Search and metaheuristics

Jaak Vilo
2010 Spring

Objective function

- An optimal solution
  - what is the measure that we optimise?
    - Any solution (satisfiability/SAT/ problem)
      - does the task have a solution?
      - is there a solution with objective measure better than X?
    - Minimal/maximal cost solution
    - A winning move in a game
    - A (feasible) solution with smallest nr of constraint violations (e.g. course time scheduling)

Search space

- Linear (list, binary search, ...)
- Trees, Graphs
- Real nr in [x,y)
  - Integers
- A point in high-dimensional space
- A subset of a larger set
- An assignment of variables (in SAT)
  - ...

The Propositional Satisfiability Problem (SAT)

Simple SAT instance (in CNF):

\[(a \lor b) \land (\neg a \lor \neg b)\]

\[\sim\] satisfiable, two models:

\[a = \text{true}, b = \text{false}\]
\[a = \text{false}, b = \text{true}\]

Tic-Tac-Toe

Consider the game of tic-tac-toe. Even if we use symmetry to reduce the search space of redundant moves, the number of possible paths through the search space is something like \(12 \times 7! = 60480\). That is a measure of the amount of work that would have to be done by a brute-force search.
TSP, nearest neighbour search

Constraints

- Time, space...
  - if optimal cannot be found, approximate
- All kinds of secondary characteristics
- Constraints
  - sometimes finding even a point in the valid search space is hard
Outline

- Best-first search
- Greedy best-first search
- A* search
- Heuristics
- Local search algorithms
- Hill-climbing search
- Simulated annealing search
- Local beam search
- Genetic algorithms

Greedy

- Set Cover
  - Greedy Approximation Algorithm
  - polynomial-time $\rho(n)$-approximation algorithm
    * $\rho(n)$ is a logarithmic function of set size

Set Cover Problem

Instance $(X, F)$:
- finite set $X$ (e.g. of points)
- family $F$ of subsets of $X$

$$X = \bigcup_{S \in F}$$

Problem: Find a minimum-sized subset $C \subseteq F$ whose members cover all of $X$.

NP-Complete

source: 91.503 textbook Cormen et al.

Greedy Set Covering Algorithm

Greedy: select set that covers the most uncovered elements

Greedy: select set that covers the most uncovered elements

The graph of function $G_2$ for $n = 2$. Infeasible solutions were as
Set Cover (proof continued)

Theorem:: GREEDY-SET-COVER is a polynomial-time \( \rho(n) \)-approximation algorithm for

\[ \rho(n) = H(\max\{ |S| : S \in F \}) \]

Proof: (continued)

Let \( C^* \) be an optimal cover. 
Let \( C \) be from GREEDY - SET - COVER. 

Cost assigned to optimal cover:

\[ |C| = \sum_{S \in C} \sum_{x \in S} c_x \]

Optimal for each \( S \in C \):

\[ \sum_{x \in S} c_x = \sum_{S \in C} \sum_{S \in C^*} \sum_{x \in S} c_x \leq \sum_{S \in C^*} \sum_{x \in S} c_x \]

We'll show that:

\[ \frac{1}{n} \sum_{x \in S} c_x \leq H(|S|) \]

And then conclude that:

\[ |C| \leq \sum_{S \in C^*} H(|S|) \leq |C^*|H(\max\{ |S| : S \in F \}) \]

How does this relate to harmonic numbers??

Classes of Search Techniques

<table>
<thead>
<tr>
<th>Search technique</th>
<th>Calculus-based techniques</th>
<th>Evolutionary techniques</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct methods</td>
<td>Simulated annealing</td>
<td></td>
</tr>
<tr>
<td>Indirect methods</td>
<td>Genetic algorithms</td>
<td></td>
</tr>
<tr>
<td>Search techniques</td>
<td>Guided random search</td>
<td>Dynamic programming</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Centralized</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Distributed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parallel</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Steady-state</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Generational</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sequential</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Local search algorithms

- In many optimization problems, the path to the goal is irrelevant; the goal state itself is the solution
- State space = set of "complete" configurations
- Find configuration satisfying constraints, e.g., \( n \)-queens
- In such cases, we can use local search algorithms
- Keep a single "current" state, try to improve it

Example: \( n \)-queens

- Put \( n \) queens on an \( n \times n \) board with no two queens on the same row, column, or diagonal

  ![Example: n-queens](image)

- May get stuck...
Problems

• Cycles
  – Memorize; Tabu search

• How to transfer valleys with bad choices only...

