Features and Constraints

- Poole, chap. 4

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:

Features and States

- Instead of reasoning explicitly in terms of states, It is often better to describe states in terms of features then reason in terms of these features.
- Often these features are not independent and they are hard constraints that specify legal combinations of assignments of values to variables.
- Example: Planning and Scheduling
  - Agent must assign a time for each action: typically there are:
    - constraints on when actions can be carried out,
    - and constraint specifying that the action must actually achieve a goal.
  - There are also often preferences over values that can be specified in term of soft constraints.
- Here we show how to generate assignments that satisfy a set of hard constraints and how to optimize a collection of soft constraints.

Features and States

- For any practical problem, and agent cannot reason in terms of states, there are simply too many !
- Most problems do not come with an explicit list of states; the states are typically describe implicitly in terms of features.
- When describing a real state space, it is usually more natural to describe the features that make up the state rather than explicitly enumerating the states.
- States and Features: are intertwined we can describe either in terms of the other
  - States can be defined in terms of features: features can be primitive and a state corresponds to an assignment of a value to each feature.
  - Features can be defined in terms of states: states can be primitive and a feature is a function of the states. Given a state, the function returns the value of the feature on that state.
- Each feature has a domain: set of values that can take. The domain of the feature is the range of function on the states.

Features and States

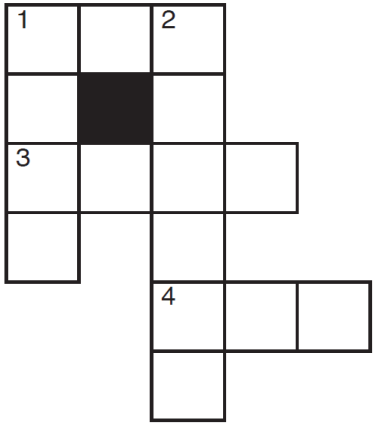
- One main advantage of reasoning in terms of features is the computational saving!
- Example binary feature, the domain has two, many states can be described by a few features:
  - 10 binary features can describe $2^{10}=1,024$ states.
  - 20 binary features can describe $2^{20}=1,048,576$ states.
  - 30 binary features can describe $2^{30}=1,073,741,824$ states.
  - 100 binary features can describe $2^{100}=1,267,650,600,228,229,401,496,703,205,376$ states.
- Reasoning in terms of 30 features may be easier than reasoning in terms of more than a billion states!
- Typically the features are not independent, in that there may be constraints on the value of different features.
  - one problem is to determine what states are possible given the features and the constraints.

Possible Worlds: Variables and Constraints

- For simplified and general formalism, features are considered without considering time explicitly→CSP description in terms of possible worlds.
  - When we are not modeling change, there is a direct 1to one mapping between features and variables, and between states and possible worlds.
  - A possible world is a possible way the world(real or imaginary) could be.
  - Possible worlds are described by algebraic variables *V*: symbol used to denote features of possible worlds: can be discrete or continuous.
  - Each variable V has an associated domain, *dom (V)*: set of values the variable can take on

Example: cross-word puzzle

- There are two representation of crossword puzzle in terms of variable:



Words:

ant, big, bus, car, has
book, buys, hold,
lane, year
beast, ginger, search,
symbol, syntax

Two variable A with dom {1,2,3} and B with dom {true, false} there are six possible worlds:

- w_0: $A=0$ and $B=true$
- w_1: $A=0$ and $B=false$
- w_2: $A=1$ and $B=true$
- w_3: $A=1$ and $B=false$
- w_4: $A=2$ and $B=true$
- w_5: $A=2$ and $B=false$

Representation 1:

variables are the numbered squares with the direction of the word(↑or ↓) and
the domains are the set of possible words that can be put in.
A possible world correspond to an assignment of word for each of the variables.

Representation 2: generic or descriptional?

variables are the individual squares and
the domain of each variable is the set of letters in the alphabet.
A possible word corresponds to an assignment of letters to each square.

- Possible worlds can be defined in terms of variables or variables can be defined in terms of possible worlds.
- Worlds can be primitive and variable is a function from possible worlds into the domain of variable; given a possible world, the function returns the value of that variable in that possible world

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:

Posing a Constraint Satisfaction Problem

A CSP is characterized by

- A set of variables $V_1, V_2, \ldots, V_n$.
- Each variable V_i has an associated domain $\mathbf{D}_{V_i}$ of possible values.
- There are hard constraints on various subsets of the variables which specify legal combinations of values for these variables.
- A solution to the CSP is an assignment of a value to each variable that satisfies all the constraints.

Example: scheduling activities

- Variables: A, B, C, D, E that represent the starting times of various activities.
- Domains: $\mathbf{D}_A = \{1, 2, 3, 4\}$, $\mathbf{D}_B = \{1, 2, 3, 4\}$, $\mathbf{D}_C = \{1, 2, 3, 4\}$, $\mathbf{D}_D = \{1, 2, 3, 4\}$, $\mathbf{D}_E = \{1, 2, 3, 4\}$
- Constraints:

$$(B \neq 3) \wedge (C \neq 2) \wedge (A \neq B) \wedge (B \neq C) \wedge (C < D) \wedge (A = D) \wedge (E < A) \wedge (E < B) \wedge (E < C) \wedge (E < D) \wedge (B \neq D).$$

- The aim is to choose a value for each variable so that the resulting possible world satisfies the constrains, that is we want a model of the constraints.
  - A model is a possible world that satisfies all the constraints.
  - A possible world *w* satisfies a set of constraints if, for every constraint the value assigned in *w* to the variables in the scope of the constraint satisfy the constraint.

Constraint Satisfaction problems

Given a CSP, there are a number of tasks that can be performed:

- Determine whether or not there is a model.
- Find a model.
- Find all of the models or enumerate the models.
- Count the number of models.
- Find the best model, given a measure of how good models are;
- Determine whether some statement holds in all models.

- Here we consider the problem of finding a model. Some of the methods can also determine if there is no solution.
- What may be more surprising is that some of the methods can find a model if one exists, but they cannot tell us that there is no model if none exists.
- CSPs are very common, so it is worth trying to find relatively efficient ways to solve them.
- Determining whether there is a model for a CSP with finite domains is NP-hard and no known algorithms exist to solve such problems that do not use exponential time in the worst case. ►just because a problem is NP-hard does not mean that all instances are difficult to solve. Many instances have structure that can be exploited.

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:

Generate-and-Test Algorithm

- Generate the assignment space $\mathbf{D} = \mathbf{D}_{V_1} \times \mathbf{D}_{V_2} \times \ldots \times \mathbf{D}_{V_n}$. Test each assignment with the constraints.
- Example:

$$\begin{aligned}\mathbf{D} &= \mathbf{D}_A \times \mathbf{D}_B \times \mathbf{D}_C \times \mathbf{D}_D \times \mathbf{D}_E \\ &= \{1,2,3,4\} \times \{1,2,3,4\} \times \{1,2,3,4\} \\ &\quad \times \{1,2,3,4\} \times \{1,2,3,4\} \\ &= \{\langle 1,1,1,1,1\rangle, \langle 1,1,1,1,2\rangle, \ldots, \langle 4,4,4,4,4\rangle\}.\end{aligned}$$

- How many assignments need to be tested for n variables each with domain size d?

In this case there are $|\boldsymbol{D}| = 4^5 = 1{,}024$ different assignments to be tested.

If each of the n variable domains has size d, then $\boldsymbol{D}$ has d^n elements. If there are e constraints, the total number of constraints tested is $O(ed^n)$. As n becomes large, this very quickly becomes intractable, and so we must find alternative solution methods.

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:

Solving CSPs Using Search

- Generate-and-test algorithms assign values to all variables before checking the constraints.
- Because individual constraints only involve a subset of the variables, some constraints can be tested before all of the variables have been assigned values.
- If a partial assignment is inconsistent with a constraint, any complete assignment that extends the partial assignment will also be inconsistent.
- In the previous scheduling problem:
  - *A=1* and *B=1* are inconsistent with the constraint *A≠B* regardless of the values of the other variables.
  - If the variables *A* and *B* are assigned values first, this inconsistency can be discovered before any values are assigned to *C*, *D*, or *E*, thus saving a large amount of work.

Backtracking Algorithms

- Systematically explore **D** by instantiating the variables one at a time
- evaluate each constraint predicate as soon as all its variables are bound
- any partial assignment that doesn't satisfy the constraint can be pruned.

Example Assignment $A = 1 \wedge B = 1$ is inconsistent with constraint $A \neq B$ regardless of the value of the other variables.

- Searching with a depth-first search, typically called **backtracking**, can be much more efficient than generate and test.
- Generate and test is equivalent to not checking constraints until reaching the leaves.
- Checking constraints higher in the tree can prune large subtrees that do not have to be searched.

CSP as Graph Searching

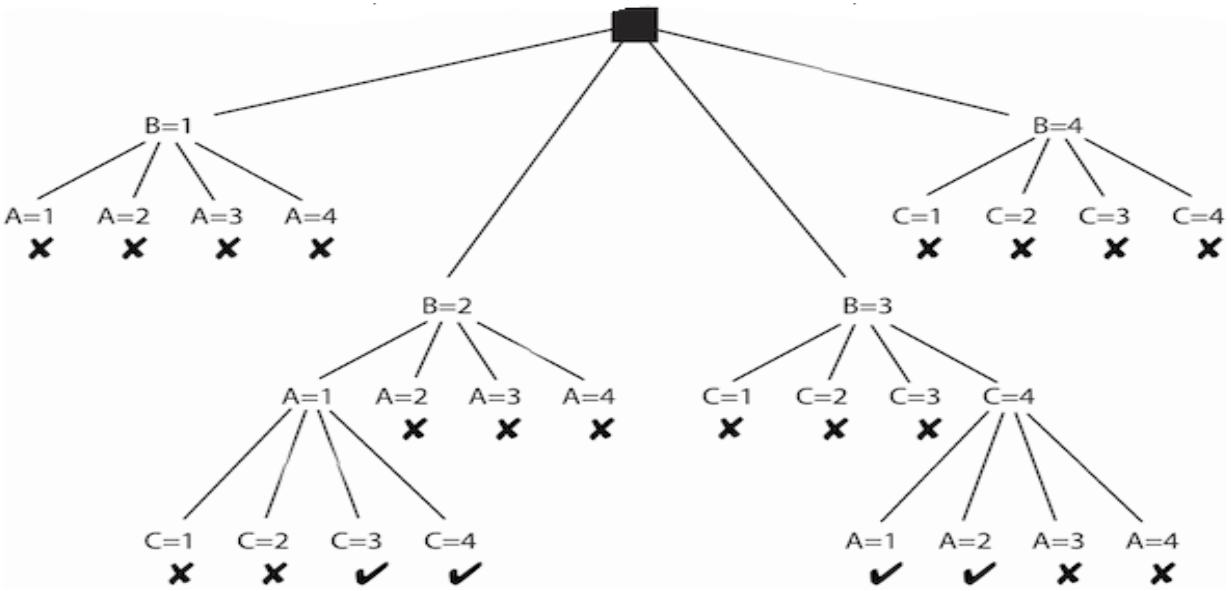
A CSP can be solved by graph-searching:

- A node is an assignment values to some of the variables.
- Suppose node N is the assignment $X_1 = v_1, \ldots, X_k = v_k$.
  Select a variable Y that isn't assigned in N.
  For each value $y_i \in dom(Y)$
  $X_1 = v_1, \ldots, X_k = v_k, Y = y_i$ is a neighbour if it is consistent with the constraints.
- The start node is the empty assignment.
- A goal node is a total assignment that satisfies the constraints.

Example

- CSP Problem: Suppose you have a CSP with the variables *A*, *B*, and *C*, each with domain *{1,2,3,4}*.

  Suppose the constraints are *A<B* and *B<C*. A possible search tree is:



- This CSP has four solutions. The leftmost one is *A=1, B=2, C=3*.
- The size of the search tree, and thus the efficiency of the algorithm, depends on which variable is selected at each time.
- A static ordering, such as always splitting on *A* then *B* then *C*, is less efficient than the dynamic ordering used here. The set of answers is the same regardless of the variable ordering.

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:

Consistency Algorithms

- Although depth-first search over the search space of assignments is usually a substantial improvement over generate and test, it still has various inefficiencies that can be overcome.

- Idea: prune the domains as much as possible before selecting values from them.
- A variable is domain consistent if no value of the domain of the node is ruled impossible by any of the constraints.
- Example: Is the scheduling example domain consistent? $\mathbf{D}_B = \{1, 2, 3, 4\}$ isn't domain consistent as $B = 3$ violates the constraint $B \neq 3$.
  - Solution delete 3 from the domain

Constraint Network

The consistency algorithms are best thought of as operating over the network of constraints formed by the CSP:

- There is a oval-shaped node for each variable.
- There is a rectangular node for each constraint.
- There is a domain of values associated with each variable node.
- There is an arc from variable X to each constraint that involves X.

Example 1: we have three variables *A*, *B*, and *C*, each with domain *{1,2,3,4}*. The constraints are *A<B* and *B<C*. The constraint network shows that there are four arcs:

(A, A<B)

(B, A<B)

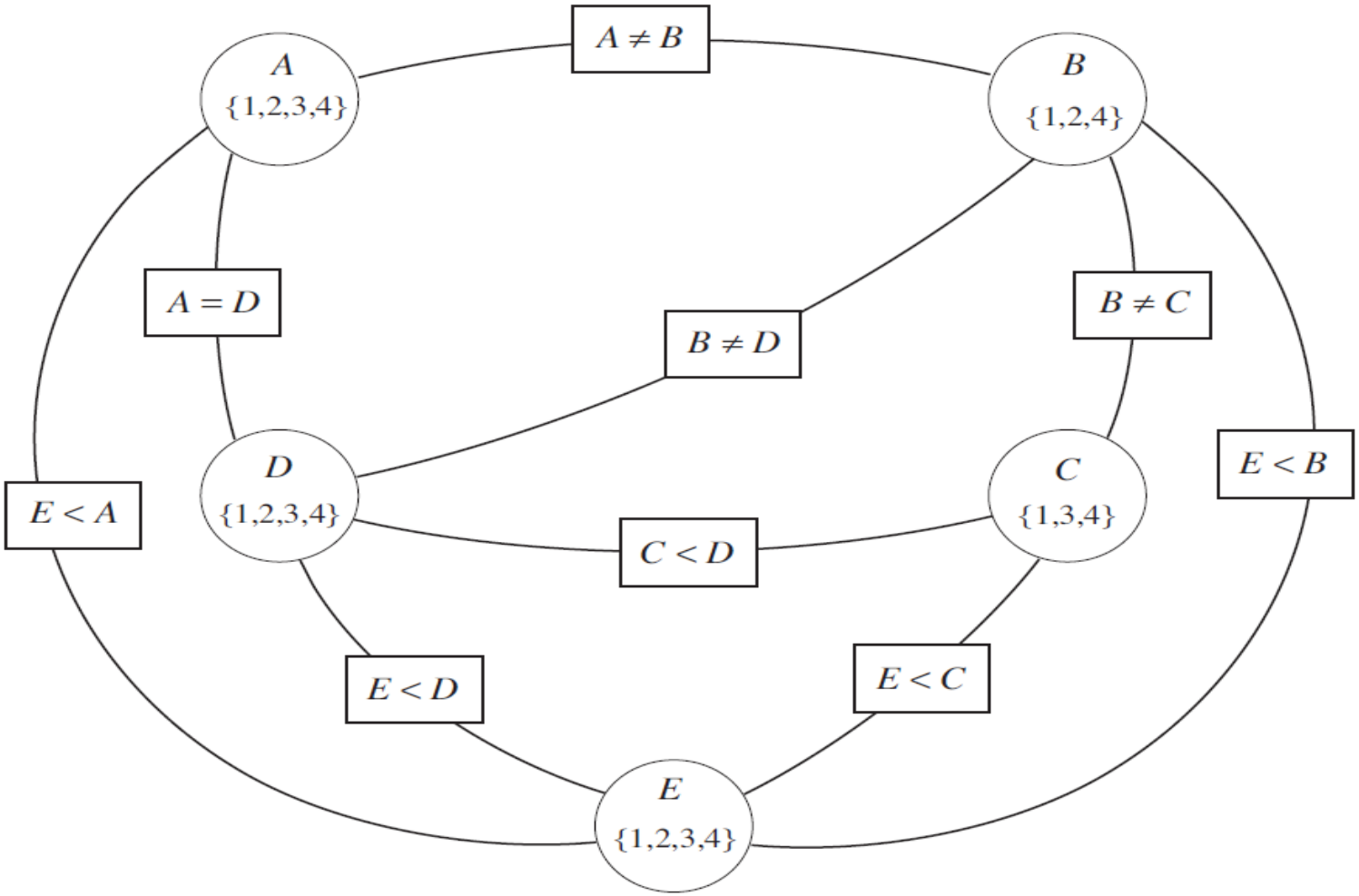
(B, B<C)

(C, B<C)

None of the arcs are arc consistent:
1st arc: is not arc consistent: *A=4* there is no corresponding value for *B* for which *A<B*. If *4* were removed from the domain of *A*, then it would be arc consistent.
2nd arc: is not arc consistent: there is no corresponding value for *A* when *B=1*.

- If an arc *(X,c)* is *not* arc consistent, there are some values of *X* for which there are no values for $Y_1,\ldots,Y_k$ for which the constraint holds. In this case, all values of *X* in D_X for which there are no corresponding values for the other variables can be deleted from D_X to make the arc *(X,c)* consistent.

Constraint Network for Example: scheduling activities (Slide 9)



Arc Consistency

- An arc $\langle X, r(X, \overline{Y}) \rangle$ is arc consistent if, for each value $x \in dom(X)$, there is some value $\overline{y} \in dom(\overline{Y})$ such that $r(x, \overline{y})$ is satisfied.
- A network is arc consistent if all its arcs are arc consistent.
- What if arc $\langle X, r(X, \overline{Y}) \rangle$ is *not* arc consistent?
  All values of X in $dom(X)$ for which there is no corresponding value in $dom(\overline{Y})$ can be deleted from $dom(X)$ to make the arc $\langle X, r(X, \overline{Y}) \rangle$ consistent.

- The arcs can be considered in turn making each arc consistent.
- When an arc has been made arc consistent, does it ever need to be checked again?

Arc Consistency Algorithm

- The arcs can be considered in turn making each arc consistent.
- When an arc has been made arc consistent, does it ever need to be checked again?
  An arc $\langle X, r(X, \overline{Y}) \rangle$ needs to be revisited if the domain of one of the Y's is reduced.
- Three possible outcomes when all arcs are made arc consistent: (Is there a solution?)
  - One domain is empty $\Longrightarrow$ no solution
  - Each domain has a single value $\Longrightarrow$ unique solution
  - Some domains have more than one value $\Longrightarrow$ there may or may not be a solution

Example (arc with no solution)

It is possible for a network to be arc consistent even though there is no solution.

- Suppose there are three variables, *A*, *B* and *C*, each with the domain *{1,2,3}*. Consider the constraints *A=B*, *B=C*, and *A≠C*.
- This is arc consistent: no domain can be pruned using any single constraint. However, there are no solutions. There is no assignment to the three variables that satisfies the constraints.

Finding solutions when AC finishes

- If some domains have more than one element $\Longrightarrow$ search
- Split a domain, then recursively solve each half.
- It is often best to split a domain in half.
- Do we need to restart from scratch?

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:

Domain Splitting -Case Analysis(1/2)

- Split a problem into a number of disjoint cases and solve each case separately. The set of all solutions to the initial problem is the union of the solutions to each case.
- Simple Case: a binary variable *X* , dom *{t,f}*, so solutions :*X=t* or *X=f*.
  - One way: set *X=t*, find all of the solutions, and then *X=f* and find all solutions. Assigning a value to a variable gives a smaller *reduced* problem to solve.
- Domain of variable with more elements: *A* with dom{1,2,3,4} different way to split it:
  - Split the domain into a case for each value. For example, split *A* into the four cases *A=1*, *A=2*, *A=3*, and *A=4*.
  - Always split the domain into two disjoint subsets. For example, split *A* into the two cases *A∈{1,2}* and the case *A∈{3,4}*.
- Notice: The first approach makes more progress with one split, but the second may allow for more pruning with fewer steps. For example, if the same values for *B* can be pruned whether *A* is 1 or 2, the second case allows this fact to be discovered once and not have to be rediscovered for each element of *A*. This saving depends on how the domains are split.
- Recursively solving the cases using domain splitting, recognizing when there is no solution based on the assignments, is equivalent to the search algorithm (Slide 14)

Domain Splitting -Case Analysis(2/2)

- Effective way to solve a CSP is to use arc consistency to simplify the network before each step of domain splitting:
  - simplify the problem using arc consistency; and,
  - if the problem is not solved, select a variable whose domain has more than one element, split it, and recursively solve each case.

Hard and Soft Constraints

- Given a set of variables, assign a value to each variable that either
  - satisfies some set of constraints: satisfiability problems — "hard constraints"
  - minimizes some cost function, where each assignment of values to variables has some cost: optimization problems — "soft constraints"
- Many problems are a mix of hard and soft constraints (called constrained optimization problems).

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

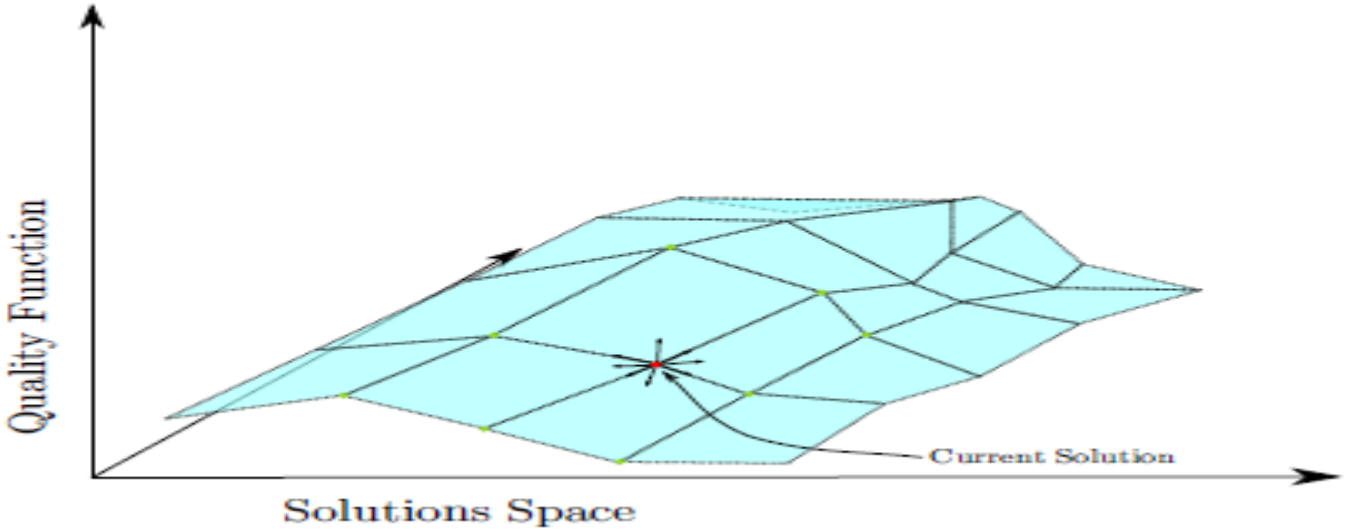
Local Search

- 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

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)

Local Search

- This walk through assignments continues until either a satisfying assignment is found and returned or some stopping criterion is satisfied.
- The stopping criterion decides when to stop the current local search and do a random restart, starting again with a new assignment.
  - A stopping criterion could be: stopping after a certain number of steps.
- This algorithm is not guaranteed to halt. In particular, it goes on forever if there is no solution, and it is possible to get trapped in some region of the search space.
- An algorithm is **complete** if it finds an answer whenever there is one.
- **Random sampling:** here the stopping criterion is always true so that the while loop is never executed. Random sampling keeps picking random assignments until it finds one that satisfies the constraints, and otherwise it does not halt.
- Random sampling is complete: given enough time, it guarantees that a solution will be found if one exists, but there is no upper bound on the time it may take. It is very slow. The efficiency depends only on the product of the domain sizes and how many solutions exist.

Local Search

- **Random walk:** here the while loop is only exited when it has found a satisfying assignment (i.e., the stopping criterion is always false and there are no random restarts). In the while loop it selects a variable and a value at random.
- Random walk is also complete in the same sense as random sampling. Each step takes less time than resampling all variables, but it can take more steps than random sampling, depending on how the solutions are distributed.

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.

Iterative Best Improvement (IBI)

- If the variables have small finite domains, a local search algorithm can consider all other values of the variable when considering a neighbor.
- If the domains are large, the cost of considering all other values may be too high.
- An alternative is to consider only a few other values, often the close values, for one of the variables. Sometimes quite sophisticated methods are used to select an alternative value.
- Local search typically considers the best neighboring assignment even if it is equal to or even worse than the current assignment.
- It is often better to make a quick choice than to spend a lot of time making the best choice. There are many possible variants of which neighbor to choose:

Greedy Descent Variants

To choose a variable to change and a new value for it:

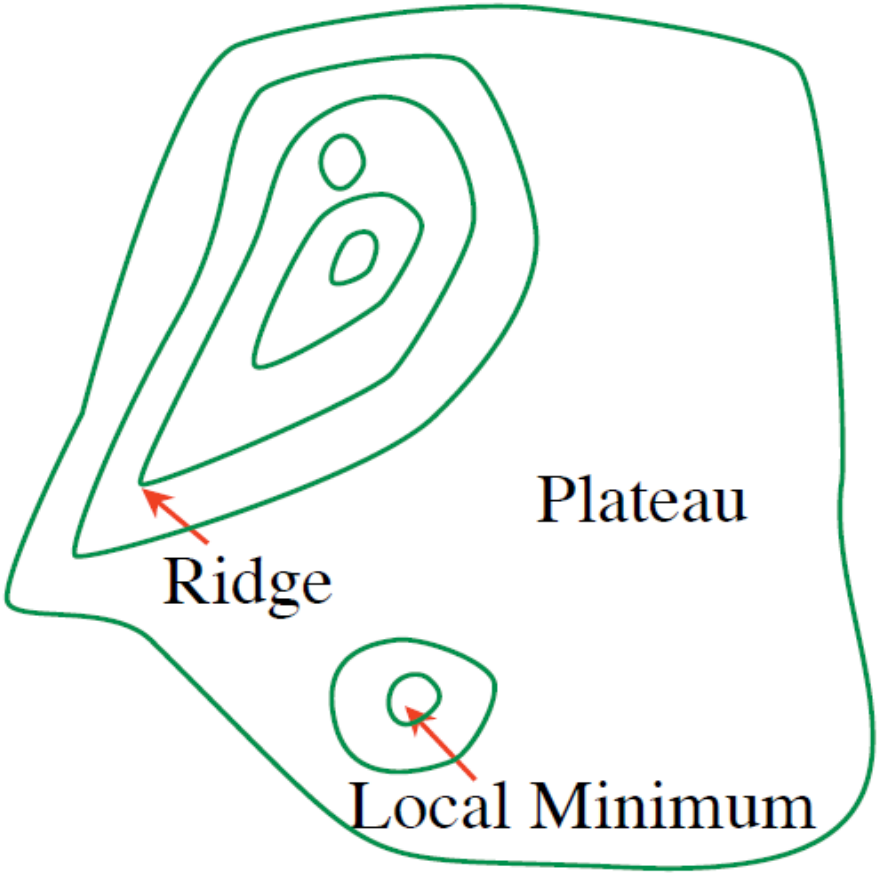
- Find a variable-value pair that minimizes the number of conflicts
- Select a variable that participates in the most conflicts.
  Select a value that minimizes the number of conflicts.
- Select a variable that appears in any conflict.
  Select a value that minimizes the number of conflicts.
- Select a variable at random.
  Select a value that minimizes the number of conflicts.
- Select a variable and value at random; accept this change if it doesn't increase 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 $v_i - \eta \frac{\partial h}{\partial X_i}$.
  η 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?

Randomized Algorithms

- IBI randomly picks one of the best neighbors of the current assignment. Randomness can also be used to escape local minima that are not global minima in two main ways:
- Random Restart: values for all variables are chosen at random. This lets the search start from a completely different part of the search space.
- Random Walk: in which some random steps are taken interleaved with the optimizing steps.
  - with the greedy descent: allows for upward steps that may enable random walk to escape a local minimum that is not a global minimum.
- A random walk is a local random move, whereas a random restart is a global random move. For problems involving a large number of variables, a random restart can be quite expensive.
- A mix of greedy descent with random moves is an instance of a class of algorithms known as **stochastic local search**.

1-Dimensional Ordered Examples

It is very difficult to visualize the search space to understand what algorithms you expect to work because there are often thousands of dimensions, each with a discrete set of values. Some of the intuitions can be gleaned from lower-dimensional problems.

Two 1-dimensional search spaces; step right or left:

- Which method would most easily find the global minimum?
- What happens in hundreds or thousands of dimensions?
- What if different parts of the search space have different structure?

1-Dimensional Ordered Examples

- Find the global minimum in the search space
- a) we would expect the greedy descent with random restart to find the optimal value quickly. Once a random choice has found a point in the deepest valley, greedy descent quickly leads to the global minimum. One would not expect a random walk to work well in this example, because many random steps are required to exit one of the local, but not global, minima.
- b) a random restart quickly gets stuck on one of the jagged peaks and does not work very well.
- A random walk combined with greedy descent enables it to escape these local minima. One or two random steps may be enough to escape a local minimum. Thus, you would expect that a random walk would work better in this search 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:

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
- But also Particle Swarm Optimization, Ant Colony Optimization, Intelligent Water Drop, Gravitational search algorithm, ...

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

- Stochastic Hill-Climbing (a successor is randomly chosen from the neighbor solutions using a probability distribution, the successor could have worst evaluation than the current solution)
- A random walk of the space of solutions is performed
- Inspired in the physics of controlled annealing (crystallization, metal alloys tempering)
- A metal alloy or dissolution is heated at high temperatures and progressively cooled in a controlled way
- If the cooling process is adequate the minimal state of energy of the system is achieved (global minimum)

Simulated Annealing

- Pick a variable at random and a new value at random.
- If it is an improvement, adopt it.
- If it isn't an improvement, adopt it probabilistically depending on a temperature parameter, T.
  - With current assignment n and proposed assignment n' we move to n' with probability $e^{(h(n')-h(n))/T}$
- Temperature can be reduced.

Probability of accepting a change:

Temperature	1-worse	2-worse	3-worse
10	0.91	0.81	0.74
1	0.37	0.14	0.05
0.25	0.02	0.0003	0.000006
0.1	0.00005	2×10^{-9}	9×10^{-14}

Annealing is a process in metallurgy where metals are slowly cooled to make them reach a state of low energy where they are very strong.

Simulated Annealing - Methodology

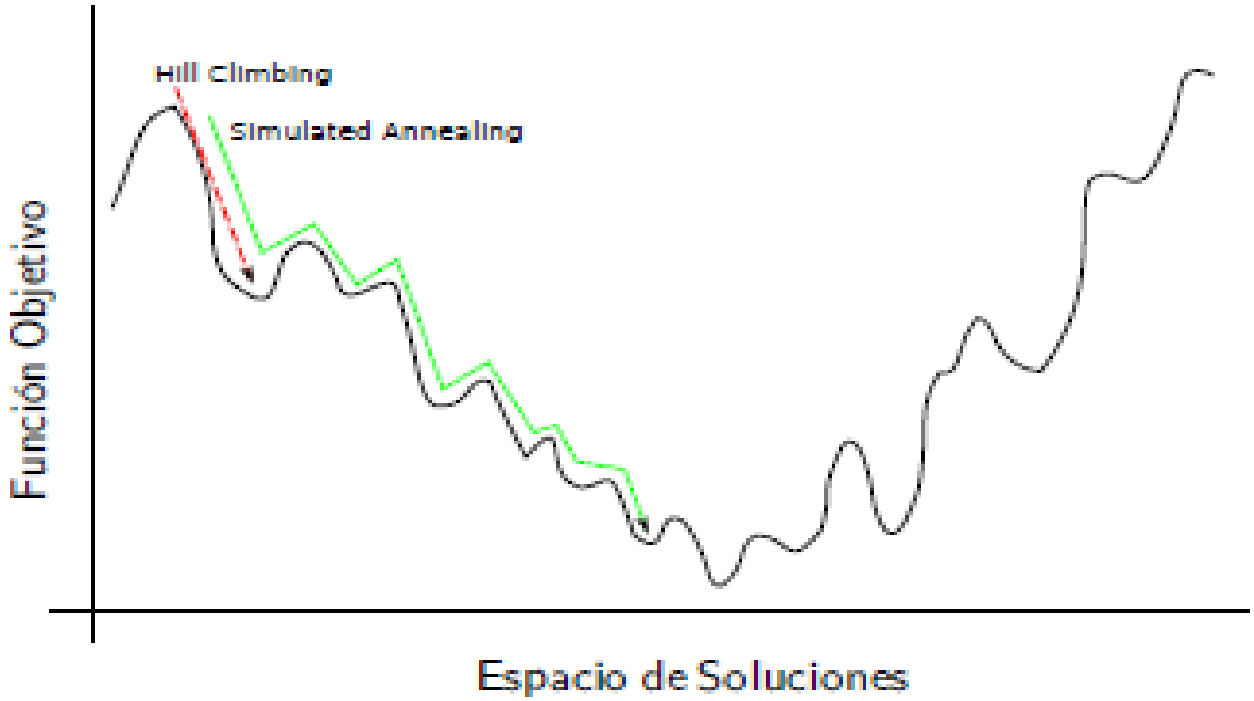
- We have to identify the elements of the problem with the elements of the physics analogy
- **Temperature**, control parameter
- **Energy**, quality of the solution $f(n)$
- **Acceptance function**, allows to decide if to pick a successor solution
  - $\mathcal{F}(\Delta f, T)$, function of the temperature 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)

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(// 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.

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.

Genetic Algorithms

- Like stochastic beam search, but pairs of individuals are combined to create the offspring:
- For each generation:
  - Randomly choose pairs of individuals where the fittest individuals are more likely to be chosen.
  - For each pair, perform a cross-over: form two offspring each taking different parts of their parents:
  - Mutate some values.
- Stop when a solution is found.

Genetic Algorithms

- 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

- 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

Crossover

- Given two individuals:

$$X_1 = a_1, X_2 = a_2, \ldots, X_m = a_m$$

$$X_1 = b_1, X_2 = b_2, \ldots, X_m = b_m$$

- Select i at random.
- Form two offspring:

$$X_1 = a_1, \ldots, X_i = a_i, X_{i+1} = b_{i+1}, \ldots, X_m = b_m$$

$$X_1 = b_1, \ldots, X_i = b_i, X_{i+1} = a_{i+1}, \ldots, X_m = a_m$$

- The effectiveness depends on the ordering of the variables.
- Many variations are possible.

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: