Constraints and Local Search

CS 486/686: Introduction to Artificial Intelligence
Overview

- **Uninformed Search**
  - Very general: assumes no knowledge about the problem
  - BFS, DFS, IDS

- **Informed Search**
  - Heuristics
  - A* search and variations

- **Search and Optimization**
  - What are the problem features?
  - Iterative improvement: hill climbing, simulated annealing
  - Genetic algorithms
Introduction

Both uninformed and informed search systematically explore the search space

Keep 1 or more paths in memory

**Solution is a path to the goal**

For many problems the path is unimportant
Examples

AV ~B V C
~A V C V D
B V D V ~E
~C V ~D V ~E
...

Agents = dispatch centers

Diagram showing trucks and depots with connections.

Diagram showing a network with various nodes and edges.
Informal Characterization

Combinatorial structure being optimized
Constraints have to be satisfied
There is a cost function
   We want to find a good solution
Search all possible states is infeasible
   Often easy to find some solution to the problem
   Often provably hard (NP-complete) to find the best solution
Typical Example: 4 Queens

Start with a “complete” state
Operators reassign variables
  Choose variable at random
  Choose value using min-conflicts heuristic
Continue until solved
Performance for N-Queens

Given a random initial state, can solve n-queens in almost constant time for arbitrary n with high probability (n=10,000,000)!

This seems to hold for almost any randomly generated CSP except for a small set!
Typical Example: TSP

Goal is to minimize the length of the route

**Constructive method**: Start from scratch and build up a solution (using A* etc)

**Iterative improvement method**: Start with solution (may be suboptimal or broken) and improve it
Iterative Improvement Methods

**Idea:** Imagine all possible solutions laid out on a landscape

**Goal:** find the highest (or lowest) point
Iterative Improvement Methods

Start at some random point (potential solution)

Generate all possible points to move to

If the set is not empty, choose a point and move to it

If you are stuck (set is empty), then restart
Hill Climbing (Gradient Descent)

Main idea: Always take a step in the direction that improves the current solution value the most

Note: Variation of best-first search

Application: Very popular for learning algorithms

“...like trying to find the top of Mt Everest in a thick fog while suffering from amnesia”, Russell and Norvig
Hill Climbing
(Discrete Version)

1. Start with some initial configuration $S$, with value $V(S)$
2. Generate $\text{Moveset}(S) = \{S_1, \ldots, S_n\}$
3. $S_{\text{max}} = \text{argmax}_{i} V(S_i)$
4. If $V(S_{\text{max}}) < V(S)$ return $S$ (local optimum)
5. Let $S \leftarrow S_{\text{max}}$ Go to 2
Judging Hill Climbing

Good news
Easy to program!
Requires no memory of where we have been!
Judging Hill Climbing

**Good news**

*Easy to program!*
Requires no memory of where we have been!

**Bad news**

Not necessarily complete
Not optimal
It can get stuck in local optima/plateaus
Improving Hill Climbing

Plateaus

• Allow for sideways moves
  • But be careful since might move sideways forever

Local Maxima

• Random restarts:

  *If at first you do not succeed, try, try again!*
Simulated Annealing

Escape local maxima by allowing “downhill moves”

1. Start with some initial configuration S, with value V(S)
2. Generate Moveset(S)= \{S_1,\ldots,S_n\}
3. Choose $S_i$ \textbf{randomly} from Moveset(S)
4. Define $\Delta V=V(S_i)-V(S)$
5. If $\Delta V>0$ then $S \leftarrow S_i$
   
   \textbf{else with probability} $p$, $S \leftarrow S_i$
6. Go to 2
What About $p$?

**Main Issue**: How should we choose the probability of making a “bad” move?

**Ideas:**

$p=0.1$ (or some fixed value)?

Decrease $p$ with time?

Make $p$ a function of $|V-V_i|$?

...
Selecting Moves in Simulated Annealing

• If new value \( V_i \) is **better** than old value \( V \) then **definitely** move to new solution

• If new value \( V_i \) is **worse** than old value \( V \) then move to new solution with *some probability*

**Boltzmann Distribution**

\[
e^{-\frac{\Delta V}{T}}
\]
Selecting Moves in Simulated Annealing

**Boltzmann Distribution**: $T > 0$ is a parameter called temperature. It starts high and decreases over time towards 0. If $T$ is close to 0 then the probability of making a bad move is almost 0.

$$e^{\frac{\Delta V}{T}}$$

![Graph showing the Boltzmann distribution with different temperatures and the probability of accepting a move.](image)
Properties to Simulated Annealing

When T is high:

- **Exploratory phase**: even bad moves have a chance of being picked (~ random walk)

When T is low:

- **Exploitation phase**: “bad” moves have low probability of being chosen (randomized hill climbing)

If T is decreased slowly enough then simulated annealing is (theoretically) guaranteed to reach optimal solution
Genetic Algorithms

• Populations are encoded into a representation which allows certain operations to occur
• An encoded candidate solution is an individual
• Each individual has a fitness
  • Numerical value associated with its quality of solution
• A population is a set of individuals
• Populations change over generations by applying operators to them
  • Operations: selection, mutation, crossover
Typical Genetic Algorithm

- Initialize: Population $P \leftarrow N$ random individuals
- Evaluate: For each $x$ in $P$, compute fitness($x$)
- Loop
  - For $i=1$ to $N$
    - Select 2 parents each with probability proportional to fitness scores
    - Crossover the 2 parents to produce a new bitstring (child)
    - With some small probability mutate child
    - Add child to population
  - Until some child is fit enough or you get bored
- Return best child in the population according to fitness function
Selection

• **Fitness proportionate selection:**
  • Can lead to overcrowding

• **Tournament selection**
  • Pick i, j at random with uniform probability
  • With probability p select fitter one

• **Rank selection**
  • Sort all by fitness
  • Probability of selection is proportional to rank

• **Softmax (Boltzmann) selection:**

\[
P(i) = \frac{e^{\text{fitness}(i)/T}}{\sum_j e^{\text{fitness}(j)/T}}
\]
Crossover

• Combine parts of individuals to create new ones
• For each pair, choose a random crossover point
• Cut the individuals there and swap the pieces

<table>
<thead>
<tr>
<th>101</th>
<th>0101</th>
<th>011</th>
<th>1110</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cross over</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>0101</td>
<td>101</td>
<td>1110</td>
</tr>
</tbody>
</table>

Implementation: use a crossover mask m
Given two parents a and b the offspring are
\[(a^m)\lor(b^{\sim m})\text{ and } (a^{\sim m})\lor(b^m)\]
Mutation

• Mutation generates new features that are not present in original population
• Typically means flipping a bit in the string

100111 mutates to 100101

• Can allow mutation in all individuals or just in new offspring
Example
Summary

Useful for optimization problems

Often the **second-best way** to solve a problem

    If you can, use A* or linear programming or ...

    But there are cool applications: Scheduling umpires for US Open, solving jigsaw puzzles,...

Need to think about how to escape from local optima

    Random restarts

    Allowing for bad moves

...