Tree/Graph search

• order defined by picking a node for expansion

• BFS, DFS

• Random, Best First, ...
  – Best – an evaluation function

Idea: use an evaluation function $f(n)$ for each node

– estimate of “desirability”

– Expand most desirable unexpanded node

Implementation:
Order the nodes in fringe in decreasing order of desirability
Priority queue

Special cases:
– greedy best-first search $f(n) = h(n)$ heuristic, e.g. estimate to goal
– $\text{A}^*$ search

$\text{A}^*$

• $f(n) = g(n) + h(n)$

– $g(n)$ – path covered so far in graph
– $h(n)$ – estimated distance from $n$ to goal

Admissible heuristics

• A heuristic $h(n)$ is admissible if for every node $n$, $h(n) \leq h^*(n)$, where $h^*(n)$ is the true cost to reach the goal state from $n$.

• An admissible heuristic never overestimates the cost to reach the goal, i.e., it is optimistic

• Example: $h_{\text{SLD}}(n)$ (never overestimates the actual road distance)

(SLD – shortest linear distance)

• Theorem: If $h(n)$ is admissible, $\text{A}^*$ using TREE-SEARCH is optimal

Optimality of $\text{A}^*$ (proof)

• Suppose some suboptimal goal $G_2$ has been generated and is in the fringe. Let $n$ be an unexpanded node in the fringe such that $n$ is on a shortest path to an optimal goal $G$.

  $f(G_2) = g(G_2)$ since $h(G_2) = 0$

  $g(G_2) > g(G)$ since $G_2$ is suboptimal

  $f(G) = g(G)$ since $h(G) = 0$

  $f(G_2) > f(G)$ from above
Optimality of A* (proof)

- Suppose some suboptimal goal $G_2$ has been generated and is in the fringe. Let $n$ be an unexpanded node in the fringe such that $n$ is on a shortest path to an optimal goal $G$.

- $f(G_2) = g(G_2)$ since $h(G_2) = 0$
- $g(G_2) > g(G)$ since $G_2$ is suboptimal
- $f(G_2) > f(G)$ from above
- Hence $f(G_2) > f(n)$, and A* will never select $G_2$ for expansion

Graph

- A Virtual graph/search space
  - valid states of Fifteen-game
  - Rubik’s cube

Solve

- Which move takes us closer to the solution?
- Estimate the goodness of the state

The Traveling Salesperson Problem (TSP)

- TSP – optimization variant:
  - For a given weighted graph $G = (V,E,w)$, find a Hamiltonian cycle in $G$ with minimal weight,
  - i.e., find the shortest round-trip visiting each vertex exactly once.
- TSP – decision variant:
  - For a given weighted graph $G = (V,E,w)$, decide whether a Hamiltonian cycle with minimal weight $\leq b$ exists in $G$.

TSP instance: shortest round trip through 532 US cities
### Search Methods

- **Types of search methods:**
  - systematic $\leftrightarrow$ local search
  - deterministic $\leftrightarrow$ stochastic
  - sequential $\leftrightarrow$ parallel

### Local Search (LS) Algorithms

- **search space $S$**
  (SAT: set of all complete truth assignments to propositional variables)
- **solution set** $S' \subseteq S$
  (SAT: models of given formula)
- **neighborhood relation** $N \subseteq S \times S$
  (SAT: neighboring variable assignments differ in the truth value of exactly one variable)
- **evaluation function** $g : S \rightarrow \mathbb{R}^+$
  (SAT: number of clauses unsatisfied under given assignment)

### Local Search:

- start from initial position
- iteratively move from current position to neighboring position
- use evaluation function for guidance

