# Features and Constraints (Continued)

- Poole, chap. 4

- *Some slides and/or pictures in the following are adapted from slides ©2013. Prof. L. Kosseim;*

# Last Class

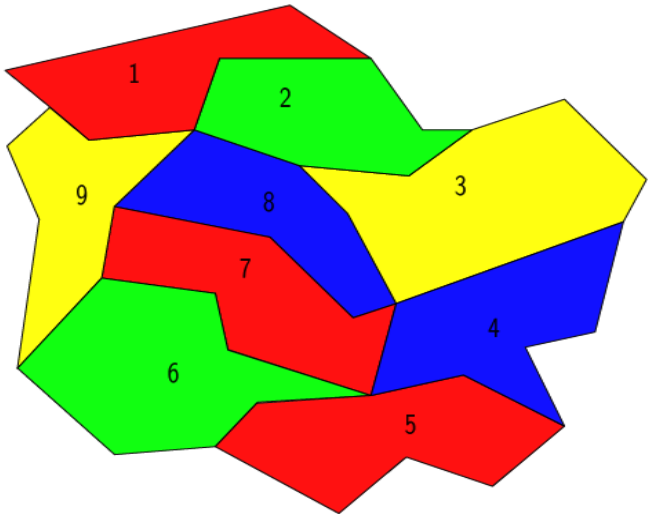
## Constraints Satisfaction:

- Components of the state:
  - Variables
  - Domains (Allowed values for the variables)
  - Binary constraints among variables
- **Goal:** To find a state that satisfies all the constraints (Set of values for the variables that satisfies the constraints)
- Examples:
  - Map coloring, Crosswords, 8-queens
  - Allocation/distribution of resources (Scheduling of tasks of a process, location of gas stations, mobile antennas, ...)
  - Automatic reasoning and logic
  - Image recognition and artificial vision
  - Task planning
  - ...

# Example

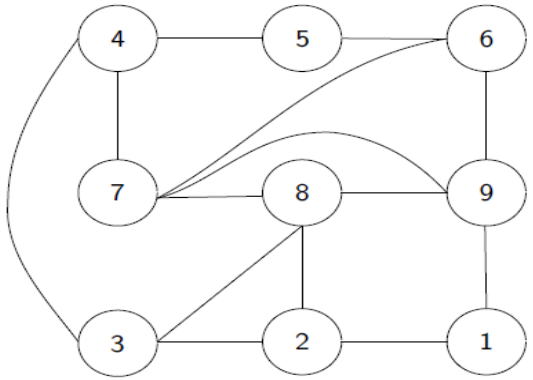
## Representation

- State = Graph of constraints
  - Variables = Labels of the vertices
  - Domains = Values of the vertices
  - Constrains = directed and labeled edges among vertices
- Example: Map coloring




Domains={Red,Green,Blue,Yellow}

Constraint := Inequality



# Features and CSP: Algorithms

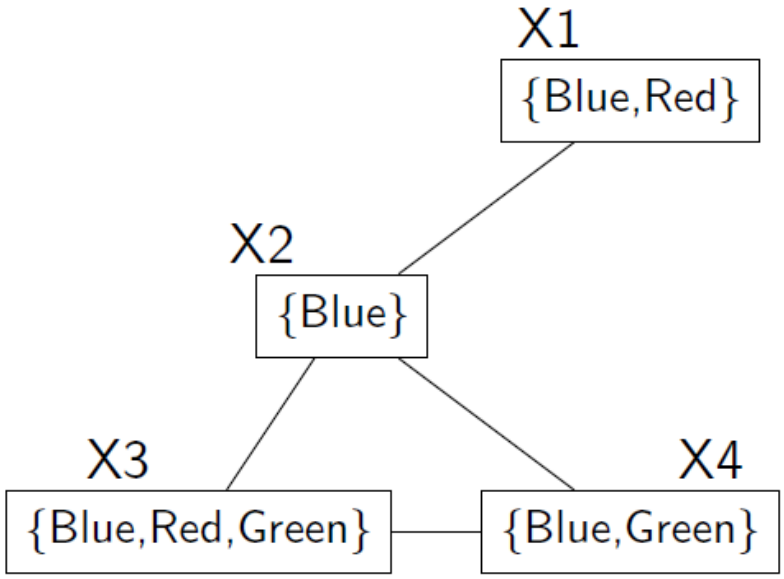
- Generate and test: Very inefficient
- Backtracking Algorithm: any partial assignment that does not satisfy the constraints can be pruned
- Consistency Algorithms: prune the domains as much as possible before selecting value from them.
  - Arc-consistency:
    - A CSP is arc-consistent if for each pair of variables $(X_i, X_j)$ and any value $v_k$ from $D_i$ there is a value $v_l$ from $D_j$ that satisfies the constraints. We are looking for the values from $X_i$ that are consistent with the constraints on the edge of this variable
    - We want that all variables are arc-consistent for all the edges they have. In other words, the domains of all the variables must be consistent with all the constraints

# Arc-Consistency Algorithm

**Algorithm:** Arc consistency

```
R ← set of edges of the problem /* both sides                    */
while the domains of the variables are modified do
    r ← get_arc(R)
    /* r_i is the variable origin of the edge                    */
    /* r_j is the variable end of the edge                       */
    foreach v ∈ the domain of r_i do
        if v does not have a value in r_j domain that satisfies r then
            delete v from r_i domain
            add all edges that end in r_i except (r_j → r_i)
        end
    end
end
```

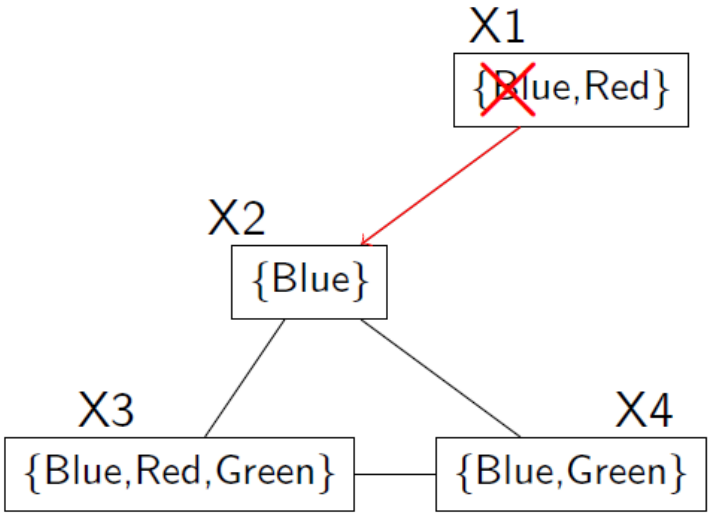
# Arc-consistency example (1/2)




Initial list of edges:
(X1,X2), (X2,X1), (X2,X3), (X3,X2),
(X2,X4), (X4,X2), (X3,X4), (X4,X3)

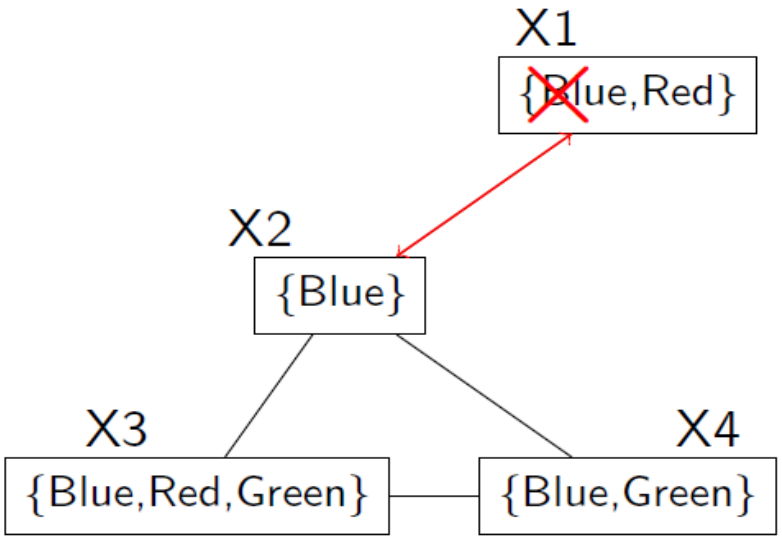
# Arc-consistency example (2/2)

1. $X_1 - X_2 \rightarrow$ Delete Blue from $X_1$




Initial list of edges:
(X1,X2), (X2,X1), (X2,X3), (X3,X2),
(X2,X4), (X4,X2), (X3,X4), (X4,X3)

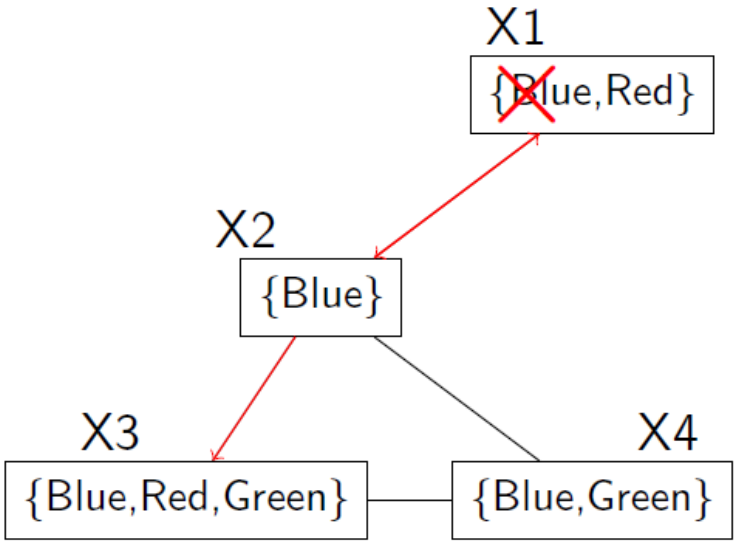
# Arc-consistency example (1/2)




1. $X_1 - X_2 \rightarrow$ Delete Blue from $X_1$
2. $X_2 - X_1 \rightarrow$ All consistent

Initial list of edges:
(X1,X2), (X2,X1), (X2,X3), (X3,X2),
(X2,X4), (X4,X2), (X3,X4), (X4,X3)

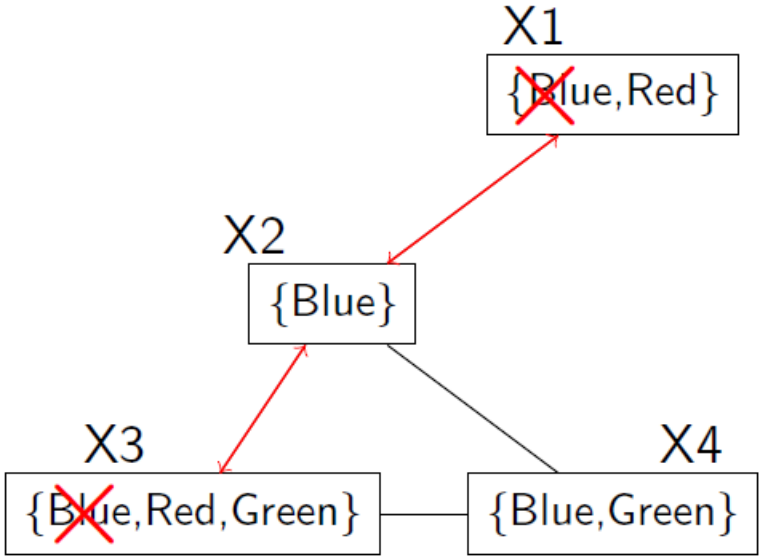
# Arc-consistency example (1/2)




1. $X_1 - X_2 \rightarrow$ Delete Blue from $X_1$
2. $X_2 - X_1 \rightarrow$ All consistent
3. $X_2 - X_3 \rightarrow$ All consistent

Initial list of edges:
(X1,X2), (X2,X1), (X2,X3), (X3,X2),
(X2,X4), (X4,X2), (X3,X4), (X4,X3)

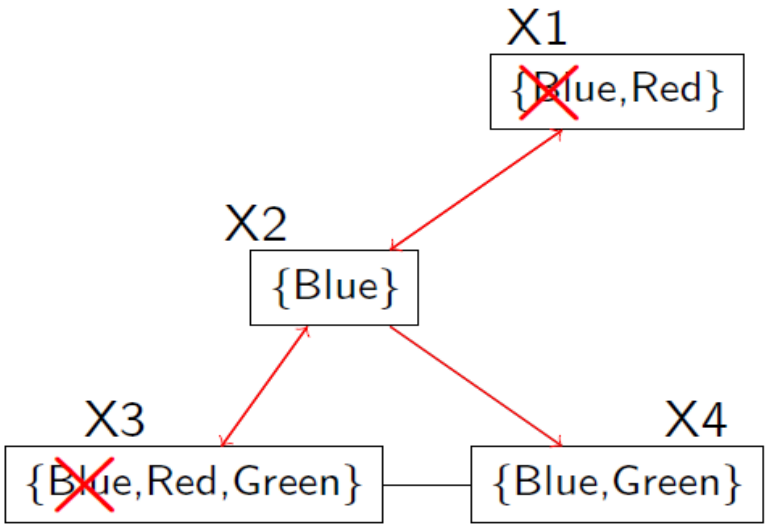
# Arc-consistency example (1/2)




1. $X_1 - X_2 \rightarrow$ Delete Blue from $X_1$
2. $X_2 - X_1 \rightarrow$ All consistent
3. $X_2 - X_3 \rightarrow$ All consistent
4. $X_3 - X_2 \rightarrow$Delete Blue from $X_3$, we should add $X_4 - X_3$ but it is in already

Initial list of edges:
(X1,X2), (X2,X1), (X2,X3), (X3,X2),
(X2,X4), (X4,X2), (X3,X4), (X4,X3)

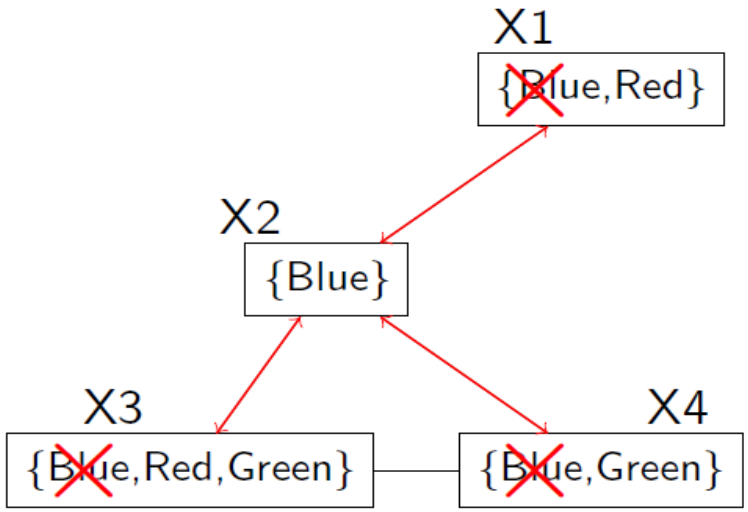
# Arc-consistency example (1/2)




1. $X_1 - X_2 \rightarrow$ Delete Blue from $X_1$
2. $X_2 - X_1 \rightarrow$ All consistent
3. $X_2 - X_3 \rightarrow$ All consistent
4. $X_3 - X_2 \rightarrow$Delete Blue from $X_3$, we should add $X_4 - X_3$ but it is in already
5. $X_2 - X_4 \rightarrow$ All consistent

Initial list of edges:
(X1,X2), (X2,X1), (X2,X3), (X3,X2),
(X2,X4), (X4,X2), (X3,X4), (X4,X3)

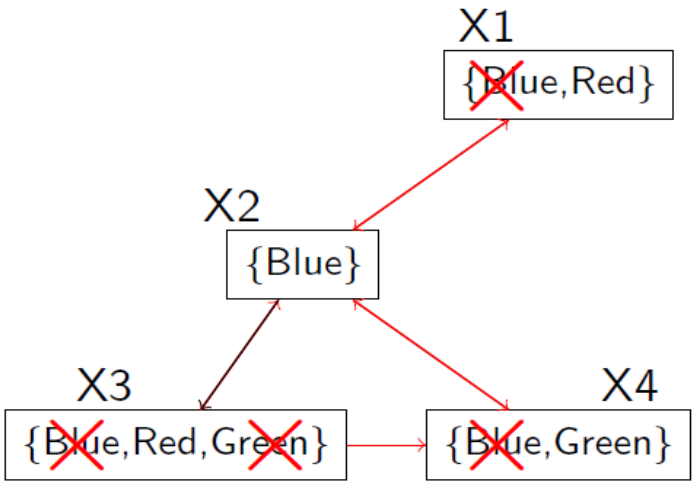
# Arc-consistency example (1/2)




1. $X_1 - X_2 \rightarrow$ Delete Blue from $X_1$
2. $X_2 - X_1 \rightarrow$ All consistent
3. $X_2 - X_3 \rightarrow$ All consistent
4. $X_3 - X_2 \rightarrow$Delete Blue from $X_3$, we should add $X_4 - X_3$ but it is in already
5. $X_2 - X_4 \rightarrow$ All consistent
6. $X_4 - X_2 \rightarrow$ Delete Blue from $X_4$, we should add $X_3 - X_4$ but it is in already

Initial list of edges:
(X1,X2), (X2,X1), (X2,X3), (X3,X2),
(X2,X4), (X4,X2), (X3,X4), (X4,X3)

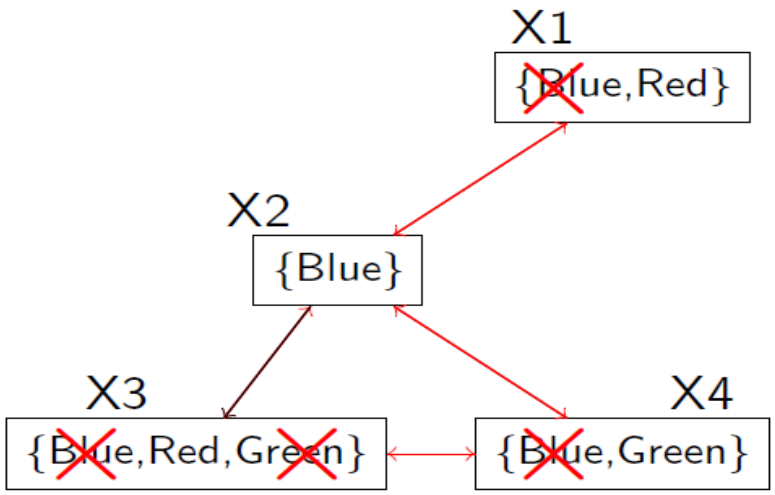
# Arc-consistency example (1/2)




Initial list of edges:
(X1,X2), (X2,X1), (X2,X3), (X3,X2), (X2,X4), (X4,X2), (X3,X4), (X4,X3)

1. $X_1 - X_2 \rightarrow$ Delete Blue from $X_1$
2. $X_2 - X_1 \rightarrow$ All consistent
3. $X_2 - X_3 \rightarrow$ All consistent
4. $X_3 - X_2 \rightarrow$Delete Blue from $X_3$, we should add $X_4 - X_3$ but it is in already
5. $X_2 - X_4 \rightarrow$ All consistent
6. $X_4 - X_2 \rightarrow$ Delete Blue from $X_4$, we should add $X_3 - X_4$ but it is in already
7. $X_3 - X_4 \rightarrow$ Delete Blue from $X_3$, we add $X_2 - X_3$

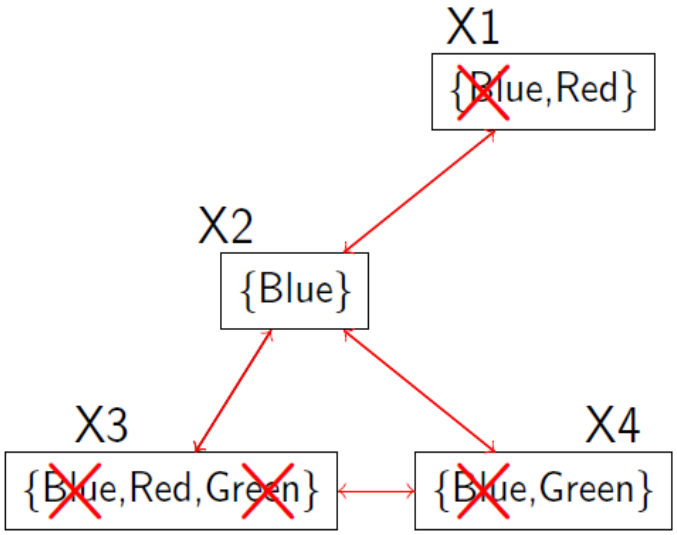
# Arc-consistency example (1/2)




Initial list of edges:
(X1,X2), (X2,X1), (X2,X3), (X3,X2), (X2,X4), (X4,X2), (X3,X4), (X4,X3)

1. $X_1 - X_2 \rightarrow$ Delete Blue from $X_1$
2. $X_2 - X_1 \rightarrow$ All consistent
3. $X_2 - X_3 \rightarrow$ All consistent
4. $X_3 - X_2 \rightarrow$Delete Blue from $X_3$, we should add $X_4 - X_3$ but it is in already
5. $X_2 - X_4 \rightarrow$ All consistent
6. $X_4 - X_2 \rightarrow$ Delete Blue from $X_4$, we should add $X_3 - X_4$ but it is in already
7. $X_3 - X_4 \rightarrow$ Delete Blue from $X_3$, we add $X_2 - X_3$
8. $X_4 - X_3 \rightarrow$ All consistent

# Arc-consistency example (1/2)




Initial list of edges:
(X1,X2), (X2,X1), (X2,X3), (X3,X2), (X2,X4), (X4,X2), (X3,X4), (X4,X3)

1. $X_1 - X_2 \rightarrow$ Delete Blue from $X_1$
2. $X_2 - X_1 \rightarrow$ All consistent
3. $X_2 - X_3 \rightarrow$ All consistent
4. $X_3 - X_2 \rightarrow$Delete Blue from $X_3$, we should add $X_4 - X_3$ but it is in already
5. $X_2 - X_4 \rightarrow$ All consistent
6. $X_4 - X_2 \rightarrow$ Delete Blue from $X_4$, we should add $X_3 - X_4$ but it is in already
7. $X_3 - X_4 \rightarrow$ Delete Blue from $X_3$, we add $X_2 - X_3$
8. $X_4 - X_3 \rightarrow$ All consistent
9. $X_2 - X_3 \rightarrow$ All consistent

# Today

- Features and States
- Constraint Satisfaction problems
- Generate-and-Test Algorithms
- Solving CSPS Using Search
- Consistency Algorithms
- Domain Splitti
- Local Search


  - Hill Climbing
- Stochastic Local Search
- Simulated Annealing
- Population based methods
  - Beam search:
  - Stochastic beam search:
  - Genetic algorithms:

# Local Search

- So far we considered algorithms that systematically search the space. If the space is finite, they will either find a solution or report that no solution exists.
- Many search spaces are too big for systematic search and are possibly even infinite.
  - In any reasonable time, systematic search will have failed.
- Methods for very large spaces.
  - The methods do not systematically search the whole search space but they are designed to find solutions quickly on average.
  - They do not guarantee that a solution will be found even if one exists, and so they are not able to prove that no solution exists.
  - They are often the method of choice for applications where solutions are known to exist or are very likely to exist.

# Local Search

- Sometimes we don't need the path that reaches a solution, we search in the space of solutions
- We want to obtain the best attainable solution in an affordable time (optimal is impossible)
- We have a function that evaluates the quality of the solution, this value is not related to a path cost
- Search is performed from an initial solution that we try to improve using actions
- The actions allow to move in the solution neigbourhood

- The heuristic function:
  - Approximates the quality of a solution (it is not a cost)
  - The goal is to optimize it (maximize or minimize)
  - Combines all the elements of the problem and its constraints (possibly using different weights for different elements)
  - There are no constrains about how the function can be, it only has to represent the quality relations among the solutions
  - It can be positive or negative

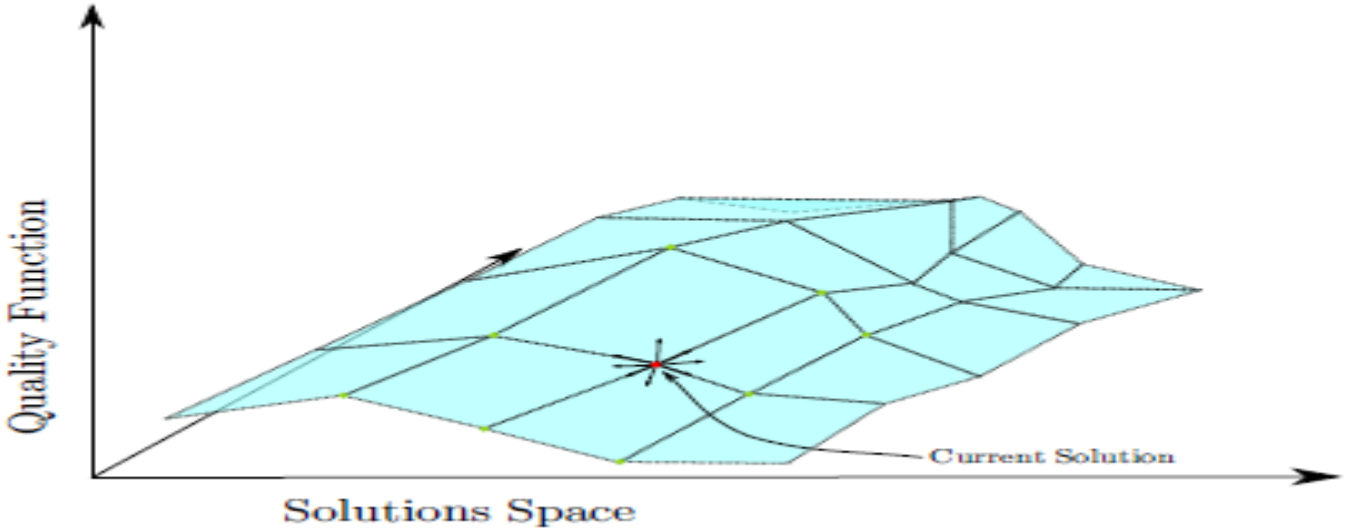
# Local Search

- Local search methods start with a complete assignment of a value to each variable and try to iteratively improve this assignment by improving steps, by taking random steps, or by restarting with another complete assignment.
- A wide variety of local search techniques has been proposed. Understanding when these techniques work for different problems forms the focus of a number of research communities, including those from both operations research and AI.

Local Search (Greedy Descent):

- Maintain an assignment of a value to each variable.
- Repeat:
  - Select a variable to change
  - Select a new value for that variable
- Until a satisfying assignment is found

# Local Search




- The size of the space of solutions doesn't allow for an optimal solution search
- It is not possible to perform a systematic search
- The heuristic function is used to prune the space of solutions (solutions that don't need to be explored)
- Usually no history of the search path is stored (minimal space complexity)

# Local Search

```
1: Procedure Local-Search(V,dom,C)
2:         Inputs
3:                 V: a set of variables
4:                 dom: a function such that dom(X) is the domain of variable X
5:                 C: set of constraints to be satisfied
6:         Output
7:                 complete assignment that satisfies the constraints
8:         Local
9:                 A[V] an array of values indexed by V
10:        repeat
11:                for each variable X do
12:                        A[X] ←a random value in dom(X);
13:                while (stopping criterion not met & A is not a satisfying assignment)
14:                        Select a variable Y and a value V ∈dom(Y)
15:                        Set A[Y] ←V
16:                if (A is a satisfying assignment) then
17:                        return A
18:        until termination
```

1

2

1. loop: is call the try
2. loop does the Local search or walk through the assignment space

# Today

- Features and States
- Constraint Satisfaction problems
- Generate-and-Test Algorithms
- Solving CSPS Using Search
- Consistency Algorithms
- Domain Splitting
- Local Search
  - Hill Climbing 

- Stochastic Local Search
- Simulated Annealing
- Population based methods
  - Beam search:
  - Stochastic beam search:
  - Genetic algorithms:

# Hill Climbing

- First-choice Hill climbing
  - First action that improves the current solution is taken
- Steepest-ascent hill climbing, gradient search
  - The best action that improves the current solution is taken

---

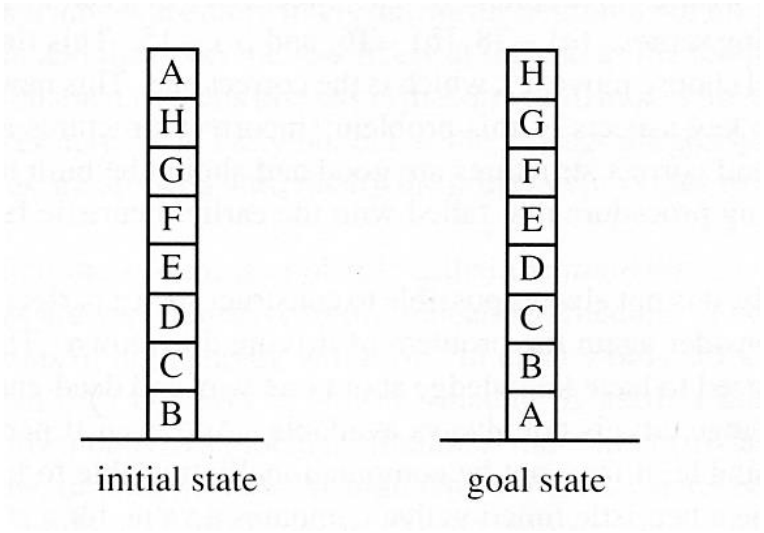
**Algorithm:** Hill Climbing

```
Current ← initial state
End ← false
while not End do
    Successors ← generate_successor(Current)
    Successors ← sort_and_prune_bad_solutions(Successors, Current)
    if not empty?(Successors) then
        Current ← best_successor(Successors)
    else
        End ← true
    end
end
```

---

- Only are considered successors those solutions with a heuristic function value better than the current solution (pruning of the space of solutions)
- A stack could be used to store the best successors to backtrack, but usually the space requirement are prohibitive
- The algorithm may not find any solution even when there are

# Example: Hill Climbing with Blocks World

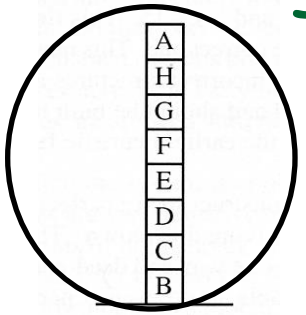



- Operators:
  - **`pickUp(Block)`**
  - **`putOnTable(Block)`**
  - **`stack(Block1,Block2)`**

- Heuristic:
  - 0 pt if a block is sitting where it is supposed to sit
  - +1 pt if a block is NOT sitting where it is supposed to sit

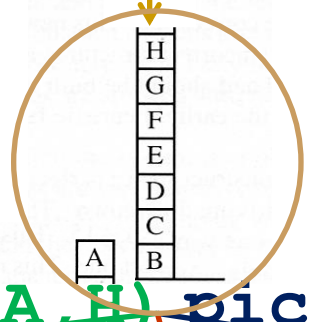
  - so lower h(n) is better
    - h(initial) = 2
    - h(goal) = 0

# Example: Hill Climbing with Blocks World




*h(n) = 2*

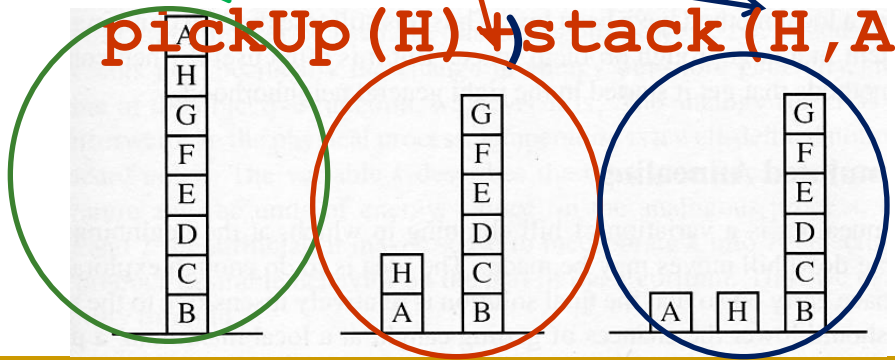
**pickUp(A)+putOnTable(A)**




*h(n) = 1*

**pickUp(A)+stack(A,H)** **pickUp(H)+putOnTable(H**

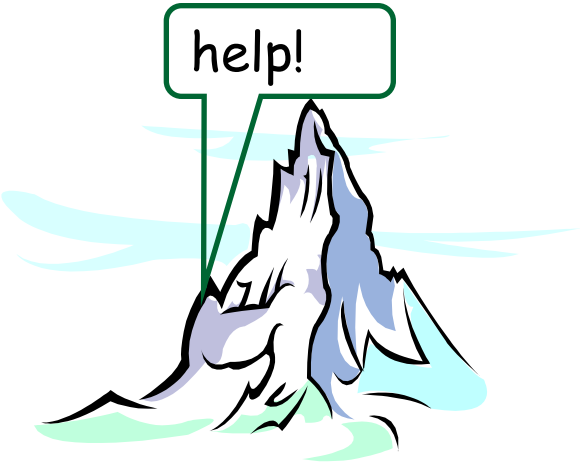
**pickUp(H)+stack(H,A)**




*h(n) = 2* *h(n) = 2* *h(n) = 2*

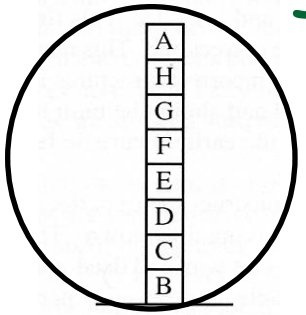
# Hill Climbing

- General hill climbing strategy:
  - as soon as you find a position that is better than the current one, select it.
  - Is a local search
  - Does not maintain a list of next nodes to visit (an *open* list)
  - Similar to climbing a mountain in the fog with amnesia … always go higher than where you are now, but never go back…




- Steepest ascent hill climbing:
  - instead of moving to the first position that is better than the current one
  - pick the best position out of all the next possible moves

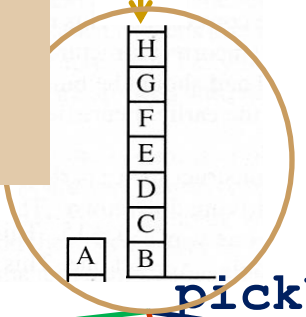
# Example: Hill Climbing with Blocks World




*h(n) = 2*

hill-climbing will stop, because all successors are higher than parent… --> local minimum

**pickUp(A)+putOnTable(A)**

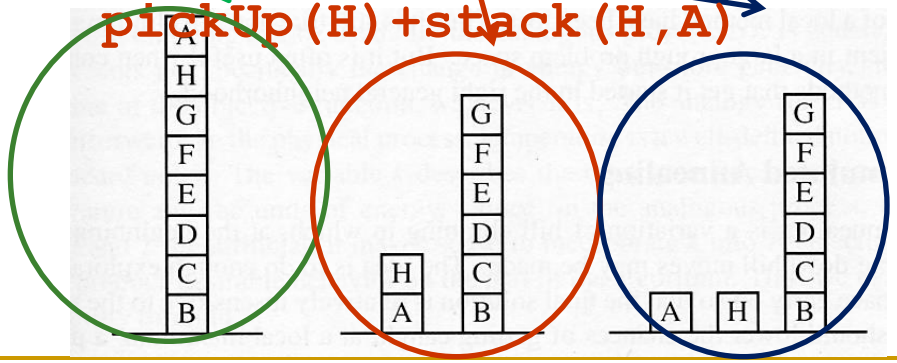



*h(n) = 1*

**pickUp(A)+stack(A,H)**

**pickUp(H)+putOnTable(H)**

**pickUp(H)+stack(H,A)**




*h(n) = 2* *h(n) = 2* *h(n) = 2*

Don't be confused…
a lower h(n) is better…

So with lower h(n), *hill-climbing* should really be called *deep-diving* ☺

# Problems with Hill Climbing

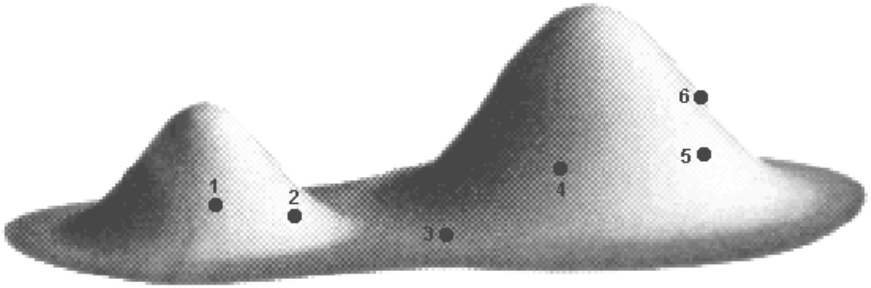
- Foothills (or local maxima)
  - reached a local maximum, not the global maximum
  - a state that is better than all its neighbors but is not better than some other states farther away.
  - at a local maximum, all moves appear to make things worse.
  - ex: 8-puzzle: we may need to move tiles temporarily out of goal position in order to place another tile in goal position
  - ex: TSP: "nearest neighbour" heuristic

# Problems with Hill Climbing

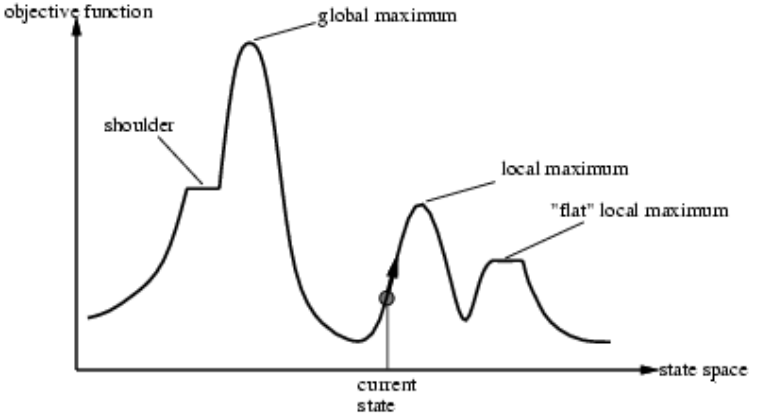
- Plateau
  - a flat area of the search space in which the next states have the same value.
  - it is not possible to determine the best direction in which to move by making local comparisons.
  - can cause the algorithm to stop (or wander aimlessly).

# Some Solutions to Hill-Climbing

- Random-restart hill-climbing
  - random initial states are generated
  - run each until it halts or makes no significant progress.
  - the best result is then chosen.




- keep going even if the best successor has the same value as current node
  - works well on a "shoulder"
  - but could lead to infinite loop on a plateau



# Summary of Hill climbing

- The characteristics of the heuristic function and the initial solution determine the success and the time of the search
- The strategy of this algorithm may end the search in a solution that is only apparently the optimal
- Problems
  - Local optima: No neighbor solution has a better value
  - Plateaus: All neighbours have the same value
  - Ridge: A sequence of local optima

- Possible solutions
  - Backtrack to a previous solution and follow another path (it is only possible if we limit the memory used for backtracking, *Beam Search*)
  - Restart the search from another initial solution looking for a better solution (*Random-restarting Hill-Climbing*)
  - Use two or more actions to explore deeper the neighbourhood after making any decision (expensive in time and space)
  - Parallel Hill-Climbing (for instance: divide the search space in regions and explore the most promising ones, possibly sharing information)

# Iterative Best Improvement (IBI)

- In **iterative best improvement**, the neighbor of the current selected node is one that optimizes some **evaluation function**.

In **greedy descent**, a neighbor is chosen to minimize an evaluation function (**hill climbing** or **greedy ascent)** when the aim is to maximize. We only consider minimization; if you want to maximize a quantity, you can minimize its negation.

- IBI requires a way to evaluate each total assignment. For CSPs, a common evaluation function is the number of constraints that are violated by the total assignment that is to be minimized.
  - A violated constraint is called a **conflict**.
  - The evaluation function : #conflicts,
  - A solution is a total assignment with an evaluation of zero.
  - Sometimes the evaluation function is refined by weighting some constraints more than others.

# Iterative Best Improvement (IBI)

A local optimum is an assignment such that no neighbor improves the evaluation function. This is also called a local minimum in greedy descent, or a local maximum in greedy ascent.

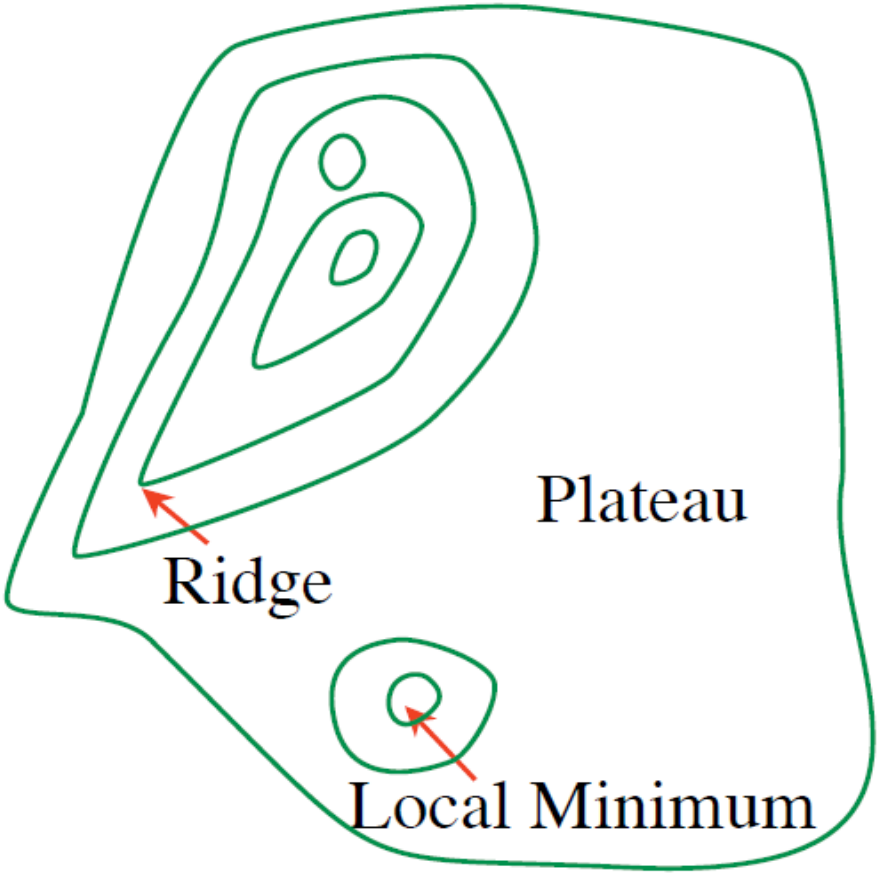
- Aim: find an assignment with zero unsatisfied constraints.
- Given an assignment of a value to each variable, a conflict is an unsatisfied constraint.
- The goal is an assignment with zero conflicts.
- Heuristic function to be minimized: the number of conflicts.

# Complex Domains

- When the domains are small or unordered, the neighbors of an assignment can correspond to choosing another value for one of the variables.
- When the domains are large and ordered, the neighbors of an assignment are the adjacent values for one of the variables.
- If the domains are continuous, Gradient descent changes each variable proportional to the gradient of the heuristic function in that direction.
  The value of variable $X_i$ goes from $v_i$ to $v_i - \eta \frac{\partial h}{\partial X_i}$.
  $\eta$ is the step size.

# Problems with Greedy Descent

- a local minimum that is not a global minimum
- a plateau where the heuristic values are uninformative
- a ridge is a local minimum where *n*-step look-ahead might help



# Randomized Greedy Descent

As well as downward steps we can allow for:

- Random steps: move to a random neighbor.
- Random restart: reassign random values to all variables.

Which is more expensive computationally?

# Today

- Features and States
- Constraint Satisfaction problems
- Generate-and-Test Algorithms
- Solving CSPS Using Search
- Consistency Algorithms
- Domain Splitting
- Local Search
  - Hill Climbing
- Stochastic Local Search
- Simulated Annealing
- Population based methods
  - Beam search:
  - Stochastic beam search:
  - Genetic algorithms:

# Stochastic Local Search

Stochastic local search is a mix of:

- Greedy descent: move to a lowest neighbor
- Random walk: taking some random steps
- Random restart: reassigning values to all variables

# Random Walk

Variants of random walk:

- When choosing the best variable-value pair, randomly sometimes choose a random variable-value pair.
- When selecting a variable then a value:
  - Sometimes choose any variable that participates in the most conflicts.
  - Sometimes choose any variable that participates in any conflict (a red node).
  - Sometimes choose any variable.
- Sometimes choose the best value and sometimes choose a random value.

# Comparing Stochastic Algorithms

- How can you compare three algorithms when
  - one solves the problem 30% of the time very quickly but doesn't halt for the other 70% of the cases
  - one solves 60% of the cases reasonably quickly but doesn't solve the rest
  - one solves the problem in 100% of the cases, but slowly?
- Summary statistics, such as mean run time, median run time, and mode run time don't make much sense.

# Other local search algorithms

- There are other local search algorithms with different inspirations like physics or biology:
  - **Simulated annealing:** Stochastic Hill-climbing inspired in the controlled cooling of metal alloys and substances dissolution
  - **Genetic Algorithms:** Parallel stochastic Hill-climbing inspired in the mechanism of natural selection

# Today

- Features and States
- Constraint Satisfaction problems
- Generate-and-Test Algorithms
- Solving CSPS Using Search
- Consistency Algorithms
- Domain Splitting
- Local Search
  - Hill Climbing
- Stochastic Local Searc
- Simulated Annealing
- Population based methods
  - Beam search:
  - Stochastic beam search:
  - Genetic algorithms:

# Simulated Annealing - Methodology

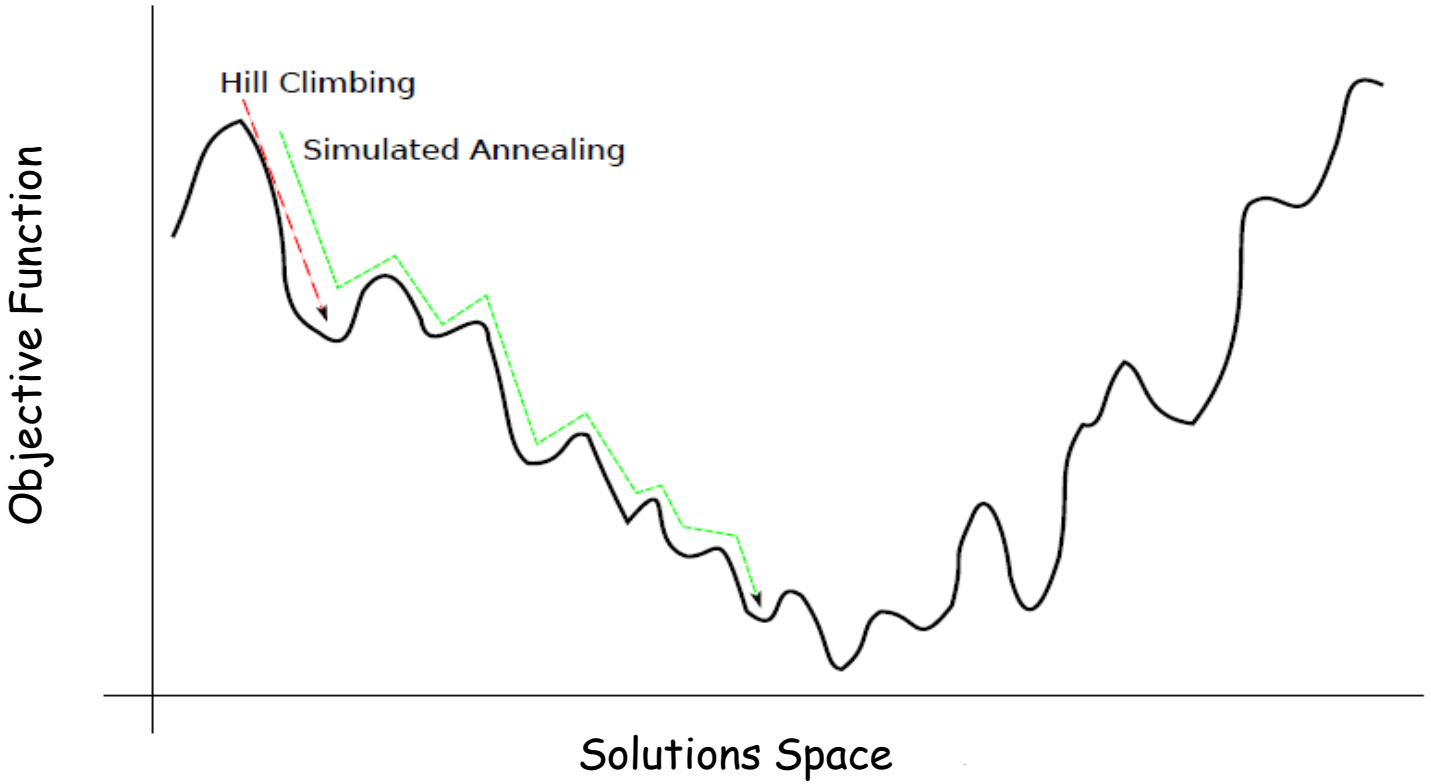
- We have to identify the elements of the problem with the elements of the physics analogy
- **Temperature**, control parameter
- **Energy**, quality of the solution $f(n)$
- **Acceptance function**, allows to decide if to pick a successor solution
  - $\mathcal{F}(\Delta f, T)$, function of the temperature and the difference of quality between the current solution and the candidate solution
  - The lower temperature, the lower the chance to choose a successor with worst evaluation
- **Cooling strategy**, number of iterations to perform, how to lower the temperature and how many successors to explore each temperature step

# Simulated annealing - canonical algorithm

**Algorithm:** Simulated Annealing

An initial temperature is chosen
**while** *temperature above zero* **do**
    `// Random walk the space of solutions`
    **for** *the chosen number of iterations* **do**
        NewSol $\leftarrow$ generate_random_successor(CurrentSol)
        $\Delta E \leftarrow f(CurrentSol) - f(NewSol)$
        **if** $\Delta E > 0$ **then**
            CurrentSol $\leftarrow$ NewSol
        **else**
            with probability $e^{\Delta E/T}$: CurrentSol $\leftarrow$ NewSol
        **end**
    **end**
    Reduce the temperature
**end**

# Simulated Annealing



# Simulated Annealing - Applications

- Used for combinatorial optimization problems (optimal configuration of a set of components) and continuous optimization (optimal in a N-dimensional space)
- Adequate for large sized problems in which the global optimal could be surrounded by lots of local optimums
- Adequate for problems where to find a discriminant heuristic is difficult (a random choice is as good as any other choice)
- Applications: TSP, Design of VLSI circuits
- Problems: To determine the value of the parameters of the algorithm requires experimentation (sometimes very extensive)

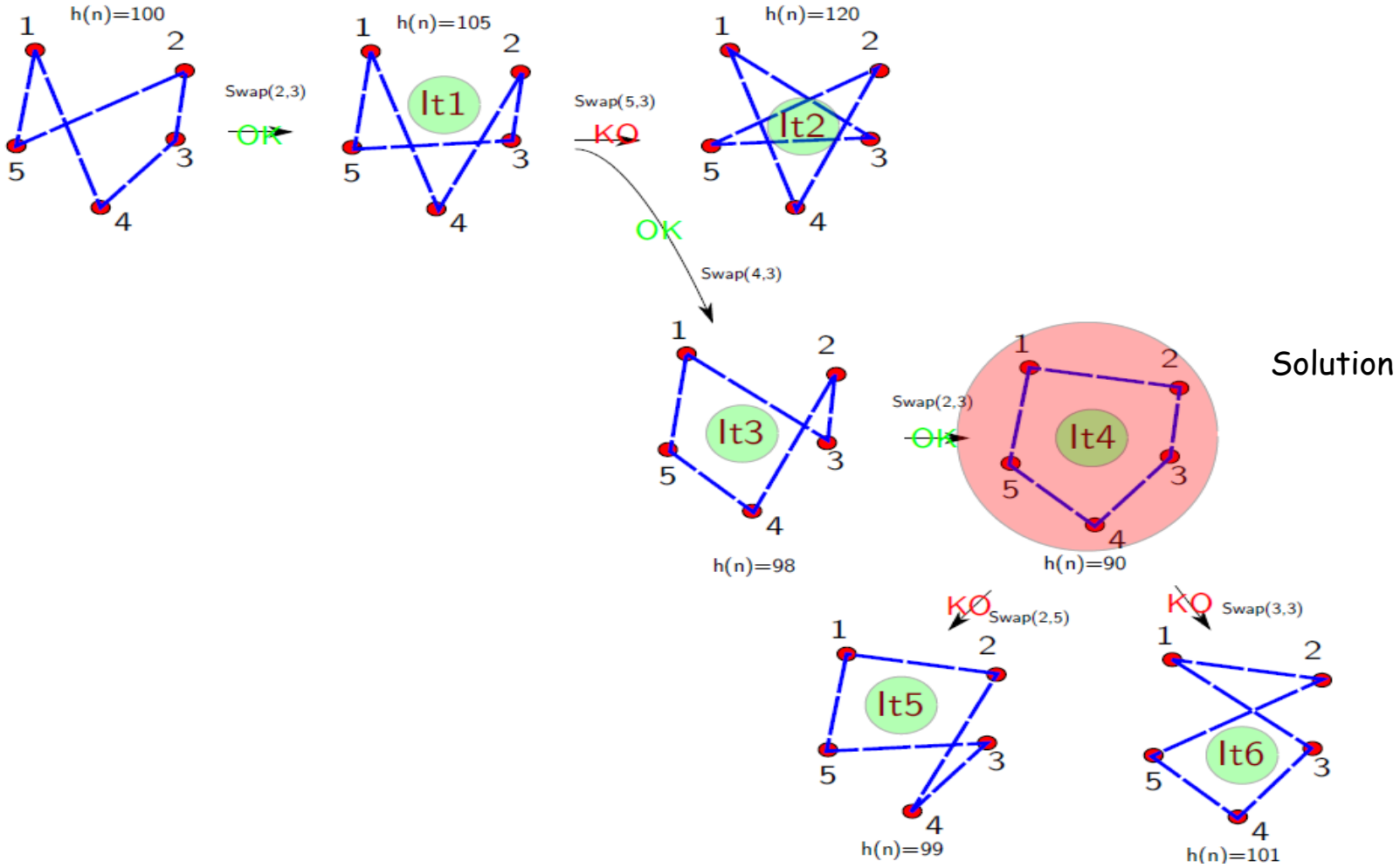
# Simulated annealing - Example - TSP

- Traveler salesman problem (TSP): Search space $N!$
- Possible actions to change a solution: Inversions, translation, interchange
- An energy function (Sum of the distance among the cities, following the order in the solution)

$$E = \sum_{i=1}^{n} \sqrt{(x_i - x_{i+1})^2 + (y_i - y_{i+1})^2} + \sqrt{(x_N - x_1)^2 + (y_N - y_1)^2}$$

- We should define an initial temperature (experimentation)
- We should determine the number of iterations for each temperature step and how is the temperature decreased

# Simulated annealing - Example - TSP



# Tabu lists

- To prevent cycling we can maintain a tabu list of the $k$ last assignments.
- Don't allow an assignment that is already on the tabu list.
- If $k = 1$, we don't allow an assignment of to the same value to the variable chosen.
- We can implement it more efficiently than as a list of complete assignments.
- It can be expensive if $k$ is large.

# Today

- Features and States
- Constraint Satisfaction problems
- Generate-and-Test Algorithms
- Solving CSPS Using Search
- Consistency Algorithms
- Domain Splitting
- Local Search
  - Hill Climbing
- Stochastic Local Search
- Simulated Annealing
- Population based methods
  - Beam search:
  - Stochastic beam search:
  - Genetic algorithms:

# Population based methods: Parallel Search

- So far the local search algorithms maintain a single current assignment. We can considers algorithms that maintain multiple assignments.
- In these algorithms, a total assignment of a value to each variable is called an **individual** and the set of current individuals is a **population**.
- Beam search: maintains the best $k$ assignments.
- Stochastic beam search: selects which assignments to propagate stochastically.
- Genetic algorithms: inspired by biological evolution, the $k$ assignments forming a population interact in various ways to produce the new population.

## Population Methods: Parallel Search

A total assignment is called an individual.

- Idea: maintain a population of $k$ individuals instead of one.
- At every stage, update each individual in the population.
- Whenever an individual is a solution, it can be reported.
- Like $k$ restarts, but uses $k$ times the minimum number of steps.

# Beam Search

- Like parallel search, with $k$ individuals, but choose the $k$ best out of all of the neighbors.
- When $k = 1$, it is greedy descent.
- When $k = \infty$, it is breadth-first search.
- The value of $k$ lets us limit space and parallelism.

# Stochastic Beam Search

- Like beam search, but it probabilistically chooses the $k$ individuals at the next generation.
- The probability that a neighbor is chosen is proportional to its heuristic value.
- This maintains diversity amongst the individuals.
- The heuristic value reflects the fitness of the individual.
- Like asexual reproduction: each individual mutates and the fittest ones survive.

# GA: Genetic Algorithms (1/4)

- Inspired in the mechanisms of natural selection
  - All living things adapt to environment because of the characteristics inherited from their parents
  - The probability of survival and reproduction is related to the quality of these characteristics (fitness of the individual to the environment)
  - The combination of good individuals could result in better adapted individuals
- We can translate this analogy to local search
  - The solutions are individuals
  - The fitness function indicates the quality of the solution
  - Combining good solutions we could obtain better solutions

# Genetic algorithms (2/4)

To solve a problem using GA we need:

- To code the characteristics of the solutions (for example as a binary string)
- A function that measures the quality of a solution (fitness function)
- Operators that combine solutions to obtain new solutions (crossover operations)
- The number of individuals in the initial population
- An strategy about how to match the individuals

# Genetic Algorithms (3/4)

- Learning technique based on evolution
- Also called evolutionary algorithm
- Based on a biological metaphor (like neural networks)
- Learning = competition among a population of evolving candidate solutions to a problem.
- A *fitness function* evaluates each solution to decide if it will contribute to the next generation of solutions
- Through *genetic operators* the algorithm creates a new population of candidate solutions from the previous generation

# Genetic Algorithms (4/4)

- Let P(t) be the set of candidate solutions at time t

- Set time t ← 0
- Initialize the population P(t) *// typically, chosen at random*
- While P(t) does not include an acceptable solution
  - Evaluate the fitness of each member of the population P(t)
  - Select pairs of solutions from P(t) based on fitness
  - Produce the offspring of these pairs using genetic operators
  - Replace the weakest candidates (based on fitness) of P(t) with these offspring
  - Mutation (optional)
  - Set time t ← t+1

# GA: Solution Representation

- Solutions are typically represented as a fixed-length string using a finite alphabet
  - In real DNA, the alphabet is AGTC (adenine, guanine, thymine, cytosine)
    - Ex: *AATAGC*
  - In machine learning, the alphabet is typically {0,1,#}
    (where # means that the feature is not relevant)
    - Ex: the features: {*citizen? Male? English? status?* } : #100

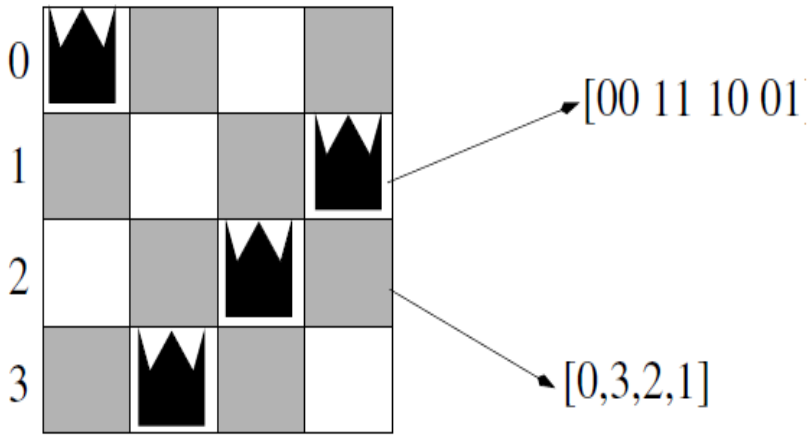
- Each element of the string is called a *gene*

# The Fitness Function

- To determine if a candidate solution is good or not
- Exact function depends a lot on the problem
- But very often, the function computes the proportion of the examples that are correctly classified by the candidate solution
  - How many training examples are correctly classified with: #110, 100#, …

# Genetic algorithms - Coding

- Usually the coding of the individuals is a binary string (it is not always the best)




- The coding defines the size of the search space and the crossover operators that are needed

# Genetic Operators

- Crossover
  - Takes 2 candidates and swaps components to produce 2 new candidates

1#0#

+

0#10

---

1#10 0#0#

- Mutation
  - Takes a single candidate and randomly changes a gene
  - Important because the initial population may not include an essential gene

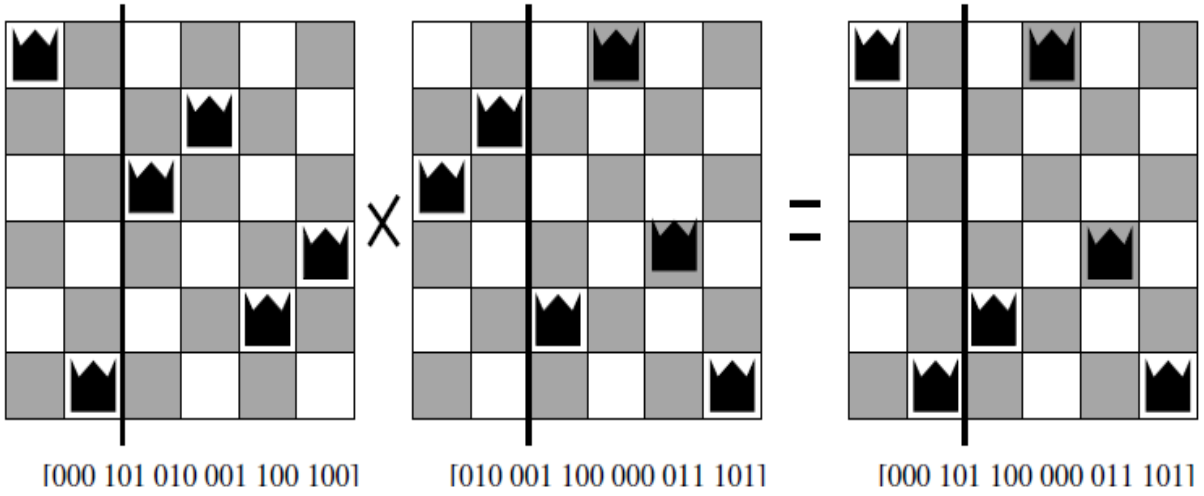
0#10

---

0#11

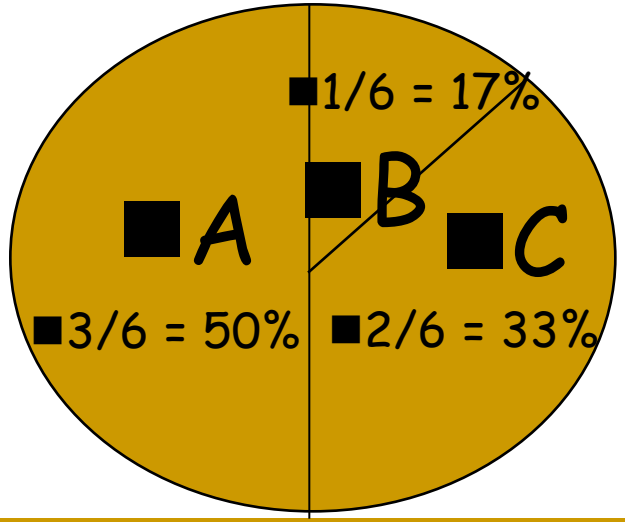
# Genetic algorithms - Operators

- The combination of individuals is done using crossover operators
- The basic operator is the one-point crossover
  - A cutting point in the coding is chosen randomly
  - The information of the two individuals is interchanged using this point



# Proportional Selection

- Main idea: better individuals get higher chance
  - Selection of parent is proportional to fitness score
  - assume 000110010111 correctly predicts 3 outputs over 6
  - So it has 50% chances to get selected for cross-over
  - Implementation: roulette wheel technique



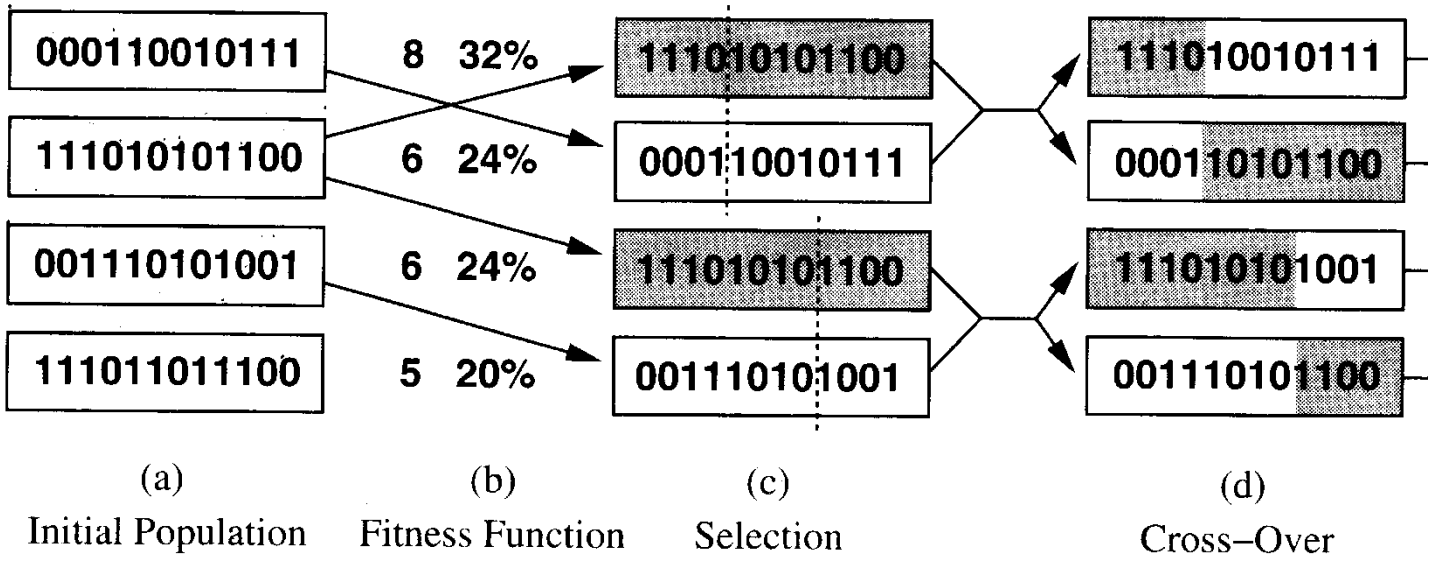

fitness(A) = 3/6

fitness(B) = 1/6

fitness(C) = 2/6

# Proportional Cross-over

- Main idea: better individuals transmit more genes
- Non-equal split cross-over



# Genetic Algorithms

- Applications:
  - Optimization - scheduling problems
  - Heuristic search
  - Machine learning
  - …
- Difficulties:
  - Finding a good representation and fitness function
  - Depends a lot on the problem domain
  - May converge to a local optimum

# Today

- Features and States
- Constraint Satisfaction problems
- Generate-and-Test Algorithms
- Solving CSPs Using Search
- Consistency Algorithms
- Domain Splitting
- Local Search
  - Hill Climbing
- Stochastic Local Search
- Simulated Annealing
- Population based methods
  - Beam search:
  - Stochastic beam search:
  - Genetic algorithms: