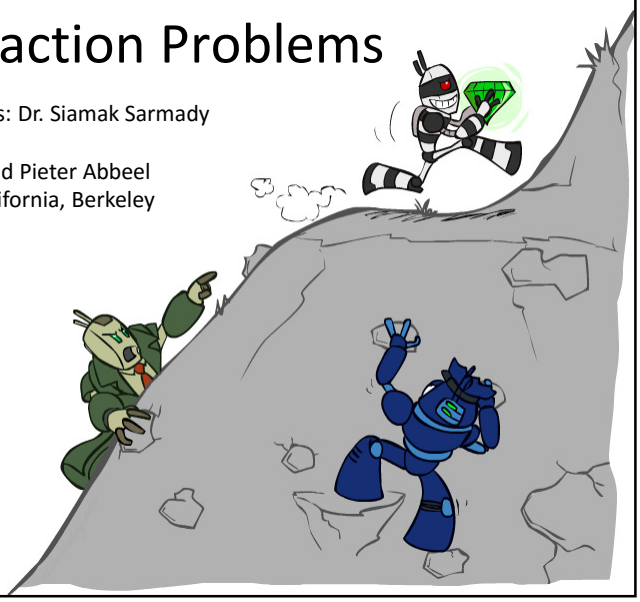
Artificial Intelligence
Chapter 5
Constraint Satisfaction Problems

Updates and Additions: Dr. Siamak Sarmady

By: Dan Klein and Pieter Abbeel
University of California, Berkeley

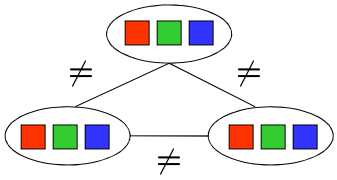


Today

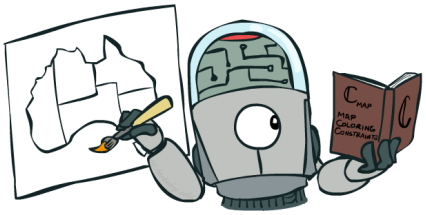
- Efficient Solution of CSPs
- Local Search

Reminder: CSPs

- CSPs:
  - Variables
  - Domains
  - Constraints
    - Implicit (provide code to compute)
    - Explicit (provide a list of the legal tuples)
    - Unary / Binary / N-ary



- Goals:
  - Here: find any solution
  - Also: find all, find best, etc.



Backtracking Search

```
function BACKTRACKING-SEARCH(csp) returns solution/failure
  return RECURSIVE-BACKTRACKING({ }, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) returns soln/failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
      if value is consistent with assignment given CONSTRAINTS[csp] then
          add {var = value} to assignment
          result ← RECURSIVE-BACKTRACKING(assignment, csp)
          if result ≠ failure then return result
          remove {var = value} from assignment
  return failure
```

- Recursive impl. but can do it in non-recursive methods…
- Start with empty assignment

- If managed to finish all return.
- Peek an unassigned var.
- Loop & Consider a new value
- If it does not break constraints
- add to assignment
- Recurse & If fine(call can find consistent vals.) , return result
- Else remove last assignment and return failure

- If cannot find anything after all just return failure

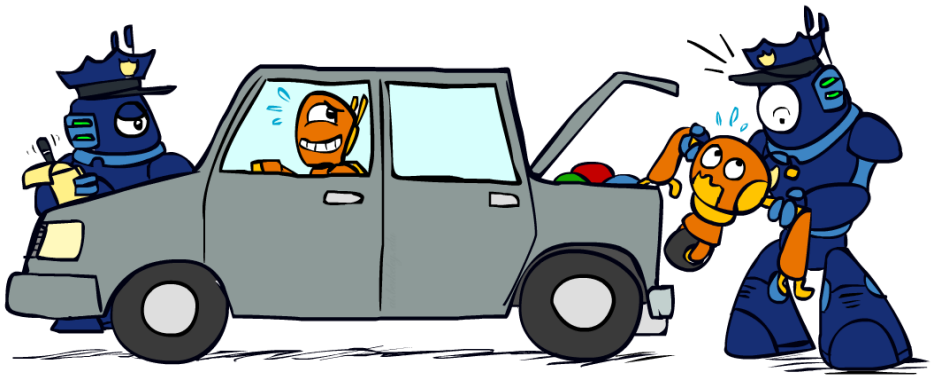
Improving Backtracking

- General-purpose ideas give huge gains in speed
  - One variable at a time, check constraints as you go
  - … but it's all still NP-hard

- Filtering: Can we detect inevitable failure early?
  - Filtering → Arc Checking, Full Consistency → Full Arc Checking
- Ordering:
  - Which variable should be assigned next? (Min Rem Values)
  - In what order should its values be tried? (Least Constraining Value)

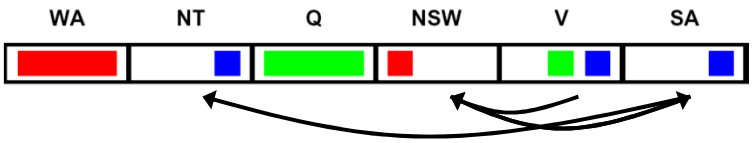
- Structure: Can we exploit the problem structure?
  - Can we use exploit CSP graph to do faster? (sub graphs? Tree conversion?)

Arc Consistency and Beyond



Arc Consistency of an Entire CSP

- A simple form of propagation makes sure all arcs are consistent:



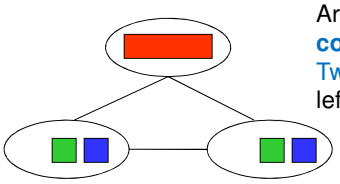
- Arc consistency detects failure earlier than forward checking
- Important: If X loses a value, neighbors of X need to be rechecked!
- Must rerun after each assignment!

Remember: Delete from the tail!

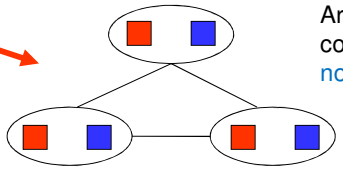
If empty domain, backtrack

Limitations of Arc Consistency

- After enforcing arc consistency:
  - Can have no solution left
  - Can have one solution left
  - Can have multiple solutions left
  - Also might have no solutions left but in a way that is not obvious
- Arc consistency still runs inside a backtracking search!



Arcs **consistent**, Two solutions left



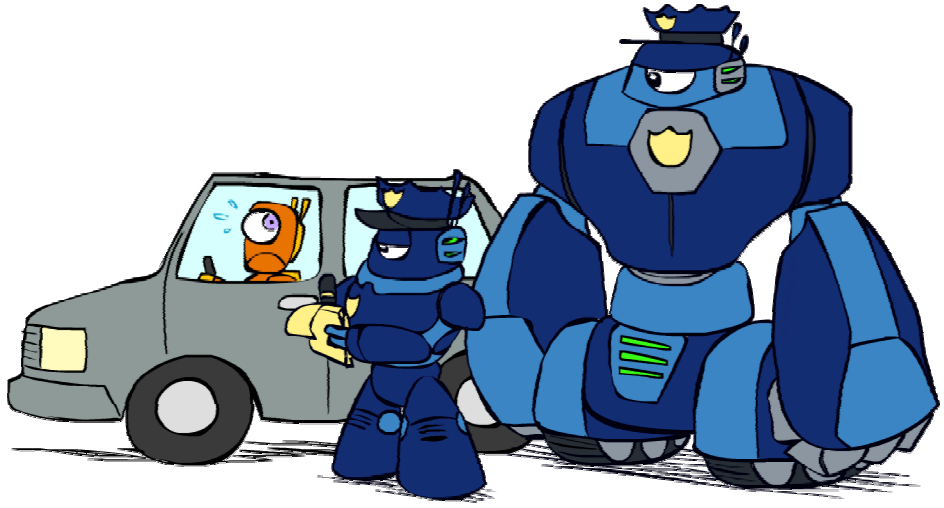
Arcs are consistent but no solution

What went wrong here?

Cannot track **three way interactions**

[Demo: coloring -- forward checking] [Demo: coloring -- arc consistency]

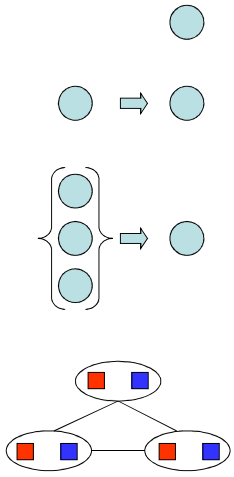
K-Consistency



Stronger Consistency Checks!

K-Consistency

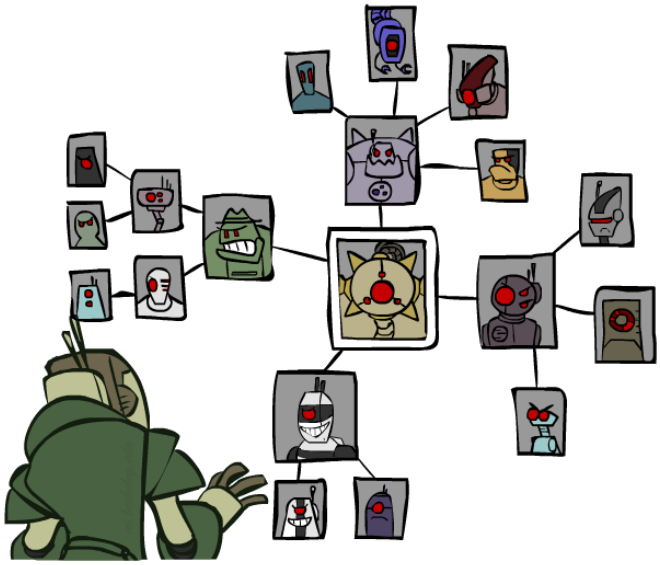
- Increasing degrees of consistency
  - **1-Consistency (Node Consistency):** Each single node's domain has a value which meets that node's unary constraints
  - **2-Consistency (Arc Consistency):** For each pair of nodes, any consistent assignment to one can be extended to the other
    - any value you chose for one of them from its domain, there is a consistent value that can be selected for the other
  - **K-Consistency:** For each k nodes, any consistent assignment to k-1 can be extended to the k^{th} node
    - If you have a consistent selection for k-1 variables, you also have a value for the K-th variable that is consistent.
    - **Strong K-consistency (next page):** for K consistency, you can check each subset to be consistent. In that case we have strong K consistency check.
    - Notice that you should keep track of more assignments (in case backtrack needed)
- Higher k more expensive to compute
- (You need to know the k=2 case: arc consistency)



Strong K-Consistency

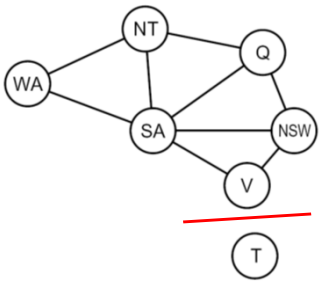
- **Strong k-consistency:** also 1, 2, …, k-2, k-1 consistent
- **Claim:** If we have strong n-consistency (among n variables), means we can solve without backtracking!
  (we may spend more time than backtracking though!)
- Why?
  - Let's assume we just enforced n-consistency (i.e. removed any inconsistent options) …
    1. Choose any assignment to any variable
    2. Choose a new variable
    3. By 2-consistency, there is a choice consistent with the first
    4. Choose a new variable
    5. By 3-consistency, there is a choice consistent with the first 2
    6. …
- Lots of middle ground between simple backtracking search and n-consistency! e.g. forward checking, full arc-consistency (k=2), path consistency (k=3) ,…

Structure



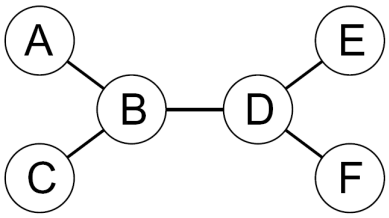
Problem Structure

- Extreme case: independent sub problems
  - Example: Tasmania and mainland do not interact
- Independent sub problems are identifiable as connected components of constraint graph
- Suppose a graph of n variables can be broken into sub problems of only c variables:
  - Worst-case solution cost is $O((n/c)(d^c))$, i.e. linear in terms of n
  - E.g., n = 80, d = 2, c =20 (80 variables, binary values, clusters of size 20)
  - 2^{80} = 4 billion years at 10 million nodes/sec (naïve backtracking search)
  - $(4)(2^{20})$ = 0.4 seconds at 10 million nodes/sec
- But the bad news is that not all graphs can be broken into separate parts…



Tree-Structured CSPs

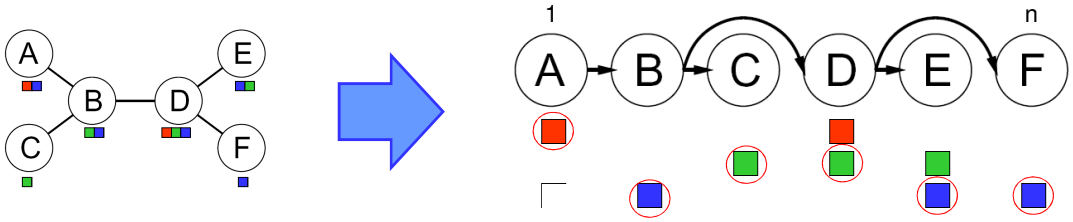
- How if we have a graph that is in fact a tree?
  - Compared to graphs where all nodes are interrelated, Trees only have limited number of binary relations.



- Theorem: if the constraint graph has no loops and only binary relations, the CSP can be solved in $O(n\ d^2)$ time
  - Compare to general CSPs, where worst-case time is $O(d^n)$
- This property also applies to probabilistic reasoning (later): an example of the relation between syntactic restrictions and the complexity of reasoning

Tree-Structured CSPs

- Algorithm for tree-structured CSPs:
  - Order: Choose a root variable (any of them), order variables so that parents precede children
    - This is one possible linearization. Assume we have initial domains of each. We just check the arcs shown...

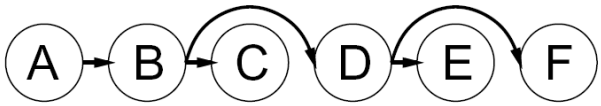


  - Remove backward: For i = n : 2, apply RemoveInconsistent(Parent(X_i),X_i)
  - Assign forward: For i = 1 : n, assign X_i consistently with Parent(X_i)
    - Note: we know that for each value selected, we have a consistent value for next variable, since we enforced consistency in previous step

- Runtime: O(n d^2) (why?)
  - Enforcing arc consistency d^2 (d values), number of arcs we consider n

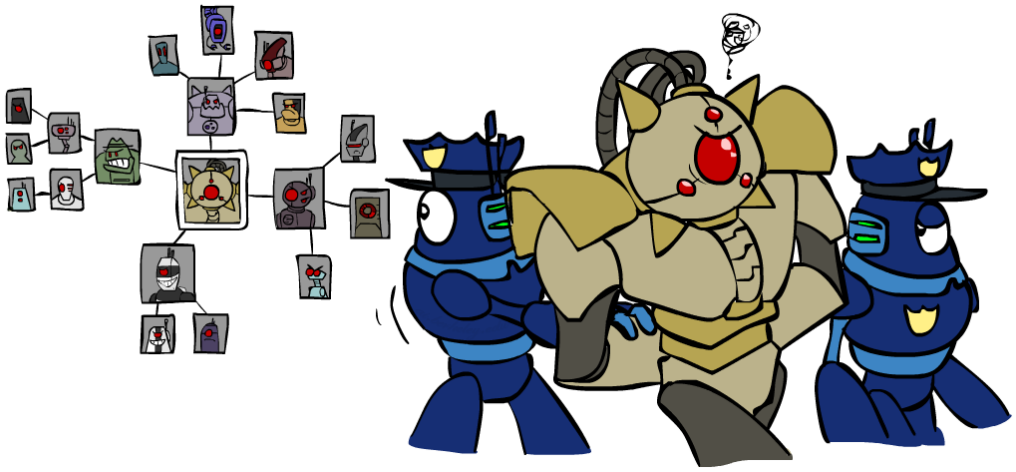
Tree-Structured CSPs

- **Claim 1:** After one backward pass, all root-to-leaf arcs are consistent
- **Proof:** Each X→Y was made consistent at one point and Y's domain could not have been reduced thereafter (because Y's children were processed before Y)
  - The only way, an arc become inconsisent is that something is removed from head (does not happen) e.g. In order to make E->F inconsistent is to remove somethig from F's domain



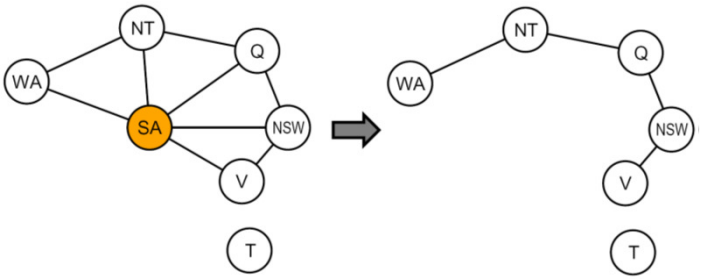
- **Claim 2:** If root-to-leaf arcs are consistent, forward assignment will not backtrack
- **Proof:** Induction on position, because we know whatever we choose for B there is something left for C …and it goes on…
- Why it works with trees and not with graphs with cycles?
  - since for example there could be a forward arc between E and F that would build a loop and F will have two parents … there is no guarantee that we have compatible values this way (2 consistency does not mean 3 consistency between D, E, F)

Improving Structure



So perhaps we can break a graph into a tree? Get rid of some nodes and you'll possibly have trees.

Nearly Tree-Structured CSPs



- **Conditioning:** instantiate a variable, prune its neighbors' domains
  e.g. SA=blue and then remove blue from the domain of neighbors, then solve like previous case
- **Cutset conditioning:** instantiate (in all ways) a set of variables such that the remaining constraint graph is a tree (in above example cutset is just one node)
- **Complexity:** Cutset size c gives runtime $O((d^c) (n-c) d^2)$, very fast for small c
  $(n-c).d^2$ for the tree, if the value assigned to Cutset did not work, you may need to test other values for it: d^c

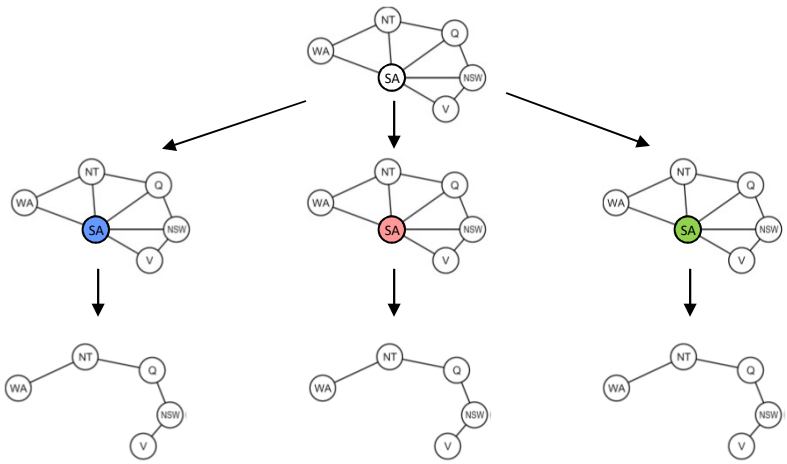
Cutset Conditioning

Choose a cutset

Instantiate the cutset (all possible ways)

