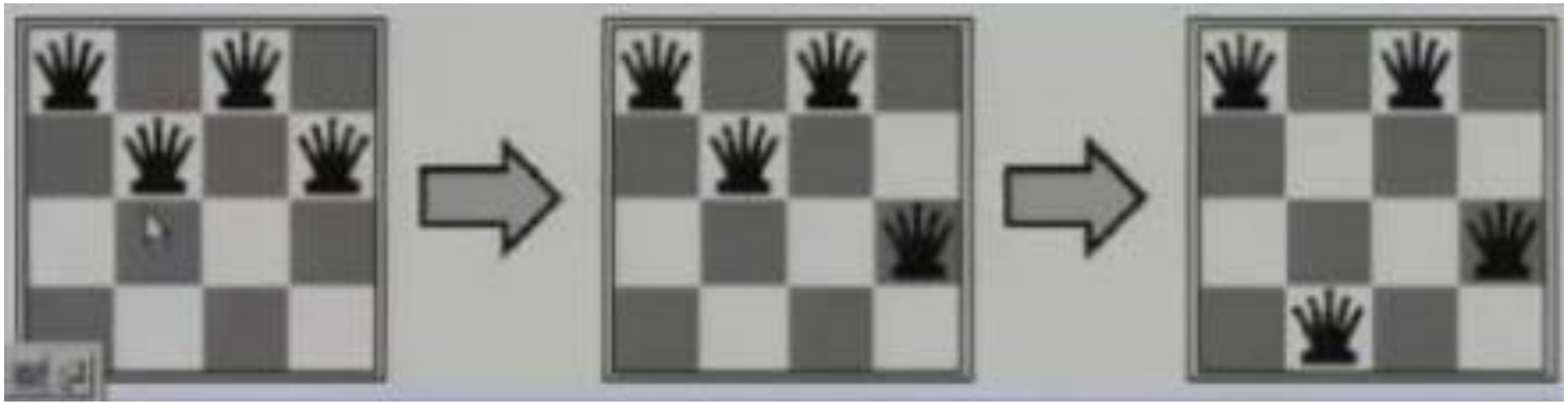
- Many local search algorithms use iterative improvement and we can try to get a solution and get improvements to that solution.
- Secondly, for these optimization problems like TSP we do not have to keep track of the path to the solution.
- When we have the solution we immediately know the path. In TSP we do not have to keep track of the path.
- Local search is ideal for such cases.
- In problems like 8 puzzle the final state does not give us any information about the path. The solution path has to be obtained. -Local search is not very good for such problems.
- But for problems like TSP where the path is not important local search is very useful.

Local search: Iterative improvement…

- Another example of a problem is the n queens problem where we have a *n* X *n* chess board and we have to put *n* queens on the board so that none of the queens are attacking each other.
- For this problem once we have the final configuration we know the solution.
- So, *n* queens problem is not concern how we get a solution. So we can define *n* queens problem as an optimization problem.
- The basic idea of the iterative methods is to start with the solution and improve it so that we can get a better solution.

Local search: Iterative improvement…

- We can reformulate our objectives as saying we want to have a board with 0 conflicts.
- Or we can say we want to minimize the number of conflicts – i.e. we want to minimize the number of pairs of queens which can attack each other.



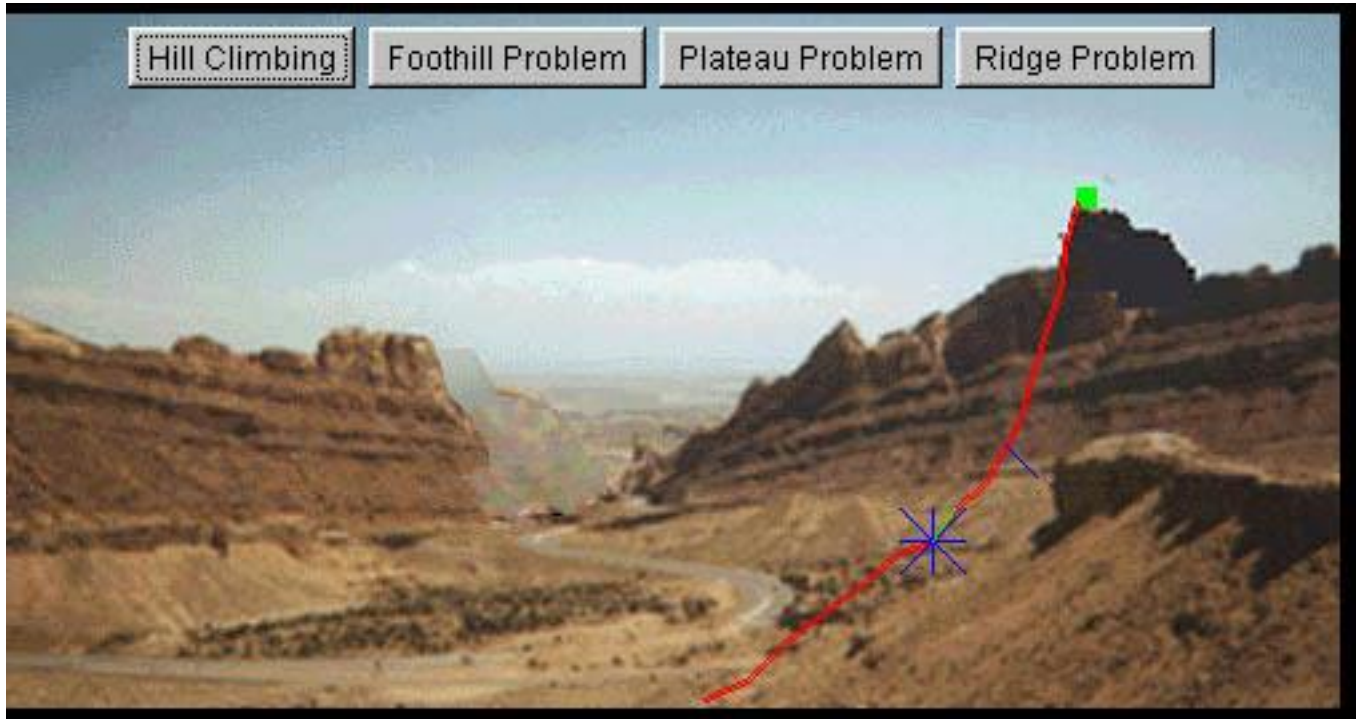
Hill climbing (or gradient ascent/descent)

- *Iteratively maximize "value" of current state, by replacing it by successor state that has highest value, as long as possible.*
- *Note: minimizing a "value" function v(n) is equivalent to maximizing –v(n), thus both notions are used interchangeably.*
- Algorithm:
  1. determine successors of current state
  2. choose successor of maximum goodness (break ties randomly)
  3. if goodness of best successor is less than current state's goodness, stop
  4. otherwise make best successor the current state and go to step 1
- No search tree is maintained, only the current state.
- Like greedy search, but only states directly reachable from the current state are considered.

