COMP 424 - Artificial Intelligence
Lecture 5: Constraint Satisfaction Problems

Instructor: Jackie CK Cheung (jcheung@cs.mcgill.ca)

Readings: R&N Ch 6

Based on slides by Joelle Pineau

Change in Office Hours Today

- My office hours today are 1pm – 3pm in MC 108N
- HW1 due in a week. Don’t start it late!

Quick recap

- **Constructive methods**: Start from scratch and build up a solution.
  - Informed / uninformed methods.

- **Iterative improvement/repair methods**: Start with a solution (which may be broken / suboptimal) and improve it.
  - Hill-climbing, simulated annealing.

Search for optimization problems:

- **Constructive methods**: Start from scratch and build up a solution.
    - Informed / uninformed methods.
- **Iterative improvement/repair methods**: Start with a solution (which may be broken / suboptimal) and improve it.
    - Hill-climbing, simulated annealing.
- **Global search**: Start from multiple states that are far apart, and go all around the state space.

Evolutionary computing

- Refers generally to computational procedures patterned after biological evolution.
- Many solutions (individuals) exist in parallel.
- Nature looks for the best individual (i.e. the fittest).

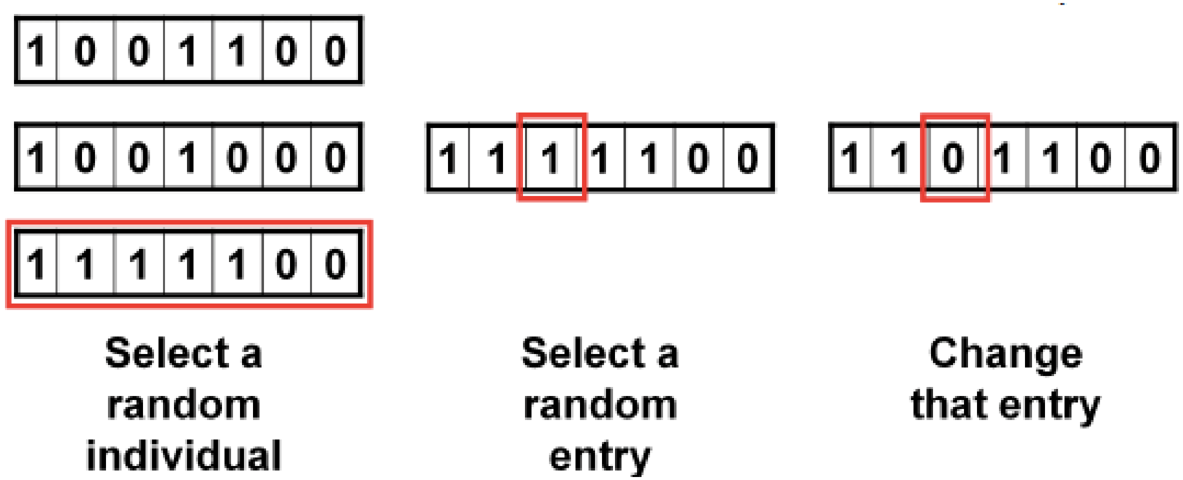
- Evolutionary search procedures are also parallel, perturbing probabilistically several potential solutions.

Genetic algorithms

- A candidate solution is called an individual.
  - In a traveling salesman problem, an individual is a tour
- Each individual has a fitness
  - fitness = numerical value proportional to quality of that solution
- A set of individuals is called a population.
- Populations change over generations, by applying operations to individuals.
  - operations = {mutation, crossover, selection}
- Individuals with higher fitness are more likely to survive & reproduce.
- Individual typically represented by a binary string:
  - allows operations to be carried out easily.

Mutation

- A way to generate desirable features that are not present in the original population by injecting random change.
  - Typically mutation just means changing a 0 to a 1 (and vice versa).
- The mutation rate controls prob. of mutation occurring
- We can allow mutation in all individuals, or just in the offspring.



Crossover

- Combine parts of individuals to create new individuals.
- Single-point crossover:
  - Choose a crossover point, cut individuals there, swap the pieces.

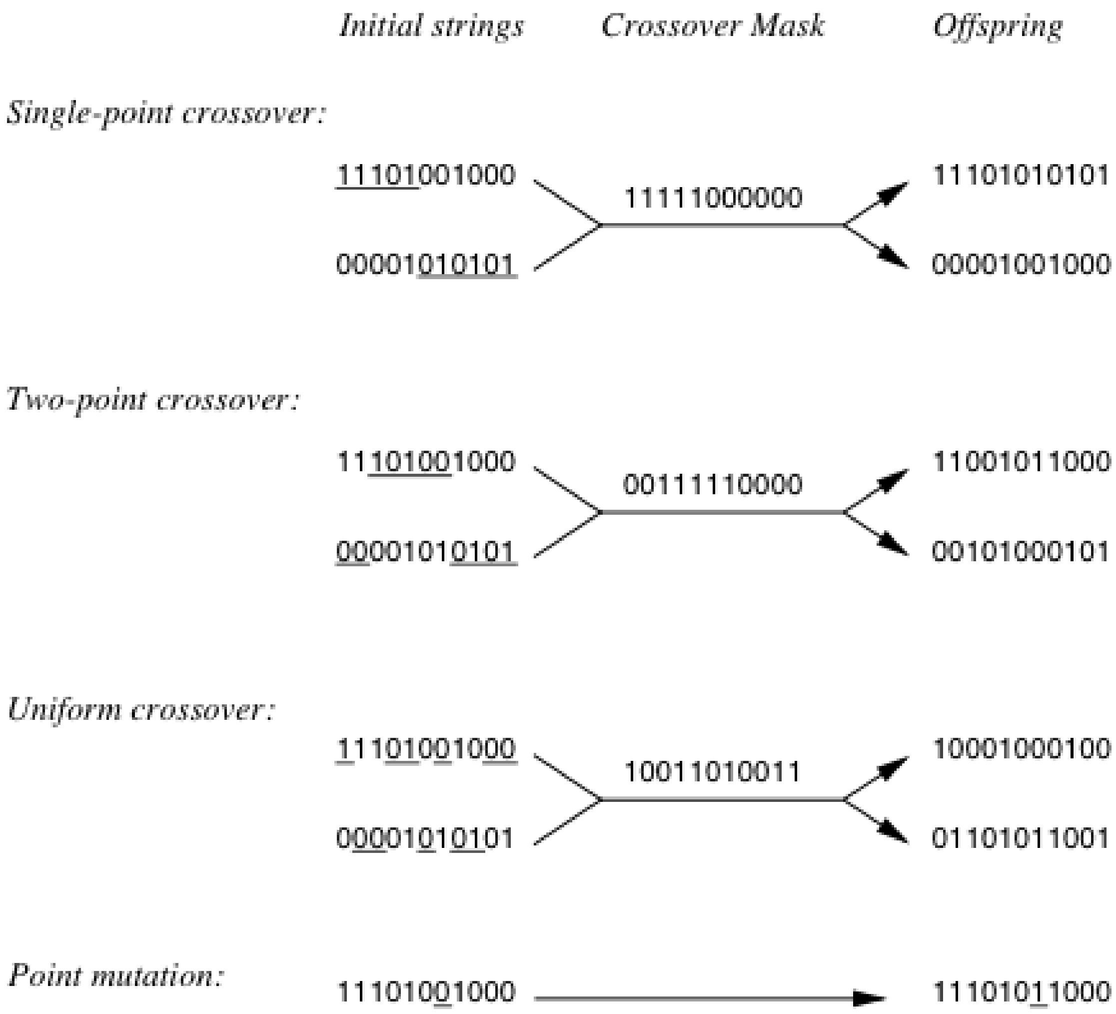
    E.g. **101|0101** → **101|1110**

    011|1110 → **011|0101**
- Implementation:
  - Use a crossover mask, which is a binary string

    E.g. mask = 1110000

- Multi-point crossover can be implemented with arbitrary mask.

Encoding operators as binary masks



Typical genetic algorithm

GA(Fitness, threshold, p, r, m)

- Initialize: $P \leftarrow p$ random individuals
- Evaluate: for each $h \in P$, compute *Fitness(h)*
- While max_h *Fitness(h) < threshold*
  - Select: Probabilistically select *(1-r)p* members of *P* to include in P_s
  - Crossover: Probabilistically select *rp/2* pairs of individuals from *P*.

    For each pair (h_1, h_2), produce two offspring by applying a crossover operator. Include all offspring in P_s.
  - Mutate: Invert a randomly selected bit in *m*p* randomly selected members of P_s
  - Update: $P \leftarrow P_s$
  - Evaluate: for each $h \in P$, compute *Fitness(h)*
- Return the individual from *P* that has the highest fitness.

Selection: Survival of the fittest

- As in natural evolution, fittest individuals are more likely to survive.
- Several ways to implement this idea:
  1. Fitness proportionate selection:

     $$P(i) = \frac{Fitness(i)}{\sum_{j=1}^{p} Fitness(j)}$$

     Can lead to crowding (multiple copies being propagated).

  2. Tournament selection:

     Pick *i, j* at random with uniform probability. With prob *p* select the fitter one. Only requires comparing two individuals.

  3. Rank selection:

     Sort all hypothesis by fitness. Probability of selection is proportional to rank.

  4. Softmax (Boltzman) selection:

     $$P(i) = \frac{e^{Fitness(i)/T}}{\sum_{j=1}^{p} e^{Fitness(j)/T}}$$

Elitism

- The best solution can "die" during evolution
- In order to prevent this, the best solution ever encountered can always be "preserved" on the side
- If the "genes" from the best solution should always be present in the population, it can also be copied in the next generation automatically, bypassing the selection process.
- **Note that the best solution ever encountered is typically saved in hill climbing and simulated annealing as well.**

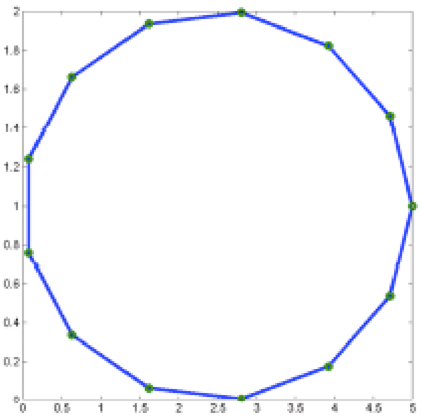
Genetic algorithms as search

- **States**: possible solutions
- **Search operators**: mutation, crossover, selection
- Relation to previous search algorithms:
  - Parallel search, since several solutions are maintained in parallel
  - Hill-climbing on the fitness function, but without following the gradient
  - Mutation and crossover should allow us to get out of local minima
  - Very related to simulated annealing.

Example: Solving TSP with a GA

- Each individual is a tour.
- Mutation swaps a pair of edges (many other operations are possible and have been tried in literature.)
- Crossover cuts the parents in two and swaps them. Reject any invalid offsprings.
- Fitness is the length of the tour.
- Note that GA operations (crossover and mutation) described here are fancier that the simple binary examples given before.

