4.1. Local Search Algorithmsand Optimization Problems

In many optimization problems, the path to the goal is irrelvant; the goal state itself is thesolution. The best state is identified from the objective function or heuristic cost function. Insuch cases, we can use local search algorithms (ie) keep only a single current state, try toimproveitinsteadofthewholesearchspaceexploredsofar.

Alocalsearchalgorithmstartsfromasearchspaceandtheniterativelymovestoaneighborsolution.Thisisonlypossibleifaneighborhoodrelationisdefinedonthesearchspace.

Terminationoflocalsearchcanbebasedonatimebound.Anothercommonchoiceisto terminate when the best solution found by the algorithm has not been improved in a givennumberof steps.Local search algorithms are typically incomplete algorithms,as the searchmay stop even if the best solution found by the algorithm is optimal. This can happen even iftermination isdue to the impossibility ofimproving the solution,as the optimalsolution canliefarfromtheneighborhoodofthesolutionscrossedbythealgorithms.

Thelocalsearchproblemisexplainedwiththestatespacelandscape.Alandscapehas:

- location-definedbythestate
- elevation - defined by the value of the heuristic costfunction or objective function.Ifelevationcorrespondstocostthenthelowestvalley*(globalminimum)*isachieved. If elevation corresponds to an objective function, then the highest peak*(globalmaximum)*isachieved.

The informed and uninformed search expands the nodes systematically in two ways:

- keeping different paths in the memory and
- selecting the best suitable path,

Which leads to a solution state required to reach the goal node. But beyond these **"classical search algorithms**," we have some **"local search algorithms" where the path cost does not matters, and only focus on solution-state needed to reach the goal node.**

A local search algorithm completes its task by traversing on a single current node rather than multiple paths and following the neighbors of that node generally.

Although local search algorithms are not systematic, still they have the following two advantages:

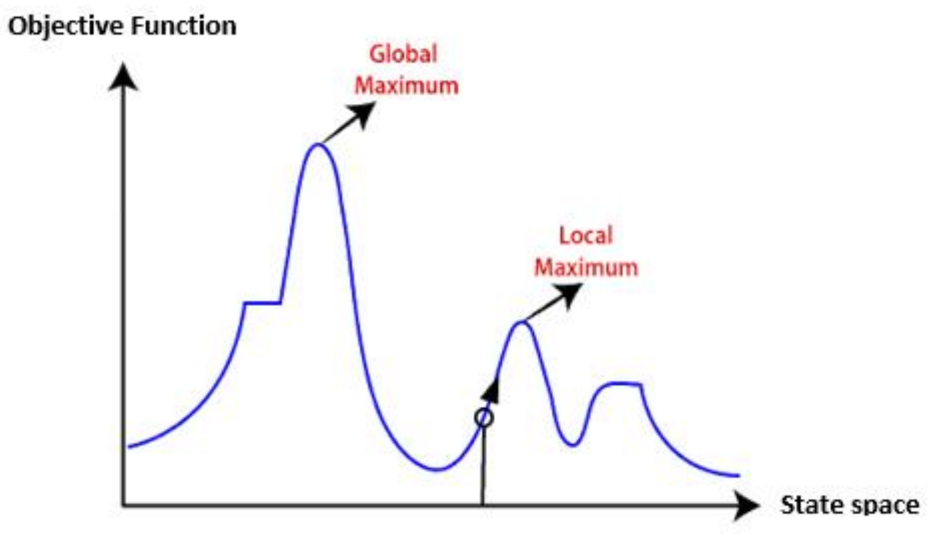
- Local search algorithms use a very little or constant amount of memory as they operate only on a single path.
- Most often, they find a reasonable solution in large or infinite state spaces where the classical or systematic algorithms do not work.

4.2. Working of a Local search algorithm

Let's understand the working of a local search algorithm with the help of an example:

Consider the below state-space landscape having both:

- **Location:** It is defined by the state.
- **Elevation:** It is defined by the value of the objective function or heuristic cost function.



A one-dimensional state-space landscape in which elevation corresponds to the objective function

The local search algorithm explores the above landscape by finding the following two points:

- **Global Minimum:** If the elevation corresponds to the cost, then the task is to find the lowest valley, which is known as **Global Minimum.**
- **Global Maxima:** If the elevation corresponds to an objective function, then it finds the highest peak which is called as **Global Maxima**. It is the highest point in the valley.