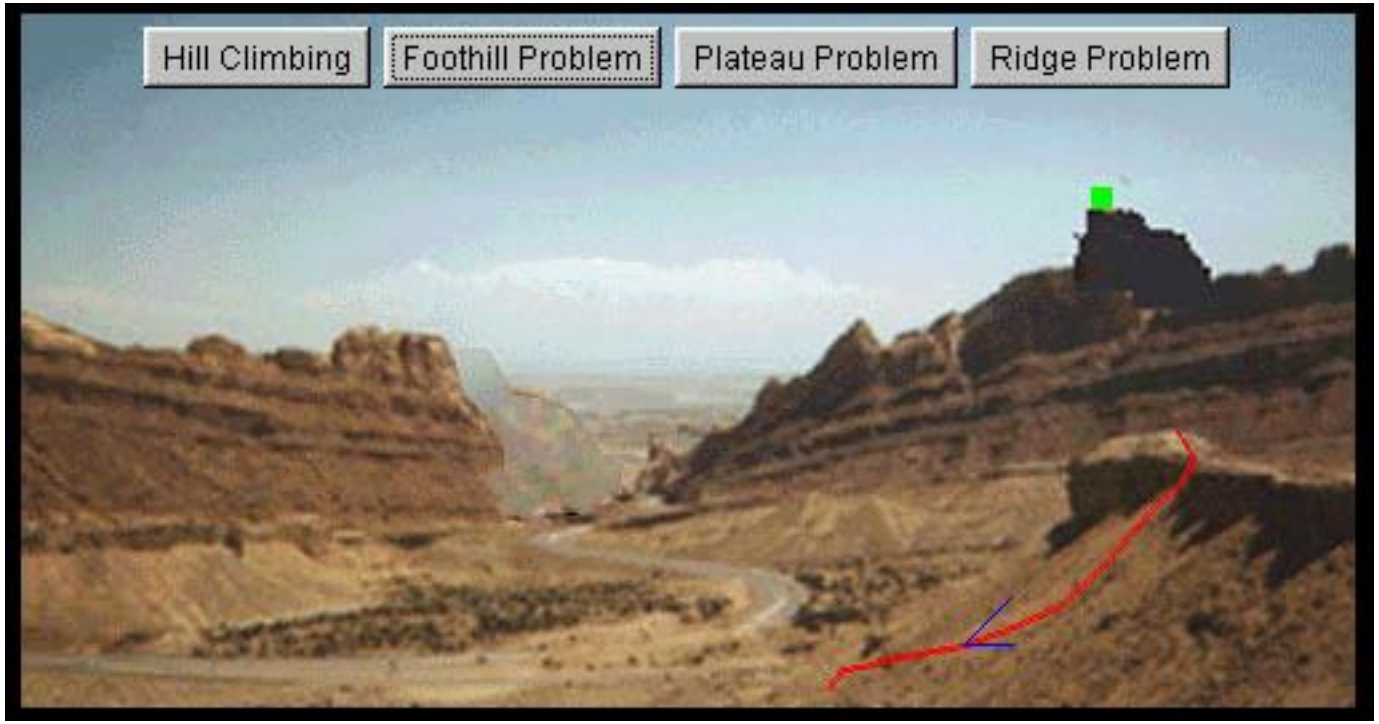
Hill climbing: Problem

- **Local maxima:**
  - Once the top of a hill is reached the algorithm will halt since every possible step leads down.
- **Plateaux**
  - If the landscape is flat, meaning many states have the same goodness, algorithm degenerates to a random walk.
- **Ridges**
  - If the landscape contains ridges, local improvements may follow a zigzag path up the ridge, slowing down the search.

