Algorithms and Data Structures
Module 4. NP-hard problems

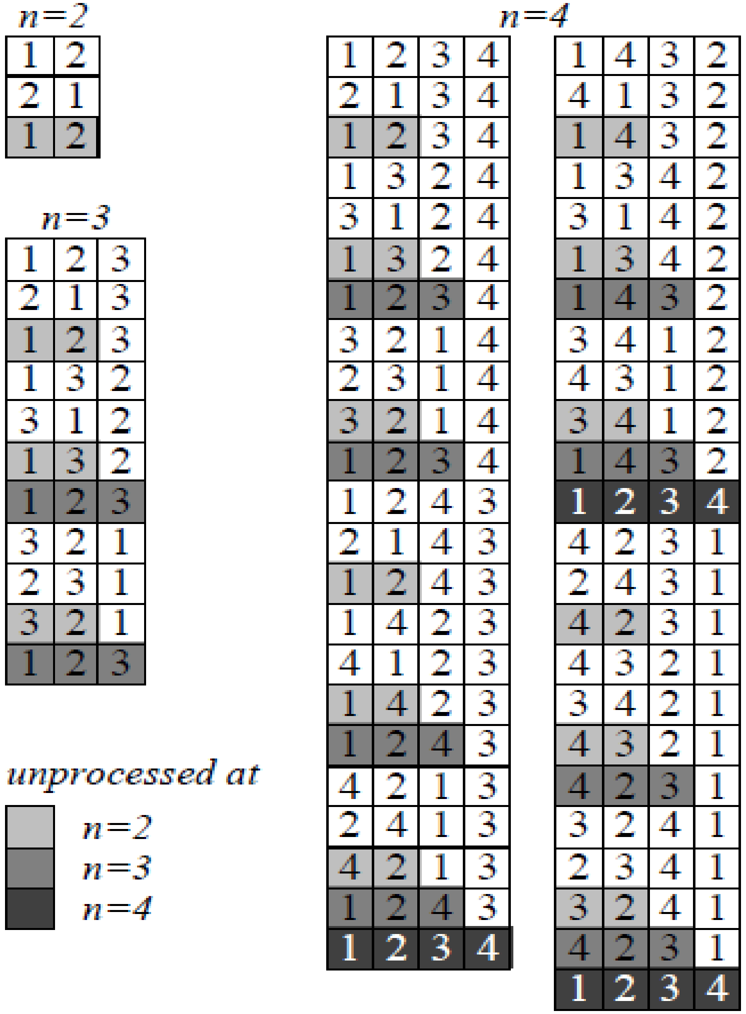
Lecture 19

Branch-and-Bound approach.

TSP: brute force

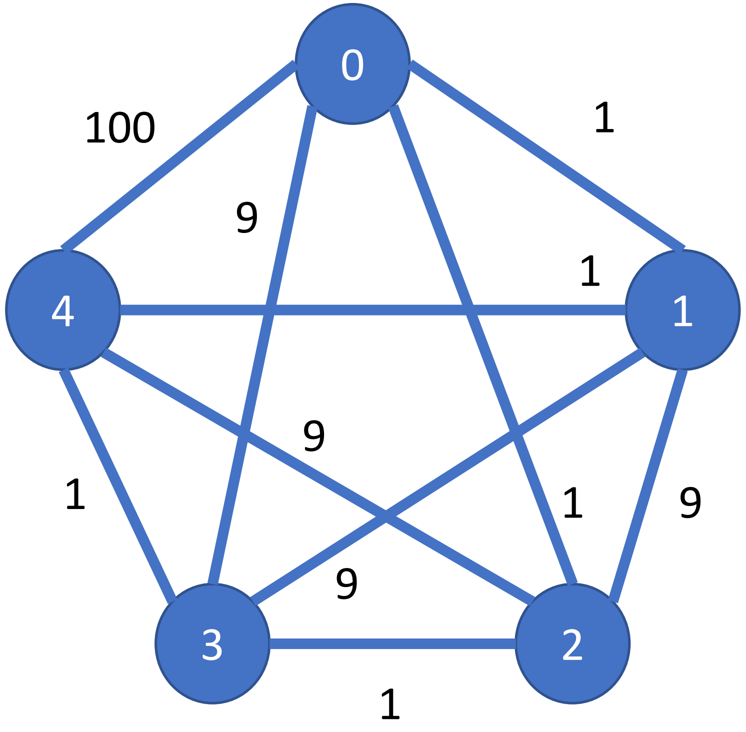
Call: `ProcessPermutations(A,k)`

```
Function ProcessPermutations(A,k)
if k = 1 then Process(A)
else
        ProcessPermutations(A, k-1);
        for i = k-1 downto 1 do
        {
           swap A[k] and A[i];
           ProcessPermutations(A, k-1);
           swap A[k] and A[i];
        }
```



TSP: brute force

Let’s consider this graph:



The optimal Hamiltonian cycle: 0,1,4,3,2,0

$n=2$

1	2
2	1
1	2

$n=3$

1	2	3
2	1	3
1	2	3
1	3	2
3	1	2
1	3	2
1	2	3
3	2	1
2	3	1
3	2	1
1	2	3

unprocessed at

n=2
n=3
n=4

$n=4$

1	2	3	4
2	1	3	4
1	2	3	4
1	3	2	4
3	1	2	4
1	3	2	4
1	2	3	4
3	2	1	4
2	3	1	4
3	2	1	4
1	2	3	4
1	2	4	3
2	1	4	3
1	2	4	3
1	4	2	3
4	1	2	3
1	4	2	3
1	2	4	3
4	2	1	3
2	4	1	3
4	2	1	3
1	2	4	3
1	2	3	4

1	4	3	2
4	1	3	2
1	4	3	2
1	3	4	2
3	1	4	2
1	3	4	2
1	4	3	2
3	4	1	2
4	3	1	2
3	4	1	2
1	4	3	2
1	2	3	4
4	2	3	1
2	4	3	1
4	2	3	1
4	3	2	1
3	4	2	1
4	3	2	1
4	2	3	1
3	2	4	1
2	3	4	1
3	2	4	1
4	2	3	1
1	2	3	4

Brute force

A brute-force algorithm generates and checks variants in a standard order which does not take into consideration features of the specific problem instance. This leads to a lot of unnecessary job.

Idea 1: if we design an algorithm that uses instance specific information, it can avoid this unnecessary job and thus speed-up calculations.

Idea 2: for many problems, including TSP, we can reject unpromising solutions based on analysis of a part of this solution (*partial solution*).

Branch-and-Bound approach

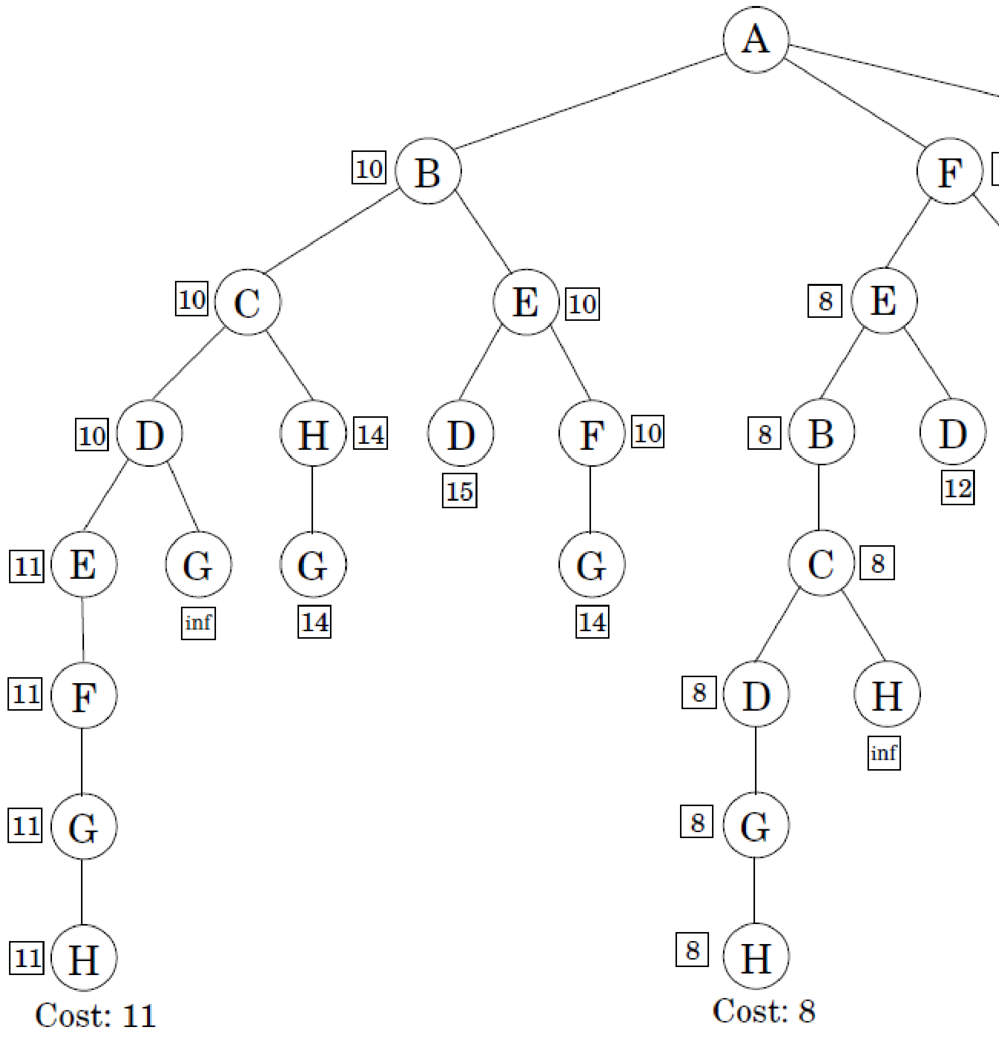
How can we detect unpromising partial solutions?

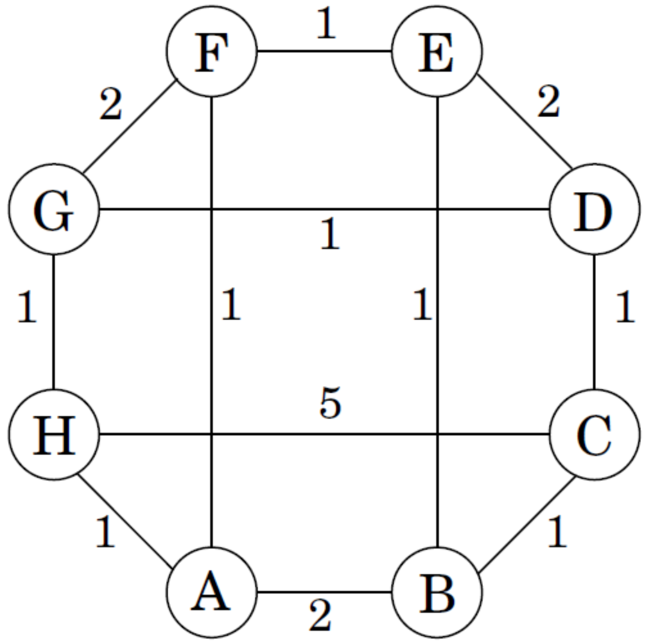
Let us suppose that we have a solution (current record-holder).

If the current partial solution cannot be augmented to a solution which is better than the current record, we can (should) reject this partial solution.

Thus, we save time by not processing any of the solutions that augment the current partial solution (the current ***branch***).

Branch-and-Bound approach





Notice that just 28 partial solutions are considered, instead of the 7! = 5,040 that would arise in a brute-force search.

Branch-and-Bound approach

So, we need a method to check, whether the current partial solution cannot be augmented to a solution which is better than the current record. This can be done with a ***bound function***.

The type of the bound depends on the type of the problem.

- For *minimization* problem we need a *lower* bound.
- For *maximization* problem we need an *upper* bound.

For TSP we need a method to calculate a lower bound.

Branch-and-Bound approach

For TSP we need a method to calculate a lower bound.

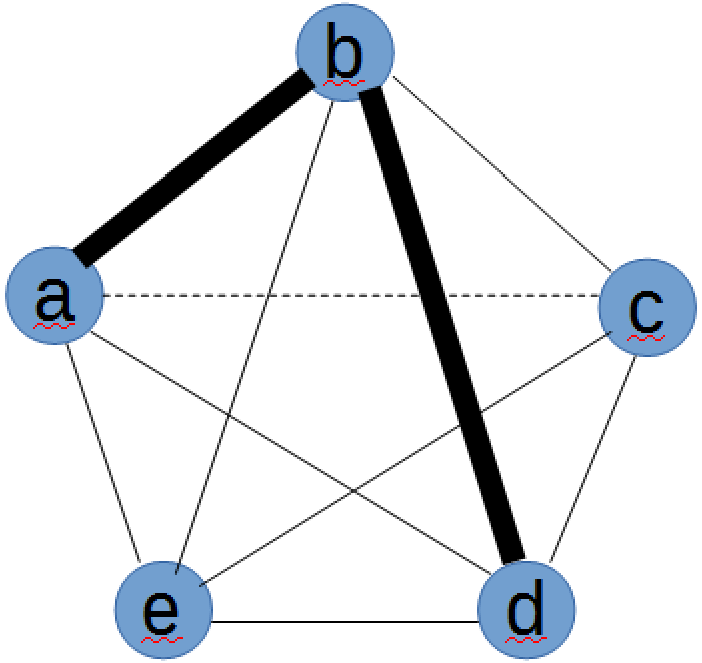
There is a plenty of such methods.

1) LowerBound = the weight of the current partial solution (path). This method is the simplest one. But it is the weakest one as well, since it does not reject many branches.
2) LowerBound = the weight of the current partial solution $+(n-k)\cdot w_{min}$. In this formula k is the number of edges in the current partial solution and w_{min} is the minimum weight of the edges still not in the solution.

Branch-and-Bound approach

2) LowerBound = the weight of the current partial solution $+(n-k) \cdot w_{min}$. In this formula k is the number of edges in the current partial solution and w_{min} is the minimum weight of the edges still not in the solution.

$$LowerBound = w(a,b) + w(b,d) + 3 \cdot w(a,c)$$



Branch-and-Bound approach

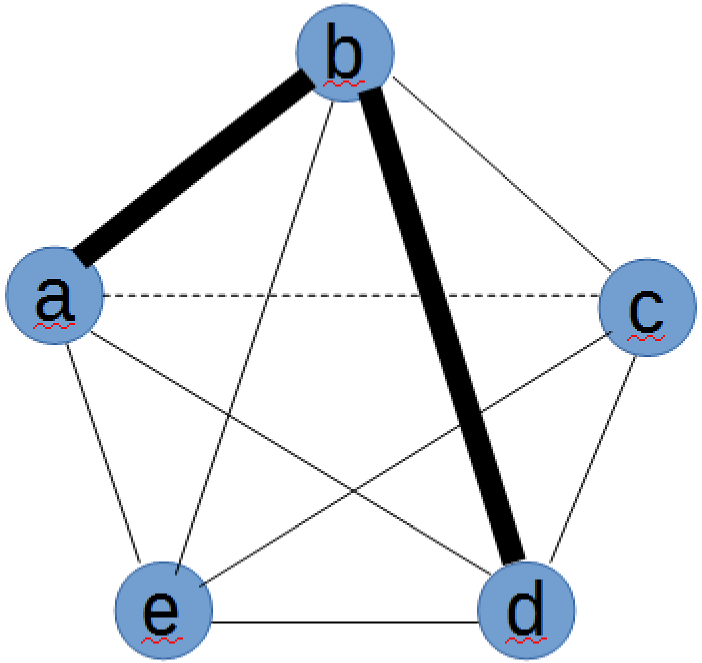
3) LowerBound = the sum of the weights of two *proper* edges incident to each vertex, divided by 2.

What edges are proper?

a) If a vertex is incident to two edges included in the partial solution, both of them are proper.

b) If a vertex is incident to only one edge from the partial solution, this edge is proper. The another proper edge is the lightest of the other incident edges.

c) If a vertex is not incident to edges from the partial solution, the two lightest incident edges are proper edges.

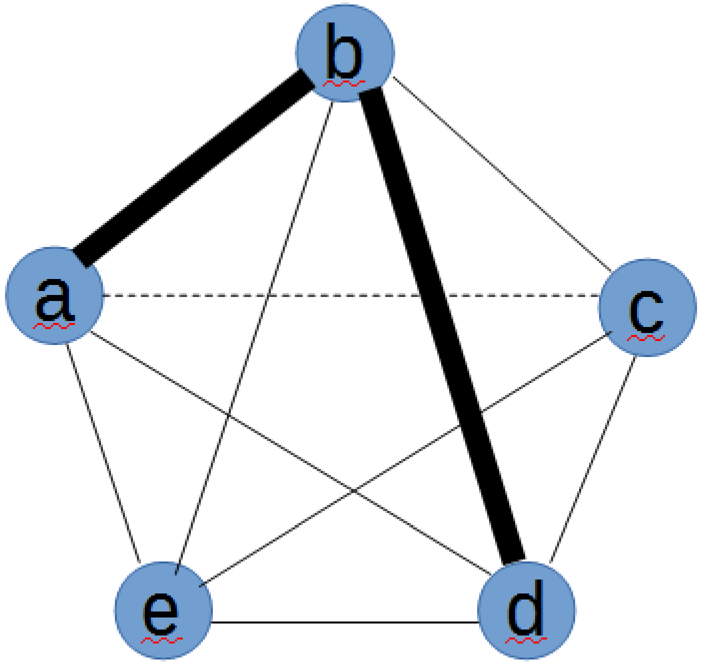


Branch-and-Bound approach

4) Let x be the start of the current partial solution, y be the end of the current partial solution. And let U denote the vertices that are not included in the current partial solution.

LowerBound = the weight of the current partial solution + the sum of the following:

- The lightest weight of an edge from x to a vertex in U.
- The lightest weight of an edge from y to a vertex in U.
- The minimum spanning tree on the subgraph induced by U. Why is it so?

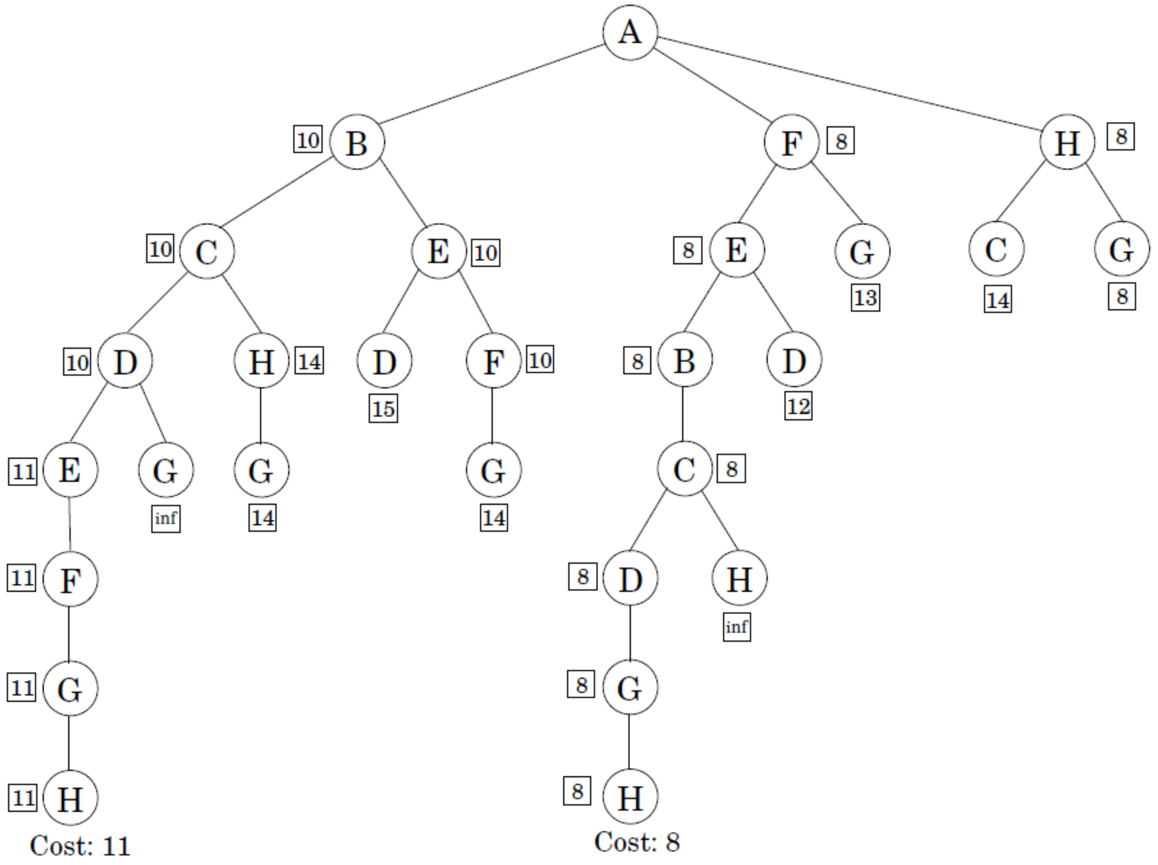


Branch-and-Bound approach

We see that there may be several different ways to calculate a bound for a particular problem. These ways differ in both accuracy and time complexity. Thus, we need an optimal trade-off.

A possible option is to use several bound functions on different phases of the process:

- More accurate though more time consuming bounds are used at the higher levels of the backtracking tree.
- Less accurate but more fast bounds are used at the lower levels.



Branch-and-Bound approach

So, a branch-and-bound algorithm recursively decomposes the set of all possible solutions into subsets (***branches***), calculates bounds for branches and rejects unpromising branches.

```
Start with some problem P_0
Let S = {P_0}, the set of active subproblems
bestsofar = ∞
Repeat while S is nonempty:
  choose a subproblem (partial solution) P ∈ S and remove it from S
  expand it into smaller subproblems P_1, P_2, ..., P_k
  For each P_i:
    If P_i is a complete solution:  update bestsofar
    else if lowerbound(P_i) < bestsofar:  add P_i to S
return bestsofar
```

Branch-and-Bound approach

Such decomposition is performed ‘*virtually*’, without explicit generating and storing the subsets of solutions. A branch is represented as a set of restrictions which the possible solutions must satisfy.

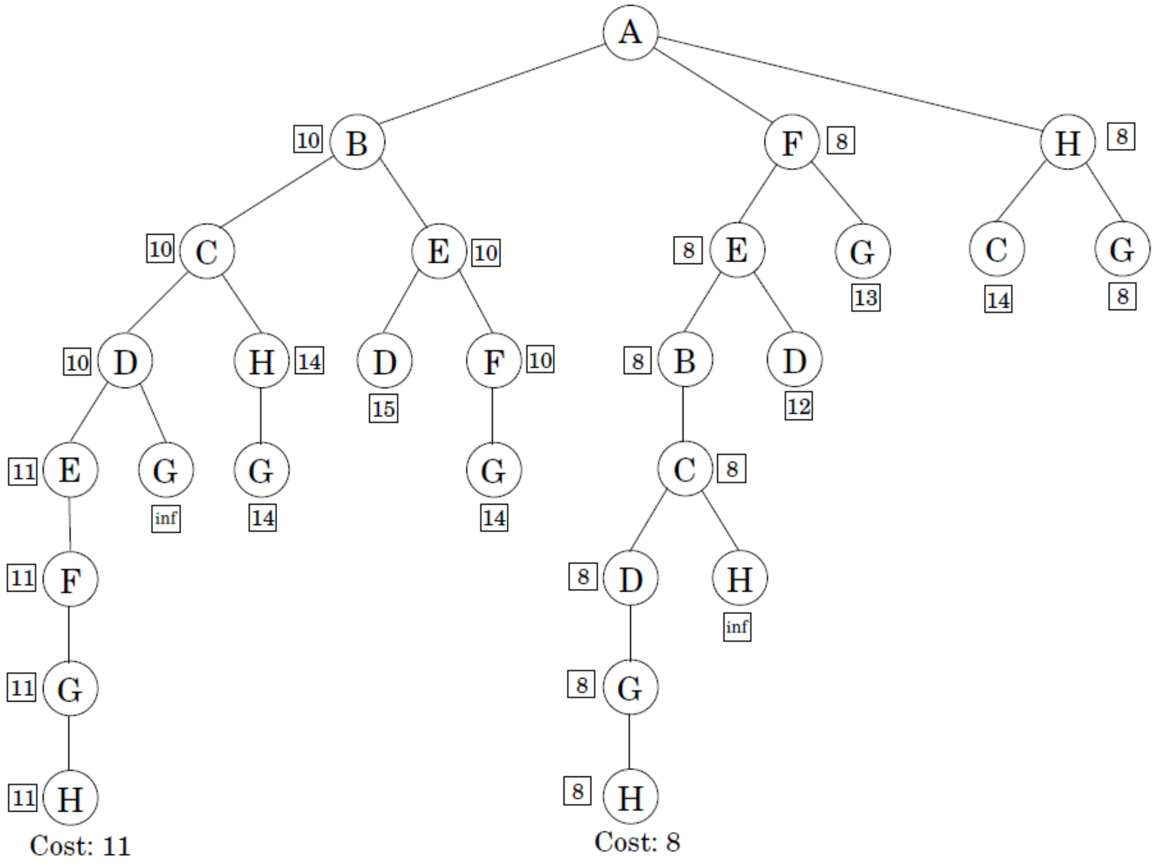
How can we generate branches? The concrete algorithm is problem specific and interdependent with the representation of solutions.

For TSP, a possible solution can be represented as either vertex permutation or a sequence of edges.

Branch-and-Bound approach

For vertex permutation representation, a branch is specified as a prefix (the starting part) of the path.

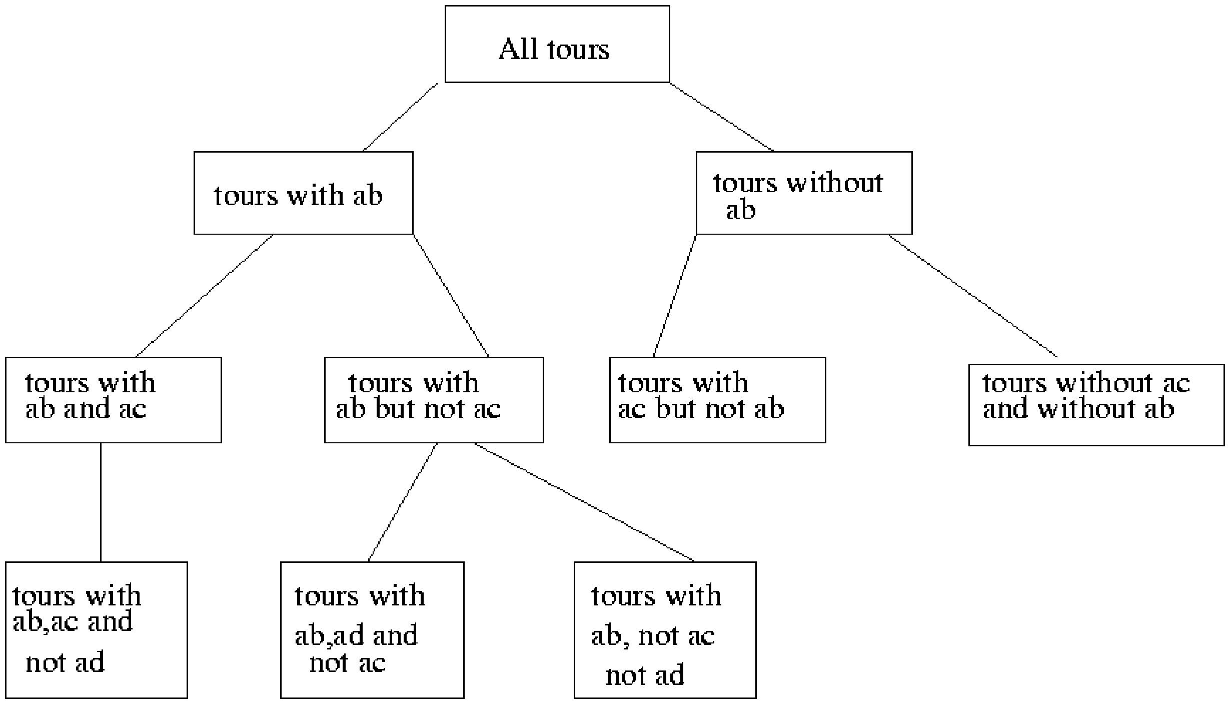
http://algorithmics.lsi.upc.edu/docs/Dasgupta-Papadimitriou-Vazirani.pdf



Branch-and-Bound approach

For edge sequence representation, a branch is specified as a set of conditions like

‘**tours with** <edge>’ and ‘**tours without** <edge>’ .



http://lcm.csa.iisc.ernet.in/dsa/node187.html

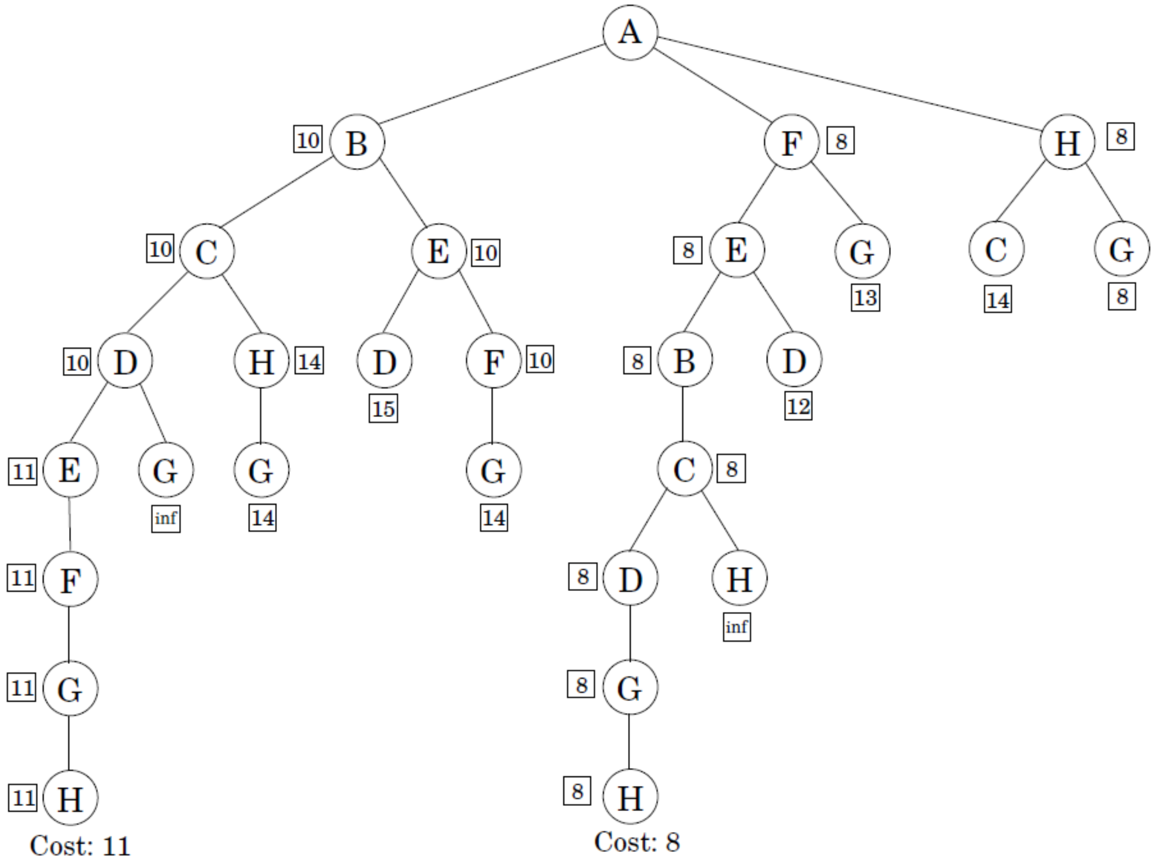
Branch-and-Bound approach

What data structures do we need to implement a branch-and-bound algorithm? For the ‘standard’ form of a B&B algorithm we need a *stack* to store partial solutions.

```
Start with some problem P_0
Let S = {P_0}, the set of active subproblems
bestsofar = ∞
Repeat while S is nonempty:
  choose a subproblem (partial solution) P ∈ S and remove it from S
  expand it into smaller subproblems P_1, P_2, ..., P_k
  For each P_i:
    If P_i is a complete solution:  update bestsofar
    else if lowerbound(P_i) < bestsofar:  add P_i to S
return bestsofar
```

Branch-and-Bound approach

If we employ a stack, we *virtually* traverse the solutions' tree in a depth-first manner.

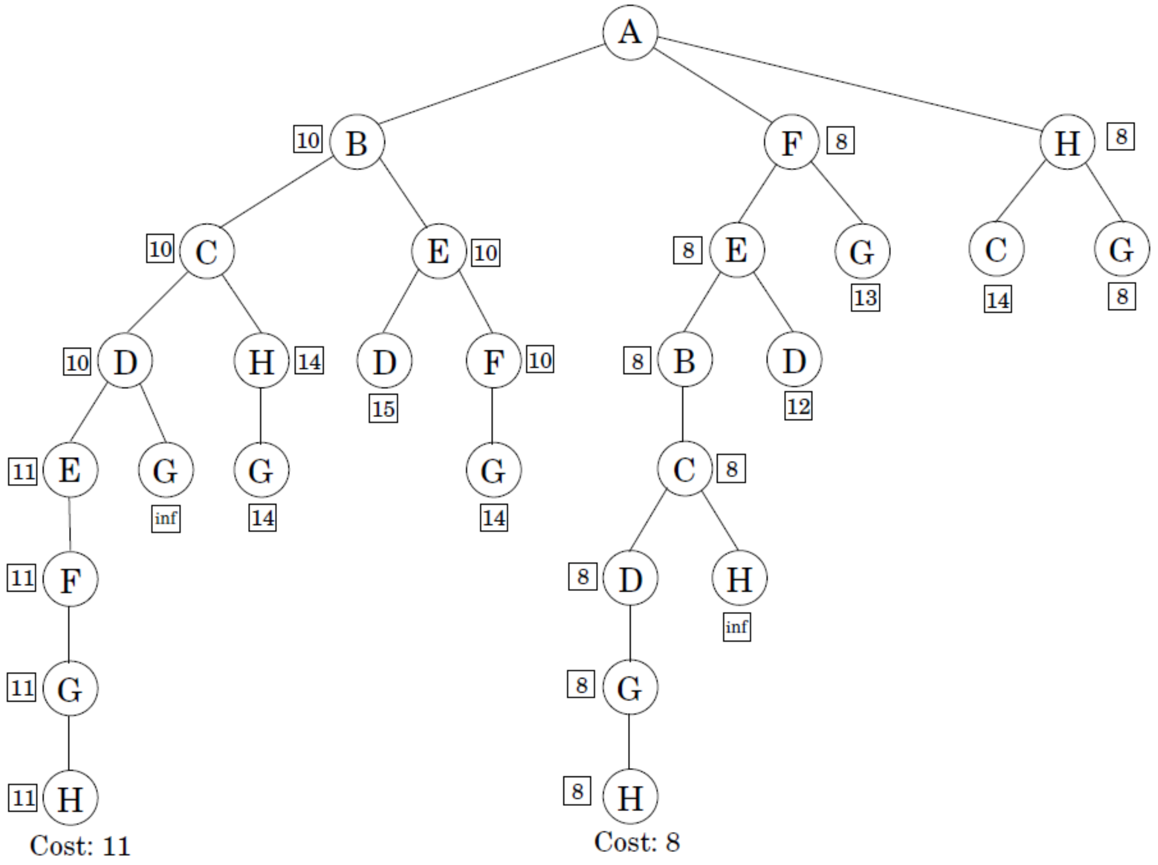


Branch-and-Bound approach

If we employ a priority queue instead of a stack, we process branches (partial solutions) in a *best-first* manner.

This option can speed-up calculations, since we have more chances to find better solution faster. And the better current record-holder we have, the more branches we reject.

But this version requires more memory for storing branches.



Branch-and-Bound approach

Summary:

1) A branch-and-bound algorithm recursively decomposes the set of all possible solutions into subsets (***branches***), calculates ***bounds*** for branches and rejects unpromising branches.
2) Branching is performed *virtually*, without explicit generating and storing the subsets of solutions. A branch is represented as a set of restrictions which the possible solutions must satisfy.
3) Both branching and bounding are problem specific. For a problem there are usually several ways to perform branching and to calculate bounds.
4) A good decision is to use several bound functions on different phases of the process: more accurate though more time consuming bounds are used at the higher levels of the backtracking tree; less accurate but more fast bounds are used at the lower levels.

Branch-and-Bound approach

Summary:

5) We can use a stack to implement a standard B&B algorithm or a priority queue to implement a time-optimized version.
6) B&B needs exponential time in the worst case ☹ For many practical problems, however, it works fast on the average, due to good rejecting rules.

A branch-and-bound algorithm can be stopped before exploring all promising branches. In this case it will need little time but will yield a *heuristic (approximate)*, rather than exact solution…