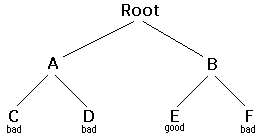
**Backtracking**

Backtracking is a form of recursion.

The usual scenario is that you are faced with a number of options, and you must choose one of these. After you make your choice you will get a new set of options; just what set of options you get depends on what choice you made. This procedure is repeated over and over until you reach a final state. If you made a good sequence of choices, your final state is a *goal state;* if you didn't, it isn't.

This needs an example.



1. Starting at Root, your options are A and B. You choose A.
2. At A, your options are C and D. You choose C.
3. C is bad. Go back to A.
4. At A, you have already tried C, and it failed. Try D.
5. D is bad. Go back to A.
6. At A, you have no options left to try. Go back to Root.
7. At Root, you have already tried A. Try B.
8. At B, your options are E and F. Try E.
9. E is good. Congratulations!

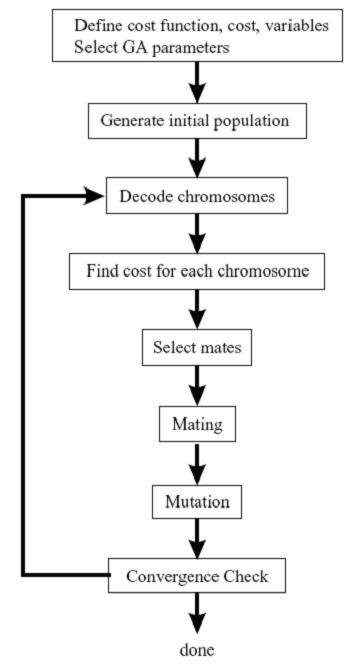
Backtracking can be applied only for problems which admit the concept of a "partial candidate solution" and a relatively quick test of whether it can possibly be completed to a valid solution. It is useless, for example, for locating a given value in an unordered table. When it is applicable, however, backtracking is often much faster than [brute force enumeration](https://en.wikipedia.org/wiki/Brute_force_search) of all complete candidates, since it can eliminate a large number of candidates with a single test.

8 queens problem

The **eight queens puzzle** is the problem of placing eight [chess](https://en.wikipedia.org/wiki/Chess)[queens](https://en.wikipedia.org/wiki/Queen_%28chess%29) on an 8×8 chessboard so that no two queens threaten each other. Thus, a solution requires that no two queens share the same row, column, or diagonal. The eight queens puzzle is an example of the more general ***n* queens problem** of placing *n* non-attacking queens on an *n*×*n* chessboard, for which solutions exist for all natural numbers *n* with the exception of *n*=2 and *n*=3.

The classic use of backtracking is the [eight queens puzzle](https://en.wikipedia.org/wiki/Eight_queens_puzzle), that asks for all arrangements of eight [chess](https://en.wikipedia.org/wiki/Chess)[queens](https://en.wikipedia.org/wiki/Queen_%28chess%29) on a standard [chessboard](https://en.wikipedia.org/wiki/Chessboard) so that no queen attacks any other. In the common backtracking approach, the partial candidates are arrangements of *k* queens in the first *k* rows of the board, all in different rows and columns. Any partial solution that contains two mutually attacking queens can be abandoned. [Edsger Dijkstra](https://en.wikipedia.org/wiki/Edsger_Dijkstra) used this problem to illustrate the power of what he called [structured programming](https://en.wikipedia.org/wiki/Structured_programming). He published a highly detailed description of a [depth-first](https://en.wikipedia.org/wiki/Depth-first_search) backtracking algorithm.

For 8-queen, we have 92 solutions. If we exclude symmetry, there are 12 solutions.



Algorithm:

- Start with one queen at the first column first row

- Continue with second queen from the second column first row

- Go up until find a permissible situation

- Continue with next queen

We place the first queen on A1:

Again with red we show the prohibited positions. It turned out that we cannot place the

third queen on the third column (we have to have a queen for each column!). In other

words we imposed a set of constraints in a way that we no longer can satisfy them in

order to find a solution. Hence we need to revise the constraints or rearrange the board up

to the state which we were stuck. Now we may ask a question what we have to change.

Since the problem happened after placing Q2 we are trying first with this queen.

OK we know that there were to possible places for Q2. B3 gives problem for the third

queen, so there is only one position left – B4:

As you can see from the new set of constraints ( the red positions) now we have

admissible position for Q3, but it will make impossible to place Q4 since the only place is

D3. Hence placing Q2 on the only one left position B4 didn’t help. Therefore the one step

backtrack was not enough. We need to go for second backtrack. Why? The reason is that

there is no position for Q2, which will satisfy any position for Q4 or Q3. Hence we need

to deal with the position of Q1.

We have started from Q1 so we will continue upward and placing the queen at A2

To find this solution we had to perform two backtracks. So what now? In order to find all

solutions we use as you can guess – backtrack!

Start again in reverse order we try to place Q4 somewhere up, which is not possible. We

backtrack to Q3 and try to find admissible place different from C1. Again we need to

backtrack. Q2 has no other choice and finally we reach Q1. We place Q1 on A3:

Continuing further we will reach the solution on the right. Is this distinct solution? No it

is rotated first solution. In fact for 4x4 board there is only one unique solution. Placing

Q1 on A4 has the same effect as placing it on A1. Hence we explored all solutions.

How implement backtrack in code. Remember that we used backtrack when we cannot

find admissible position for a queen in a column. Otherwise we go further with the next

column until we place a queen on the last column. Therefore your code must have

fragment:

int PlaceQueen(int board[8], int row)

If (Can place queen on ith column)

PlaceQueen(newboard, 0)

Else

PlaceQueen(oldboard,oldplace+1)

End

If you can place queen on ith column try to place a queen on the next one, or backtrack

and try to place a queen on position above the solution found for i-1 column.

Knapsack problem

The **knapsack problem** or **rucksack problem** is a problem in [combinatorial optimization](https://en.wikipedia.org/wiki/Combinatorial_optimization): Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. It derives its name from the problem faced by someone who is constrained by a fixed-size[knapsack](https://en.wikipedia.org/wiki/Knapsack) and must fill it with the most valuable items.

There are many variations of the knapsack problem that have arisen from the vast number of applications of the basic problem. The main variations occur by changing the number of some problem parameter such as the number of items, number of objectives, or even the number of knapsacks.

### Multi-objective knapsack problem[[edit](https://en.wikipedia.org/w/index.php?title=Knapsack_problem&action=edit&section=14" \o "Edit section: Multi-objective knapsack problem)]

This variation changes the goal of the individual filling the knapsack. Instead of one objective, such as maximizing the monetary profit, the objective could have several dimensions. For example, there could be environmental or social concerns as well as economic goals. Problems frequently addressed include portfolio and transportation logistics optimizations [[20]](https://en.wikipedia.org/wiki/Knapsack_problem#cite_note-20)[[21]](https://en.wikipedia.org/wiki/Knapsack_problem#cite_note-21)

As a concrete example, suppose you ran a cruise ship. You have to decide how many famous comedians to hire. This boat can handle no more than one ton of passengers and the entertainers must weigh less than 1000 lbs. Each comedian has a weight, brings in business based on their popularity and asks for a specific salary. In this example you have multiple objectives. You want, of course, to maximize the popularity of your entertainers while minimizing their salaries. Also, you want to have as many entertainers as possible.

### Multi-dimensional knapsack problem[[edit](https://en.wikipedia.org/w/index.php?title=Knapsack_problem&action=edit&section=15" \o "Edit section: Multi-dimensional knapsack problem)]

In this variation, the weight of knapsack item {\displaystyle i} is given by a D-dimensional vector {\displaystyle {\overline {w\_{i}}}=(w\_{i1},\ldots ,w\_{iD})} and the knapsack has a D-dimensional capacity vector {\displaystyle (W\_{1},\ldots ,W\_{D})}. The target is to maximize the sum of the values of the items in the knapsack so that the sum of weights in each dimension {\displaystyle d} does not exceed {\displaystyle W\_{d}}.

Multi-dimensional knapsack is computationally harder than knapsack; even for {\displaystyle D=2}, the problem does not have [EPTAS](https://en.wikipedia.org/wiki/EPTAS) unless P{\displaystyle =}NP.[[22]](https://en.wikipedia.org/wiki/Knapsack_problem#cite_note-22) However, the algorithm in [[23]](https://en.wikipedia.org/wiki/Knapsack_problem#cite_note-CohenGrebla-23) is shown to solve sparse instances efficiently. An instance of multi-dimensional knapsack is sparse if there is a set {\displaystyle J=\{1,2,\ldots ,m\}} for {\displaystyle m<D} such that for every knapsack item {\displaystyle i}, {\displaystyle \exists z>m} such that {\displaystyle \forall j\in J\cup \{z\},\ w\_{ij}\geq 0} and {\displaystyle \forall y\notin J\cup \{z\},w\_{iy}=0}. Such instances occur, for example, when scheduling packets in a wireless network with relay nodes.[[23]](https://en.wikipedia.org/wiki/Knapsack_problem#cite_note-CohenGrebla-23) The algorithm from [[23]](https://en.wikipedia.org/wiki/Knapsack_problem#cite_note-CohenGrebla-23) also solves sparse instances of the multiple choice variant, multiple-choice multi-dimensional knapsack.

