Lecture 21

Branch & Bound,

Traveling Salesperson Problem

EECS 281: Data Structures & Algorithms

http://xkcd.com/399

Recall: Types of Problems

- Constraint Satisfaction problems
  - Can we satisfy all given constraints?
  - If yes, how do we satisfy them?
    (need a specific solution)
  - May have more than one solution
  - Examples: sorting, mazes, spanning tree
- Optimization problems
  - Must satisfy all constraints (can we?) and
  - Must minimize an objective function subject to those constraints
  - Examples: giving change, Minimum Spanning Tree

Recall: Types of Problems

- Constraint Satisfaction problems
  - Stop when found a satisfying solution
- Optimization problems
  - Can’t stop early
  - Must develop set of possible solutions
    - Called *feasibility or promising set*
  - Then, must pick ‘best’ solution

Recall: Types of Problems

- Constraint Satisfaction problems
  - are to *Backtracking*, as
- Optimization problems
  - are to *Branch and Bound*

Outline

- We will discuss **branch & bound** through the lens of the **Traveling Salesperson Problem**

Hamiltonian Cycle

Definition: Given a graph $G = (V, E)$, find a cycle that traverses each node exactly once

Note: No vertex (except the first/last) may appear twice

Traveling Salesperson Problem

Definition: Hamiltonian cycle with least weight

A Hamiltonian Cycle is a cycle which includes every vertex

Types of Algorithm Problems

- Constraint Satisfaction problems
  - e.g., Hamiltonian Cycle
  - are to *Backtracking*, as
- Optimization problems
  - e.g., Traveling Salesperson
  - are to *Branch and Bound*

Recall: Backtracking

- Branch on every possibility
- Maintain one or more “partial solutions”
- Check every partial solution for validity
  - If a partial solution violates some constraint, it makes no sense to extend it (so drop it), i.e., **backtrack**

Branch-and-bound, a.k.a. B&B

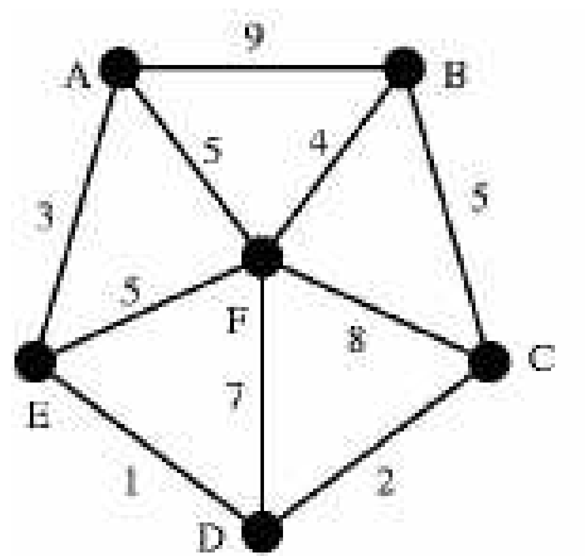
- The idea of backtracking extended to *optimization* problems
- You are minimizing a function with this useful property:
  - A partial solution is pruned if its cost $\geq$ cost of best known complete solution
  - e.g., the length of a path or tour
- If the cost of a partial solution is too big **drop this partial solution**

Bounding in B&B

- The efficiency of B&B is based on "bounding away" (aka "pruning") unpromising partial solutions
- The earlier you know a solution is not promising, the less time you spend on it
- The more accurately you can compute partial costs, the earlier you can bound
- Sometimes it's worth spending extra effort to compute better bounds

