Lecture 14

- Connectedness in graphs
- Spanning trees in graphs
- Finding a minimal spanning tree
- Time costs of graph problems and NP-completeness
- Finding a minimal spanning tree: Prim’s and Kruskal’s algorithms
- Intro to disjoint subsets and union/find

Reading: Weiss, Ch. 9, Ch 8
Connectedness of graphs

• Some definitions:

• An undirected graph is *connected* if
  • For every vertex \( v \) in the graph, there is a path from \( v \) to every other vertex

• A directed graph is *strongly connected* if
  • For every vertex \( v \) in the graph, there is a path from \( v \) to every other vertex

• A directed graph is *weakly connected* if
  • The graph is not strongly connected, but the underlying undirected graph (i.e., considering all edges as undirected) is connected

• A graph is *completely connected* if for every pair of distinct vertices \( v_1, v_2 \), there is an edge from \( v_1 \) to \( v_2 \)
Connected graphs: an example

• Consider this undirected graph:

- Is it connected?
- Is it completely connected?
Strongly/weakly connected graphs: an example

- Consider this directed graph:

  ![Graph Diagram]

- Is it strongly connected?
- Is it weakly connected?
- Is it completely connected?
Spanning trees

- We will consider spanning trees for undirected graphs
- A spanning tree of an undirected graph G is an undirected graph that...
  - contains all the vertices of G
  - contains only edges of G
  - has no cycles
  - is connected

- So, only connected graphs have spanning trees

- A spanning tree is called “spanning” because it connects all the graph’s vertices

- A spanning tree is called a “tree” because it has no cycles (recall the definition of cycle for undirected graphs)

- What is the root of the spanning tree?
  - you could pick any vertex as the root; the vertices adjacent to that one are then the children of the root; etc.
Spanning trees: examples

- Consider this undirected graph G:
Spanning tree? Ex. 1

• Is this graph a spanning tree of G?
Spanning tree? Ex. 2

- Is this graph a spanning tree of G?
Spanning tree? Ex. 3

- Is this graph a spanning tree of $G$?
Spanning tree? Ex. 4

- Is this graph a spanning tree of G?
Multiple spanning trees

- In general a graph can have more than one spanning tree. All these are spanning trees of that graph $G$ (and there are more):

- Note: The spanning tree for a graph with $N$ vertices always has $N-1$ edges (like a tree!)
Finding a spanning tree in an unweighted graph

• A spanning tree in an unweighted graph is easy to construct...

• Use the basic unweighted single-source shortest-path algorithm (breadth-first search):
  • (That algorithm is designed for directed graphs. Convert an undirected graph to a directed one by treating each undirected edge as two parallel directed edges)

• Pick any vertex as the start vertex $s$. (Think of it as the root of the spanning tree.)

• When done, the “prev” indices in the table will give, for each vertex in the spanning tree, the index of its parent

• This represents a spanning tree because (if the graph is connected) each vertex except the start vertex will have exactly one parent, and each vertex appears in the table

• So, a spanning tree can be found in an unweighted graph in time $O(|V| + |E|)$

• ...What about weighted graphs?
Minimum spanning trees in a weighted graph

• A single graph can have many different spanning trees

• They all must have the same number of edges, but if it is a weighted graph, they may differ in the total weight of their edges

• Of all spanning trees in a weighted graph, one with the least total weight is a minimum spanning tree (MST)

• It can be useful to find a minimum spanning tree for a graph: this is the least-cost version of the graph that is still connected, i.e. that has a path between every pair of vertices

• How to do it?
Finding a minimum spanning tree: Prim’s algorithm

• As you know, minimum weight paths from a start vertex can be found using Djikstra’s algorithm

• At each stage, Djikstra’s algorithm extends the best path from the start vertex (priority queue ordered by total path cost) by adding edges to it

• To build a minimum spanning tree, you can modify Djikstra’s algorithm slightly to get Prim’s algorithm

• At each stage, Prim’s algorithm adds the edge that has the least cost from any vertex in the spanning tree being built so far (priority queue ordered by single edge cost)

• Like Djikstra’s algorithm, Prim’s algorithm has worst-case time cost $O(|E| \log |V|)$

• We will look at another algorithm: Kruskal’s algorithm, which also is a simple greedy algorithm

• Kruskal’s has the same big-O worst case time cost as Prim’s, but in practice it can be made to run faster than Prim’s, if efficient supporting data structures are used
A note about graph algorithm time costs

- So far we have mentioned these graph problems:
  - Find shortest path in unweighted graphs
    - Solved by basic breadth-first search: $O(|V|+|E|)$ worst case
  - Find shortest path in weighted graphs
    - Solved by Dijkstra’s algorithm: $O(|E| \log |V|)$ worst case
  - Find minimum-cost spanning tree in weighted graphs
    - Solved by Prim’s or Kruskal’s algorithm: $O(|E| \log |V|)$ worst case

- The “greedy” algorithms used for solving these problems have polynomial time cost functions in the worst case
  - since $|E| \leq |V|^2$, Dijkstra’s, Prim’s and Kruskal’s algorithms are $O(|V|^3)$

- As a result, these problems can be solved in a reasonable amount of time, even for large graphs; they are considered to be ‘tractable’ problems

- However, many graph problems do not have any known polynomial time solutions...!
Intractable graph problems

• For many interesting graph problems, the best known algorithms to solve them have exponential time costs $O(2^{|V|})$

• In the worst case, these intractable problems simply cannot be solved exactly, except for quite small graphs (say, 50 or at most 100 vertices, even on the world’s fastest computers); the best known algorithms for these problems take too long to run

• For these problems, simple greedy best-first algorithms do not work... Essentially the best approach known for solving them exactly is basically to try all the possibilities, and there can be exponentially many possibilities to try

• These intractable graph problems are often members of the class called “NP-complete” problems, which includes many non-graph problems as well...
**NP-complete problems**

- A problem that can be exactly *solved* in time that is a polynomial function of the size of the problem is in the class “P” (for *Polynomial* time)

- A problem whose solution can be *checked for correctness* in time that is a polynomial function of the size of the problem is in the class “NP” (for *Nondeterministic Polynomial* time)
  - A “nondeterministic” computer could guess the solution to the problem, and then check if it is a solution in polynomial time, and never give a wrong answer
  - Note that the class P is contained in NP

- A problem that is in NP, and is as hard as any problem in NP (an algorithm for it is also essentially an algorithm for any NP problem) is *NP-complete*

- For all the NP-complete problems, the best known algorithms take exponential time in the worst case...
  - ...However, nobody has yet been able to *prove* that there are no polynomial time algorithms for them! If you find one, you will be instantly very very famous

- What are some of the NP-complete graph problems?...
Examples of intractable graph problems

• Here are a few examples of the many graph problems that are NP-complete, and so seem to require $O(2^{|V|})$ time worst-case:

  • “Hamiltonian circuit”: Given a graph, say whether the graph has a cycle that includes all the vertices of the graph exactly once.

  • “Travelling salesman”: Given a weighted graph, find the Hamiltonian circuit that has the smallest total cost.

  • “Longest path”: Given a graph and two vertices $s$ and $d$, find the longest path from $s$ to $d$ that doesn’t contain any cycles. (But note that “shortest path” is solvable in polynomial time!)

  • “Shortest total path length spanning tree”: Given a graph, find the spanning tree that has the smallest total path lengths between every pair of vertices

  • “Steiner tree”: Given a graph $(V,E)$ and a subset $S$ of $V$, find the minimum-cost spanning tree that spans every vertex in $S$ (and may also span some other vertices) (but note that if $S=V$, the problem is solvable in polynomial time!)
The problem with intractable problems

• If a problem is NP-complete, the best known algorithms to solve it require exponentially many steps in the worst case

• Simple greedy algorithms do not work for these problems
  • backtracking, or some other way of looking at, and checking, possible alternatives is usually required...
  • ... and there are exponentially many alternatives to check!
  • For example:
    • The problem has $N$ boolean variables, and you need to check the $2^N$ possible different assignments of truth values to them
    • The problem has $N$ items, and you need to check each of the $2^N$ different subsets of those items

• Because of the exponential time costs of the best known solutions to these problems, you have to either...
  • restrict yourself to small instances of the problems, or
  • try to find approximate algorithms that are fast, but not always exactly correct
Finding a minimum spanning tree: Kruskal’s algorithm

- Prim’s algorithm starts with a single vertex, and grows it by adding edges until the MST is built: it builds the MST ‘top-down’

- Kruskal’s algorithm starts with a forest of single-node trees (one for each vertex in the graph) and joins them together by adding edges until the MST is built; it builds the MST ‘bottom-up’
Kruskal’s algorithm

- Pseudocode for Kruskal’s MST algorithm, on a weighted undirected graph $G = (V, E)$:
  1. Create a forest of one-node trees, one for each vertex $v$ in $V$
  2. Create a priority queue containing all the edges in $E$, ordered by edge weight
  3. While fewer than $|V|-1$ edges have been added to the forest:
     1a. Delete the smallest-weight edge, $(v_i, v_j)$, from the priority queue.
     1b. If $v_i$ and $v_j$ already belong to the same tree in the forest, go to 3a. (Adding this edge would create a cycle.)
     1c. Else, $v_i$ and $v_j$ are in different trees. Join those vertices with that edge (this joins their trees, reducing the number of trees in the forest by 1), and continue.

- When run on a connected graph, the forest will finally contain one tree, which is a minimum spanning tree
Kruskal’s algorithm: input

- Run Kruskal’s algorithm on this weighted undirected graph:
Kruskal’s algorithm: output

- Show the result here:

- What is the total cost of this spanning tree?

- Is there another spanning tree with lower cost? With equal cost?
Implementing Kruskal’s algorithm

- To make Kruskal’s algorithm efficient, all steps in the algorithm must be implemented efficiently:
  - initializing the priority queue must be efficient (2)
  - delete-min in the priority queue must be efficient (3a)
  - testing whether two vertices are already in the same tree in the forest must be efficient (3b)
  - joining two trees in the forest must be efficient (3c)

- Using a binary heap to implement a priority queue leads to efficient steps (2) and (3a):
  - building the heap: \(O(|E|)\)
  - each delete-min operation: \(O(\log|E|)\)

- What is an efficient way to do steps 3b and 3c?
  - This requires looking at doing efficient computations with representations of equivalence classes...
Equivalence relations

- An equivalence relation \( E(x,y) \) over a domain \( S \) is a boolean function that satisfies these properties for every \( x,y,z \) in \( S \):
  - \( E(x,x) \) is true (reflexivity)
  - If \( E(x,y) \) is true, then \( E(y,x) \) is true (symmetry)
  - If \( E(x,y) \) and \( E(y,z) \) are true, then \( E(x,z) \) is true (transitivity)

- For example: Given an undirected graph \( G \). Suppose for any vertices \( v_1, v_2 \) in \( G \), \( E(v_1, v_2) \) is true if and only if \( v_1 \) is connected to \( v_2 \) (i.e., there is a path from \( v_1 \) to \( v_2 \)). Then \( E() \) is an equivalence relation over the vertices of \( G \):
  - Every vertex is connected to itself (reflexivity)
  - If \( v_1 \) is connected to \( v_2 \), then \( v_2 \) is connected to \( v_1 \) (it’s an undirected graph)
  - If \( v_1 \) is connected to \( v_2 \), and \( v_2 \) is connected to \( v_3 \), then \( v_1 \) is connected to \( v_3 \)

- For another example: Suppose \( E(x,y) \) is true if and only if \( x, y \) are integers and \( x=y \). Then \( E() \) is an equivalence relation over the integers
Equivalence classes

• An equivalence relation E() over a set S defines a system of equivalence classes within S
  • The equivalence class of some element x of S is that set of all y in S such that E(x,y) is true
  • Note that every equivalence class defined this way is a subset of S
  • The equivalence classes are disjoint subsets: no element of S is in two different equivalence classes
  • The equivalence classes are exhaustive: every element of S is in some equivalence class

• For example: Given an undirected graph G. Suppose for any vertices v₁, v₂ in G, E(v₁, v₂) is true if and only if v₁ is connected to v₂. Then the equivalence classes defined by E() are the connected components of G

• For another example: Suppose E(x,y) is true if and only if x, y are integers and x=y. Then the equivalence classes defined by E() are all singleton sets: each integer is its own equivalence class
Computing with equivalence classes

• A common problem is this:

• You are given a set $S$ of items

• You are given some pairs of items in $S$ that satisfy some equivalence relation $E()$

• Given that information, you want to do things like
  • Determine how many equivalence classes there are in $S$, as defined by the pairs of items satisfying $E()$ that you have seen so far
  • Given an item in $S$, determine which equivalence class is it in
  • Given two items in $S$, determine whether they are in the same equivalence class
  • Given a new pair of items in $S$ that satisfy $E()$, update the system of equivalence classes appropriately
Computing equivalence classes: an example

- You are manipulating equivalence classes when building a minimum spanning tree using Kruskal’s algorithm
  - The initial set of items is the set of vertices in the graph
  - Initially, each vertex is its own equivalence class of one
  - The algorithm considers edges in order of increasing cost. For each edge:
    - It is accepted if the vertices it connects are not in the same equivalence class
    - If it is accepted, the equivalence classes containing the vertices it connects are
      joined into one equivalence class
  - When $|V|-1$ edges have been accepted, the algorithm terminates
  - The edges that have been accepted are the edges of the minimum spanning tree
  - These equivalence-class computations can be done very efficiently using a “disjoint subset”, “union/find” structure. More next time...
Next time

- An application of disjoint subsets
- Disjoint subset structures and union/find algorithms
- Union-by-size and union-by-height
- Find with path compression
- Amortized cost analysis

Reading: Weiss, Ch. 8