Compute residual CSP for each assignment

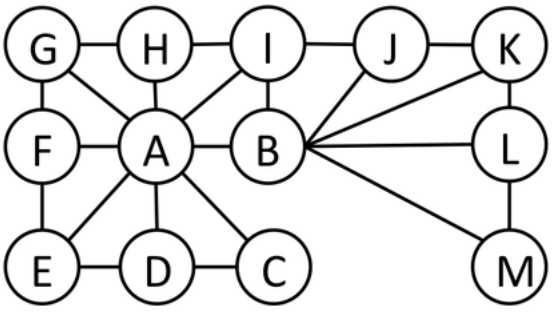
Solve the residual CSPs (tree structured)



If you want more than 1 variable in your cutset, it becomes a backtracking search. Every time, you cut some variables until you are done (starting by cutting bigger numbers?)

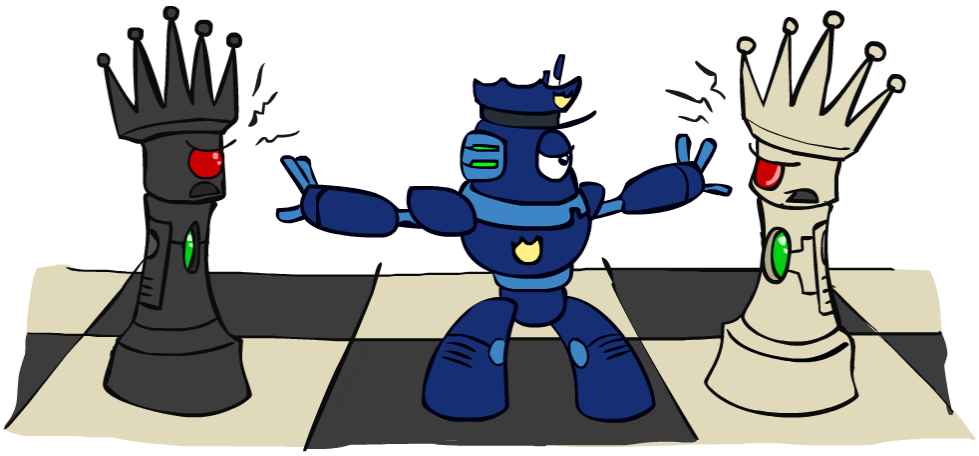
Cutset Quiz

- Find the smallest cutset for the graph below.



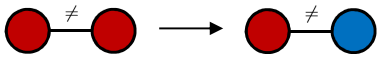
There are algorithms to identify the smallest cutset. For example they select nodes with largest numbers of constraints and then check what happens if you cut them away… if needed, enhance the cutset…

Iterative Improvement



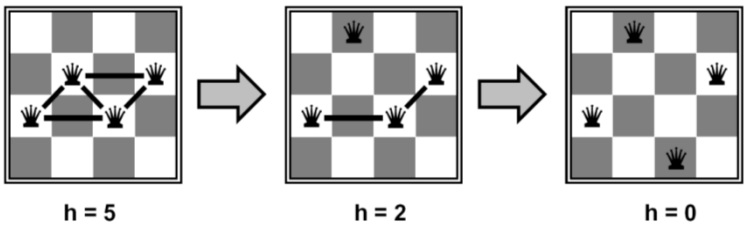
Iterative Improvement Algorithms for CSPs

- **Local search methods:** typically work with "complete" states, i.e., all variables assigned
- To apply to CSPs:
  - **Initial:** Start with fully assigned values to all variables
  - **Selection:** Take an assignment with unsatisfied constraints
  - **Improve:** *reassign* variable values
  - **No fringe:** Live on the edge (i.e. all variables assigned, depth=n) and no track of what you did before.