**Two main classes:**
- local search on *partial solutions*
- local search on *complete solutions*

### Local search for partial solutions

- Order the variables in some order.
- Span a tree such that at each level a given value is assigned a value.
- Perform a depth-first search.
- But, use heuristics to guide the search. Choose the best child according to some heuristics. *(DFS with node ordering)*

### Construction Heuristics for partial solutions

- **search space:** space of partial solutions
- **search steps:** extend partial solutions with assignment for the next element
- solution elements are often ranked according to a greedy evaluation function
**Nearest Neighbor heuristic for the TSP:**

- at any city, choose the closest yet unvisited city
  - choose an arbitrary initial city $i(1)$
  - at the $i$th step choose city $n(i + 1)$ to be the city $j$ that minimises $d(i, j); j \neq i(k), 1 \leq k \leq i$
- running time $O(n^2)$
- worst case performance $\frac{NN(x)}{OPT(x)} \leq 0.5(\log n) + 1$
- other construction heuristics for TSP are available

**DFS**

- Once a solution has been found (with the first dive into the tree) we can continue search the tree with DFS and backtracking.
- In fact, this is what we did with DFBnB.
- DFBnB with node ordering.

**Iterative Improvement (Greedy Search):**

- initialize search at some point of search space
- in each step, move from the current search position to a neighboring position with better evaluation function value

**Iterative Improvement for SAT**

- **Initialization:** randomly chosen, complete truth assignment
- **neighborhood:** variable assignments are neighbors if they differ in truth value of one variable
- **neighborhood size:** $O(n)$ where $n = \text{number of variables}$
- **evaluation function:** number of clauses unsatisfied under given assignment
Hill climbing

- Choose the neighbor with the largest improvement as the next state

\[
\text{f-value} = \text{evaluation(state)}
\]

Hil climbing function

\[
\begin{align*}
\text{Hill-Climbing}(\text{problem}) & \quad \text{returns a solution state} \\
\text{current} & \gets \text{Make-Node(Initial-State[problem])} \\
\text{loop do} & \\
\text{next} & \gets \text{a highest-valued successor of current} \\
\text{if Value[next] < Value[current] then} & \quad \text{return current} \\
\text{current} & \gets \text{next} \\
\text{end}
\end{align*}
\]

Problems with local search

Typical problems with local search (with hill climbing in particular)

- getting stuck in local optima
- being misguided by evaluation/objective function

Stochastic Local Search

- randomize initialization step
- randomize search steps such that suboptimal/worsening steps are allowed
- improved performance & robustness
- typically, degree of randomization controlled by noise parameter

Stochastic Local Search

Pros:
- for many combinatorial problems more efficient than systematic search
- easy to implement
- easy to parallelize

Cons:
- often incomplete (no guarantees for finding existing solutions)
- highly stochastic behavior
- often difficult to analyze theoretically/empirically

Simple SLS methods

- Random Search (Blind Guessing):
- In each step, randomly select one element of the search space.
- (Uninformed) RandomWalk:
- In each step, randomly select one of the neighbouring positions of the search space and move there.
Random restart hill climbing

\[ f-value = \text{evaluation(state)} \]

Randomized Iterative Improvement:
- initialize search at some point of search space search steps:
- with probability \( p \), move from current search position to a randomly selected neighboring position
- otherwise, move from current search position to neighboring position with better evaluation function value.
- Has many variations of how to choose the randomly neighbor, and how many of them
- Example: Take 100 steps in one direction (Army mistake correction) – to escape from local optima.

