Exam 3 Review

CSE 101

Winter 2023

Exam Details

- In class
- Randomized assigned seats
- You may use 6 one-sided pages of notes
- No textbook or electronic aids
- No need to provide proofs unless asked for
- 3 Questions in 45 minutes
  - 1st straightforward implementation of algorithm
  - 2nd requires some thought
  - 3rd can be quite tricky

This Review

- Brief outline of topics that might show up on the exam
- To see anything in more depth use other review options.

Other Review Options

- Lecture podcasts / slides
- Textbook
- OH questions
- Old exams from problem archive

Topics

- Greedy Algorithms
  - Huffman Codes
  - MSTs

- Dynamic Programming
  - Basic Concepts
  - LCSS
  - Knapsack
  - CMM
  - All-Pairs Shortest Path
  - MIS in Trees
  - Traveling Salesman

Non-Fix Length Encodings

- Suppose instead we had to decode:

`AAABAACBAABADAAA`

- 16 Letters requires 32 bits.
- Note that there are a lot of `A`s here. If we could find a way to encode them with fewer bits, we could save a lot.

Prefix Free Encodings

Definition: An encoding is prefix-free if the encoding of no letter is a prefix of the encoding of any other.

Optimal Encoding

Problem: Given a string, S, find a prefix-free encoding that encodes S using the fewest number of bits.

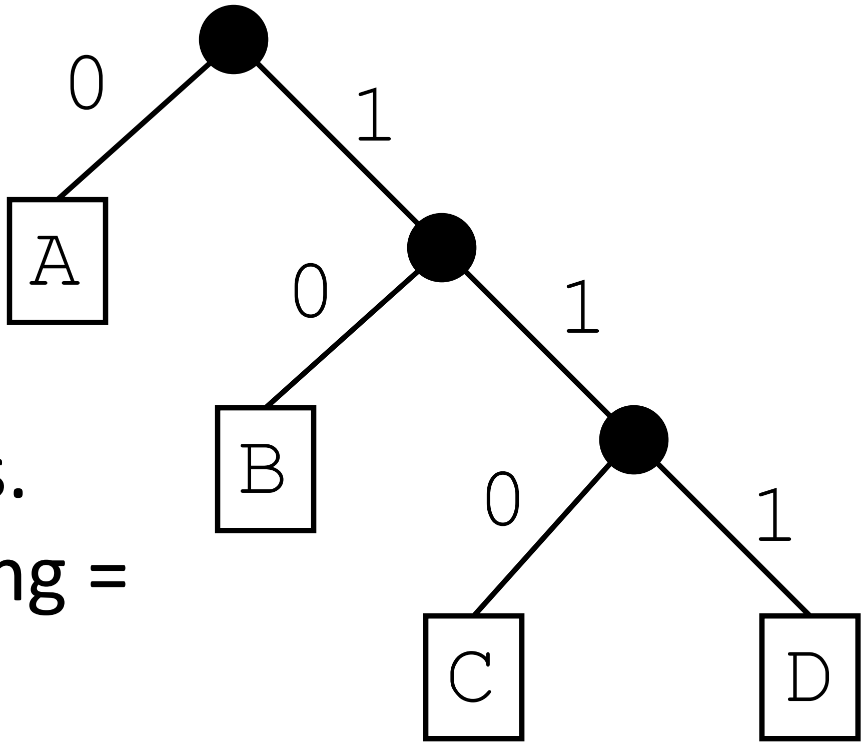
How Long is the Encoding?

If for each letter x in our string, x appears f(x) times and if we encode x as a string of length ℓ(x), the total encoding length is:

$$\sum f(x)\cdot\ell(x).$$

Tree Representation

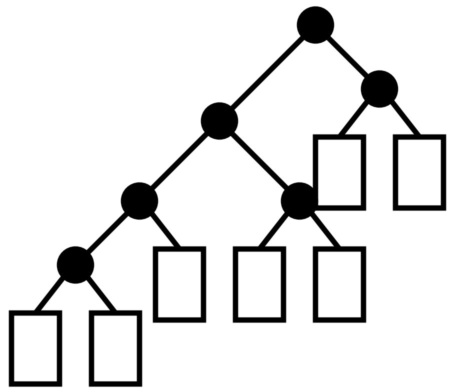
Can represent prefix-free encoding as a tree.



Letters are leaves.
Length of encoding =
Depth of leaf.

Placement of Leaves

Suppose we know the tree structure. Where do we put the letters?



Objective = Σ freq(x)depth(x)

Want least frequent letters at lowest depth.

<u>Letter frequencies</u>

```
Ax10, Bx15,
Cx4,  Dx22,
Ex31, Fx5,
Gx19
```

Siblings

- No matter what the tree structure, two of the deepest leaves are siblings.
- Can assume filled by two least frequent elements.
- Can assume that two least frequent elements are siblings!

Algorithm

```
HuffmanTree(L)
  While(at least two left)
    x, y ← Two least frequent
    z new node f(z) ← f(x)+f(y)
    x and y children of z
    Replace x and y with z in L
  Return remaining elt of L
```

Optimized Algorithm

```
HuffmanTree(L)
  Priority queue Q                          } O(n log(n))
  Insert all elements of L to Q             }
  While(|Q| ≥ 2)  } O(n) Iterations
    x ← Q.DeleteMin()                       }
    y ← Q.DeleteMin()                       }
    Create z, f(z) = f(x) + f(y)            } O(log(n))
    x and y children of z                   }
    Q.Insert(z)                             }
  Return Q.DeleteMin()
```

Runtime: O(n log(n))

Trees

Definition: A tree is a connected graph, with no cycles.

A spanning tree in a graph G, is a subset of the edges of G that connect all vertices and have no cycles.

If G has weights, a minimum spanning tree is a spanning tree whose total weight is as small as possible.

Basic Facts about Trees

Lemma: For an undirected graph G, any two of the below imply the third:

1. |E| = |V|-1
2. G is connected
3. G has no cycles

Corollary: If G is a tree, then |E| = |V|-1.

Minimum Spanning Tree

Problem: Given a weighted, undirected graph G, find a spanning tree of G with the lowest possible weight.

Greedy Idea

How do you make an MST?

- Try using the cheapest edges.

