Local Search Algorithms

This lecture topic

Read Chapter 4.1-4.2

Next lecture topic

Read Chapter 5

(Please read lecture topic material before and after each lecture on that topic)

You will be expected to know

- Local Search Algorithms
  - Hill-climbing search
    - Gradient Descent in continuous spaces
  - Simulated annealing search
  - Local beam search
  - Genetic algorithms
  - (where applicable) Linear Programming
- Random Restart & Tabu Wrappers for above
- Difficulties: Local optima, plateaus, ridges, etc.
- Minimize cost, maximize value

  Value ≈ Constant − Cost; Cost ≈ Constant − Value

Local search algorithms

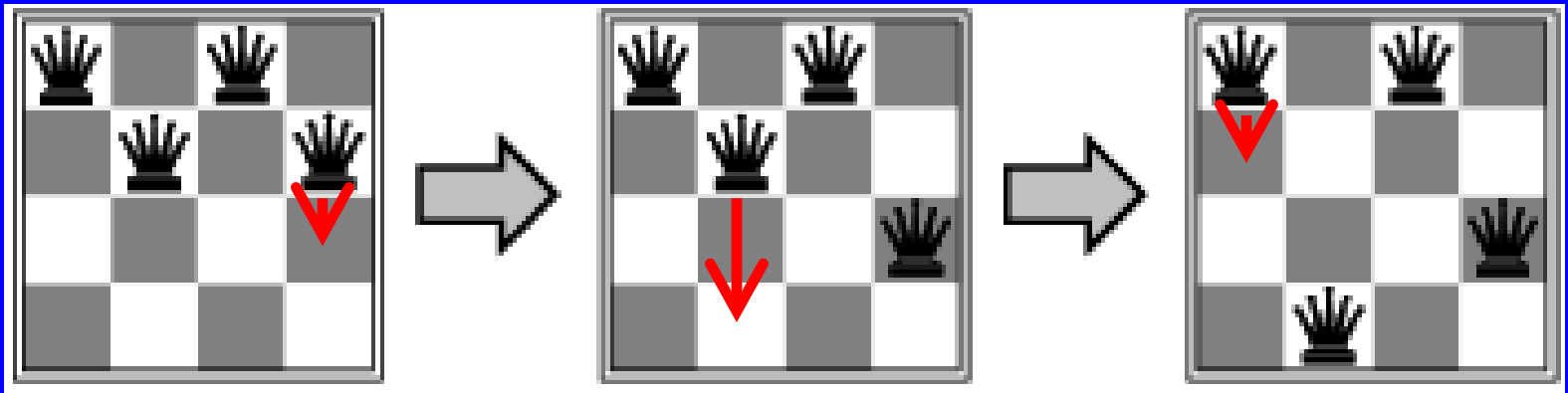
- In many problems, the path to the goal is irrelevant; the goal state itself is the solution
  - Local search: Widely used for BIG problems
  - Returns good, but not optimal, solutions

- Search space = set of "complete" configurations
- Solution = configuration satisfying constraints
  - E.g., N-queens, VLSI layout, Airline flight scheduling

- Keep single "current" state, or small set of states.
  - Try to improve it or them.
- Very memory efficient (keep one or a few states)
  - You get to control how much memory you use.

Example: *n*-queens

- Goal: Put *n* queens on an *n* × *n* board
  - No queens on same row, column, or diagonal
- Neighbor: move 1 queen to another row
  - Search: Go from one neighbor to the next….



Note that a state cannot be an incomplete configuration with m<n queens

Algorithm Design Considerations

- How do you represent your problem?
- What is a “complete state”?
- What is your objective function?
  - How do you measure cost or value of a state?
- What is a “neighbor” of a state?
  - Or, what is a “step” from one state to another?
  - How can you compute a neighbor or a step?
- Are there any constraints you can exploit?

Random Restart Wrapper

- These are stochastic local search methods
  - Different solution for each trial and initial state
- Almost every trial hits difficulties (see below)
  - Most trials will not yield a good result (sadly)
- Many random restarts improve your chances
  - Many “shots at goal” may, finally, get a good one
- Restart a random initial state; many times
  - Report the best result found; across many trials

Random Restart Wrapper

```
BestResultFoundSoFar <- infinitely bad;
UNTIL ( you are tired of doing it ) DO {
    Result <- ( local search from random initial state );
    IF ( Result is better than BestResultFoundSoFar )
        THEN ( set BestResultFoundSoFar to Result );
    }
RETURN BestResultFoundSoFar;
```

Typically, “you are tired of doing it” means that some resource limit is exceeded, e.g., number of iterations, wall clock time, CPU time, etc. It may also mean that Result improvements are small and infrequent, e.g., less than 0.1% Result improvement in the last week of run time.

Tabu Search Wrapper

- Recently visited states added to a tabu-list
  - Temporarily excluded from being visited again.
- Force solver away from explored regions
  - (In principle) avoids getting stuck in local minima.
- Implemented as Hash table + FIFO queue
  - Unit time cost per step; constant memory cost.
- You control how much memory is used
  - Run close to the edge but don’t blow out.

Tabu Search Wrapper


```
UNTIL ( you are tired of doing it ) DO {
    set Neighbor to makeNeighbor( CurrentState );
    IF ( Neighbor is in HASH ) THEN ( discard Neighbor );
        ELSE { push Neighbor onto FIFO, pop OldestState;
               remove OldestState from HASH, insert Neighbor;
               set CurrentState to Neighbor;
               run yourFavoriteLocalSearch on CurrentState; } }
```

Local Search Algorithms

- Hill-climbing search
  - Gradient Descent in continuous spaces
  - Newton's method to find roots
- Simulated annealing search
- Local beam search
- Genetic algorithms
- (where applicable) Linear Programming

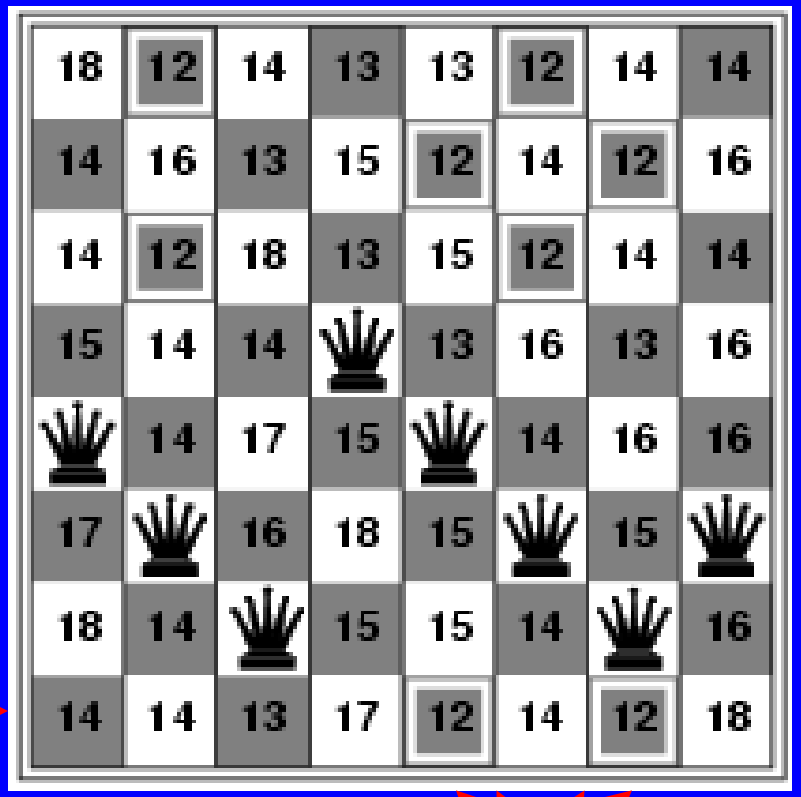
Hill-climbing search

- "Like climbing Everest in thick fog with amnesia"

```
function HILL-CLIMBING( problem) returns a state that is a local maximum
    inputs: problem, a problem
    local variables: current, a node
                     neighbor, a node

    current ← MAKE-NODE(INITIAL-STATE[problem])
    loop do
        neighbor ← a highest-valued successor of current
        if VALUE[neighbor] ≤ VALUE[current] then return STATE[current]
        current ← neighbor
```

Hill-climbing search: 8-queens problem

- *h* = number of pairs of queens that attack each other, either directly or indirectly (*h = 17* for this state)



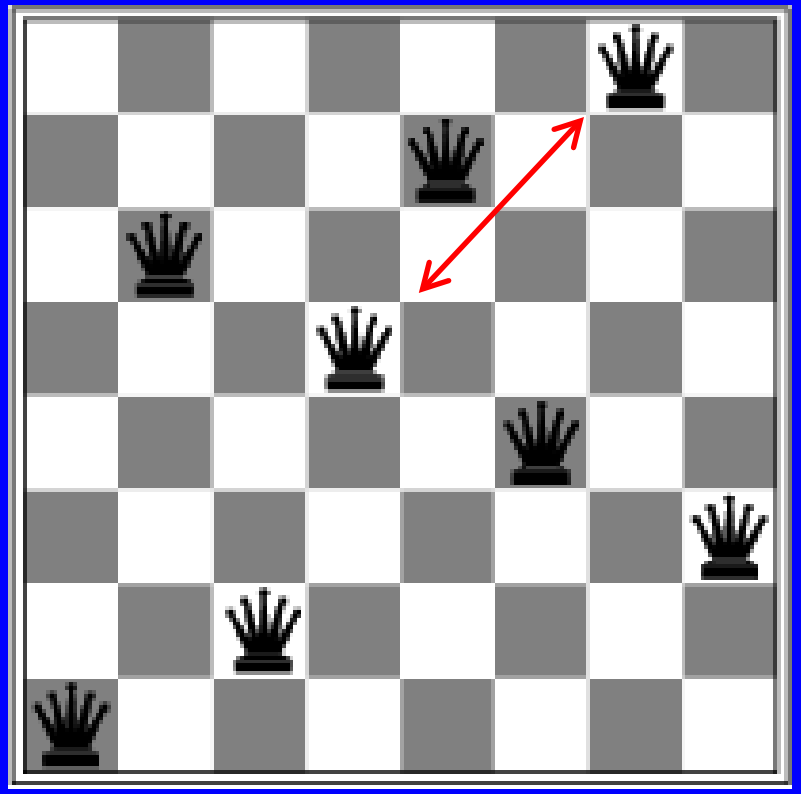
Each number indicates *h* if we move a queen in its column to that square

12 (boxed) = best *h* among all neighors; select one randomly

Hill-climbing search: 8-queens problem

A local minimum with $h = 1$

All one-step neighbors have higher h values

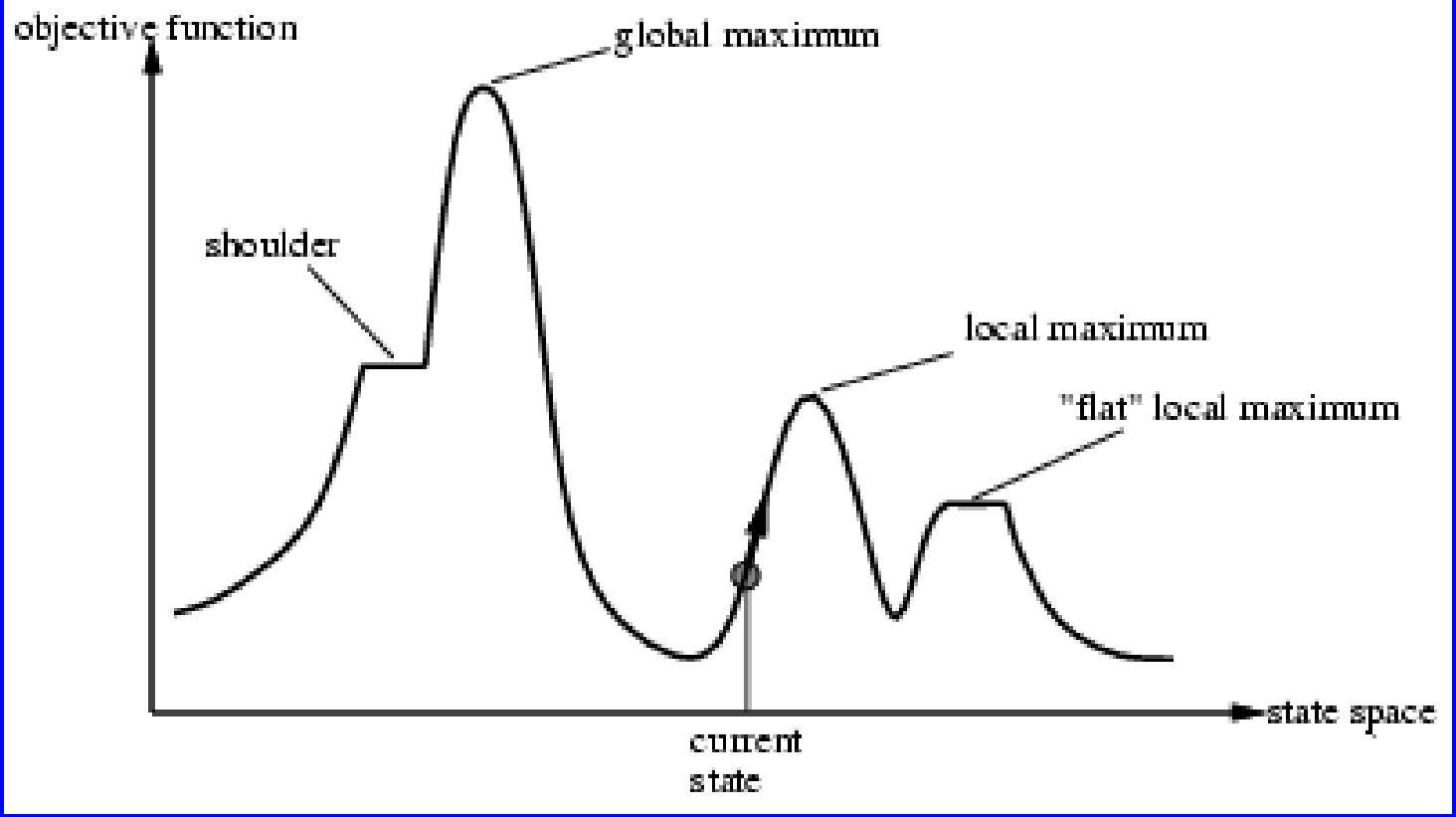


What can you do to get out of this local minimum?

Hill-climbing Difficulties

These difficulties apply to ALL local search algorithms, and become MUCH more difficult as the dimensionality of the search space increases to high dimensions.

- Problems: depending on state, can get stuck in local maxima
  - Many other problems also endanger your success!!



Hill-climbing Difficulties

These difficulties apply to ALL local search algorithms, and become MUCH more difficult as the dimensionality of the search space increases to high dimensions.

- Ridge problem: Every neighbor appears to be downhill
  - But the search space has an uphill!! (worse in high dimensions)

Ridge:
Fold a piece of paper and hold it tilted up at an unfavorable angle to every possible search space step. Every step leads downhill; but the ridge leads uphill.

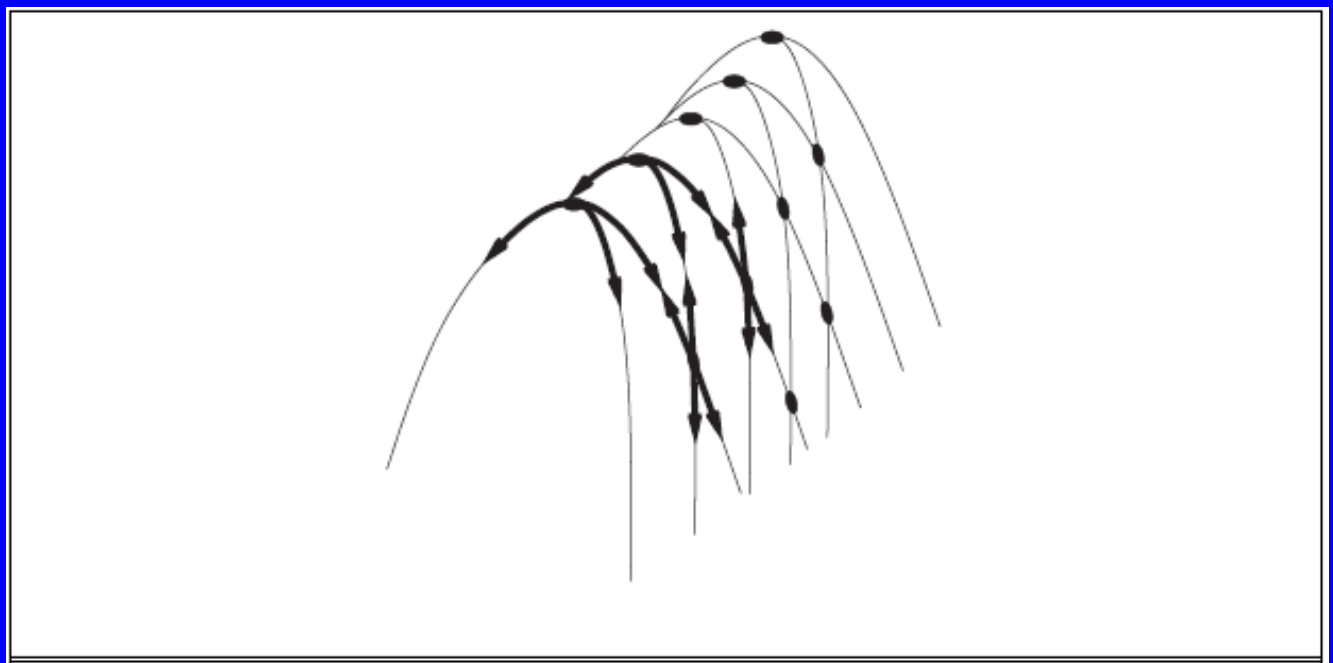
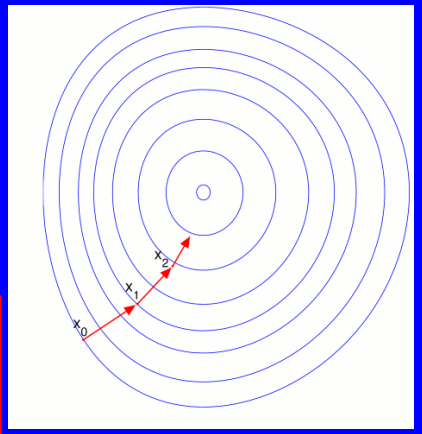


