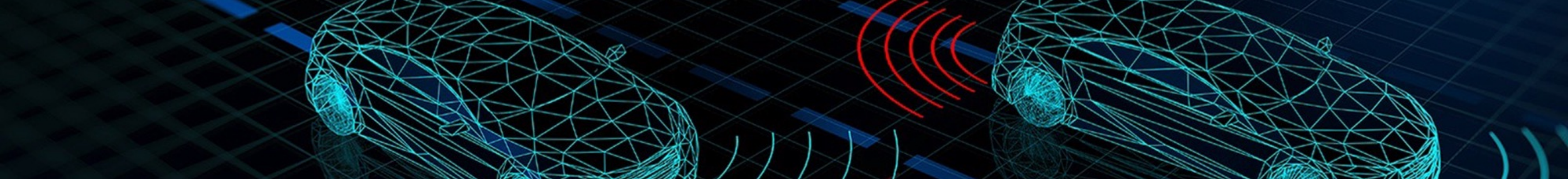
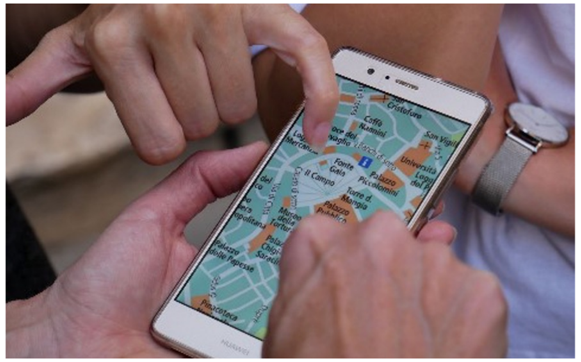


AI3603: Artificial Intelligence: Principles and Applications

Lecture 4 Constraint Satisfaction Problems

Yue Gao

Shanghai Jiao Tong University

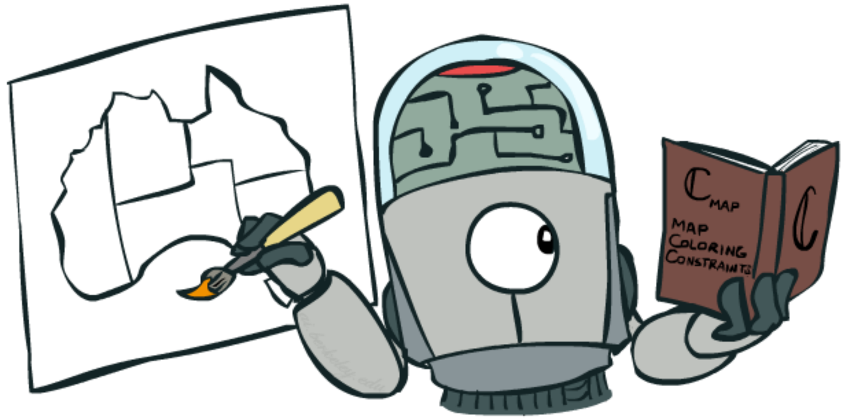


content

Constraint Satisfaction Problems

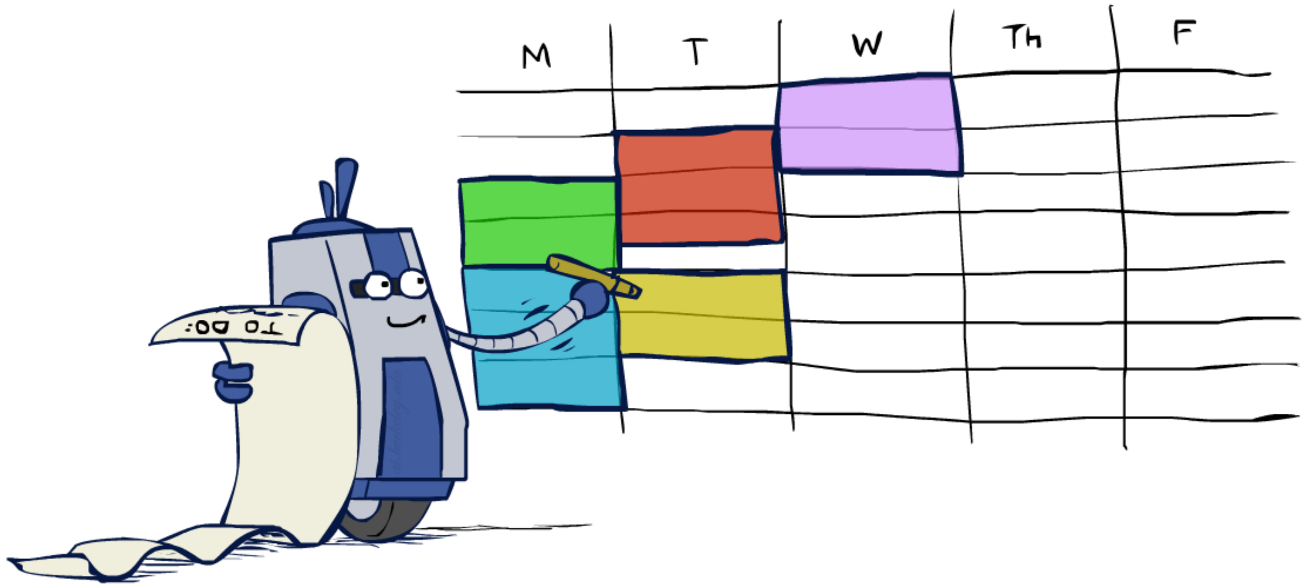
- Standard search problems:
  - Goal test can be any function over states
  - Successor function can also be anything

- Constraint satisfaction problems (CSPs):
  - A special subset of search problems
  - State is defined by variables X_i with values from a domain D (sometimes D depends on i)
  - Goal test is a set of constraints specifying allowable combinations of values for subsets of variables



Real-World CSPs

- Assignment problems: e.g., who teaches what class
- Timetabling problems: e.g., which class is offered when and where?
- Transportation scheduling
- Factory scheduling
- Circuit layout
- Fault diagnosis
- …



- Many real-world problems involve real-valued variables…

Improving Backtracking

- General-purpose ideas give huge gains in speed

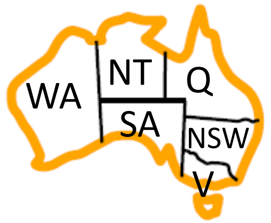
- Ordering:
  - Which variable should be assigned next?
  - In what order should its values be tried?

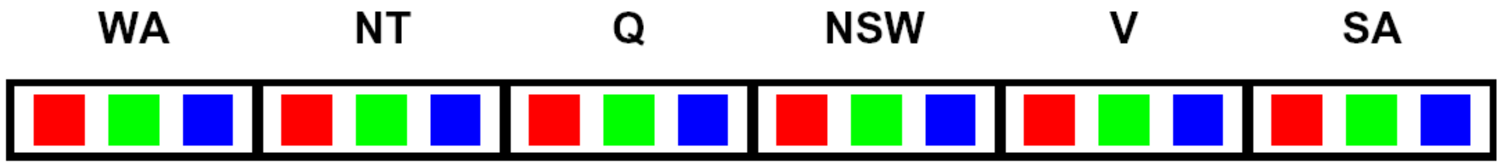
- Filtering: Can we detect inevitable failure early?

- Structure: Can we exploit the problem structure?

Filtering: Forward Checking

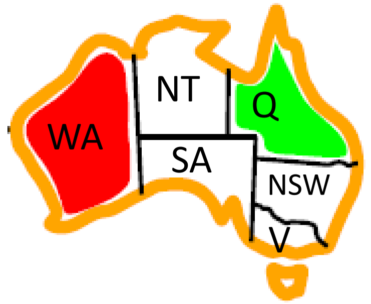
- Filtering: Keep track of domains for unassigned variables and cross off bad options
- Forward checking: Cross off values that violate a constraint when added to the existing assignment

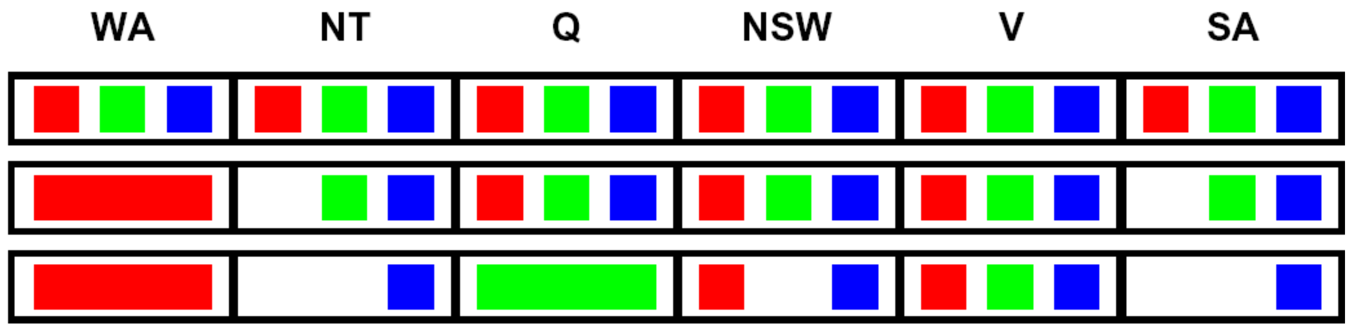




Filtering: Constraint Propagation

- Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures:

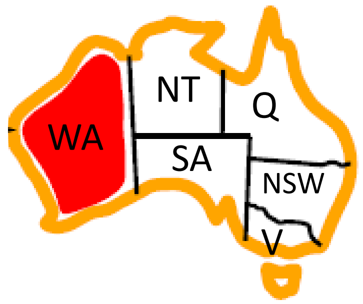


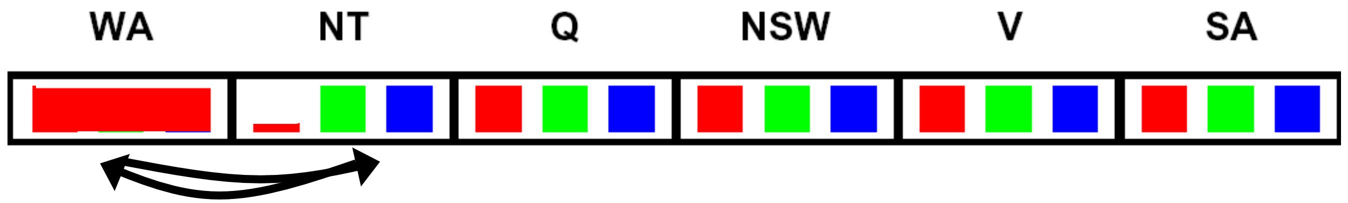


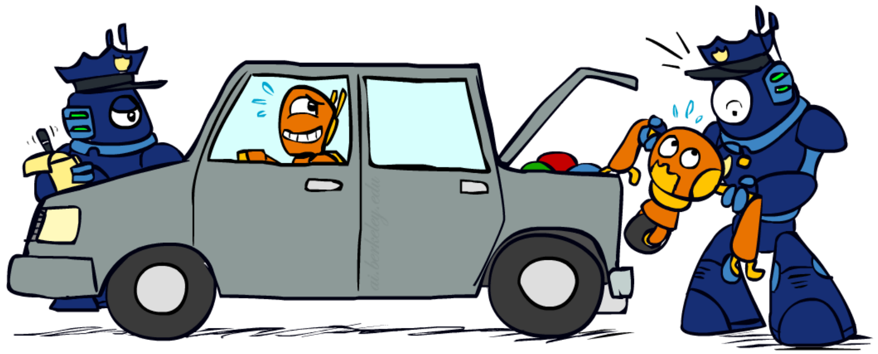
- NT and SA cannot both be blue!
- Why didn’t we detect this yet?
- *Constraint propagation:* reason from constraint to constraint

Consistency of A Single Arc

- An arc X $\rightarrow$ Y is consistent iff for *every* x in the tail there is *some* y in the head which could be assigned without violating a constraint





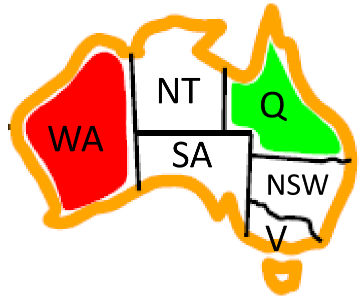


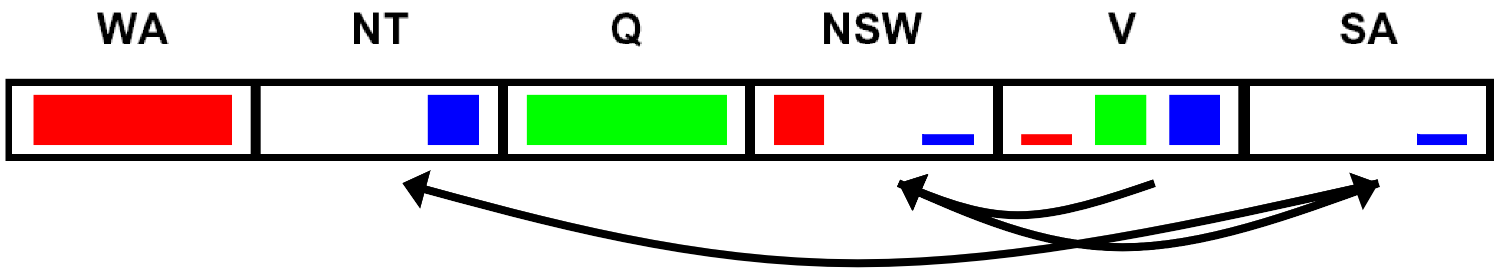
Delete from the tail!

- Forward checking: Enforcing consistency of arcs pointing to each new assignment

Arc Consistency of an Entire CSP

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





- Important: If X loses a value, neighbors of X need to be rechecked!
- Arc consistency detects failure earlier than forward checking
- Can be run as a preprocessor or after each assignment
- What's the downside of enforcing arc consistency?

Remember: Delete from the tail!

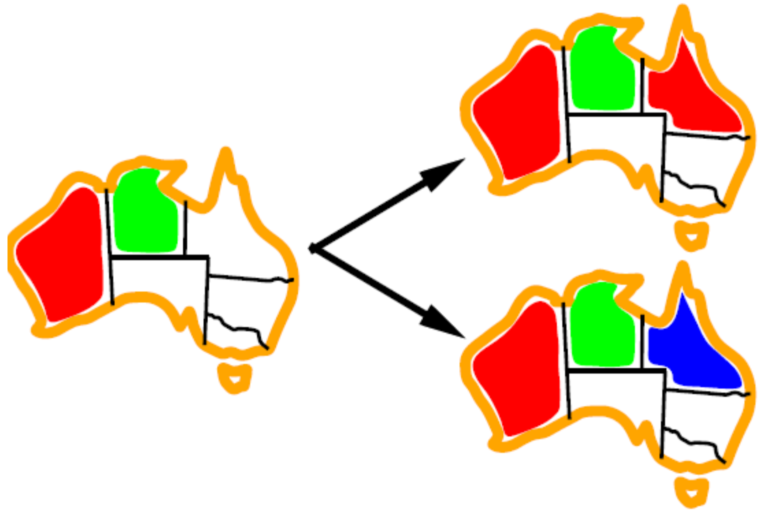
Ordering: Minimum Remaining Values

- Variable Ordering: Minimum remaining values (MRV):
  - Choose the variable with the fewest legal left values in its domain

- Why min rather than max?
- Also called “most constrained variable”
- “Fail-fast” ordering

Ordering: Least Constraining Value

- Value Ordering: Least Constraining Value
  - Given a choice of variable, choose the *least constraining value*
  - I.e., the one that rules out the fewest values in the remaining variables
  - Note that it may take some computation to determine this! (E.g., rerunning filtering)

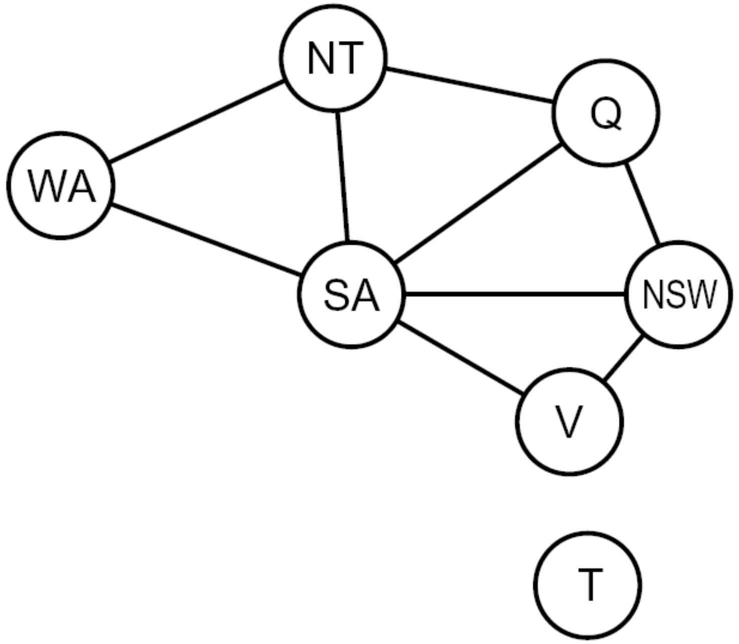


- Why least rather than most?

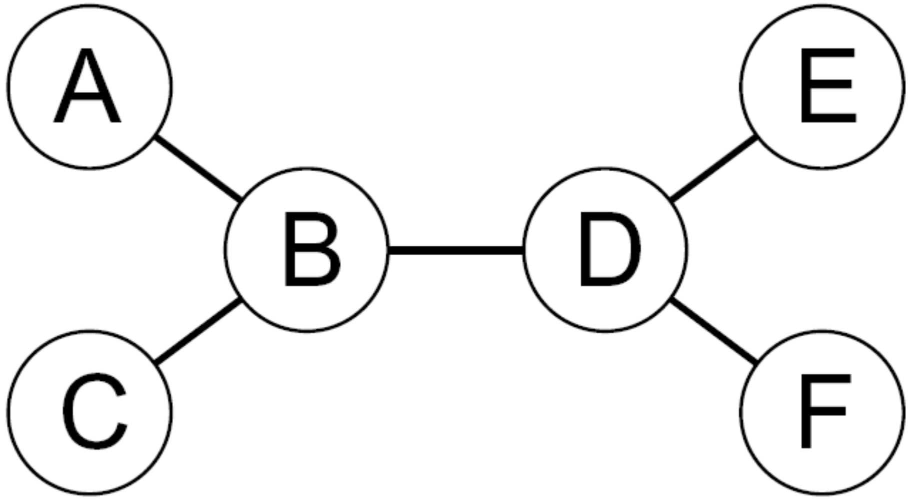
- Combining these ordering ideas makes 1000 queens feasible

Problem Structure

- Extreme case: independent subproblems
  - Example: Tasmania and mainland do not interact
- Independent subproblems are identifiable as connected components of constraint graph
- Suppose a graph of n variables can be broken into subproblems of only c variables:
  - Worst-case solution cost is $O((n/c)(d^c))$, linear in n
  - E.g., n = 80, d = 2, c =20
  - 2^{80} = 4 billion years at 10 million nodes/sec
  - $(4)(2^{20})$ = 0.4 seconds at 10 million nodes/sec



Tree-Structured CSPs

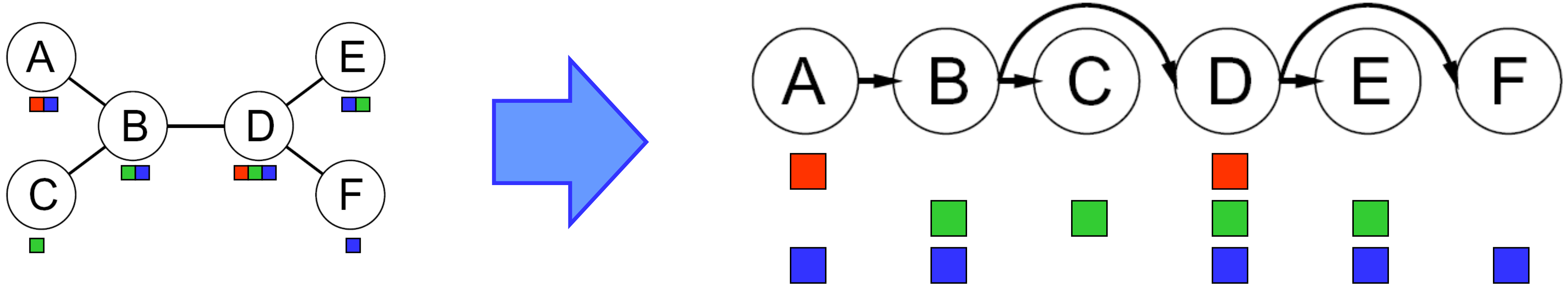


- Theorem: if the constraint graph has no loops, the CSP can be solved in $O(n\ d^2)$ time
  - Compare to general CSPs, where worst-case time is $O(d^n)$

- This property also applies to probabilistic reasoning: an example of the relation between syntactic restrictions and the complexity of reasoning

