VE281

Data Structures and Algorithms

Backtracking and Branch-and-Bound

Outline

- Hard problems and solution space
- Backtracking
- Branch-and-Bound

Hard Problems

- Many hard problems require you to find either a **subset** or **permutation** that satisfies some constraints and (possibly also) optimizes some objective function
- **Subset problems**
  - Solution requires you to find a **subset** of n elements that must satisfy some constraints and possibly optimize some objective function
- **Permutation problems**
  - Solution requires you to find a **permutation** of n elements that must satisfy some constraints and possibly optimize some objective function

Example: Subset Sum Problem

- Given a set of positive integers $S = \{s_1, s_2, \dots, s_n\}$ and another integer c, does any subset of S has a sum exactly equal to c?
- Example: $S = [9,4,6,3,5,1,8]$ and $c = 18$
- Answer: Yes, subset = $\{9,4,5\}$

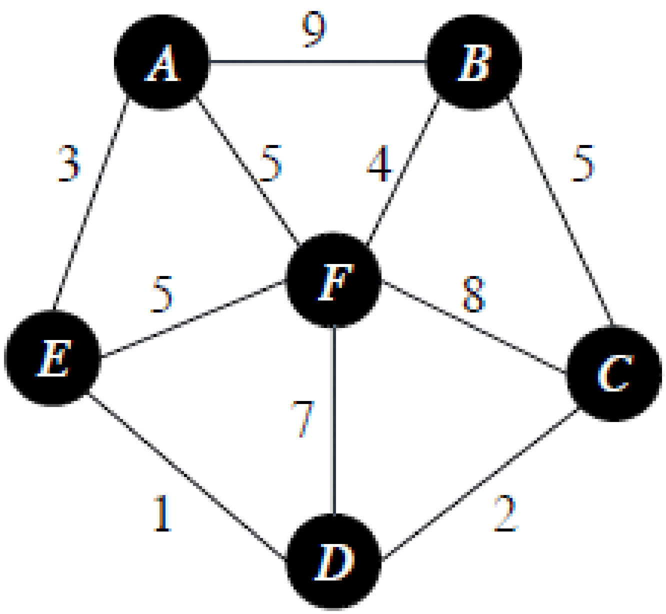
Example: Boolean Satisfiability Problem

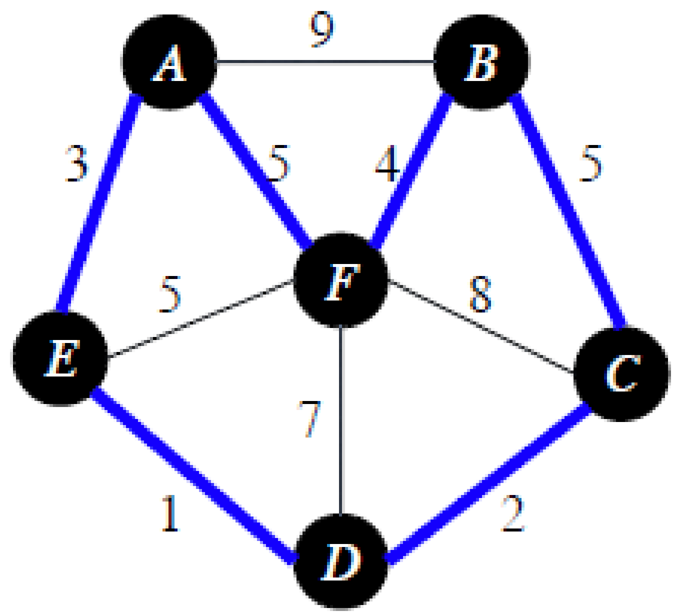
- Called **SAT** for short
- Given a Boolean function represented in the **Conjunctive Normal Form (CNF)**
  - E.g., $\Phi = (a + c)(b + c)(\bar{a} + \bar{b} + \bar{c})$
- Find an assignment of the variables so that function = 1
  - Could have many satisfying assignment; return **any one** is fine
  - However, if there are no satisfying assignments at all, prove it and return this info.
    - We call this **unSAT**
- It is a subset problem. What is the subset?

The set of variables you set to 1

Travelling Salesmen Problem (TSP)

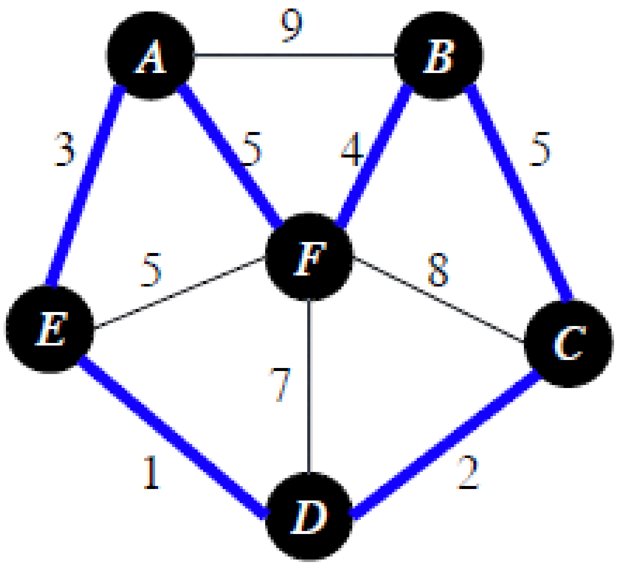
- Find tour of minimum length starting and ending in same node and visiting every node exactly once
- It is a permutation problem





Aside: Hamiltonian Cycle

- A **Hamiltonian Cycle** in a connected, weighted, undirected graph G is a simple cycle that begins at a vertex v, passes through every vertex exactly once, and terminates at v
- TSP problem seeks a Hamiltonian Cycle with minimal weight in a given connected, weighted, undirected graph G



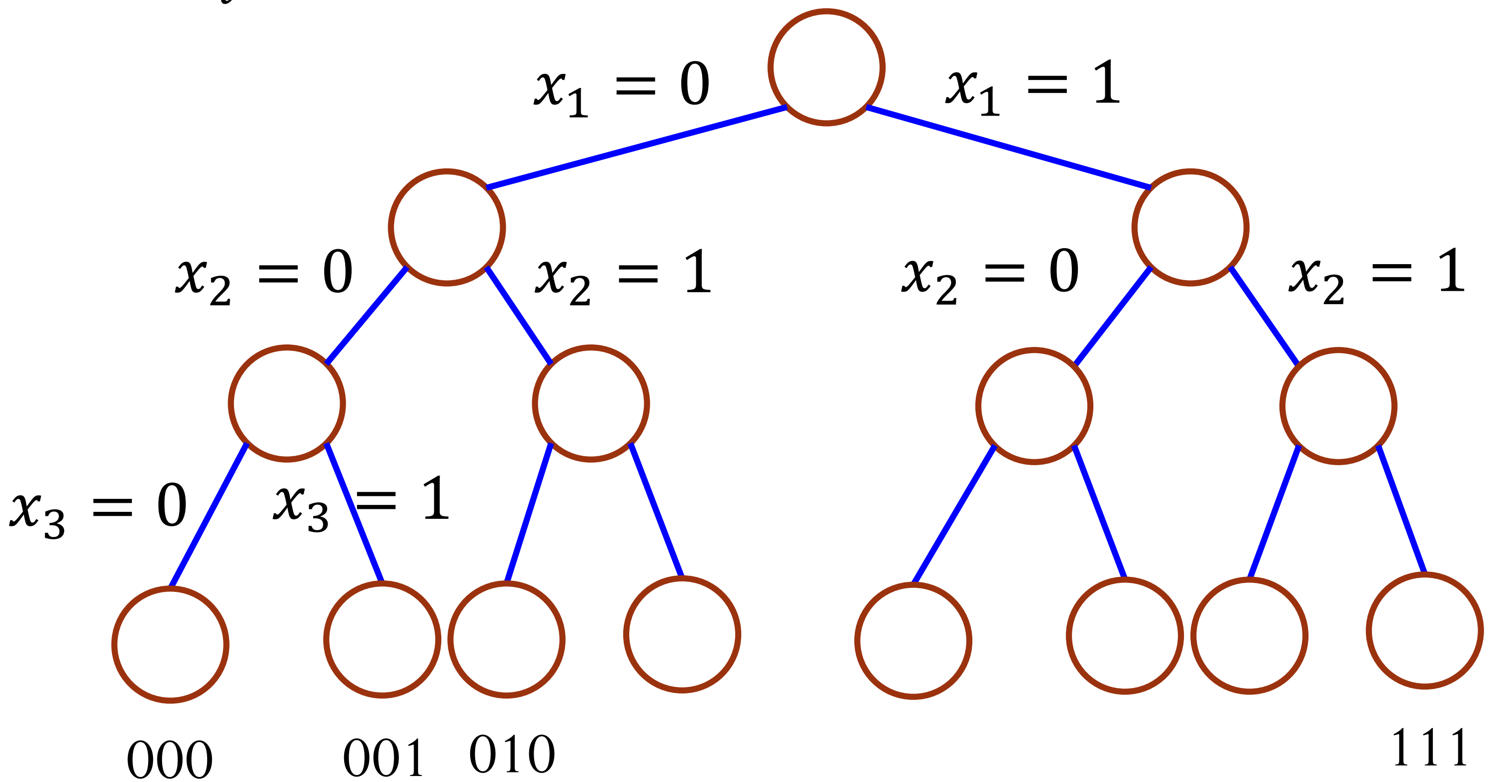
Solution Space

- For subset problem
  - Solution space is composed of all subsets
  - How many? 2^n
  - E.g., subset sum problem
  - We encode a subset by a combination $(x_1, x_2, \dots, x_n) \in \{0,1\}^n$: $x_i = 1/0$ means the i-th item is/is not in the subset

