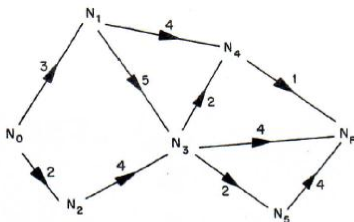
3. Paths, trees, and connectivity

- Shortest paths problem
- Cayley's theorem
- Prüfer code
- Connectivity

Shortest paths problem

- a directed graph $G = (V, E)$, and positive edge lengths $\{w_{ij}\}$
- find the shortest paths from a node $s \in V$ to all other nodes



fact 1. if a path $(v_1, v_2, \cdots, v_{k-1}, v_k)$ is the shortest path, then all subpaths (v_1, v_2), (v_1, v_2, v_3), $\cdots$ are shortest paths

fact 2. let d_i be the distance of the shortest path to node i. Then, a path P from the source to node i is a shortest path if and only if $d_j = d_k + w_{kj}$ for all (k, j) in the path

fact 3. there exists a **shortest path tree** where the unique path along the tree from the source to any node is a shortest path

Dijkstra's algorithm for finding the shortest paths

- ⋆ initialize the permanent label with the source s as $P = \{(s, 0)\}$ and temporary label $T = \{\}$ and repeat the following
- ⋆ distance update: find the minimum distance to all neighbors of the *permanent labelled set* by computing

$$d_i = \min_{j \in P} d_j + w_{j,i}$$

  and update the *temporary label set*

$$T = \{(i, d_i) \mid i \text{ in the neighborhood of } S\}$$

- ⋆ node selection: select a node in T with the smallest distance and move it to P

step		permanent label	temporary label
0		$\{(N_0, 0)\}$	$\{\}$
1	distance update	$\{(N_0, 0)\}$	$\{(N_1, 3), (N_2, 2)\}$
1	node selection	$\{(N_0, 0), (N_2, 2)\}$	$\{(N_1, 3)\}$
2	distance update	$\{(N_0, 0), (N_2, 2)\}$	$\{(N_1, 3), (N_3, 6)\}$
2	node selection	$\{(N_0, 0), (N_2, 2), (N_1, 3)\}$	$\{(N_3, 6)\}$
3	$\vdots$	$\vdots$	$\vdots$

correctness of Dijkstra's algorithm

proof by induction

- ★ let P_t be the set of nodes and corresponding distances in permanent labels at step t
- ★ assuming that the distances in P_t are shortest path distances
- ★ prove that the distance of the node selected at $t+1$ is also the shortest distance

- ★ let j be the node selected at step $t+1$, and the shortest path includes an edge (i, j) for a node i in P_t with shortest distance d_i and an edge length w_{ij}
- ★ we prove that $d_j = d_i + w_{ij}$ is the shortest path distance from the source s to node j
- ★ consider another path from the source to node j that includes the edge (k, ℓ) for a node $k \in P_t$ and a node $\ell \notin P_t$
- ★ then the distance of this path is at least $d_k + w_{k\ell}$
- ★ by construction we know that $d_i + w_{ij} \leq d_k + w_{k\ell}$, because otherwise node ℓ would have been selected at step $t+1$
- ★ this proves that $d_j = d_i + w_{ij}$ is the shortest path distance

running time of Dijkstra's algorithm

- distance update at time k: out-degree of the node selected at time k
- node selection at time k: worst-case search over $n-k$ nodes
- total complexity: $\sum_{i=1}^{n} \text{out-deg}(i) + \sum_{k=1}^{n-1}(n-k) = |E| + \frac{n(n-1)}{2}$

negative cycles

- ⋆ the arguments so far holds even if we have edges with negative weights, as long as there is no negative cycle
- ⋆ if there exists a cycle with negative weights, then the shortest path between two nodes is not well defined (we can repeat the negative cycle many times)
- ⋆ in this case, we want a robust algorithm which can detect presence of negative cycles when they exist
- ⋆ while Dijkstra algorithm does not confirm whether a negative cycle exists or not, Bellman-Ford algorithm does (shortest paths decrease after $|V|-1$ iterations if there is a negative cycle)

define $A(i,k)$ to be the length of the shortest path from the source to node i that uses at most k edges (it is ∞ is such path does not exist)

Bellman-Ford algorithm for finding shortest paths

- ⋆ for $k=1,\ldots,|V|-1$
- ⋆ for all $i \in V$ compute

$$A(i,k) = \min_{j \in N(i)} \{A(j,k-1) + w_{j,i}\} \;,$$

where it is assumed that $w_{i,i} = 0$ for all $i \in V$

running time is $|E|$ operations per step and at most $|V|-1$ steps, which gives the running time of $O(|E|\,|V|)$

example: single-duty crew scheduling

The following table illustrates a number of possible duties for the drivers of a bus company. We wish to ensure, at the lowest possible cost, that at least one driver is on duty for each hour of the planning period (9 am to 5 pm). Formulate and solve this scheduling problem as a shortest path problem.

Duty hours	9-1	9-2	12-3	12-5	2-5	1-4	4-5
Cost	30	18	21	38	20	22	9

Review: Minimum Spanning Tree

Q. Suppose we wanted to find a minimum spanning tree T for a connected graph G with positive edge weights. Which of the following algorithms will always produce such a tree? Justify your answers.

(a) For each vertex v, use Dijkstra's algorithm to find the shortest paths from v to all other vertices, and store any edge used in any of these paths in a set T_v. Pick T to be any T_v with minimum weight.

(b) Sort the edges of G in decreasing order of weight. Initially, set $T = G$. Then, iterate along this list of edges, removing an edge from T if and only if doing this does not disconnect T.

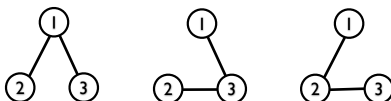
(c) Pick any cycle in G, and remove an edge of maximum cost contained in this cycle. Repeat until no cycles are left. The resulting graph is T.

- **cycle property** of a MST: in any cycle C in the graph, if an edge has larger weight than any of the other edges in C, then this edge cannot belong to an MST.

Counting labeled trees

consider a fixed **labeled** set of vertices $V = \{1, \cdots, n\}$

two (undirected) trees $T_1(V, E_1)$ and $T_2(V, E_2)$ are different if $E_1 \neq E_2$



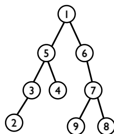
Cayley's theorem provides the number of distinct trees on V, and is one of the most beautiful and elegant results in combinatorics

Cayley's Theorem. The number of labeled trees with n vertices is n^{n-2}

to prove Caley's theorem, we define a one-to-one mapping between labeled trees and **Prüfer codes** which are strings of length $n-2$

to build a Prüfer code of a tree $T(V,E)$ we use the following iterative procedure

- ⋆ input: tree T
- ⋆ output: Prüfer code $P = a_1 a_2 \cdots a_{n-2}$
- ⋆ for $t = 1, 2, \ldots, n-1$ do
- ⋆ Let vertex i be the leaf with smallest label in T at step t, and vertex j is its parent
- ⋆ Remove vertex i and edge (i,j) from T
- ⋆ Let $a_t = j$, $b_t = i$
- ⋆ end for
- ⋆ return $P = a_1 a_2 \ldots a_{n-2}$



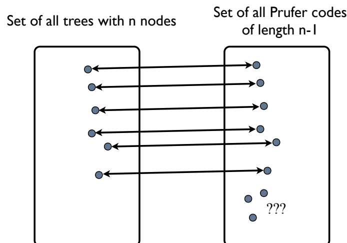
$b = 2, 3, 4, 5, 1, 6, 8, 7$
$a = 3, 5, 5, 1, 6, 7, 7, 9$

- **proposition.** a_{n-1} is always n.
- **proof.** A tree always has at least two leaves. Hence, n will never be the smallest label in the leaves, until the last iteration when we have a single node n.

- **proposition.** $a_i, \ldots, a_{n-2}$ are the set of non-leaves at step $i-1$.
- **proof.** Each of $a_i, \ldots, a_{n-2}$ is a parent at some point in the future.

- **proposition.** Given $P = a_1, \ldots, a_{n-2}$ and $a_{n-1} = n$, one can always reconstruct $b = b_2 \ldots b_{n-2}$.
- **proof by construction.** We start reconstruction from b_1 and use it to reconstruct b_2, and so on. First, notice that b_1 is the smallest labeled leaf. Although we do not know which labels are on the leaves, we do know that b_1 never appears in $a_1, \ldots, a_{n-1}$ since b_1 is immeiately removed from the tree at first iteration. Hence, b_1 is the smallest label that does not appear in $a_1, \ldots, a_{n-1}$.
  Similarly, b_i does not appear in $a_i, \ldots, a_{n-1}$, since b_i is removed at step i. Further, it does not appear in $b_1 \ldots, b_{i-1}$, since those labels are already removed in previous steps. Hence, b_i is the smallest number in $\{b_1, \ldots, b_{i-1}\}^c \cap \{a_i, \ldots, a_{n-2}\}^c$.

- **proposition.** Every tree is mapped to a unique Prüfer code.
- **proof by contradiction.** If every tree is not mapped to a unique tree, then there exists two trees T_1 and T_2 that are mapped to the same code P. However, we know that we can reconstruct the original tree, if we have P. This is a contradiction, since the reconstruction can only be one of T_1 or T_2.



- **proposition.** There are n^{n-2} Prüfer codes of length $n-2$.
- **proof.** A Prüfer code is a sequence of numbers $a_1, \ldots, a_{n-2}$. Each a_i can take any values in $\{1, \ldots, n\}$.

- **proposition.** Every n^{n-2} Prüfer code P leads to a tree.
- **proof.**
  1. Given any $P = a_1, \ldots, a_{n-2}$, we can find $b_1, \ldots, b_{n-1}$ using the reconstruction rule.
  2. a and b define a graph where we start from a single node n and add node b_{n-1} to a_{n-1}, and so on.
  3. the resulting graph is a tree since we added one node and one edge connecting this node to the tree from previous step.
  4. Hence, every code P maps to a tree.

this prove the following:
Cayley's Theorem. The number of labeled trees with n vertices is n^{n-2}

other applications of Prüfer code

- ⋆ what does the number of appearance of a label in $a_1, \ldots, a_{n-1}$ mean?
- ⋆ node i with degree d_i appears exactly $(d_i - 1)$ times

- ⋆ given a degree sequence $d_1, d_2, \ldots, d_{n-1}, d_n$ such that $\sum_{i=1}^{n} d_i = 2(n-1)$, what is the number of trees such that degree of node i is exactly d_i?

$$\binom{n-2}{d_1 - 1, \ldots, d_n - 1} = \frac{(n-2)!}{(d_1 - 1)! \cdots (d_n - 1)!}$$

- ⋆ one can generate uniformly random Prüfer code and convert it to a tree to generate a labelled tree uniformly at random

Graph properties

- connectivity

  the **vertex cut** of an undirected graph G for two distinct nodes i and j is the minimum number of nodes whose removal disconnects i and j

  the **edge cut** of an undirected graph G for two distinct nodes i and j is the minimum number of edges whose removal disconnects i from j

  - the **vertex connectivity** of an undirected graph G is the size of the minimal vertex cut in G

    (**Menger's theorem**) the size of the minimum vertex cut in an undirected graph G between two distinct nodes i and j is equal to the maximum number of pairwise vertex-independent paths from i to j

  - the **edge connectivity** of an undirected graph G is the size of the minimal edge cut in G

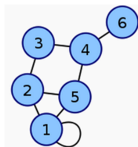
    (**Menger's theorem**) the size of the minimum edge cut in an undirected graph G between two distinct nodes i and j is equal to the maximum number of pairwise edge-independent paths from i to j (proof is left as a homework)

$$\text{vertex connectivity} \leq \text{edge connectivity} \leq \text{minimum degree}$$

- adjacency matrix

  Define an adjacency matrix $A \in \mathbb{R}^{|V| \times |V|}$ of an undirected graph $G = (V, E)$

$$A_{ij} = \begin{cases} 1 & (i,j) \in E \\ 0 & (i,j) \notin E \end{cases}$$



$$\begin{pmatrix} 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

  properties of the adjacency matrix

  ★ let $d = A\mathbb{1}$, where $\mathbb{1} = [1, 1, \ldots, 1]$ is the all-ones vector then d_i is the degree of node i

  ★ $\mathbb{1}^T A \mathbb{1} = \sum_{i,j} A_{ij} = 2|E|$, since we are counting each edge twice

  ★ from the above two equations, we get that

$$\sum_i d_i = 2|E|$$

  let $B = A^k$, what does B_{ij} mean?

incidence matrix

Define an all-vertex incidence matrix $A \in \mathbb{R}^{|V| \times |E|}$ of an undirected graph $G = (V, E)$

$$A_{ij} = \begin{cases} 1 & \text{if } i \text{ is one end of the edge } e_j \\ 0 & \text{otherwise} \end{cases}$$

Define an all-vertex incidence matrix $A \in \mathbb{R}^{|V| \times |E|}$ of a directed graph $G = (V, E)$

$$A_{ij} = \begin{cases} 1 & \text{if } i \text{ is the initial node of the edge } e_j \\ -1 & i \text{ is the final node of the edge } e_j \\ 0 & \text{otherwise} \end{cases}$$

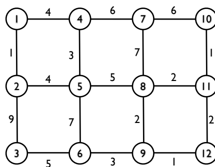
- ⋆ since every column of an all-vertex incidence matrix contains exactly two non-zero entries, we can remove a row from the matrix and still have sufficient information to define the graph
- ⋆ the **incidence matrix** of a graph is defines by removing a row from the all-vertex incidence matrix (notice it is not unique)

Homework 3

Problem 1. (MATLAB exercise)

- We want to write a MATLAB code for computing the minimum spanning tree given a weights adjacency matrix of an undirected graph. We will implement Kruskal's algorithm and verify it on the following graph. Download two files `mst.m` and `existscycle.m` from course website. `exists.m` contains a function that takes as input an adjacency matrix and outputs 1 if there is a cycle and 0 if there is no cycle. `mst.m` is the main algorithm, with missing parts that you need to fill in. In the code provided, **A0** is the input weighted adjacency matrix. **MST** is the adjacency matrix of the minimum spanning tree that we recursively grow using Kruskal's algorithm. **A** is the residual graph that starts as **A0** and erases any edge that has been included in **MST** or any edge that creates cycles when augmented to current **MST**. At each recursion, we compute a candidate edge **candidate** by choosing the minimum weight edge from **A**. Then, we want to created an adjacency matrix **augmentT** that adds **candidate** to current **MST**, and check if there is a cycle in the augmented tree. If there is a cycle, we erase **candidate** from A, since we will never use it in our MST. If there is no cycle, we add **candidate** to **MST** and remove **candidate** from A. We continue until we have found $n-1$ edges in our MST.

Homework 3 problem 1 continued



(a) Fill in the missing parts in `mst.m`.
(b) Output the edges in MST and also the weights, for example

$$(1,2), 1$$
$$(1,4), 4$$
$$\vdots$$

You do not need to sort the edges in any particular order. Print out your source code and your output of part (b) and submit as a solution to this problem.

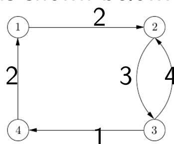
Homework 3

Problem 2.

We consider a directed graph with n nodes. The graph is specified by its **weighted adjacency matrix** $A \in \mathbb{R}^{n \times n}$, defined as

$$A_{ij} = \begin{cases} w_{ij} & \text{if there is an edge from node } i \text{ to node } j \\ 0 & \text{otherwise.} \end{cases}$$

Note that the edges are *oriented*, i.e., $A_{34} = w_{34}$ means that there is an edge from node 3 to node 4 with weight w_{34}. For simplicity we do not allow self-loops, i.e., $A_{ii} = 0$, for all $i \in \{1, \ldots, n\}$. A simple illustration of this notation is shown below:



The weighted adjacency matrix for this example is

$$A = \begin{bmatrix} 0 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 4 & 0 & 1 \\ 2 & 0 & 0 & 0 \end{bmatrix}$$

Homework 3 problem 2 continued

The rest of this problem concerns a specific graph, given in the file `directedgraph.m` on the course web site. For each of the following questions, you must give the answer explicitly. you must also explain clearly how you arrived at your answer.

We provide an algorithm for computing the shortest path from a specified source s to all other nodes in the graph in the file `shortest.m`. To answer the following questions, use this algorithm as a starting point and modify it as necessary.

(*a*) What is the length of a shortest path from node 1 to node 18?

(*b*) What is the length of a shortest path from node 1 to node 18, that **does** pass through node 19? (you can travel the same edge twice if you need to)

(*c*) What is the length of a shortest path from node 1 to node 18, that **does not** pass through node 9?

(*d*) What is the length of a shortest path from node 1 to node 18, that **does** pass through edge (5,6)?

Print out your source code and your answers to the above questions as a solution to this problem.

Homework 3

Problem 3.

A complete undirected graph is a graph where every node is adjacent to all the other nodes. Given a labelled complete graph $G = (V, E)$, Cayley's theorem provides the number of spanning trees on this complete graph with n nodes, which is n^{n-2}.

Consider a labelled complete bipartite graph $G = (A, B, E)$ where a node in A is adjacent to all the nodes in B but no nodes in A. Also, a node in B is adjacent to all nodes in A but no nodes in B. We label $A = \{1, \ldots, |A|\}$ and $B = \{|A| + 1, \ldots, |A| + |B|\}$.

In this problem, we want to count the number of spanning trees on this labelled complete bipartite graph G by counting the number of allowed Prüfer codes.

Notice that not all Prüfer codes are allowed in this case, since some Prüser codes generate trees that are not allowed, i.e. violate the bipartite graph assumption by connecting two nodes in A or two nodes in B. We claim that there is a one-to-one correspondence between a valid bipartite tree and the following bipartite Prüfer code.

Homework 3

Problem 3. (continued)

to build a bipartite Prüfer code (a, c) of a tree $T(A, B, E)$ we use the following iterative procedure to generate two sequences a and c of lengths $|A| - 1$ and $|B| - 1$ respectively

- ★ input: bipartite tree T
- ★ output: bipartite Prüfer code $P = (a_1, a_2, \cdots, a_{|A|-1}), (c_1, \cdots, c_{|B|-1})$
- ★ for $t = 1, 2, \ldots, n-1$ do
- ★ Let vertex i be the leaf with smallest label in T at step t, and vertex j is its parent
- ★ Remove vertex i and edge (i, j) from T
- ★ if $i \in A$ then set $a_t = j$ and $b_t = i$
- ★ else set $c_t = j$ and $d_t = i$
- ★ end for
- ★ return $P = (a_1, a_2, \cdots, a_{|A|-1}), (c_1, c_2, \cdots, c_{|B|-1})$

(a) Show that $a_{|A|} = n$.

(b) Explain how to recover the sequence in $b = (b_1, \cdots, b_{|A|})$ and $d = (d_1, \cdots, d_{|B|-1})$ from P.

(c) Show that the reconstructed graph is always a tree satisfying the bipartite condition.

(d) Show that the total number of labelled trees on a complete bipartite graph is $|A|^{|B|-1}|B|^{|A|-1}$.

Homework 3

Problem 4.

We define the distance between two nodes in a graph $G = (V, E)$ as the number of edges in the shortest path between those two vertices. A vertex in G is central if its greatest distance from any other vertex is as small as possible. This distance is called **radius**. The **diameter** of a graph is defined as the greatest distance between two nodes in the graph, that is

$$\begin{aligned} \text{radius}(G) &\triangleq \min_{i \in V} \max_{j \in V} d(i, j) \\ \text{diameter}(G) &\triangleq \max_{i,j \in V} d(i, j) \end{aligned}$$

(a) Prove that for every graph G

$$\text{radius}(G) \leq \text{diameter}(G) \leq 2 \cdot \text{radius}(G)$$

(b) Prove that a graph G of radius at most k and maximum degree at most d where d is an integer greater than two, has fewer than $\frac{d}{d-2}(d-1)^k$ vertices.

Homework 3

Problem 5. (Menger's theorem)

- An amateur graph theorist, in his scribblings, might invent the following two definitions of k-edge connectivity. a directed graph G is k-edge connected if
  - (i) G remains connected after removing any $(k-1)$ edges.
    OR
  - (ii) There are at least k edge-disjoint paths between every pair of nodes in G.

(a) Recall that a directed graph G is connected if and only if for each pair of nodes i and j there exists a directed path from node i to node j. And two paths are edge-disjoint if they do not share any edges. It is clear that if G satisfy definition (ii) then it also satisfies definition (i). Prove that if G remains connected after removing any $(k-1)$ edges then there are at least k edge-disjoint paths between every pair of nodes in G.

(b) Edge conenctivity $\lambda(G)$ of an undirected graph $G = (V, E)$ is defined as the minimum k such that the graph is k-edge conected. Give a polynomial-time algorithm for computing $\lambda(G)$.