Tree-Structured CSPs

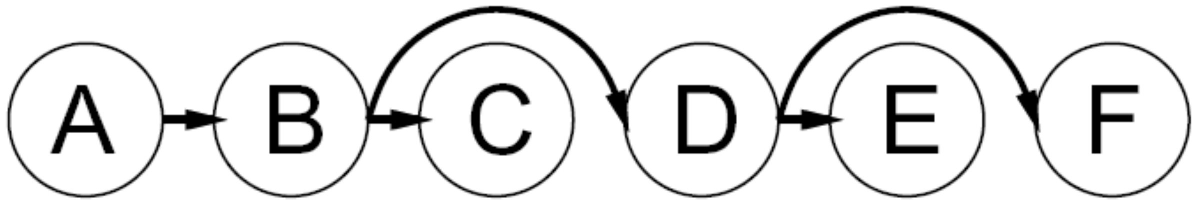
- Algorithm for tree-structured CSPs:
  - Order: Choose a root variable, order variables so that parents precede children



  - 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)
- Runtime: O(n d^2)

Tree-Structured CSPs

- Claim 1: After 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)

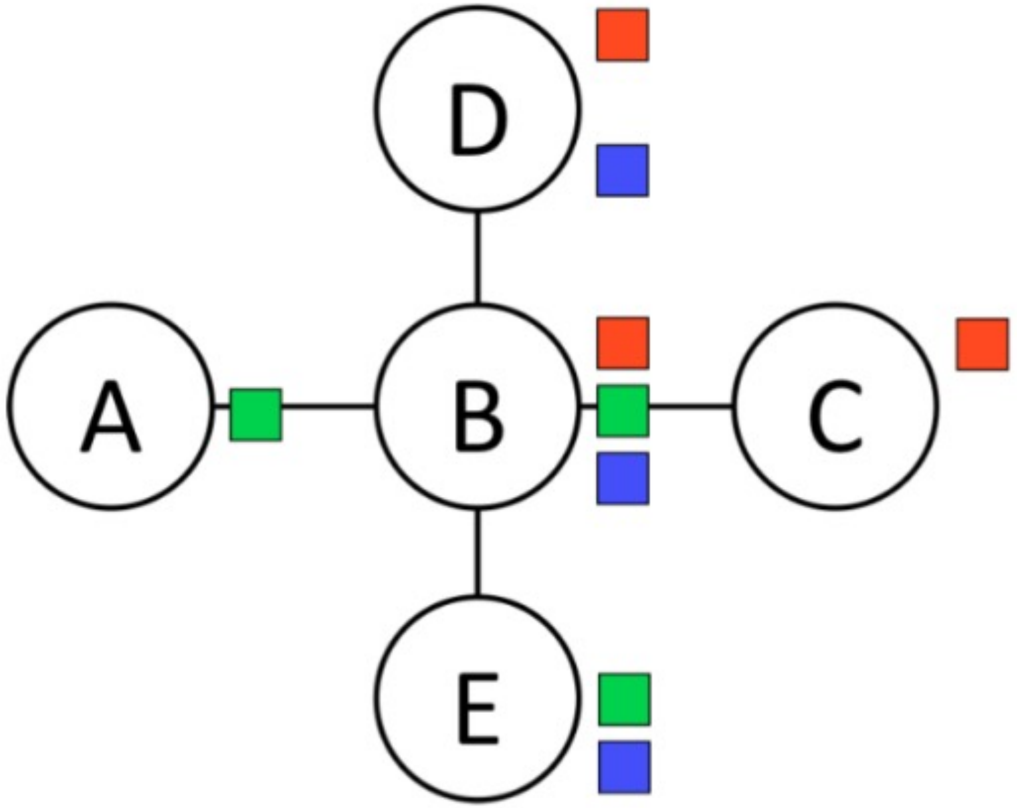


- Claim 2: If root-to-leaf arcs are consistent, forward assignment will not backtrack
- Proof: Induction on position

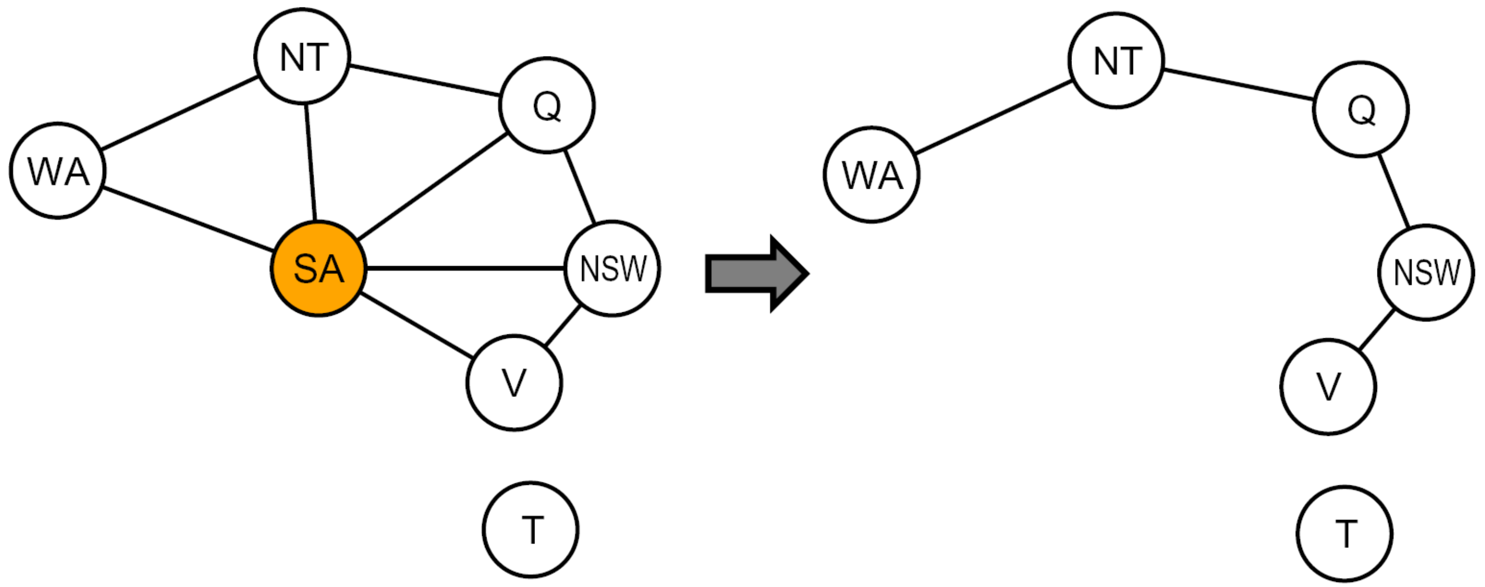
- Why doesn't this algorithm work with cycles in the constraint graph?

Tree-Structured CSPs

- A as the root node



Nearly Tree-Structured CSPs



- Conditioning: instantiate a variable, prune its neighbors' domains
- Cutset conditioning: instantiate (in all ways) a set of variables such that the remaining constraint graph is a tree
- Complexity?
- Cutset size c gives runtime $O((d^c) (n-c) d^2)$, very fast for small c

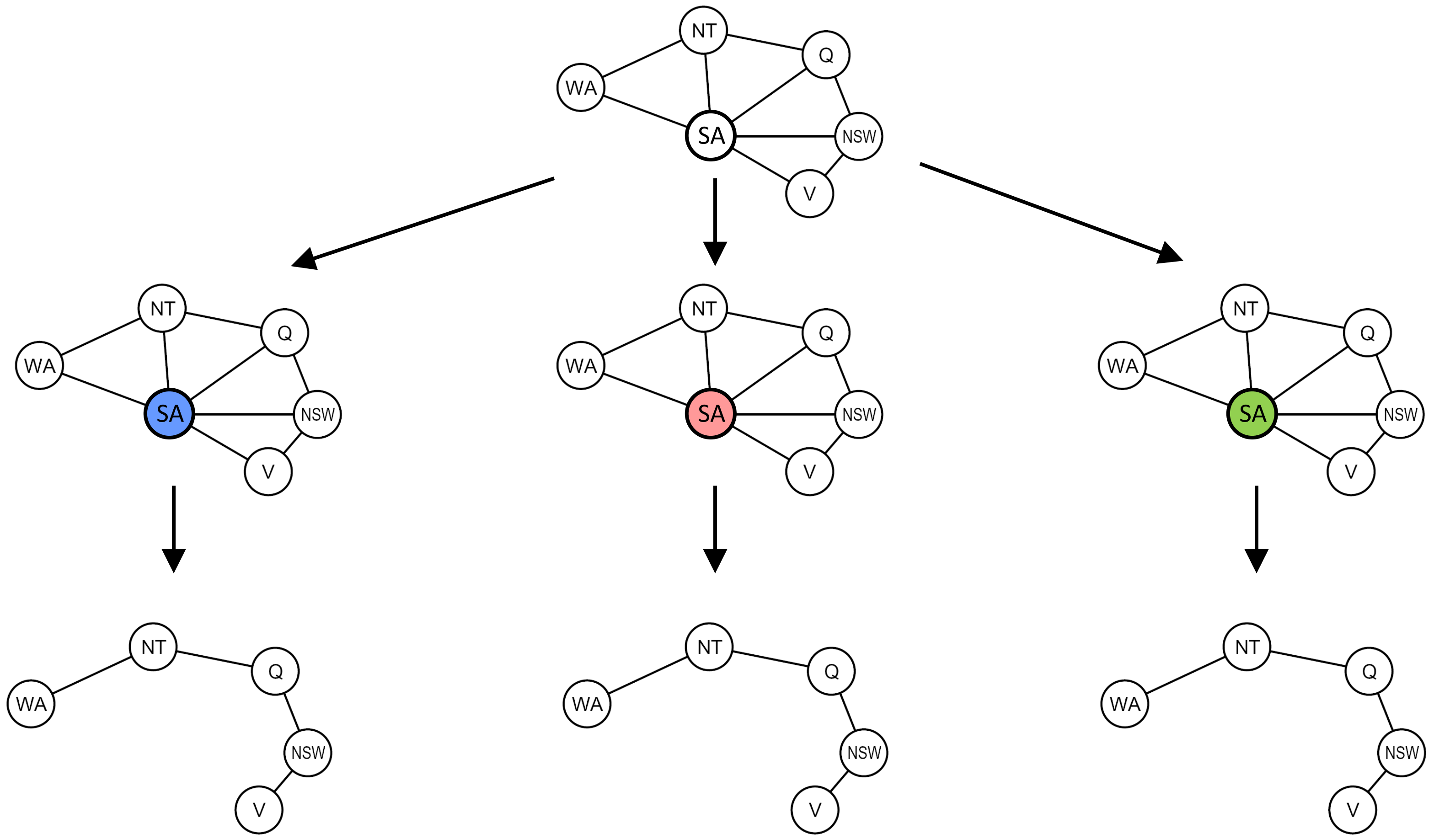
Cutset Conditioning

Choose a cutset

Instantiate the cutset (all possible ways)

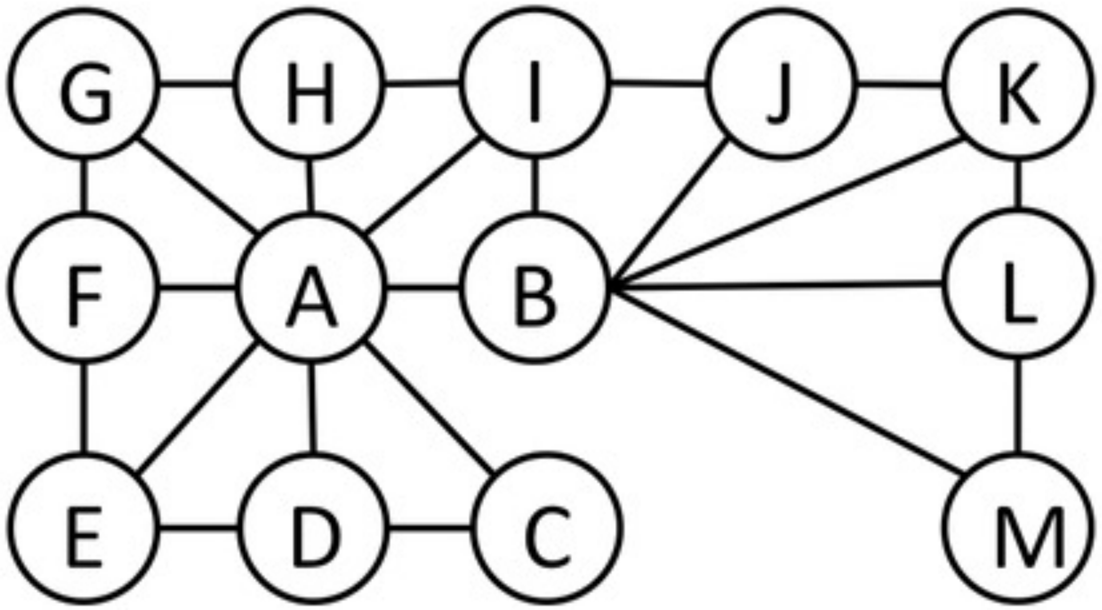
Compute residual CSP for each assignment

Solve the residual CSPs (tree structured)



Cutset Quiz

- Find the smallest cutset for the graph below.

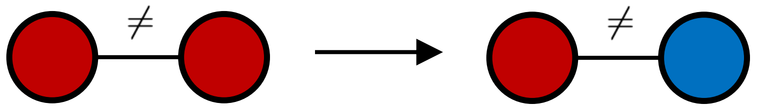


content

Iterative Algorithms for CSPs

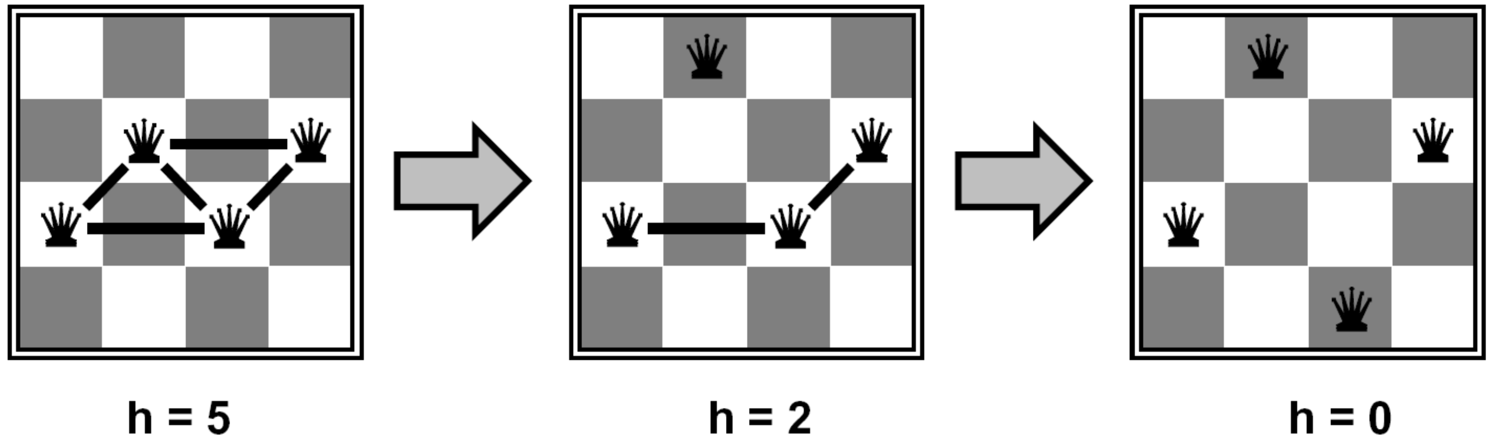
- Local search methods typically work with "complete" states, i.e., all variables assigned

- To apply to CSPs:
  - Take an assignment with unsatisfied constraints
  - Operators *reassign* variable values



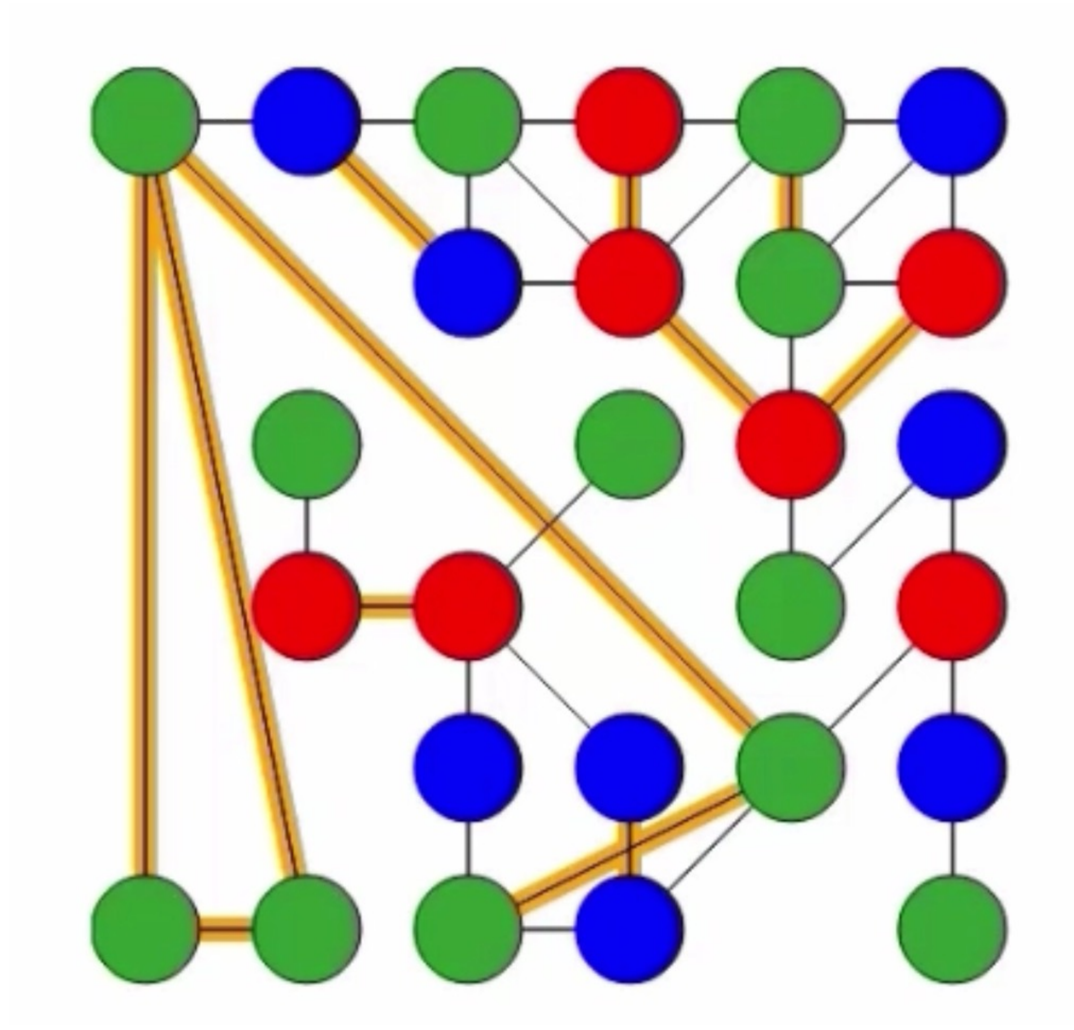
- Algorithm: While not solved,
  - Variable selection: randomly select any conflicted variable
  - Value selection: min-conflicts heuristic:
    - Choose a value that violates the fewest constraints
    - I.e., hill climb with h(n) = total number of violated constraints

Example: 4-Queens

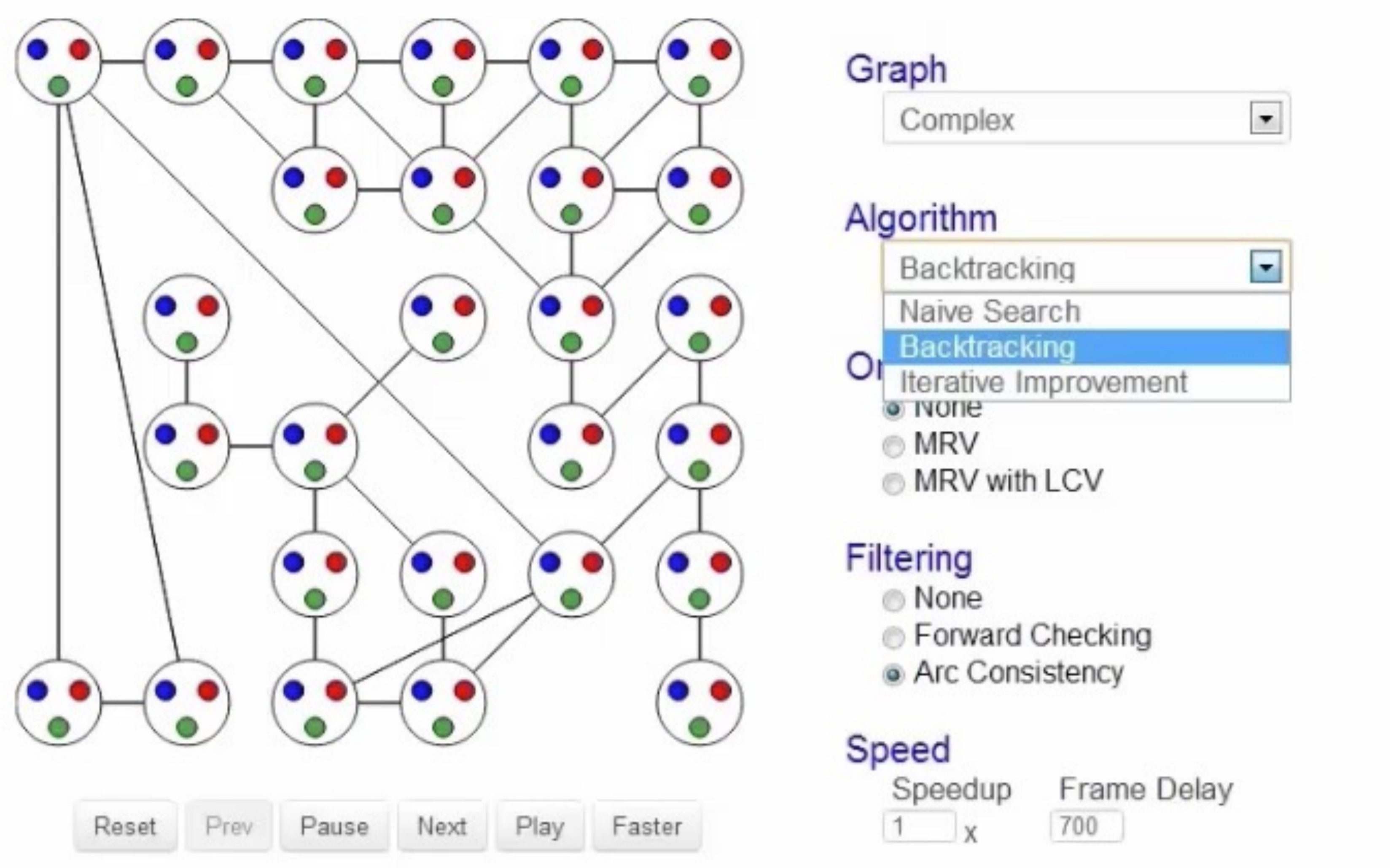


- States: 4 queens in 4 columns (4^4 = 256 states)
- Operators: move queen in column
- Goal test: no attacks
- Evaluation: c(n) = number of attacks

Video of Demo Iterative Improvement – Coloring



Video of Demo Iterative Improvement – Coloring



Graph

Complex

Algorithm

Backtracking

Or

[✓ None]
MRV
MRV with LCV

Filtering

None
Forward Checking
[✓ Arc Consistency]

Speed

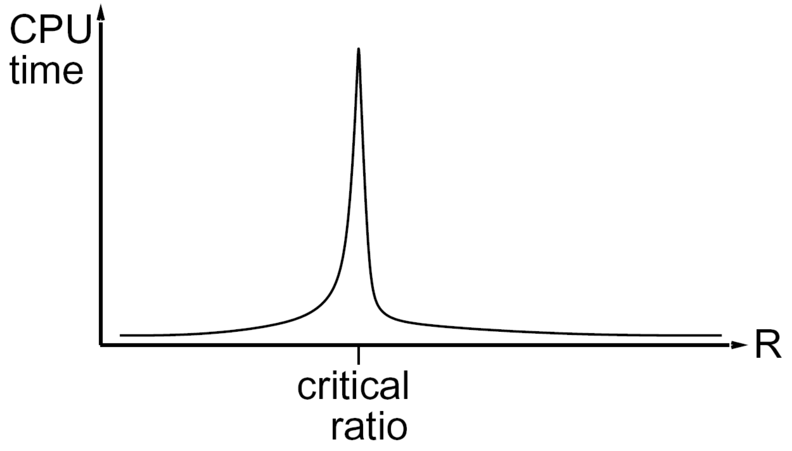
Speedup 1 x

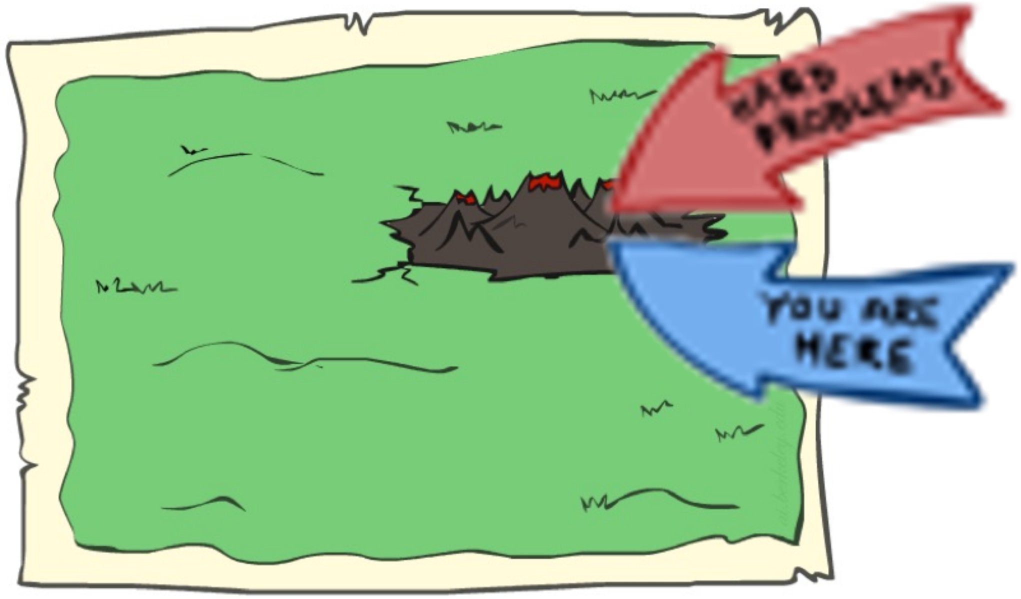
Frame Delay 700

Performance of Min-Conflicts

- Given random initial state, can solve n-queens in almost constant time for arbitrary n with high probability (e.g., n = 10,000,000)!
- The same appears to be true for any randomly-generated CSP *except* in a narrow range of the ratio

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



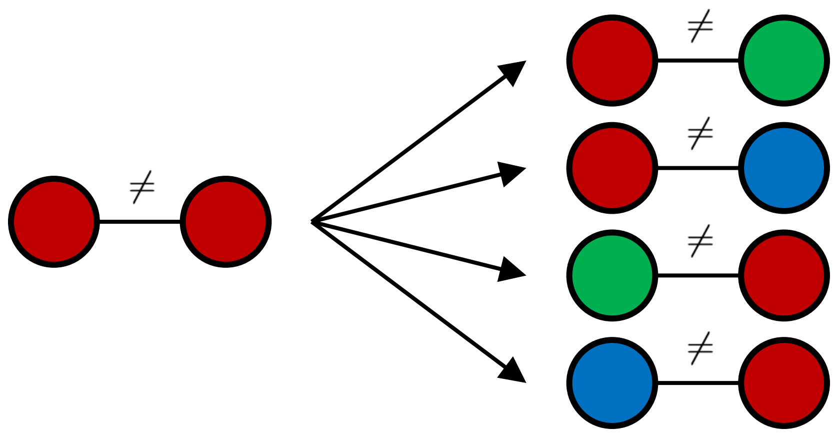


Local Search

- Tree search keeps unexplored alternatives on the openlist (ensures completeness)

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

- New successor function: local changes

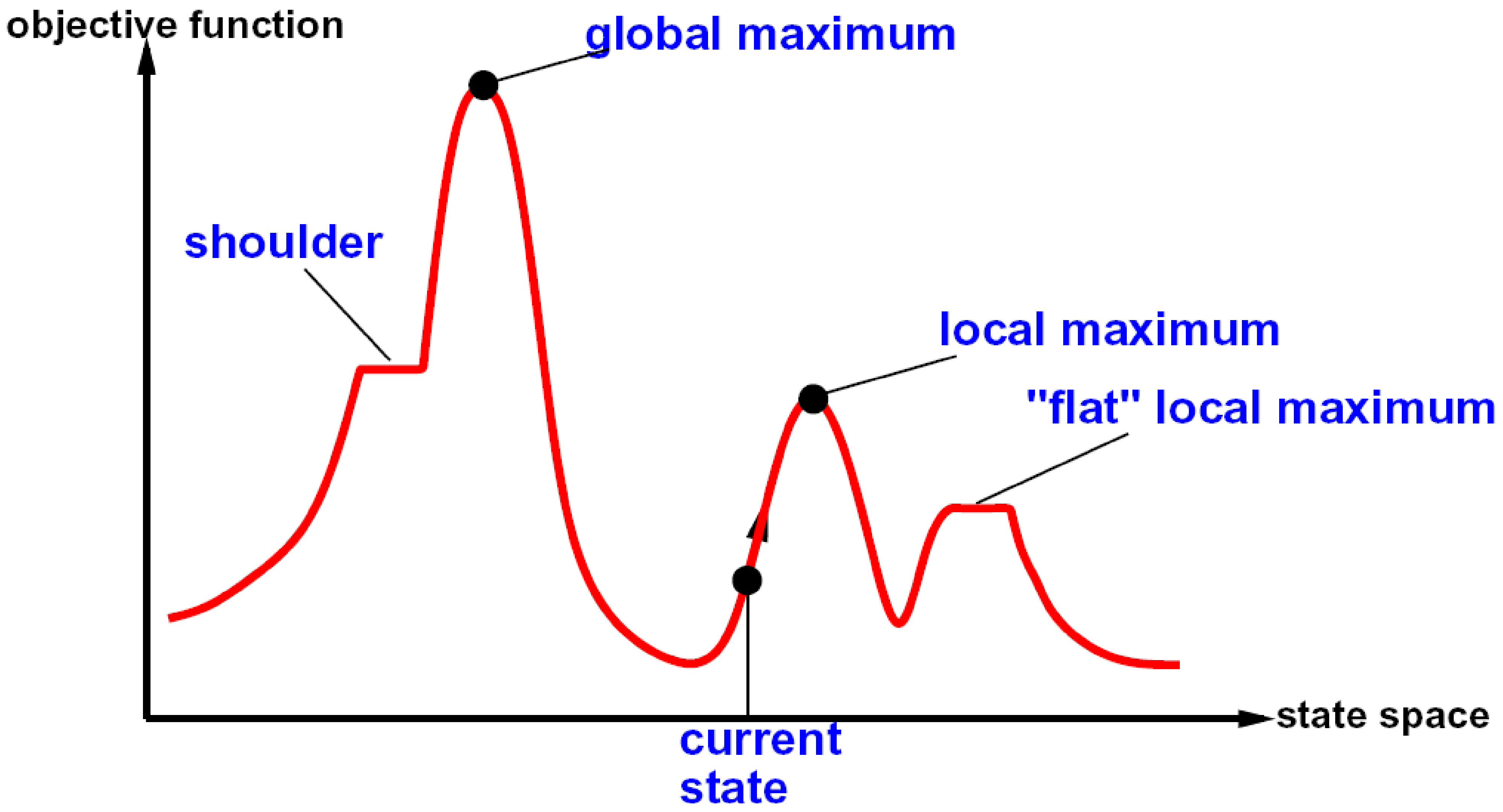


- Generally much faster and more memory efficient (but incomplete and suboptimal)

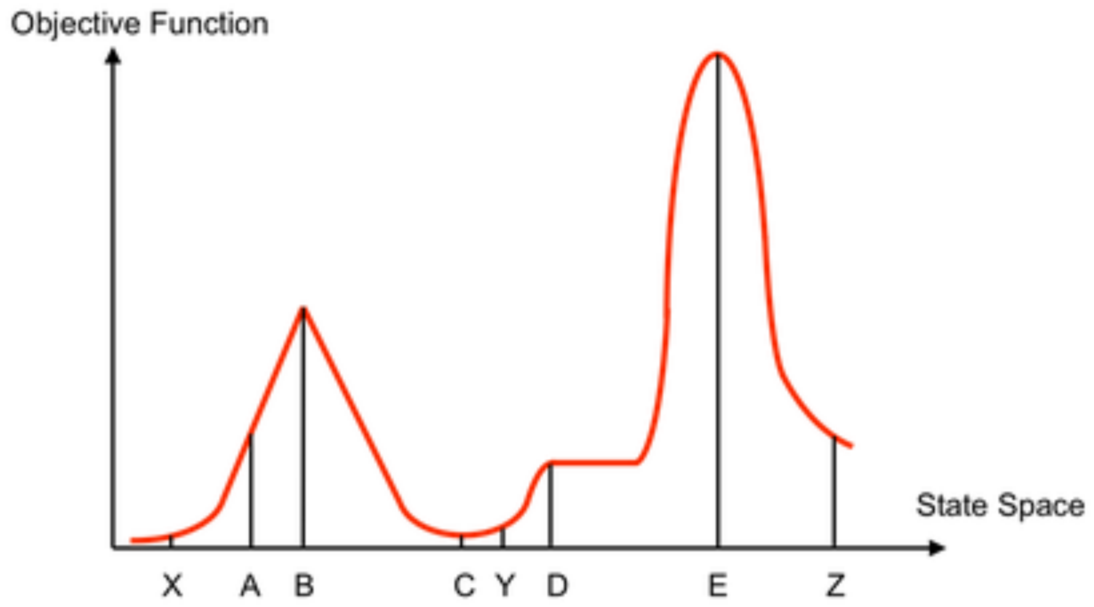
Hill Climbing

- Simple, general idea:
  - Start wherever
  - Repeat: move to the best neighboring state
  - 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



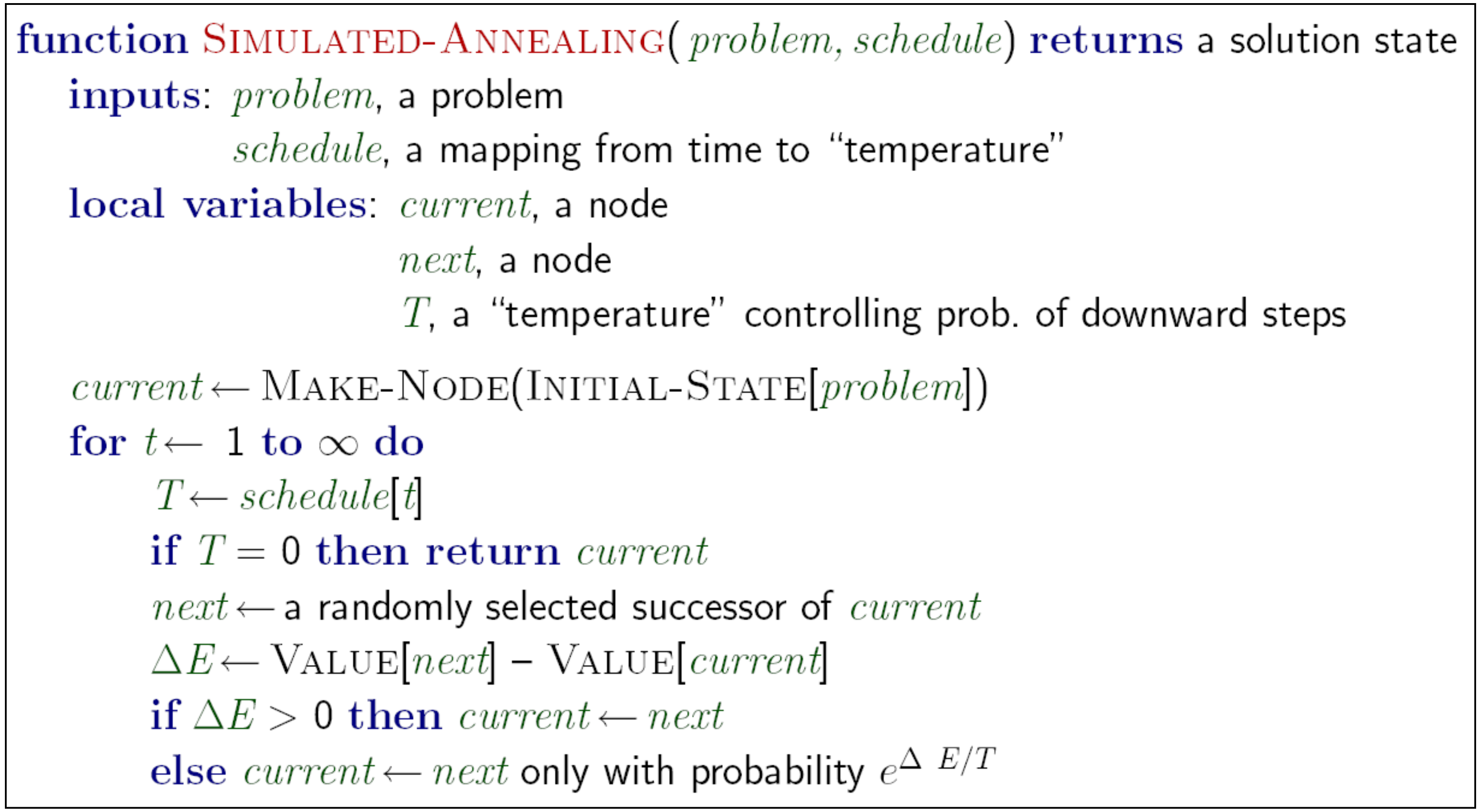
Starting from X, where do you end up ?

Starting from Y, where do you end up ?

Starting from Z, where do you end up ?

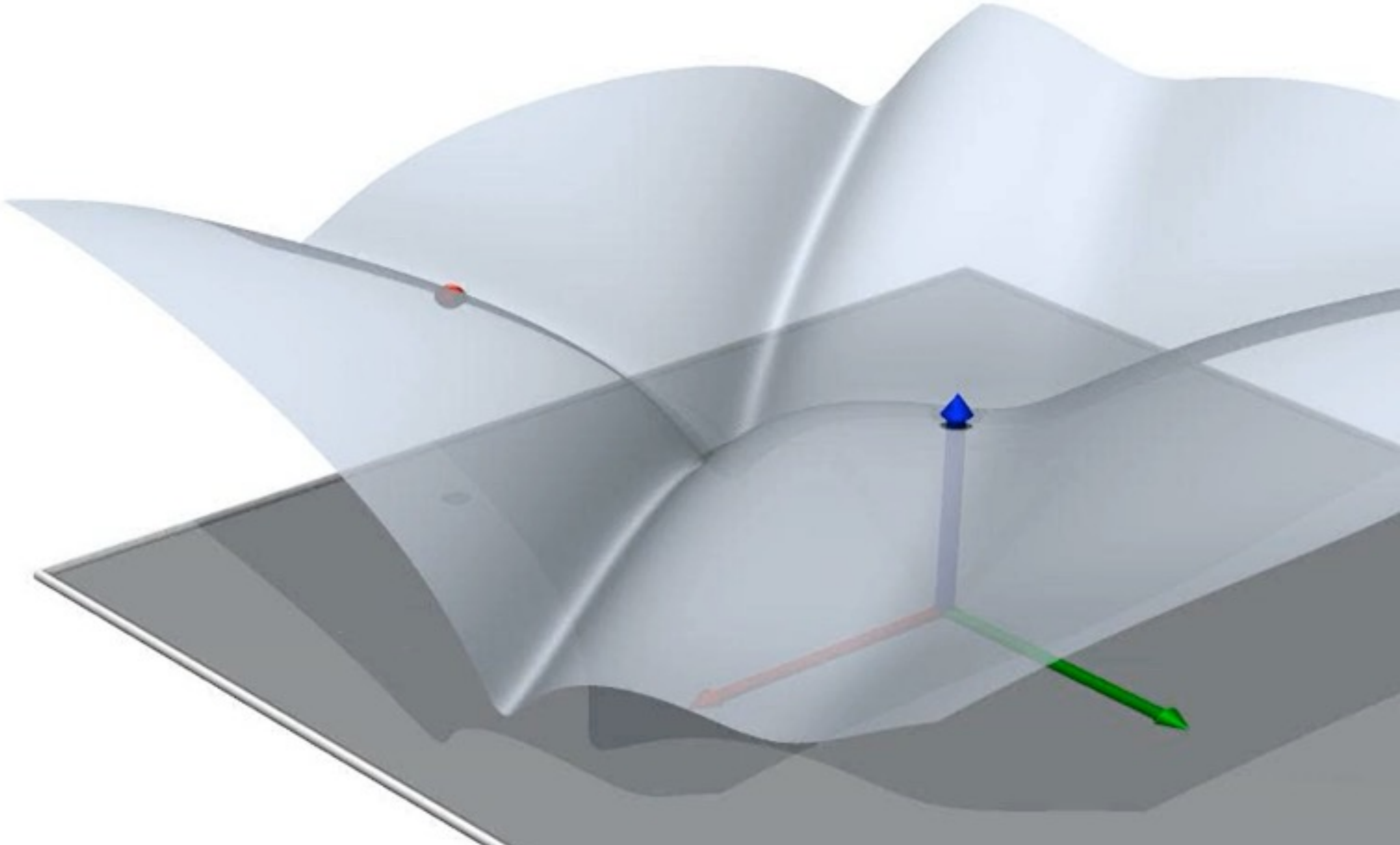
Simulated Annealing

- Idea: Escape local maxima by allowing downhill moves
  - But make them rarer as time goes on



```
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 $e^{\Delta E/T}$
```



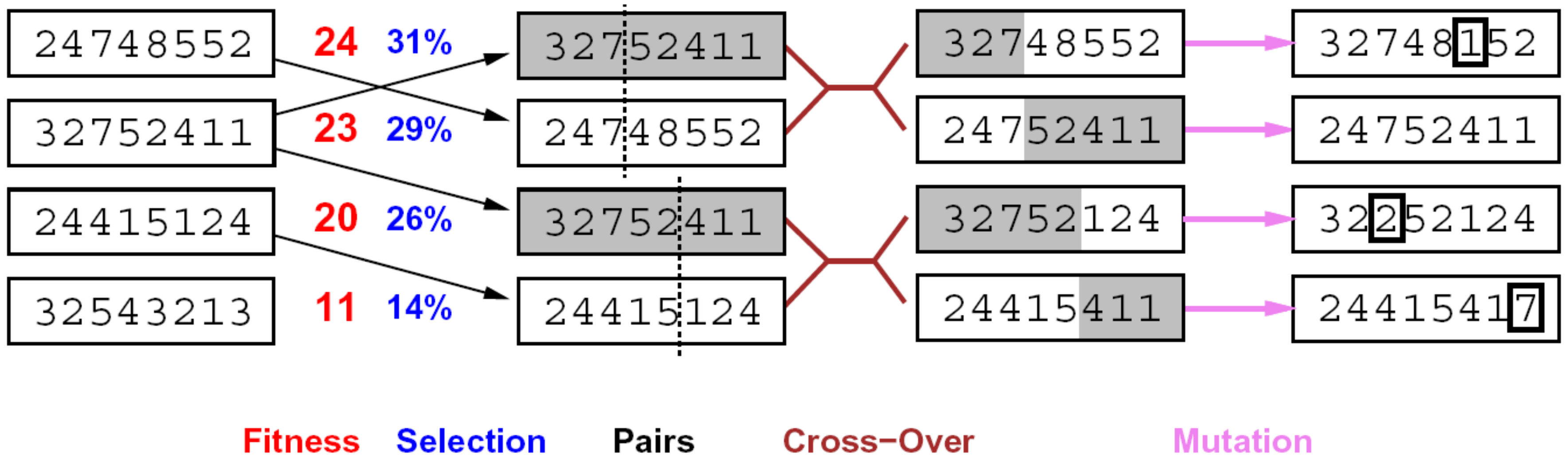
Simulated Annealing

- Theoretical guarantee:
  - Stationary distribution: $p(x) \propto e^{\frac{E(x)}{kT}}$
  - If T decreased slowly enough, will converge to optimal state!

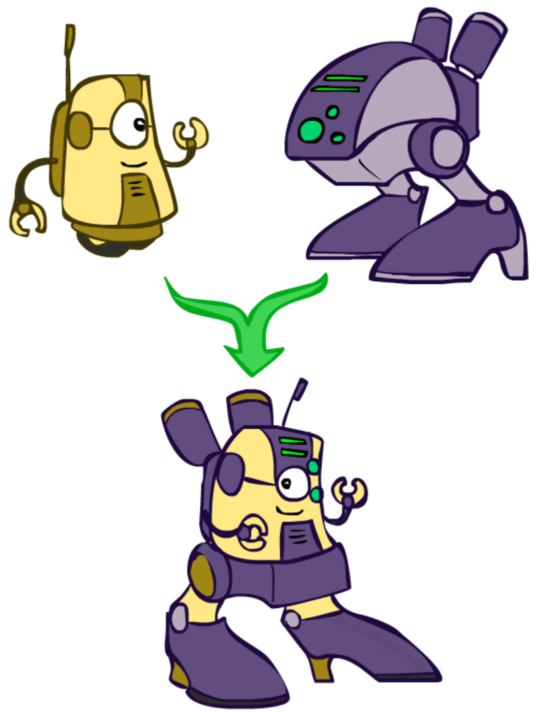
- Is this an interesting guarantee?

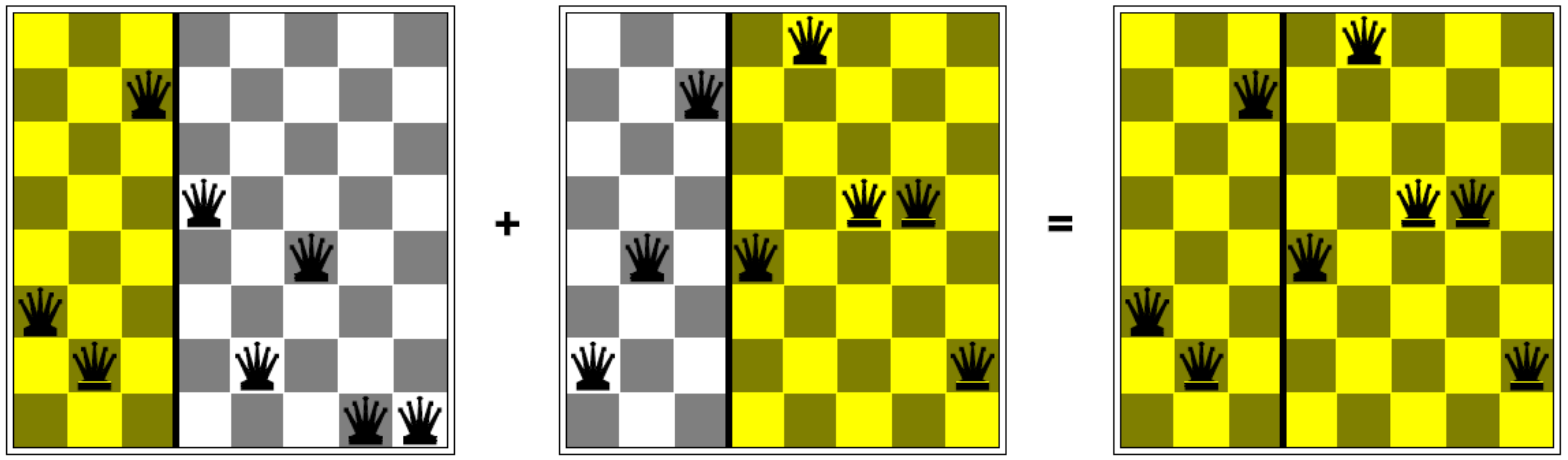
- Sounds like magic, but reality is reality

遗传算法



24748552
32752411
24415124
32543213
24 31%
23 29%
20 26%
11 14%
32752411
24748552
32752411
24415124
32748552
24752411
32752124
24415411
32748152
24752411
32252124
24415417
Fitness
Selection
Pairs
Cross−Over
Mutation





+
=

遗传算法

遗传算法

- 生物进化
  - 自然法则
    - 优胜劣汰
    - 适者生存
  - 有性繁殖
    - 基因通过有性繁殖不断进行混合和重组

- 遗传算法
  - 从生物界按照自然选择和有性繁殖、遗传变异的自然进化现象中得到启发，而设计的一种优化搜索算法

遗传算法

- 应用
  - 函数优化
  - 组合优化：旅行商、图形化分…
  - 生产调度：车间调度、生产规划…
  - 自动控制：控制器、参数辨识…
  - 机器人智能控制：机器人路径规划、运动轨迹规划…
  - 图像处理与模式识别：特征提取、图像分割…
  - 人工生命：进化模型、学习模型、行为模型…
  - 遗传程序设计
  - 机器学习

遗传算法:机构选型

遗传算法

- **个体（agent)**
  - 个体就是模拟生物个体而对问题中的对象（一般就是问题的解）的一种称呼
  - 一个个体也就是搜索空间中的一个点
- **种群**
  - 种群(population)就是模拟生物种群而由若干个体组成的群体
  - 它一般是整个搜索空间的一个很小的子集
  - 通过对种群实施遗传操作，使其不断更新换代而实现对整个空间的搜索

基本概念

- **适应度**(fitness)
  - 借鉴生物个体对环境的适应程度，而对问题中的个体对象所设计的表征其优劣的一种测度
- **适应度函数**(fitness function)
  - 问题中的全体个体与其适应度之间的一个对应关系
  - 一般是一个实值函数
  - 该函数就是遗传算法中指导搜索的评价函数

基本概念

- **染色体**(chromosome)
  - 染色体是由若干基因组成的位串（生物学）
  - 个体对象由若干字符串组成来表示（遗传算法）
- **遗传算法**(genetic algorithm)
  - 染色体就是问题中个体的某种字符串形式的编码表示
  - 染色体以字符串来表示
  - 基因是字符串中的一个个字符

个体		染色体
9	----	1001
（2，5，6）	----	010 101 110

基本概念

- **遗传算子**(genetic operator)
  - 选择(selection)
  - 交叉(crossover)
  - 变异(mutation)

选择算子

选择算子

- 模拟生物界优胜劣汰的自然选择法则的一种染色体运算
- 从种群中选择适应度较高的染色体进行复制，以生成下一代种群

算法：

- 个体适应度计算
  - 在被选集中每个个体具有一个选择概率
  - 选择概率取决于种群中个体的适应度及其分布
  - 个体适应度计算，即个体选择概率计算
- 个体选择方法
  - 按照适应度进行父代个体的选择

选择算子

- **个体适应度计算**
  - 按比例的适应度计算(proportional fitness assignment)
  - 基于排序的适应度计算(rank-based fitness assignment)
- **个体选择方法**
  - 轮盘赌选择(roulette wheel selection)
  - 随机遍历抽样(stochastic universal sampling)
  - 局部选择(local selection)
  - 截断选择(truncation selection)
  - 锦标赛选择(tournament selection)

按比例的适应度计算

算法：

对一个规模为N的种群S，按每个染色体$x_i \in S$的选择概率$P(x_i)$所决定的选中机会，分N次从S中随机选择N个染色体，并进行复制

$$P(x_i) = \frac{f(x_i)}{\sum_{j=1}^{N} f(x_j)}$$

其中：

- f为适应度函数
- $f(x_i)$为x_i的适应度

优胜劣汰

1. 概率越高，随机选中概率越大
2. 概率越高，选中次数越多
3. 适应度高的染色体后代越多

算法过程

具体操作过程是：

- 先计算出群体中所有个体的适应度的总和 fi （i=1.2,...,M）;
- 其次计算出每个个体的相对适应度的大小 fi /总和 fi ，它即为每个个体被遗传到下一代群体中的概率，
- 每个概率值组成一个区域，全部概率值之和为1；
- 最后再产生一个0到1之间的随机数，依据该随机数出现在上述哪一个概率区域内来确定各个个体被选中的次数。

交叉算子

交叉算子

- 交换、交配、杂交
- 互换两个染色体某些位上的基因
- 随机化算子，生成新个体

交叉算子

一点杂交

- 产生一个在1到L－1之间的随机数I
- 配对的两个串相互对应的交换从i＋1到L的位段

交叉算子

例3.1

设染色体s1 = 1011 0111 00

染色体s2 = 0001 1100 11

交换其后2位基因

$$\begin{array}{l} s_1\text{:}\ \ 1011\ 0111 \,\vdots\, 00 \\ s_2\text{：}\ 0001\ 1100 \,\vdots\, 11 \end{array} \xrightarrow{\text{单点交叉}} \begin{array}{l} s_1'\text{：}\ 1011\ 0111 \,\vdots\, 11 \\ s_2'\text{：}\ 0001\ 1100 \,\vdots\, 00 \end{array}$$

变异算子

变异算子

- 突变
- 改变染色体某个/些位上的基因
- 随机化算子，生成新个体
- 次要算子，但在恢复群体中失去的多样性方面具有潜在的作用

变异算子

例4.1

设染色体s = 1011 0111 00

$$s_1: \ 1011\ 011\boxed{1}\ 00 \xrightarrow{\text{二进制变异}} s_1'\text{：}\ 1011\ 011\boxed{0}\ 00$$

基本遗传算法

遗传算法

- 对种群中的染色体反复做三种遗传操作
- 使其朝着适应度增高的方向不断更新换代，直至出现了适应度满足目标条件的染色体为止

算法拓展

- 遗传算法在自然与社会现象模拟、工程计算等方面得到了广泛的应用
- 基本遗传算法是Holland提出的一种统一的最基本的遗传算法，简称SGA（Simple Genetic Algorithm）、CGA（Canonical Genetic Algorithm）
- 其它的“GA类”算法称为GAs(Genetic Algorithms）,可以把GA看作是GAs的一种特例

基本遗传算法

参数

- 种群规模
  - 种群的大小，用染色体个数表示
- 最大换代数
  - 种群更新换代的上限，也是算法终止一个条件
- 交叉率Pc
  - 参加交叉运算的染色体个数占全体染色体总数的比例
  - 取值范围：0.4-0.99
- 变异率Pm
  - 发生变异的基因位数占全体染色体的基因总位数的比例
  - 取值范围：0.0001-0.1
- 染色体编码长度 L

基本遗传算法

步1：在问题空间U上定义一个适应度函数f(x)，给定种群规模N，交叉率Pc，变异率Pm，代数Gen

步2: 随机产生U中的N个染色体$s_1,s_2…s_N$，组成初始种群$S=\{s_1,s_2…s_N\}$，置代数t=1

步3: 若终止条件满足，则取S中适应度最大的染色体作为所求结果，算法结束

步4: 计算S中每个染色体的适应度f()

步5: 按选择概率$p(s_i)$所决定的选中机会，每次从S中随机选中1个染色体并将其复制，共做N次，然后将复制得到的N染色体组成群体S_1

步6：按Pc所决定的参加交叉的染色体数c，从S_1中随机确定c个染色体，配对进行交叉操作，并用产生的染色体代替原染色体，组成群体S_2

步7：按Pm所决定的变异次数m，从S_2中随机确定m个染色体，分别进行变异操作，并用产生的新染色体代替原染色体，组成群体S_3

步8：将群体S_3作为新种群，即用S_3代替S， Gen = Gen +1，转步3

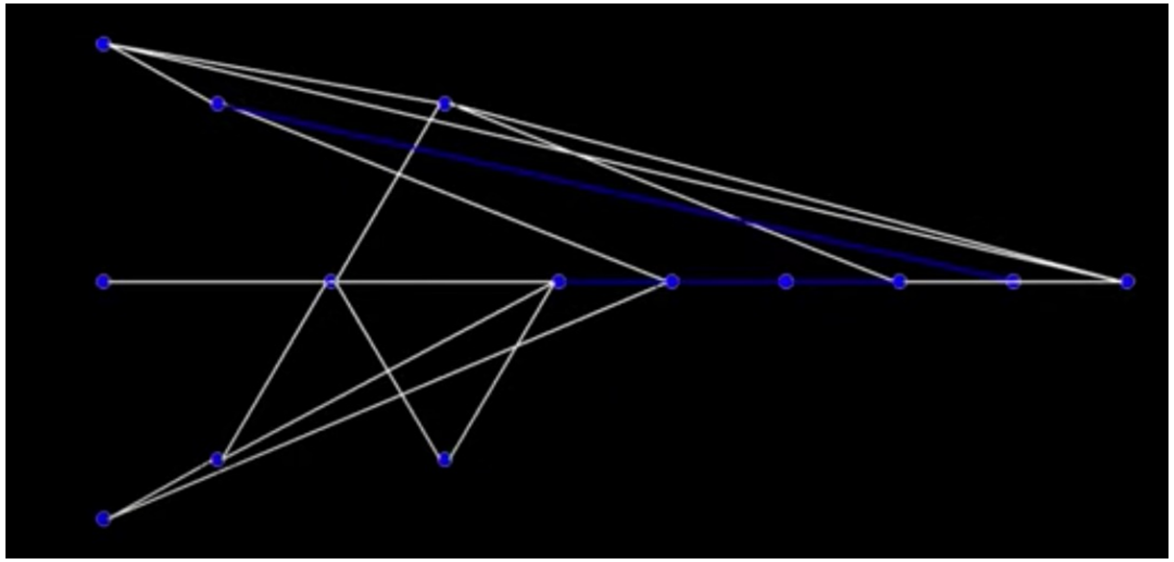
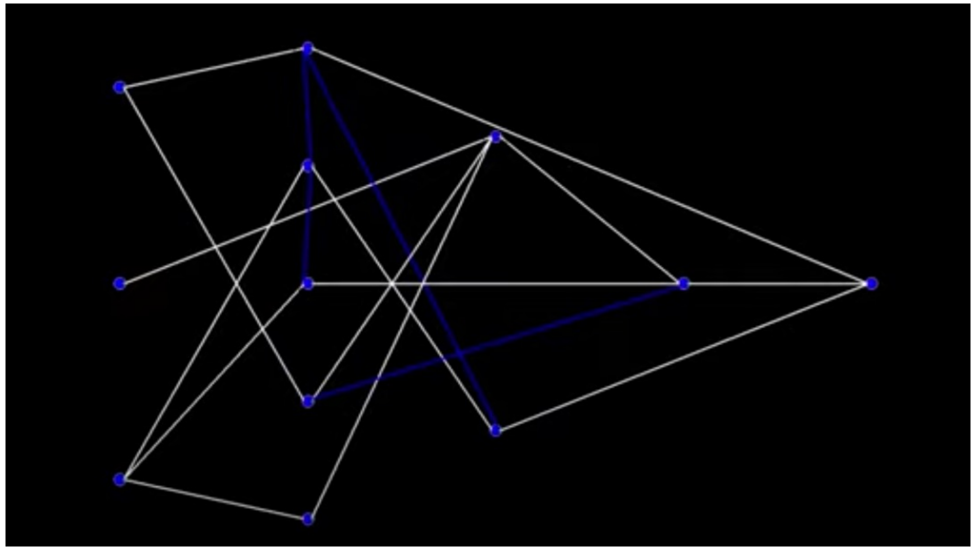
算法比较

遗传算法	图搜索
解空间搜索	问题空间搜索->解
随机搜索、随机选取初始点集/种群	固定初始/目标节点
寻找最优解/次优解	寻找解
点集->点集、并行计算	点->点
需适应度函数	需先验知识
全局搜索	约束较多

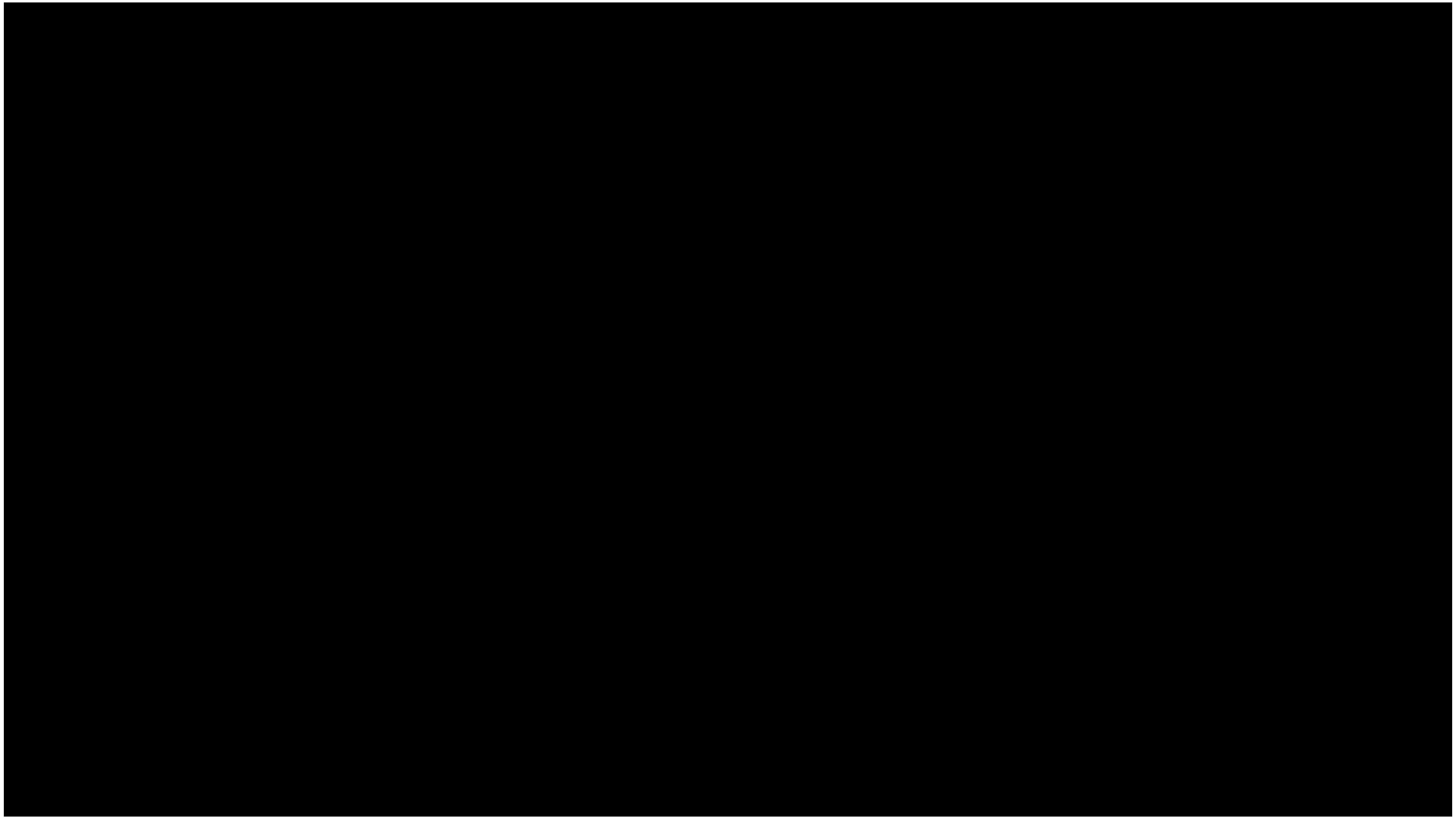
Genetic Algorithms Flappy Bird

Genetic Algorithms Flappy Bird

- NeuroEvolution of Augmenting Topologies(NEAT)
- https://www.youtube.com/watch?v=ihX3-WDua2I&ab_channel=NeatAI

Genetic Algorithms Super Marios



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
  - Filtering
  - Structure
- Iterative min-conflicts is often effective in practice
- Local Search