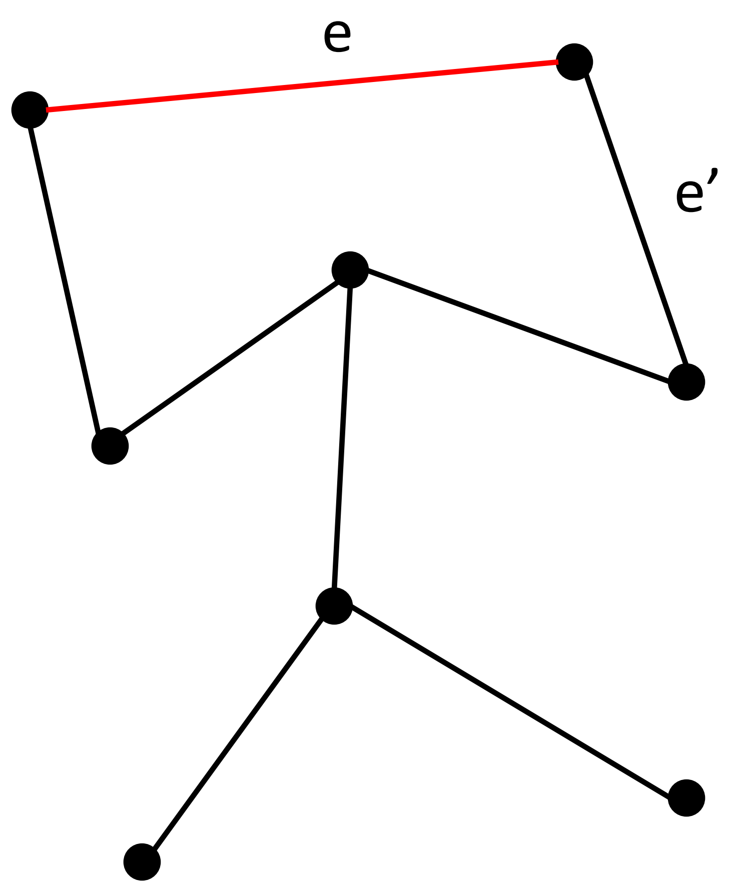
<u>Proposition:</u> In a graph G, let e be an edge of lightest weight. Then there exists an MST of G containing e. Furthermore, if e is the unique lightest edge, then *all* MSTs contain e.

Proof

- Consider tree T not containing e.
- With extra edge, no longer a tree, must contain a cycle.
- Remove edge e' from cycle to get T'.
- |T'|=|V|-1, and connected, so T' is a tree.
- wt(T') = wt(T)+wt(e)-wt(e')

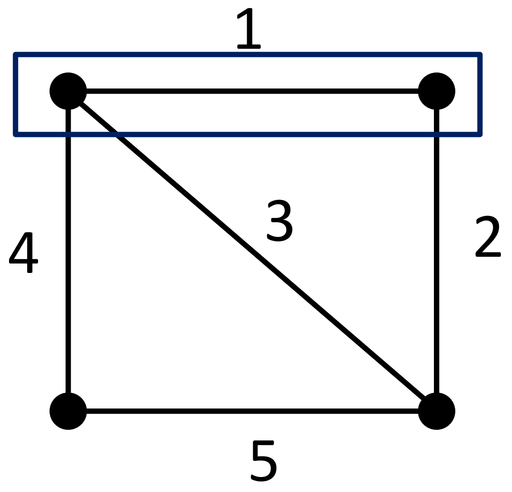
  $\leq$ wt(T)

(because wt(e) is minimal).



Example

- Lightest edge in MST.
  - Then what?
- Merge those vertices & repeat.



Algorithm

- When more than one vertex, add lightest edge, and merge.
  - Repeat and then undo merges.
- Easier: An edge hasn’t been merged away iff it does not create a cycle with already chosen edges.

Algorithm

```
Kruskal(G)
  T ← {}
  While(|T| < |V|-1)
    Find lightest edge e that
      doesn't create cycle with T
    Add e to T
  Return T
```

Optimizations

Two things are slow here:

1) Testing every edge every iteration.
2) Needing to test connectivity for every edge.

Kruskal Version 2

```
Kruskal(G)
  Sort edges by weight
  T ← {}
  For e ∈ E in increasing order
    If e does not form cycle
      Add e to T
  Return T
```

Better Cycle Testing

How do we test if edge (v,w) forms a cycle?

If v and w are in the same connected component of the graph formed by T.

Need a data structure. That can:

- Add edges to T.
- Test if two vertices in same CC.

Union Find Data Structure

Maintains several sets. Each has a representative element.

Operations:

- New(e) – Creates a new set with element e.
- Rep(a) – Returns the representative element of a’s set.
- Join(a,b) – Merges a’s set with b’s.

Note: Check of v & w in same set by testing if Rep(v) = Rep(w).

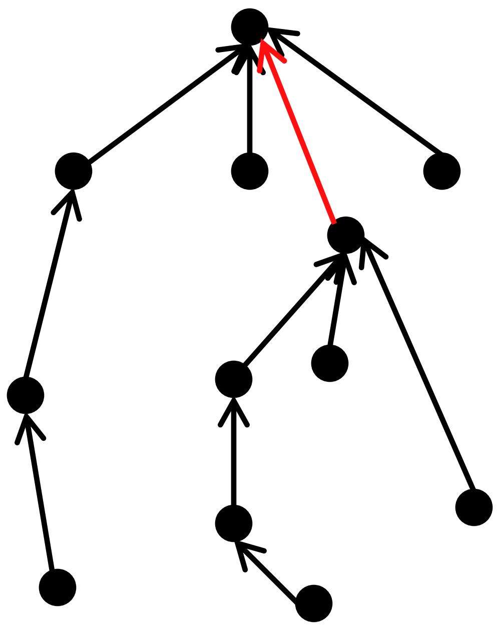
Kruskal Version 3

```
Kruskal(G)
  Sort edges by weight
  T ← {}
  Create Union Find
  For v ∈ V, New(v)
  For (v,w) ∈ E in increasing order
    If Rep(v) ≠ Rep(w)
      Add (v,w) to T
      Join(v,w)
  Return T
```

Runtime:$O(|E|\log|E|)$

Union Find Implementation

Basic Idea: Each set is a rooted tree with edges pointing towards the representative.



New: Create new node – O(1)

Rep: Follow pointers to root

- O(depth)

Join: Have one Rep point to other.

- O(depth)

Always connect shallow to deep

Other Algorithms

There are many other ways to create MST algorithms. Kruskal searches the whole graph for light edges, but you can also grow from a point.

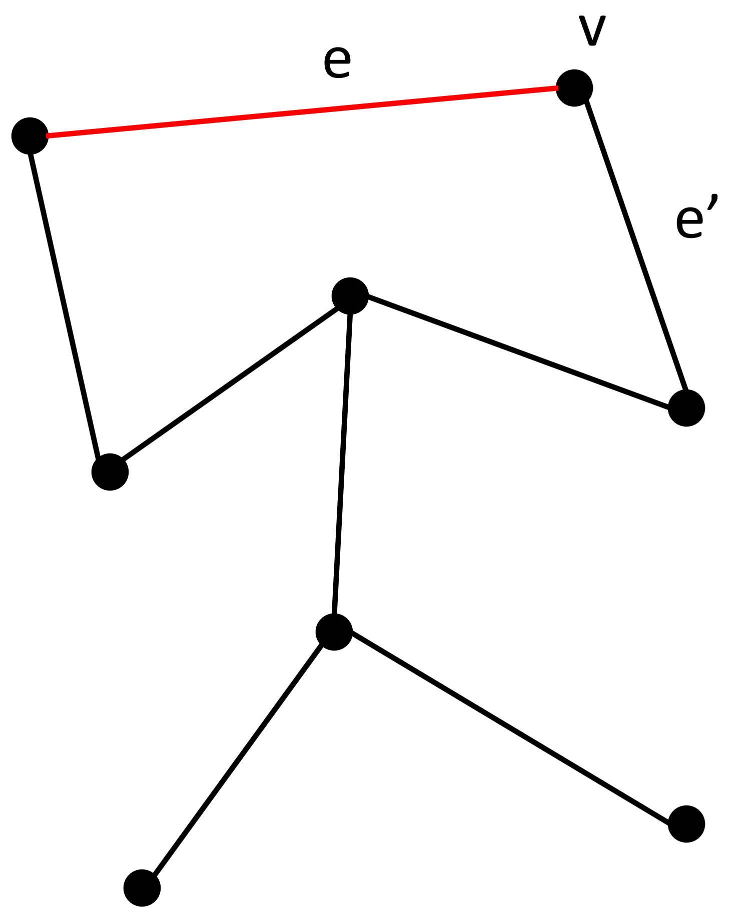
Proposition: In a graph G, with vertex v, let e be an edge of lightest weight adjacent to v. Then there exists an MST of G containing e. Furthermore, if e is the unique lightest edge, then *all* MSTs contain e.

Proof

- Consider tree T not containing e.
- With extra edge, no longer a tree, must contain a cycle.
- Remove other edge e' adjacent to v from cycle to get T'.
- |T'|=|V|-1, and connected, so T' is a tree.
- wt(T') = wt(T)+wt(e)-wt(e')

  ≤ wt(T)

(because wt(e) is minimal).



Prim’s Algorithm

So instead of checking *all* edges, you can just check edges from v.

You can then contract edge and repeat.

Prim’s Algorithm: Add lightest edge that connects v to a new vertex.

Implementation very similar to Dijkstra.

Prim's Algorithm

```
Prim(G,w)
  Pick vertex s                  \\ doesn't matter which
  For v ∈ V, b(v) ← ∞            \\ lightest edge into v
  T ← {}, b(s) ← 0
  Priority Queue Q, add all v with key=b(v)
  While(Q not empty)
    u ← DeleteMin(Q)
    If u ≠ s, add (u,Prev(u)) to T
    For (u,v) ∈ E
      If w(u,v) < b(v)
        b(v) ← w(u,v)
        Prev(v) ← u
        DecreaseKey(v)
  Return T
```