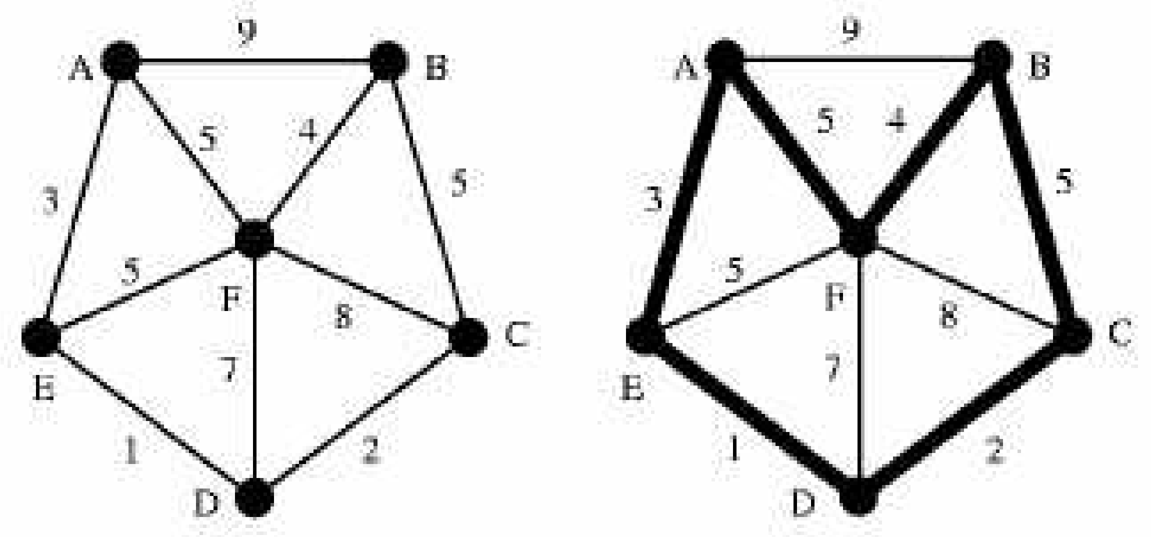
Example: TSP

- Find tour of minimum length starting and ending in same city and visiting every city exactly once

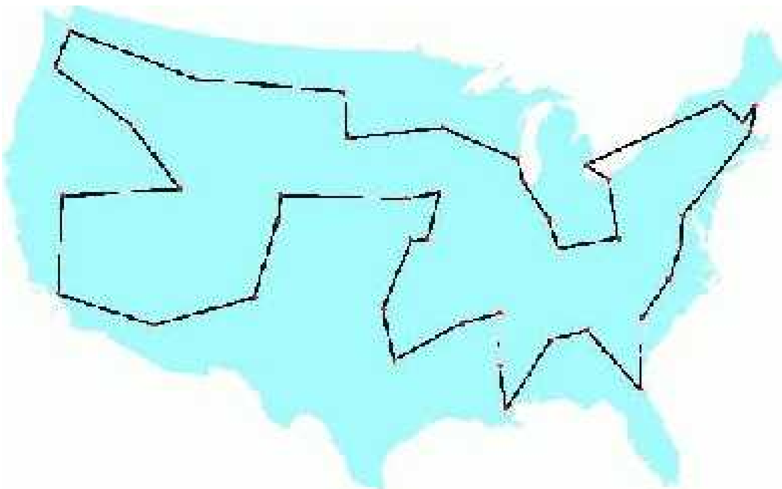


Example: TSP

- Find tour of minimum length starting and ending in same city and visiting every city exactly once



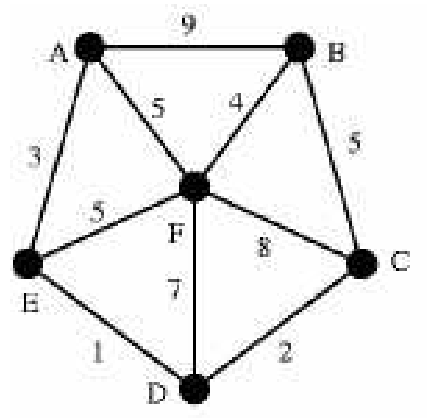
TSP: (NP) Hard Problem!

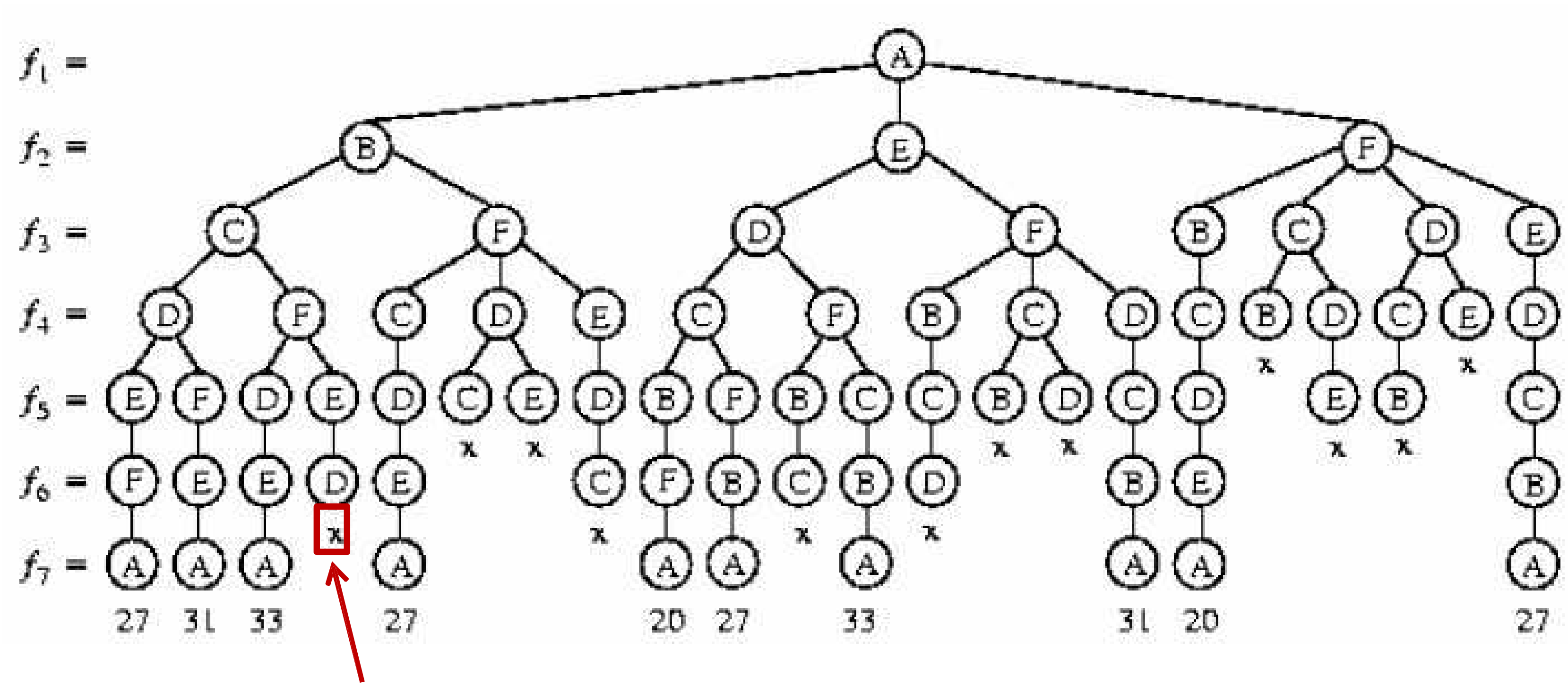


1954: n = 49

2004: n = 24978

TSP with Backtracking



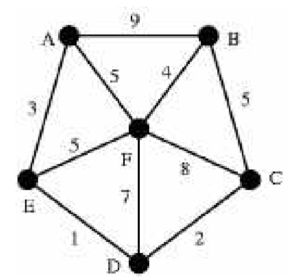


Dead end in the graph = unprompting partial solution
(adjacent vertices are already visited)

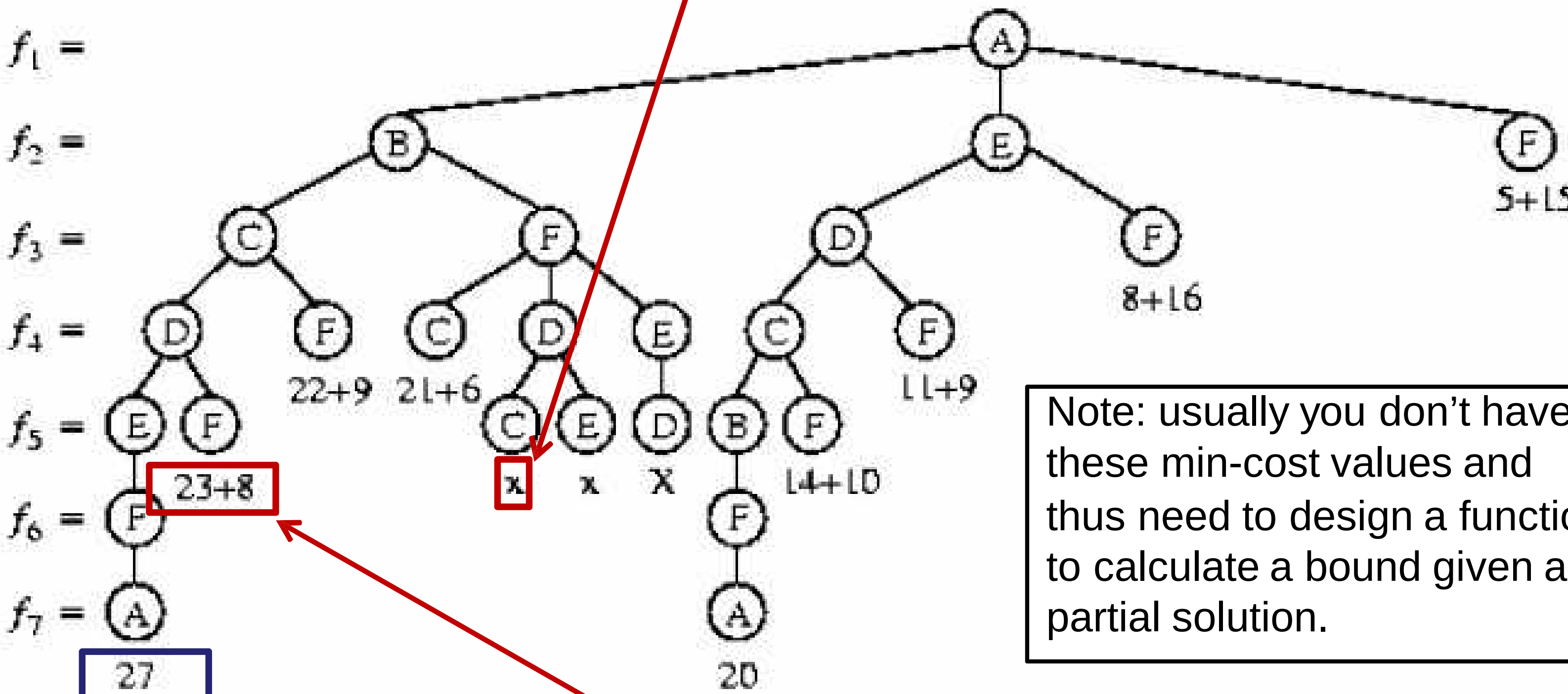
Key to B&B is **Bound**

- Start with an “infinity” bound
- Find first complete solution – use its cost as an upper bound to prune the rest of the search
- If another complete solution yields a lower cost – that will be the new upper bound
- When search is done, the current upper bound will be min

Advantage of TSP with B&B



Dead end



Note: usually you don't have these min-cost values and thus need to design a function to calculate a bound given a partial solution.

Best solution so far

Min cost if we complete a cycle from this partial solution.

If > 27 ➔ unpromising partial solution

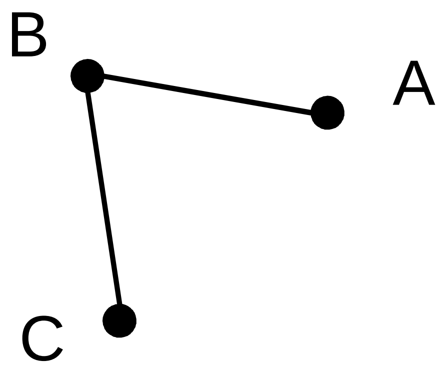
Bounding Function

- Some vertices are connected so far, some vertices are not
- For ONLY the unvisited vertices, connect them together with lowest possible cost
- Then connect the visited vertices to the unvisited
- Yes, this function considers solutions that violate constraints, but it’s a LOWER bound so it’s OK

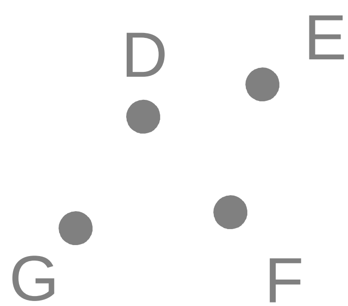
Bounding Function

- Estimate must be ≤ reality
- The bounding function must have complexity better than just continuing TSP for the *k* vertices not yet visited:
  - For instance, $O(k^2)$ is better than $O(k!)$ for most values of *k*
- What method can we use to find the lowest cost way to connect *k* vertices together in $O(k^2)$ time?

Partial TSP Example



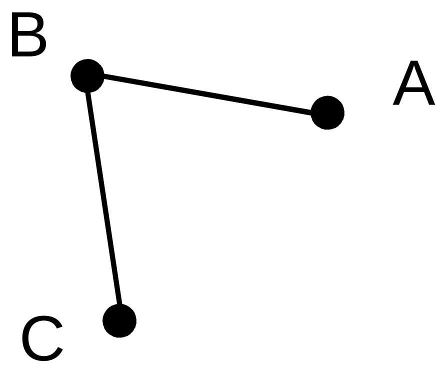
Current path: A - B - C



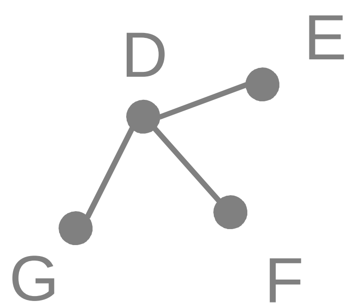
What’s the best way
to connect D, E, F, and G
to each other?

Unvisited vertices: D, E, F, and G

Connect Unvisited Nodes Together



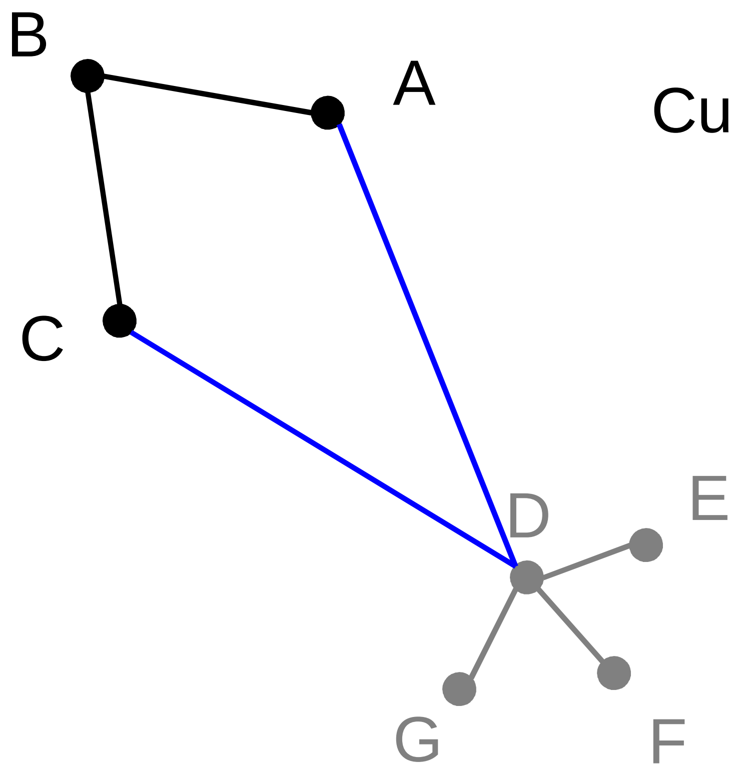
Current path: A - B - C



How many edges are we missing? A full TSP tour would have V edges (7), currently we have 5...

Unvisited vertices: D, E, F, and G

Connect Partial Tour to Unvisited



Current path: A - B - C

Connect from A-B-C to D-E-F-G in the best, cheapest, fastest way possible

Unvisited vertices: D, E, F, and G

General Form: Branch & Bound

```
type checknode(node v, type currBest)
    node u
    if (promising(v, currBest))
        if (solution(v)) then
            update(currBest)
        else
            for each child u of v
                checknode(u, currBest)
    return currBest
```

General Form: Branch & Bound

`solution()`

- Check ‘depth’ of solution (Constraint satisfaction)

`update()`

- If new solution better than current solution, then update (Optimization)

`checknode()`

- Called only if promising and not solution

General Form: Branch & Bound

`promising()`

- Different for each application, but must return false when lowerbound(v) ≥ currBest
- A return of false is what causes pruning

`lowerbound()`

- Estimate of solution based upon
  - Cost so far, plus
  - Under estimate of cost remaining (aka bound)

Key to B&B is **Bound**

We can get smarter and smarter on the bound

However, calculation of the bound may become prohibitive

Generating Permutations

```
template <typename T>
void genPerms(deque<T> &q, vector<T> &s) {
    // s: prefix of permutation, q: everything else
    unsigned size = q.size();
    if (q.empty()) {
        cout << s << '\n';
        return;
    } // if
    for (unsigned k = 0; k != size; k++) {
        s.push_back(q.front());
        q.pop_front(); genPerms(q,
        s); q.push_back(s.back());
        s.pop_back();
    } // for
} // genPerms()
```

For Project 4

```
template <typename T>
void genPerms(deque<T> &unvisited, vector<T>     &path) {
    if (unvisited.empty()) {
        // Do something with the path
        return;
    } // if
    if (!promising(unvisited, path))
        return;
    for (unsigned k = 0; k != unvisited.size();           k++) {
        path.push_back(unvisited.front());
        unvisited.pop_front();
        genPerms(unvisited, path);
        unvisited.push_back(path.back());
        path.pop_back();
    } // for
} // genPerms()
```

Summary Branch and Bound

- Method to prune search space for optimization problems
- Need to keep current best solution
- Need to have lower bound estimate of alternative paths
- If lower bound estimate is greater than current best, then prune