Search space
- Problem: depending on initial state, can get stuck in local maxima

General iterative Algorithms
- general and “easy” to implement
- approximation algorithms
- must be told when to stop
- hill-climbing
- convergence

General iterative search
- Algorithm
  - Initialize parameters and data structures
  - construct initial solution(s)
  - Repeat
    - Repeat
      - Generate new solution(s)
      - Select solution(s)
    - Until time to adapt parameters
    - Update parameters
  - Until time to stop
- End

Iterative search
- Most popular algorithms of this class
  - Genetic Algorithms
    - Probabilistic algorithm inspired by evolutionary mechanisms
  - Simulated Annealing
    - Probabilistic algorithm inspired by the annealing of metals
  - Tabu Search
    - Meta-heuristic which is a generalization of local search
Hill-climbing search

- Problem: depending on initial state, can get stuck in local maxima

Simulated annealing

Combinatorial search technique inspired by the physical process of annealing [Kirkpatrick et al. 1983, Cerny 1985]

Outline

- Select a neighbor at random.
- If better than current state go there.
- Otherwise, go there with some probability.
- Probability goes down with time (similar to temperature cooling)

Simulated annealing

Acceptance criterion

- Metropolis acceptance criterion
  - better solutions are always accepted
  - worse solutions are accepted with probability

\[ e^{\Delta E/T} \sim \exp \left( \frac{E(s) - E(s')}{T} \right) \]

Annealing

- parameter \( T \), called temperature, is slowly decreased

Generic choices for annealing schedule

- initial temperature \( T_0 \)
  - (example: based on statistics of evaluation function)
- cooling schedule — how to change temperature over time
  - (example: geometric cooling, \( T_{n+1} = \alpha \cdot T_n, n = 0, 1, \ldots \))
- number of iterations at each temperature
  - (example: multiple of the neighbourhood size)
- stopping criterion
  - (example: no improved solution found for a number of temperature values)
**Pseudo code**

```
function Simulated-Annealing(problem, schedule) returns solution state
    current ← Make-Node(Initial-State[problem])
    for t ← 1 to infinity
        T ← schedule[t]  // T goes downwards.
        if T = 0 then return current
        next ← Random-Successor(current)
        ΔE ← f-Value[next] - f-Value[current]
        if ΔE > 0 then current ← next
        else current ← next with probability e(ΔE/T)
    end
```

---

**Example application to the TSP** [Johnson & McGeoch 1997]

baseline implementation:
- start with random initial solution
- use 2-exchange neighborhood
- simple annealing schedule;
  - relatively poor performance
improvements:
- look-up table for acceptance probabilities
- neighborhood pruning
- low-temperature starts

---

**Summary-Simulated Annealing**

**Simulated Annealing . . .**
- is historically important
- is easy to implement
- has interesting theoretical properties (convergence), but these are of very limited practical relevance
- achieves good performance often at the cost of substantial run-times

---

**Examples for combinatorial problems:**

- finding shortest/cheapest round trips (TSP)
- finding models of propositional formulae (SAT)
- planning, scheduling, time-tabling
- resource allocation
- protein structure prediction
- genome sequence assembly

---

**SAT**

**SAT Problem – decision variant:**
For a given propositional formula \( \Phi \),
decide whether \( \Phi \) has at least one model.

**SAT Problem – search variant:**
For a given propositional formula \( \Phi \), if \( \Phi \) is satisfiable,
find a model, otherwise declare \( \Phi \) unsatisfiable.

---

**The Propositional Satisfiability Problem (SAT)**

**Simple SAT instance (in CNF):**

\[
(a \lor b) \land (\neg a \lor \neg b)
\]

\(\sim\) satisfiable, two models:

\[
\begin{align*}
a &= \text{true}, b &= \text{false} \\
a &= \text{false}, b &= \text{true}
\end{align*}
\]
## Tabu Search

- Combinatorial search technique which heavily relies on the use of an explicit memory of the search process [Glover 1989, 1990] to guide search process
- Memory typically contains only specific attributes of previously seen solutions
- Simple tabu search strategies exploit only short term memory
- More complex tabu search strategies exploit long term memory

### Tabu search – exploiting short term memory

- In each step, move to best neighboring solution although it may be worse than current one
- To avoid cycles, tabu search tries to avoid revisiting previously seen solutions by basing the memory on attributes of recently seen solutions
- Tabu list stores attributes of the \( t_l \) most recently visited
- Solutions; parameter \( t_l \) is called tabu list length or tabu tenure
- Solutions which contain tabu attributes are forbidden

