Local Search and Optimization Problems

- Hill-climbing
- Simulated annealing
- Local beam search
- Genetic Algorithms

Local Search Algorithms

In many optimization problems the path to a goal state is irrelevant. The goal state itself is the solution.

Example:

- Finding a configuration satisfying certain constraints, e.g., the 8-queens problem or a job-shop scheduling problem.

In such cases, we can use **iterative improvement**: start with a single current state, and try to improve it!

The same framework is applicable to problems where the path appears to be of interest (e.g., TSP) if these problems can be casted in a more appropriate (but equivalent) way.
Local Search Algorithms (cont'd)

Local search algorithms work as follows:

- Pick a “solution” from the search space and evaluate it. Define this as the current solution.
- Apply a transformation to the current solution to generate and evaluate a new solution.
- If the new solution is better than the current solution then exchange it with the current solution; otherwise discard the new solution.
- Repeat steps 2 and 3 until no transformation in the given set improves the current solution.

Thus local search algorithms operate using a single current state (rather than multiple paths as e.g., A*) and generally move only to neighbours of that state.

At each step of a local search algorithm we have a complete but imperfect solution to a search problem. Other algorithms we saw previously (e.g., A*) work with partial solutions and extend them to complete ones.

Good properties of local search algorithms:

- Constant space
- Suitable for on-line as well as off-line problems.
- Can find reasonable solutions in large solution spaces where exhaustive search would fail miserably.
Iterative Improvement Algorithms

Idea: Start with a “solution” and make modifications until you reach a solution. Graphically:

Example: The Travelling Salesman Problem (TSP)

TSP: Let $G$ be a (directed or undirected) graph of $n$ nodes with each edge assigned a non-negative cost. Find the lowest-cost path of $G$ that visits each node only once and returns to a given initial node.

Solution set: the set of permutations of the $n$ cities.

Any permutation yields an ordered list of the cities to be visited, starting at the salesman’s home base, and continuing to the last location before returning home.
Algorithm 2-Opt

1. Start with an arbitrary complete tour $T$ (i.e., a random permutation).

2. Define the \textbf{neighbourhood} of $T$ as the set of all tours that can be reached by a \textbf{2-interchange} move.

3. \textbf{Search} in the neighbourhood of $T$ for a new tour $T'$. If this tour is better than $T$ (i.e., it has lower cost), then replace $T$ with $T'$.

   If you cannot find a better tour, \textbf{terminate}.

The resulting permutation is called \textbf{2-optimal}.

\textbf{2-Interchange Move}

This move \textbf{deletes} two non-adjacent edges, thus breaking the tour into two paths, and then \textbf{reconnects} those paths in the other possible way.
The State Space Landscape

Hill-Climbing Search (Gradient Steepest Ascent)

function HILL-CLIMBING(problem)
returns a state that is a local maximum
inputs: problem, a problem
local variables: current, a node
 neighbour, a node

current ← MAKE_NODE(RANDOM_STATE[problem])
loop do
  neighbour ← a highest-valued successor of current
  if VALUE[neighbour] ≤ VALUE[current] then return current
  current ← neighbour
end
Hill-Climbing Search (cont’d)

- Successors are searched in a systematic way.
- When choosing a highest-valued successor, break ties randomly.
- Change “highest-valued” to “lowest-valued” and ≤ to ≥ to get “gradient steepest descent” (applicable to minimization problems).

Example: 8-queens

Formal specification:

- States: any arrangement of 8 queens on board
- Actions: Move a queen within its column.
  In each step we have 8*7=56 possible successor states.
- Goal test: 8 queens on board, none attacked
- **Evaluation function (cost):** Number of pairs of queens that attack each other.

Thus we have a **minimization problem**: find a state that minimizes the evaluation function.
Example: 8-queens (cont’d)

The value of cost for the above state is 17. The numbers in the squares show the new costs if a queen is moved within its column.

Hill-Climbing (cont’d)

Problems:
- Local optima
- Plateaux (flat local optima or shoulders)
- Ridges

How can we cope with these problems? The proper choice might be problem dependent.
Hill-Climbing (cont’d)

Example: 8-queens (cont’d)

The value of cost for the above state is 1. All the neighbours of this state have cost > 1 thus we have a local minimum.
Example: Ridges

Here we have a sequence of local maxima; these are very difficult to navigate using local search.

Hill-Climbing for 8-queens

Let us start with a randomly generated 8-queens state. Then, steepest ascent hill-climbing performs as follows:

- It solves 14% of the problems within 4 steps on average.
- It gets stuck in local optima 86% of the time within 3 steps on average.

Reminder: Total state space $8^8 \approx 17$ million states.
**Sideways Moves**

When hill-climbing reaches a plateau and there are no uphill moves then it stops. Alternatively, we could resort to a **sideways move**: a move to a state which has the **same value** as the current one.

However, we have to be careful so that we do not go into an infinite loop (i.e., when we are on a plateau that is not a shoulder). An idea that works in some cases is to limit the number of consecutive sideways moves.

**Example:** If we limit the number of consecutive sideways moves to 100 in the 8-queens problem, this raises the percentage of solved problems to 94%.

---

**Variations of Hill-Climbing**

- **First-choice hill climbing:** Generates successors randomly until one is generated that is better than the current state.
  
  This is a good strategy for states with many (e.g., thousands) of successors.

- **Stochastic hill climbing:** Chooses randomly among the uphill moves. The probability of selection can vary with steepness.
  
  This variation converges more slowly than steepest ascent, but in some state landscapes it finds better solutions.
How do we Avoid Local Optima?

We will present two algorithms that avoid local optima:

- Random-Restart Hill-Climbing
- Simulated Annealing

Random-Restart Hill-Climbing

Advice: If at first you don’t succeed, try, try again!

Random-restart hill-climbing conducts a series of hill-climbing searches from randomly generated initial states, stopping when a goal is found.

With probability approaching 1, we will eventually generate a goal state as the initial state.

If each hill-climbing search has a probability $p$ of success, then the expected number of restarts required to reach a solution is $1/p$. 
Random-Restart Hill-Climbing (cont’d)

Example: 8-queens
As we saw earlier, \( p \approx 0.14 \).
In this case we need roughly 7 iterations (6 failures and 1 success).

**Expected number of steps:** Number of steps of one successful iteration plus \( 1/p - 1 \) times the number of steps of a failed iteration. This is roughly \( 6*3+4=22 \) steps in our case (using the number of steps for success/failure computed earlier).

Random restart hill-climbing is **very effective** for 8-queens. Even for **three million queens**, the approach can find solutions in under a minute.

Random-Restart Hill-Climbing (cont’d)

The success of random-restart hill-climbing depends very much on the **shape** of the state space.
There are practical problems with state spaces with very bad shapes.
Simulated Annealing

A hill-climbing algorithm that never makes “downhill” moves towards states with lower value can be incomplete.

A random walk, i.e., moving to a successor chosen uniformly at random from the set of successors, is complete (proof?) but extremely inefficient.

How can we combine both?

This is a classical tradeoff between exploration of the search space and exploitation of the imperfect solution at hand. How do we resolve this tradeoff?

Simulated Annealing (cont’d)

Physical analogue: Annealing of metals is the process used to temper or harden metals and glass by heating them to a high temperature and then gradually cooling them, thus allowing the material to coalesce into a low-energy crystalline state.

The discovery of the simulated annealing algorithm is an instance of the use of ideas from statistical mechanics (an area of condensed matter physics) to solving large and complex optimization problems.

Statistical mechanics concentrates on analyzing aggregate properties of large numbers of atoms to be found in samples of liquid or solid matter. See the paper on simulated annealing by Kirkpatrick et. al. in Science, Volume 220, Number 4598, May 1983.
### Statistical Mechanics and Optimization

<table>
<thead>
<tr>
<th>Physical System</th>
<th>Optimization Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>state</td>
<td>feasible solution</td>
</tr>
<tr>
<td>energy</td>
<td>evaluation function</td>
</tr>
<tr>
<td>ground state</td>
<td>optimal solution</td>
</tr>
<tr>
<td>quenching</td>
<td>local search</td>
</tr>
<tr>
<td>temperature</td>
<td>control parameter $T$</td>
</tr>
<tr>
<td>careful annealing</td>
<td>simulated annealing</td>
</tr>
</tbody>
</table>

---

#### Simulated Annealing (cont’d)

Simulated annealing solves the tradeoff among **exploration** and **exploitation** as follows.

At every iteration, a **random** move is chosen. If it improves the situation then the move is accepted, otherwise it is accepted with some probability less than 1.

The probability decreases exponentially with the badness of the move. It also decreases with respect to a **temperature parameter** $T$.

Simulated annealing starts with a high value of $T$ and then $T$ is gradually reduced. At high values of $T$, simulated annealing is like pure random search. Towards the end of the algorithm when the values of $T$ are quite small, simulated annealing resembles ordinary hill-climbing.
Simulated Annealing (cont’d)

**function** SIMULATED-ANNEALING(problem, schedule)
**returns** a solution state

**inputs:** problem, a problem

  schedule, a mapping from time to “temperature”

**local variables:** current, a node next, a node T, the temperature

`current ← MAKE_NODE(RANDOM_STATE[problem])`

for `t ← 1 to ∞` do
  `T ← schedule[t]`
  if `T = 0` then return `current`
  `next ←` a randomly selected successor of `current`
  `ΔE ← VALUE[next] − VALUE[current]`
  if `ΔE > 0` then `current ← next`
  else `current ← next` only with probability `e^{ΔE/T}`

---

**Example**

Let us assume that the current and next point in a search space differ by 13 (i.e., `ΔE = −13`). Then:

<table>
<thead>
<tr>
<th>T</th>
<th>e^{ΔE/T}</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.000002</td>
</tr>
<tr>
<td>5</td>
<td>0.0743</td>
</tr>
<tr>
<td>10</td>
<td>0.2725</td>
</tr>
<tr>
<td>20</td>
<td>0.52</td>
</tr>
<tr>
<td>50</td>
<td>0.77</td>
</tr>
<tr>
<td>10^{10}</td>
<td>0.9999...</td>
</tr>
</tbody>
</table>

Thus, at high values of `T`, simulated annealing behaves like a random walk; at low values of `T`, it behaves like hill-climbing.
Simulated Annealing (cont’d)

Simulated annealing finds a **global optimum** with probability approaching 1 if the schedule lowers $T$ slowly enough.

The exact bound for parameter $t$ and schedule for $T$ is usually problem dependent. Thus we need to **experiment heavily** with every new problem at hand to see whether simulated annealing makes a difference.

Simulated annealing is a **very popular algorithm** and has been used to solve various classes of interesting optimization problems (e.g., VLSI layout problems, job-shop scheduling problems etc.)

Local Beam Search

**Idea:** Why not keep more than just one state (e.g., $k$ states) in memory?

At each iteration, all the successors of the $k$ states are generated. If one of them is a solution then we halt. Otherwise $k$ states are selected and the process is repeated. We expect good successors to “attract the attention”.

**Diversity** is important so we don’t get stuck in bad regions of the search space.

**Stochastic beam search** chooses $k$ successors at random, with the probability of choosing a successor being an increasing function of its value.

Similar to natural selection?
Genetic Algorithms

A genetic algorithm (GA) is a variant of stochastic beam search in which successor states are generated by combining two parent states (sexual reproduction).

Concepts:

- **Individuals** represent states. They are denoted by strings over an alphabet usually \{0, 1\}.
- **Populations** are sets of individuals.
- **Fitness function** is an evaluation function for rating each individual.

Operations:

- **Reproduction**: a new individual is born by combining two parents.
- **Mutation**: a new individual is slightly modified.
Example: 8-queens

(a) Initial Population
(b) Fitness Function
(c) Selection
(d) Cross-Over
(e) Mutation
A Genetic Algorithm

function Genetic-Algorithm(population, Fitness-Fn) returns an individual
inputs: population, a set of individuals
         Fitness-Fn, a function that measures the fitness of an individual
repeat
    new_population ← ∅
    loop for i from 1 to Size(population) do
        x ← RANDOM-SELECTION(population, Fitness-Fn)
        y ← RANDOM-SELECTION(population, Fitness-Fn)
        child ← REPRODUCE(x, y)
        if (small random probability) then child ← MUTATE(child)
        add child to new_population
    population ← new_population
until some individual is fit enough, or enough time has elapsed
return the best individual in population, according to Fitness-Fn

Genetic Algorithms (continued)

Intuitively the advantage of genetic algorithms comes from the ability of crossover to combine large blocks of letters that have evolved independently to produce useful functions.

The theory of genetic algorithms explains how this works using the idea of a schema.

Example: 246*****

Representation of instances is very important in genetic algorithms. There is still much work to be done in order to understand under what conditions genetic algorithms work very well.
Readings

- Chapter 4, Section 4.3 of AIMA.
- Parts of Sections 3 and 5 of the book: