Advanced Algorithmics (6EAP)
Search and meta-heuristics

Jaak Vilo
2011 Spring
Search

• **for what?**
  – a solution
  – the (best possible (approximate?)) solution

• **from where?**
  – search space (all valid solutions or paths)

• **under which conditions?**
  – compute time, space, ...
  – constraints, ...
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):

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

\(\not\rightarrow\) satisfiable, two models:

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

\[a = \text{false}, \ b = \text{true}\]
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
An Instance of the Traveling Salesman Problem

Cost of Nearest Neighbor Path, AEDBCA = 550
1 Introduction

Figure 1.2: Global and local optima of a two-dimensional function.
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
# Types of games

<table>
<thead>
<tr>
<th></th>
<th>Deterministic</th>
<th>Chance</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Perfect information</strong></td>
<td>chess, checkers, go, othello</td>
<td>backgammon monopoly</td>
</tr>
<tr>
<td><strong>Imperfect information</strong></td>
<td>battleships, blind tictactoe</td>
<td>bridge, poker, scrabble nuclear war</td>
</tr>
</tbody>
</table>
An interesting constrained numerical optimization test case emerged recently; the problem (Keane, 1994) is to maximize a function:

\[ G2(\bar{x}) = \left| \frac{\sum_{i=1}^{n} \cos^4(x_i) - 2 \prod_{i=1}^{n} \cos^2(x_i)}{\sqrt{\sum_{i=1}^{n} x_i^2}} \right|, \]

subject to

\[ \prod_{i=1}^{n} x_i \geq 0.75, \quad \sum_{i=1}^{n} x_i \leq 7.5n, \quad \text{and bounds } 0 \leq x_i \leq 10 \text{ for } 1 \leq i \leq n. \]

Function \( G2 \) is nonlinear and its global maximum is unknown, lying somewhere near the origin. The problem has one nonlinear constraint and one linear constraint; the latter one is inactive around the origin and will be forgotten in the following.

\[ G2(x) = (\Sigma \cos^4(x_i) - 2 \prod \cos^2(x_i))/\sqrt{\Sigma i \, x_i^2}, \]

where \( 0 \leq x_i \leq 10 \) and

\[ \prod x_i \geq 0.75 \]
The graph of function $G^2$ for $n = 2$. Infeasible solutions were as
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 \(X\)

\[ X = \bigcup_{S \in F} S \]

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

\[ X = \bigcup_{S \in C} S \]

source: 91.503 textbook Cormen et al.
Greedy Set Covering Algorithm

Greedy-Set-Cover$(X, \mathcal{F})$

1. $U \leftarrow X$
2. $\mathcal{C} \leftarrow \emptyset$
3. while $U \neq \emptyset$
4. do select an $S \in \mathcal{F}$ that maximizes $|S \cap U|$
5. $U \leftarrow U - S$
6. $\mathcal{C} \leftarrow \mathcal{C} \cup \{S\}$
7. return $\mathcal{C}$

Greedy: select set that covers the most uncovered elements

source: 91.503 textbook Cormen et al.
Set Cover

**Theorem:** \textsc{Greedy-Set-Cover} is a polynomial-time $\rho(n)$-approximation algorithm for

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

**Proof:**

The $d$th harmonic number $H_d = \sum_{i=1}^{d} \frac{1}{i} = H(d)$, $H(0) = 0$

Algorithm runs in time polynomial in $n$.

$S_i = \text{ith subset selected}$ \hspace{10mm} selecting $S_i$ costs 1

$c_x = \text{cost of element } x \in X$ \hspace{10mm} paid only when $x$ is covered for the first time

$$c_x = \frac{1}{|S_i - (S_1 \cup S_2 \cup \ldots \cup S_{i-1})|}$$

assume $x$ is covered for the first time by $S_i$

(spread cost evenly across all elements covered for first time by $S_i$)

Number of elements covered for first time by $S_i$
Set Cover (proof continued)

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

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

**Proof:** (continued)

Let \( C^* \) be an optimal cover

\[ C \] be cover from GREEDY - SET - COVER

The cost assigned to optimal cover:

\[
\sum_{S \in C^*} \left( \sum_{x \in S} c_x \right)
\]

1 unit is charged at each stage of algorithm

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

Each \( x \) is in \( \geq 1 \) \( S \) in \( C^* \)

\[
\sum_{S \in C^*} \left( \sum_{x \in S} c_x \right) \geq \sum_{x \in X} c_x
\]

\[
|C| \leq \sum_{S \in C^*} \left( \sum_{x \in S} c_x \right)
\]
Set Cover (proof continued)

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

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

**Proof:** (continued)

How does this relate to harmonic numbers??

The $d$th harmonic number $H_d = \sum_{i=1}^{d} \frac{1}{i} = H(d)$

We'll show that:

\[ \sum_{x \in S} c_x \leq H(|S|) \quad \text{for any set } S \in \mathcal{F} \]

And then conclude that:

\[ |C| \leq \sum_{S \in \mathcal{C}^*} H(|S|) \leq |C^*|H(\max\{|S| : S \in \mathcal{F}\}) \]
Classes of Search Techniques

- Search techniques
  - Calculus-based techniques
    - Direct methods
      - Fibonacci
    - Indirect methods
      - Newton
  - Guided random search techniques
    - Evolutionary algorithms
    - Simulated annealing
  - Enumerative techniques
    - Dynamic programming
- Evolutionary strategies
  - Genetic algorithms
    - Parallel
      - Centralized
      - Distributed
    - Sequential
      - Steady-state
      - Generational
Local Search
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

• 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
  – $A^*$ search
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: $h_{SLD}(n)$ (never overestimates the actual road distance) (SLD – shortest linear distance)

- **Theorem**: If $h(n)$ is admissible, A* using TREE-SEARCH is optimal
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
- $f(G) = g(G)$ 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 $G$.

\[f(G_2) = g(G_2)\quad\text{since } h(G_2) = 0\]
\[g(G_2) > g(G)\quad\text{since } G_2 \text{ is suboptimal}\]
\[f(G) = g(G)\quad\text{since } h(G) = 0\]
\[f(G_2) > f(G)\quad\text{from above}\]
\[\text{Hence } f(G_2) > f(n), \text{ and A* will never select } G_2 \text{ for expansion}\]
A* path-finder
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
• How many are misplaced? (7)

• How far have they been misplaced? Sum of theoretical shortest paths to the correct place

• A* search towards a final goal
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** $g : S \to \mathbb{R}^+$
  (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 on partial 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 $\pi(1)$
  - at the $i$th step choose city $\pi(i + 1)$ to be the city $j$ that minimises $d(\pi(i), j); j \neq \pi(k), 1 \leq k \leq i$

- running time: $O(n^2)$

- worst case performance:
  \[ \frac{NN(x)}{OPT(x)} \leq 0.5(\lceil \log_2 n \rceil + 1) \]

- other construction heuristics for TSP are available
Nearest neighbor tour through 532 US cities
**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.
local search on complete solutions
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 iff 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

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

\[
\text{while } f\text{-value}(\text{state}) > f\text{-value}(\text{next-best}(\text{state}))
\]

\[
\text{state} := \text{next-best}(\text{state})
\]
Hill climbing

**function** Hill-Climbing(*problem*) **returns** a solution state

`current ← Make-Node(Initial-State[*problem*])`

**loop do**

`next ← a highest-valued successor of current`

**if** Value[*next*] < Value[*current*] **then** return *current*

`current ← next`

**end**
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

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

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

Annealing

- parameter $T$, called temperature, is slowly decreased
\[ \delta = -10 \]

\[ T = 0.1 \text{ to } 10,000 \]
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)
Simulated Annealing


Fig. 9. Results at four temperatures for a clustered 400-city traveling salesman problem. The points are uniformly distributed in nine regions. (a) $T = 1.2$, $\alpha = 2.0567$; (b) $T = 0.8$, $\alpha = 1.515$; (c) $T = 0.4$, $\alpha = 1.055$; (d) $T = 0.0$, $\alpha = 0.7839$. 
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)

    $\Delta E ← f$-Value[next] - f-Value[current]

    if $\Delta E > 0$ then current ← next

    else current ← next with probability $e^{\Delta E/T}$

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
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:

\[a = \text{true}, b = \text{false}\]
\[a = \text{false}, b = \text{true}\]
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 *tl* most recently visited
• solutions; parameter *tl* 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]
• Bart Selman, Cornell

www-verimag.imag.fr/~maler/TCC/selman-tcc.ppt

Ideas from physics, statistics, combinatorics, algorithmics ...
Fundamental challenge: Combinatorial Search Spaces

• Significant progress in the last decade.

• How much?
  
  • For propositional reasoning:
  • -- We went from 100 variables, 200 clauses (early 90’s)
  • to 1,000,000 vars. and 5,000,000 constraints in
  • 10 years. Search space: from $10^{30}$ 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 combinational 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?
What is BIG?

Consider a real-world Boolean Satisfiability (SAT) problem

The instance bmc-ibm-6.cnf, IBM LSU 1997:

\[
\begin{align*}
\text{p cnf} & \\
& -1 7 0 \\
& -1 6 0 \\
& -1 5 0 \\
& -1 -4 0 \\
& -1 3 0 \\
& -1 2 0 \\
& -1 -8 0 \\
& -9 15 0 \\
& -9 14 0 \\
& -9 13 0 \\
& -9 -12 0 \\
& -9 11 0 \\
& -9 10 0 \\
& -9 -16 0 \\
& -17 23 0 \\
& -17 22 0 \\
\end{align*}
\]

i.e., \((\neg x_1) \lor x_7\)
\((\neg x_1) \lor x_6\)
etc.

\(x_1, x_2, x_3, \text{etc. our Boolean variables (set to True or False)}\)

Set \(x_1\) to False ??
10 pages later:

\[
\begin{align*}
185 &= 0 \\
185 &= 1 \\
177 &= 169 \\&= 161 \\&= 153 \\&= 145 \\&= 137 \\&= 129 \\&= 121 \\&= 113 \\&= 105 \\&= 97 \\
89 &= 81 \\&= 65 \\&= 57 \\&= 49 \\&= 41 \\
33 &= 25 \\&= 17 \\&= 9 \\&= 1 \\&= -185 \\
186 &= -187 \\
186 &= -188 \\
\end{align*}
\]

... \[\text{i.e., (x}_{177} \text{ or } x_{169} \text{ or } x_{161} \text{ or } x_{153} \ldots \]
\[x_{33} \text{ or } x_{25} \text{ or } x_{17} \text{ or } x_{9} \text{ or } x_{1} \text{ or } (\text{not } x_{185})\]

clauses / constraints are getting more interesting...

*Note* \(x_{1}\) ...
4000 pages later:

10236  -10050  0
10236  -10051  0
10236  -10235  0
10008  10009  10010  10011  10012  10013  10014
  10015  10016  10017  10018  10019  10020  10021
  10022  10023  10024  10025  10026  10027  10028
  10029  10030  10031  10032  10033  10034  10035
  10036  10037  10038  10039  10040  10041  10042
  10043  10044  10045  10046  10047  10048  10049
  10050  10051  10235  -10236  0
10237  -10008  0
10237  -10009  0
10237  -10010  0

...
Finally, 15,000 pages later:

\[
\begin{align*}
-7 & 260 0 \\
7 & -260 0 \\
1072 & 1070 0 \\
-15 & -14 -13 -12 -11 -10 0 \\
-15 & -14 -13 -12 -11 10 0 \\
-15 & -14 -13 -12 11 -10 0 \\
-15 & -14 -13 -12 11 10 0 \\
-7 & -6 -5 -4 -3 -2 0 \\
-7 & -6 -5 -4 -3 2 0 \\
-7 & -6 -5 -4 3 -2 0 \\
-7 & -6 -5 -4 3 2 0 \\
185 & 0
\end{align*}
\]

Combinatorial search space of truth assignments: \(2^{50000} \approx 3.160699437 \cdot 10^{15051}\)

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>40,66s</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_0-no-1</td>
<td>&gt;3000s</td>
<td>0,01s</td>
<td>0s</td>
<td>0s</td>
</tr>
<tr>
<td>2dlx__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 ’89, Dimacs ’93, China ’96, SAT-02, SAT-03, SAT-04, SAT05.

  • E.g. at SAT-2004 (Vancouver, May 04):
    • --- 35+ solvers submitted
    • --- 500+ industrial benchmarks
    • --- 50,000+ instances available on the WWW.
Real-World Reasoning
Tackling inherent computational complexity

Example domains cast in propositional reasoning system (variables, rules).

- High-Performance Reasoning
- Temporal/ uncertainty reasoning
- Strategic reasoning/Multi-player

No. of atoms on earth $10^{47}$
Seconds until heat death of sun
Protein folding calculation (petaflop-year)

Technology Targets
- DARPA Research Program
- Multi-Agent Systems
- Hardware/Software Verification
- Military Logistics
- Deep space mission control
- Car repair diagnosis

Variables

Rules (Constraints)
“Genetic Algorithms are good at taking large, potentially huge search spaces and navigating them, looking for optimal combinations of things, solutions you might not otherwise find in a lifetime.”

- Salvatore Mangano

*Computer Design*, May 1995
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
The Genetic Algorithm (cont.)

- Provide efficient, effective techniques for optimization and machine learning applications
- Widely-used today in business, scientific and engineering circles
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

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

- **reproduction** → **children** → **modification**
  - **parents** → **population** → **evaluated children** → **evaluation**
  - **deleted members** → **discard** → **modified children**
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 genues.
Example

= stochastic local beam search + generate successors from pairs of states
8-queen example

GAs require states encoded as strings (GPs use programs)

Crossover helps iff substrings are meaningful components
Summary: Genetic Algorithms

Gene0c

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

- Calculus-based techniques
  - Direct methods
    - Fibonacci
  - Indirect methods
    - Newton
- Guided random search techniques
- Evolutionary algorithms
- Genetic algorithms
  - Parallel
    - Centralized
    - Distributed
  - Sequential
    - Steady-state
    - Generational
- Enumerative techniques

Guided random search techniques:
- Simulated annealing
- Dynamic programming

Evolutionary strategies:
- Evolutionary algorithms
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 σ’s
- Mutation:
  - Gaussian, lognormal scheme with σ-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

- **Reproduction**: Parents → Children
- **Modification**: Evaluated Children → Modified Children
- **Evaluation**: Modified Children
- **Population**: Parents, Evaluated Children, Deleted Members
- **Discard**: Deleted Members
Population

Chromosomes could be:

- Bit strings \((0101 \ldots 1100)\)
- Real numbers \((43.2 \ -33.1 \ldots 0.0 \ 89.2)\)
- Permutations of element \((E11 \ E3 \ E7 \ldots E1 \ E15)\)
- Lists of rules \((R1 \ R2 \ R3 \ldots R22 \ R23)\)
- Program elements \((\text{genetic programming})\)
- ... any data structure ...
Reproduction

Parents are selected at random with selection chances biased in relation to chromosome evaluations.
Chromosome Modification

- Modifications are stochastically triggered
- Operator types are:
  - Mutation
  - Crossover (recombination)
Mutation: Local Modification

Before: \((1 \ 0 \ 1 \ 1 \ 0 \ 1 \ 1 \ 0)\)
After: \((0 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1 \ 0)\)