- **Iterative Min Conflict Algorithm:**
  - **While not solved:**
    - **Variable selection:** randomly select any conflicted variable (not necessarily with more violations)
    - **Improve (value selection):** select a value that produces min-conflicts
    - **heuristic:**
      - Choose a value that violates the fewest constraints
      - I.e., hill climb with h(n) = total number of violated constraints

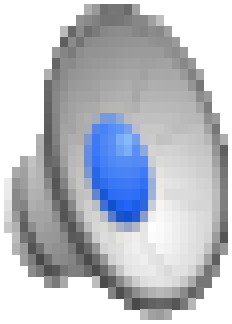
Iterative Min Conflict - Example: 4-Queens



- **States:** 4 queens in 4 columns (4^4 = 256 states)
- **Operators:** move queen in column
- **Goal test:** no attacks
- **Evaluation:** c(n) = number of attacks

[Demo: n-queens – iterative improvement (L5D1)]
[Demo: coloring – iterative improvement]

Video of Demo Iterative Improvement – n Queens



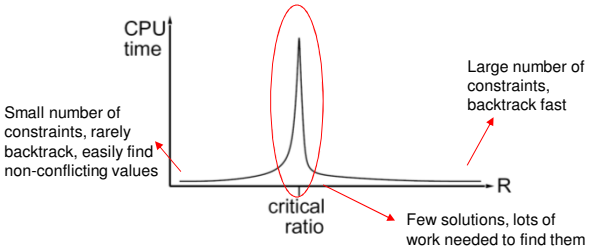
It is not like backtracking, you may return and modify the same variable many times.

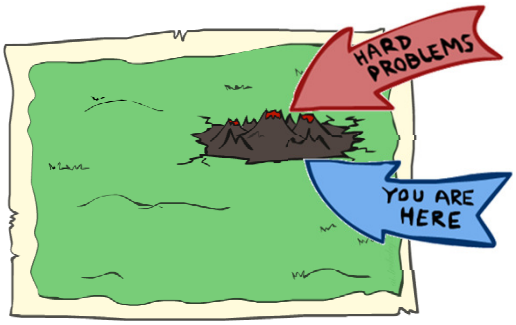
Video of Demo Iterative Improvement – Coloring

Performance of Min-Conflicts

- Given random initial state, the method can solve n-queens in almost constant time for arbitrary n with high probability (e.g., n = 10,000,000)! But not guaranteed to solve!
- The same appears to be true for any randomly-generated CSP *except* in a narrow range of the ratio

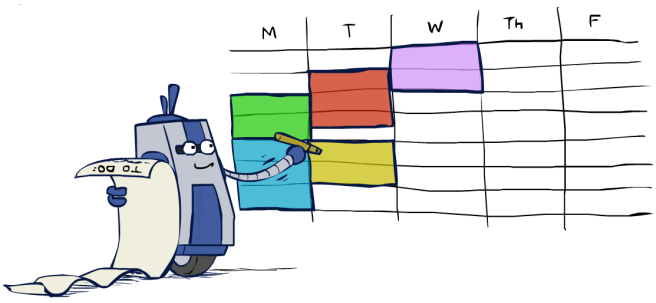
$$R = \frac{\text{number of constraints}}{\text{number of variables}}$$





Summary: CSPs

- CSPs are a special kind of search problem:
  - States are partial assignments
  - Goal test defined by constrai
- Basic solution: backtracking sea
- Speed-ups:
  - Ordering (values and variables)
  - Filtering
  - Structure



- Iterative min-conflicts is often effective in practice

Local Search

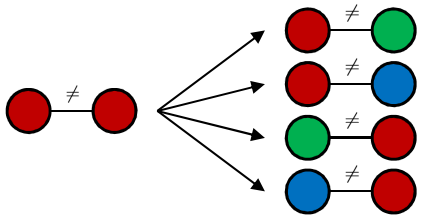
Local Search

- **Tree search:** keeps unexplored alternatives on the fringe (ensures completeness)

  vs.

- **Local search:** improve a single option until you can't make it better (no fringe!)

- **New successor function:** local changes

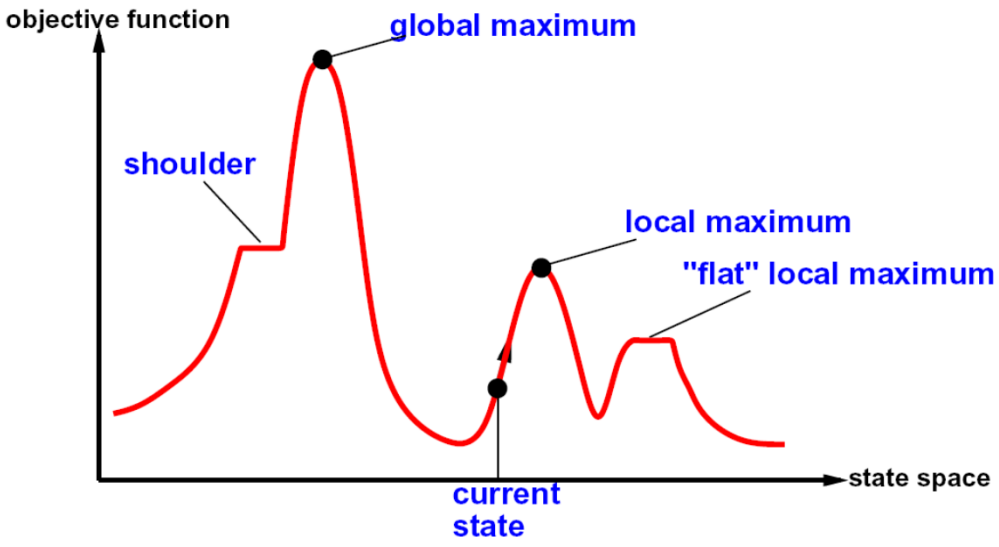


- Generally much faster and more memory efficient (but incomplete and suboptimal)

