When you have a problem on generic graphs, the top principle is that: design the graph (that can be used with a classic algorithm), instead of inventing a new algorithm from scratch. Why so? First, designing a novel graph algorithm is hard, so it’s best to avoid it. Next, classical algorithms often have good implementation from standard libraries, reducing implementation effort. Below, we’ll illustrate this principle via several examples.

**Warmup: graphs with node weights.** Suppose that we are given a directed graph in which nodes are associated with non-negative cost, and we want to find a shortest path from nodes x to y. One might be tempted to modify Dijkstra’s algorithm, but this is a bad approach. Instead, one can create another edge-weighted graph to use Dijkstra’s algorithm, by setting the weight of the edge \((i, j)\) as the cost of node \(j\).

![Graph with node weights](image.png)

**Figure 8.1:** A directed graph with node cost (left), and its edge-weighted version (right).

**Another Example: Maximum Spanning Tree.** Suppose that we are given a weighted undirected graph \(G = (V, E, w)\)—assuming for simplicity that the weights here are non-negative—and want to find a spanning tree of maximum weight. (An application of this problem is to find a “backbone” of a computer network with the highest total quality; the weight \(w(u, v)\) models the quality of the link between node \(u\) and node \(v\).)

One might think of going back to tweak the Prim/Kruskal algorithms for Maximum Spanning Tree but this approach is not recommended.

Instead, let’s try to turn this back to the classic Minimum Spanning Tree problem. How? Recall that maximizing \(f(x)\) is the same as minimizing \(-f(x)\). So let’s define a new weight function \(w'\) such as \(w'(e) = -w(e)\) for every edge \(e \in E\), and let \(G' = (V, E, w')\). Now, a minimum spanning tree in \(G'\) is a maximum spanning tree in \(G\) and vice versa. So far so good. However, we suddenly realize that the weights in \(G'\) are negative. Will Prim/Kruskal mess up with negative weights as Dijkstra? Can we update the weights in \(G'\) to make them non-negative without affecting the minimum spanning trees, so that we can safely run Prim/Kruskal without any fear?

How to make negative weights non-negative? (Multiplying \(-1\) is a valid answer, but this gets us back to the original problem, namely finding the Maximum Spanning Tree of \(G\).) One possible approach is to add a big positive number \(C\) to every weight. Specifically, let \(C = \max\{w(e) \mid e \in E\}\), namely \(C\) is the heaviest weight in \(G\). If we update \(w'(e) = w'(e) + C\) for every \(e \in E\) then \(G'\) becomes a graph of non-negative weights. In this process, the cost of any (not necessarily minimum) spanning tree \(T\) of \(G'\) is added an amount of...
(n − 1)C, where n = |V|, since T has exactly n − 1 edges. Thus this edge update doesn’t affect the minimum spanning trees in G′. See Figure 8.2 for an illustration. The process above also implicitly suggests that Prim/Kruskal work well with negative weights.

Figure 8.2: Graph transformation for Maximum Spanning Tree. On the left is the original graph G, and on the right is the graph G′ with updated non-negative weight. A weight w′(e) in G′ is obtained from a weight w(e) in G via w′(e) = −w(e) + 8. A maximum spanning tree in G corresponds to a minimum spanning tree in G′.

It’s also instructive to ask if we can use the same trick for Dijkstra to handle graphs of negative weights. Unfortunately the answer here is No. The reason is that when we add a cost C to each weight, we change the shortest path in the graph. See Figure 8.3 below for an illustration.

Figure 8.3: A counter-example for Dijkstra’s algorithm. On the left is the original graph G, and on the right is a graph G′ with updated non-negative weight. A weight w′(e) in G′ is obtained from a weight w(e) in G via w′(e) = w(e) + 2. The shortest path from A to D in G is A → B → E → D, but the shortest path from A to D in G′ is A → C → D.

Yet Another Example: Water Supply. Consider the following problem. In a city there are n houses, each of which is in need of a water supply. It costs B[i] dollars to build a well at house i, and costs C[i, j] to build a pipe between houses i and j. A house can receive water if either there’s a well built there or there is some path of pipes to a house with a well. Design an algorithm to find the minimum amount of money needed to supply every house with water. See the left picture in Figure 8.4 for an example. There you can,
say dig wells at houses 1, 3, 4 and build a pipe from house 1 to house 2 with total cost $9 + 1 + 12 + 3 = 25$. However, the optimal solution is to dig a well at house 4, and build pipe from house 4 to house 1, from house 1 to house 2, and from house 1 to house 3, with total cost $1 + 5 + 3 + 6 = 15$.

The difficulty of the problem above is in modeling its graph. One can immediately see that we should use a node $v_i$ to represent house $i$, and the weight of edge $(v_i, v_j)$ is $C[i, j]$. But when it gets to the cost $B[k]$, things start to be difficult. Where should we put this information? When I taught this in class, some students told me that we should represent $B[k]$ as the weight of node $v_k$. However, as mentioned above, placing weights on nodes is a bad approach. Instead, we should find an edge $e$ whose end point is $v_k$ to place the weight $B[k]$ there. Fine, but what’s the other end point of $e$? It seems incorrect to use any other node in the graph, so we should create a new node for this endpoint. The naive approach is to create a dedicated node $u_k$ for $B_k$ and let $e = \{u_k, v_k\}$.

What’s wrong with the naive approach? First, it’s unclear what’s the meaning of the node $u_k$, so even if we can somehow solve the problem, we don’t fully understand our solution yet. If this philosophical musing appears moot to you, let’s discuss a real obstacle here. Our given problem involves picking a subset of edges with the minimum cost, so using Minimum Spanning Tree seems natural. However, because the node $u_k$ has only one incident edge (namely $\{u_k, v_k\}$), we are forced to pick this edge in our Minimum Spanning Tree. In other words, in every house we have to build a well! The naive approach gives us a completely wrong solution.

Let’s now try to resolve the philosophical question above, namely the meaning of $u_k$. In creating the edge $\{u_k, v_k\}$, we are trying to create a passage from house $k$ to something. What would digging a well at house $k$ possibly lead to? Water, of course. So $u_k$ is the modeling of water source, and $B[k]$ models the cost of connecting house $k$ to water source. But now it seems wrong to use several nodes $u_1, \ldots, u_n$ to model the same water source. Instead, let’s use a single node $v_0$ to model it. So we would create edge $\{v_0, v_k\}$ for every $k$, with weight $B[k]$.

Now the modeling seems to fit well. If we want water at a house, either we should directly connect it with the water source $v_0$, or create a path that eventually leads to the water source. See Figure 8.4 for an illustration. The solution is a minimum spanning tree of this graph.