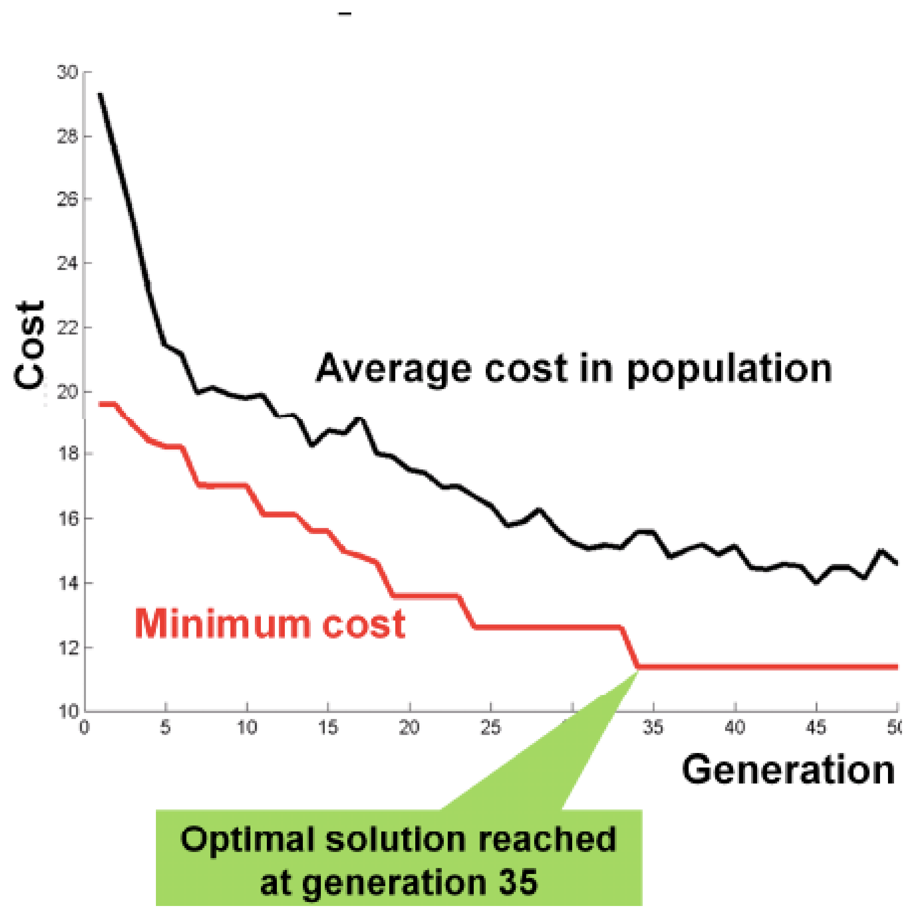
Example: Solving TSP with a GA



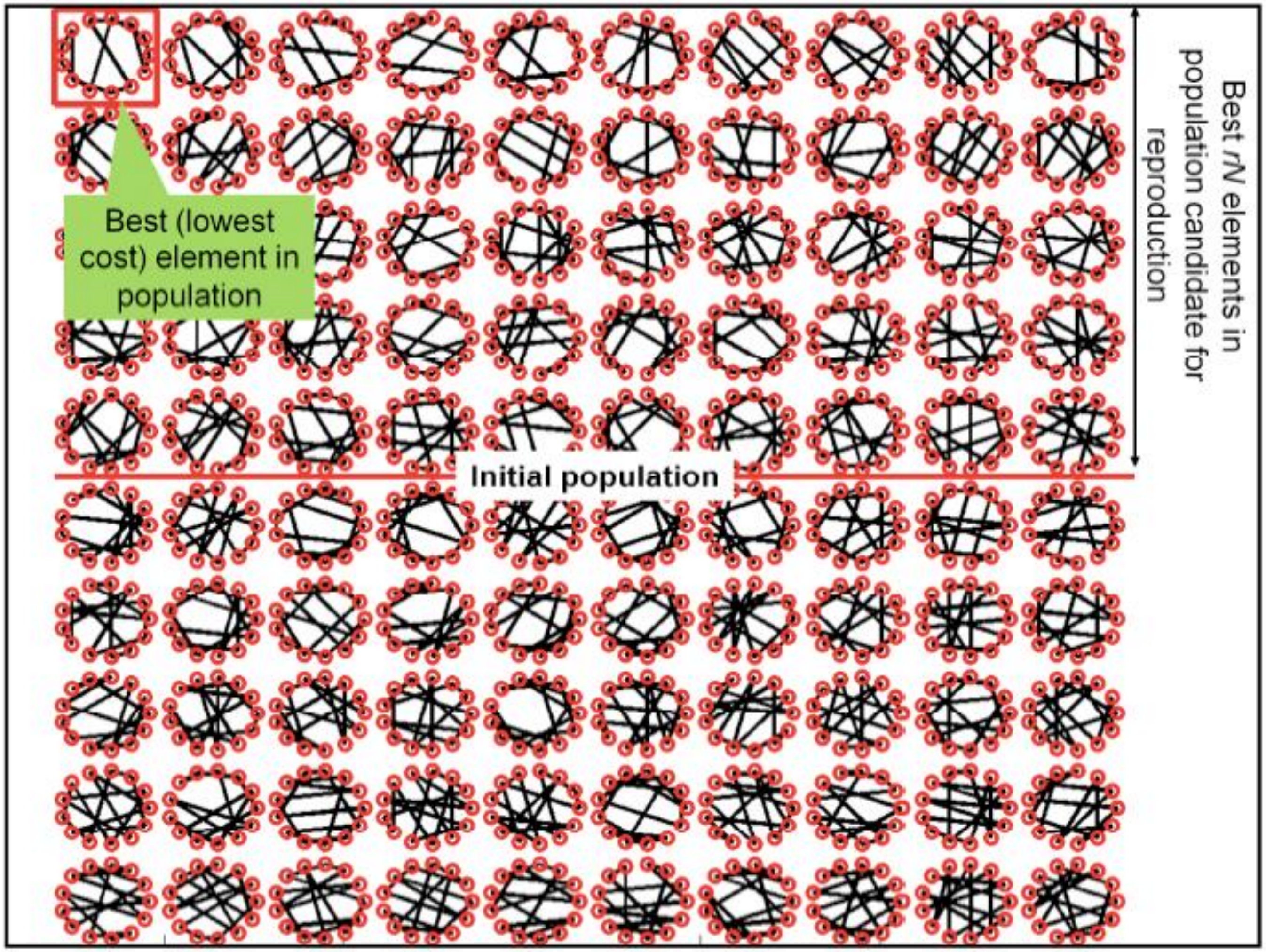
N = 13

P = 100 elements in population

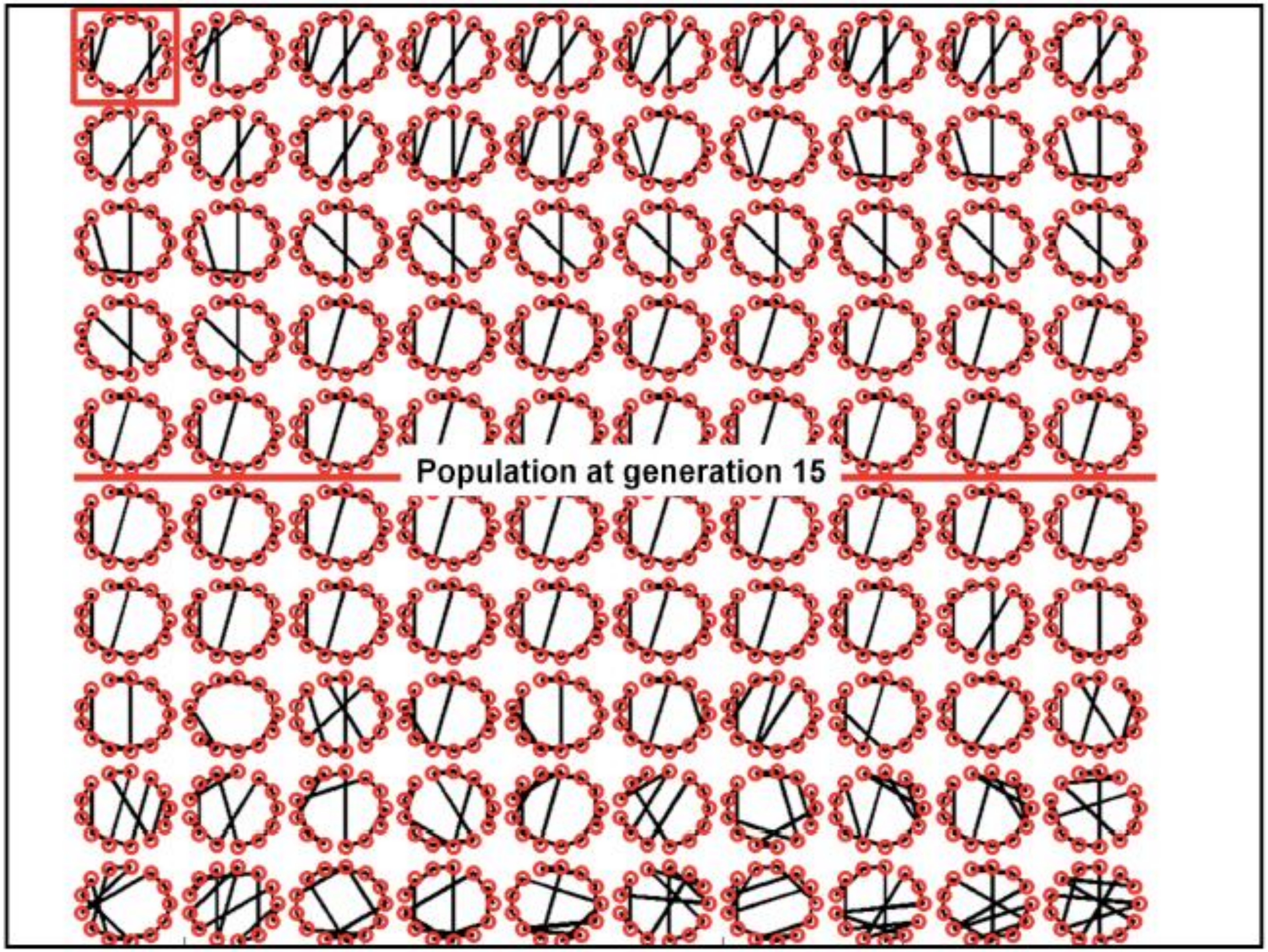
μ = 4% mutation rate
r = 50% reproduction rate



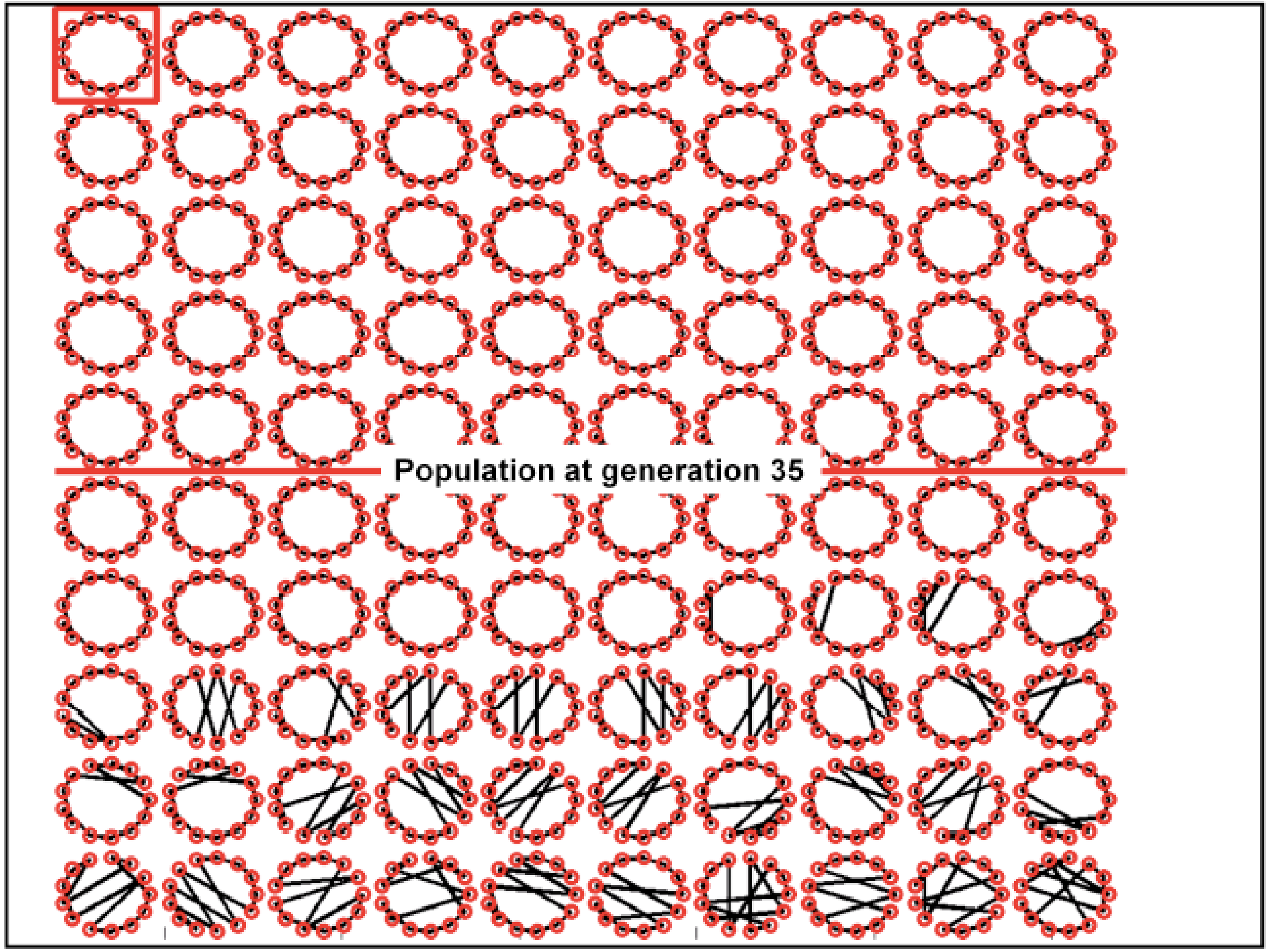
TSP example: Initial generation



TSP example: Generation 15



TSP example: Generation 30



The good and bad of GAs

- Good:
  - Intuitively appealing, due to evolution analogy.
  - If tuned right, can be very effective (good solution with few steps.)
- Bad:
  - Performance depends crucially on the problem encoding. Good encodings are difficult to find!
  - Many parameters to tweak! Bad parameter settings can result in very slow progress, or the algorithm is stuck in local minima.
  - With mutation rate is too low, can get overcrowding (many copies of the identical individuals in the population).

Next topic: Searching with Constraints

Searching with constraints

- Many interesting problems have strict constraints:

  E.g. Must visit city A (to re-supply) before visiting city B (to sell).

- How can we incorporate this information in the search process?

  - At a minimum: Ensure the search will be limited to solutions that respect the constraints. Sometimes very few “legal solutions”.

  - **Ideally**: Use the constraints to narrow the search space.

Example

- Color a map so that no adjacent territories have the same color.

Constraint satisfaction problems (CSPs)

- A **CSP** is defined by:
  - Set of **variables** V_i, that can take values from domain D_i
  - Set of **constraints** specifying what combinations of values are allowed (for subsets of variables, eg. pairs of variables)
  - Constraints can be represented:
    - Implicitly, as a function, testing for the satisfaction of the constraint. E.g. $C_1 \neq C_2$
    - Explicitly, as a list of allowable values. E.g. (C_1=R, C_2=G), (C_1=G, C_2=R), (C_1=B, C_2=R), …
- A **CSP solution** is an assignment of values to variables such that all the constraints are true.
- We typically want to find ***any solution*** or find that there is ***no solution***.

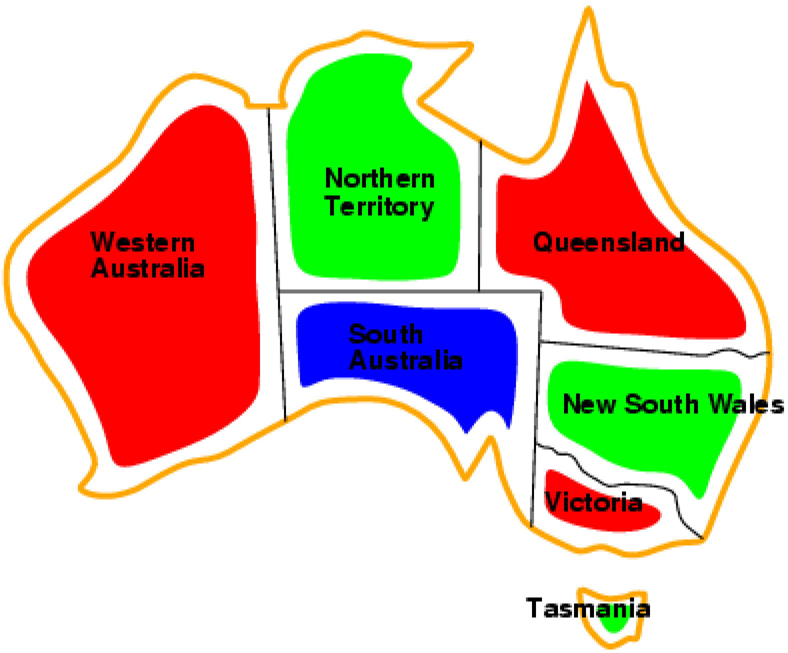
Example

- Variables *WA, NT, Q, NSW, V, SA, T*
- Domains D_i = {red,green,blue}

- Constraints: adjacent regions must have different colors

- E.g., WA ≠ NT

Example



- Solutions are complete and consistent assignments.

- E.g., WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = green

Varieties of variables

- Boolean variables (e.g. satisfiability)
- Finite domain, discrete variables (e.g. colouring)
- Infinite domain, discrete variables (e.g. start/end time of operation in scheduling)
- Continuous variables.

Problem complexity? Ranges from solvable in poly-time (e.g. linear programming) to NP-complete to undecidable.

Varieties of constraints

- Unary: involve one variable and one constraint.
- Binary.
- Higher-order (involve 3 or more variables).

Varieties of constraints

- Unary: involve one variable and one constraint.
- Binary.
- Higher-order (involve 3 or more variables).
- Relations:
  - $T_1 + d_1 \leq T_2$ (Task$_2$ has to come after Task$_1$) Scheduling
  - *Alldiff* (variables in a row), *Alldiff* (variables in a column), *Alldiff* (variables in a 3x3 square). Sudoku
- Preferences (soft constraints): can be represented using costs and lead to constrained optimization problems.

Real-world CSPs

Often involves allocating *limited* resources:

- Timetable problems (e.g. which class is offered when, where)
- Hardware configuration.
- Transportation scheduling. Factory scheduling. Floor planning.
- Puzzle solving (crosswords, Sudoku)

Overview of approaches for solving CSPs

- **Constructive approach**
  - State is defined by the set of values assigned so far.
  - Apply forward search to fill the solution.
  - This is a general purpose algorithm which works for all CSPs.

- **Random approach**
  - Start with a broken but complete assignment of values to variables.
  - Gradually fix broken constraints by re-assigning variables.
  - Essentially use optimization approaches (hill-climbing, simulated annealing).

Constructive search for CSPs

- Problem definition:
  - State: defined by set of values assigned so far, could be *partial* and/or *inconsistent* assignment
  - Initial state: all variables are unassigned.
  - Operators: assign a value to an unassigned variable.
  - Goal test: all variables assigned, no constraint violated. i.e. *complete* and *consistent* assignment

Constructive search for CSPs

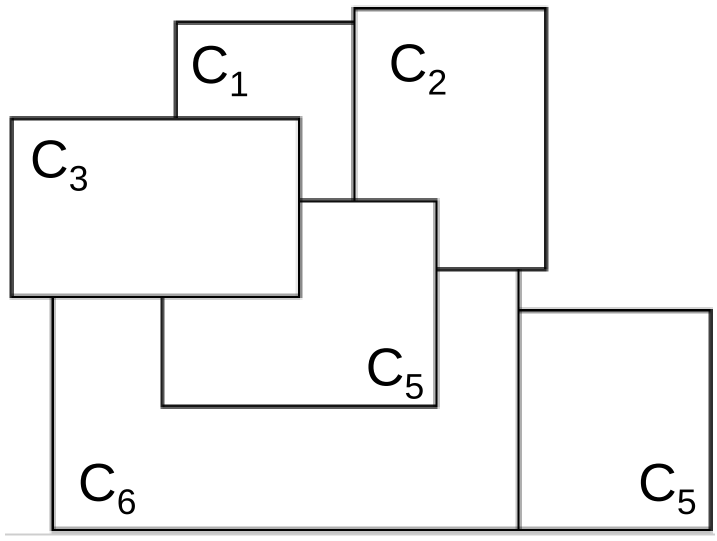
- Problem definition:
  - State: defined by set of values assigned so far, could be *partial* and/or *inconsistent* assignment
  - Initial state: all variables are unassigned.
  - Operators: assign a value to an unassigned variable.
  - Goal test: all variables assigned, no constraint violated. i.e. *complete* and *consistent* assignment

- Problem has deterministic action, fully observable state.

  Important observation: Depth is limited to the number of variables, *n*. So we can apply DFS (or depth-limited search)

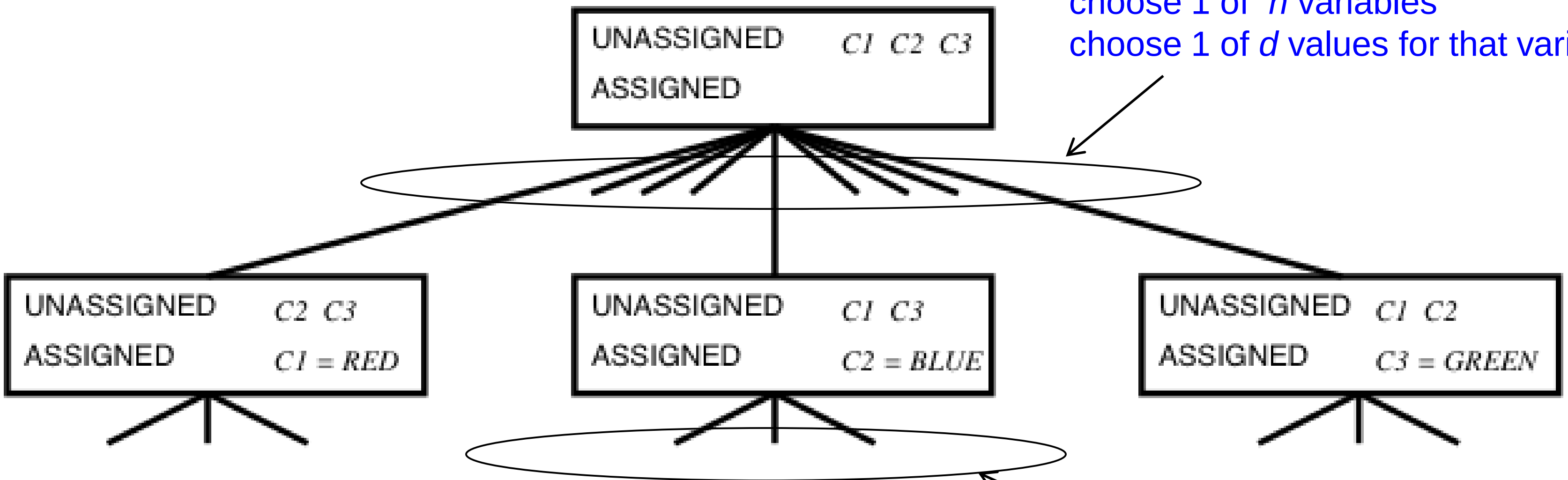
Example

- Color abstract map so that adjacent countries don't same color.
  - Variables: Countries C_i
  - Domains: {Red, Blue, Green}
  - Constraints: $\{C_1 \neq C_2, C_1 \neq C_5, \ldots\}$



Standard uninformed search for map coloring

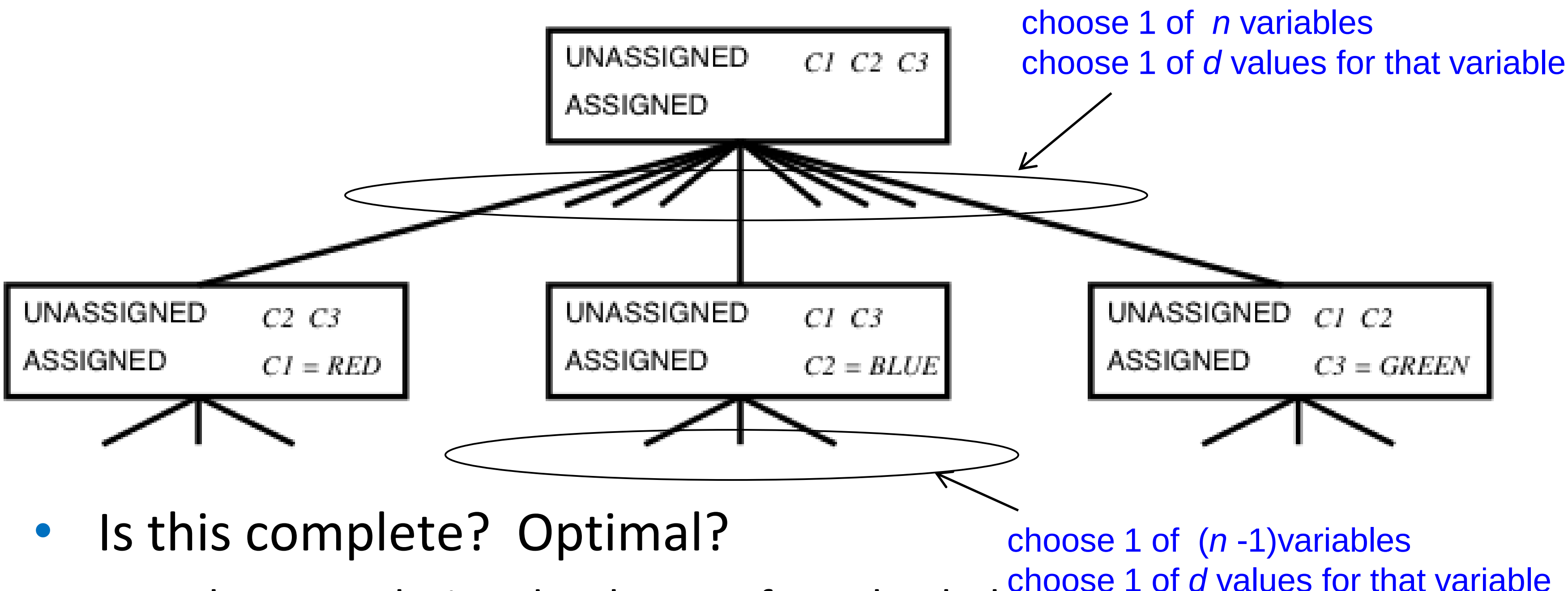
choose 1 of *n* variables
choose 1 of *d* values for that variable



choose 1 of (*n* -1)variables
choose 1 of *d* values for that variable

- Is this complete? Optimal?
- Is this a practical approach? What is the complexity?

Standard uninformed search for map coloring



- Is this complete? Optimal?

  Yes: known solution depth. Yes: If we check the constraints.

- Is this a practical approach? What is the complexity?

  $[n \times d] \times [(n-1) \times d] \times [(n-2) \times d] \times \ldots\ldots \times [2 \times d] \times d = n! \cdot d^n$

Analysis of the simple approach

Branching factor is very high: $\sum_i d$ (*i* sums over unassigned variables).

BUT: There can be only d^n unique complete assignments.

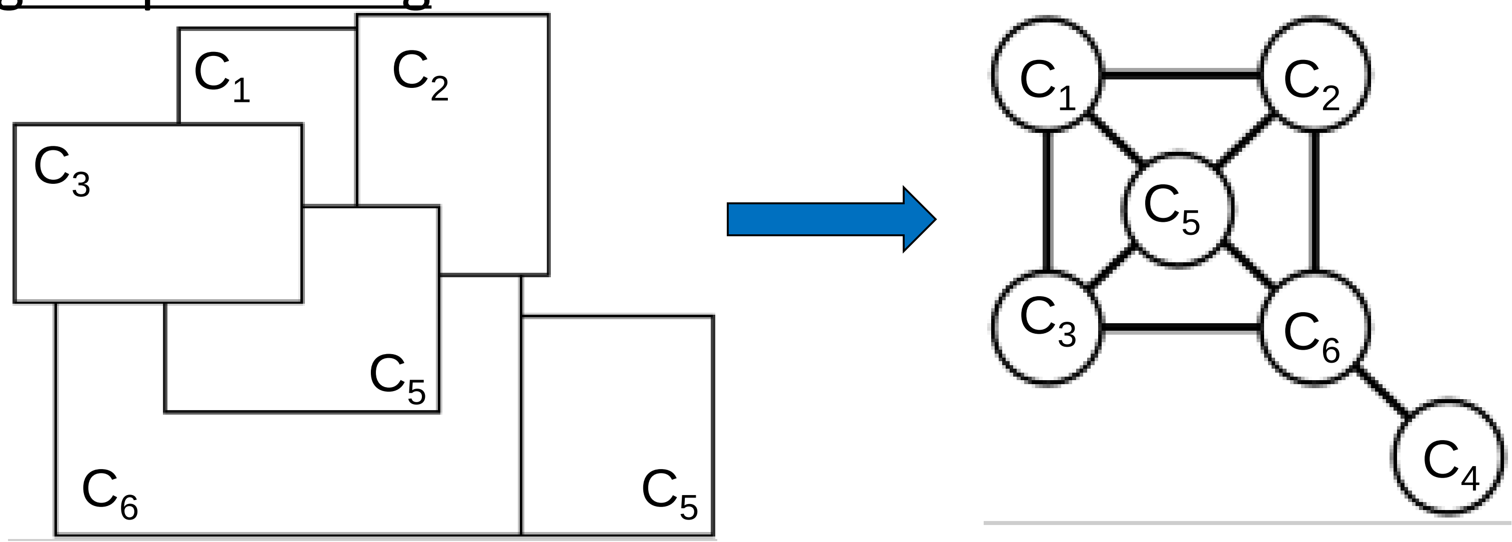
More important observations:

- Order in which variables are assigned is irrelevant -> Many paths are equivalent!
- Adding assignments cannot correct a violated constraint!

Constraint graph

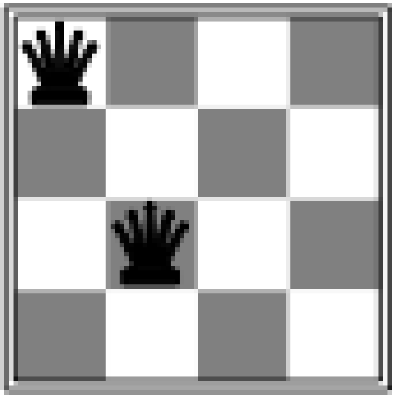
- Need to reason about constraints
- Constraint graph: nodes are variables, arcs show constraints.
- Graph structure can be exploited to accelerate solution search.

<u>E.g. Map colouring:</u>



Exercise: 4 queens problem

- Put 4 queens on 4x4 board so that none attack each other:

  Partial assignment: 

- Formulate this as a CSP by defining:
  - Variables and domains
  - Constraints
- Draw the constraint graph

4 Queens Problem

Put one queen per column. Let value indicate row of each queen.

- Variables: $\{Q_1, Q_2, Q_3, Q_4\}$
- Domain (same for all variables): $\{1, 2, 3, 4\}$
- Constraints:

  $Q_i \neq Q_j$ (cannot be in same row)

  $|Q_i - Q_j| \neq |i - j|$ (cannot be in same diagonal)

- Can also translate each constraint into set of allowable values for its variables:
  - Values for (Q_1, Q_2): $(Q_1 = 1, Q_2 = 3)$,$(Q_1 = 1, Q_2 = 4)$,$(Q_1 = 2, Q_2 = 4)$ etc.

4 Queens Problem

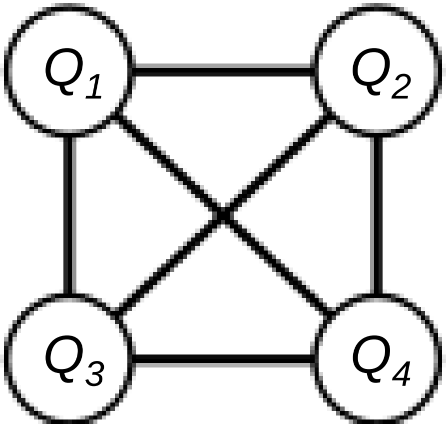
Put one queen per column. Let value indicate row of each queen.

- Variables: $\{Q_1, Q_2, Q_3, Q_4\}$
- Domain (same for all variables): $\{1, 2, 3, 4\}$
- Constraints:

  $Q_i \neq Q_j$ (cannot be in same row)

  $|Q_i - Q_j| \neq |i - j|$ (cannot be in same diagonal)

- Constraint graph:

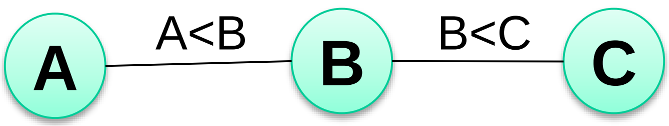


Using constraint graph

- Perform **inference** using constraint graph in order to reduce search space
- **Idea**: Pre-process the graph to remove obvious inconsistencies
  - A **variable** is **arc-consistent** if every value in its domain satisfies that variable's binary constraints.
  - A **network** is **generalized arc-consistent** if every value in the domain of every variable are simultaneously arc-consistent.

Example: consistency constraints

A CSP with variables A, B, C, each with domain {1, 2, 3, 4}:



Arc-consistent variables:

A={1, 2, 3}; B={2, 3}; C={2, 3, 4}

Generalized arc-consistent variables:

A={1, 2}; B={2, 3}; C={3, 4}

How does this help us? Reduced domain = reduced search.

Algorithms: **AC-3** (R&N 6.2)

Backtracking search

Like Depth-First Search, but **fix order of variable assignment** (so $b=|D_i|$).

Backtracking search

Like Depth-First Search, but **fix order of variable assignment** (so $b=|D_i|$).

Algorithm:

- Select an unassigned variable, X.
- For each $value=\{x_1,\ldots, x_n\}$ in the domain of that variable
  - If the value satisfies the constraints, let $X = x_i$ and exit the loop.
- If an assignment was found, move to the next variable.
- If no assignment, go back to preceding variable and try different value.

- This is the basic uninformed algorithm for CSPs.
  - Can solve n-queens for $n \approx 25$.

Forward checking

Idea: Keep track of legal values for unassigned variables.

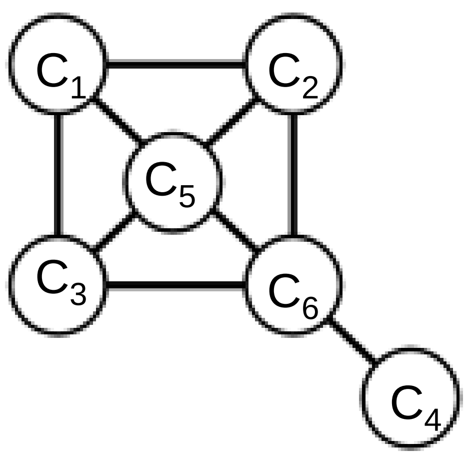
- When you assign a variable X
  - look at each unassigned variable Y connected to X (by a constraint)
  - delete from Y's domain any value that is inconsistent the value of X
- Can solve n-queens for $n \approx 30$.

Forward checking

Idea: Keep track of legal values for unassigned variables.

- When you assign a variable X
  - look at each unassigned variable Y connected to X (by a constraint)
  - delete from Y's domain any value that is inconsistent the value of X

E.g. Map coloring.



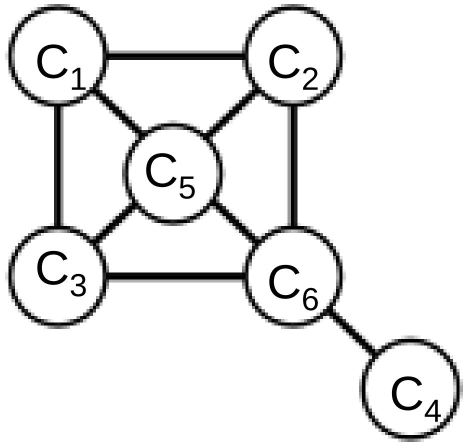
	Nothing assigned		
C1	RGB		
C2	RGB		
C3	RGB		
C4	RGB		
C5	RGB		
C6	RGB		

Forward checking

Idea: Keep track of legal values for unassigned variables.

- When you assign a variable X
  - look at each unassigned variable Y connected to X (by a constraint)
  - delete from Y's domain any value that is inconsistent the value of X

E.g. Map coloring.



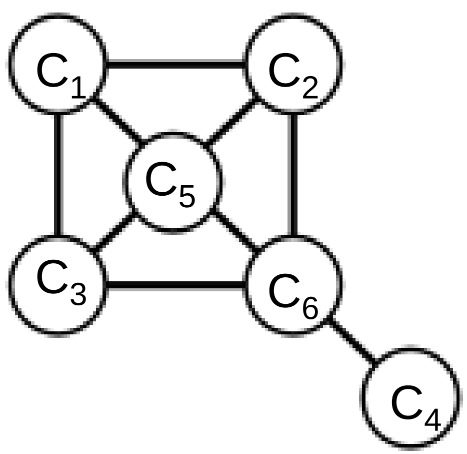
	Nothing assigned	Assign C1 = Red	
C1	RGB	R	
C2	RGB	GB	
C3	RGB	GB	
C4	RGB	RGB	
C5	RGB	GB	
C6	RGB	RGB	

Forward checking

Idea: Keep track of legal values for unassigned variables.

- When you assign a variable X
  - look at each unassigned variable Y connected to X (by a constraint)
  - delete from Y's domain any value that is inconsistent the value of X

E.g. Map coloring.



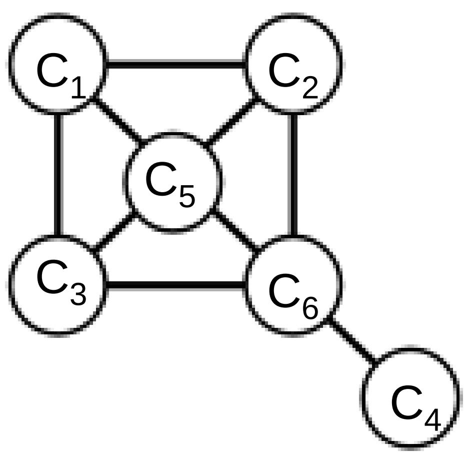
	Nothing assigned	Assign C1 = Red	Assign C2 = G
C1	RGB	R	R
C2	RGB	GB	G
C3	RGB	GB	GB
C4	RGB	RGB	RGB
C5	RGB	GB	B by forward checking!
C6	RGB	RGB	RB

Forward checking

Idea: Keep track of legal values for unassigned variables.

- When you assign a variable X
  - look at each unassigned variable Y connected to X (by a constraint)
  - delete from Y's domain any value that is inconsistent the value of X

E.g. Map coloring.



	Nothing assigned	Assign C1 = Red	Assign C2 = G
C1	RGB	R	R
C2	RGB	GB	G
C3	RGB	GB	GB
C4	RGB	RGB	RGB
C5	RGB	GB	B by forward checking!
C6	RGB	RGB	RB

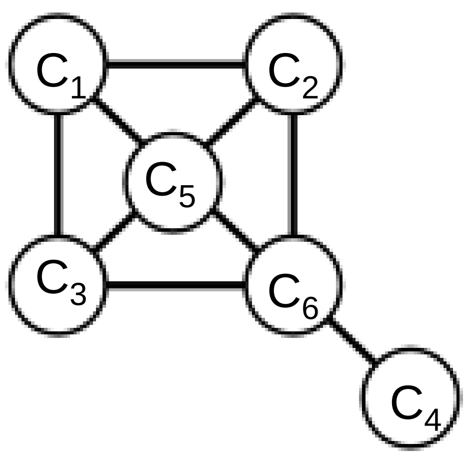
Can also apply **generalized arc-consistency**!

Forward checking

Idea: Keep track of legal values for unassigned variables.

- When you assign a variable X
  - look at each unassigned variable Y connected to X (by a constraint)
  - delete from Y's domain any value that is inconsistent the value of X

E.g. Map coloring.



	Nothing assigned	Assign C1 = Red	Assign C2 = G
C1	RGB	R	R
C2	RGB	GB	G
C3	RGB	GB	~~GB~~ G
C4	RGB	RGB	RGB
C5	RGB	GB	B
C6	RGB	RGB	~~RB~~ R

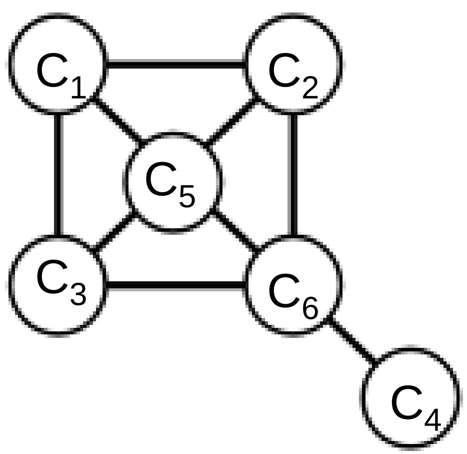
Can also apply **generalized arc-consistency**!

Forward checking

Idea: Keep track of legal values for unassigned variables.

- When you assign a variable X
  - look at each unassigned variable Y connected to X (by a constraint)
  - delete from Y's domain any value that is inconsistent the value of X

E.g. Map coloring.



	Nothing assigned	Assign C1 = Red	Assign C2 = G
C1	RGB	R	R
C2	RGB	GB	G
C3	RGB	GB	~~GB~~ G
C4	RGB	RGB	~~RGB~~ GB
C5	RGB	GB	B
C6	RGB	RGB	~~RB~~ R

Can also apply **generalized arc-consistency**!

Heuristics for CSP search

- More intelligent decisions on:
  - Which variable to assign next?
  - Which value to choose for the variable?

E.g. Sudoku:

	2		4	5	6	7	8	9
4	5	7		8		2	3	6
6	8	9	2	3	7		4	
		5	3	6	2	9	7	4
2	7	4		9		6	5	3
3	9	6	5	7	4	8		
	4		6	1	8	3	9	7
7	6	1		4		5	2	8
9	3	8	7	2	5		6	

Common heuristics (for faster search)

- To select a variable:
  1. Minimum-remaining values: Choose the variable that is the most constrained (i.e. fewest legal values).

  2. Degree heuristic: Choose the variable that imposes the most constraints on the remaining variables.
     - Use this to break ties from Minimum-remaining value heuristic

Common heuristics (for faster search)

- To select a variable:
    1. Minimum-remaining values: Choose the variable that is the most constrained (i.e. fewest legal values).

    2. Degree heuristic: Choose the variable that imposes the most constraints on the remaining variables.
        - Use this to break ties from Minimum-remaining value heuristic

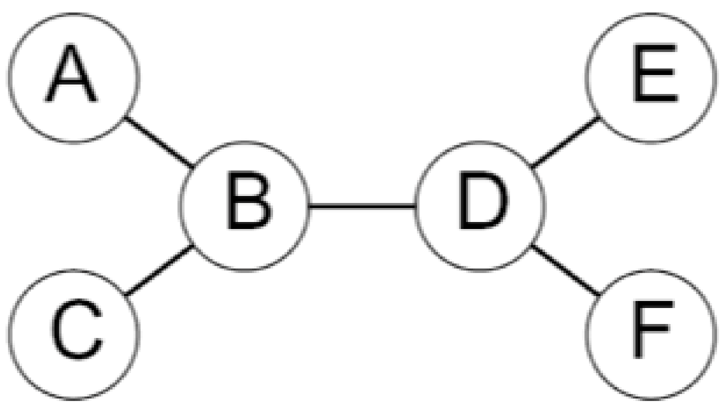
- To select a value:
    - Least-constraining value: Assign the value that rules out the fewest values for other variables.

Taking advantage of problem structure

- Worst-case complexity is d^n (where d is the number of possible values and n is the number of variables.
- But a lot of problems are much easier!
- Disjoint components can be solved independently.

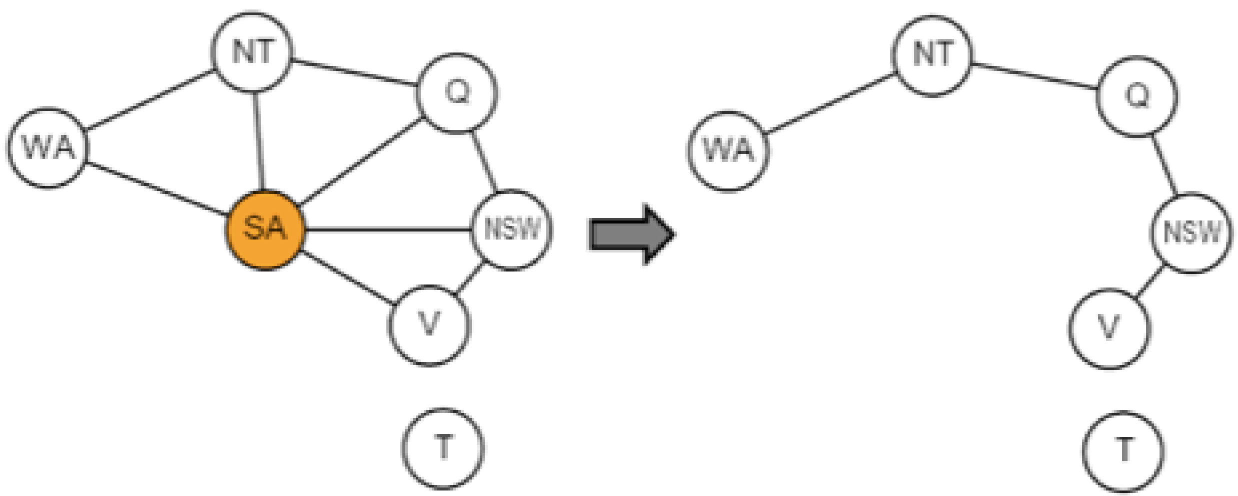
Taking advantage of problem structure

- Worst-case complexity is d^n (where d is the number of possible values and n is the number of variables.
- But a lot of problems are much easier!
- Disjoint components can be solved independently.
- Tree-structured constraint graph: complexity is $O(nd^2)$



Taking advantage of problem structure

- Nearly-tree structured graph: complexity is $O(d^c(n\text{-}c)d^2)$ using **cutset conditioning**:
  - Find a set of variables which, when removed, turn graph into tree.
  - Instantiate them all possible ways. Good if c (size of cutset) is small.



Key insight of CSP: Leverage structure to accelerate solving!

Local search for CSPs

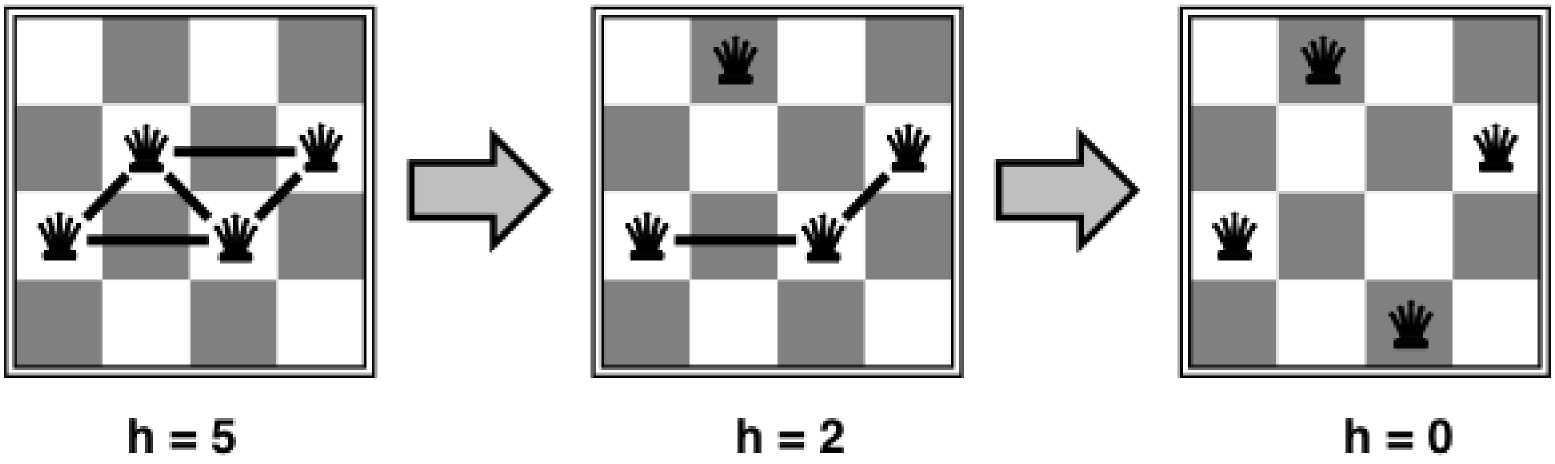
General idea: Iterative improvement algorithm

- Start with a broken but complete assignment of values to variables.
- Allow variable assignments that don't satisfy some constraints.
- Randomly select any conflicted variables.
- Operators reassign variable values.
  - **Min-conflicts heuristic** chooses value that violates the fewest number of constraints.

a.k.a. Hill-climbing optimization! (Could also use simulated annealing.)

Example: 4-Queens

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



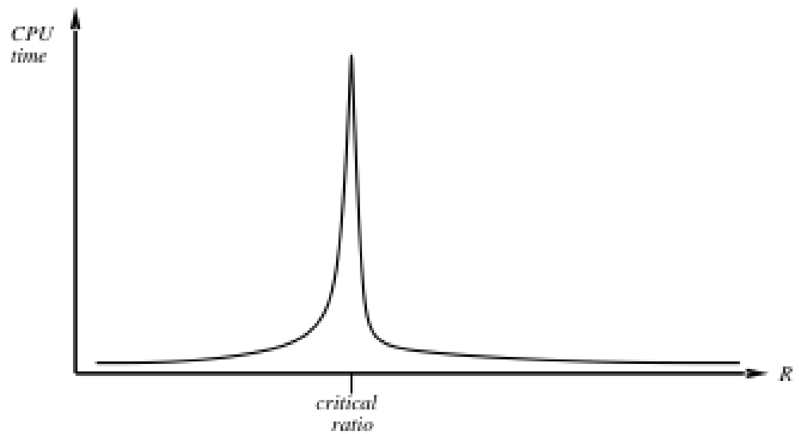
Performance of min-conflicts heuristic

- Given random initial state, can solve *n*-queens in almost constant time for arbitrary *n* with high probability (e.g. $n=10^7$). Why?

Performance of min-conflicts heuristic

- Given random initial state, can solve *n*-queens in almost constant time for arbitrary *n* with high probability (e.g. $n=10^7$). Why?
- The same appears to be true for many randomly-generated CSPs, <u>except</u> in a narrow range of the ratio:

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



Summary

- CSPs are everywhere. Be able to recognize them!
- Know how to cast CSP solving as a search problems.
- Understand basic concepts: constraint graph, arc consistency.
- Understand both constructive and iterative improvement methods to solve CSPs.
- Know how to apply the various heuristics:

  minimum-remaining-values, least-constraining value, degree
- Iterative improvement methods using min-conflict heuristic are very general and often work better.