Foundations of Artificial Intelligence

Local Search

CS472 – Fall 2007
Filip Radlinski

Scaling Up

- So far, we have considered methods that systematically explore the full search space, possibly using **principled** pruning (A* etc.).
- The current best such algorithms (RBFS / SMA*) can handle search spaces of up to 10^{100} states
  → ~ 500 binary valued variables.
- But search spaces for some real-world problems might be much bigger - e.g. $10^{30,000}$ states.
- Here, a completely different kind of search is needed.
  → *Local Search Methods*

Example

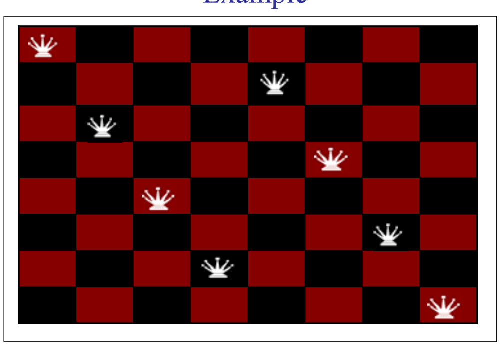
QuickTime™ and a
TIFF (LZW) decompressor
are needed to see this picture.

QuickTime™ and a
TIFF (LZW) decompressor
are needed to see this picture.

Optimization Problems

- We're interested in the Goal State - not in how to get there.
- Optimization Problem:
  - State: vector of variables
  - Objective Function: f: *state* → $\Re$
  - Goal: find state that maximizes or minimizes the objective function
- Examples: VLSI layout, job scheduling, map coloring, N-Queens.

Example



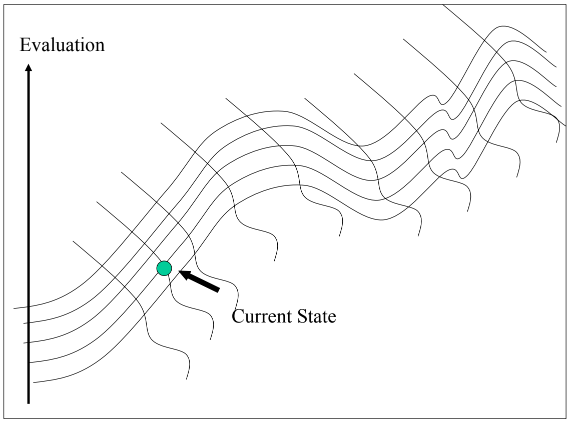
Local Search Methods

- Applicable to optimization problems.
- Basic idea:
  - use a single **current state**
  - don't save paths followed
  - generally move only to successors/neighbors of that state
- Generally require a **complete state description**.

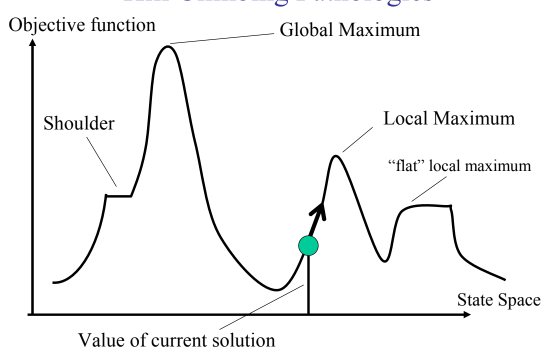
Hill-Climbing Search

function HILL-CLIMBING (***problem***) returns a solution state
  inputs: ***problem***, a problem
  static: ***current***, a node

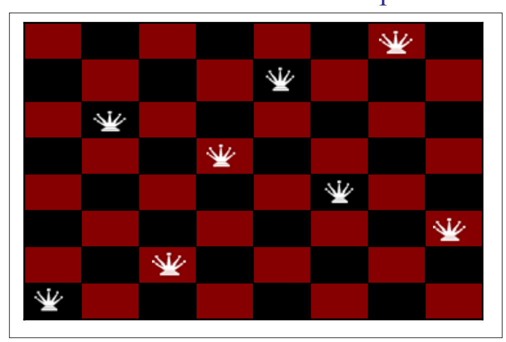
current ← MAKE-NODE(INITIAL-STATE[***problem***])
loop do
  next ← a highest-valued successor of ***current***
  if VALUE[next] < VALUE[current] then return ***current***
  current ← ***next***
end



Hill Climbing Pathologies



Local Maximum Example



Improvements to Basic Local Search

Issue: How to move more quickly to successively higher plateaus and avoid getting "stuck" **local maxima**.

Idea: Introduce downhill moves ("noise") to escape from long plateaus (or true local maxima).

Strategies:

- Random-restart hill-climbing
  => Multiple runs from randomly generated initial states
- Tabu search
- Simulated Annealing
- Genetic Algorithms

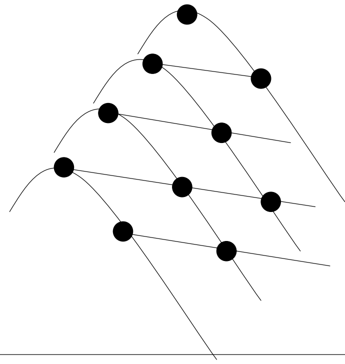
Variations on Hill-Climbing

Random restarts: Simply restart at a new random state after a pre-defined number of steps.

Local Beam Search: Run the random starting points in parallel, always keeping the k most promising states

current ← k initial states
for *t* ← 1 to infinity do
  new ← expand every state in *current*
  if *f(best-in-new)* < *f(best-in-current)* then
    return *best-in-current*
  current ← best k states in *new*

Ridges



Simulated Annealing

Idea:
Use conventional hill-climbing techniques, but occasionally take a step in a direction other than that in which the rate of change is maximal.

As time passes, the probability that a down-hill step is taken is gradually reduced and the size of any down-hill step taken is decreased.

Kirkpatrick *et al*. 1982; Metropolis *et al*.1953.

Simulated Annealing Algorithm

current ← initial state
for *t* ← 1 to infinity do

> *T* ← *schedule*[*t*]
> if *T* = 0 then return *current*
> *next* ← randomly selected successor of *current*
> $\Delta E \leftarrow f(next) - f(current)$
> if $\Delta E > 0$ then *current* ← *next*
> else *current* ← next only with probability $e^{\Delta E/T}$

Genetic Algorithms

- Approach mimics *evolution*.
- Usually presented using a rich (and different) vocabulary: fitness, populations, individuals, genes, crossover, mutations, etc.
- Still, can be viewed quite directly in terms of standard **local search**.

Genetic Algorithms

Inspired by biological processes that produce genetic change in populations of individuals.

Genetic algorithms (GAs) are local search procedures that usually the following basic elements:

- A Darwinian notion of **fitness**: the most fit individuals have the best chance of survival and reproduction.
- "Crossover" operators:
  - Parents are selected.
  - Parents pass their genetic material to children.
- Mutation: individuals are subject to random changes in their genetic material.

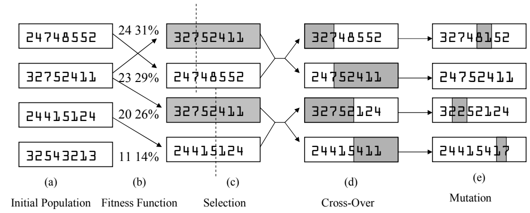
Features of Evolution

- High degree of parallelism (many individuals in a population)
- New individuals ("next state / neighboring states"):
  Derived by combining "parents" ("crossover operation")
  Random changes also happen ("mutations")
- Selection of next generation:
  Based on survival of the fittest: the most fit parents tend to be used to generate new individuals.

General Idea

- Maintain a population of individuals (states / strings / candidate solutions)
- Each individual is evaluated using a **fitness function**, i.e. an objective function. The fitness scores force individuals to compete for the privilege of survival and reproduction.
- Generate a sequence of generations:
  - From the current generation, select **pairs** of individuals (based on fitness) to generate new individuals, using **crossover**.
- Introduce some noise through random **mutations**.
- Hope that average and maximum fitness (i.e. value to be optimized) increases over time.

GA: High-level Algorithm



Genetic algorithms as search

- Genetic algorithms are local heuristic search algorithms.
- Especially good for problems that have large and poorly understood search spaces.
- Genetic algorithms use a randomized parallel beam search to explore the state space.
- You must be able to define a good fitness function, and of course, a good state representation.

GA *(Fitness, Fitness_threshold,p,r,m)*

- P← randomly generate *p* individuals
- For each *i* in *P*, compute *Fitness(i)*
- While [$\max_i$ *Fitness*(*i*)] < *Fitness_threshold*
  1. Probabilistically **select** *(1-r)p* members of *P* to add to P_S.
  2. Probabilistically choose *(r·p)/2* pairs of individuals from *P*.
     For each pair, $\langle i_1, i_2 \rangle$ apply **crossover** and add the offspring to P_S
  3. **Mutate** *m·p* random members of P_S
  4. $P \leftarrow P_S$
  5. For each *i* in *P*, compute *Fitness(i)*
- Return the individual in *P* with the highest fitness.