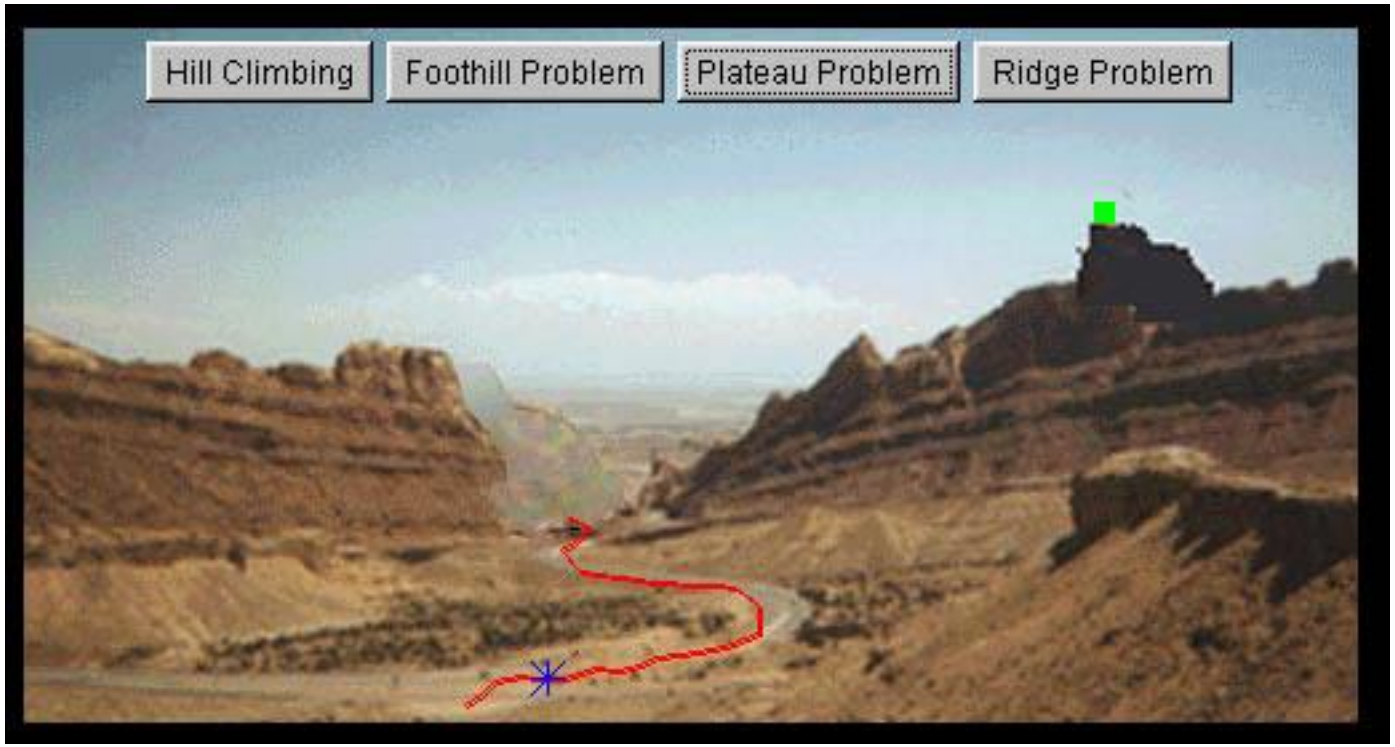
Hill Climbing



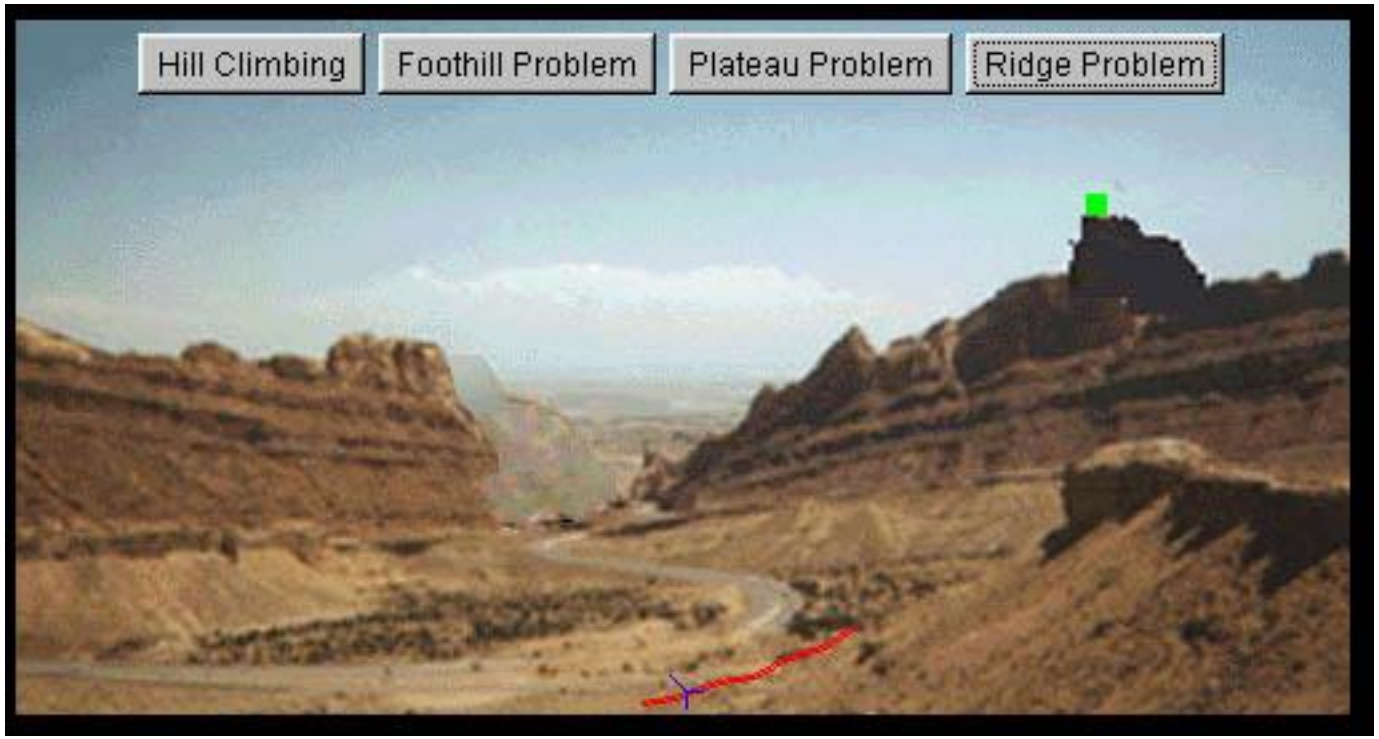
Local Maxima



Plateaux


Hill Climbing
Foothill Problem
Plateau Problem
Ridge Problem

Ridges


Hill Climbing
Foothill Problem
Plateau Problem
Ridge Problem

Hill climbing: Problem

- Shape of state space landscape strongly influences the success of the search process. A very spiky surface which is flat in between the spikes will be very difficult to solve.
- Can be combined with nondeterministic search to recover from local maxima.
- **Random-restart hill-climbing** is a variant in which reaching a local maximum causes the current state to be saved and the search restarted from a random point.