Runtime:
$O(|V|\log|V| + |E|)$
Slightly better than Kruskal

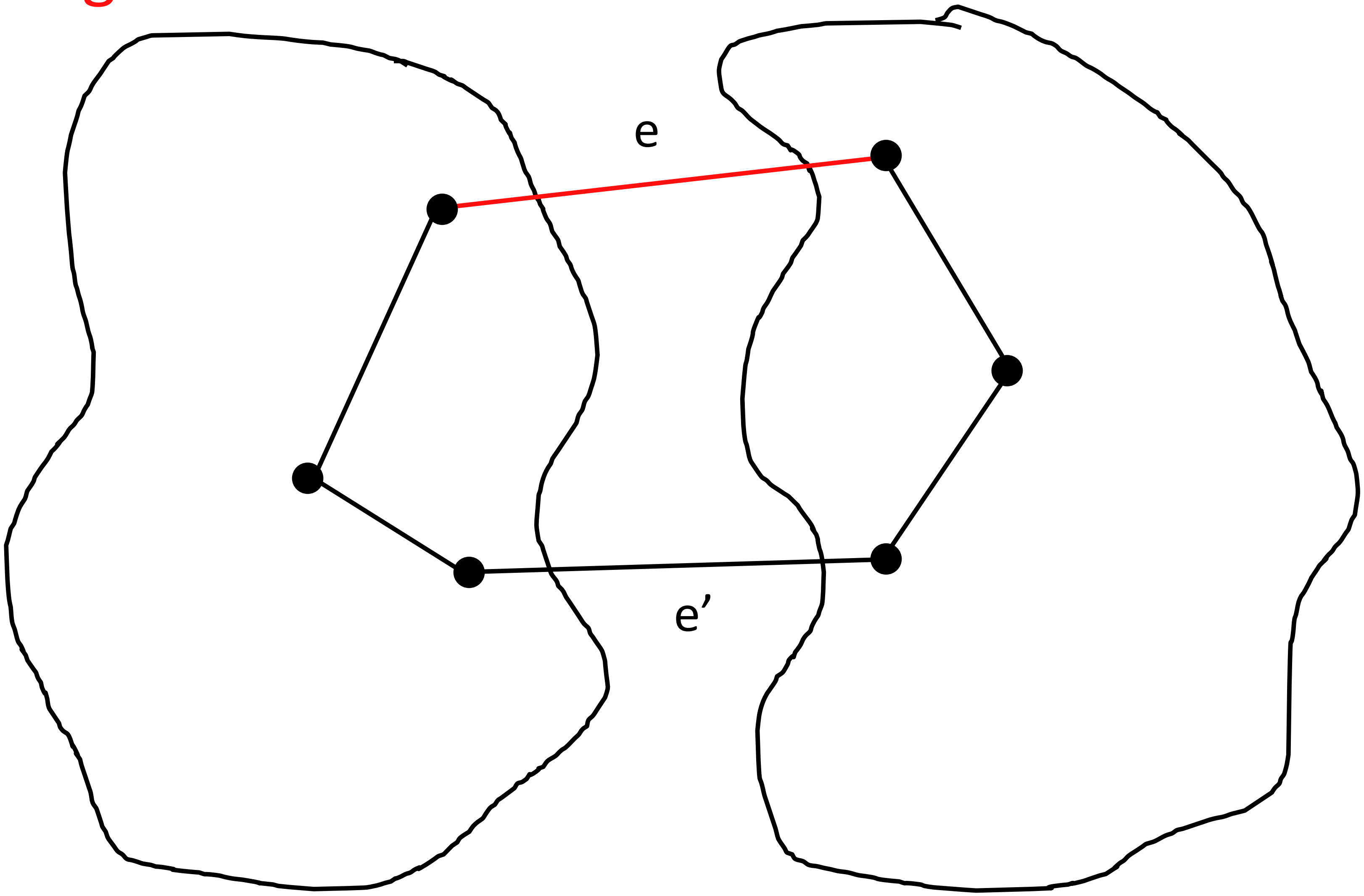
Analysis

At any stage, have some set S of vertices connected to s. Find cheapest edge connecting S to S^C.

Proposition: In a graph G, with a cut C, let e be an edge of lightest weight crossing C. Then there exists an MST of G containing e. Furthermore, if e is the unique lightest edge, then *all* MSTs contain e.

Cycle must have e’ crossing cut!

Proof



Dynamic Programming (Ch 6)

- Background and past examples
- Longest Common Subsequence
- Knapsack
- Chain Matrix Multiplication
- All-Pairs Shortest Paths
- Independent Sets of Trees
- Travelling Salesman

Simplify by Tabulating

Instead of computing these values recursively, compute them one at a time, recording them. Then in the future, you only need to do table lookups.

Dynamic Programming

Our final general algorithmic technique:

1. Break problem into smaller subproblems.
2. Find recursive formula solving one subproblem in terms of simpler ones.
3. Tabulate answers and solve all subproblems.

Subsequences

Given a sequence, say `ABCBA`, a <u>subsequence</u> is the sequence obtained by deleting some letters and leaving the rest in the same order.

Longest Common Subsequence

We say that a sequence is a common subsequence of two others, if it is a subsequence of both.

For example `ABC` is a common subsequence of $\underline{\mathtt{A}}\mathtt{D}\underline{\mathtt{BC}}\mathtt{A}$ and $\underline{\mathtt{A}}\mathtt{A}\underline{\mathtt{B}}\mathtt{B}\underline{\mathtt{C}}$.

Problem: Given two sequences compute the longest common subsequence. That is the subsequence with as many letters as possible.

Case Analysis

How do we compute LCSS($A_1A_2…A_n$, $B_1B_2…B_m$)?

Consider cases for the common subsequence:

1. It does not use A_n.
2. It does not use B_m.
3. It uses both A_n and B_m and these characters are the same.

Case 1

If the common subsequence does not use A_n, it is actually a common subsequence of

$A_1A_2…A_{n-1}$, and $B_1B_2…B_m$

Therefore, in this case, the longest common subsequence would be LCSS($A_1A_2…A_{n-1}$, $B_1B_2…B_m$).

Case 2

If the common subsequence does not use $\mathtt{B}_{\mathtt{m}}$, it is actually a common subsequence of

$\mathtt{A}_1\mathtt{A}_2\ldots\mathtt{A}_\mathtt{n}$, and $\mathtt{B}_1\mathtt{B}_2\ldots\mathtt{B}_{\mathtt{m-1}}$

Therefore, in this case, the longest common subsequence would be LCSS($\mathtt{A}_1\mathtt{A}_2\ldots\mathtt{A}_\mathtt{n}$, $\mathtt{B}_1\mathtt{B}_2\ldots\mathtt{B}_{\mathtt{m-1}}$).

Case 3

If a common subsequence uses both A_n and B_m…

- These characters must be the same.
- Such a subsequence is obtained by taking a common subsequence of:
  $A_1A_2…A_{n-1}$, and $B_1B_2…B_{m-1}$
  and adding a copy of $A_n = B_m$ to the end.
- The longest length of such a subsequence is
  LCSS($A_1A_2…A_{n-1}$, $B_1B_2…B_{m-1}$)+1.

Recursion

On the other hand, the longest common subsequence must come from one of these cases. In particular, it will always be the one that gives the biggest result.

LCSS($A_1A_2\ldots A_n$, $B_1B_2\ldots B_m$) =

    Max(LCSS($A_1A_2\ldots A_{n-1}$, $B_1B_2\ldots B_m$),
        LCSS($A_1A_2\ldots A_n$, $B_1B_2\ldots B_{m-1}$),
        [LCSS($A_1A_2\ldots A_{n-1}$, $B_1B_2\ldots B_{m-1}$)+1])

[where the last option is only allowed if A_n = B_m]

Recursion

$$\begin{tikzcd}
\boxed{\mathrm{LCSS}(A_{1\ldots n},B_{1\ldots m})} \arrow[r] \arrow[d] \arrow[dr] & \boxed{\mathrm{LCSS}(A_{1\ldots n},B_{1\ldots m-1})} \arrow[r] \arrow[d] \arrow[dr] & {} \\
\boxed{\mathrm{LCSS}(A_{1\ldots n-1},B_{1\ldots m})} \arrow[r] \arrow[d] \arrow[dr] & \boxed{\mathrm{LCSS}(A_{1\ldots n-1},B_{1\ldots m-1})} \arrow[r] \arrow[d] \arrow[dr] & {} \\
\boxed{\mathrm{LCSS}(A_{1\ldots n-2},B_{1\ldots m})} & \boxed{\mathrm{LCSS}(A_{1\ldots n-2},B_{1\ldots m-1})} & {}
\end{tikzcd}$$

Key Point: Subproblem reuse

Only ever see LCSS($A_1A_2\ldots A_k$, $B_1B_2\ldots B_\ell$)

Base Case

Our recursion also needs a base case.

In this case we have:

LCSS($\emptyset$, $B_1 B_2 \ldots B_m$) = LCSS($A_1 A_2 \ldots A_n$, $\emptyset$) = 0.

Algorithm

```
LCSS($A_1A_2…A_n$,$B_1B_2…B_m$)
  Initialize Array T[0…n,0…m]
    \\ T[i,j] will store LCSS($A_1A_2…A_i$,$B_1B_2…B_j$)
  For i = 0 to n
    For j = 0 to m
      If (i = 0) OR (j = 0)
        T[i,j] ← 0
      Else If $A_i$ = $B_j$
        T[i,j] ← max(T[i-1,j],T[i,j-1],T[i-1,j-1]+1)
      Else
        T[i,j] ← max(T[i-1,j],T[i,j-1])
  Return T[n,m]
```

Notes about DP

- General Correct Proof Outline:
  - Prove by induction that each table entry is filled out correctly
  - Use base-case and recursion
- Runtime of DP:
  - Usually
    [Number of subproblems]x[Time per subproblem]

More Notes about DP

- Finding Recursion
  - Often look at first or last choice and see what things look like without that choice
- Key point: Picking right subproblem
  - Enough information stored to allow recursion
  - Not too many

Problem: Knapsack

You are a burglar and are in the process of robbing a home. You have found several valuable items, but the sack you brought can only hold so much weight, what is the best combination of items to steal?

Specification

You have an available list of items. Each has a (non-negative integer) weight, and value. Your sack also has a capacity.

The goal is to find the collection of items so that:

1. The total weight of all the items is less than the capacity
2. Subject to 1, the total value is as large as possible.

Variations

There are two slight variations of this problem:

1. Each item can be taken as many times as you want.
2. Each item can be taken at most once.

Subproblems (multiple copies version)

Subproblem: BestValue(Capacity’).

Recursion

What is BestValue(C)?

Possibilities:

- No items in bag
  - Value = 0
- Item i in bag
  - Value = BestValue(C-weight(i)) + value(i)

<u>Recursion:</u> BestValue(C) =

$\text{Max}(0, \text{Max}_{wt(i) \leq C} (\text{val}(i)+\text{BestValue}(C-\text{wt}(i))))$

Algorithm

```
Knapsack(Wt,Val,Cap)
    Create Array T[0…Cap]
    For C = 0 to Cap
        T[C] ← 0
        For items i with Wt(i) ≤ C
            If T[C] < Val(i)+T[C-Wt(i)]
                T[C] ← Val(i)+T[C-Wt(i)]
    Return T[Cap]
```

O(Cap) Subproblems

O(#items) time/subproblem

Runtime: O([Cap] [#Items])

Non Repeating Items

- Imagine items coming along a conveyor belt. You decide one at a time whether or add to your sac.
- Last item: either add or don’t.
  - Add: $\text{BestValue}_{\leq n-1}(\text{Cap-Wt}(n)) + \text{Val}(n)$
  - Don’t add: $\text{BestValue}_{\leq n-1}(\text{Cap})$
- We only need subproblems of the form $\text{BestValue}_{\leq k}(\text{Cap})$.

Recursion

$\text{BestValue}_{\leq k}(\text{Cap})$ = Highest total value of items with total weight at most Cap using only items from the first k.

Base Case: $\text{BestValue}_{\leq 0}(C) = 0$

Recursion: $\text{BestValue}_{\leq k}(C)$ is the maximum of

1. $\text{BestValue}_{\leq k-1}(C)$
2. $\text{BestValue}_{\leq k-1}(C - \text{Wt}(k)) + \text{Val}(k)$
   [where this is only used if $\text{Wt}(k) \leq \text{Cap}$]

Runtime

- Number of Subproblems: O([Cap] [#items])
- Time per subproblem O(1)
  - Only need to compare two options.
- Final runtime O([Cap][#items]).

Chain Matrix Multiplication

How long does it take to multiply matrices?

Recall if C = A·B then

$$C_{xz} = \Sigma A_{xy}B_{yz}.$$

Suppose A is an nxm matrix and B is mxk. Then for each entry of C (of which there are nk), we need to sum m terms.

Runtime O(nmk)*

*Can do slightly better with Strassen, but we'll ignore this for now.

More than two Matrices

Next suppose that you want to multiply three matrices ABC.

Can do it two different ways,

A(BC) OR (AB)C.

Time depends on order!

Problem Statement

Problem: Find the order to multiply matrices A_1, A_2, A_3,…,A_m that requires the fewest total operations.

In particular, assume A_1 is an n_0 x n_1 matrix, A_2 is n_1 x n_2, generally A_k is an n_{k-1} x n_k matrix.

Recursion

- We need to find a recursive formulation.
- Often we do this by considering the last step.
- For some value of k, last step: $(A_1A_2\ldots A_k)\cdot(A_{k+1}A_{k+2}\ldots A_m)$
- Number of steps:
  - CMM($A_1,A_2,\ldots,A_k$) to compute first product
  - CMM($A_{k+1},\ldots,A_m$) to compute second product
  - $n_0n_kn_m$ to do final multiply
- Recursion CMM($A_1,\ldots,A_m$) = $\min_k[\text{CMM}(A_1,\ldots,A_k)+\text{CMM}(A_{k+1},\ldots,A_m)+n_0n_kn_m]$

Subproblems

- What subproblems do we need to solve?
  - We cannot afford to solve all possible CMM problems.
- CMM(A_1,…,A_m) requires CMM(A_1,…,A_k) and CMM(A_k,…,A_m).
- These require CMM(A_i,A_{i+1},…,A_j), but nothing else.
- Only need subproblems C(i,j) = CMM(A_i,A_{i+1},…,A_j) for $1 \leq i \leq j \leq m$.
  - Fewer than m^2 total subproblems.
  - Critical: Subproblem reuse.

Full Recursion

Base Case: $C(i,i) = 0$.
(With a single matrix, we don't have to do anything)

Recursive Step:

$C(i,j) = \min_{i \leq k < j}[C(i,k)+C(k+1,j)+n_i n_k n_j]$

Solution order: Solve subproblems with smaller $j-i$ first. This ensures that the recursive calls will already be in your table.

Runtime

Number of Subproblems: One for each $1 \le i \le j \le m$. Total: $O(m^2)$.

Time per Subproblem: Need to check each $i \le k < j$. Each check takes constant time. $O(m)$.

Final Runtime: $O(m^3)$

All Pairs Shortest Paths

Problem: Given a graph G with (possibly negative) edge weights, compute the length of the shortest path between every pair of vertices.