4.3. Types

- Hill-climbing Search
- Simulated Annealing
- Local Beam Search

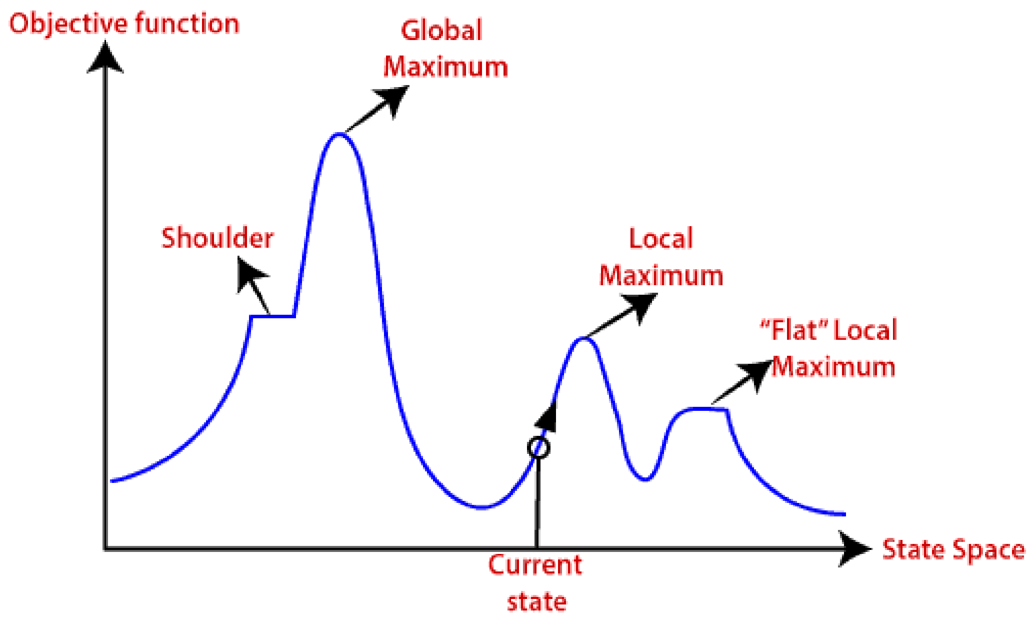
4.3.1. Hill Climbing Algorithm

Hill climbing search is a local search problem. *The purpose of the hill climbing search is to climb a hill and reach the topmost peak/point of that hill.* It is based on the **heuristic search technique** where the person who is climbing up on the hill estimates the direction which will lead him to the highest peak.

State-space Landscape of Hill climbing algorithm

To understand the concept of hill climbing algorithm, consider the below landscape representing the **goal state/peak** and the **current state** of the climber. The topographical regions shown in the figure can be defined as:

- **Global Maximum:** It is the highest point on the hill, which is the goal state.
- **Local Maximum:** It is the peak higher than all other peaks but lower than the global maximum.
- **Flat local maximum:** It is the flat area over the hill where it has no uphill or downhill. It is a saturated point of the hill.
- **Shoulder:** It is also a flat area where the summit is possible.
- **Current state:** It is the current position of the person.

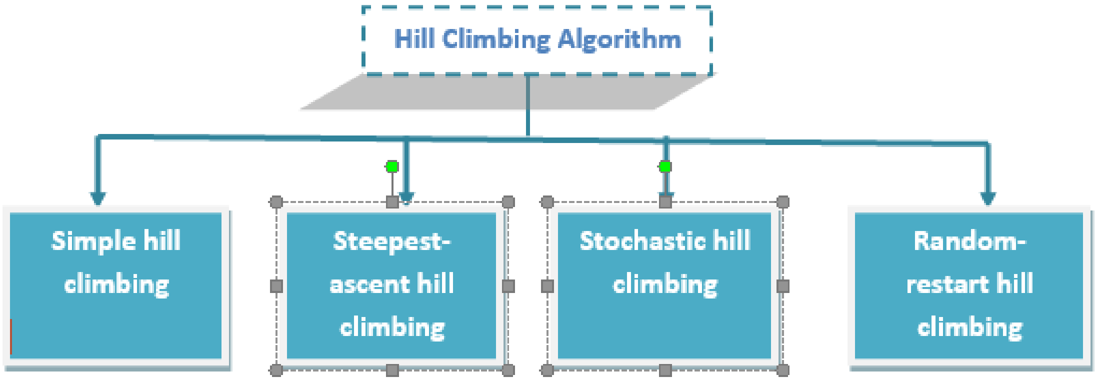


A one-dimensional state-space landscape in which elevation corresponds to the objective function

Types of Hill climbing search algorithm

There are following types of hill-climbing search:

1. Simple hill climbing
2. Steepest-ascent hill climbing
3. Stochastic hill climbing
4. Random-restart hill climbing



1. Simple hill climbing search

Simple hill climbing is the simplest technique to climb a hill. The task is to reach the highest peak of the mountain. Here, the movement of the climber depends on his move/steps. If he finds his next step better than the previous one, he continues to move else remain in the same state. This search focus only on his previous and next step.

Simple hill climbing Algorithm

1. Create a **CURRENT** node, **NEIGHBOUR** node, and a **GOAL** node.
2. If the **CURRENT node=GOAL node**, return **GOAL** and terminate the search.
3. Else **CURRENT node<= NEIGHBOUR node,** move ahead.
4. Loop until the goal is not reached or a point is not found.

2. **Steepest-ascent hill climbing**

Steepest-ascent hill climbing is different from simple hill climbing search. Unlike simple hill climbing search, It considers all the successive nodes, compares them, and choose the node which is closest to the solution. Steepest hill climbing search is similar to **best-first search** because it focuses on each node instead of one.

Note: Both simple, as well as steepest-ascent hill climbing search, fails when there is no closer node.

Steepest-ascent hill climbing algorithm

1. Create a **CURRENT** node and a **GOAL** node.
2. If the **CURRENT node=GOAL** node, return **GOAL** and terminate the search.
3. Loop until a better node is not found to reach the solution.
4. If there is any better successor node present, expand it.
5. When the **GOAL** is attained, return **GOAL** and terminate.

3. **Stochastic hill climbing**

Stochastic hill climbing does not focus on all the nodes. It selects one node at random and decides whether it should be expanded or search for a better one.

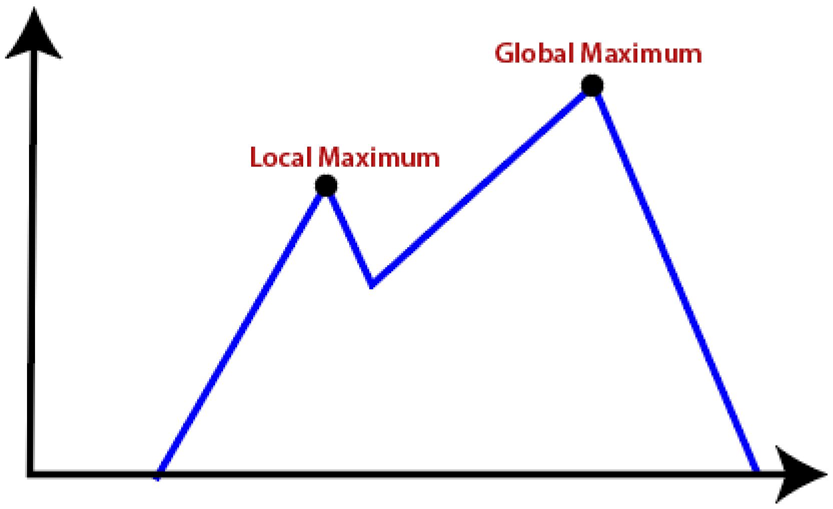
Random-restart hill climbing

Random-restart algorithm is based on **try and try strategy**. It iteratively searches the node and selects the best one at each step until the goal is not found. The success depends most commonly on the shape of the hill. If there are few plateaus, local maxima, and ridges, it becomes easy to reach the destination.

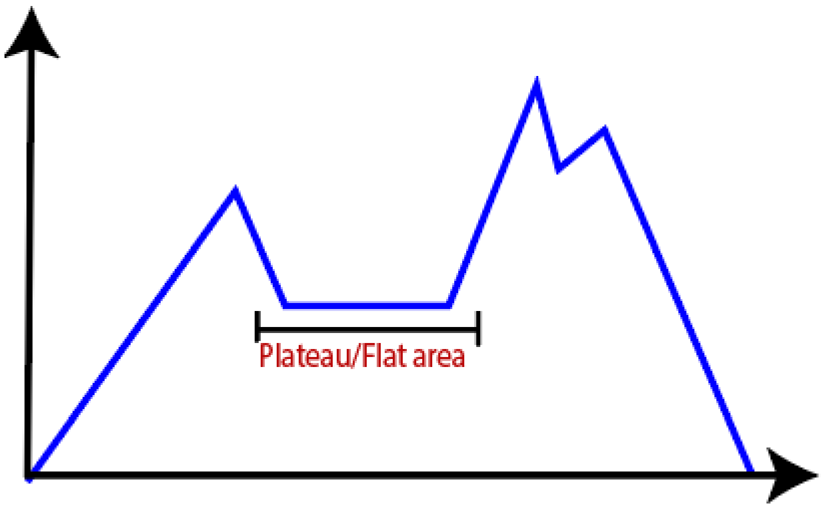
Limitations of Hill climbing algorithm

Hill climbing algorithm is a fast and furious approach. It finds the solution state rapidly because it is quite easy to improve a bad state. But, there are following limitations of this search:

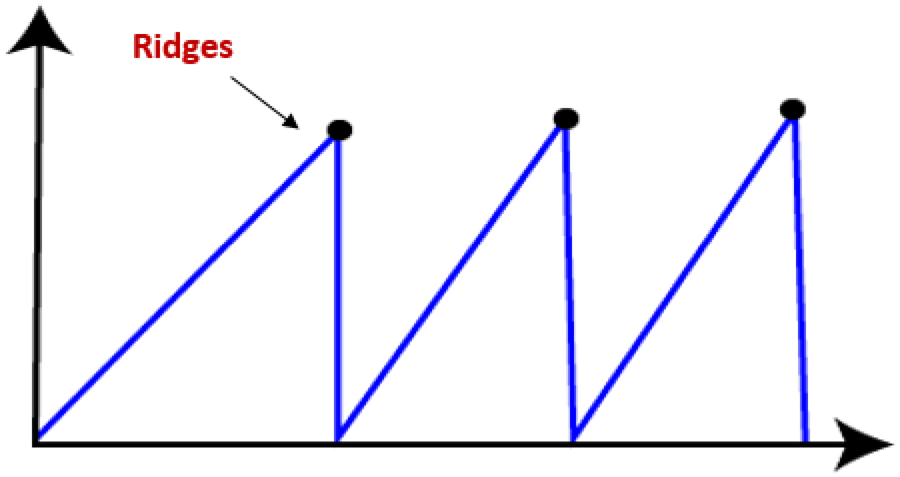
- **Local Maxima:** It is that peak of the mountain which is highest than all its neighboring states but lower than the global maxima. It is not the goal peak because there is another peak higher than it.



- **Plateau:** It is a flat surface area where no uphill exists. It becomes difficult for the climber to decide that in which direction he should move to reach the goal point. Sometimes, the person gets lost in the flat area.



- **Ridges:** It is a challenging problem where the person finds two or more local maxima of the same height commonly. It becomes difficult for the person to navigate the right point and stuck to that point itself.



4.3.2. Simulated Annealing

Simulated annealing is similar to the hill climbing algorithm. It works on the current situation. It picks **a random move** instead of picking **the best move**. If the move leads to the improvement of the current situation, it is always accepted as a step towards the solution state, else it accepts the move having **a probability less than 1**. This search technique was first used in **1980** to solve **VLSI layout** problems. It is also applied for factory scheduling and other large optimization tasks.

4.3.3. Local Beam Search

Local beam search is quite different from random-restart search. It keeps track of **k** states instead of just one. It selects **k** randomly generated states, and expand them at each step. If any state is a goal state, the search stops with success. Else it selects the best **k** successors from the complete list and repeats the same process. In random-restart search where each search process runs independently, but in local beam search, the necessary information is shared between the parallel search processes.

Disadvantages of Local Beam search

- This search can suffer from a lack of diversity among the **k** states.
- It is an expensive version of hill climbing search.

4.4. Constraint Satisfaction Problems

constraint satisfaction problems, or **CSPs** for short, are a flexible approach to searching that have proven useful in many AI-style problems

CSPs can be used to solve problems such as

- **graph-coloring**: given a graph, the a coloring of the graph means assigning each of its vertices a color such that no pair of vertices connected by an edge have the same color
    - in general, this is a very hard problem, e.g. determining if a graph can be colored with 3 colors is NP-hard
    - many problems boil down to graph coloring, or related problems
- **job shop scheduling**: e.g. suppose you need to complete a set of tasks, each of which have a duration, and constraints upon when they start and stop (e.g. task c can't start until both task a and task b are finished)
    - CSPs are a natural way to express such problems
- **cryptarithmetic puzzles**: e.g. suppose you are told that TWO + TWO = FOUR, and each of the letters corresponds to a *different* digit from 0 to 9, and that a number can't start with 0 (so T and F are not 0); what, if any, are the possible values for the letters?
    - while these are not directly useful problems, they are a simple test case for CSP solvers

the basic idea is to have a set of variables that can be assigned values in a constrained way

a CSP consists of three main components:

- XX: a set of **variables** {X1,…,Xn}{X1,…,Xn}
- DD: a set of **domains** {D1,…,Dn}{D1,…,Dn}, one domain per variable
  - i.e. the domain of XiXi is DiDi, which means that XiXi can only be assigned values from DiDi
  - we're only going to consider **finite domains**; you can certainly have CSPs with infinite domains (e.g. real numbers), but we won't consider such problems here
- CC is a set of **constraints** that specify allowable assignments of values to variables
  - for example, a **binary constraint** consists of a pair of different variables, (Xi,Xj)(Xi,Xj), and a set of pairs of values that XiXi and XjXj can take on at the same time
  - we will usually only deal with binary constraints; constraints between three or more variables are possible (e.g. Xi,Xj,XkXi,Xj,Xk are all different), but they don't occur too frequently, and can be decomposed into binary constraints

example 1: suppose we have a CSP as follows:

- three variables X1X1, X2X2, and X3X3
- domains: D1={1,2,3,4}D1={1,2,3,4}, D2={2,3,4}D2={2,3,4}, D3={3,7}D3={3,7}
  - so X1X1 can only be assigned one of the values 1, 2, 3, or 4
- constraints: no pair of variables have the same value, i.e. X1≠X2X1≠X2, X1≠X3X1≠X3, and X2≠X3X2≠X3; we can explicitly describe each of these constraints as a relation between the two variables where the pairs show the allowed values that the variables can be simultaneously assigned, i.e.
  - X1≠X2={(1,2),(1,3),(1,4),(2,3),(2,4),(3,2),(3,4),(4,2),(4,3)}X1≠X2={(1,2),(1,3),(1,4),(2,3),(2,4),(3,2),(3,4),(4,2),(4,3)}
  - X1≠X3={(1,3),(1,7),(2,3),(2,7),(3,7),(4,3),(4,7)}X1≠X3={(1,3),(1,7),(2,3),(2,7),(3,7),(4,3),(4,7)}
  - X2≠X3={(2,3),(2,7),(3,7),(4,3),(4,7)}X2≠X3={(2,3),(2,7),(3,7),(4,3),(4,7)}

these three constraints are each binary constraints because they each constrain 2 variables

example 2: suppose we have the same variables and domains as in the previous example, but now the constraints are X1<X2X1<X2 and X2+X3≤5X2+X3≤5

- domains: D1={1,2,3,4}D1={1,2,3,4}, D2={2,3,4}D2={2,3,4}, D3={3,7}D3={3,7}
- both of the constraints are binary constraints, because they each involve two variables
- X1<X2={(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)}X1<X2={(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)}
- X2+X3≤9={(2,3),(3,3)}X2+X3≤9={(2,3),(3,3)}

by looking at the constraints for this problem, we note the following:

- the only possible value for X3X3 is 3, because 3 is the only value for X3X3 in the constraint X2+X3≤9X2+X3≤9
- we can also see the constraint X2+X3≤9X2+X3≤9 that X2X2 *cannot* be 4; so in the constraint X1<X2X1<X2 we can discard all pairs whose second value is 4, giving: X1<X2={(1,2),(1,3),(2,3)}X1<X2={(1,2),(1,3),(2,3)}
- so the final solutions to this problem are: (X1=1,X2=1,X3=3)(X1=1,X2=1,X3=3), (X1=1,X2=3,X3=3)(X1=1,X2=3,X3=3), (X1=2,X2=3,X3=3)(X1=2,X2=3,X3=3)

example 3: suppose we have the same variables and domains as in the previous example, but now with the two constraints X1+X2=X3X1+X2=X3, and X1X1 is even

- domains: D1={1,2,3,4}D1={1,2,3,4}, D2={2,3,4}D2={2,3,4}, D3={3,7}D3={3,7}
- X1+X2=X3X1+X2=X3 is a **ternary constraint**, because it involves 3 variables
- X1+X2=X3={(1,2,3),(3,4,7),(4,3,7)}X1+X2=X3={(1,2,3),(3,4,7),(4,3,7)}
- X1X1 is even is a **unary** constraint, because it involves one variable
- X1X1 is even ={2,4}={2,4}

- looking at the triples in X1+X2=X3X1+X2=X3, the only one where *X_1* is even is (4,3,7)(4,3,7), and so the only solution to his problem is (X1=4,X2=3,X3=7)(X1=4,X2=3,X3=7)

4.4.1. Some Terminology and Basic Facts

- variables can be assigned, or unassigned
- if a CSP has some variables assigned, and those assignments don’t violate any constraints, we say the CSP is **consistent**, or that it is **satisfied**
    - that partial sets of CSP variable can be satisfied is an important part of many CSP algorithms that work by satisfying one variable at a time
- if all the variables of a CSP are assigned a value, we call that a **complete assignment**
- a **solution** to a CSP complete assignment that is consistent, i.e. a complete assignment that does not violate any constraints
- depending on the constraints, a CSP may have 0, 1, or many solutions
    - a CSP with no solutions is sometimes referred to as **over-constrained**
    - a CSP with many solutions is sometimes referred to as **under-constrained**
- depending upon the problem being solved and its application, we may want just 1 solution, or we might want multiple solutions
    - for over-constrained CSPs, we may even be satisfied with a solution that violates the fewest constraints
- if a domain is empty, then the CSP is no solution
- if a domain has only one value, then that is the only possible value for the corresponding variable
- the size of the **search space** for a CSP is |D1|·|D2|·…·|Dn||D1|·|D2|·…·|Dn|
    - this is the number of n-tuples that the CSP is searching through
    - this is often a quick and easy way to estimate the potential difficulty of a CSP: the bigger the search space, the harder the problem might be
        - this is just a rule of thumb: it’s quite possible that a problem with a large search space is easier than a problem with s small search space,

perhaps because there are many more solutions to be found in the large search space

4.4.2. Constraint Propagation: Arc Consistency

as mentioned above, we are only considering CSPs with finite domains, and with binary constraints between variables

it turns out that many such CSPs can be translated into a simpler CSP that is guaranteed to have the same solutions

the idea we will consider here is **arc consistency**

the CSP variable XiXi is said to be **arc consistent** with respect to variable XjXj if for every value in DiDi there is at least one corresponding value in DjDj that satisfies the binary constraint on XiXi and XjXj

an entire CSP is said to be **arc consistent** if every variable is arc consistent with every other variable

if a CSP is not arc consistent, then it turns out that there are relatively efficient algorithms that will make it arc consistent

- and the resulting arc consistent CSP will have the same solutions as the original CSP, but often a smaller search space
- for some problems, making a CSP arc consistent may be all that is necessary to solve it

the most popular arc consistency algorithm as AC-3

- on a CSP with cc constraints a maximum domain size of dd, AC-3 runs in O(cd3)O(cd3) time

```
functionAC-3(csp)
```

```
returns:falseisaninconsistencyisfound
trueotherwise
cspismodifiedtobearc-consistent
input:CSPwithcomponents(X,D,C)
localvariables:queueofarcs(edges)in
theCSPgraph

queue<--allarcsincsp
whilequeueisnotemptydo
(X_i,X_j)<--queue.pop_first()
ifRevise(csp,X_i,X_j)then
ifsize(D_i)==0thenreturnfalse
foreachX_kinneighbors(X_i)-{X_j}do
add(X_k,X_i)toqueue
endfor
endif
endwhile
returntrue



functionRevise(csp,X_i,X_j)
returnstrueiffdomainD_ihasbeenchanged

revised<--false
foreachxinD_ido
ifnovalueyinD_jallows(x,y)tosatisfy
theconstraintbetweenX_iandX_jthen
deletexfromD_i
revised<--true
```

```
endfor
returnrevised
```

example. Consider the following CSP with variables A, B, C, and D, and:

D_A = {1, 2, 3} D_B = {2, 4} D_C = {1, 3, 4} D_D = {1, 2}

A < B A < D B = D B != C C != D

it’s useful to draw the corresponding constraint graph, and then to apply AC-3 to that by hand to see how the domains changein this particular problem, AC-3 reduces 3 of the 3 domains down to a single value

Backtracking Search

arc consistency can often reduce the size of the domains of a CSP, but it can’t always guarantee to find a solution

so some kind of search is needed

intuitively, the idea backtracking search is to repeat the following steps

- pick an unassigned variable
- assign it a value that doesn’t violate any constraints
    - if there is no such value, then backtrack to the previous step and choose a new variable

a basic backtracking search is complete, i.e. it will find solutions if they exist, but in practice it is often very slow on large problems

- keep in mind that, in general, solving CSPs is NP-hard, and so the best known general-purpose solving algorithms run in worst-case exponential time

so various improvements are made to basic backtracking, such as:

- rules for choosing the next variable to assigned
    - one rule is to always choose the variable with the **minimum remaining values** (**MRV**) in its domain
        - the intuition is that small domains have fewer choices and so will hopefully lead to failure more quickly than big domains
        - in practice, usually performs better than random or static variable choice, but it depends on the problem
- rules for choosing what value to assign to the current variable
    - one rule is to choose the **least-constraining value** from the domain of the variable being assigned, i.e. the value that rules out the fewest choices for neighbor variables
    - in practice, can perform well, but depends on the problem
- interleaving searching and inference: **forward checking**
    - after a variable is assigned in backtracking search, we can rule out all domain values for associated variables that are inconsistent with the assignment
    - forward-checking combined with the MRV variable selection heuristic is often a good combination
- rules for choosing what variable to backtrack to
    - basic backtracking is sometimes called **chronological backtracking** because it always backtracks to the most recently assigned variable
    - but it's possible to do better, e.g. conflict-directed backtracking

an interesting fact about these improvements on backtracking search is that they are domain-independent — they can be applies to and CSP

- you can, of course, still use domain-specific heuristics if you have any

it's also worth pointing out that fast and memory-efficient implementations are a non-trivial engineering challenge, and require some thought to implement efficiently

- plus, experiments with different combinations of heuristic rules are often needed to find the most efficient variation

Local Search: Min-conflicts

backtracking search solves a CSP by assigning one variable at a time

another approach to solving a CSP is to assign *all* the variables, and then modify this assignment to make it better

this is a kind of local search on CSPs, and for some problems it can be extremely effective

example: 4-queens problem

- place 4 queens randomly on a 4-by-4 board, one queen per column
- pick a queen, and re-position it in its column so that it is attacking the fewest number of other queens; this is know as the **min-conflicts** heuristic
- repeat the previous step until there are no possible moves that reduce conflicts; if the puzzle is solved, then you’re done; if it’s not solved, then re-start the entire process with a new initial random placement of queens

local search using min-conflicts can be applied to any CSP as follows:

```
functionMin-Conflicts(csp,max_steps)
returns:solution,orfailure
Inputs:csp,aconstraintsatisfactionproblem
max_steps,numberofstepsallowedbeforegivingup

current<--aninitialcompleteassignmentforCSP
fori=1tomax_stepsdo
ifcurrentisasolution,thenreturncurrent
var<--randomlychosenconflictedvariablefromcsp.VARIABLES
value<--thevaluevforvarthatminimizesCONFLICTS(var,v,current,csp)
```

```
setvar=valueincurrent
end
```

the CONFLICTS functions returns the count of the number of variables in the rest of the assignment that violate a constraint when var=v