Approximation

This section considers two algorithms that do not offer an accurate solution to the posed problem, but the solution they give can be an adequate *approximation*. If we look for the maximum value of some function $f(x)$, an algorithm that returns a result that is always within a factor of c of the actual value (which may be hard to find) is called a *c-approximation* algorithm.

Approximation is especially useful with *NP*-complete problems, which do not have known polynomial algorithms. In other cases, approximation is used to find solutions more efficiently than by completely solving the problem, sacrificing some accuracy in the process. For example, a $O(\log n)$ 2-approximation algorithm may be more useful for large inputs than an $O(n^3)$ precise algorithm.

Traveling Salesman

To perform a formal analysis, we need to formalize somewhat the *traveling salesman* problem referred to earlier. We are given a graph with a weight function w that assigns the graph's edges positive values—you can think of this weight function as the distance between cities. The weight function satisfies the triangle inequality, which definitely holds if we keep to the analogy of traveling between cities on a Euclidean surface:

For all vertices x, y, z $\quad w(x, y) + w(y, z) \geq w(x, z)$

The task, then, is to visit every vertex in the graph (every city on the salesman's map) exactly once and return to the starting vertex (the company headquarters), making sure the sum of edge weights along this path is minimal. Let $wOPT$ be this minimal weight. (The equivalent decision problem is *NP*-complete, as we have seen.)

The approximation algorithm proceeds as follows. First, construct a *minimal spanning tree* (MST) for the graph. Let $wMST$ be the tree's total weight. (A minimal spanning tree is a subgraph that touches every vertex, does not have cycles, and has the minimum total edge weight of all such trees.)

We can assert that $wMST \leq wOPT$, because $wOPT$ is the total weight of a *cyclic* path that visits every vertex; removing any edge from it produces a spanning tree, and $wMST$ is the total weight of the *minimum* spanning tree. Armed with this observation, we produce a 2-approximation to $wOPT$ using the minimum spanning tree as follows:

1. Construct an MST. There is a known $O(n \log n)$ greedy algorithm for this.
2. Traverse the MST from its root by visiting every node and returning to the root. The total weight of this path is $2wMST \leq 2wOPT$.
3. Fix the resulting path such that no vertex is visited more than once. If the situation in Figure 9-4 arises—the vertex y was visited more than once—then fix the path by removing the edges (x, y) and (y, z) and replacing them with the edge (x, z). This can only *decrease* the total weight of the path because of the triangle inequality.

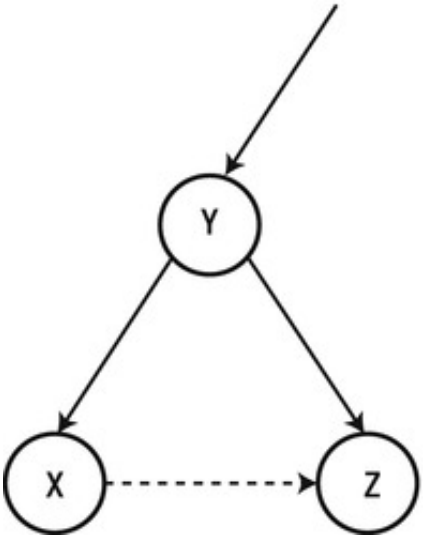

Figure 9-4. Instead of traversing Y twice, we can replace the path (X, Y, Z) with the path (X, Z)

The result is a 2-approximation algorithm, because the total weight of the path produced is still, at most, double the weight of the optimal path.

Maximum Cut

We are given a graph and need to find a *cut*—a grouping of its vertices into two disjoint sets—such that the number of edges crossing between the sets (crossing the cut) is *maximal*. This is known as the *maximum cut* problem, and solving it is quite useful for planning in many engineering fields.

We produce a very simple and intuitive algorithm that yields a 2-approximation:

1. Divide the vertices into two arbitrary disjoint sets, *A* and *B*.
2. Find a vertex *v* in *A* that has more neighbors in *A* than in *B*. If not found, halt.
3. Move *v* from *A* to *B* and go back to step 2.

First, let *A* be a subset of the vertices and *v* a vertex in *A*. We denote by $\deg_A(v)$ the number of vertices in *A* that *v* has an edge with (i.e. the number of its neighbors in *A*). Next, given two subsets *A*, *B* of the vertices, we denote by $e(A, B)$ the number of edges between vertices in two different sets and by $e(A)$ the number of edges between vertices in the set *A*.

When the algorithm terminates, for each *v* in *A,* it holds that $\deg_B(v) \geq \deg_A(v)$—otherwise the algorithm would repeat step 2. Summing over all the vertices, we obtain that $e(A, B) \geq \deg_B(v_1) + \ldots + \deg_B(v_k) \geq \deg_A(v_1) + \ldots + \deg_A(v_k) \geq 2e(A)$, because every edge on the right hand side was counted twice. Similarly, $e(A, B) \geq 2e(B)$ and, therefore, $2e(A, B) \geq 2e(A) + 2e(B)$. From this we obtain $2e(A, B) \geq e(A, B) + e(A) + e(B)$, but the right hand side is the total number of edges in the graph. Therefore, the number of edges crossing the cut is at least one-half the total number of edges in the graph. The number of edges crossing the cut cannot be larger than the total number of edges in the graph, so we have a 2-approximation.

Finally, it is worth noting that the algorithm runs for a number of steps that is linear in the number of edges in the graph. Whenever step 2 repeats, the number of edges crossing the cut increases by at least 1, and it is bounded from above by the total number of edges in the graph, so the number of steps is also bounded from above by that number.