SI 335, Unit 5: Graph Search

Daniel S. Roche (roche@usna.edu)

Spring 2015

1 Graphs

> NOTE: All this section should be review from Data Structures, IC 312. Feel free to skip to the next section of the notes if you feel pretty solid on this.

Graphs (sometimes called "networks") are an extremely useful and versatile way of representing information. At its core, a graph is a collection of items (the nodes) and relationships between them (edges). Here is a very small sampling of different kinds of information that can be represented with a graph:

- Cities in a country or continent and the highways that connect them. Can be used by services like Google Maps to tell you the fastest way to get somewhere.
- People in an organization and the relationships between them
- ("boss", "acquaintance", "friend", etc.).
- Computers and servers in a network and the connections between them. Problems we might ask are how to get a message from one machine to another and how to best configure the network for redundancy if one machine goes down.
- Web pages and the hyperlinks between them
- Dependencies in a Makefile
- Scheduling tasks given some constraints (like "A has to happen after B" or "A and B can't happen simultaneously".)

Because of this, graphs rightfully belong to the class of basic computational structures alongside other things like numbers, lists, and collections. We'll examine a few of the most important algorithms and problems with graphs but there are many, many more that arise in all kinds of varied circumstances.

1.1 Definitions and Terminology

A graph $G = (V, E)$ is a tuple of "nodes" or "vertices" in the set V, and "edges" in the set E that connect the vertices. Each edge goes between two vertices, so we will write it as an ordered pair (u, v) for $u, v \in V$. If G is a weighted graph, then there is also a "weight function" ω such that $\omega(u, v)$ is some non-negative number (possibly infinity) for every edge $(u, v) \in E$.

For convenience we will always write $n = |V|$ for the number of nodes and $m = |E|$ for the number of edges in a graph. You can see that $m \le n^2$, and if the graph doesn't trivially separate into disconnected pieces, then $m \ge n - 1$ as well. We say that a graph is "sparse" if m is close to n, and a graph is "dense" if m is closer to n^2. (Yes, this is a bit fuzzy!)

The most general kind of graph we will examine is a *directed, weighted, simple* graph. The "directed" part means that the edges have arrows on them, indicating the direction of the edge. And a "simple" graph is one that doesn't allow repeated edges (in the same direction, with the same endpoints) or loops (edges from a node back onto itself).

Time for an example: Human migration (immigration) between countries. Each vertex in the graph is a location (country, region, state) and an edge represents people moving from one location to another. The weight on each

edge is the number of people that made that move in some given time frame. Rather than me try and make a pretty picture here, why don't you look here, here, or here — look at page 11.

From this general picture we can have simplifications. An *unweighted graph* is one without weights on the edges, which we can represent as a weighted graph where every edge has weight 1. Unweighted graphs are common when the relationships represented by edges constitute "yes/no" questions. For example, in a web graph the nodes might be web sites and the edges mean "this page links to that page".

There are also *undirected graphs*, which we can represent as a directed graph in which every edge goes in both directions. That is, $(u, v) \in E \Leftrightarrow (v, u) \in E$. Undirected graphs might indicate that there is no hierarchical difference between the endpoints of an edge. For example, a computer network graph with edges representing a point-to-point connection with a certain bandwidth might be undirected.

Of course some graphs are also undirected and unweighted. For example, we could have a graph where each vertex is a Midshipman and an edge between two Mids means they are in a class together.

This might also be a good time to mention an important detail: weighted graphs are often *complete*. A complete graph is one in which every possible edge exists. For example, in the migration flow graph, we can always draw an edge from one country to another, even if the weight on the edge might be 0 (meaning no migration in that direction).

1.2 Representations

How can we store a graph in a computer? We need to represent the vertices V and the edges E. Representing the vertices is easy: we just assign them numbers from 0 up to $n - 1$ and use those numbers to refer to the vertices. Perhaps the labels (example: country name) are stored in an array by vertex index.

The tricky part is how to store the edges. We will at various points consider three fundamental ways of representing edges:

- **Adjacency Matrix**. an $n \times n$ array stores at location A[i][j] the weight of the edge from vertex i to vertex j. Since we have to specify a weight for every single edge, we usually choose some default value like 0 or infinity for the weight of an edge that isn't actually in the graph. The size of this representation is always $\Theta(n^2)$. Generally works well for dense graphs.
- **Adjacency List**. For each vertex u, we store a list of all the *outgoing edges* from u in the graph (that is, all edges that start at u). These edges are represented by a pair (v, x), where (u, v) is an edge in the graph with weight x. So the overall representation is an array of lists of pairs. One list for each vertex, one pair for each edge. In an unweighted graph, we can just store the index of the successor vertices v rather than actual pairs. The size of this representation is always $\Theta(n + m)$. Generally works a bit better for sparse graphs because this representation uses less space when m is much smaller than n^2.
- **Implicit**. This is similar to the previous representation, except that rather than storing the array of lists *explicitly*, it is implicitly given by a function that takes a node index and returns the list of outgoing edges from that node. This is really important for huge graphs that couldn't possibly stored in a computer, like the web graph. We can even have infinite graphs (!) in this representation.

For the example above, we would first number the nodes a-e as 0, 1, 2, 3, 4. Then the adjacency matrix representation would be:

	0	1	2	3	4
0	0	∞	10	22	∞
1	∞	0	53	∞	45
2	21	∞	0	∞	33
3	∞	∞	∞	0	∞
4	∞	∞	∞	19	0

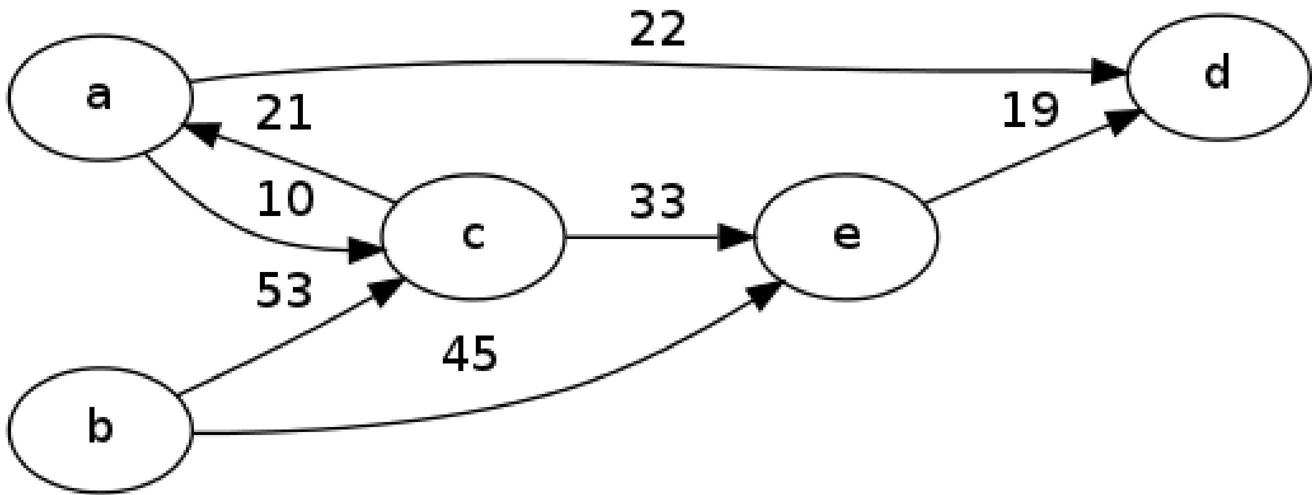

Figure 1: Example 1

Notice that the entries on the main diagonal (like the weight of a node from an edge to itself) were all set to 0, and the entries corresponding to edges not in the graph were set to ∞. This is just a choice which would depend on the application. For example these choices would make sense if the edges represent distances between nodes.

The adjacency list representation, on the other hand, would be something like

```
[(  (2,10),  (3,22)  ),
 (  (2,53),  (4,45)  ),
 (  (0,21),  (4,33)  ),
 (  ),
 (  (3,19)  )
]
```

1.3 Graph Search

Often we want to explore all the nodes and edges in a graph, in the search for a certain node or path in the graph. There are a variety of ways to accomplish this that follow the following basic template:

1. Initialize a set *fringe* with your starting vertex, and a set of *visited* vertices, initially empty.
2. Remove an unvisited vertex from the fringe.
3. Mark that node as *visited*, and add all its neighbors to the fringe.
4. Repeat steps (2) and (3) until all the vertices have been visited.

This description is pretty vague, and that's a good thing! Depending on the details - how the fringe is initialized, how the visited nodes are stored, how to decide which unvisited node is removed next, and what to do when each node gets visited. These differences, and especially the *data structures* that get used for each one, form the main single-source graph search algorithm that you are already familiar with. Let's review:

- **Depth-first search**. The fringe is a LIFO stack, which means the algorithm is going to go deeper and deeper until it reaches the end of the line or loops back on itself. This is useful for discovering ancestor-predecesor relationships in a directed graph. And it's fast, $O(m + n)$ time if using an adjacency list representation for the graph.
- ***Breadth-first search**. The fringe is a FIFO queue, so that the algorithm explores everything one hop away from the starting point, then two hops away, and so forth. This is useful for finding the shortest path in an unweighted graph. It's fast too, $O(m + n)$ time if using the adjacency list representation.

- **Dijkstra's Algorithm**. *In this case, the fringe is a minimum priority queue, and the priorities are the total distance from the starting vertex to each vertex in the fringe. It can be used to find the shortest path from the starting vertex to* any* other vertex, in weighted or unweighted graphs.

  But Dijkstra's is a bit slower than the other searches. For sparse graphs, you want to use an adjacency list for the graph and a heap for the fringe, and you get $O(n + m \log n)$ time. For dense graphs, you want to use an adjacency matrix for the graph and an unsorted array for the fringe, and you get $O(n^2)$ time.

1.4 All-pairs shortest paths

The APSP problem is: given a graph G, find the shortest paths (or at least the *lengths* of the shortest paths) from any vertex to any other vertex. This is useful, for example, in computing fixed routing tables for a large network.

It's also useful when there are going to be many shortest-path searches over the same graph, because you can compute all the answers ahead of time. This is called a *precomputation/query model* of computation, and can cause some big savings. For example, Google does something like this to make Google Maps searches return instantaneously - they've computed (most of) the answers ahead of time!

The really dumb way to do this is to run Dijkstra's algorithm separately between all pairs of vertices in the graph. Since there are $\frac{n(n-1)}{2}$ pairs of vertices, the total cost (for a dense graph) would be $O(n^4)$. Not good!

But then you remember that one run of Dijkstra's algorithm actually finds the shortest path from one vertex to *every other* vertex in the graph. So actually you only have to run Dijkstra's n times, for a total of $O(n^3)$ running time.

But there is another way! The Floyd-Warshall algorithm works by initializing a matrix the same as the adjacency matrix of the graph, with 0's along the diagonal and infinity's where two vertices are not connected with an edge. Then you repeatedly consider allowing each vertex as a potential "stopping point" and update all the shortest paths accordingly. Since there are $O(n^2)$ pairs of vertices, and n possible stopping points (= n rounds of updating), the total cost of this algorithm is $\Theta(n^3)$.

Notice that Floyd-Warshall is the same asymptotic running time as doing a repeated Dijkstra's algorithm for dense graphs, but in practice the Floyd-Warshall algorithm would be much, much faster because it is so much simpler. So much that I can describe it in a few lines of Python code:

```
def FloydWarshall(AM):
    '''Calculates EVERY shortest path length between any two vertices
       in the original adjacency matrix graph.'''
    L = copy(AM)
    n = len(AM)
    for k in range(0, n):
        for i in range(0, n):
            for j in range(0, n):
                L[i][j] = min(L[i][j], L[i][k] + L[k][j])
    return L
```

The real brilliance of this solution lies in the space savings: as we compute the shortest-paths matrix for the next value of k, we can just overwrite the existing values! So we just need one additional n by n matrix, and n iterations through a double-nested loop.

You should recognize that this is actually a dynamic programming solution; it keeps track of a table and builds it up from the bottom-up, one step at a time, until we have the entire, complete answer. Its simplicity and elegance are what have made the Floyd-Warshall algorithm still useful more than **50 years** after it was invented in 1963.

2 Transitive Closure

NOTE: Now this is new stuff. Start paying attention!

Consider an n by n maze, which is just a grid with n^2 positions and some walls between them (or not). We might want to ask whether it is possible to get from one position to another.

Or say we have a bunch of airports, and a directed edge between two airports if there is a direct flight from one to another. We might want to ask whether there is any series of flights to travel from one airport to another.

Or maybe we have a list of elements in some set, and some comparisons between them (typically $\leq$, $\geq$, or both), and then we want to know about the relationship between two elements that have not been directly compared. If we create a graph with one vertex for each element in the set, and an edge from one to another meaning "is less than or equal to", this problem is just like the other two.

All three of these scenarios are about *reachability* in a graph: given a directed graph and a pair of vertices, tell me whether there is any path from one to the other. If we want to answer this question for a specific pair of vertices, the best way is just to do a depth-first search, starting from the starting vertex. If the destination vertex is visited before the starting vertex is colored black, then there is a path.

But in many situations, such as the ones described above, the graph stays the same for a whole series of reachability questions. So we have another instance of the precomputation/query model. What if we want to compute the answer to *every possible* reachability question ahead of time? Then we have the transitive closure problem:

> **Problem**: Transitive Closure
>
> **Input**: Directed graph G = *(V,E)*
>
> **Output**: For every pair of vertices $(u, v) \in V$, true/false depending whether there is any path in G from u to v.

As always, a good first step is comparing this problem to others that we have seen. We can always solve the transitive closure problem by solving the all-pairs shortest paths problem — v is reachable from u iff the shortest path from u to v has length less than ∞.

Therefore (using repeated Dijkstra's or Floyd-Warshall) we can solve the transitive closure problem in $\Theta(n^3)$ time. But it seems like this solution is doing far too much work: is computing all of the optimal paths between vertices actually just as hard as answering whether such a path exists?

2.1 Back to matrices

The way to make transitive closure faster is to think about it in a completely different way, as a matrix problem rather than a graphs problem. Here's how.

Define T_k to be an n by n matrix whose (i, j)'th entry is 1 if there is a path from i to j that uses at most k edges, and 0 otherwise. These are called "boolean matrices" because their entries are all 0 or 1, like true or false.

With this definition, T_0 is just the so-called "identity matrix", which has 1's along the main diagonal and 0's everywhere else. This indicates that every node is reachable from itself, but not from anywhere else, if paths are only allowed to have length 0.

What about T_1? It should be a matrix that has a 1 anywhere there is an edge between two vertices. Well this is just the standard adjacency matrix for the graph itself, with 1 between any pair of connected vertices, 1's along the diagonal (everything connected to itself), and 0's everywhere else.

What we're ultimately interested in of course is T_{n-1}, which will be the transitive closure matrix for the whole graph, allowing paths of any length. To see how this could be computed, we use the following principle: if vertex i can reach vertex j using at most $k + 1$ edges, then there is some intermediate vertex ℓ such that i can reach ℓ in k edges, and there is a direct edge from ℓ to j.

Looping all the n possible intermediate vertices $0 \leq \ell \leq n - 1$, and writing $\wedge$ for the "AND" operation and $\vee$ for the "OR" operation, we have the following formula for the (i, j)'th entry of matrix T_{k+1}:

$$T_{k+1}[i, j] = (T_k[i, 0] \wedge T_k[0, j]) \vee (T_k[i, 1] \wedge T_k[1, j]) \vee \cdots \vee (T_k[i, n - 1] \wedge T_k[n - 1, j]).$$

Now for the crazy part: this is just like the usual matrix multiplication of matrix T_k times matrix A, except that addition has been replaced by OR and multiplication has been replaced by AND. Actually this kind of algebra has a name that you should know already: boolean algebra.

Therefore the transitive closure matrix T_{n-1} can be computed, using boolean arithmetic, as the product of A times itself $n-1$ times, which we can write as a matrix exponentiation like A^{n-1}. You could compute this like $A, A^2, A^3, A^4, A^5, \ldots A^{n-1}$, which would require $O(n)$ matrix multiplications. If each matrix multiplication costs $O(n^3)$, the total cost will be $O(n^4)$. This is bad - it's slower than the Floyd-Warshall algorithm that we already know can be used to solve transitive closure as well.

But wait! We can do better, by recalling the square-and-multiply trick from Unit 3. The square and multiply algorithm shows how to do exponentiation to the power n using only $O(\log n)$ multiplications. We applied it to modular exponentiation for use in RSA, but it can be used just as well for matrix exponentiation, meaning A^{n-1} can be computed using just $O(\log n)$ matrix multiplications, for a total of $O(n^3 \log n)$ time. That's much better, but still not as fast as where we started with the $O(n^3)$ Floyd-Warshall algorithm.

But wait! We can do better, by applying the fast matrix multiplication algorithms such as Strassen's algorithm from Unit 4. To see how this works, we have to turn boolean arithmetic into regular integer arithmetic, but it's not too hard. As usual, 0 means "false", but we will extend the meaning of "true" to include any integer greater than 0. Normal integer multiplication works fine for AND, because the product of two non-negative integers is non-negative if and only if they are both non-negative. And addition works for OR because the sum of two non-negative integers is non-negative whenever at least one of them is non-negative.

This means that each of the $\Theta(\log n)$ boolean matrix products required to solve the transitive closure problem can be accomplished by doing a normal integer multiplication, and then changing every number greater than 1 to a 1. That is, you can solve Transitive Closure by running Strassen's algorithm $O(\log n)$ times. Since Strassen's costs $\Theta(n^{\log_2 7})$, we can solve transitive closure in $\Theta(n^{\log_2 7} \log n)$, which is $O(n^{2.81})$. In fact, this is the fastest way that anyone knows how to solve transitive closure, and using matrix multiplication is the first way anyone figured out how to do it faster than the Floyd-Warshall algorithm.

Take a moment to reflect. We just solved a problem on connectivity in graphs by using a divide-and-conquer algorithm for multiplying integer matrices. THIS HAS BLOWN YOUR MIND. I'll wait while you call your parents and friends to tell them the exciting news.

3 Greedy Algorithms

An *optimization problem* is one for which there are many possible solutions, and our task is to find the "best" one. Of course the definition of "best" will depend on the problem.

We have already seen a few optimization problems: matrix chain multiplication (what's the *best* ordering?), shortest paths. It's also important to realize that some problems are not optimization problems: sorting (only one correct answer), multiplication (only one correct answer), DAG linearization (no "best" answer).

An *algorithm design paradigm* that is useful for many optimization type problems is that of "greedy" algorithms. There are basic steps to setting up a greedy algorithm. First, we have to find a way to construct any solution as a series of steps or "moves". Usually, but not always, these "moves" will just mean choosing a part of the final answer. Then we have to come up with a way to make a "greedy choice": to make the next move or choose the next part of the answer, based on the current status. A general characteristic of greedy algorithms is that this "greedy choice" is made in some simplistic way, so that it's fast. The tough part is proving figuring out how good the solution constructed from a series of greedy choices will be.

3.1 Scheduling Appointments

Say a certain professor is setting up EI appointments by request. He has a number of requests for appointments, but unfortunately they are overlapping so some will have to be denied. How can the professor choose which appointments to schedule so that the maximum number of requests are fulfilled?

Specifically, we have n requests, each of which consists of a starting and an ending time. We want to find a subset of these n requests so that none of the times overlap. An optimal solution will be the largest such subset.

For example, the requests might be

Name	Start	End
Billy	8:30	9:00
Susan	9:00	10:00
Brenda	8:00	8:20
Aaron	8:55	9:05
Paul	8:15	8:45
Brad	7:55	9:45
Pam	9:00	9:30

Solving this problem with a greedy algorithm means choosing which requests to honor, one at a time. After we choose each request, we eliminate all overlapping requests from the possibilities, and continue on until there are none left.

With all greedy algorithms, the key is figuring out what greedy choice we should make. Let's consider three options:

1. **First come, first served**. Pick the request that starts earliest. In the example above, we will pick Brad first. But this eliminates everyone else! So only one request is honored.
2. **Shortest first**. The problems with the previous approach seems to be that we picked a really long appointment first. So instead, let's give preference to the shortest appointments first. This will result in choosing Aaron (eliminating Billy, Susan, Brad, and Pam), and then choosing Brenda (eliminating Paul). So two requests are honored in this case.
3. **Earliest ending first**. Even though Aaron asked for a very short appointment, the time slot overlapped with four others, which was bad. So let's try one more option: choosing the appointment with the earliest ending time, regardless of when it starts. So we will first choose Brenda (eliminating Brad and Paul), then Billy (eliminating Aaron), and then Pam (eliminating Susan). Three requests — the best yet!

All we know at this point is that approaches (1) and (2) do not always give optimal solutions. As it turns out, (3) does always produce an optimal solution. But how could we prove this? There are really two things we need to show: that the greedy choice is always part of an optimal solution, and that the rest of the optimal solution can be found recursively.

> **Lemma**: The request that ends earliest is always part of an optimal solution.
>
> **Proof**: Say the maximum number of requests that can be honored is k, and let
>
> $R_1, R_2, R_3, \ldots, R_k$
>
> be *any* optimal solution, ordered by the times of each request. Furthermore, call R^* the request with the earliest starting time.
>
> Now suppose that $R_1 \neq R^*$, meaning that the earliest-ending request was not included in this optimal solution. Then we simply replace them and consider the solution
>
> $R^*, R_2, R_3, \ldots, R_k.$
>
> This is still a valid solution: since R^* has the earliest ending time, its ending time is earlier than R_1, and therefore it doesn't overlap with R_2. But this solution still consists of k requests that get honored, and so it is in fact an *optimal* solution.

Here's the second thing we have to prove:

> **Lemma**: Say R^* is the earliest-ending request, and
>
> $R^*, R_2, R_3, \ldots, R_k$

is any optimal solution that includes R^*. Then

$R_2, R_3, \ldots, R_k$

is an optimal solution to the subproblem consisting of all the requests that start after R^* finishes.

Proof: Suppose $R_2, \ldots, R_k$ is not an optimal solution to the stated subproblem. Then there exists a valid schedule of requests, all starting after R^* finishes, with size at least k. But then we could add R^* to this sequence and get a sequence of at least $k+1$ requests that could all be honored.

This is a contradiction, however, because it would mean that $R^*, R_2, \ldots, R_k$ is not an optimal solution to the original problem. Therefore the original assumption was false, and $R_2, \ldots, R_k$ is indeed an optimal solution to the subproblem.

Based on these two facts, we know that the greedy algorithm that repeatedly chooses the request with the earlier ending time and that does not conflict with the other requests always produces optimal solutions.

By the way, there is an important pattern to these proofs that is worth noticing: in order to prove that some proposed solution is optimal, we consider *any* optimal solution to the problem, and show that our proposed solution is at least as good as that one. This pattern will come up repeatedly in the analysis of optimization algorithms.

4 Spanning Trees

Remember that a graph is called a *tree* if it's connected and has no cycles. A graph-theory tree is different from a typical computer science tree because there isn't a designated root node with children and all that, i.e., there's no hierarchy defined, just a bunch of connected nodes with no cycles.

An important fact about trees is that a tree on n vertices always has exactly $n-1$ edges. You can prove it by induction if you want!

But most graphs are not trees; they have many more edges and they have cycles and all that. A typical problem in graph theory is to find a tree that lives within this graph: a *spanning tree* is a tree within a graph that touches every node in the graph. In other words, a spanning tree is the graph with all cycles removed.

For example, the bold edges in the graph below form a spanning tree in the graph. (Image credit: Wikipedia)

Spanning trees have numerous important applications such as guaranteeing connectivity in a network. They also uniquely define paths in graphs: for any two vertices u and v, and any spanning tree M, there is exactly one path between u and v that uses only vertices in M. (If there were more than one path, then we would have a cycle, which can't exist in a tree.)

Dijkstra's algorithm for single-source shortest paths actually creates a spanning tree in the graph. Starting with the source vertex, every time a new vertex is added in the main loop of Dijkstra's algorithm, we can think of this as adding a new node and edge to the tree. The spanning tree that results has the useful property that the paths in that tree are all shortest paths from the source node.

Thought of this way (as computing a single-source shortest-paths spanning tree), Dijkstra's algorithm is actually a greedy algorithm! Starting with a size-1 tree containing just the source vertex, the "greedy choice" made by Dijkstra's is to always include the unexplored vertex that is closest (in terms of total weighted path distance) to the source vertex. We already know that Dijkstra's algorithm finds optimal solutions, so this is another example of a greedy approach that always works.

In finding spanning trees, we are often more concerned with the total weight of all the edges in the tree, rather than the distances from a certain node. A spanning tree whose total edge weights is minimal is called a *minimum spanning tree* (MST). Finding MSTs is important for applications such as setting up a network in a building using the least amount of wiring.

4.1 Prims Algorithm

We need an algorithm to find a minimum spanning tree in a graph. Prim's algorithm works very similarly to Dijkstra's algorithm, and actually follows the same general search template as BFS and DFS as well. The only

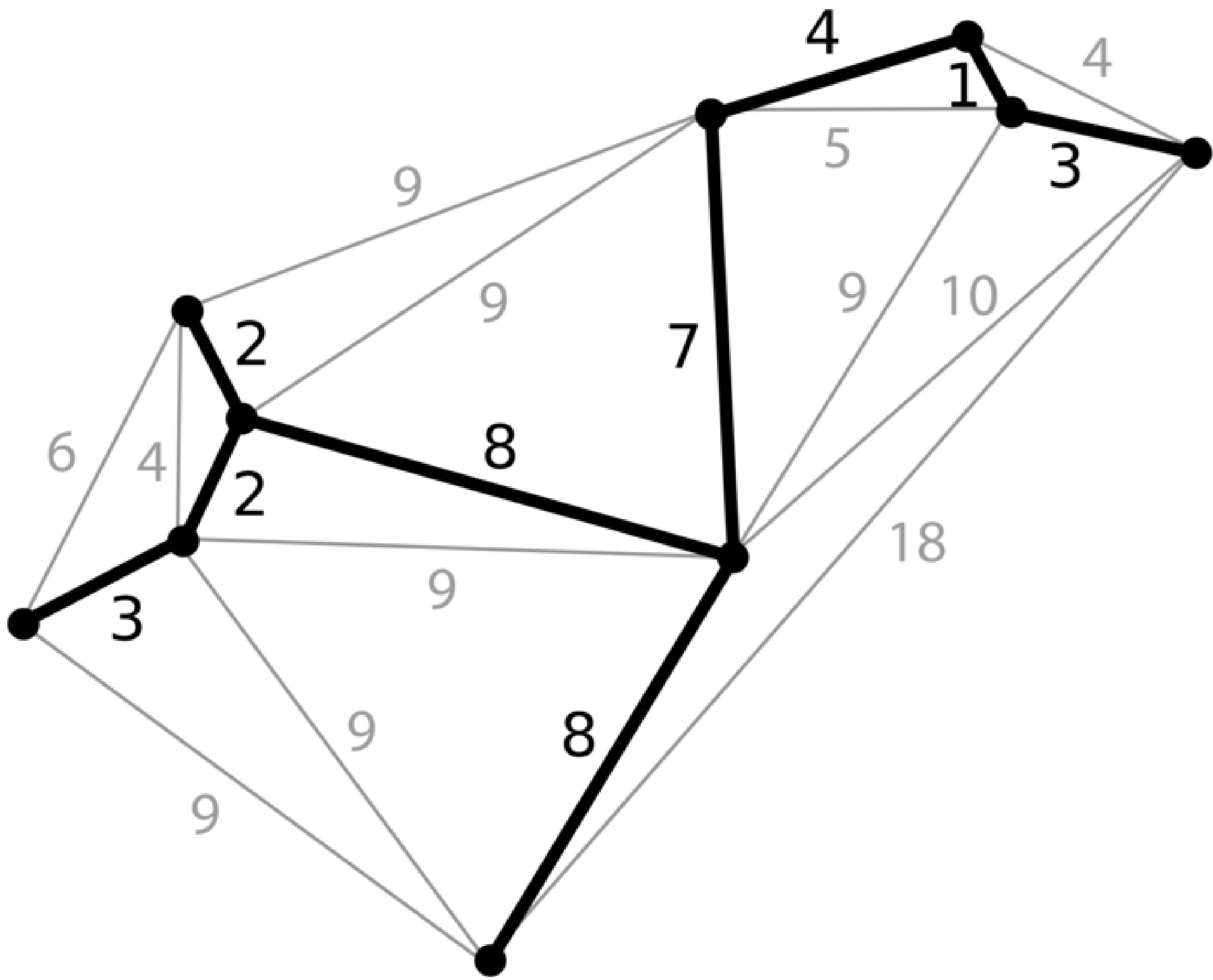

Figure 2: MST example

difference from Dijkstra's algorithm is that instead of making the priority of each node be its *total* distance from the root node, instead we assign each node's priority according to the length of the single edge to that node.

(Note: this algorithm was also independently discovered earlier by someone named Jarnak, but Prim still usually gets the name and the fame.)

You should recognize that this is in fact a greedy algorithm! Starting from any point, the greedy choice at each step is, "Add the shortest edge connecting my spanning tree so far to the rest of the graph." Actually, you could classify Dijkstra's algorithm itself as greedy for the same reason.

As with all greedy algorithms, the question we are faced with is whether Prim's will always give the optimal solution, the smallest-weight spanning tree in the graph. To prove that, we will start with the following important fact about MSTs:

> **Theorem**: Let u be any vertex in a graph G, and let (u, v) be any edge of least weight from among all of u's outgoing edges. Then the edge (u, v) is part of some Minimum Spanning Tree of G.
>
> **Proof**: We follow a similar idea to proving many other optimization strategies: assume that some MST exists that does not contain our desired edge (u, v), then transform it into another MST that does contain it.
>
> Let T be any MST of the graph G, and assume that T does not contain the edge (u, v). Because T is a spanning tree, there must be some path from vertex v to vertex u in the tree T.
>
> Now consider the tree T plus the additional edge (u, v). The result will have a cycle, formed by the path from v to u that already existed in T, plus this additional edge (u, v) that completes the cycle. We also know that, since it's a cycle, there is also a *different* edge that touches vertex u in the cycle, say (u, v').
>
> Well now consider T, plus the additional edge (u, v), and minus the edge (u, v'). Call this new subgraph T'. We know that T' is a spanning tree, because it's a tree (we broke the cycle by removing (u, v')), and because everything is still connected. Furthermore, weight of the edge we added (u, v) must be less than or equal to the weight of the edge we removed (u, v'), so the total weight of all edges in T' is less than or equal to the total weight of T.
>
> Since we assumed T was a MST of the graph, then T' is a MST of the graph as well. Since T' contains our desired edge (u, v), the statement is proven.

Remember that there are always two parts to proving that a greedy algorithm always gives the optimal solution: you have to prove (1) that the greedy choice is always part of some optimal solution, and (2) that the resulting problem after including that greedy choice is a recursive version of the original problem itself.

Our theorem has proven (1), namely that the very first edge chosen by Prim's algorithm is part of some MST. As for (2), we have to consider a slightly different version of the algorithm. Rather than keeping track of which vertices have been "visited" as we would actually do in Prim's algorithm, imagine we collapsed the two vertices together whenever we add a new edge to the tree. (We saw in class what this would look like.)

Although that's not *actually* the way Prim's algorithm works, we can see that the edges chosen by the algorithm if we did it that way would be exactly the same as the edges that are actually chosen. And the advantage of thinking of it like this is that we can now easily justify part (2) of the optimality proof, since collapsing two nodes in the graph would just result in a smaller graph, on which we can recursively apply the same theorem to in order to show that the entire collection of edges produced by Prim's algorithm is in fact a MST.

What about the running time of Prim's algorithm? As usual, this is the "easy part" of the greedy algorithm analysis. It's going to be just like Dijkstra's algorithm, depending on the data structures used. If you have a sparse graph stored in an adjacency list, then you want to use a heap for the fringe, and the running time will be $O(n + m \log n)$. If you have a dense graph stored in adjacency matrix, then you want to use an unsorted array for the fringe, and the running time is $O(n^2)$.

4.2 Kruskals Algorithm

There is also a second greedy approach to finding MSTs in a graph.

Rather than building up a single tree from an arbitrarily-chosen starting vertex, Kruskal's builds up multiple minimum spanning trees simultaneously (called a "forest"). Initially, we start with n trees which are all just the single vertices of the graph. Then at each step, choose the least-weight edge among all those that would connect two trees in the forest. After doing this $n-1$ times, a MST for the whole graph will result.

The proof why Kruskal's algorithm works and always gives an optimal solution to the MST problem is really similar to the proof of Prim's algorithm. The initial greedy choice is definitely going to part of an MST because of the same theorem that we used before. And we can use the same "collapsing nodes" idea to say why the whole thing works.

What's really different about Kruskal's algorithm is the data structures used to implement it. You might not realize this because it's so easy to do Kruskal's on a small example by hand, but are are two aspects that are easy to do "on paper" but which requires some thought when you implement them in a computer.

First, how are you going to go through the edges in order of least weight to greatest? If you have to look through all the edges at each step to see what is the smallest one you haven't considered yet, that will contribute a $O(m^2)$ cost to the algorithm - slow! Instead, the first thing you want to do for Kruskal's algorithm is make a list of all the edges and sort them by weight. Using MergeSort or some other fast sorting algorithm, that can be done in $O(m \log m)$ time.

The second issue is a question we have to answer at every step of the algorithm: given these two nodes, are they already connected (in the same tree), or are they not yet connected (in different trees in the forest)? For that, we are going to need a new kind of data structure.

4.3 Disjoint Set ADT

In order to implement Kruskal's algorithm, we need a way of keeping track of which vertices are connected and which ones aren't. We need to be able to check whether two vertices are connected already, and if they aren't, we need a way to connect them (and everything they're connected to as well).

It would be possible to just do this within the graph, but it would be slow. Each time you considered a new edge, you would need to search in the graph to see if the two endpoints are already connected. Performing that search would cost at most $O(n)$, and since you have to potentially consider all m edges, the total running time for this approach would be $O(nm)$. That's much slower than Prim's algorithm for MST; no good!

Instead, we will use a data structure to store this information for us. We have a big forest full of little trees, and we need to keep track of which tree each node is in. If two nodes are in different trees, you can safely add the edge between them to combine those trees and not introduce any cycles. But if two nodes are in the same tree, adding the edge between them would create a cycle.

The kind of data structure we need to keep track of this information is called a *disjoint-set data structure*. The name comes from the idea that we have a universe of n items (the vertices in the original graph), which are split into some number of disjoint sets (the trees). The operations you can do are to look up the name of the set each item is in, or to combine two existing sets to make a bigger one.

This leads to an ADT with three operations:

- **create(items)**: add all items, each as a separate set
- **find(x)**: find the set containing x
- **union(x,y)**: combine the sets containing x and y into a single set

There are many ways we could implement this data structure, which might have different trade-offs between the cost of these different operations. For our purpose here, we just want to make Kruskal's algorithm run as fast as possible, and that means we want the **find** operation to be really quick.

So the data structure to use for the disjoint set is a hashtable of linked lists. Your hashtable will map vertex names to linked lists, where each vertex's entry in the hashtable points to a linked list containing the names of all the vertices in that vertex's set. We will take the "name" of the set to simply be the first name in the linked list. Crucially, *all vertices in the same set point to the same, shared linked list.* At any point in time, the number of hashtable entries is always n (the number of vertices), but the number of linked lists might be smaller depending on how many sets there are.

Doing a “find” operation just means looking up that node in the hashtable, and returning the name of the first label in the linked list. Which is $O(1)$ time; fast.

The slightly tricky operation is “union”. You have two linked lists of items, and you want to merge them together into a single set. The two things you need to do are (1) add all the items in one list to the end of the other list, and (2) update the hashtable pointers for everything you added, so they point to the other list.

If you are union-ing two sets of the same size, which one gets added to which doesn’t matter. But if one of the sets is larger, you want to follow the *weighted union heuristic*, which says to always merge the smaller set into the bigger one. That’s a good idea because it minimizes the number of pointers that have to get changed, and makes the whole thing faster.

The worst-case cost of a single union operation is $O(n)$, which can happen at the very end, say, if you end up with two equal-size sets and have to combine them together.

But that worst-case behavior can only happen once, at the very end! Notice that the total number of union operations that can ever be performed is $n - 1$, since at that point you have just one big set left. Well, the **total** worst-case cost of those $n - 1$ union operations is not $O(n^2)$ as you might expect, but rather $O(n \log n)$.

In class we framed this with a recurrence relation to understand that total. Another way of understanding it is to consider a single vertex: every time its pointer changes, that means the size of the set it’s going into is at least doubling. The most number of times you can double before reaching n is $\lg n$. If each element’s pointer can only get updated $O(\log n)$ times, the total number of pointer updates, and therefore the total cost of all unions, is $O(n \log n)$.

You may recall from Data Structures class that there’s a name for this phenomenon, when a single operation has a slow worst-case cost, but the total over many operations is better. We want to use *amortized analysis*, and talk about the amortized cost per operation if we were to spread out the costs evenly. For “union”, the amortized cost means spreading out the $O(n \log n)$ total over the $O(n)$ operations that you would need to get there, which leads to an amortized cost of $O(\log n)$ per operation. Not quite as fast as “find”, but still pretty great!

4.4 Kruskal Example

Let’s have an example, shall we?

First thing to do is sort the edges by weight:

```
(C,D,3)   (F,G,5)   (C,F,6)   (D,G,8)   (A,B,11)   (D,F,14)
(B,E,19)  (B,D,21)  (A,C,25)  (E,G,26)  (D,E,27)
```

This graph has 7 nodes, so we initially have 7 hashtable entries, each pointing to 7 different linked lists, like

```
A -> L1          L1 = [A]
B -> L2          L2 = [B]
C -> L3          L3 = [C]
D -> L4          L4 = [D]
E -> L5          L5 = [E]
F -> L6          L6 = [F]
G -> L7          L7 = [G]
```

Now the fun begins. The first edge to consider is (C,D,3). We do a **find** to check whether C and D are in the same tree yet. **find(C)** returns C and **find(D)** returns D, so that means they are in different sets and we want to include this edge in the MST.

Every time we include a new MST edge, we have to perform a **union** operation in the data structure. That means adding everything in one linked list to the other one, and updating the pointers of everything that got added. Here’s what the structure looks like after doing **union(C,D)**:

```
A -> L1          L1 = [A]
B -> L2          L2 = [B]
C -> L3          L3 = [C,D]
D -> L3          L4 = (erased)
```

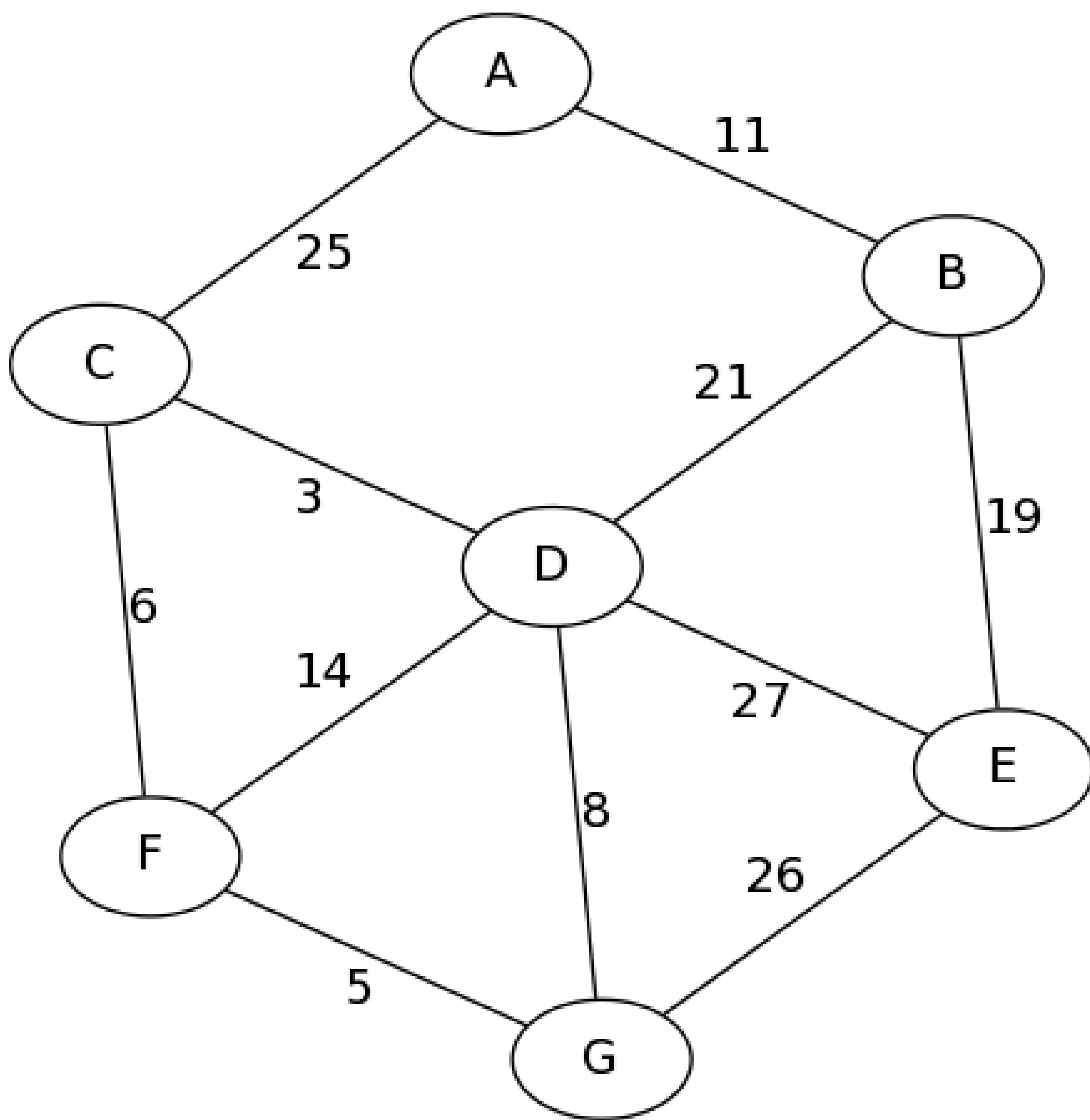

Figure 3: MST example

```
E −> L5          L5 = [E]
F −> L6          L6 = [F]
G −> L7          L7 = [G]
```

The next edge is (F,G,5), and that gets added to the MST as well, causing another union. Note that which list gets added to which doesn't matter, because both of them have the same size. We end up with:

```
A −> L1          L1 = [A]
B −> L2          L2 = [B]
C −> L3          L3 = [C,D]
D −> L3          L4 = (erased)
E −> L5          L5 = [E]
F −> L7          L6 = (erased)
G −> L7          L7 = [G,F]
```

Next is (C,F,6). We first do a **find(C)** operation, which returns the first thing in C's list, namely C. And a **find(F)** returns G. That's how the algorithm knows that C and F are in different trees, so it's safe to add the edge without creating a cycle. As usual, adding the edge (C,F) to the MST means we perform another union operation, this time combining two size-2 sets. Again, since they're the same size, which one gets added to which doesn't matter; either way 2 pointers have to be updated.

```
A −> L1          L1 = [A]
B −> L2          L2 = [B]
C −> L7          L3 = (erased)
D −> L7          L4 = (erased)
E −> L5          L5 = [E]
F −> L7          L6 = (erased)
G −> L7          L7 = [G,F,C,D]
```

The next edge is (D,G,8). You can see visually that adding this would create a cycle in with the other 3 edges that have already been added to the MST. But how does the algorithm know this? It does **find(D)**, which returns G (the first node in D's list), and **find(G)**, which also returns G. That means these are in the same set, so the algorithm discards this edge and moves on to the next one. Notice that no union-ing happens for this edge.

Next edge is (A,B,11), which gets added to the MST because A and B are in different sets. After the ensuing **union**, the data structure looks like

```
A −> L1          L1 = [A,B]
B −> L1          L2 = (erased)
C −> L7          L3 = (erased)
D −> L7          L4 = (erased)
E −> L5          L5 = [E]
F −> L7          L6 = (erased)
G −> L7          L7 = [G,F,C,D]
```

Next is (D,F,14), which is discarded since **find(D)** and **find(F)** both return G.

The edge (B,E,19) is next, which will be added to the MST because B and E are not in the same set yet. This time, when performing the **union** operation, *the order actually matters.* The smaller set is [E], so that's the one that should be added to the end of the other list, requiring only a single pointer update. If we did it the other way around, it would require two pointers to be updated.

```
A −> L1          L1 = [A,B,E]
B −> L1          L2 = (erased)
C −> L7          L3 = (erased)
D −> L7          L4 = (erased)
E −> L1          L5 = (erased)
F −> L7          L6 = (erased)
G −> L7          L7 = [G,F,C,D]
```

And now, finally, we come to edge (B,D,21), which is going to connect these two pieces and complete the MST. Again, the smaller set gets added to the end of the larger one in our data structure:

```
A -> L7          L1 = (erased)
B -> L7          L2 = (erased)
C -> L7          L3 = (erased)
D -> L7          L4 = (erased)
E -> L7          L5 = (erased)
F -> L7          L6 = (erased)
G -> L7          L7 = [G,F,C,D,A,B,E]
```

There are four more edges in the list, but no point in going through them since the MST is already complete.

4.5 Analysis of Kruskal

The first step of Kruskal's algorithm is to sort the edges by weight, which costs $O(m \log m)$ time.

After that, we have to do $2m$ **find** operations and $n-1$ **union** operations on the disjoint-set data structure. As discussed above, we know that each find costs $O(1)$ and each union costs $O(\log n)$ amortized. Therefore the total cost of the disjoint set operations is $O(m + n \log n)$.

In most cases, the number of edges exceeds the number of nodes, so the whole cost will actually be dominated by the sorting step $O(m \log m)$. Accounting for the fact that

$\log m \le \log n^2 = 2 \log n \in O(\log n)$,

the total cost in general would be $O((m+n) \log n)$.

This is basically the same as Prim's algorithm cost.

5 Matchings

Greedy algorithms are starting to look pretty good for solving problems with graphs! Let's look at another example.

Consider the following problem: There are a n college students that will all be living in the same residence next year. They all know each other, and have all submitted preference forms indicating everyone else that they might like to live with. Each room holds at most two students. Your task is to figure out how to assign roommates so that the largest number of them end up with someone they requested.

The first step to solving this and many other problems is modeling the situation with a graph. And in this case the graph is pretty easy: the vertices are the students, and there is an edge between any two students that requested each other as roommates. (Yes, we are ignoring the potentially thorny situation of unrequited requests.)

So the problem is: given an undirected, unweighted graph, find the largest possible subset of edges such that no vertex sits on two different edges in the set. (In our original problem, this means the greatest number of happily-paired roommates, such that each student is in at most one pair.) Here's an example input graph to for this problem. Do you see an optimal solution?

As it turns out, there is a word for a subset of edges that don't touch: it's called a *matching* in the graph. The problem we're considering here is called the "maximum matching problem" and is a classic question to consider in graph theory as well as algorithms.

Now you know the technique that we're going to throw at this problem: greedy algorithms! And there is a very simple greedy solution to the maximum matching problem: repeatedly pick an edge (any edge), then remove those two vertices and all adjacent edges from the graph, and recurse. When there aren't any edges left, we're done.

This shows the result of applying this greedy strategy to the example graph above. The edges that make up the matching are in blue, and the original edges of the graph are shown as dashed lines:

Unfortunately, this is not an optimal solution. In fact, there is an example with only four vertices that demonstrates that the greedy strategy will produce sub-optimal solutions (can you come up with that example?). By contrast, here is an actual maximal matching for our running example graph:

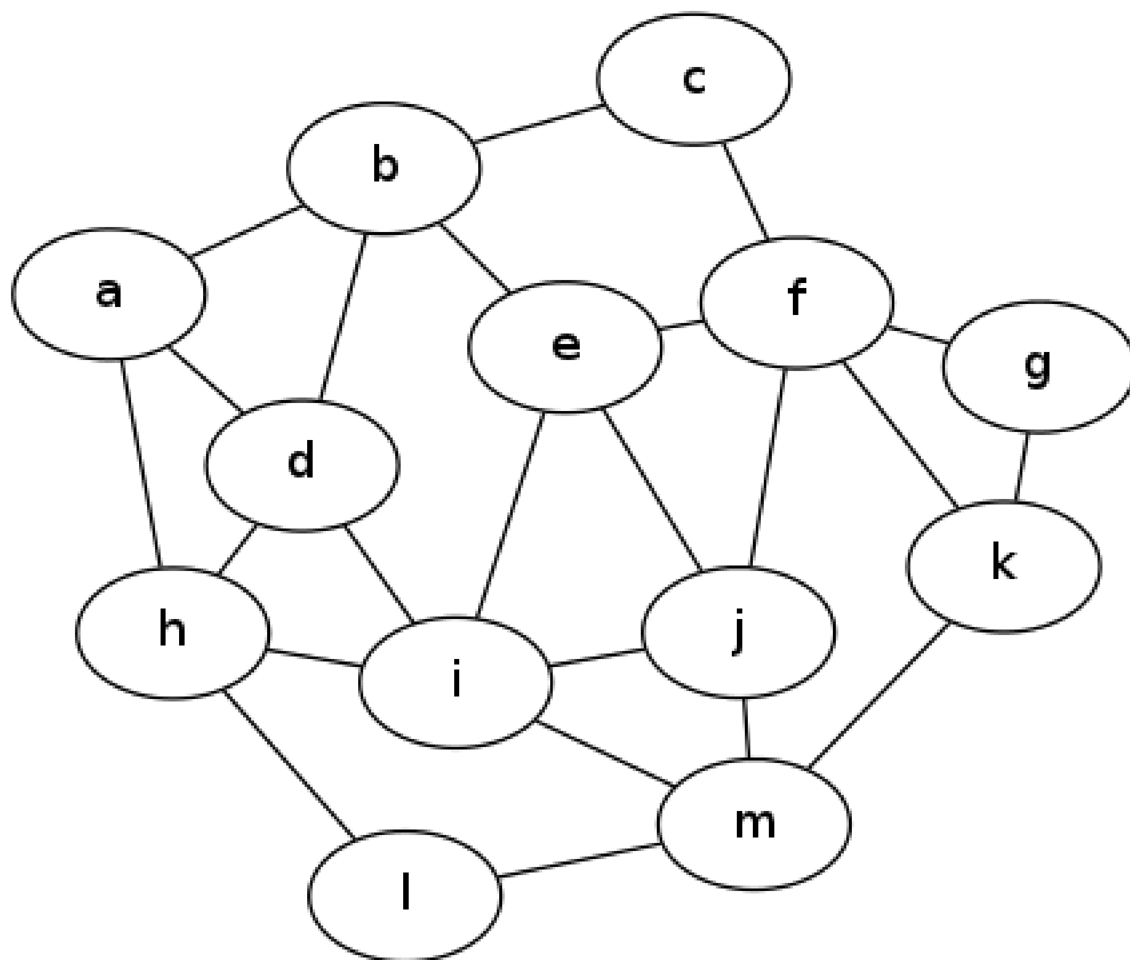

Figure 4: Matchings graph 1

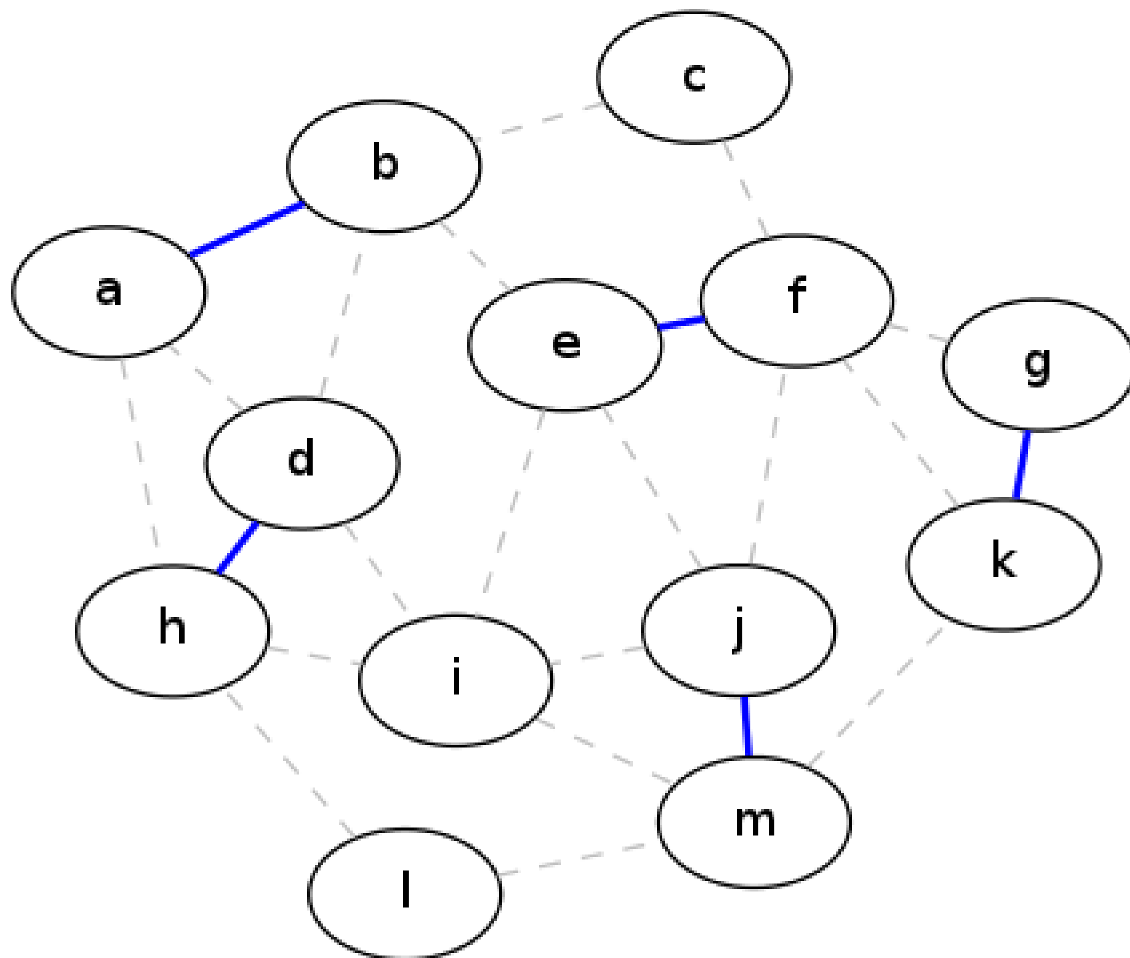

Figure 5: A greedy matching

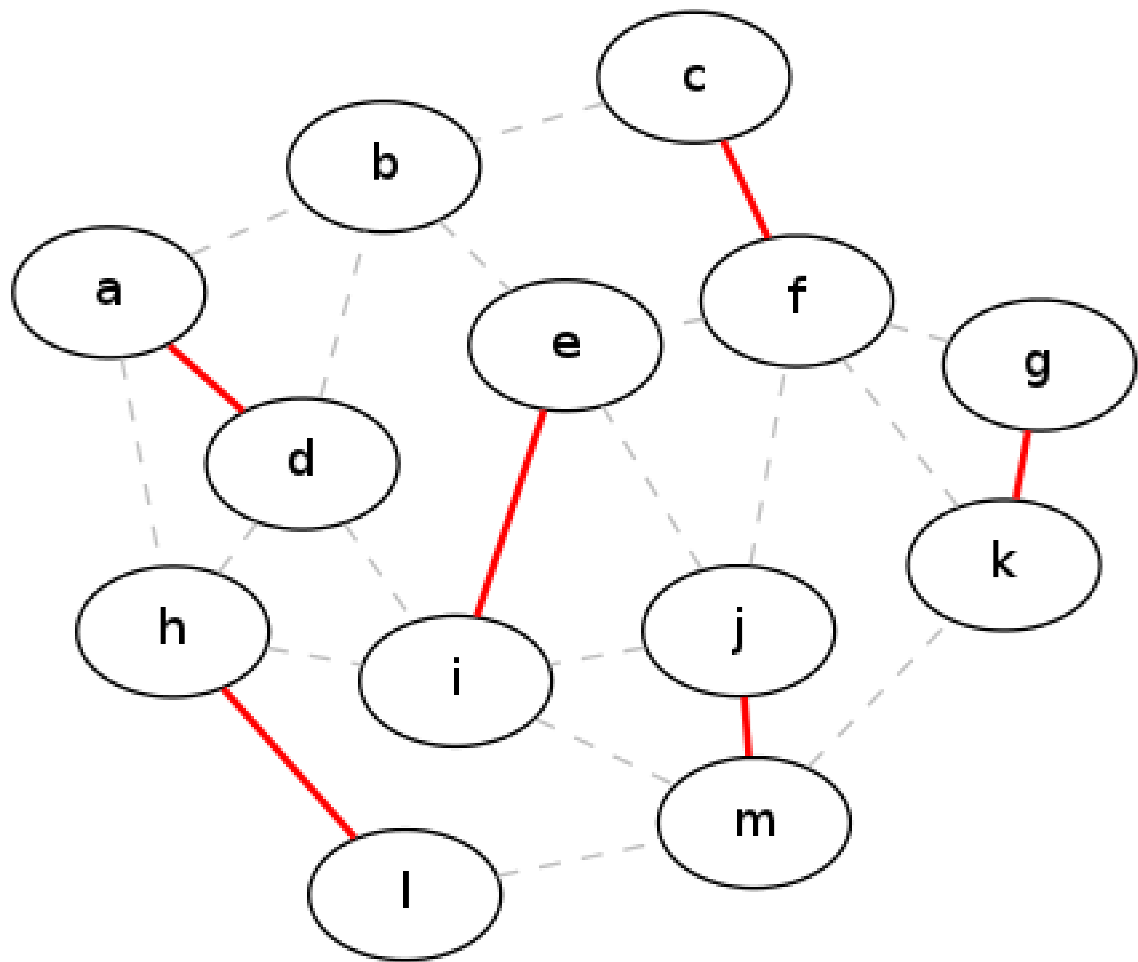

Figure 6: Optimal matching

The key shortcoming to the greedy strategy in this case is that never allows us to "change our minds" about a partial solution. The key to making greedy choices is that they can never be undone. The example above shows that we can end up with a case where the choices we have already made prevent us from adding any more edges to the graph, so we are "stuck" with a sub-optimal solution.

A sensible question to ask at this point is, how bad is the greedy solution? In the example above, the optimal solution has size six, whereas the greedy solution had size five. So is the greedy solution always at least one less than the size of the optimal? Or maybe five-sixths the size of the optimal? The next theorem answers these questions:

> **Theorem**: Any matching produced by the greedy strategy has at least half as many edges as a maximum matching.
>
> **Proof**: For the same graph G, consider a matching M_1 produced by the greedy strategy, and a maximum matching M_2. We want to compare the edges in each of these matchings.
>
> First, remove all the edges that the two matchings have in common. Now consider the subgraph of G that results from combining the unique edges in M_1 and M_2. Call this combined graph G^*. What does this graph look like?
>
> One thing we know is that every edge in G^* has degree at most 2 — that is, there are at most two edges touching it. That's because if there were three edges, two of them would have to be in the same matching which is impossible.
>
> Now because each vertex has degree at most two, this means that the graph G^* must just be a collection of simple paths and cycles. These are simply the only kind of things that you can get in a graph with maximum degree 2. Furthermore, the edges in these paths and cycles must be alternating between M_1 and M_2.
>
> How long are the paths and cycles? They can't have length 1, because that would mean there's an edge which could have been added to one of the matchings without removing any of the other edges, but it wasn't added — a contradiction.
>
> Finally, because the edges alternate, any path with length 2 has at most one more edge from M_2 than from M_1. Therefore the worst case comes with length-three paths, which will have twice as many edges from M_2 as from M_1. Since there may be arbitrarily many of these length-3 paths in G^*, in the worst case M_2 will have twice as many edges as M_1. In other words, the maximal matching will never have more than twice as many edges as a greedy matching.

The following picture shows the two matchings (greedy and optimal) for the previous example, overlaid to show the graph G^* from the proof:

5.1 Speed-Correctness Tradeoff

Incidentally, there *is* an algorithm for the maximum matching problem that always finds solutions, and it's faster than the "brute-force" method of trying every possible subset of edges. The first algorithm to solve this problem in polynomial-time was invented by Jack Edmonds in 1965 and is called (appropriately) "Edmonds's algorithm", or sometimes "Edmonds's matching algorithm" to distinguish it from his other results. You can read the original paper here if you like. It's based on starting with the greedy algorithm and them improving it by finding certain alternating-color paths and loops, like in the proof above.

You might recognize the name Edmonds from a few units ago: he's the guy who (along with Cobham) proposed that every "tractable" algorithm must be polynomial-time. That is, its cost must be bounded by $O(n^k)$ for some constant k, and where n is the total size of the inputs. Well section 2 of that paper on matchings is exactly where Edmonds made this claim, in the context of matching algorithms!

Part of the reason that Edmonds felt the need to talk about what is meant by a "tractable" algorithm is that his algorithm for finding maximum matchings is pretty slow: the worst-case cost is $\Theta(n^4)$. This is polynomial-time, but the growth rate is faster than most other algorithms we have studied.

Although faster algorithms for this problem have been developed since then, none of them is as fast as the greedy algorithm above, which has worst-case cost $\Theta(n+m)$ if the adjacency lists representation is used. So we have here the

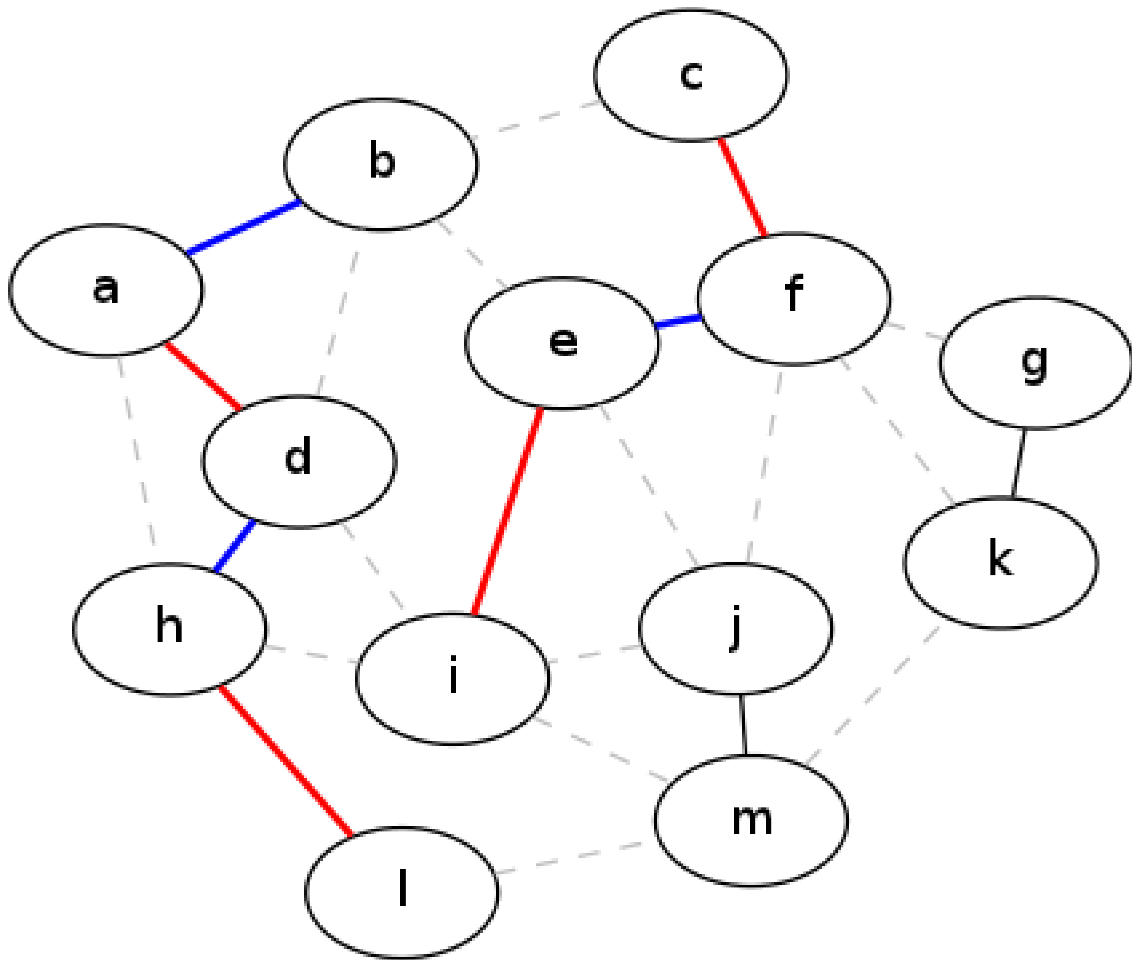

Figure 7: G-star graph

first example of a problem where *correctness* (or more properly, *optimality*) *can be traded off against other resources such as time and space.* Depending on the specific circumstances, a sub-optimal solution might be acceptable if we can get it really quickly. Or maybe we want to spend the extra time and get the best possible answer. The point is that there is a balance of concerns to account for, and not a single "best" algorithm.

6 Vertex Cover

Consider the following scenario: You have a bunch of locations (vertices) that are connected by some waterways (edges). Your task is to choose which locations to use as bases, so that every waterway is touched by at least one base. That is, you want to choose a subset of the vertices so that every edge is adjacent to a vertex in the subset.

This problem is called *vertex cover*, and the optimization question is to find the smallest such subset, the *minimal vertex cover*. For example, have a look at this graph (same one we used for matching):

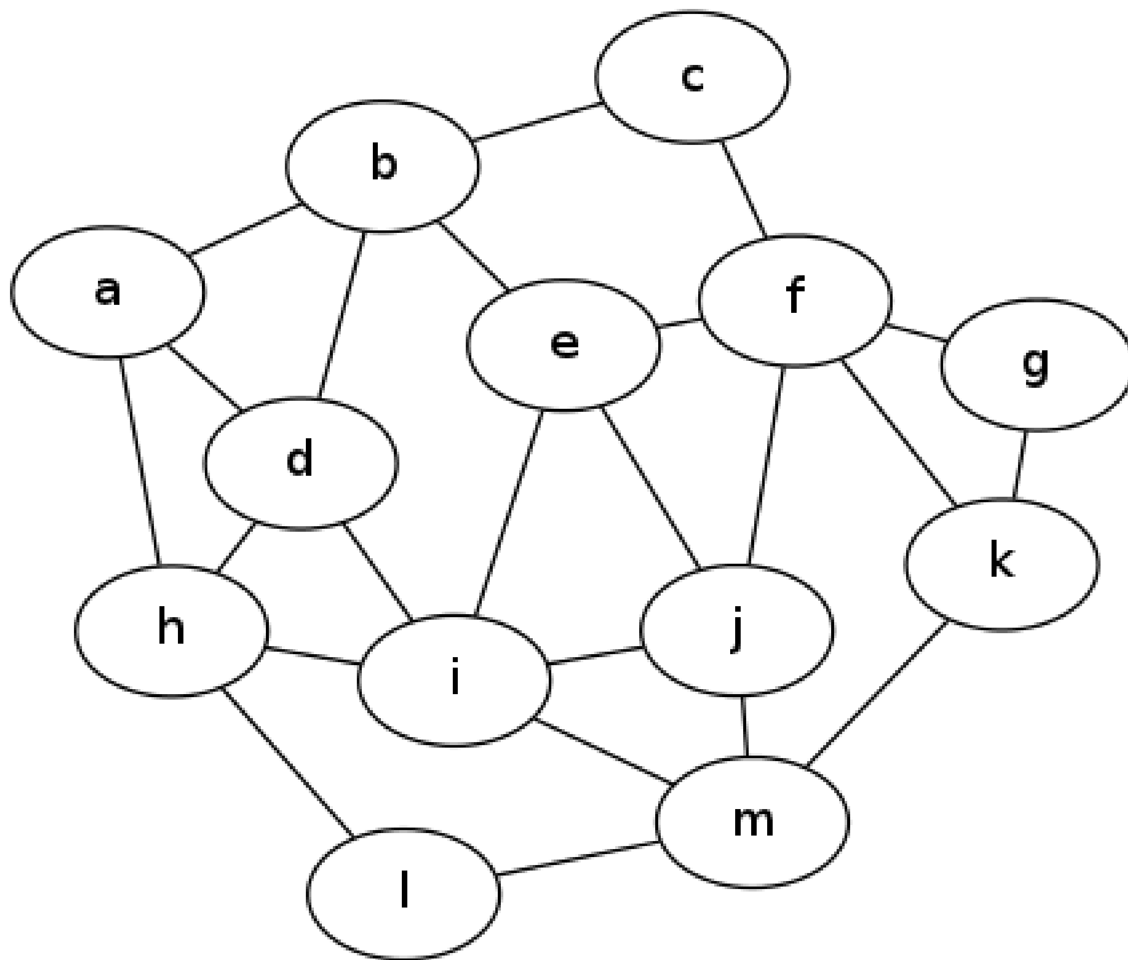

Figure 8: Vertex Cover Example

Here are some vertex covers for this graph:

```
1 {b, c, d, f, g, h, i, j, l}
2 {a, c, d, e, g, i, j, l, m}
3 {a, b, d, e, f, g, h, j, k, m}
4 {a, b, d, f, i, j, k, l}
5 {a, b, e, f, h, i, k, m}
6 {b, d, f, h, i, j, k, m}
```

Now we know that the first three can't be *minimal* vertex covers. What about the last 3, with 8 vertices each? They're the smallest we've found so far, but so what? Can you see any way of confirming that 8 is the smallest vertex cover *without* having to manually check every possible one?

6.1 Vertex Covers and Matchings

There is a deep and very important connection between vertex covers and maximal matchings in the same graph. We didn't define "maximal matching" before, but it means any matching that can't be trivially added to. (Notice that it is definitely not the same as a maxi*mum* matching!) For example, the greedy algorithm for matchings always outputs a maximal matching, but not necessarily a maximum one.

The following two lemmas relate the size of maximal matchings and vertex covers.

> **Lemma**: Every vertex cover is at least as large as any matching in the same graph.
>
> **Proof**: For each edge in the matching, one of its two endpoints must be in the vertex cover, or else it's not a vertex cover. But since it's a matching, none of the endpoints of any of the edges in the matching are the same.
>
> In other words, just to cover the edges in the matching itself, the size of the vertex cover must be at least the size of the matching. Since the matching is part of the original graph, all those edges must be covered, and we get the stated inequality. QED.

For the example above, we know from before that the maximum matching in this graph has 6 edges. Therefore, from the lemma, every vertex cover must have at least 6 vertices in it. Unfortunately, this isn't quite enough to show that 8 vertices constitute a minimal vertex cover.

The next lemma gives an inequality in the other direction, to intracately tie these two problems together.

> **Lemma**: The minimum vertex cover is at most twice as large as any maximal matching in the same graph. (Note maxiMAL, not maxiMUM.)
>
> **Proof**: Let $G = (V, E)$ be any unweighted, undirected graph. Take any maximal matching M in G. Now let $C \subseteq V$ be a set of vertices consisting of every endpoint of every edge in M. I claim that C is a vertex cover.
>
> To prove the claim that C is a vertex cover, suppose by way of contradition that it isn't. This means that there's some edge in E that doesn't touch any vertex in C. But this means that this edge doesn't touch any edge in M, and therefore it could be added to M to produce a larger matching. This contradicts the statement that M is a maximal matching. So the original assumption must be false; namely, C is a vertex cover.
>
> From the way C was defined, its size is two times the size of M. Since the minimum vertex cover must be no larger than C, the statement of the lemma holds in every case. QED.

Combining these two statements, we see that any maximal matching M in a graph provides an upper and a lower bound on the size of the minimum vertex cover c:

$$|M| \leq c \leq 2|M|$$

Moreover, the second lemma is *constructive*, meaning that it doesn't just give us a bound on the size of the vertex cover, it actually tells us how to find one. Let's examine that algorithm more carefully.

6.2 Approximating VC

Here's the algorithm to approximate vertex cover, using the greedy algorithm for finding a maximal matching, and the construction described in the second lemma above:

```
def approxVC(G):
    C = set() # makes an empty set
    for (u,v,w) in G.edges():
        if u not in C and v not in C:
            C.add(u)
            C.add(v)
    return C
```

This algorithm is basically just finding a greedy matching and adding both vertices of every edge in the matching to the vertex cover. Unfortunately this doesn't always give the exact minimum vertex cover. But how close is it? The lemmas above provide the answer.

> **Theorem**: approxVC(G) always finds a vertex cover that is at most twice as large as the minimum vertex cover.
>
> **Proof**: From the second lemma, we know that the set C returned by the algorithm is always a vertex cover of G, and the size of C is $2|M|$, where M is the greedy matching that is being found. But from the first lemma, we also know that any vertex cover must be at least as large as M. Therefore C is at most twice as large as the minimum vertex cover.

We therefore say that approxVC has an approximation ratio of 2 for the minimum vertex cover problem. We also saw a similar result earlier for the matching problem itself: the greedy matching algorithm returns a matching that is at at least one-half the size of the optimal maximum matching.

So we have a factor-two approximation algorithm for vertex cover, that uses a factor-two approximation algorithm for maximal matching. Now we know that there is a polynomial-time algorithm that always finds an optimal maximum matching — can we plug this into the VC approximation algorithm to solve this one in polynomial-time too?

The answer, which might be shocking, is a resounding no! These two optimization problems are optimizing in opposite directions: *minimum* vertex cover versus *maximum* matching. Therefore the very best matching algorithm will not give the best vertex cover. To approximate VC most effectively, we want a matching that is maximal (can't be added to), but which is *not* maximum. And the greedy matching algorithm provides this exactly! So the irony is that our sub-optimal matching algorithm which is fastest (the greedy one) gives the best approximation for vertex cover!

Now we are still left with the question of whether we can actually find the minimum vertex cover exactly, in polynomial-time. Can you think of any algorithm? If you do, please let me know, because there a 1-million dollar prize that we could split. But more on that in the next unit…

7 Traveling Salesman

We have one more hard problem to consider, and it's one of the most famous and classical computer science problems with graphs.

Say you are a delivery truck driver and you have a bunch of stops for deliveries. You want to schedule a route that starts and ends at your warehouse and visits every stop exactly once. And of course you'd like to do this in the shortest time possible! So how do you decide the route? That's this problem:

> **Traveling Salesman Problem**: TSP(G)
>
> **Input**: Weighted graph G
>
> **Output**: Least-weight cycle in G that goes through every vertex exactly once.

This is an important problem with numerous other applications, and it's also notoriously difficult to solve. In fact, we'll learn in the next unit that a fast solution to TSP, like the minimum Vertex Cover problem, would earn you a million bucks!

In other words, there almost certainly is no optimal, fast solution to this problem. What we're going to look at is a quick overview of various strategies to *try and solve it anyway*. The approaches we look at will either have to compromise in running time (taking worst-case exponential time) or in the quality of the answer (not always giving the optimum answer).

Here is a graph for an example TSP problem that we'll use repeatedly:

The dashed lines both have length 6 (not drawn). You should convince yourself that the optimal TSP tour has total length 18.

7.1 Minimum Spanning Trees

Just like matchings were used to approximate the vertex cover problem, minimum spanning trees (MSTs) are often used to approximate TSP. Here's the connection:

> **Theorem**: The length of any travelling salesman tour in a graph G is at least as much as the total size of a minimum spanning tree in G.
>
> **Proof**: Take any travelling salesman tour (cycle); call it T. Now remove the last edge from that cycle. The result is a path that touches every vertex exactly once; call this path P.
>
> What do we know about P? Well, it connects every vertex and it has exactly $|V| - 1$ edges. Therefore... it's a tree! In fact, it's a *spanning tree* since it touches every vertex. Therefore the length of P must be greater than or equal to the size of any MST in the graph.
>
> And finally, since P was formed by removing an edge from T, the length of the original travelling salesman tour T must be greater than the length of P, which in turn is at least as large as the size of any MST.

Since we know at least two fast (polynomial-time) algorithms to compute MSTs (Prim's and Kruskal's), this theorem provides a quick way to find a lower bound on the size of the optimal TSP solution. (This is similar to how the size of any matching gave a lower bound on the size of a minimum vertex cover in a graph.)

7.2 Branch and Bound

Here's a basic algorithm to compute the TSP:

1. Pick a starting vertex u arbitrarily
2. Explore every possible path in the graph from u, depth-first.
3. When the depth-first search gets to a leaf (can't go any further), check if the path contains every vertex. If so, add the edge back to u from the ending vertex, and that gives a Hamiltonian cycle
4. After completing the depth-first exploration, return the least-weight Hamiltonian cycle that has been found.

Now this is a very slow algorithm; it's cost in the worst case is something like $\Theta(|V|!)$, which is the total number of paths in the graph. This is not particularly surprising — we already said that no fast, optimal solution will exist for TSP.

So what can we do? How about instead of trying to improve the worst case, we improve the *best case* instead? The idea is to compute the optimal solution as fast as we can, and in many cases it should work in polynomial-time.

"Branch and bound" is a popular and effective method for finding optimal solutions to hard problems. What branch-and-bound requires is a quick way to estimate a *lower bound* on the cost of the remaining TSP tour, given some partial tour (i.e., a path). This lower bound estimate is usually called a "heuristic" (this corresponds to the same idea in the A* shortest-path search that you may have heard of). We will use the heuristic to speed up the search as follows:

- The order vertices are explored in is *in order of the heuristic*. If the heuristic actually gives a good estimate on the true cost of the remaining TSP tour, then this will lead us to finding the optimal tour sooner rather than later in the DFS.

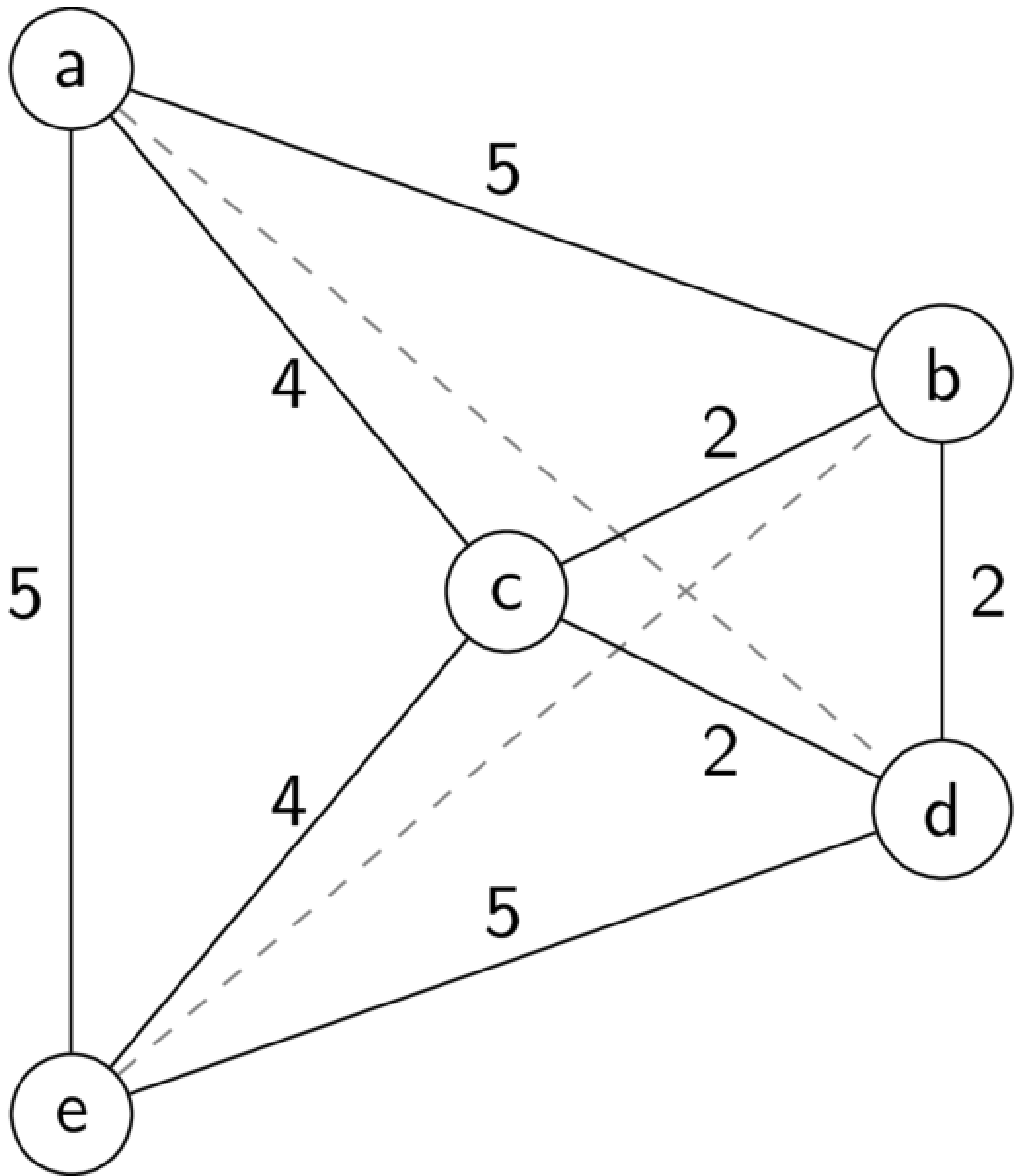

Figure 9: Example TSP problem

- We always keep track of the best solution found so far. If the heuristic gives a *lower bound* on the remaining subproblem that is worse than the best TSP tour that we've already found, then we can **stop exploring** from that path and move on to the next option.

Basically, what the heuristic gives us is a way to (1) find the optimal solution quickly, and (2) eliminate sub-optimal solutions from the search without having to completely explore them.

Now the very best heuristic would be to just find the optimal TSP tour on that subproblem — but of course we don't want to do this because that would take a really long time! So how could we get a quick lower bound on the optimal TSP tour? We just saw it — compute the minimum spanning tree! Specifically, given a partial TSP tour (path in the graph), the heuristic adds up the length of the path, the shortest edges going out from the ends of the path to the rest of the graph, and the size of a MST in the rest of the graph. This gives a lower bound on the smallest TSP tour that includes that path.

Here's a concrete example. Consider the following graph, with the partial tour explored so far highlighted in blue (b-c-d), and the edges that can't possibly be part of this tour reduced to gray:

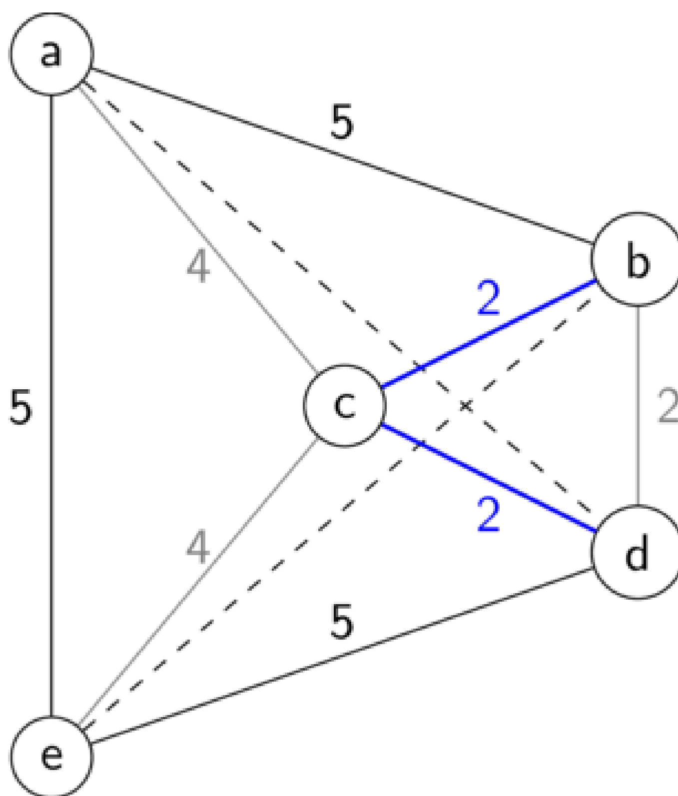

Figure 10: Partial TSP tour

The branch-and-bound heuristic would first compute a MST in the rest of the graph not including the path so far. In this case that is just a MST on the two vertices a and e, which will consist of just the edge between them, weight 5. Then we add the length of the path so far (4), plus the shortest edges going from the ends of this path to the rest of the graph (5 and 5), plus the MST on the rest (5), for a total of 19. So if the optimal length-18 solution had already been found, this path could be eliminated without having to explore it further.

I know this seems really trivial on such a small graph, but on much larger graphs a huge savings is obtained from being able to eliminate large parts of the depth-first exploration. Many other heuristics have been developed as well, some that work only in certain situations. There is always a balance between using a very fast but inaccurate heuristic (makes the search go faster, but has to explore more), or a slower but more accurate heuristic (more time spent on the heuristic, less time spent exploring). The best choice usually depends on some intuitions about whatever the actual *meaning* of the graph is for a particular application.

7.3 Metric TSP

In many applications, the graph for TSP corresponds to some kind of actual map where the nodes have actual locations, and the distances between them correspond to the weights of the edges in the graph. This is called the *Euclidean TSP* problem because the nodes are just points on a 2-dimensional Euclidean plane.

One property that points on a plane obey is the *triangle inequality*: the distance from a to b is always less than or equal the distance of a to c plus c to b; that is, detours never make the trip faster. In other words, *the shortest path between any two vertices is the single edge between them.* Since this is a property that can be obeyed by graphs besides just the ones that are points in a plane, this is called the *Metric TSP* problem.

Unfortunately, Metric TSP is still a very hard problem, so we're still not going to have a polynomial-time algorithm. Notice that the branch-and-bound approach above *always* finds the optimal solution, and *sometimes* takes polynomial time. What we'll do here is flip that around, so that we *always* take polynomial-time, but only *sometimes* find the optimal solution.

Again, the idea boils down to minimum spanning trees! Here it is:

1. Compute a MST of the graph.
2. Duplicate every edge in the MST. This makes a cycle with *duplicate edges*.
3. Go through this duplicated-edge cycle, and just skip every vertex that gets repeated, taking the shortcut to the next *unvisited* vertex instead.
4. The result is a TSP tour!

Here's an example to see how this works. On the left is a MST of the graph, which gives the cycle $(a, c, b, c, d, c, e, c, a)$. Now we just go through this but remove every *second* occurrence of a vertex (besides revisiting a at the end); the result is (a, c, b, d, e, a) — a length-18 minumum TSP tour!

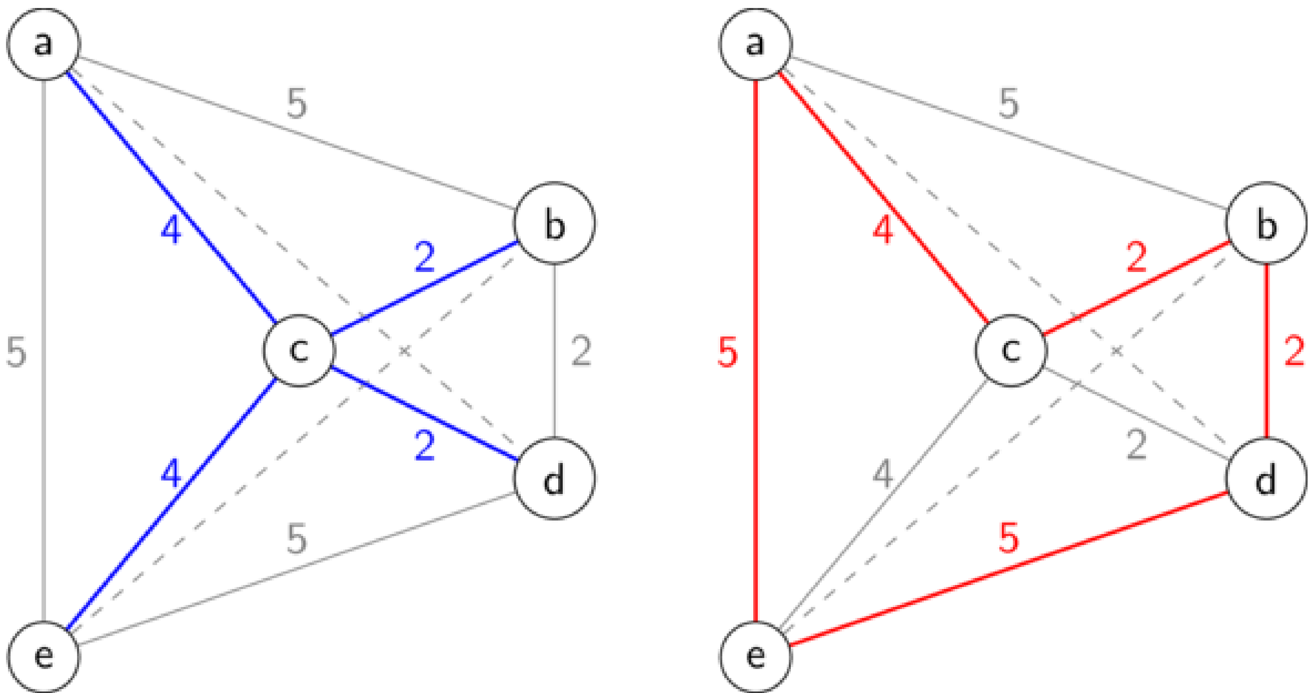

Figure 11: Approximating Metric TSP with MSTs

Of course, we won't always be quite so lucky to end up with the optimal solution, but how bad could this be? Well, we know the size of the MST is less than the optimal TSP tour, so the original cycle from step (2) could be at most two times the optimal (since every edge in the MST is repeated). And then because this is a *metric TSP*, taking the "shortcuts" on step (3) can only make the cycle shorter. Therefore this algorithm always returns a TSP tour whose length is at most 2 times the optimum length.

This is eerily similar to the factor-2 approximation algorithm we saw for vertex cover by using a maximal matching. It's kind of awesome that these two hard problems can't be solved exactly in polynomial-time, but can be approximated very quickly. This isn't always the case of course; the original TSP problem (without triangle inequality), for example, can't be approximated to *any* constant factor in polynomial-time!

7.4 Greedy TSP

Our final approach for solving TSP is by using greedy algorithms. Of course, not every greedy algorithm is created equal! Here are two ideas:

- **Nearest neighbor**: Start an any arbitrary vertex and create a path, at each step choosing the next vertex that is closest to the current endpoint and hasn't been included in the path yet. At the end, just connect the endpoints of the path to get a cycle.
- **Smallest good edge**: Repeatedly add the shortest edge that only touches at most one edge already added. This will again result in a path, and we finish by connecting its endpoints.

Notice that both of these greedy algorithms require that every possible edge in the graph exists (this is called a *complete* graph). In class we saw how each greedy algorithm works on our example graph, resulting in the following two sub-optimal solutions:

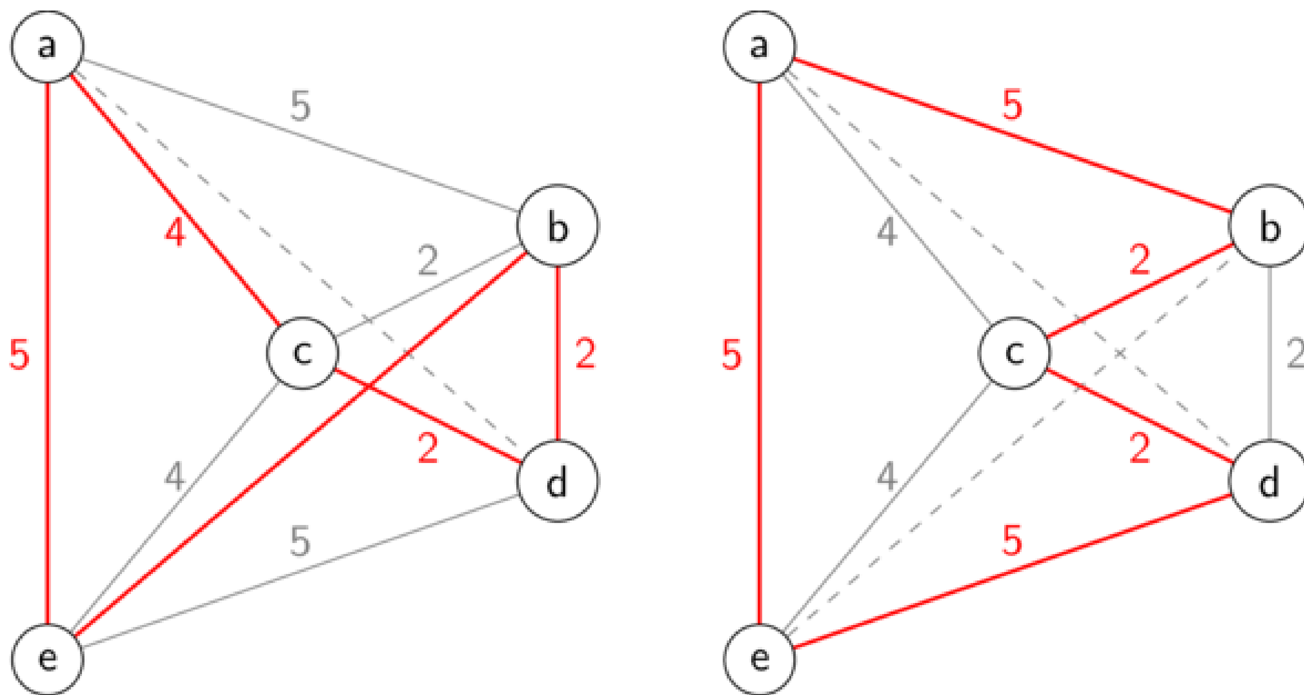

Figure 12: Greedy TSP algorithms

Of course these greedy approaches don't give optimal solutions, but they will find candidates very quickly.

7.5 Iterative Refinement

Once we have an almost-optimal solution, from the MST-based approximation or from a greedy approach, we don't have to stop at that! There's a general approach called *iterative refinement* that takes a sub-optimal solution and tries to make it better, by changing some small part.

The iterative refinement technique that gets used most commonly for TSP is called "2-OPT" and it essentially works by taking any *quadrilateral* (4-node loop) that appears in the graph, and swapping two edges of the quadrilateral

that are in the current TSP tour, for two edges that are not. That is, we take two edges (w, x) and (y, z) that are in the current tour, and we replace them with (w, y) and (x, z).

To see how this fits into the whole big picture, consider the whole tour, which must look like

$(w, x, \text{PATH1}, y, z, \text{PATH2}, w)$

where PATH1 goes from x to y and PATH2 goes from z back to w. What the 2-OPT refinement does is switch those two edges, which also necessitates reversing PATH1 into PATH1-REV, which goes from y to x, so we get:

$(w, y, PATH1 - REV, x, z, PATH2, w)$

Voilà! The cool thing is, we can do this for any pair of edges in the current path. The "OPT" part of 2-OPT is that we choose the two edges to swap that will give the *most* improvement.

For example, in the TSP tour produced by the "nearest neighbor" greedy strategy above, we can swap the edges (a, c) and (b, d) (shown on the left in red) with the blue edges in the TSP tour on the right, which happens to be optimal.

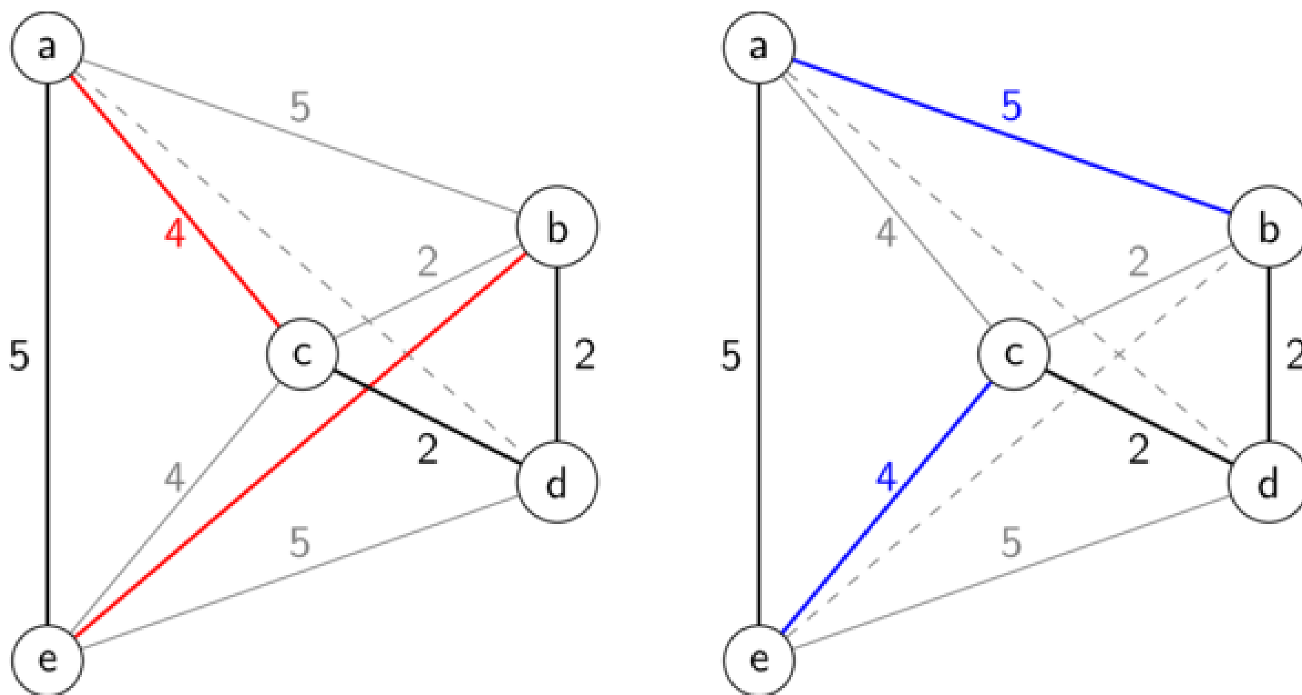

Figure 13: 2-OPT refinement

So now we have a better algorithm to solve TSP:

1. Get a TSP tour using the MST approximation or a greedy algorithm
2. Refine it with 2-OPT
3. Keep repeating (2) until no 2-OPT refinement can improve the cost

Each 2-OPT refinement only needs to examine every pair of edges in the tour to see what the best pair to swap will be. Since there are $|V|$ edges in the tour, the cost of each 2-OPT step is $\Theta(|V|^2)$ — polynomial time! Step (1) obviously costs polynomial-time too, and yet we know that this algorithm doesn't produce the optimum TSP solution in polynomial-time. So what's going on? One of two things must be true: Either

- We can have an exponential chain of 2-OPT refinements before finding the optimum, or
- It's possible to have a tour which can't be improved by any 2-OPT move, but yet which is not optimal.

In fact, one of these two properties must hold for *any* iterative refinement technique for a hard problem that has no polynomial-time solution. Can you figure out which one it is for 2-OPT?