The IHS (Increasing Height Shelf) algorithm is optimal for 2D knapsack (packing squares into a two-dimensional unit size square): when there are at most five square in an optimal packing.[[24]](https://en.wikipedia.org/wiki/Knapsack_problem#cite_note-24)

### Multiple knapsack problem[[edit](https://en.wikipedia.org/w/index.php?title=Knapsack_problem&action=edit&section=16" \o "Edit section: Multiple knapsack problem)]

This variation is similar to the [Bin Packing Problem](https://en.wikipedia.org/wiki/Bin_packing_problem). It differs from the Bin Packing Problem in that a subset of items can be selected, whereas, in the Bin Packing Problem, all items have to be packed to certain bins. The concept is that there are multiple knapsacks. This may seem like a trivial change, but it is not equivalent to adding to the capacity of the initial knapsack. This variation is used in many loading and scheduling problems in Operations Research and has a [PTAS](https://en.wikipedia.org/wiki/Polynomial-time_approximation_scheme)[[25]](https://en.wikipedia.org/wiki/Knapsack_problem#cite_note-25)

### Quadratic knapsack problem[[edit](https://en.wikipedia.org/w/index.php?title=Knapsack_problem&action=edit&section=17" \o "Edit section: Quadratic knapsack problem)]

As described by Wu et al.:

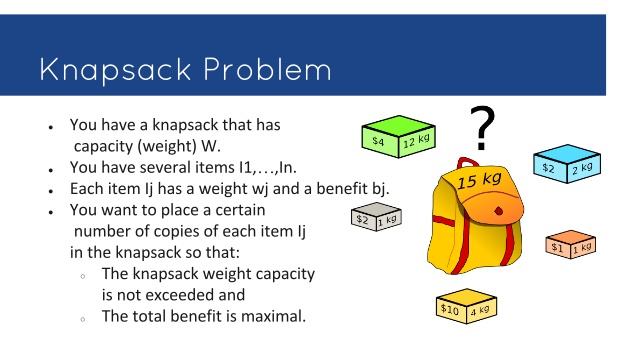
The quadratic knapsack problem (QKP) maximizes a quadratic objective function subject to a binary and linear capacity constraint.[[26]](https://en.wikipedia.org/wiki/Knapsack_problem#cite_note-QKP-26)

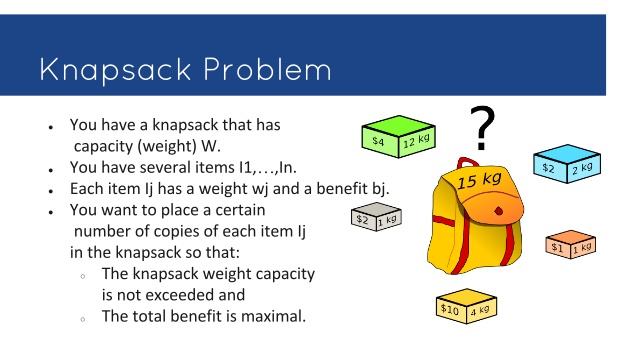
The quadratic knapsack problem was discussed under that title by Gallo, Hammer, and Simeone in 1980.[[27]](https://en.wikipedia.org/wiki/Knapsack_problem#cite_note-27) However, Gallo and Simeone[[28]](https://en.wikipedia.org/wiki/Knapsack_problem" \l "cite_note-28) attribute the first treatment of the problem to Witzgall[[29]](https://en.wikipedia.org/wiki/Knapsack_problem" \l "cite_note-29) in 1975.

### Subset-sum problem[[edit](https://en.wikipedia.org/w/index.php?title=Knapsack_problem&action=edit&section=18" \o "Edit section: Subset-sum problem)]

The [subset sum problem](https://en.wikipedia.org/wiki/Subset_sum_problem) is a special case of the decision and **0-1** problems where each kind of item, the weight equals the value: {\displaystyle w\_{i}=v\_{i}}. In the field of [cryptography](https://en.wikipedia.org/wiki/Cryptography), the term*knapsack problem* is often used to refer specifically to the subset sum problem and is commonly known as one of [Karp's 21 NP-complete problems](https://en.wikipedia.org/wiki/Karp%27s_21_NP-complete_problems).[[30]](https://en.wikipedia.org/wiki/Knapsack_problem#cite_note-30)

The generalization of subset sum problem is called multiple subset-sum problem, in which multiple bins exist with the same capacity. It has been shown that the generalization does not have an FPTAS.[[31]](https://en.wikipedia.org/wiki/Knapsack_problem#cite_note-31)





|  |
| --- |
| # include<stdio.h>    void knapsack(int n, float weight[], float profit[], float capacity) {     float x[20], tp = 0;     inti, j, u;     u = capacity;       for (i = 0; i< n; i++)        x[i] = 0.0;       for (i = 0; i< n; i++) {        if (weight[i] > u)           break;        else {           x[i] = 1.0;           tp = tp + profit[i];           u = u - weight[i];        }     }       if (i< n)        x[i] = u / weight[i];       tp = tp + (x[i] \* profit[i]);       printf("\nThe result vector is:- ");     for (i = 0; i< n; i++)        printf("%f\t", x[i]);       printf("\nMaximum profit is:- %f", tp);    }    int main() {     float weight[20], profit[20], capacity;     intnum, i, j;     float ratio[20], temp;       printf("\nEnter the no. of objects:- ");     scanf("%d", &num);       printf("\nEnter the wts and profits of each object:- ");     for (i = 0; i<num; i++) {        scanf("%f %f", &weight[i], &profit[i]);     }       printf("\nEnter the capacityacity of knapsack:- ");     scanf("%f", &capacity);       for (i = 0; i<num; i++) {        ratio[i] = profit[i] / weight[i];     }       for (i = 0; i<num; i++) {        for (j = i + 1; j <num; j++) {           if (ratio[i] < ratio[j]) {              temp = ratio[j];              ratio[j] = ratio[i];              ratio[i] = temp;                temp = weight[j];              weight[j] = weight[i];              weight[i] = temp;                temp = profit[j];              profit[j] = profit[i];              profit[i] = temp;           }        }     }       knapsack(num, weight, profit, capacity);     return(0);  } |

**Output :**



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | Enter the no. of objects:- 7    Enter the wts and profits of each object:-  2 10  3 5  5 15  7 7  1 6  4 18  1 3    Enter the capacity of knapsack:- 15    The result vector is:- 1.000000        1.000000        1.000000        1.000000      1.000000        0.666667        0.000000    Maximum profit is:- 55.333332 |

**Concept of Knapsack:**

            The knapsack is nothing but a sack where in which we need to place the given items according to the capacity of the knapsack. In case of backtracking we consider the profits but in dynamic programming we consider weights.

**Algorithm:**

            Bound (CP, CW, K)

            {

                        b=CP, c=CW;

                        fori=k+1 to ndp

                        {

                                    c=c + w [i];

                                    if (c<m) then

                                    b= b + p[i];

                                    else

                                    return b+(1-(c-m)/w[i])\*p[i]                             ;

                                    return b;

                        }

            }

The **knapsack problem** or**rucksack problem** is a problem in combinatorial optimization: Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. It derives its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most valuable items.

In solving of **knapsack problem using backtracking method** we mostly consider the profit but in case of dynamic programming we consider weights.

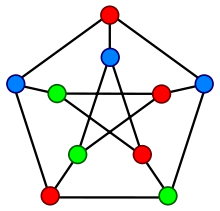
The idea of **backtracking** is to construct solutions one component at a time and evaluate such partially constructed solutions. This partially constructed solution can be developed further without violating the problem constraints. It is convenient to implement this kind of processing by constructing a tree of choices being made called the “State Space Tree”. Its root represents an initial state before the search for the solution begins. The nodes of the first level in the tree represent the choices for the first component of a solution and the nodes of a second level represent the choices for the second component and so on.

A node in the state space tree is promising if it corresponds to the partials constructed solution that may lead to the complete solution otherwise the nodes are called non-promising. Leaves of the tree represent either the non- promising dead end or complete solution found by the algorithm.

**Graph coloring**

In [graph theory](https://en.wikipedia.org/wiki/Graph_theory), **graph coloring** is a special case of [graph labeling](https://en.wikipedia.org/wiki/Graph_labeling); it is an assignment of labels traditionally called "colors" to elements of a [graph](https://en.wikipedia.org/wiki/Graph_(discrete_mathematics)) subject to certain constraints. In its simplest form, it is a way of coloring the vertices of a graph such that no two adjacent [vertices](https://en.wikipedia.org/wiki/Vertex_(graph_theory)" \o "Vertex (graph theory))share the same color; this is called a **vertex coloring**. Similarly, an [edge coloring](https://en.wikipedia.org/wiki/Edge_coloring) assigns a color to each edge so that no two adjacent edges share the same color, and a **face coloring** of a planar graph assigns a color to each face or region so that no two faces that share a boundary have the same color.

Vertex coloring is the starting point of the subject, and other coloring problems can be transformed into a vertex version. For example, an[edge coloring](https://en.wikipedia.org/wiki/Edge_coloring) of a graph is just a vertex coloring of its [line graph](https://en.wikipedia.org/wiki/Line_graph), and a face coloring of a [plane graph](https://en.wikipedia.org/wiki/Plane_graph) is just a vertex coloring of its [dual](https://en.wikipedia.org/wiki/Dual_graph). However, non-vertex coloring problems are often stated and studied *as is*. That is partly for perspective, and partly because some problems are best studied in non-vertex form, as for instance is edge coloring.

****

If all colors are assigned,

print vertex assigned colors

Else

a. Trying all possible colors, assign a color to the vertex

b. If color assignment is possible, recursivelty assign colors to next vertices

c. If color assignment is not possible, de-assign color, return False

# Game Trees

In this chapter we look at using trees for game playing, in particular the problems of searching game trees. We will classify all games into the folowing three groups

1. Single-player pathfinding problems.
   * Rubik’s Cube
   * Sliding puzzle.
   * Travelling Salesman Problem.
2. Two-player games.
   * Chess
   * Checkers
   * Othello
3. Constraint satisfaction problems.
   * Eight Queens
   * Sudoku

Each game consists of a problem space, an initial state, and a single (or a set of) goal states. A problem space is a mathematical abstraction in a form of a tree:

* the root represents current state
* nodes represent states of the game
* edges represent moves
* leaves represent final states (win, loss or draw)

For example, in the 8-Puzzle game

* nodes: the different permutations of the tiles.
* edges: moving the blank tile up, down, right or left.

For some problems, the choice of a problem space is not so obvious. One general rule is that a smaller representation, in the sense of fewer states to search, is often better then a larger one. A problem space is characterized by two major factors.

**The branching factor** - the average number of children of the nodes in the space.

* The eight puzzle has a branching factor of 2.13
* Rubik’s cube has a branching factor of 13.34
* Chess has a branching factor of about 35

**The solution depth** - the length of the shortest path from the initial node to a goal node.

The size of a solution space:

* Tic-Tac-Toe is 9! = 362,880
* 8-puzzle - 9!/2
* Checkers - 10^40
* Chess - 10^120 (40 moves, 35 branch factor - 35^(2\*40))

How to search for a move?

## Brute-Force Searches

* Breadth-First Search (BFS)

BFS expands nodes in order of their depth from the root.

Implemented by first-in first-out (FIFO) queue.

BFS will find a shortest path to a goal.

Time/Space Complexity - branching factor b and the solution depth d. Generate all the nodes up to level d.

total number of nodes = 1 + b + b^2 + ... + b^d = O(b^d)

BFS will exhaust the memory in minutes.

* Depth-First Search (DFS)

Implemented by LIFO stack

Space Complexity is linear in the maximum search depth.

DFS generate the same set of nodes as BFS - Time Complexity is O(b^d)

The first solution DFS found may not be the optimal one.

On infinite tree DFS may not terminate.

* Depth-First Iterative-Deepening

First performs a DFS to depth one. Than starts over executing DFS to depth two and so on

### Minimax

We consider games with two players in which one person's gains are the result of another person's losses (so called zero-sum games). The minimax algorithm is a specialized search algorithm which returns the optimal sequence of moves for a player in an zero-sum game. In the game tree that results from the algorithm, each level represents a move by either of two players, say A- and B-player. Below is a game tree for the tic-tac-toe game

The minimax algorithm explores the entire game tree using a depth-first search. At each node in the tree where A-player has to move, A-player would like to play the move that maximizes the payoff. Thus, A-player will assign the maximum score amongst the children to the node where Max makes a move. Similarly, B-player will minimize the payoff to A-player. The maximum and minimum scores are taken at alternating levels of the tree, since A and B alternate turns.

The minimax algorithm computes the minimax decision for the leaves of the game tree and than backs up through the tree to give the final value to the current state.

## Heuristic Search

So far we have looked at search algorithms that can in principle be used to systematically search the whole search space. Sometimes however it is not feasible to search the whole search space - it's just too big. In this case we need to use heuristic search. The basic idea of heuristic search is that, rather than trying all possible search paths, we focus on paths that seem to be getting us closer to the goal state. Of course, we generally can't be sure that we are really near the goal state, but we might be able to have a good guess. Heuristics are used to help us make that guess.

To use heuristic search we need an *evaluation function* that rankes nodes in the search tree according to some criteria (for example, how close we are to the target). This function provides a quick way of guessing.

**Best First Search.**   
The search is similar to BFS, but instead of taking the first node it always chooses a node with the best score, according to an evaluation function. If we create a good evalution function, best first search may drastically cut down the amount of search time.

Here are the two most important properties of a heuristic function:

* it must provide an accurate estimator of the cost to reach a goal.
* it must be cheap to compute.
* it always must be a lower bound on actual solution cost.

**A\* algorithm**   
Best first search doesn't take into account the cost of the path from a start state to the current state. So, we may find a solution but it may be not a very good solution. There is a variant of best first search known as A\* which attempts to find a solution which minimizes the total cost of the solution path. This algorithm combines advantages of breadth first search with advantages of best first search.

In the A\* algorithm the score assigned to a node is a combination of the cost of the path so far A(S) and the estimated cost E(S) to solution.

H(S) = A(S) + E(S)

The A\* algorithm looks the same as the best first algorithm, but we use this slightly more complex evaluation function.

BRANCH AND BOND METHOD

* Branch and bound is a systematic method for solving optimization problems
* B&B is a rather general optimization technique that applies where the greedy method and dynamic programming fail.
* However, it is much slower. Indeed, it often leads to exponential time complexities in the worst case.
* On the other hand, if applied carefully, it can lead to algorithms that run reasonably fast on average.
* The general idea of B&B is a BFS-like search for the optimal solution, but not all nodes get expanded (i.e., their children generated). Rather, a carefully selected criterion determines which node to expand and when, and another criterion tells the algorithm when an optimal solution has been found.

Illustration on the Job Assignment Problem

* Input: n jobs, n employees, and an n x n matrix A where Aij be the cost if person i performs job j.
* Problem: find a one-to-one matching of the n employees to the n jobs so that the total cost is minimized.
* formally, find a permutation f such that C(f), where

C(f)=A1f(1) + A2f(2) + ... + Anf(n)

is minimized.

* A brute-force method would generate the whole solution tree, where every path from the root to any leaf is a solution, then evaluate the C of each solution, and finally choose the path with the minimum cost.
* Illustration on this specific instance of the job-assignment problem:

|  |  |  |
| --- | --- | --- |
| 2 | 4 | 5 |
| 2 | 7 | 10 |
| 5 | 3 | 7 |

* In informal terms, the problem is to choose a single number from each row such that (1) no two numbers are chosen from the same columns, and (2) the sum of the chosen numbers is minimized.
* The brute force method
* The first idea of B&B is to develop "a predictor" of the likelihood (in a loose sense) of a node in the solution tree that it will lead to an optimal solution. This predictor is quantitative.
* With such a predictor, the B&B works as follows:
  1. Which node to expand next: B&B chooses the live node with the best predictor value
  2. B&B simply expands that node (i.e., generate all its children)
  3. the predictor value of each newly generated node is computed, the just expanded node is now designated as a dead node, and the newly generated nodes are designated as live nodes.
  4. Termination criterion: When the best node chosen for expansion turn out to be a final leaf (i.e., at level n), that when the algorithm terminates, and that node corresponds to the optimal solution. The proof of optimality will be presented later on.
* What could that predictor be?
* In the case of minimization problem, one candidate predictor of any node is the cost so far . That is, each node corresponds to (partial) solution (from the root to that node). The cost-so-far predictor is the cost of the partial solution.
* Apply this preliminary algorithm on the above specific instance of the job assignment problem

|  |  |  |
| --- | --- | --- |
| 2 | 4 | 5 |
| 2 | 7 | 10 |
| 5 | 3 | 7 |

* A better predictor for the job assignment problem is:

(cost so far) + (sum of the minimums of the remaining rows)

* Apply B&B with the new predictor to the same instance of the job assignment problem

|  |  |  |
| --- | --- | --- |
| 2 | 4 | 5 |
| 2 | 7 | 10 |
| 5 | 3 | 7 |

A yet another predictor is cost so far + sumni=k+1pi

where pi is the minimum value in row i of the cost matrix A, such that p is not in the column of any of the terms chosen in the partial solution so far.

* Apply B&B with the last predictor to the same instance of the job assignment problem

|  |  |  |
| --- | --- | --- |
| 2 | 4 | 5 |
| 2 | 7 | 10 |
| 5 | 3 | 7 |

The General Branch and Bound Algorithm :-

* Each solution is assumed to be expressible as an array X[1:n] (as was seen in Backtracking).
* A predictor, called an approximate cost function CC, is assumed to have been defined.
* Definitions:
  + A live node is a node that has not been expanded
  + A dead node is a node that has been expanded
  + The expanded node (or E-node for short) is the live node with the best CC value.
* The general B&B algorithm follows:

Procedure B&B()

begin

E: nodepointer;

E := new(node); -- this is the root node which

-- is the dummy start node

H: heap; -- A heap for all the live nodes

-- H is a min-heap for minimization problems,

-- and a max-heap for maximization problems.

while (true) do

if (E is a final leaf) then

-- E is an optimal solution

print out the path from E to the root;

return;

endif

Expand(E);

if (H is empty) then

report that there is no solution;

return;

endif

E := delete-top(H);

endwhile

end

Procedure Expand(E)

begin

- Generate all the children of E;

- Compute the approximate cost value CC of each child;

- Insert each child into the heap H;

end

NP Hard

**NP-hardness** ([*n*on-deterministic *p*olynomial-time](https://en.wikipedia.org/wiki/NP_(complexity)) hard), in [computational complexity theory](https://en.wikipedia.org/wiki/Computational_complexity_theory), is a class of problems that are, informally, "at least as hard as the hardest problems in [NP](https://en.wikipedia.org/wiki/NP_(complexity))". More precisely, a problem *H* is NP-hard when every problem *L* in [NP](https://en.wikipedia.org/wiki/NP_(complexity)) can be [reduced](https://en.wikipedia.org/wiki/Reduction_(complexity)) in [polynomial time](https://en.wikipedia.org/wiki/Polynomial_time) to *H*, that is given a solution for *L* we can verify it is a solution for *H*in polynomial time.[[1]](https://en.wikipedia.org/wiki/NP-hardness#cite_note-Leeuwen-1):80[[2]](https://en.wikipedia.org/wiki/NP-hardness#cite_note-2) As a consequence, finding a polynomial algorithm to solve any NP-hard problem would give polynomial algorithms for all the problems in NP, which is unlikely as many of them are considered hard.[[3]](https://en.wikipedia.org/wiki/NP-hardness#cite_note-3)

A common misconception is that the *NP* in "NP-hard" stands for "non-polynomial" when in fact stands for "Non-deterministic polynomial acceptable problems".[[4]](https://en.wikipedia.org/wiki/NP-hardness#cite_note-4) Although it is suspected that there are no polynomial-time algorithms for NP-hard problems, this has never been proven.[[5]](https://en.wikipedia.org/wiki/NP-hardness#cite_note-5) Moreover, the class [P](https://en.wikipedia.org/wiki/P_(complexity)) in which all problems can be solved in polynomial time, is contained in the [NP](https://en.wikipedia.org/wiki/NP_(complexity)) class.[[6]](https://en.wikipedia.org/wiki/NP-hardness#cite_note-6)

## Definition

A [decision problem](https://en.wikipedia.org/wiki/Decision_problem) *H* is NP-hard when for every problem *L* in NP, there is a [polynomial-time reduction](https://en.wikipedia.org/wiki/Polynomial-time_reduction) from *L* to *H*[[1]](https://en.wikipedia.org/wiki/NP-hardness#cite_note-Leeuwen-1):80 An equivalent definition is to require that every problem *L*in NP can be solved in [polynomial time](https://en.wikipedia.org/wiki/Polynomial_time) by an [oracle machine](https://en.wikipedia.org/wiki/Oracle_machine) with an oracle for *H*.[[7]](https://en.wikipedia.org/wiki/NP-hardness#cite_note-7) Informally, we can think of an algorithm that can call such an oracle machine as a subroutine for solving *H*, and solves *L* in polynomial time, if the subroutine call takes only one step to compute.

Another definition is to require that there is a polynomial-time reduction from an [NP-complete](https://en.wikipedia.org/wiki/NP-complete) problem *G* to *H*.[[1]](https://en.wikipedia.org/wiki/NP-hardness#cite_note-Leeuwen-1):91 As any problem *L* in NP reduces in polynomial time to *G*, *L*reduces in turn to *H* in polynomial time so this new definition implies the previous one. It does not restrict the class NP-hard to decision problems, for instance it also includes[search problems](https://en.wikipedia.org/wiki/Search_problem), or [optimization problems](https://en.wikipedia.org/wiki/Optimization_problem).

## Consequences

* If P ≠ NP, then NP-hard problems cannot be solved in polynomial time;
* If an optimization problem *H* has an NP-complete decision version *L*, then *H* is NP-hard.

An example of an NP-hard problem is the decision [subset sum problem](https://en.wikipedia.org/wiki/Subset_sum_problem), which is this: given a set of integers, does any non-empty subset of them add up to zero? That is a[decision problem](https://en.wikipedia.org/wiki/Decision_problem), and happens to be NP-complete. Another example of an NP-hard problem is the optimization problem of finding the least-cost cyclic route through all nodes of a weighted graph. This is commonly known as the [traveling salesman problem](https://en.wikipedia.org/wiki/Traveling_salesman_problem).

**Np-complete**

In [computational complexity theory](https://en.wikipedia.org/wiki/Computational_complexity_theory), a [decision problem](https://en.wikipedia.org/wiki/Decision_problem) is **NP-complete** when it is both in [NP](https://en.wikipedia.org/wiki/NP_(complexity)) and [NP-hard](https://en.wikipedia.org/wiki/NP-hard). The set of NP-complete problems is often denoted by **NP-C** or **NPC**. The abbreviation **NP** refers to "[nondeterministic](https://en.wikipedia.org/wiki/Nondeterministic_algorithm) [polynomial time](https://en.wikipedia.org/wiki/Polynomial_time)".

Although any given solution to an NP-complete problem can be verified quickly (in polynomial time), there is no known efficient way to locate a solution in the first place; indeed, the most notable characteristic of NP-complete problems is that no fast solution to them is known. That is, the time required to solve the problem using any currently known [algorithm](https://en.wikipedia.org/wiki/Algorithm) increases very quickly as the size of the problem grows. As a consequence, determining whether or not it is possible to solve these problems quickly, called the [P versus NP problem](https://en.wikipedia.org/wiki/P_versus_NP_problem), is one of the principal [unsolved problems in computer science](https://en.wikipedia.org/wiki/List_of_open_problems_in_computer_science) today.