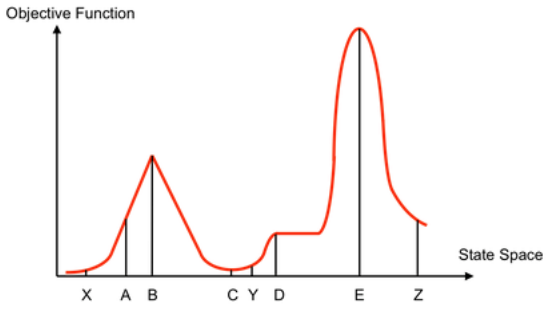
Hill Climbing

- Simple, general idea:
  - **Initial:** Start wherever
  - **Repeat:** move to the best neighboring state
  - **Stop:** If no neighbors better than current, quit
- What's bad about this approach?
  - Complete?
  - Optimal?
- What's good about it?

Hill Climbing Diagram



Hill Climbing Quiz



Starting from X, where do you end up ?

Starting from Y, where do you end up ?

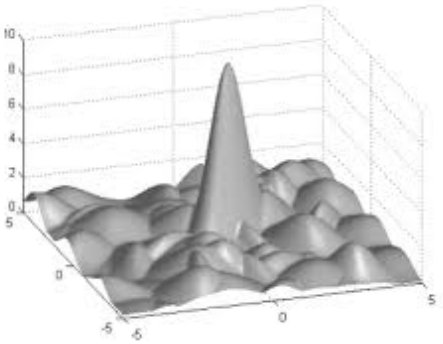
Starting from Z, where do you end up ?

Think: How about we start from thousands or millions of random points and do the hill climbing for each point?

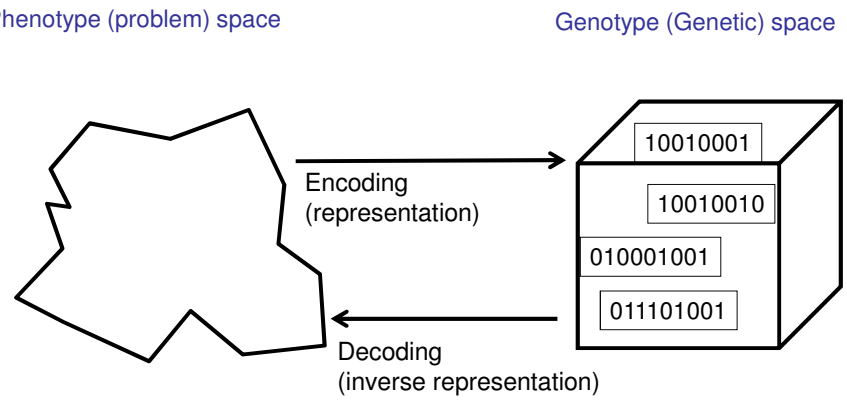
That would become methods like genetic algorithm etc.

A typical genetic algorithm implementation

- Genetic Algorithm is an optimization and search algorithm.
- It uses a crossover and fittest selection method for local search (i.e. finding local optimal)
- It uses a mutation function for global search (i.e. finding global optimal)



Representation



Initial Random Population

Population

- A pool of population (chromosomes) is randomly created
- The size of population is considered 100 (dividable by 4)

Fitness Function

- A fitness function is defined which determines how fit each of the chromosomes are.

010111011	F=1.5
011101001	F=2.3
…	
111001101	F=1.4
001001001	F=3.3
110101001	F=2.2
011000001	F=0.9
001100001	F=1.8
111101000	F=2.15
011100001	F=3.1
000101001	F=0.5

Sorting and Parent Selection

Sorting

- We sort the population by their fitness

Parent Selection

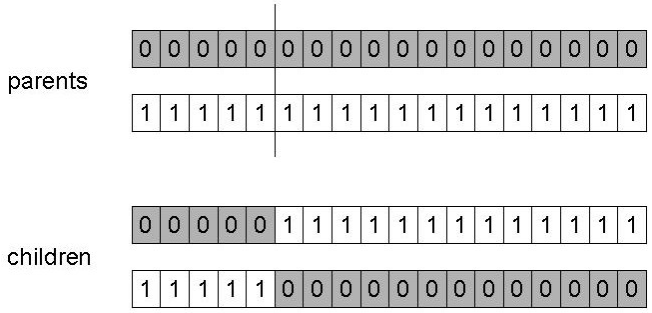
- We keep the top 50 individuals (more fit)
- We throw away 50 worse half
- Randomly select pairs

010111011	F=3.5
011101001	F=3.4
111001101	F=3.4
001001001	F=3.35
110101001	F=3.3
011000001	F=3.1
001100001	F=2.9
111101000	F=2.85
011100001	F=2.8
000101001	F=2.75
…	

Production, Step 1

Crossover (recombination)

- We have 50 parent chromosomes, each pair will produce 2 Childs.
- Choose a random point on the two parents
- Split parents at this crossover point
- Create children by exchanging tails
- P_c typically in range (0.6, 0.9)



010111011

011101001

111001101

001001001

110101001

011000001

001100001

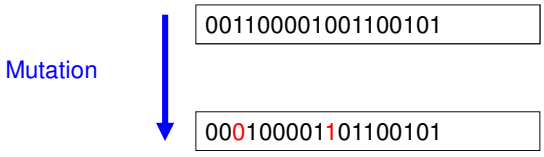
111101000

...

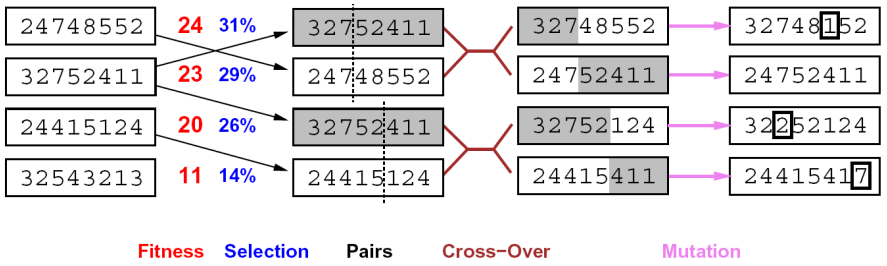
Production, Step 2

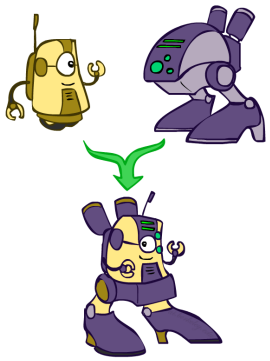
Mutation

- Alter each gene independently with a probability p_m
- p_m is called the mutation rate, typically between 1/pop_size and 1/chromosome_length
- The mutation rate should normally be very small (or it will ruin good results)
- We may consider another rate which randomly determines which/how many of the children are mutated.



Genetic Algorithms





- Genetic algorithms use a natural selection metaphor
  - Keep best N hypotheses at each step (selection) based on a fitness function
  - Also have pairwise crossover operators, with optional mutation to give variety
- Possibly the most misunderstood, misapplied (and even maligned) technique around

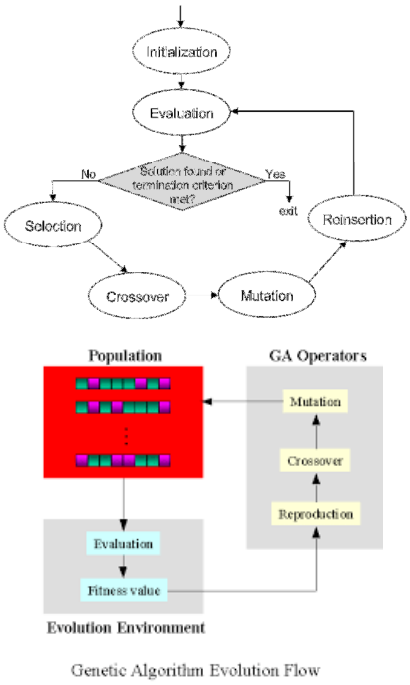
Loop and Conclusion

Loop and Stop Condition

- Now we sort the population again (which is now again 100)
- We calculate average fitness.
- We repeat reproduction.
- We stop whenever the improvement in average fitness is less than a specific percentage

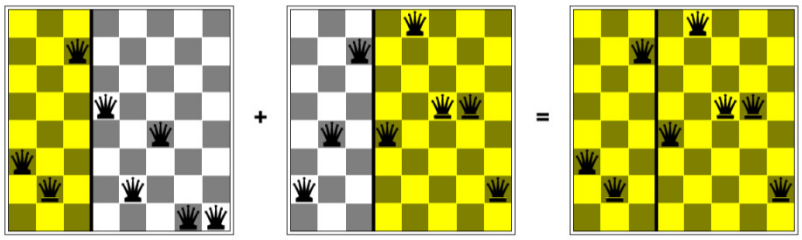
Conclusion

- Now we sort again and select the best individual (the most fit) and use it as the answer



Genetic Algorithm Evolution Flow

Example: N-Queens



- Why does crossover make sense here?
- When wouldn't it make sense?
- What would mutation be?
- What would a good fitness function be?