Note: `Bellman-Ford` computes single-source shortest paths. Namely, for some fixed vertex s it computes all of the shortest paths lengths d(s,v) for every v.

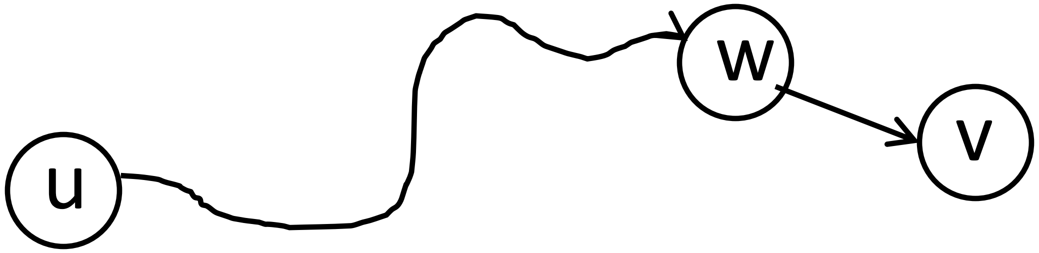
Repeated `Bellman-Ford`

Easy Algorithm: Run `Bellman-Ford` with source s for each vertex s.

Runtime: $O(|V|^2|E|)$

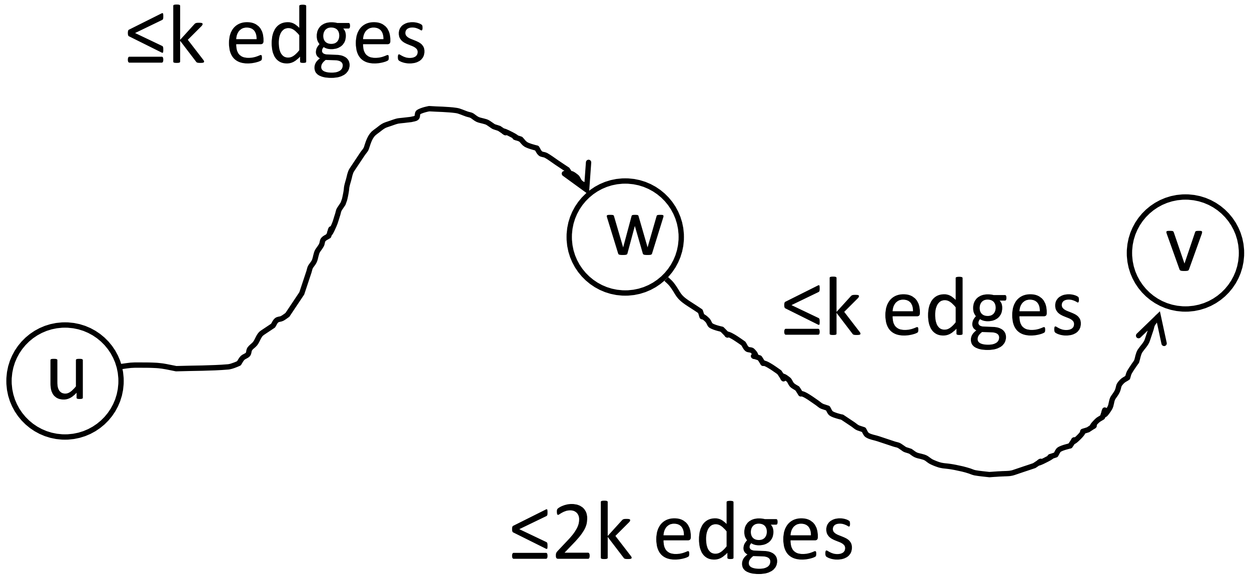
Dynamic Program

- Let $d_k(u,v)$ be the length of the shortest u-v path using at most k edges.



- Consider last edge.
- Length k-1 path from u to w, edge from w to v.
- $d_k(u,v) = \min_w[d_{k-1}(u,w)+ \ell(w,v)]$

Matrix Multiplication Method



$$d_{2k}(u,v) = \min_{w \in V}(d_k(u,w) + d_k(w,v)).$$

Algorithm

Runtime: $O(|V|^3 \log|V|)$ $O(|V|^2)$

Base Case:

$$d_1(u,v) = \begin{cases} 0 & \text{if } u = v \\ \ell(u,v) & \text{if } (u,v) \in E \\ \infty & \text{otherwise} \end{cases}$$

Recursion: Given $d_k(u,v)$ for all u, v compute $d_{2k}(u,v)$ using $d_{2k}(u,v) = \min_{w \in V}(d_k(u,w) + d_k(w,v)).$

$O(|V|^3)$

End Condition: Compute $d_1, d_2, d_4, \ldots d_m$ with $m > |V|$.

$O(\log|V|)$ iterations

Floyd-Warshall Algorithm

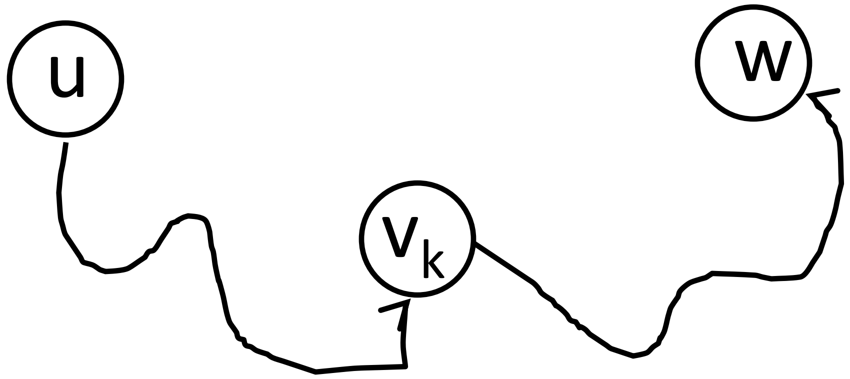
- Label vertices $v_1, v_2, \ldots, v_n$.
- Let $d_k(u,w)$ be the length of the shortest u-w path using only $v_1, v_2,\ldots,v_k$ as intermediate vertices.
- **<u>Base Case:</u>**

$$d_0(u,w) = \begin{cases} 0 & \text{if } u = w \\ \ell(u,w) & \text{if } (u,w) \in E \\ \infty & \text{otherwise} \end{cases}$$

Recursion

Break into cases based on whether shortest path uses v_k.

- The shortest path not using v_k has length $d_{k-1}(u,w)$.
- The shortest path using v_k has length $d_{k-1}(u,v_k)+d_{k-1}(v_k,w)$.



Algorithm

Runtime: O(|V|^3)

O(|V|^2)

Base Case:

$$d_0(u,w) = \begin{cases} 0 & \text{if } u = w \\ \ell(u,w) & \text{if } (u,w) \in E \\ \infty & \text{otherwise} \end{cases}$$

O(|V|^2)

Recursion: For each u, w compute:

$$d_k(u,w) = \min(d_{k-1}(u,w), d_{k-1}(u,v_k) + d_{k-1}(v_k,w)).$$

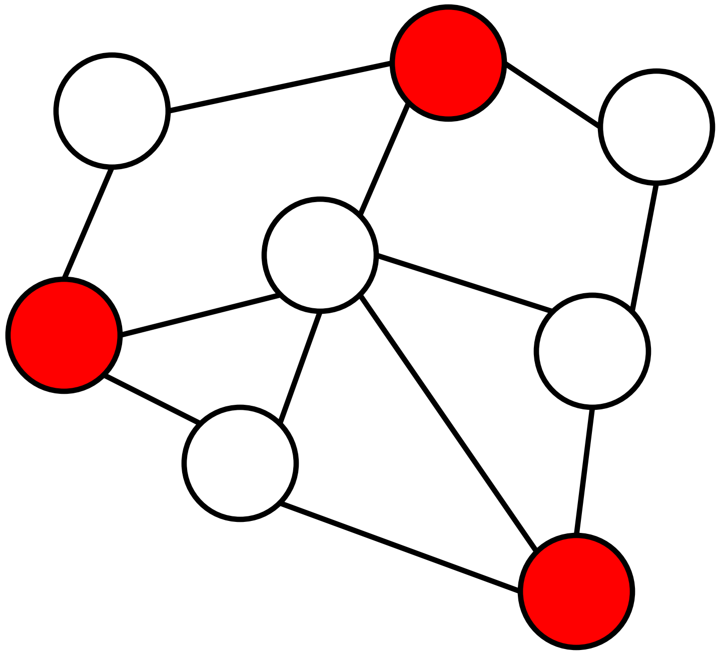
End Condition: d(u,w) = d_n(u,w) where n = |V|.

O(|V|) Iterations

Independent Set

Definition: In an undirected graph G, an independent set is a subset of the vertices of G, no two of which are connected by an edge.

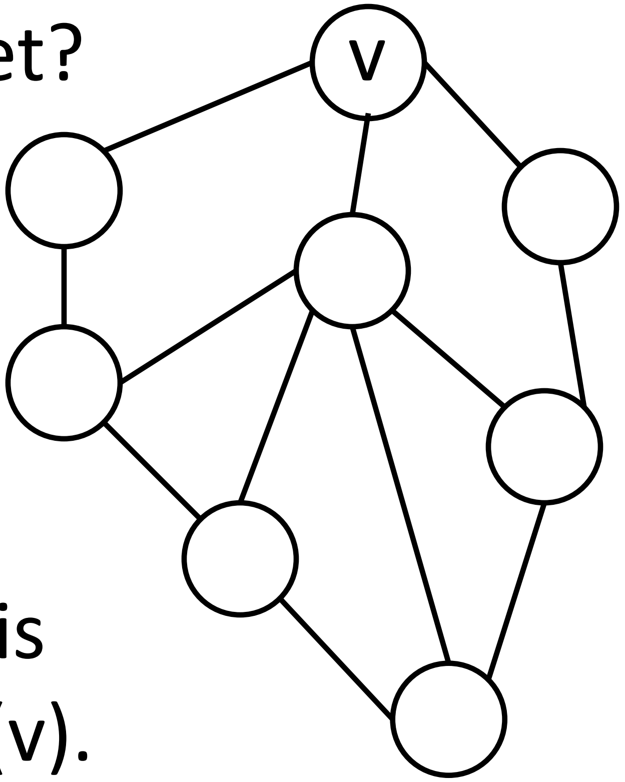
Problem: Given a graph G compute the largest possible size of an independent set of G.



Call answer I(G).

Simple Recursion

Is vertex v in the independent set?



<u>If not:</u> Maximum independent set is an independent set of G-v.
I(G) = I(G-v).

<u>If so:</u> Maximum independent set is v plus an independent set of G-N(v).
I(G) = 1+I(G-N(v)).

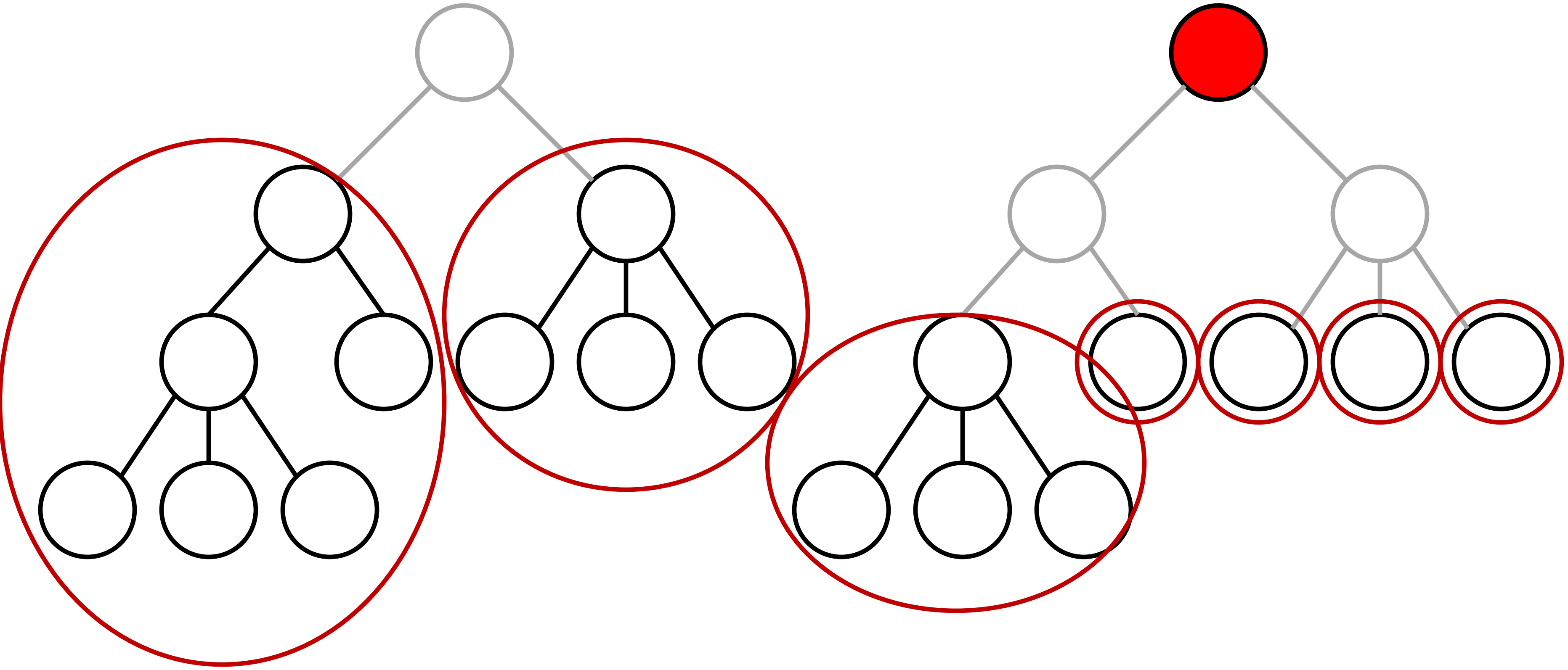
<u>Recursion:</u> I(G) = max(I(G-v), 1+I(G-N(v)))

Independent Sets and Components

<u>Lemma:</u> If G has connected components $C_1, C_2, \ldots, C_k$ then

$$I(G) = I(C_1)+I(C_2)+\ldots+I(C_k).$$

Independent Sets of Trees



Subproblems are all subtrees!

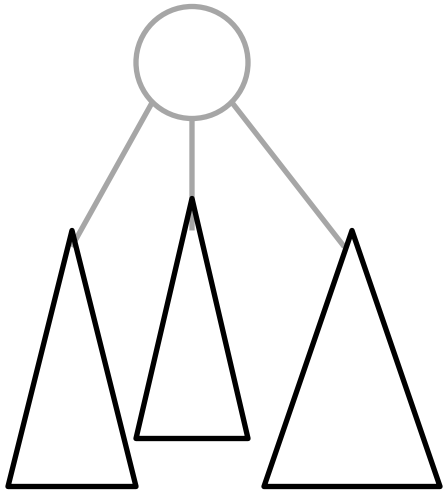
Recursion

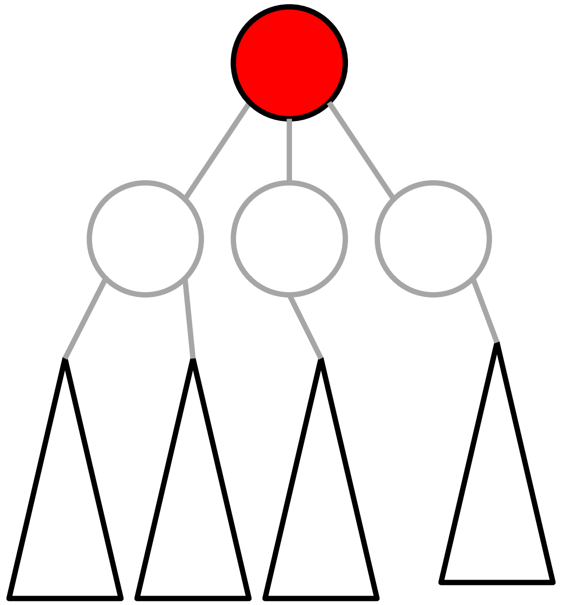
Root not used:

I(G) = Σ I(children's subtrees)

Root is used:

I(G) = 1+Σ I(grandchildren's subtrees)



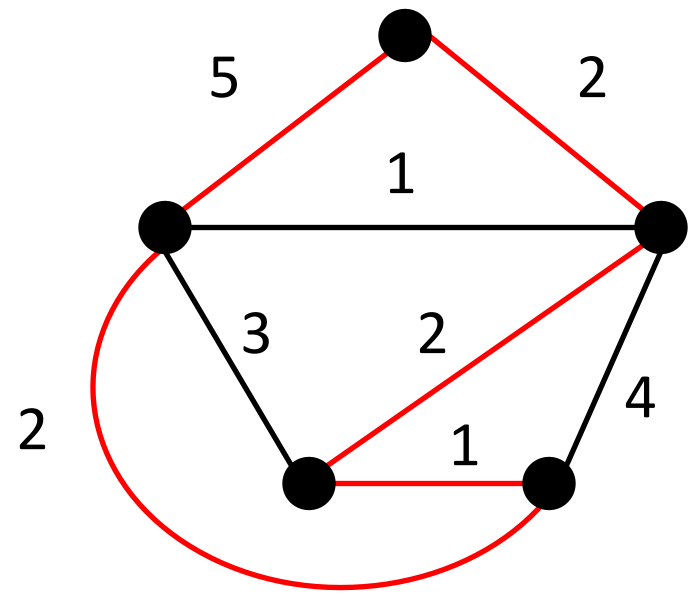


Travelling Salesman Problem

In your job as a door-to-door vacuum salesperson, you need to plan a route that takes you through n different cities. In order to space things out, you do not want to get back to the start until you have visited all cities. You also want to do so with as little travel as possible.

Formal Definition

Problem: Given a weighted (undirected) graph G with n vertices find a cycle that visits each vertex exactly once whose total weight is as small as possible.



2+1+2+2+5 = 12

Naïve Algorithm

- Try all possible paths and see which is cheapest.
- Runtime ≈ n!

Recursion

$\text{Best}_{st,L}(G)$ = Best s-t path that uses <u>exactly</u> the vertices in L.

- Last edge is some $(v,t) \in E$ for some $v \in L$.
- Cost is $\text{Best}_{sv,L-t}(G) + \ell(v,t)$.

<u>Full Recursion:</u>

$\text{Best}_{st,L}(G) = \min_v[\text{Best}_{sv,L-t}(G) + \ell(v,t)]$.

Runtime Analysis

Number of Subproblems:

L can be any subset of vertices (2^n possibilities)

s and t can be any vertices (n^2 possibilities)

$n^2 2^n$ total.

Time per Subproblem:

Need to check every v ($O(n)$ time).

Final Runtime:

$O(n^3 2^n)$

[can improve to $O(n^2 2^n)$ with a bit of thought]