- For permutation problem
  - Solution space is composed of all permutations
  - How many? $n!$
  - E.g., TSP

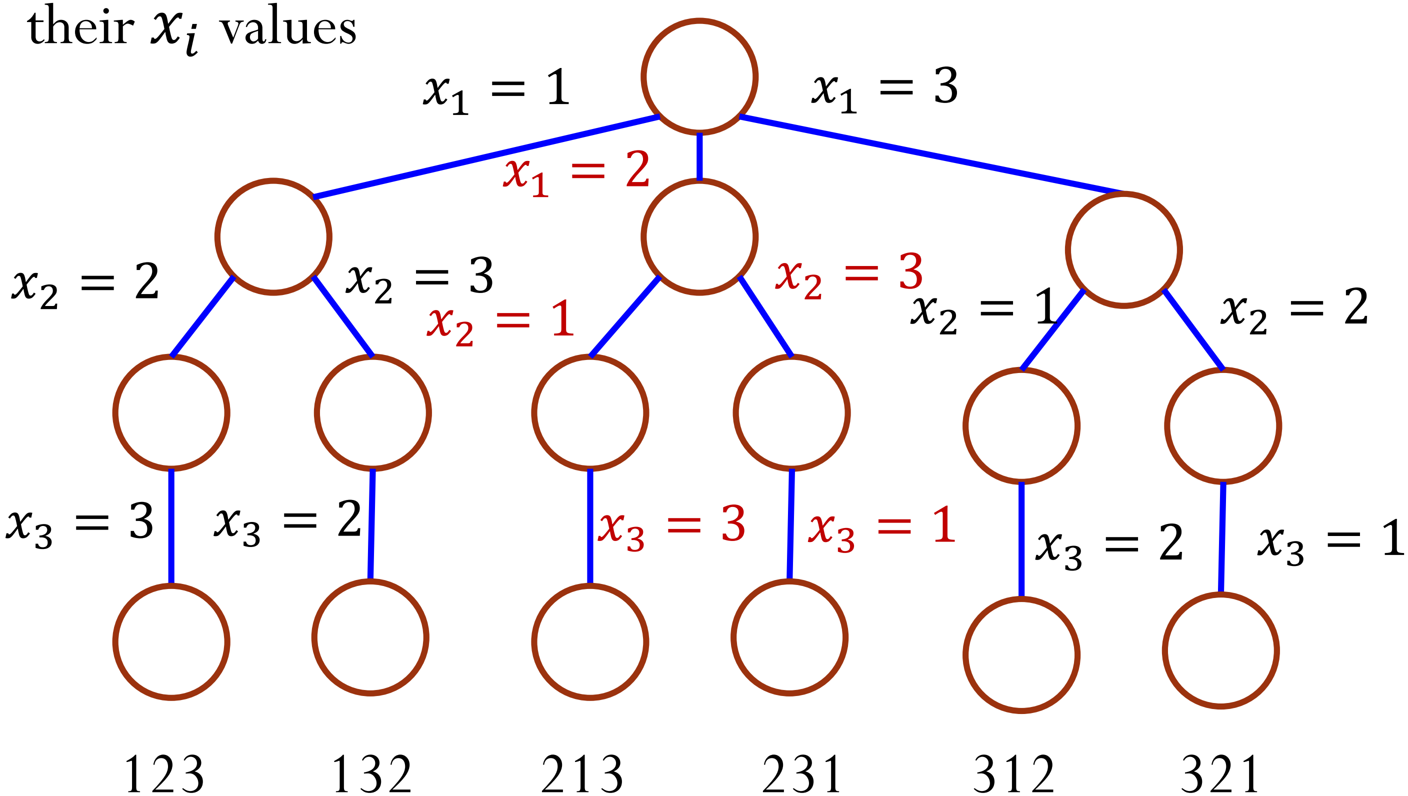
Tree Organization Of Solution Space

- For a size n subset problem, the tree structure has 2^n leaves
  - At level i, the members of the solution space are partitioned by their x_i values



Tree Organization Of Solution Space

- For a size n permutation problem, the tree structure has $n!$ leaves
  - At level i, the members of the solution space are partitioned by their x_i values



Outline

- Hard problems and solution space
- Backtracking
- Branch-and-Bound

Algorithm Design Methods

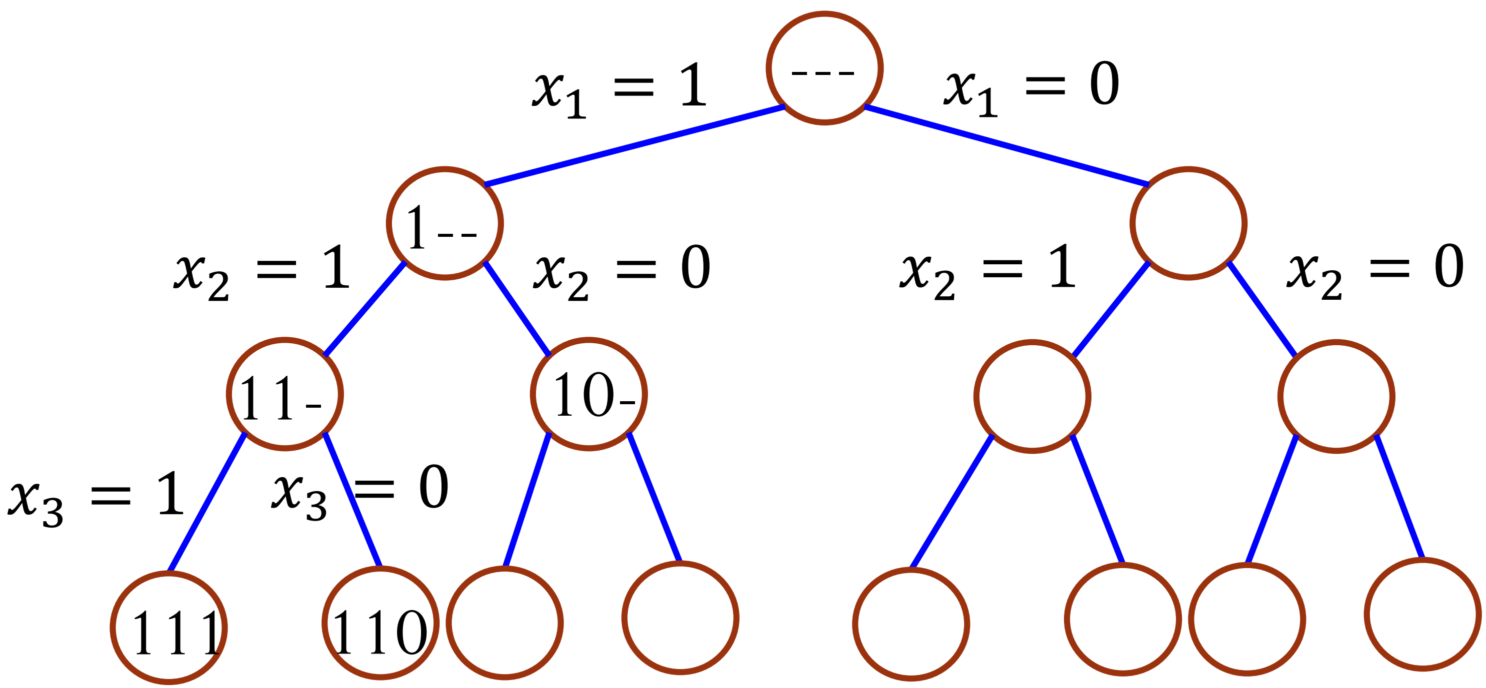
- We have learned three ways to design algorithms:
  - Greedy method.
  - Divide and conquer.
  - Dynamic programming.

- We will briefly talk two more:
  - Backtracking.
  - Branch and bound.

Backtracking

- A strategy for searching a solution and backing up when some constraint is violated.

- Construct the state-space tree
  - nodes: partial solutions
  - edges: choices in extending partial solutions
  - branch on every possibility

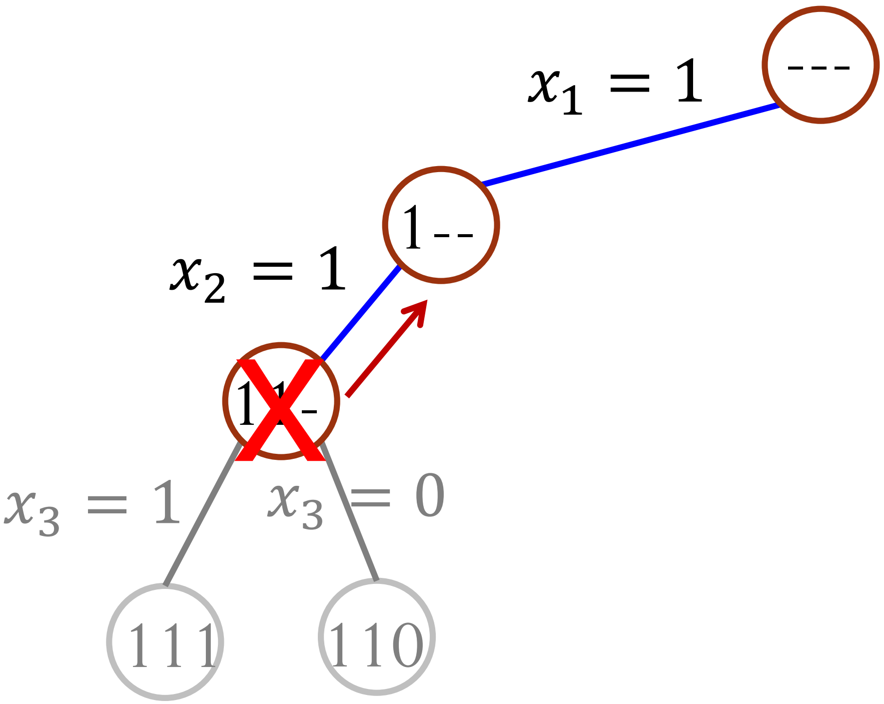


Backtracking

- Explore the state space tree using **depth-first search**, **prune** unpromising subtrees
  - Check every partial solution against constraints
  - If a partial solution violates some constraint, it makes no sense to extend it further
  - Stop exploring subtrees rooted at nodes that cannot lead to a solution and **backtrack** to such a node's parent to continue the search
  - Recursion and backtracking usually combined together to solve the problem

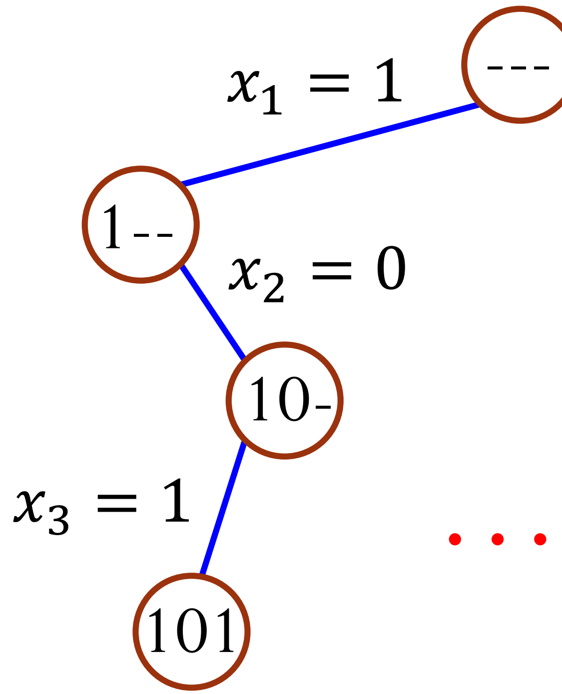
Backtracking Example

- Subset sum problem: $S = [10,5,2]$ and $c = 14$



Backtracking Example

- Subset sum problem: $S = [10,5,2]$ and $c = 14$



Backtracking: General Form

```
void checknode(node v) {
  // v: node in state-space tree

  if (promising(v)){
    if (solution(v)) then
      return soln;
    else
      for each child u of v
        checknode(u);
  }
}
```

promising(v): check whether partial solution v satisfies all constraints

solution(v): if v is already a full solution

Backtracking: Summary

- Backtracking allows pruning of unpromising branches in state-space tree
  - Better than brute-force enumeration
- All backtracking algorithms have a similar form (pruned DFS)
- Often, most difficult/costly part is determining **promising()**

Outline

- Hard problems and solution space
- Backtracking
- Branch-and-Bound

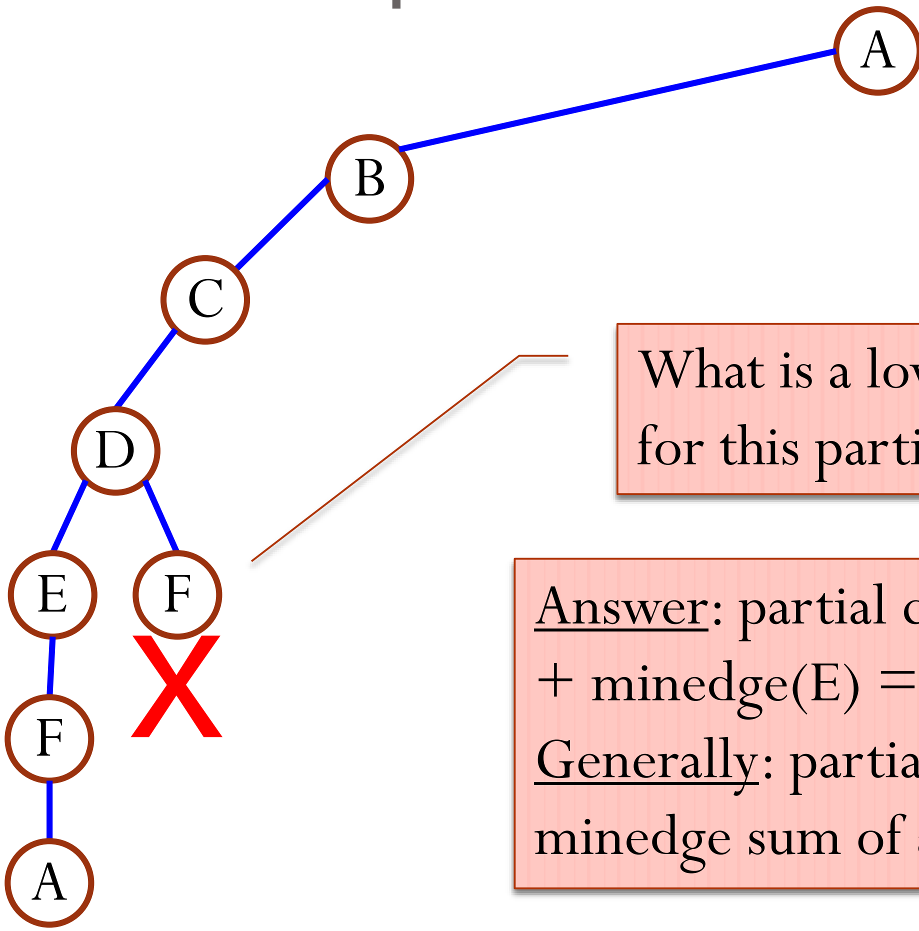
Branch-and-Bound

- **Branch**: enumerate all possible next steps from current partial solution and construct the state-space tree
- **Bound**: if a partial solution violates some constraint or if the objective function evaluates to a higher cost than that of the best full solution so far, **prune** the branch
  - Typically, we evaluate a **lower bound** of the partial solution. If lower bound >= best full solution so far, can prune
- Branch-and-bound can be used with or without backtracking:
  - If **DFS** is used, once a branch is pruned, backtrack to the previous partial solution and try another branch
  - If **BFS** is used, branch-and-bound is done without backtracking

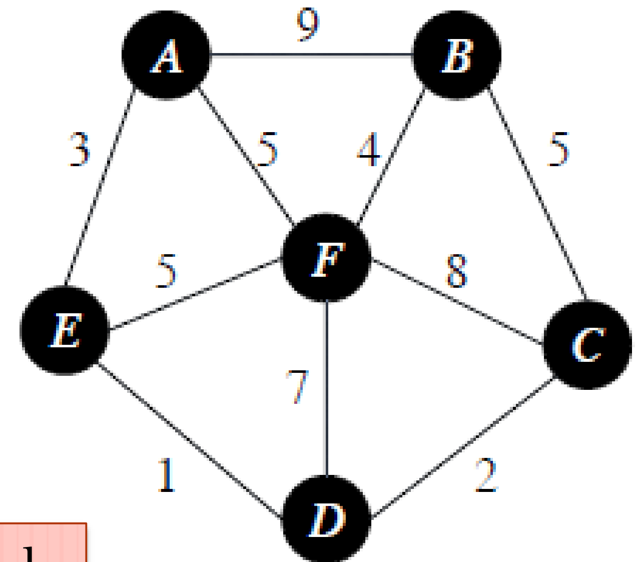
Branch-and-Bound

- The efficiency of branch-and-bound is based on **pruning** unpromising partial solutions
  - The sooner (higher up in the state-space tree) you know a solution is unpromising, the less time you spend on its subtree
  - The more accurately you can bound the solution cost, the better
  - Sometimes it is worth spending extra effort to compute better bounds
  - If no such info is available, assume solution cost is ∞ and branch-and-bound degenerates into enumeration

Example: TSP



27



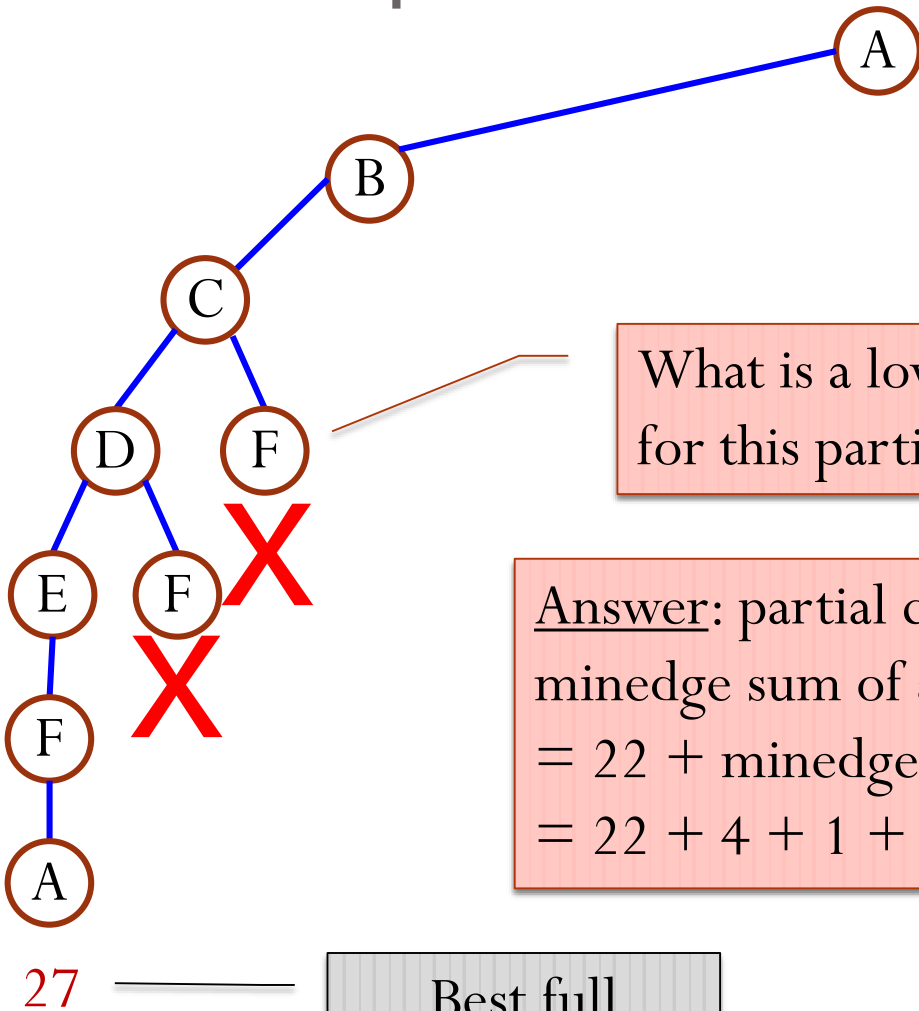
What is a lower bound for this partial solution?

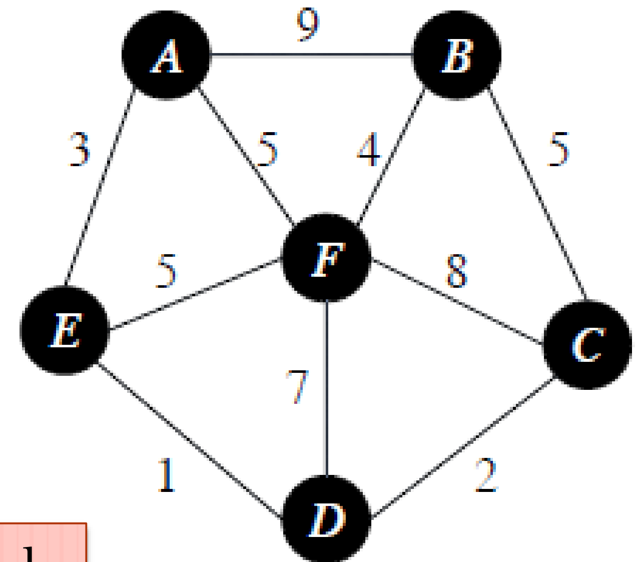
<u>Answer</u>: partial cost + minedge(F) + minedge(E) = 23 + 4 + 1 = 28

<u>Generally</u>: partial cost + minedge(current node) + minedge sum of all rest unvisited nodes

Best full solution so far

Example: TSP



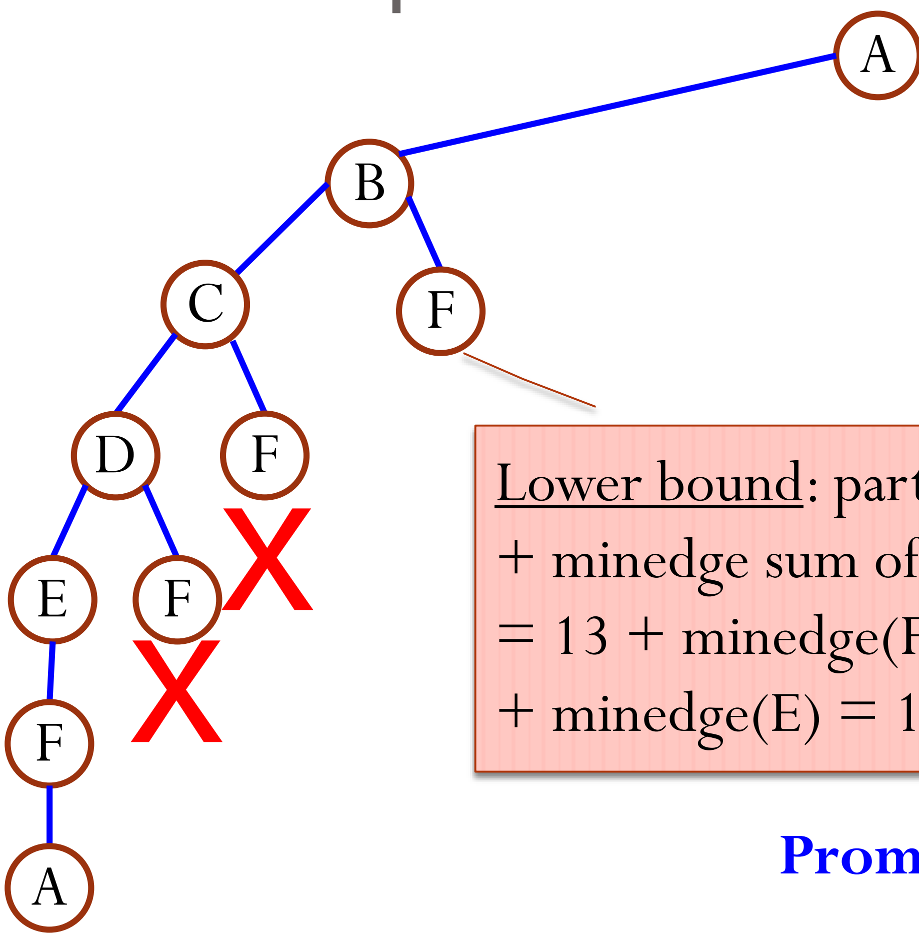


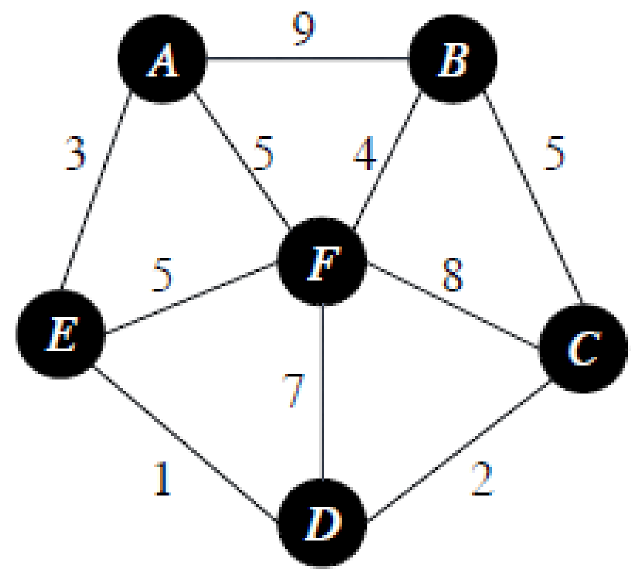
What is a lower bound for this partial solution?

Answer: partial cost + minedge(current node) + minedge sum of all rest unvisited nodes
= 22 + minedge(F) + minedge(D) + minedge(E)
= 22 + 4 + 1 + 1 = 28

Best full solution so far

Example: TSP





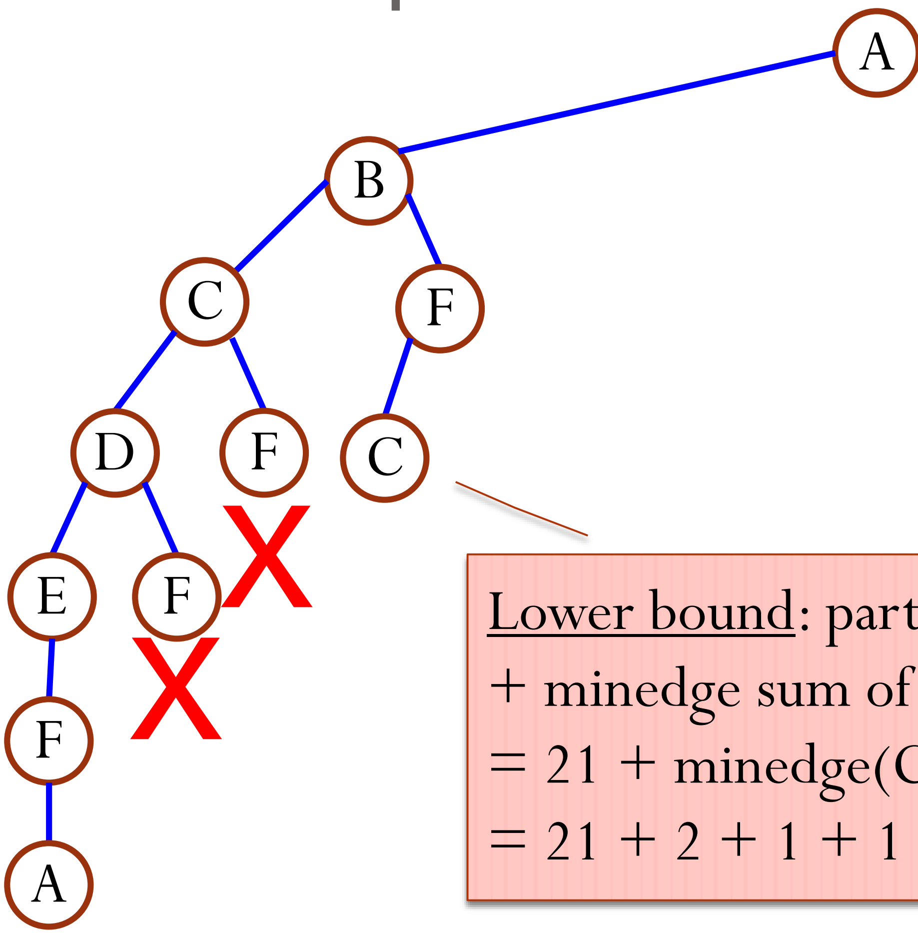
Lower bound: partial cost + minedge(current node) + minedge sum of all rest unvisited nodes
= 13 + minedge(F) + minedge(C) + minedge(D) + minedge(E) = 13 + 4 + 2 + 1 + 1 = 21

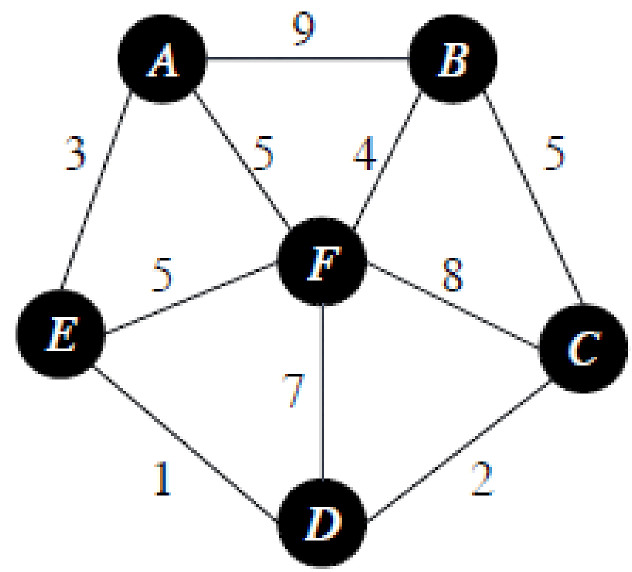
Promising. Continue branch!

27

Best full solution so far

Example: TSP





<u>Lower bound</u>: partial cost + minedge(current node) + minedge sum of all rest unvisited nodes
= 21 + minedge(C) + minedge(D) + minedge(E)
= 21 + 2 + 1 + 1 = 25

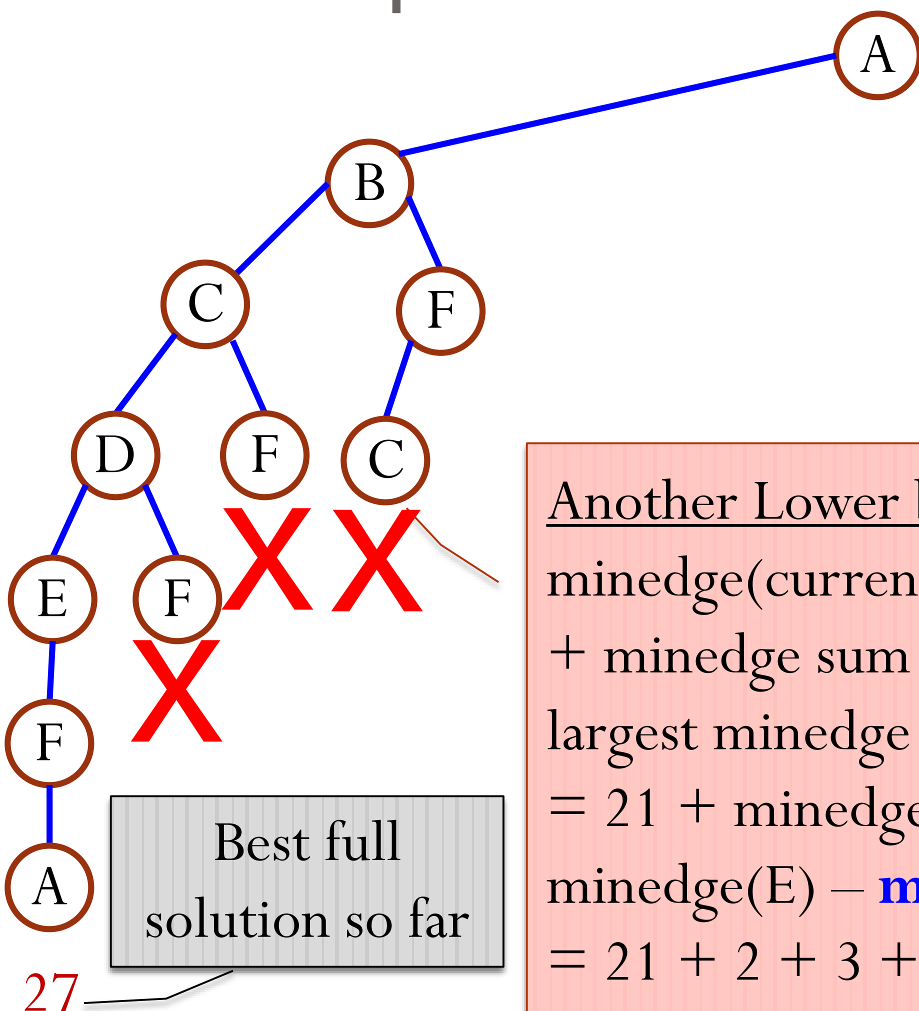
27

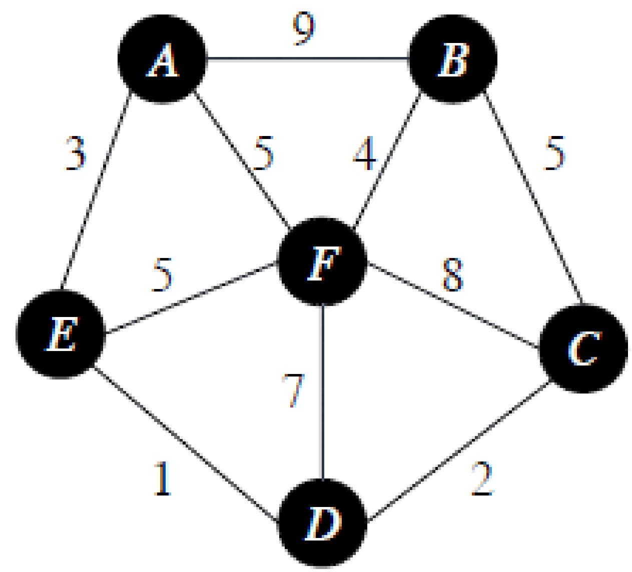
Best full solution so far

Promising. Continue branch!

… but, is there a better bound?

Example: TSP





<u>Another Lower bound</u>: partial cost + minedge(current node) + minedge(root) + minedge sum of all rest unvisited nodes except the largest minedge

= 21 + minedge(C) + minedge(A) + [minedge(D) + minedge(E) – **max** {minedge(D), minedge(E)}]

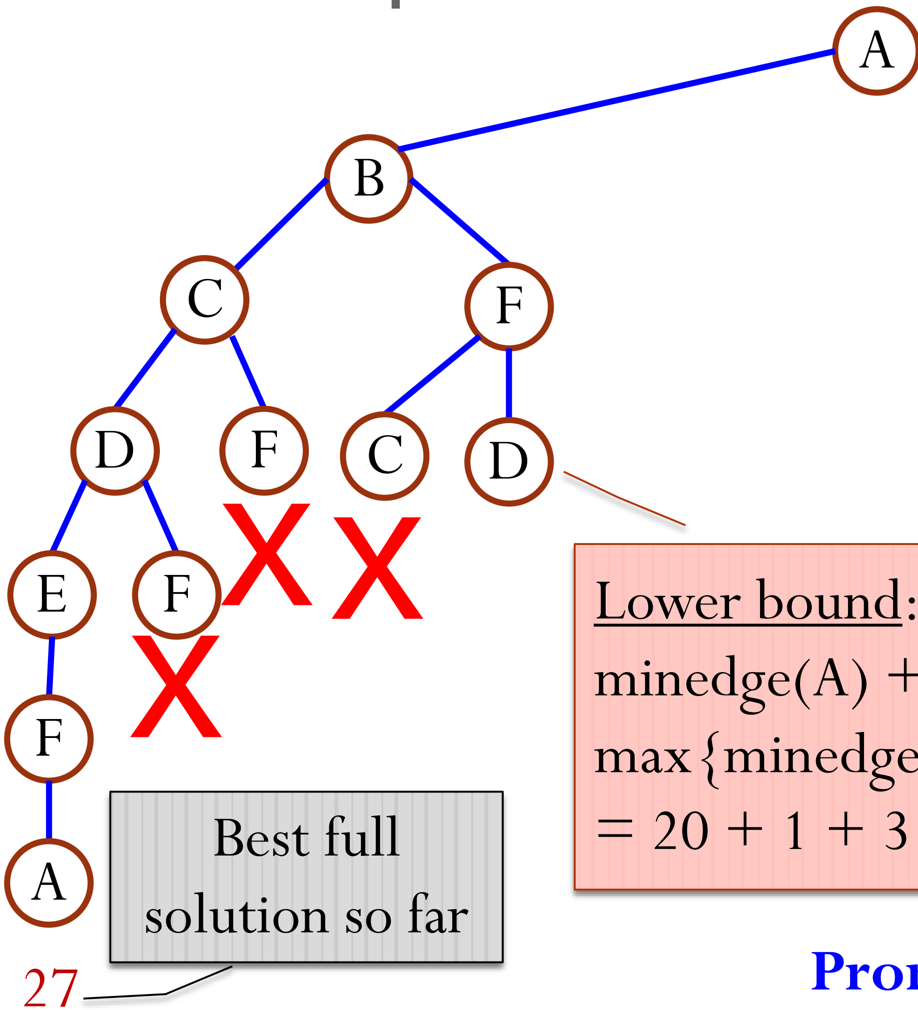
= 21 + 2 + 3 + [1+1-1] = 27

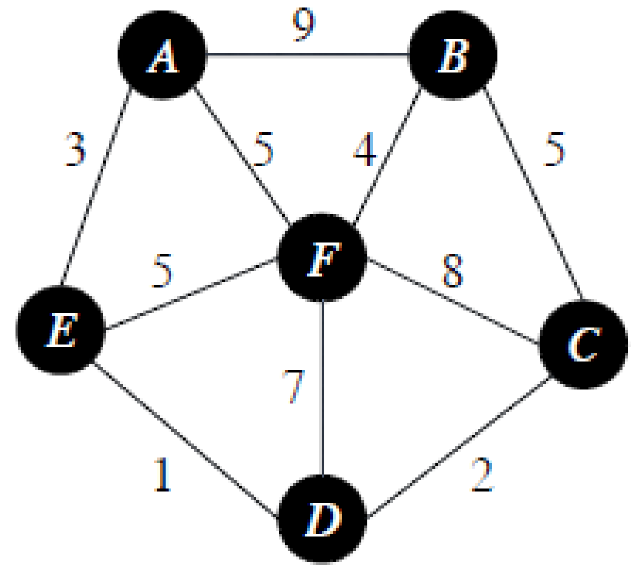
If the bound is **not strictly** smaller, can also prune!

Branch-and-Bound

- The efficiency of branch-and-bound is based on **pruning** unpromising partial solutions
  - The sooner (higher up in the state-space tree) you know a solution is unpromising, the less time you spend on its subtree
  - The more accurately you can bound the solution cost, the better
  - Sometimes it is worth spending extra effort to compute better bounds

Example: TSP

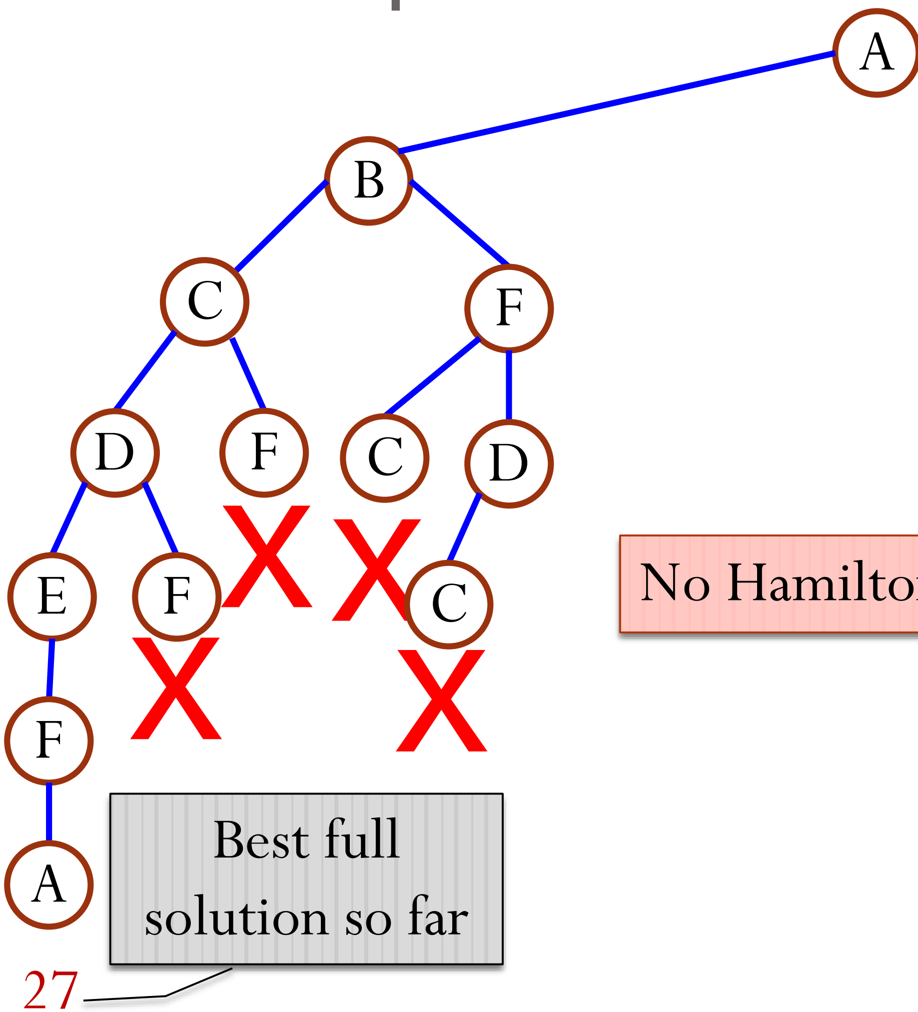




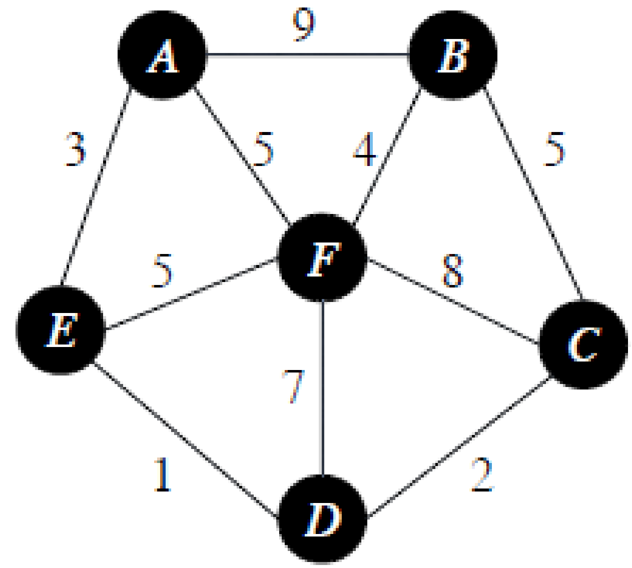
<u>Lower bound</u>: = partial cost + minedge(D) + minedge(A) + [minedge(E) + minedge(C) – max{minedge(C), minedge(E)}]
= 20 + 1 + 3 + [1+2-2] = 25

Promising. Continue branch!

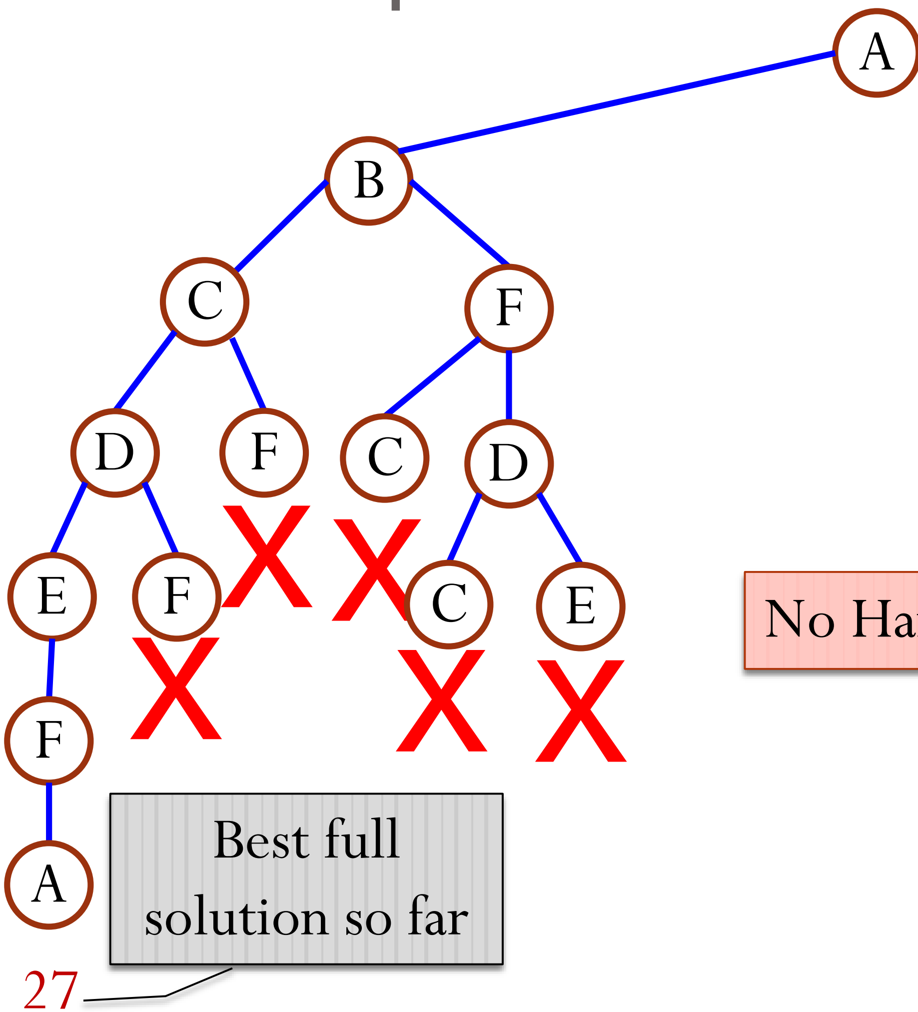
Example: TSP

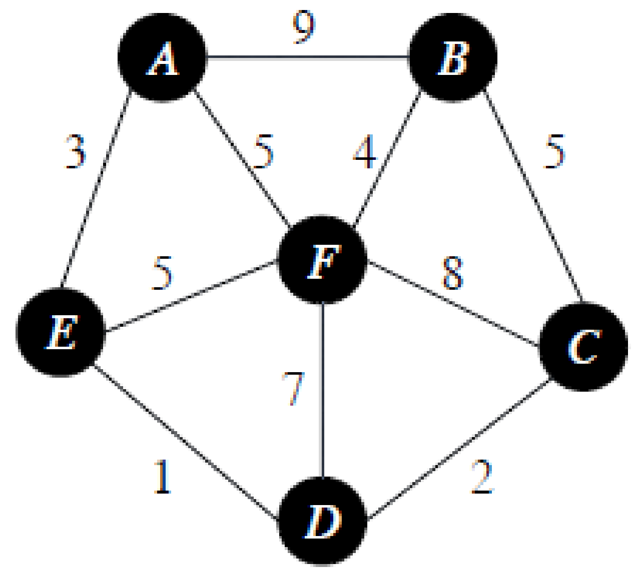


No Hamiltonian cycle possible!



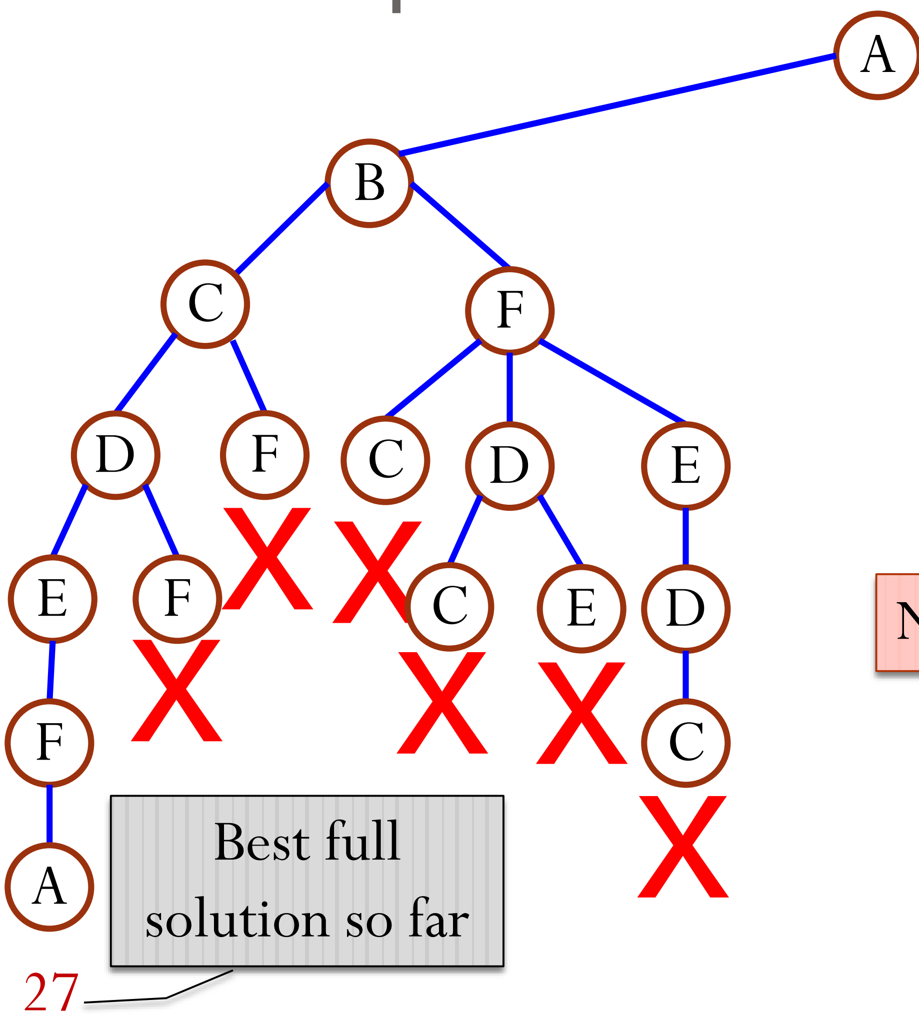
Example: TSP



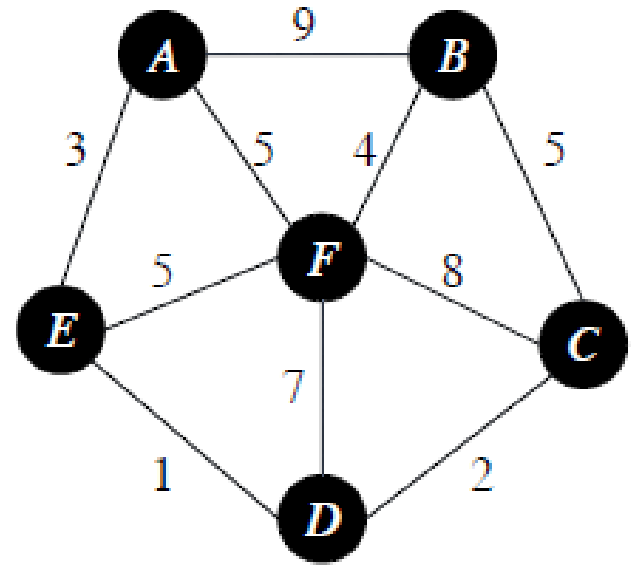


No Hamiltonian cycle possible!

Example: TSP

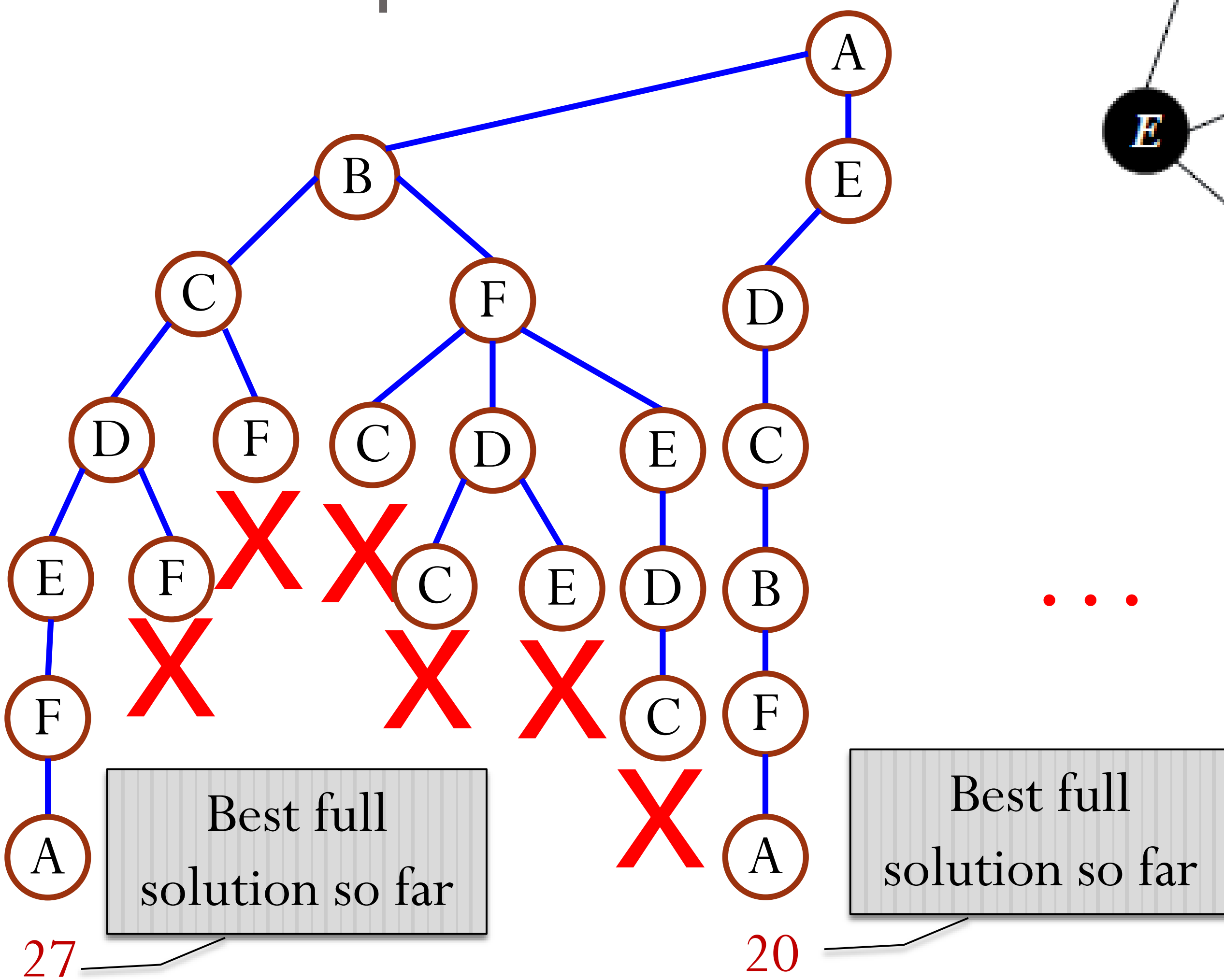


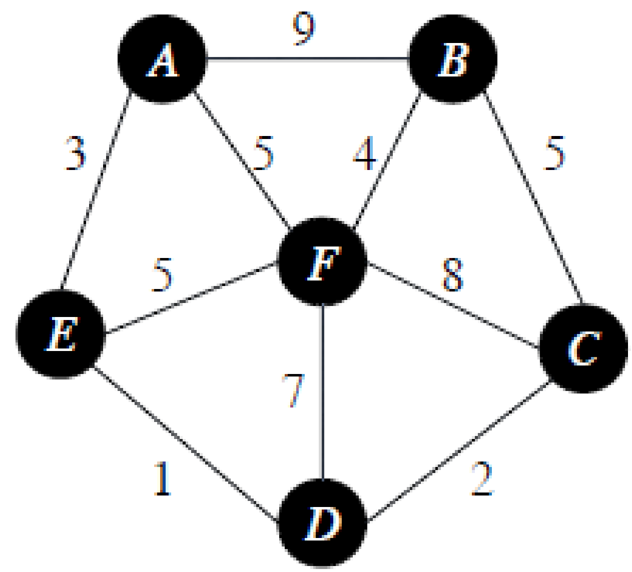
Best full solution so far



No Hamiltonian cycle possible!

Example: TSP





Branch-and-Bound: Summary

- Bound is important
  - The sooner (higher up in the state-space tree) you know a solution is unpromising, the less time you spend on its subtree
  - The more accurately you can bound the solution cost, the better
  - Sometimes it is worth spending extra effort to compute better bounds

- Constructing an initial good solution will also help reduce runtime
  - For 0/1 knapsack, can use a greedy algorithm to find an initial good solution