Local Search

Adapted from Russell and Norvig, Artificial Intelligence, Chapter 4
Marsland, Machine Learning, Section 9.4-9.6
Lugar, Artificial Intelligence, Chapters 3 & 4
Search in Artificial Intelligence

- In Artificial Intelligence, problem solving is thought of as searching for solutions in a space of solutions.
- For example, consider playing a 1-person game like the game of 15-or 8-puzzle.

The problem is generated by “creating” the search space for the game and then searching it for a path from the start to the end.
The search space for the 8-puzzle game is shown below. We can search this space in many different ways. Depending on the game or the problem, the search space can be very large. For example, the search space for chess is supposedly of size $\Theta(10^{120})$. 
Search space for 8-puzzle

- Search algorithms include the so-called “uninformed” searches or exhaustive searches: Breadth-first search, Depth-first search, etc.
- Since the search space is potentially very large, it is not possible to obtain solutions in the time needed by exhaustive search. For example, in chess, we may have only a few minutes (say, 2 minutes or 5 minutes) to make a move, depending on the tournament rules.
- In such situations, we abandon exhaustive searches in favor of so-called heuristic searches.
- In heuristic search, we assign a heuristic evaluation to every node using some mathematical function based on the configuration of the state of the problem (here, the state of the board).
- Heuristics may be simple or complicated. Simple informed ones are preferred.
- Heuristic search is not perfect. It may sometimes find bad solutions, but most of the time it does reasonably well.
- In simple heuristic search, given a board position, we can create all possible next states and choose the one with the maximum heuristic value.
Search space for 8-puzzle

- Below are three simple heuristics for the 8-puzzle game.
- We can combine the third one with one of the first two to obtain something still better.

<table>
<thead>
<tr>
<th>Tiles out of place</th>
<th>Sum of distances out of place</th>
<th>2 x the number of direct tile reversals</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>0</td>
</tr>
</tbody>
</table>
Using Heuristics in the 8-puzzle Game

\[ g(n) = 0 \]

\[ g(n) = 1 \]

Values of \( f(n) \) for each state,

\[ 6 \quad 4 \quad 6 \]

where:
\[ f(n) = g(n) + h(n), \]
\[ g(n) = \text{actual distance from } n \] to the start state, and
\[ h(n) = \text{number of tiles out of place.} \]
• Below we see the search space for a simple heuristic search.
• There are more complex heuristic searches with names like A-search, A*-search, etc.
• The goal is to search quickly and find a reasonable solution, if not the best.
Local Search Algorithms

- The search algorithms we have seen so far include systematic search (breadth-first, depth-first, iterative deepening, etc.) where we look at the entire search space in a systematic manner till we have found a goal (or all goals, if we have to).
- We also have seen heuristic search (best-first, A*-search) where we compute an evaluation function for each candidate node and choose the one that has the best heuristic value (the lowest heuristic value is usually the best).
- In both systematic search and heuristic search, we have to keep track of what’s called the frontier (some books call it open or agenda) data structure that keeps track of all the candidate nodes for traversal in the next move.
- We make a choice from this frontier data structure to choose the next node in the search space we go to.
- We usually have to construct the path from the start node to the goal node to solve a problem.
Local Search or Iterative Improvement

- In many optimization problems, we need to find an optimal path to the goal from a start state (e.g., find the shortest path from a start city to an end city), but in many other optimization problems, the path is irrelevant.

- The goal state itself is the solution.
  - Find optimal configuration (e.g., TSP), or
  - Find configuration satisfying constraints (e.g., n-queens)

- Here, it is not important how we get to the goal state, we just need to know the goal state.

- In such cases, can use iterative improvement algorithms: keep a single “current” state, and try to improve it.
Iterative improvement

- The evaluation can be the length of the tour in the TSP problem.

**Example: Travelling Salesperson Problem**

Start with any complete tour, perform pairwise exchanges
Iterative improvement example: n-queens

- Goal: Put n chess queens on an n x n board, with no two queens on the same row, column, or diagonal.

- Here, goal state is initially unknown but is specified by constraints that it must satisfy.
An 8-queens state and its successor

Figure 4.3  FILES: figures/8queens-successors.eps (Wed Nov 4 16:23:55 2009) figures/8queens-local-minimum.eps (Wed Nov 4 16:14:15 2009). (a) An 8-queens state with heuristic cost estimate $h = 17$, showing the value of $h$ for each possible successor obtained by moving a queen within its column. The best moves are marked. (b) A local minimum in the 8-queens state space; the state has $h = 1$ but every successor has a higher cost.
Plotting evaluations of a space

- In the case of the 8-queens problem, each node has $8 \times 7 = 56$ successors.
- If we plot all the evaluations of successor states when we are at a certain state, we get a graph. In this case, it will be a 3-D graph. We will have $x$ and $y$ for the board positions and $z$ will be the evaluation.
- Similarly, for any problem, we can plot the evaluations of all successor nodes.
Plotting evaluations of a space

- A search space can look like the graph below. Here we assume state space has only one dimension.
Plotting evaluations of a space

- Depending on the search/optimization problem, the search spaces can look very bizarre/difficult to navigate.
- The vertical value is the heuristic value, the search space is 2-D.
Plotting evaluations of a space

- Some additional examples of search space landscape.
Hill climbing (or gradient ascent/descent)

- This is a very simple algorithm for finding a local maximum.
- Iteratively maximize “value” of current state, by replacing it by successor state that has highest value, as long as possible.
- In other words, we start from an initial position (possibly random), the move along the direction of steepest ascent/gradient (descent/negative of gradient, for minimization), go along the direction for a little while; and repeat the process.
We usually don’t know the underlying search space when we start solving a problem.

How would Hill-Climbing do on the following problems?
What we think hill-climbing looks like

In other words, when we perform hill-climbing, we are short-sighted. We can’t really see far. Thus, we make local best decisions with the hope of finding a global best solution. This may be an impossible task.
Problems with Hill climbing

- Local optima (maxima or minima): A local maximum is a peak that is higher than each of its neighboring states, but lower than the global maximum.

- Ridges: A sequence of local maxima. Ridges are very difficult to navigate for a hill-climbing algorithm.

- Plateaus: An area of the state space where the evaluation function is flat. It can be a flat local maximum, from which no uphill path exists, or a shoulder, from which progress is possible.
Statistics from a real problem: 8-queen problem

- Starting with a randomly generated 8-queens state, hill-climbing gets stuck 86% of the time at a local maximum or a ridge or a plateau.
- That means, it can solve only 14% of the 8-queens problems given to it.
- However, hill-climbing is very efficient.
  - It takes only 4 steps on an average when it succeeds.
  - It takes only 3 steps when getting stuck.
- This is very good, in some ways, in a search space that is $8^8 \approx 17$ million states (Each queen can be in 8 places, independent of each other. First queen can be in 8 places, second can be in 8 places, ....).
- What we want: An algorithm that compromises on efficiency, but can solve more problems.
Some ideas about making Hill-climbing better

- Allowing sideways moves
  - When stuck on a ridge or plateau (i.e., all successors have the same value), allow it to move anyway hoping it is a shoulder and after some time, there will be a way up.

- How many sideways moves?
  - If we always allow sideways moves when there are no uphill moves, an infinite loop may occur if it’s not a shoulder.

- Solution
  - Put a limit on the number of consecutive sideways moves allowed.

- Experience with a real problem
  - The authors allowed 100 consecutive sideways moves in the 8-queens problem.
  - This raises the percentage of problems solved from 14% to 94%
  - However, now it takes on average 21 steps when successful and 64 steps when it fails.
Variants of Hill-climbing

- **Stochastic hill-climbing**
  - Choose at random from among the uphill moves, assuming several uphill moves are possible. Not the steepest.
  - It usually converges more slowly than steepest ascent, but in some state landscapes, it finds better solutions.

- **First-choice hill climbing**
  - Generates successors randomly until one is generated that is better than current state.
  - This is a good strategy when a state may have hundreds or thousands of successor states.

- **Steepest ascent, hill-climbing with limited sideways moves, stochastic hill-climbing, first-choice hill-climbing are all incomplete.**
  - Complete: A local search algorithm is complete if it always finds a goal if one exists.
  - Optimal: A local search algorithm is complete if it always finds the global maximum/minimum.
Variants of Hill-climbing

• Random-restart hill-climbing
  • If you don’t succeed the first time, try, try again.
  • If the first hill-climbing attempt doesn’t work, try again and again and again!
  • That is, generate random initial states and perform hill-climbing again and again.
• The number of attempts needs to be limited, this number depends on the problem.
• With the 8-queens problem, if the number of restarts is limited to about 25, it works very well.
• For a 3-million queen problem with restarts (not sure how many attempts were allowed) with sideways moves, hill-climbing finds a solution very quickly.
• Luby et al. (1993) has showed that in many cases, to restart a randomized search after a particular fixed amount of time is much more efficient than letting each search continue indefinitely.
Final Words on Hill-climbing

- Success of hill-climbing depends on the shape of the state space landscape.
- If there are few local maxima and plateaus, random-start hill-climbing with sideways moves works well.
- However, for many real problems, the state space landscape is much more rugged.
- NP-complete problems are hard because they have exponential number of local maxima to get stuck on.
- In spite of all the problems, random-hill climbing with sideways moves works and other approximation techniques work reasonably well on such problems.
Simulated Annealing

- A hill-climbing algorithm that never makes a “downhill” move toward states with lower value (or higher cost) is guaranteed to be incomplete, because it can get stuck in a local maximum.
- In contrast, a purely random walk—that is, moving to a successor chosen uniformly at random from the set of successors—is complete but extremely inefficient.
- Therefore, it is reasonable to try to combine hill-climbing with a random walk in some way to get both efficiency and completeness.
- **Simulated annealing** is one such algorithm.
Simulated Annealing

- To discuss simulated annealing, we need to switch our point of view from hill climbing to **gradient descent** (i.e., minimizing cost; simply place a negative sign in front of the cost function in hill climbing).
Simulated Annealing

- Analogy: Imagine the task of getting a ping-pong ball into the deepest crevice in a bumpy surface.
- If we let the ball roll, it will come to rest at a local minimum.
- If we shake the surface, we can bounce the ball out of the local minimum.
- The trick is to shake just hard enough to bounce the ball out of the local minima, but not hard enough to bounce the ball out of the global minimum.
- The simulated annealing algorithm starts by shaking hard (i.e., at a high temperature) and then gradually reduces the intensity of shaking (i.e., lower the temperature)
Minimizing energy

- In this new formulation of the problem
  - We compare our state space to that of a physical system (a bouncing ping-pong ball in a bumpy surface) that is subject to natural laws
  - We compare our value function to the overall potential energy $E$ of the system.
- On every update, we want $\Delta E \leq 0$, i.e., the energy of the system should decrease.
- We start with a high energy state and reduce the energy of the system slowly letting it randomly walk/jump around/search at that energy level or temperature it is at, before reducing the energy level or temperature.
Local Minima Problem

- Question: How do you avoid this local minimum?
Consequences of the Occasional Ascents

Desired effect

Help escaping the local optima.

Adverse effect

Might pass global optima after reaching it

(easy to avoid by keeping track of best-ever state)
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 (standard hill-climbing, in this case hill-descending).
- Otherwise, do not give up, but instead flip a coin and accept the transition with a given probability (that is accept successors that do make the state better with a certain probability).
- In other words, although we want our next step to be a good step, if necessary we allow bad steps that take us in the opposite direction with a non-zero probability.
Simulated annealing algorithm

• Idea: Escape local extrema by allowing “bad moves,” but gradually decrease their size and frequency.
• Even though it says $\infty$ number of times, it means a large no of times.
• We are working with a situation where a state with a lower energy is a better state.

for $t=1$ to $\infty$ do
    $T \leftarrow$ schedule $(t)$
    if $T=0$ then return current
    next $\leftarrow$ a randomly selected successor of current
    $\Delta E = next.value - current.value$
    if $\Delta E < 0$ then current $\leftarrow$ next
    else current $\leftarrow$ next only with probability $e^{-\Delta E / T}$
endfor
Note on simulated annealing: limit cases

- The **for** loop of the simulated annealing algorithm is similar to hill-climbing (but we are working with a situation where lower energy is better).
- However, instead of picking the best move, it picks a random move (like stochastic hill climbing).
- If the random move improves the situation, it is always accepted.
- Otherwise, the algorithm accepts the move with some probability less than 1.
- The probability decreases exponentially with the “badness” of the move—the amount $\Delta E$ by which the evaluation is worsened. I.e., a less bad move is more likely.
- The probability also decreases as the “temperature” $T$ goes down: “bad” moves are more likely to be allowed at the start when $T$ is high, and they become more unlikely as $T$ decreases.
- If the “schedule” lowers $T$ slowly enough, the algorithm will find a global optimum with probability approaching 1.
Local Beam Search

- Keeping just one node in memory might seem an extreme reaction to the problem of memory limitation, but this is what we do in local searches and in simulated annealing.
- **Local Beam Search** algorithm keeps track of k states rather than 1.
- It begins with k randomly generated states.
- At each step, all the successors of these k states are generated.
- If any of them is a goal, the algorithm halts.
- Otherwise, it selects the best k successors from the complete list and repeats.
Local Beam Search vs. Hill Climbing

- At first sight, a local beam search with $k$ states may seem like running $k$ random restart hill-climbing algorithms in parallel.
- However, there is a difference.
- In parallel random-restart hill-climbing, each search process runs independently of the others.
- In a local beam search, useful information is passed among the parallel search threads.
- A local beam search algorithm quickly abandons unfruitful searches and moves its resources to where the most progress is being made.
- Problem: All $k$ states can quickly become concentrated in a small region of the search space, making it an expensive version of hill climbing.
- Solution: **Stochastic local beam search**: Instead of choosing the $k$ best successors, choose $k$ successors at random.