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 1 0 0 0) → (0 1 0 0 1 0 0 0) C1
P2: (1 1 0 1 1 0 1 0) → (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 (subolutions 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:

\[
\begin{array}{cccccccc}
\ast & \ast \\
\text{Parent1} & (3 & 5 & 7 & 2 & 1 & 6 & 4 & 8) \\
\text{Parent2} & (2 & 5 & 7 & 6 & 8 & 1 & 3 & 4) \\
\hline
\text{Child} & (5 & 8 & 7 & 2 & 1 & 6 & 3 & 4)
\end{array}
\]

This operator is called the \textit{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)
Solution (Distance = 800)
Solution \( k \) (Distance = 652)

TSP30 (Performance = 652)
Best Solution (Distance = 420)
Overview of Performance

TSP30 - Overview of Performance

- Best
- Worst
- Average

Generations (1000)
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>gas 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 Optimization</td>
<td>set covering, travelling salesman, routing, bin packing, graph colouring and partitioning</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 (i.e., 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
Genetic Programming

- 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
## Summary

<table>
<thead>
<tr>
<th></th>
<th>ES</th>
<th>EP</th>
<th>GA</th>
<th>GP</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Representation</strong></td>
<td>Real-valued</td>
<td>Real-valued</td>
<td>Binary-Valued</td>
<td>Lisp S-expressions</td>
</tr>
<tr>
<td><strong>Self-Adaptation</strong></td>
<td>Standard deviations and covariances</td>
<td>Variance</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td><strong>Fitness</strong></td>
<td>Objective function values</td>
<td>Scaled objective function value</td>
<td>Scaled objective function value</td>
<td>Scaled objective function value</td>
</tr>
<tr>
<td><strong>Mutation</strong></td>
<td>Main operator</td>
<td>Only operator</td>
<td>Background operator</td>
<td>Background operator</td>
</tr>
<tr>
<td><strong>Recombination</strong></td>
<td>Different variants, important for self-adaptation</td>
<td>None</td>
<td>Main Operator</td>
<td>Main Operator</td>
</tr>
<tr>
<td><strong>Selection</strong></td>
<td>Deterministic, extinctive</td>
<td>Probabilistic, extinctive</td>
<td>Probabilistic, preservative</td>
<td>Probabilistic, preservative</td>
</tr>
</tbody>
</table>
Evolutionary design

• Karl Sims Evolved Virtual Creatures (1994)
  – http://www.youtube.com/watch?v=F0OHycypSG8
  – http://video.google.com/videoplay?docid=7219479512410540649#
  – course work - 2005


• http://vimeo.com/7074089
EVOLUTIONARY DESIGN BY COMPUTERS
Edited by Peter J. Bentley

"DARWIN WOULD LOVE THIS BOOK"
RICHARD DAWKINS
Figure 9.1 *Mutator* keeps a bank of genes and their forms (generated by *Form Grow*), which it displays to the artist. Based on judgements made by the artist, *Mutator* generates and displays new forms, assisting the artist to search for interesting forms and bank the results.
structure expression:
  horn
  ribs (gene1)
  grow (gene2)
  stack (gene3)
  bend (gene4)
  twist (gene5)

corresponding gene vector:
< gene1, gene2, gene3, gene4, gene5 >

Figure 9.5 An example of a structure expression (created by the artist) and its corresponding gene vector (to be evolved by Mutator).
Figure 9.6 A frame of nine mutations. The parent is in the centre surrounded by offspring.
inbreeding

distant marriage
Figure 9.13 Spliced (left above), weighted average (left below) and dominant recessive (right)
Figure 9.16 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.
Figure 9.24 The forms layed out in a continuous Mutator session much as they would be in an animation such as the film ‘Mutations’.
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)

• http://www.gametrailers.com/user-movie/spore-14min-2007-demonstration/86368
• http://eu.spore.com/home.cfm?lang=en