## Example: Tabu Search for SAT / MAX-SAT

- **Neighborhood**: assignments which differ in exactly one variable instantiation
- **Tabu attributes**: variables
- **Tabu criterion**: flipping a variable is forbidden for a given number of iterations
- **Aspiration criterion**: if flipping a tabu variable leads to a better solution, the variable’s tabu status is overridden

[Hansen & Jaumard 1990; Selman & Kautz 1994]

## Fundamental challenge: Combinatorial Search Spaces

### Significant progress in the last decade.

### How much?

- For propositional reasoning:
  - We went from 100 variables, 200 clauses (early 90s)
  - to 1,000,000 vars. and 5,000,000 constraints in
  - 10 years. Search space: from \( 10^{100} \) to \( 10^{300,000} \).

- Applications: Hardware and Software Verification,
  - Test pattern generation, Planning, Protocol Design,
  - Routers, Timetabling, E-Commerce (combinatorial auctions), etc.

- How can deal with such large combinatorial spaces and still do a decent job?
- I’ll discuss recent formal insights into
  - combinatorial search spaces and their practical implications that makes searching such ultra-large spaces possible.
  - Brings together ideas from physics of disordered systems (spin glasses), combinatorics of random structures, and algorithms.
  - But first, what is BIG?
I.e., ((not x_1) or x_7)
((not x_1) or x_6)
etc.

What is BIG?
Consider a real-world Boolean Satisfiability (SAT) problem

The instance bcl-la-6.cscf. DIMACS 1997:

\[
\begin{align*}
\phi & = ( \neg x_0 \lor x_2 ) \land ( \neg x_1 \lor x_7 ) \\
& \quad \land ( \neg x_1 \lor x_6 ) \\
& \quad \land ( \neg x_1 \lor x_1 ) \\
& \quad \land ( \neg x_1 \lor x_9 ) \\
& \quad \land ( \neg x_1 \lor x_{10} ) \\
& \quad \land ( \neg x_1 \lor x_{11} ) \\
& \quad \land ( \neg x_1 \lor x_{12} ) \\
\end{align*}
\]

Set x_1 to False ??

10 pages later:

\[
\begin{align*}
& i.e., (x_{177} \lor x_{169} \lor x_{161} \lor x_{153} \ldots \\
& \quad \lor x_{33} \lor x_{25} \lor x_{17} \lor x_9 \lor x_1 \lor (\not x_{185}))
\end{align*}
\]

clauses / constraints are getting more interesting…

Note x_{17} …

4000 pages later:

\[
\begin{align*}
10736 & \to 10550 0 \\
10236 & \to 10551 0 \\
10326 & \to 10236 0 \\
10098 & \to 10009 010101 101013 101014 \\
13035 & \to 10010 101014 101010 101020 10025 \\
10082 & \to 10502 101020 10003 10010 101027 10027 1005 \\
10295 & \to 10502 101013 101001 100106 100110 \\
10306 & \to 10502 100106 100107 100009 100309 100309 \\
10091 & \to 10502 100106 100106 100030 100309 100309 \\
10098 & \to 10091 100101 101010 101011 101001 \\
10105 & \to 10502 101011 101006 101050 10050 101036 \\
10474 & \to 10502 101004 100490 10050 101036 100326 100236 0 \\
10037 & \to 10050 0 \\
10707 & \to 10001 0 \\
10257 & \to 10010 0 \\
\end{align*}
\]

Finally, 15,000 pages later:

\[
\begin{align*}
& \land (\not x_{185}) \\
\end{align*}
\]

Combinatorial search space of truth assignments:

\[
\begin{align*}
\text{HOW?}
\end{align*}
\]

Current SAT solvers solve this instance in approx. 1 minute!

Progress SAT Solvers

<table>
<thead>
<tr>
<th>Instance</th>
<th>Posit'94</th>
<th>Grasp'96</th>
<th>Sato'98</th>
<th>Chaff'01</th>
</tr>
</thead>
<tbody>
<tr>
<td>ssa2670-136</td>
<td>40.68s</td>
<td>1.2s</td>
<td>0.95s</td>
<td>0.02s</td>
</tr>
<tr>
<td>bf1355-638</td>
<td>1805.21s</td>
<td>0.11s</td>
<td>0.04s</td>
<td>0.01s</td>
</tr>
<tr>
<td>pret150_25</td>
<td>&gt;3000s</td>
<td>0.21s</td>
<td>0.09s</td>
<td>0.01s</td>
</tr>
<tr>
<td>dubois100</td>
<td>&gt;3000s</td>
<td>11.85s</td>
<td>0.08s</td>
<td>0.01s</td>
</tr>
<tr>
<td>aim200-2_no-1</td>
<td>&gt;3000s</td>
<td>0.01s</td>
<td>0s</td>
<td>0s</td>
</tr>
<tr>
<td>2dx___bug005</td>
<td>&gt;3000s</td>
<td>&gt;3000s</td>
<td>&gt;3000s</td>
<td>2.9s</td>
</tr>
<tr>
<td>c6288</td>
<td>&gt;3000s</td>
<td>&gt;3000s</td>
<td>&gt;3000s</td>
<td>&gt;3000s</td>
</tr>
</tbody>
</table>

Source: Marques Silva 2002

- From academically interesting to practically relevant.
- We now have regular SAT solver competitions.
  - Germany ’99, Dimacs ’93, China ’96, SAT-02, SAT-03, SAT-04, SAT-05.
- E.g. at SAT-2004 (Vancouver, May 04):
  - ~ 35+ solvers submitted
  - ~ 500+ industrial benchmarks
  - ~ 50,000+ instances available on the WWW.
The Genetic Algorithm

- Directed search algorithms based on the mechanics of biological evolution
- Developed by John Holland, University of Michigan (1970's)
  - To understand the adaptive processes of natural systems
  - To design artificial systems software that retains the robustness of natural systems

Components of a GA

A problem to solve, and ...
- Encoding technique (gene, chromosome)
- Initialization procedure (creation)
- Evaluation function (environment)
- Selection of parents (reproduction)
- Genetic operators (mutation, recombination)
- Parameter settings (practice and art)

Simple Genetic Algorithm

```plaintext
{ initialize population; evaluate population; while TerminationCriteriaNotSatisfied { select parents for reproduction; perform recombination and mutation; evaluate population; } }
```
The GA Cycle of Reproduction

Genetic algorithms

- How to generate the next generation.
  - 1) Selection: we select a number of states from the current generation. (we can use the fitness function in any reasonable way)
  - 2) crossover: select 2 states and reproduce a child.
  - 3) mutation: change some of the genes.

Example

8-queen example

Summary: Genetic Algorithms

Genetic Algorithms
- use populations, which leads to increased search space exploration
- allow for a large number of different implementation choices
- typically reach best performance when using operators that are based on problem characteristics
- achieve good performance on a wide range of problems

Classes of Search Techniques
**Example application: evolving checkers players (Fogel’02)**

- Neural nets for evaluating future values of moves are evolved
- NNs have fixed structure with 5046 weights, these are evolved + one weight for “kings”
- Representation:
  - vector of 5046 real numbers for object variables (weights)
  - vector of 5046 real numbers for $\sigma$’s
- Mutation:
  - Gaussian, lognormal scheme with $\sigma$-first
  - Plus special mechanism for the kings’ weight
- Population size 15

**Example application: evolving checkers players (Fogel’02)**

- Tournament size $q = 5$
- Programs (with NN inside) play against other programs, no human trainer or hard-wired intelligence
- After 840 generation (6 months!) best strategy was tested against humans via Internet
- Program earned “expert class” ranking outperforming 99.61% of all rated players

---

**The GA Cycle of Reproduction**

![DAG of the GA cycle](image)

- **parents** → **children** → **modified children**
- **population** → **evaluation** → **evaluated children** → **discard**
- **reproduction** → **modification**

**Population**

- Chromosomes could be:
  - Bit strings: (0101 ... 1100)
  - Real numbers: (43.2 -33.1 ... 0.0 89.2)
  - Permutations of element: (E11 E3 E7 ... E1 E15)
  - Lists of rules: (R1 R2 R3 ... R22 R23)
  - Program elements: (genetic programming)
  - ... any data structure ...

---

**Reproduction**

- **parents** → **children**
- Parents are selected at random with selection chances biased in relation to chromosome evaluations.

**Chromosome Modification**

- **children** → **modified children**
- Modifications are stochastically triggered
- Operator types are:
  - Mutation
  - Crossover (recombination)
Mutation: Local Modification

Before: (1 0 1 1 0 1 1 0)

After: (0 1 1 0 1 1 0)

Before: (1.38 -69.4 326.44 0.1)

After: (1.38 -67.5 326.44 0.1)

- Causes movement in the search space (local or global)
- Restores lost information to the population

Crossover: Recombination

P1 (0 1 1 0 0 1 1 0)            (0 1 0 1 0 0 0)    C1
P2 (1 1 1 0 1 1 0 1)            (1 1 1 1 1 0 1 0)    C2

Crossover is a critical feature of genetic algorithms:
- It greatly accelerates search early in evolution of a population
- It leads to effective combination of schemata (subsolutions on different chromosomes)

Evaluation

- The evaluator decodes a chromosome and assigns it a fitness measure
- The evaluator is the only link between a classical GA and the problem it is solving

Deletion

- Generational GA: entire populations replaced with each iteration
- Steady-state GA: a few members replaced each generation

An Abstract Example

Distribution of Individuals in Generation 0

Distribution of Individuals in Generation N

A Simple Example

"The Gene is by far the most sophisticated program around."

- Bill Gates, Business Week, June 27, 1994
A Simple Example

The Traveling Salesman Problem:

Find a tour of a given set of cities so that
- each city is visited only once
- the total distance traveled is minimized

Representation

Representation is an ordered list of city numbers known as an order-based GA.

1) London 3) Dunedin 5) Beijing 7) Tokyo
2) Venice 4) Singapore 6) Phoenix 8) Victoria

CityList1 (3 5 7 2 1 6 4 8)
CityList2 (2 5 7 6 8 1 3 4)

Crossover

Crossover combines inversion and recombination:

Parent1 (3 5 7 2 1 6 4 8)
Parent2 (2 5 7 6 8 1 3 4)
Child (5 8 7 2 1 6 3 4)

This operator is called the *Order1* crossover.

Mutation

Mutation involves reordering of the list:

Before: (5 8 7 2 1 6 3 4)
After: (5 8 6 2 1 7 3 4)

TSP Example: 30 Cities

Solution i (Distance = 941)
Considering the GA Technology

“Almost eight years ago ... people at Microsoft wrote a program [that] uses some genetic things for finding short code sequences. Windows 2.0 and 3.2, NT, and almost all Microsoft applications products have shipped with pieces of code created by that system.”

-- Nathan Myhrvold, Microsoft Advanced Technology Group, Wired, September 1995

Issues for GA Practitioners

- Choosing basic implementation issues:
  - representation
  - population size, mutation rate, ...
  - selection, deletion policies
  - crossover, mutation operators
- Termination Criteria
- Performance, scalability
- Solution is only as good as the evaluation function (often hardest part)
Benefits of Genetic Algorithms

- Concept is easy to understand
- Modular, separate from application
- Supports multi-objective optimization
- Good for “noisy” environments
- Always an answer; answer gets better with time
- Inherently parallel; easily distributed

Benefits of Genetic Algorithms (cont.)

- Many ways to speed up and improve a GA-based application as knowledge about problem domain is gained
- Easy to exploit previous or alternate solutions
- Flexible building blocks for hybrid applications
- Substantial history and range of use

When to Use a GA

- Alternate solutions are too slow or overly complicated
- Need an exploratory tool to examine new approaches
- Problem is similar to one that has already been successfully solved by using a GA
- Want to hybridize with an existing solution
- Benefits of the GA technology meet key problem requirements