Selecting Most Fit Individuals

Individuals are chosen probabilistically for survival and crossover based on **fitness proportionate selection**:

$$\Pr(\boldsymbol{i}) = \frac{\boldsymbol{Fitness}(\boldsymbol{i})}{\sum_{j=1}^{p} \boldsymbol{Fitness}(\boldsymbol{i}_j)}$$

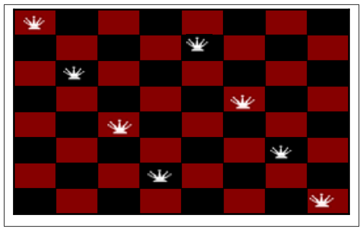
Other selection methods include:

- **Tournament Selection:** 2 individuals selected at random. With probabiltiy p, the more fit of the two is selected. With probability (1-p), the less fit is selected.
- **Rank Selection:** The individuals are sorted by fitness and the probability of selecting an individual is proportional to its rank in the list.

Binary string representations

- Individuals are usually represented as a string over a finite alphabet, usually bit strings.
- Individuals represented can be arbitrarily complex.
- E.g. each component of the state description is allocated a specific portion of the string, which encodes the values that are acceptable.
- Bit string representation allows crossover operation to change multiple values in the state description. Crossover and mutation can also produce previously unseen values.

8-queens State Representation



option 1: 86427531
option 2: 111 101 011 001 110 100 010 000

Mutation

Mutation: randomly toggle one bit

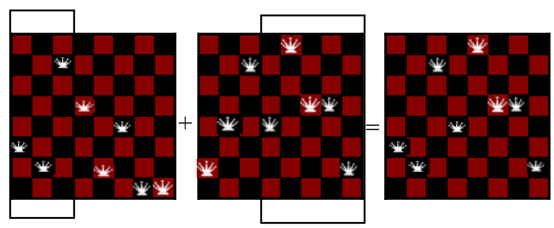
Individual A: 1 0 0 1 0 1 1 1 0 1

Individual A': 1 0 0 **0** 0 1 1 1 0 1

Mutation

- The **mutation** operator introduces random variations, allowing solutions to jump to different parts of the search space.
- What happens if the mutation rate is too low?
- What happens if the mutation rate is too high?
- A common strategy is to use a high mutation rate when search begins but to decrease the mutation rate as the search progresses.

Crossover Example



Another Example

World championship chocolate chip cookie recipe.

	Flour	Sugar	Salt	Chips	Vanilla	Fitness
1	4	1	2	16	1	
2	4.5	3	1	14	2	
3	2	1	1	8	1	
4	2.2	2.5	2.5	16	2	
5	4.1	2.5	1.5	10	1	
6	8	1.5	2	8	2	
7	3	1.5	1.5	8	2	

Generation 1

Crossover Operators

Single-point crossover:

Parent A: **1 0 0 1 0 1 1 1 0 1**

Parent B: 0 1 0 1 1 1 0 1 1 0

Child AB: **1 0 0 1 0** 1 0 1 1 0

Child BA: 0 1 0 1 1 **1 1 1 0 1**

Uniform Crossover

Uniform crossover:

Parent A: **1 0 0 1 0 1 1 1 0 1**

Parent B: 0 1 0 1 1 1 0 1 1 0

Child AB: **1** 1 **0 1** 1 1 **1** 1 **0 1**

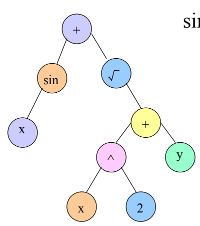
Child BA: 0 **0** 0 1 **0 1** 0 **1** 1 0

Remarks on GA's

- In practice, several 100 to 1000's of strings.
- **Crowding** can occur when an individual that is much more fit than others reproduces like crazy, which reduces diversity in the population.
- In general, GA's are highly sensitive to the representation.
- Value of crossover difficult to determine (so far) (→local search).

Genetic Programming

In **Genetic Programming**, programs are evolved instead of bit strings. Programs are represented by trees. For example:



$$\sin(\boldsymbol{x}) + \sqrt{\boldsymbol{x}^2 + \boldsymbol{y}}$$

Local Search - Summary

Surprisingly efficient search method.

- Wide range of applications.
  - any type of optimization / search task
- Handles search spaces that are too large
  - (e.g., 10^{1000}) for systematic search
- Often best available algorithm when lack of global information.
- Formal properties remain largely elusive.
- Research area will most likely continue to thrive.