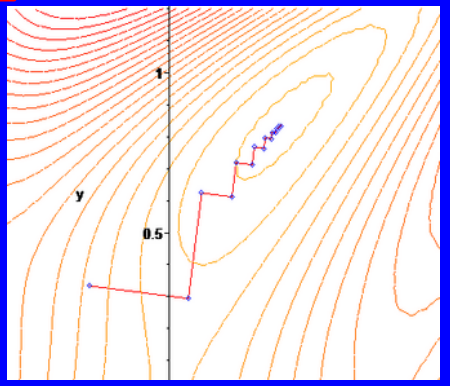
Figure 4.4 **FILES: figures/ridge.eps (Tue Nov 3 16:23:29 2009).** Illustration of why ridges cause difficulties for hill climbing. The grid of states (dark circles) is superimposed on a ridge rising from left to right, creating a sequence of local maxima that are not directly connected to each other. From each local maximum, all the available actions point downhill.

Gradient Descent

Hill-Climbing in Continuous Spaces



Gradient = the most direct direction up-hill in the objective (cost) function, so its negative minimizes the cost function.



* Assume we have some cost-function: $C(x_1,\ldots,x_n)$
and we want minimize over continuous variables X1,X2,..,Xn

1. Compute the *gradient :* $\frac{\partial}{\partial x_i} C(x_1,\ldots,x_n) \qquad \forall i$

2. Take a small step downhill in the direction of the gradient:

$$x_i \rightarrow x'_i = x_i - \lambda \frac{\partial}{\partial x_i} C(x_1,\ldots,x_n) \qquad \forall i$$

3. Check if $C(x_1,..,x'_i,..,x_n) < C(x_1,..,x_i,..,x_n)$

4. If true then accept move, if not reject.

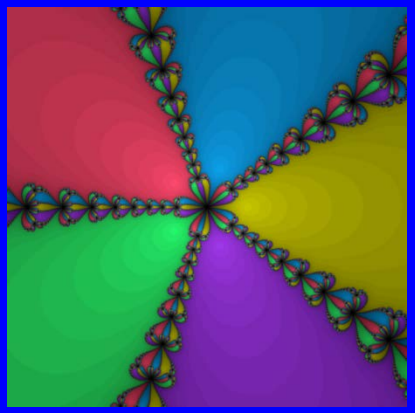
5. Repeat.

Gradient Descent

Hill-climbing in Continuous Spaces

- How do I determine the gradient?
  - Derive formula using multivariate calculus.
  - Ask a mathematician or a domain expert.
  - Do a literature search.
- Variations of gradient descent can improve performance for this or that special case.
  - See Numerical Recipes in C (and in other languages) by Press, Teukolsky, Vetterling, and Flannery.
  - Simulated Annealing, Linear Programming too
- Works well in smooth spaces; poorly in rough.

Newton's Method



Basins of attraction for x5 − 1 = 0;
darker means more iterations to converge.

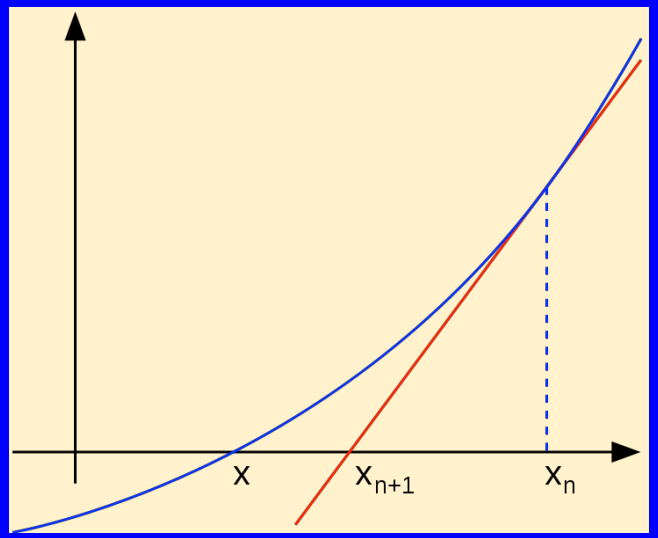
- Want to find the roots of f(x)
  - A root of f(x) is a value of x for which f(x)=0.

- To do that, we compute the tangent at Xn and compute where it crosses the x-axis.

$$\nabla f(x_n) = \frac{0 - f(x_n)}{x_{n+1} - x_n} \Rightarrow x_{n+1} = x_n - \frac{f(x_n)}{\nabla f(x_n)}$$

- Optimization: find roots of $\nabla f(x_n)$

$$\nabla\nabla f(x_n) = \frac{0 - \nabla f(x_n)}{x_{n+1} - x_n} \Rightarrow x_{n+1} = x_n - \frac{\nabla f(x_n)}{[\nabla\nabla f(x_n)]}$$



- Does not always converge & sometimes unstable.
- If it converges, it converges very fast
- Works well for smooth non-pathological functions where derivative approximates root.
- Works poorly for wiggly ill-behaved functions where derivative leads away from root.

Simulated annealing search

- **Idea:**

Escape local maxima by allowing some "bad" moves but gradually decrease their frequency

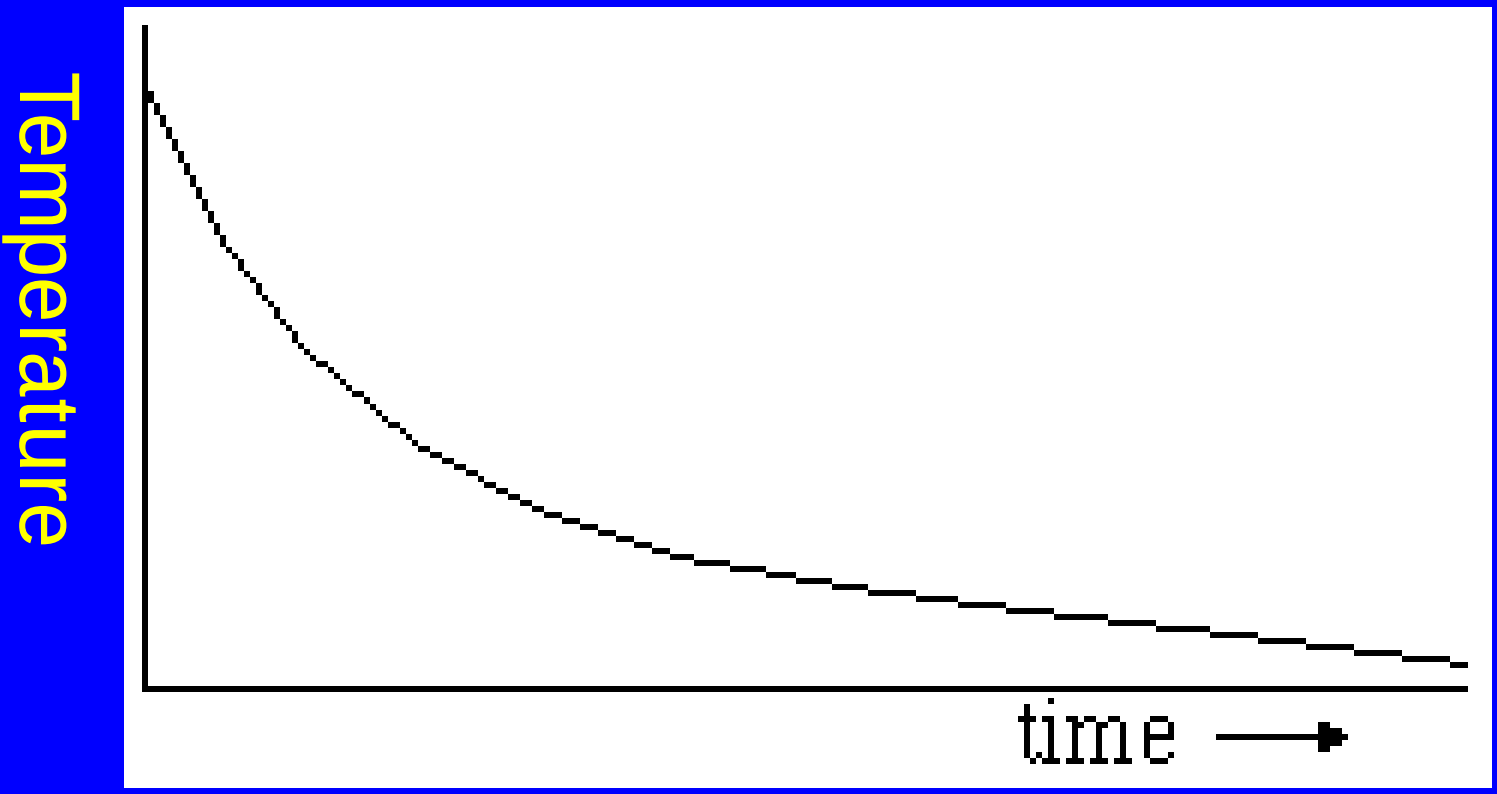
```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
  inputs: problem, a problem
          schedule, a mapping from time to "temperature"
  local variables: current, a node
                   next, a node
                   T, a "temperature" controlling prob. of downward steps

  current ← MAKE-NODE(INITIAL-STATE[problem])
  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^(Δ E/T)
```

Improvement: Track the BestResultFoundSoFar. Here, this slide follows Fig. 4.5 of the textbook, which is simplified.

Typical Annealing Schedule

Usually a Decaying Exponential

Axis Values are Scaled to Fit Problem



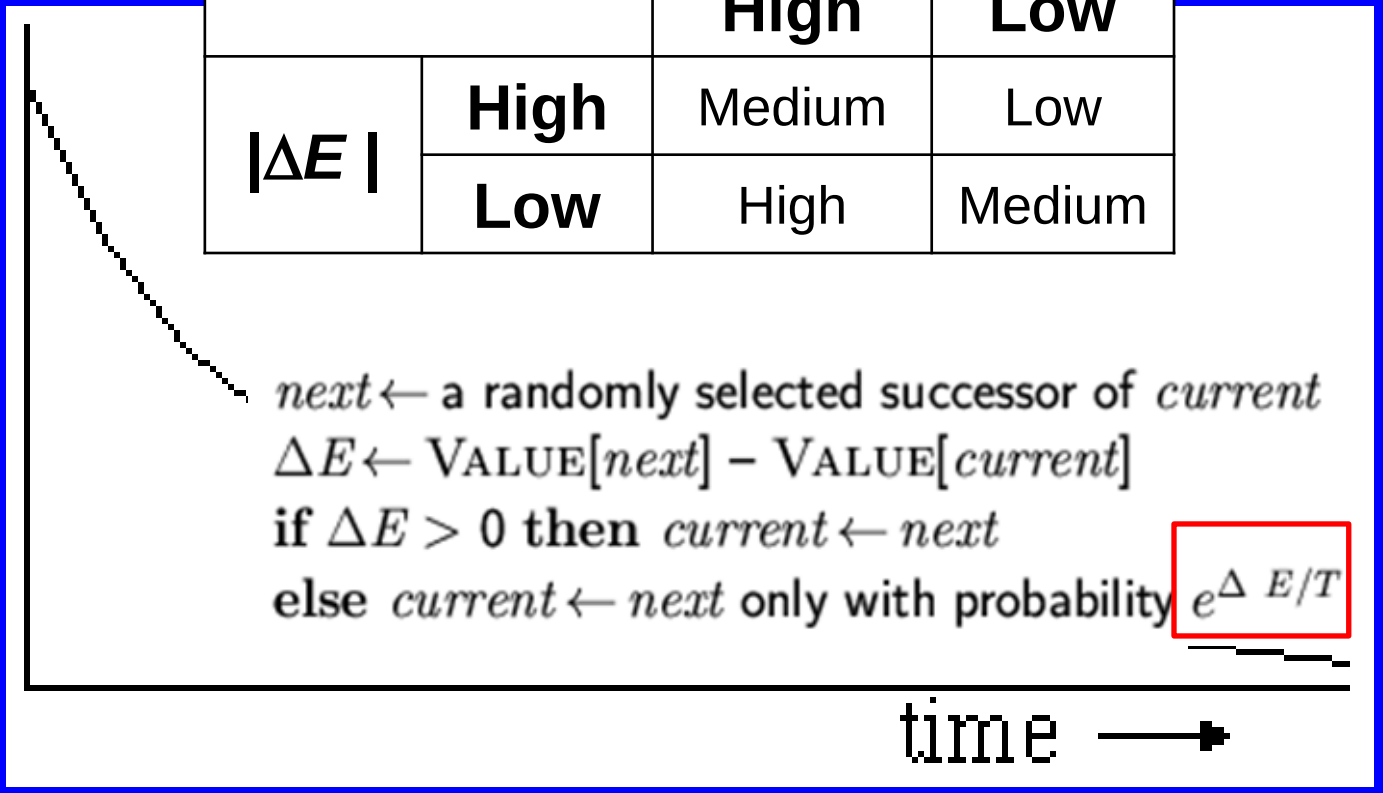
P(accepting a worse successor)

Decreases as Temperature T decreases

Increases as | ΔE | decreases

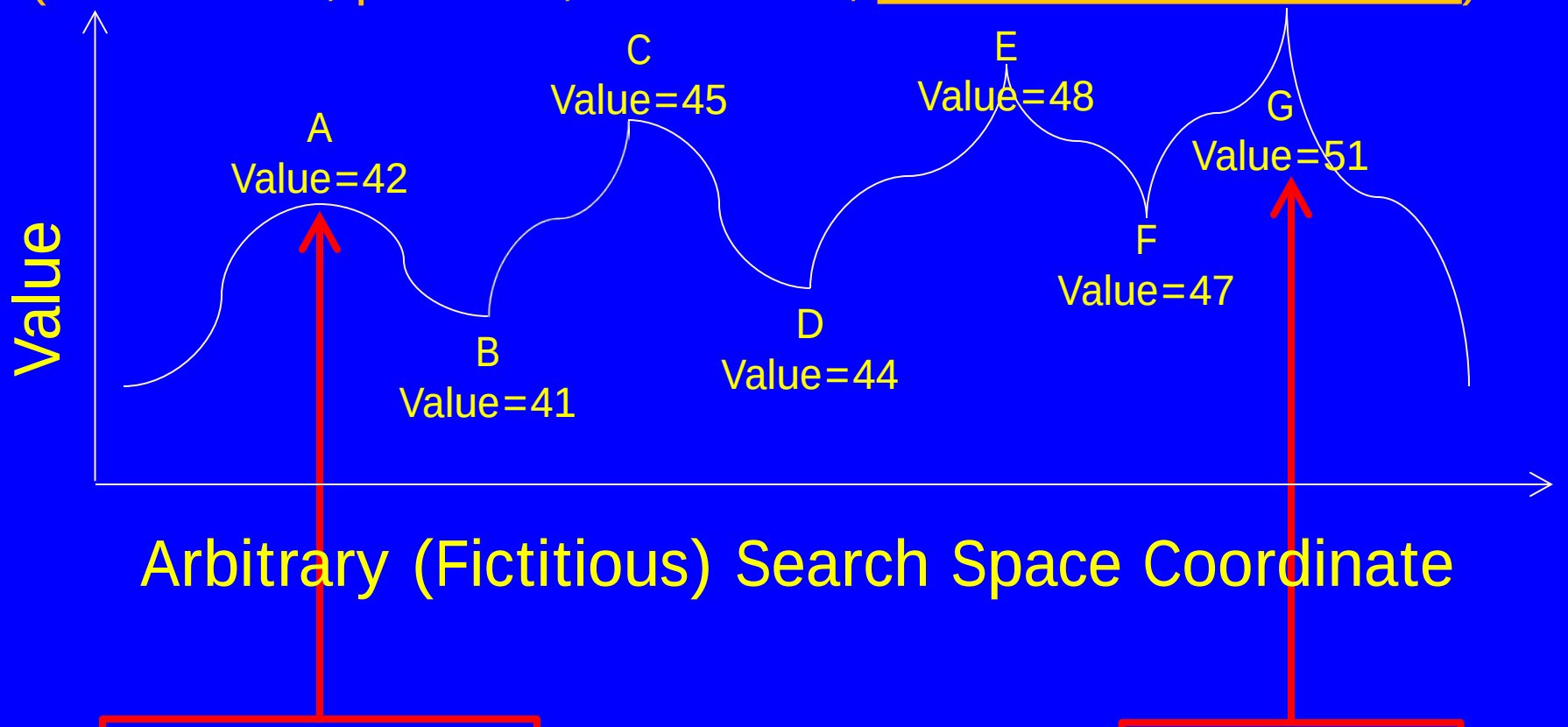
(Sometimes step size also decreases with T)

$e^{\Delta E / T}$		Temperature T	
		High	**Low**
$\lvert\Delta E\rvert$	**High**	Medium	Low
	Low	High	Medium



Goal: "Ratchet" up a jagged slope

(see HW #2, prob. #5; here T = 1; cartoon is NOT to scale)



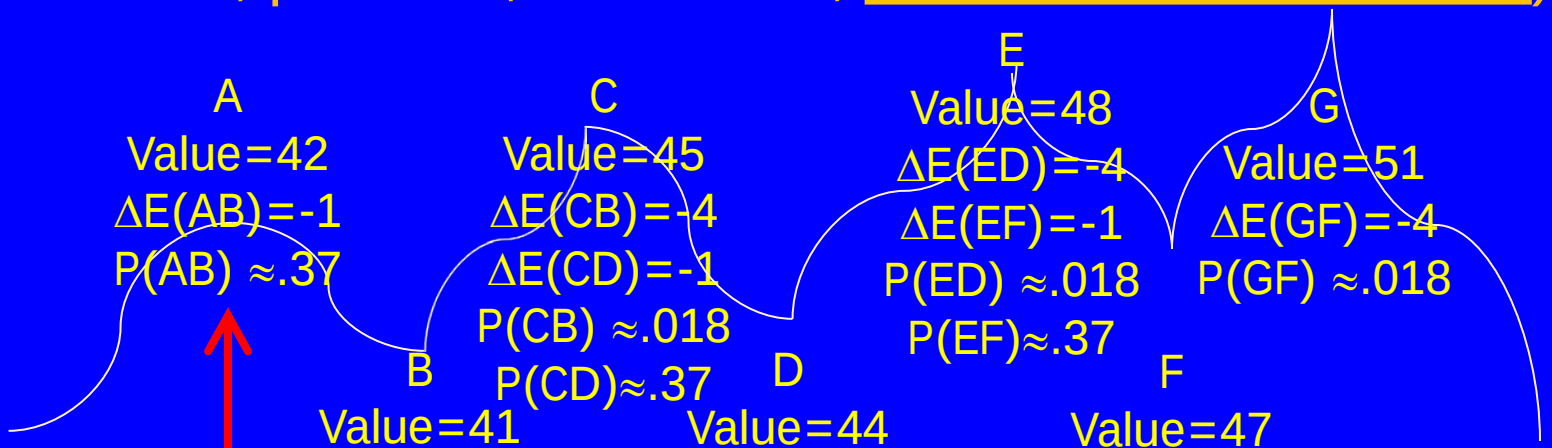
Your "random restart wrapper" starts here.

You want to get here. HOW??

This is an illustrative cartoon.

Goal: "Ratchet" up a jagged slope

(see HW #2, prob. #5; here T = 1; cartoon is NOT to scale)



Your "random restart wrapper" starts here.

This is an illustrative cartoon.

x	-1	-4
e^x	≈.37	≈.018

From A you will accept a move to B with P(AB) ≈.37.
From B you are equally likely to go to A or to C.
From C you are ≈20X more likely to go to D than to B.
From D you are equally likely to go to C or to E.
From E you are ≈20X more likely to go to F than to D.
From F you are equally likely to go to E or to G.
Remember best point you ever found (G or neighbor?).

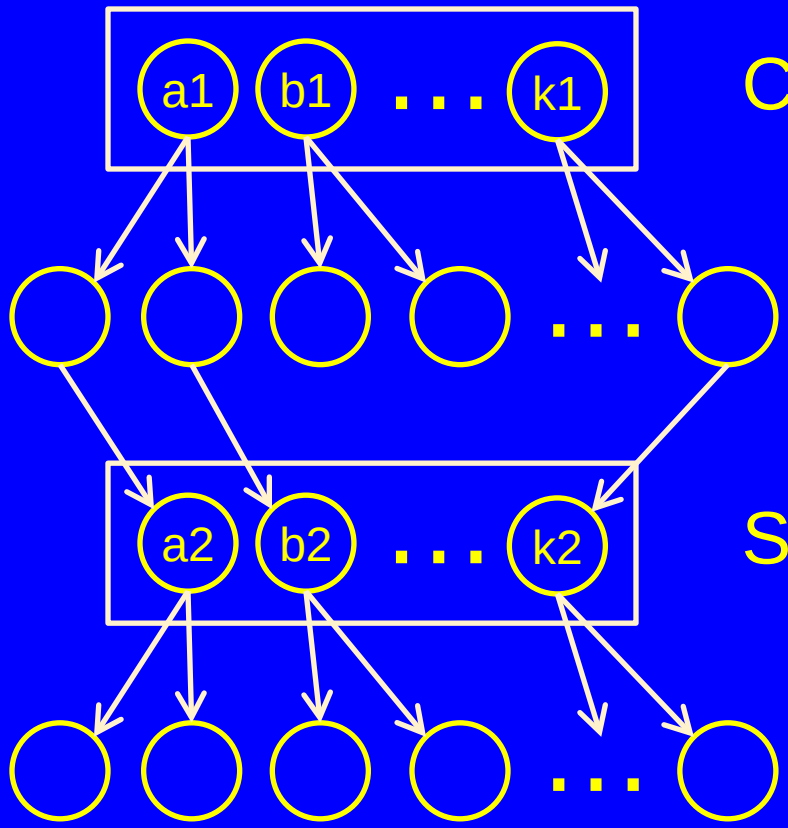
Properties of simulated annealing search

- One can prove: If *T* decreases slowly enough, then simulated annealing search will find a global optimum with probability approaching 1
  - However, this may take a VERY, VERY long time.
  - Note that, any finite search space, RANDOM GUESSING also will find a global optimum with probability approaching 1.
  - So, ultimately this is a very weak claim.
- Often works very well in practice
  - But usually VERY, VERY slow.
- Widely used in “very big” problems:
  - VLSI layout, national airline scheduling, large logistics ops, etc.

Local beam search

- Keep track of *k* states rather than just one.
- Start with *k* randomly generated states.
- At each iteration, all the successors of all *k* states are generated.
- If any one is a goal state, stop; else select the *k* best successors from the complete list and repeat.
- Concentrates search effort in areas believed to be fruitful.
  - May lose diversity as search progresses, resulting in wasted effort.

Local beam search



Create *k* random initial states

Generate their children

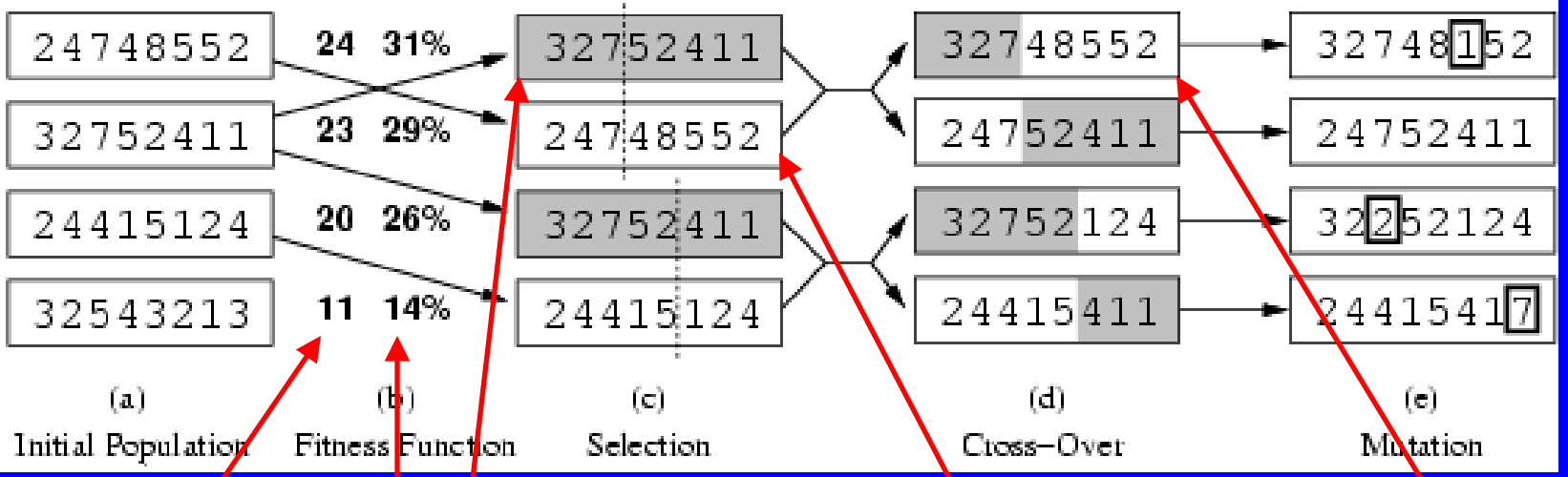
Select the *k* best children

Repeat indefinitely…

Is it better than simply running *k* searches?
Maybe…??

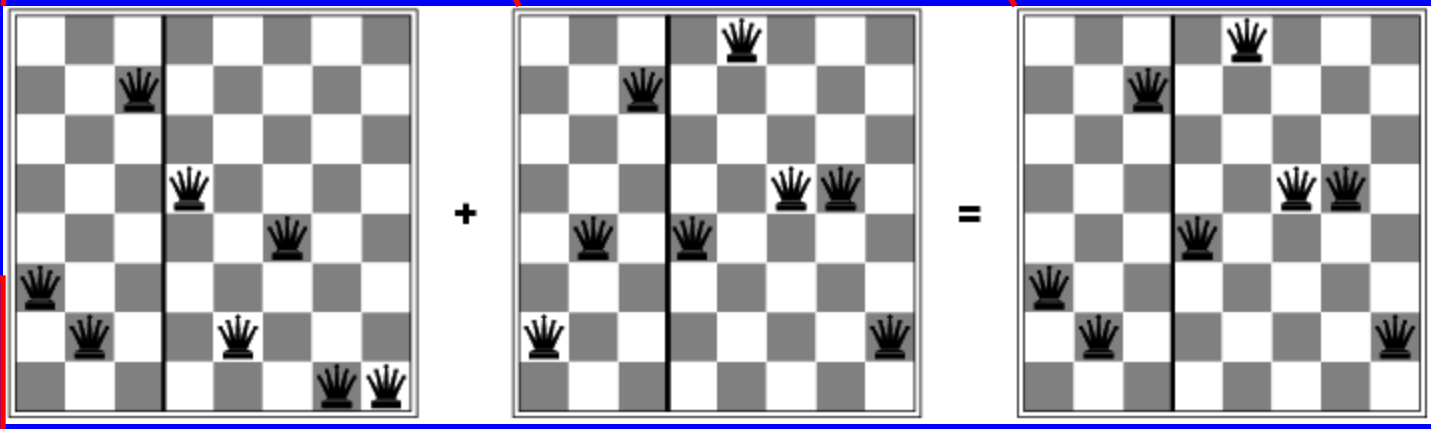
Genetic algorithms (Darwin!!)

- A state = a string over a finite alphabet (an individual)
- Start with *k* randomly generated states (a population)
- Fitness function (= our heuristic objective function).
  - Higher fitness values for better states.
- Select individuals for next generation based on fitness
  - P(individual in next gen.) = individual fitness/Σ population fitness
- Crossover fit parents to yield next generation (off-spring)
- Mutate the offspring randomly with some low probability



fitness = #non-attacking queens

probability of being in next generation = fitness/(Σ_i fitness_i)



How to convert a fitness value into a probability of being in the next generation.

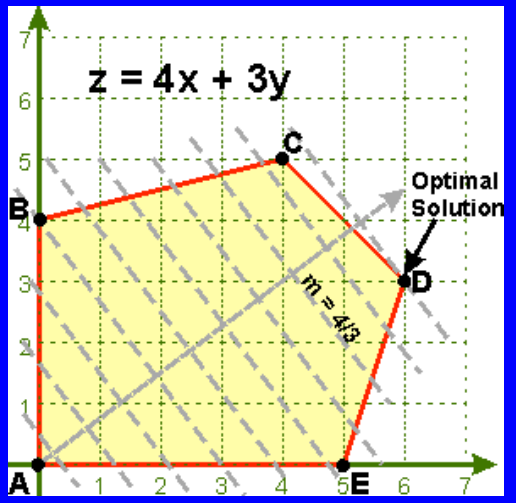
- Fitness function: #non-attacking queen pairs
  - min = 0, max = 8 × 7/2 = 28
- Σ_i fitness_i = 24+23+20+11 = 78
- P(child_1 in next gen.) = fitness_1/(Σ_i fitness_i) = 24/78 = 31%
- P(child_2 in next gen.) = fitness_2/(Σ_i fitness_i) = 23/78 = 29%; etc

Linear Programming
Efficient Optimal Solution
For a Restricted Class of Problems

Problems of the sort:

$$\text{maximize } c^T x$$
$$\text{subject to : } Ax \le a;\ Bx = b$$

- Very efficient "off-the-shelves" solvers are available for LPs.
- They quickly solve large problems with thousands of variables.



Linear Programming Constraints

- Maximize: $z = c_1 x_1 + c_2 x_2 + \ldots + c_n x_n$
- Primary constraints: $x_1 \geq 0,\ x_2 \geq 0,\ \ldots,\ x_n \geq 0$
- Arbitrary additional linear constraints:

$$a_{i1} x_1 + a_{i2} x_2 + \ldots + a_{in} x_n \leq a_i,\ (a_i \geq 0)$$

$$a_{j1} x_1 + a_{j2} x_2 + \ldots + a_{jn} x_n \geq a_j \geq 0$$

$$b_{k1} x_1 + b_{k2} x_2 + \ldots + b_{kn} x_n = b_k \geq 0$$

- Restricted class of linear problems.
  - Efficient for very large problems(!!) in this class.

Summary

- Local search maintains a complete solution
  - Seeks to find a consistent solution (also complete)
- Path search maintains a consistent solution
  - Seeks to find a complete solution (also consistent)
- Goal of both: complete and consistent solution
  - Strategy: maintain one condition, seek other
- Local search often works well on large problems
  - Abandons optimality
  - Always has some answer available (best found so far)