Search Types

- Backtracking state-space search
- Local Search and Optimization
- Constraint satisfaction search
- Adversarial search

Local Search and Optimization

- Previous searches: keep paths in memory, and remember alternatives so search can backtrack. Solution is a path to a goal.
- Path may be irrelevant, if only the final configuration is needed (8-queens, IC design, network optimization, …)

Local Search

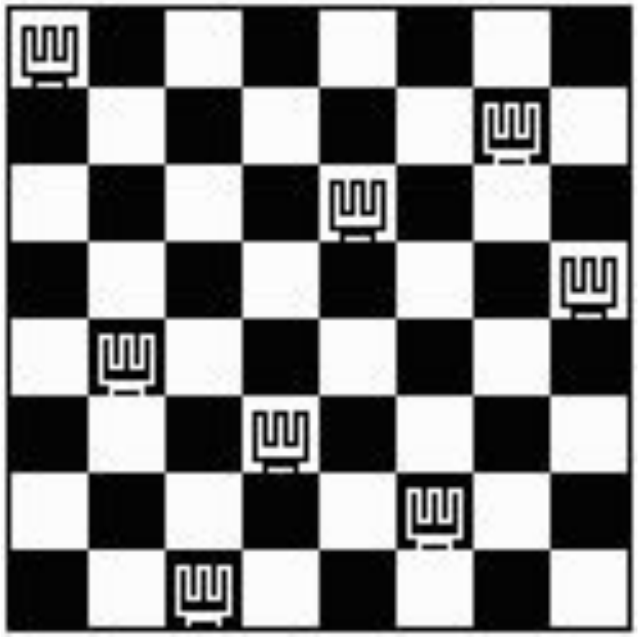
- Use a single current state and move only to neighbors.
- Use little space
- Can find reasonable solutions in large or infinite (continuous) state spaces for which the other algorithms are not suitable

Optimization

- Local search is often suitable for optimization problems. Search for best state by optimizing an objective function.
- F(x) where often x is a vector of continuous or discrete values
- Begin with a complete configuration
- A successor of state S is S with a single element changed
- Move from the current state to a successor state
- Low memory requirements, because the search tree or graph is not maintained in memory (paths are not saved)

Examples

- 8 queens: find an arrangement of 8 queens on a chess board such that no two queens are attacking each other
- Start with some arrangement of the queens, one per column
- X[j]: row of queen in column j
- Successors of a state: move one queen
- F: # pairs attacking each other



Examples

- Traveling salesman problem: visit each city exactly once
- Start with some ordering of the cities
- State representation – order of the cities visited (for eg)
- Successor state: a change to the current ordering
- F: length of the route

Examples

- Flight Travel problem
- See file on schedule. We will look at the details later, but the general problem is for all members of the Glass family to travel to the same place.
- A state consists of flights for each member of the family.
- Successor states: All schedules that have one person on the next later or the next earlier departing or returning flight.
- F: sum of different types of costs ($$, time, etc)

Examples

- Cryptosystems: resistance to types of attacks is often discussed in terms of Boolean functions used in them
- Much work on constructing Boolean functions with desired cryptographic properties (balancedness, high nonlinearity, etc.)
- One approach: local search, where states are represented with truth tables (that's a simplification); a successor results from a change to the truth table; objective functions have been devised to assess (estimate) relevant qualities of the Boolean functions

Examples

- Racing yacht hull design
- Design representation has multiple components, including a vector of B-Spline surfaces.
- Successors: modification of a B-Spline surface (e.g.).
- Objective function estimates the time the yacht would take to traverse a course given certain wind conditions (e.g.)

Comparison to tree/graphsearch framework

- Chapter 3: start state is typically not a complete configuration. Chapter 4: all states are
- Chapter 3: binary goal test; Chapter 4: no binary goal test, unless one can define one in terms of the objective function for the problem (e.g., no attacking pairs in 8-queens)
- h: estimate of the distance to the nearest goal
- objective function: preference/quality measure – how good is this state?
- Chapter 3: saving paths
- Chapter 4: start with a complete configuration and make modifications to improve it

Visualization

- States are laid out in a landscape
- Height corresponds to the objective function value
- Move around the landscape to find the highest (or lowest) peak
- Only keep track of the current states and immediate neighbors

Local Search Alogorithms

- Two strategies for choosing the state to visit next
  - Hill climbing
  - Simulated annealing
- Then, an extension to multiple current states:
  - Genetic algorithms

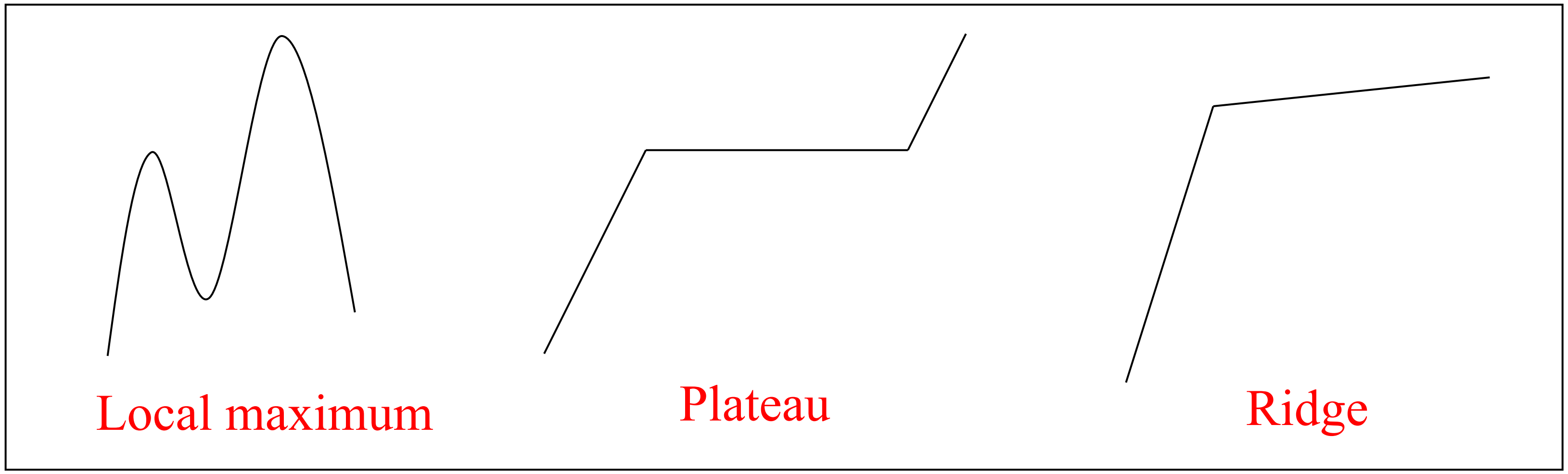
Hillclimbing
(Greedy Local Search)

- Generate nearby successor states to the current state
- Pick the best and replace the current state with that one.
- Loop

Hill-climbing search problems

(this slide assumes maximization rather than minimization)

- *Local maximum*: a peak that is lower than the highest peak, so a suboptimal solution is returned
- *Plateau*: the evaluation function is flat, resulting in a random walk
- *Ridges*: slopes very gently toward a peak, so the search may oscillate from side to side



Random restart hill-climbing (note: there are many variants of hill climbing)

- Start different hill-climbing searches from random starting positions stopping when a goal is found
- Save the best result from any search so far
- If all states have equal probability of being generated, it is complete with probability approaching 1 (a goal state will eventually be generated).
- Finding an optimal solution becomes the question of sufficient number of restarts
- Surprisingly effective, if there aren't too many local maxima or plateaux

Simulated Annealing

- Based on a metallurgical metaphor
  - Start with a temperature set very high and slowly reduce it.
  - Run hillclimbing with the twist that you can occasionally replace the current state with a worse state based on the current temperature and how much worse the new state is.

Simulated Annealing

- Annealing: harden metals and glass by heating them to a high temperature and then gradually cooling them
- At the start, make lots of moves and then gradually slow down

Simulated Annealing

- More formally…
  - Generate a random new neighbor from current state.
  - If it's better take it.
  - If it's worse then take it with some probability proportional to the temperature and the delta between the new and old states.

Simulated annealing

- Probability of a move decreases with the amount ΔE by which the evaluation is worsened
- A second parameter T is also used to determine the probability: high T allows more worse moves, T close to zero results in few or no bad moves
- *Schedule* input determines the value of T as a function of the completed cycles

```
function Simulated-Annealing(start,schedule)
    current ← start
    for t ← 1 to ∞ do
        T ← schedule[t]
        if T=0 then return current
        next ← a randomly selected successor of current
        ΔE ← Value[next] – Value[current]
        if  ΔE > 0 then current ← next
        else current  ← next only with probability $e^{\Delta E/T}$
```

Intuitions

- Hill-climbing is incomplete
- Pure random walk, keeping track of the best state found so far, is complete but very inefficient
- Combine the ideas: add some randomness to hill-climbing to allow the possibility of escape from a local optimum

Intuitions

- the algorithm wanders around during the early parts of the search, hopefully toward a good general region of the state space
- Toward the end, the algorithm does a more focused search, making few bad moves

Theoretical Completeness

- There is a proof that if the schedule lowers T slowly enough, simulated annealing will find a global optimum with probability approaching 1
- In practice, that may be way too many iterations
- In practice, though, SA can be effective at finding good solutions

Local Beam Search

- Keep track of k states rather than just one, as in hill climbing
- In comparison to beam search we saw earlier, this algorithm is state-based rather than node-based.

Local Beam Search

- Begins with k randomly generated states
- At each step, all successors of all k states are generated
- If any one is a goal, alg halts
- Otherwise, selects best k successors from the complete list, and repeats

Local Beam Search

- Successors can become concentrated in a small part of state space
- Stochastic beam search: choose k successors, with probability of choosing a given successor increasing with value
- Like natural selection: successors (offspring) of a state (organism) populate the next generation according to its value (fitness)

Genetic Algorithms

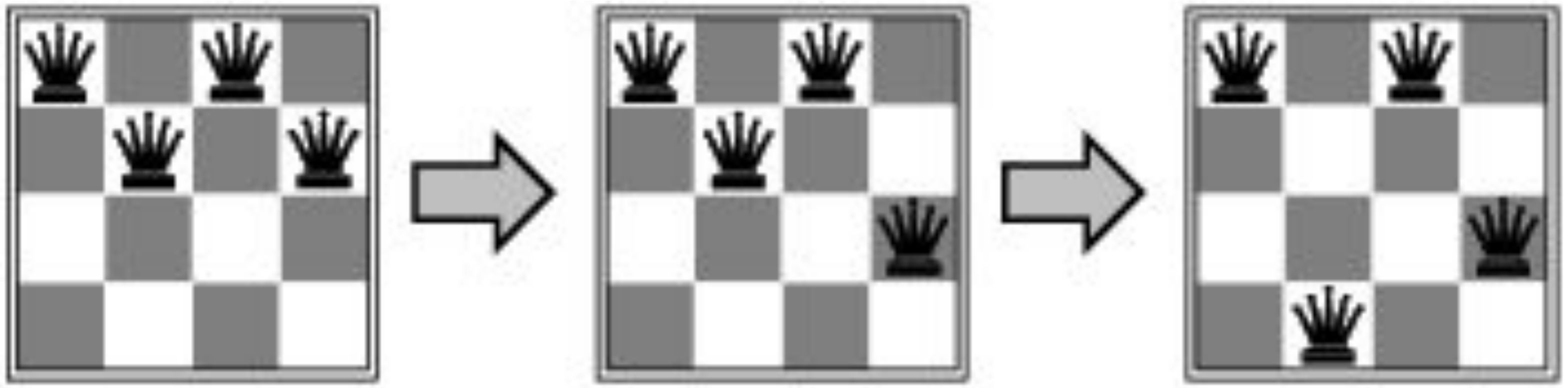
- Variant of stochastic beam search
- Combine two parent states to generate successors

```
function GA (pop, fitness-fn)
Repeat
   new-pop = {}
   for i from 1 to size(pop):
      x = rand-sel(pop,fitness-fn)
      y = rand-sel(pop,fitness-fn)
      child = reproduce(x,y)
      if (small rand prob): child ←mutate(child)
      add child to new-pop
   pop = new-pop
Until an indiv is fit enough, or out of time
Return best indiv in pop, according to fitness-fn
```

```
function reproduce(x,y)
   n = len(x)
   c = random num from 1 to n
   return:
     append(substr(x,1,c),substr(y,c+1,n)
```

Example: *n*-queens

- Put *n* queens on an *n* × *n* board with no two queens on the same row, column, or diagonal



Genetic Algorithms Notes

- Representation of individuals
  - *Classic approach*: individual is a string over a finite alphabet with each element in the string called a *gene*
  - Usually binary instead of AGTC as in real DNA
- Selection strategy
  - Random
  - Selection probability proportional to fitness
  - Selection is done with replacement to make a very fit individual reproduce several times
- Reproduction
  - Random pairing of selected individuals
  - Random selection of *cross-over* points
  - Each gene can be altered by a random *mutation*

Genetic Algorithms
When to use them?

- Genetic algorithms are easy to apply
- Results can be good on some problems, but bad on other problems
- Genetic algorithms are not well understood

Example Local Search Problem Formulation

- Group travel: people traveling from different places: See chapter4example.txt on the course schedule.
- From Segaran, T. Programming Collective Intelligence, O'Reilly, 2007.

Wrapup

- The relevant part of the book is Chapter 4.1