Some GA Application Types

<table>
<thead>
<tr>
<th>Domain</th>
<th>Application Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control</td>
<td>gain pipeline, pole balancing, missile evasion, pursuit</td>
</tr>
<tr>
<td>Design</td>
<td>semiconductor layout, aircraft design, keyboard configuration, communication networks</td>
</tr>
<tr>
<td>Scheduling</td>
<td>manufacturing, facility scheduling, resource allocation</td>
</tr>
<tr>
<td>Robotics</td>
<td>trajectory planning</td>
</tr>
<tr>
<td>Machine Learning</td>
<td>designing neural networks, improving classification algorithms, classifier systems</td>
</tr>
<tr>
<td>Signal Processing</td>
<td>filter design</td>
</tr>
<tr>
<td>Game Playing</td>
<td>poker, checkers, prisoner’s dilemma</td>
</tr>
<tr>
<td>Combinatorial</td>
<td>set covering, travelling salesman, routing, bin packing, graph colouring and partitioning</td>
</tr>
<tr>
<td>Optimization</td>
<td></td>
</tr>
</tbody>
</table>

Review

4 main types of Evolutionary Algorithms

- Genetic Algorithm - John Holland
- Genetic Programming - John Koza
- Evolutionary Programming - Lawerence Fogel
- Evolutionary Strategies - Ingo Rechenberg

Genetic Algorithms

- Most widely used
- Robust
  - uses 2 separate spaces
    - search space - coded solution (genotype)
    - solution space - actual solutions (phenotypes)
- Genotypes must be mapped to phenotypes before the quality or fitness of each solution can be evaluated
Genetic Programming

- Specialized form of GA
- Manipulates a very specific type of solution using modified genetic operators
- Original application was to design computer program
- Now applied in alternative areas eg. Analog Circuits
- Does not make distinction between search and solution space.
- Solution represented in very specific hierarchical manner

Evolutionary Strategies

- Like GP no distinction between search and solution space
- Individuals are represented as real-valued vectors.
  - Simple ES
    - one parent and one child
    - Child solution generated by randomly mutating the problem parameters of the parent.
  - Susceptible to stagnation at local optima

Evolutionary Strategies (cont’d)

- Slow to converge to optimal solution
- More advanced ES
  - have pools of parents and children
- Unlike GA and GP, ES
  - Separates parent individuals from child individuals
  - Selects its parent solutions deterministically

Evolutionary Programming

- Resembles ES, developed independently
- Early versions of EP applied to the evolution of transition table of finite state machines
- One population of solutions, reproduction is by mutation only
- Like ES operates on the decision variable of the problem directly (ie Genotype = Phenotype)
- Tournament selection of parents
  - better fitness more likely a parent
  - children generated until population doubled in size
  - everyone evaluated and the half of population with lowest fitness deleted.

General Idea of Evolutionary Algorithms

- Evolves more complex structures - programs, Lisp code, neural networks
- Start with random programs of functions and terminals (data structures)
- Execute programs and give each a fitness measure
- Use crossover to create new programs, no mutation
- Keep best programs
- For example, place lisp code in a tree structure, functions at internal nodes, terminals at leaves, and do crossover at sub-trees - always legal in Lisp

Figure 1: The general iterations of evolutionary Algorithms (GA/ES).
Evolutionary design

• Karl Sims Evolved Virtual Creatures (1994)
  - http://www.youtube.com/watch?v=F00hycpeG5B
  - http://video.google.com/videoplay?docid=7219479512410540649#
  - course work - 2005
• http://vimeo.com/7074089
Figures 8.5. A frame of nine mutations. The parent is in the center surrounded by offspring.

Figures 8.5a. A subset view of the same frame showing the different mutation rates.

Figures 8.6. A more detailed view of the evolutionary process, showing the progression from parent to offspring.

Figures 8.6a. Extract from an evolutionary tree. The tree has become too large to display clearly, so the artist has restricted the display to include only frames between one level above and one level below the current frame. Cousin frames are not displayed.
Conclusions

Question: ‘If GAs are so smart, why ain’t they rich?’
Answer: ‘Genetic algorithms are rich - rich in application across a large and growing number of disciplines.’

- David E. Goldberg, Genetic Algorithms in Search, Optimization and Machine Learning

Games: Spore (2007)