While a method for computing the solutions to NP-complete problems using a reasonable amount of time remains undiscovered, [computer scientists](https://en.wikipedia.org/wiki/Computer_scientist) and [programmers](https://en.wikipedia.org/wiki/Computer_programmer) still frequently encounter NP-complete problems. NP-complete problems are often addressed by using [heuristic](https://en.wikipedia.org/wiki/Heuristic_(computer_science)) methods and [approximation algorithms](https://en.wikipedia.org/wiki/Approximation_algorithm).

## Formal definition of NP-completeness

## A decision problem {\displaystyle \scriptstyle C}c is NP-complete if:

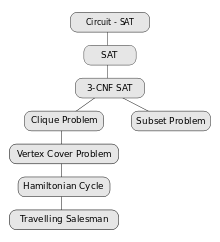
1. {\displaystyle \scriptstyle C}c is in NP, and
2. Every problem in NP is [reducible](https://en.wikipedia.org/wiki/Many-one_reduction) to {\displaystyle \scriptstyle C}c in polynomial time.[[1]](https://en.wikipedia.org/wiki/NP-completeness#cite_note-1)

{\displaystyle \scriptstyle C} can be shown to be in NP by demonstrating that a candidate solution to {\displaystyle \scriptstyle C}c can be verified in polynomial time.

Note that a problem satisfying condition 2 is said to be [NP-hard](https://en.wikipedia.org/wiki/NP-hard), whether or not it satisfies condition 1.[[2]](https://en.wikipedia.org/wiki/NP-completeness#cite_note-2)

A consequence of this definition is that if we had a polynomial time algorithm (on a [UTM](https://en.wikipedia.org/wiki/Universal_Turing_machine), or any other [Turing-equivalent](https://en.wikipedia.org/wiki/Turing_completeness) [abstract machine](https://en.wikipedia.org/wiki/Abstract_machine)) for c {\displaystyle \scriptstyle C}cc c, we could solve all problems in NP in polynomial time.

## NP-complete problems

[](https://en.wikipedia.org/wiki/File:Relative_NPC_chart.svg)

Some NP-complete problems, indicating the [reductions](https://en.wikipedia.org/wiki/Reduction_(complexity)) typically used to prove their NP-completeness

An interesting example is the [graph isomorphism problem](https://en.wikipedia.org/wiki/Graph_isomorphism_problem), the [graph theory](https://en.wikipedia.org/wiki/Graph_theory) problem of determining whether a [graph isomorphism](https://en.wikipedia.org/wiki/Graph_isomorphism) exists between two graphs. Two graphs are [isomorphic](https://en.wikipedia.org/wiki/Isomorphic) if one can be [transformed](https://en.wikipedia.org/wiki/Isomorphism) into the other simply by renaming [vertices](https://en.wikipedia.org/wiki/Vertex_(graph_theory)). Consider these two problems:

* Graph Isomorphism: Is graph G1 isomorphic to graph G2?
* Subgraph Isomorphism: Is graph G1 isomorphic to a subgraph of graph G2?

The Subgraph Isomorphism problem is NP-complete. The graph isomorphism problem is suspected to be neither in P nor NP-complete, though it is in NP. This is an example of a problem that is thought to be **hard**, but is not thought to be NP-complete.

The easiest way to prove that some new problem is NP-complete is first to prove that it is in NP, and then to reduce some known NP-complete problem to it. Therefore, it is useful to know a variety of NP-complete problems. The list below contains some well-known problems that are NP-complete when expressed as decision problems.

* [Boolean satisfiability problem (SAT)](https://en.wikipedia.org/wiki/Boolean_satisfiability_problem)
* [Knapsack problem](https://en.wikipedia.org/wiki/Knapsack_problem)
* [Hamiltonian path problem](https://en.wikipedia.org/wiki/Hamiltonian_path_problem)
* [Travelling salesman problem](https://en.wikipedia.org/wiki/Travelling_salesman_problem) (decision version)
* [Subgraph isomorphism problem](https://en.wikipedia.org/wiki/Subgraph_isomorphism_problem)
* [Subset sum problem](https://en.wikipedia.org/wiki/Subset_sum_problem)
* [Clique problem](https://en.wikipedia.org/wiki/Clique_problem)
* [Vertex cover problem](https://en.wikipedia.org/wiki/Vertex_cover_problem)
* [Independent set problem](https://en.wikipedia.org/wiki/Independent_set_problem)
* [Dominating set problem](https://en.wikipedia.org/wiki/Dominating_set_problem)
* [Graph coloring problem](https://en.wikipedia.org/wiki/Graph_coloring_problem)