  - After several restarts, return the best state found. With enough restarts, this method will find the optimal solution.
- **Gradient descent** is an inverted version of hill-climbing in which better states are represented by lower *cost* values. Local *minima* cause problems instead of local maxima.

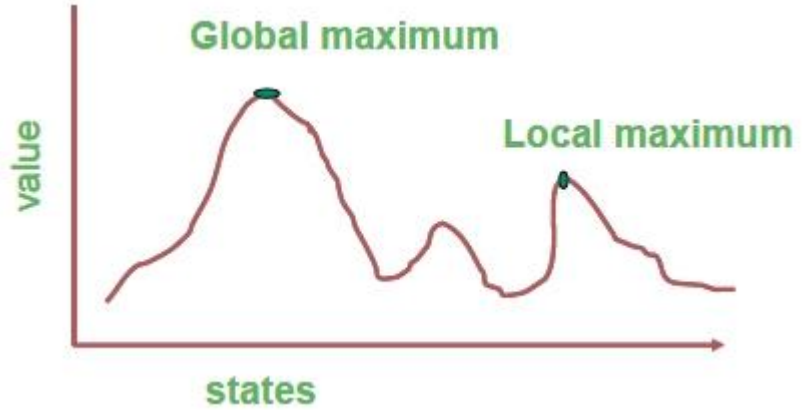
Hill climbing: Problem…

- *Complete state formulation for 8 queens*

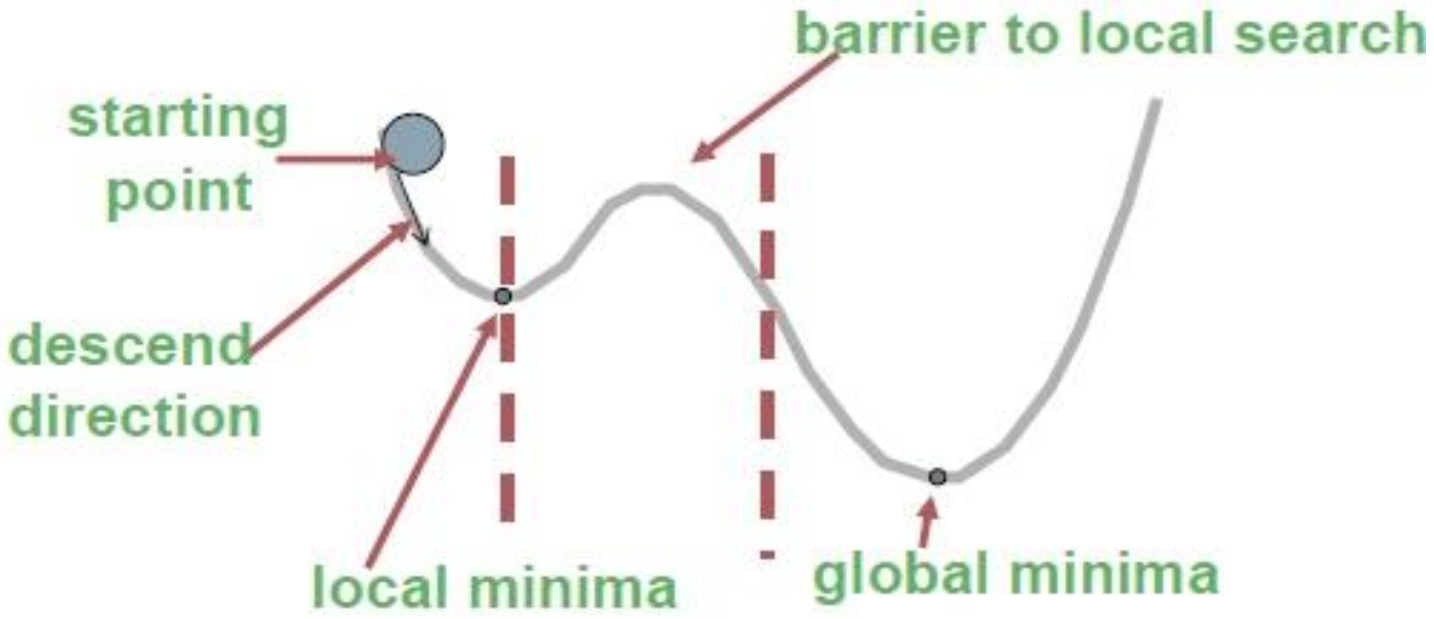
  – Successor function: move a single queen to another square in the same column

  – Cost: number of pairs that are attacking each other.

- *Minimization problem*
- *Problem: depending on initial state, may get stuck in local extremum.*



How do you avoid this local minima?



Consequences of Occasional Ascents

Simulated annealing: basic idea

- *From current state, pick a random successor state;*
- *If it has better value than current state, then "accept the transition," that is, use successor state as current state;*
- *Otherwise, do not give up, but instead flip a coin and accept the transition with a given probability (that is lower as the successor is worse).*
- *So we accept to sometimes "un-optimize" the value function a little with a non-zero probability.*

Simulated annealing

- Annealing is a process in metallurgy where metals are slowly cooled to make them reach a state of low energy where they are very strong.
- Simulated annealing is an analogous method for optimization. It is typically described in terms of thermodynamics.
- The random movement corresponds to high temperature; at low temperature, there is little randomness.
- Simulated annealing is a process where the temperature is reduced slowly, starting from a random search at high temperature eventually becoming pure greedy descent as it approaches zero temperature.
- The randomness should tend to jump out of local minima and find regions that have a low heuristic value; greedy descent will lead to local minima.
- At high temperatures, worsening steps are more likely than at lower temperatures.

Simulated annealing

- To control how many worsening steps are accepted, there is a positive real-valued temperature T.
- Suppose A is the current assignment of a value to each variable.
- Suppose that $h(A)$ is the evaluation of assignment A to be minimized.
- For solving constraints, h is typically the number of conflicts (so we need to minimize the number of conflicts).
- Simulated annealing selects a neighbor at random, which gives a new assignment A'. If $h(A') \leq h(A)$, it accepts the assignment and A' becomes the new assignment. Otherwise, the assignment is only accepted randomly with probability
$$e^{(h(A)-h(A'))/T}.$$
- Thus, if $h(A')$ is close to $h(A)$, the assignment is more likely to be accepted.
- If the temperature is high, the exponent will be close to zero, and so the probability will be close to 1.
- As the temperature approaches zero, the exponent approaches -∞, and the probability approaches zero.

Simulated annealing…

- Instead of restarting from a random point, we can allow the search to take some downhill steps to try to escape local maxima.
- Probability of downward steps is controlled by **temperature** parameter.
- High temperature implies high chance of trying locally "bad" moves, allowing nondeterministic exploration.
- Low temperature makes search more deterministic (like hill-climbing).
- Temperature begins high and gradually decreases according to a predetermined **annealing schedule**.
- Initially we are willing to try out lots of possible paths, but over time we gradually settle in on the most promising path.
- If temperature is lowered slowly enough, an optimal solution will be found.
- In practice, this schedule is often too slow and we have to accept suboptimal solutions.

Simulated annealing: Algorithm

```
set current to start state
for time = 1 to infinity {
   set Temperature to annealing_schedule[time]
   if Temperature = 0 {
      return current
   }
   randomly pick a next state from successors of current
   set ΔE to value(next) - value(current)
   if ΔE > 0 {
      set current to next
   } else {
      set current to next with probability e^(ΔE/Temperature)
   }
}
```

Simulated Annealing