A data structure for bicategories with application to speeding up an approximation algorithm

Philip N. Klein

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-93-47
November 1993

A data structure for bicategories, with application to speeding up an approximation algorithm

Philip N. Klein
Brown University

Abstract

We introduce a data-structure problem on graphs we call the *bicategory* problem, and a data structure that solves it in $O(\sqrt{m}\log m)$ amortized time, where m is the number of edges. We show how this data structure can be used in quickly computing approximately minimum-cost networks. The resulting time bound for the approximation algorithm is $O(n\sqrt{m}\log m)$ where n and m are the number of nodes and edges in the input graph, an improvement over the previously known bound of $O(n^2\sqrt{\log\log n})$ when the input graph is sparse .

1 Introduction

The Steiner tree problem in networks is a classic problem in optimization, proved NP-complete by Karp in his original paper [5]. Given a graph with costs on its edges, and given a subset of the nodes called *terminals*, the goal is to select a minimum-cost connected subgraph spanning all the terminals. Several approximation algorithms have been designed for this problem (a few are [7, 9, 10]), but it was not until 1988 that Mehlhorn [8] devised a way to implement one quickly, in $O(m + n\log n)$ time, where m is the number of edges in the input graph, and n is the number of nodes.

Recently an approximation algorithm was given by Agrawal, Klein, and Ravi for a generalization of the Steiner tree problem [1]. In this generalization, instead of being given a set of terminals along with the graph, one is given a set of pairs of nodes (s_i, t_i). The goal is to select a minimum-cost subgraph (not necessarily connected) in which each s_i is connected to the corresponding t_i. Thus the generalization allows for a more precise specification of the required connectivity.

This algorithm was in turn generalized by Goemans and Williamson [4] to approximately solve a large class of network-design problems. This class includes, for example, the non-fixed point-to-point connection problem, in which one is given two sets of nodes, a source set S and a terminal set T, and the goal is to find a subgraph every component of which has the same number of S-nodes as T-nodes. The algorithm of Goemans and Williamson was extended by Klein and Ravi [6] to handle 2-connectivity requirements, useful in designing networks that are robust against single-link failures.

A straightforward implementation of these algorithms requires $O(n^2 \log n)$ time. Gabow, Goemans, and Williamson [?] show how to implement them (and a more general algorithm that handles higher connectivity requirements) in $O(n^2\sqrt{\log\log n})$ time. In this paper, we show how to implement these algorithms in $O(n\sqrt{m}\log n)$ time, i.e. much more quickly if the input graph is fairly sparse. In particular, we capture the computational bottleneck of the algorithms as a data-structure problem, and we give a data structure for that problem.

The data-structure problem is quite natural. We call it the *bicategory* problem. We start with a directed graph G. Each node of G is assigned to one of a small number of categories. Each edge of G is thereby assigned an ordered pair of categories, the pair consisting of those assigned to the edge's endpoints. We call the set of edges assigned a given pair of categories a *bicategory*. The data-structure problem is to represent the graph and the category assignment so that one can efficiently manipulate a bicategory (increase the costs of all edges in it, find the minimum-cost edge in it), and can also efficiently change the category assigned to a node (and thereby implicitly change the bicategories of its incident edges). One also must be able to contract a given edge, coalescing its endpoints.

We show that the time to carry out the algorithm of Goemans and Williamson is dominated by a series of $O(n)$ such data-structure operations. Moreover, we describe a data structure supporting these operations that requires $O(\sqrt{m}\log n)$ amortized time per operation, as long as at least $\sqrt{m}$ operations are executed. (The need for amortization results from the fact that parallel edges may arise.) Since $\sqrt{m} = O(n)$, it follows that one can approximately solve in $O(n\sqrt{m}\log n)$ time the generalized Steiner tree problem, the non-fixed point-to-point connection problem, two-connected versions of these problems, and many other network-design problems.

2 Preliminaries

We describe the data-structure problem more formally. There is an underlying directed graph G with costs on the edges. (The corresponding problem in which G is undirected is a special case of the problem we address.) Each node v of G is assigned to one of C categories. We will assume that C is constant (in the application, $C = 2$). Let $c(v)$ denote the category containing v. Each edge of G is assigned to a bicategory; the bicategory containing the edge uv is $(c(u), c(v))$. We use the variable b to denote a bicategory. Note that the number of bicategories is C^2.

We want to support the following operations:

- DECREASECOST(b, δ), where b is a bicategory and δ is a real number. This operation decreases by δ the cost of all edges in the specified bicategory.
- FINDMIN(b), where b is a bicategory. This operation returns the minimum-cost edge in the specified bicategory.
- CHANGECATEGORY(v, c), where v is a node and c is a category. This operation changes the category of v to c. (All the edges incident to v have their bicategory implicitly changed).
- CONTRACTEDGE(e, c), where e is an edge. This operation coalesces the endpoints of e, assigning the resulting node to category c, and removes the edge e from the graph.

Next we sketch the approximation algorithm of Goemans and Williamson, and show how the data-structure operations described above can be used to implement the algorithm. The implementation of the two-connectivity algorithm of Klein and Ravi is similar.

The input is an undirected graph with costs $c(e)$ on the edges. The algorithm consists of two phases. The second phase can be executed quickly, so we focus on the first phase. The first phase constructs a forest F of candidate solution edges over a series of iterations. In each iteration, the algorithm selects an edge between two distinct components of the current forest F, and adds the edge to the forest, merging the two components into one. Some of the components are designated *active*. A function $f(C)$ indicates whether the component C is active or not: $f(C) = 1$ if C is active. Whether the endpoints of an edge are active affects how the cost of the edge changes over the course of the algorithm. In particular, each node v of the graph has an *age* $d(v) \geq 0$. Initially all the ages are zero, and they are increased during the algorithm. The *reduced cost* $\hat{c}(uv)$ of an edge is defined by $\hat{c}(uv) = c(uv) - d(u) - d(v)$. (Goemans and Williamson do not explicitly define reduced costs.)

The first phase is sketched below.

1. $F \leftarrow \emptyset$.

2. $d(v) \leftarrow 0$ for every node v.

3. While there is an active component of F,

   (a) select the edge uv minimizing $\hat{c}(uv)/(f(C_u) + f(C_v))$, where C_u and C_v are the distinct components containing u and v respectively. Let δ be the minimum value.

   (b) $F \leftarrow F \cup \{uv\}$.

   (c) For every active component C, increase the age of all nodes in C by δ.

There are several things to note about the algorithm. First, no edges with endpoints within the same component are ever examined. In this sense, it is as if the nodes in each component are all coalesced, i.e. that every time an edge is added to F, that edge is contracted in the graph. Second, step 3c, the step in which the ages of nodes are increased, is equivalent to decreasing the reduced costs of some of the edges. In particular, increasing the ages of nodes in active components by δ corresponds to (1) decreasing by δ the reduced costs of edges with one endpoint in an active component, and (2) decreasing by 2δ the reduced cost of edges with both endpoints in active components. Third, the edge selection in step 3a can be broken into the following steps: find the minimum-reduced-cost edge e_1 among the edges with one active endpoint, find the minimum-reduced-cost edge e_2 among the edges with two active endpoints, and compare the reduced cost of e_1 to half the reduced cost of e_2.

Using these observations, we can reformulate the algorithm to use our data-structure operations. Every time an edge is added to F, the edge is contracted in the graph, coalescing its endpoints. Hence components in the above formulation correspond to nodes in the new formulation. We therefore refer to active nodes instead of active components. Because the data structure assumes a directed graph, we arbitrarily assign directions to the edges of our input graph. The following algorithm does not depend on the directions of the edges.

1. $F \leftarrow \emptyset$.

2. $\hat{c}(e) \leftarrow c(e)$ for every edge e.

3. Assign each node to one of the two categories, ACTIVE or INACTIVE, according to whether the node is active or not.

4. While there is an active node,

   (a) Select the minimum-reduced-cost edge e_1 from the bicategories (ACTIVE, INACTIVE) and (INACTIVE, ACTIVE). Let $\delta_1 = \hat{c}(e_1)$.

   (b) Select the minimum-reduced-cost edge e_2 from the bicategory (ACTIVE, ACTIVE). Let $\delta_2 = \hat{c}(e_2)/2$.

   (c) If $\delta_1 \leq \delta_2$ then assign $e \leftarrow e_1$ and $\delta \leftarrow \delta_1$. Otherwise, assign $e \leftarrow e_2$ and $\delta \leftarrow \delta_2$.

   (d) $F \leftarrow F \cup \{e\}$.

   (e) Contract the edge e, determining and assigning the category of the resulting node.

   (f) Decrease by δ the reduced costs of edges in the bicategories (ACTIVE, INACTIVE) and (INACTIVE, ACTIVE).

   (g) Decrease by 2δ the reduced costs of edges in the bicategory (ACTIVE, ACTIVE).

The selection step can be implemented using three calls to FINDMIN. Steps 4f and 4g can be executed using a total of three calls to DECREASECOST In step 4e, the selected edge is contracted and the category of the resulting node must be determined. The method for determining that category is application-dependent; it depends on which components are designated active. In specific applications such as the generalized Steiner tree problem and the nonfixed point-to-point problem, this information is easy to maintain; the time required is dominated by other calculations. We therefore do not address this issue here.

We see that the algorithm can be implemented using a constant number of data-structure operations per iteration, for a total of $O(n)$ data-structure operations.

3 The bicategory data structure

We implement the data structure using a two-level priority queue for each bicategory. That is, for each bicategory b there is a priority queue $Q(b)$. The elements of $Q(b)$ are priority queues $Q(v,b)$, one for each node v of the graph. The elements of $Q(v,b)$ are the edges in bicategory b that are *assigned* to node v. Each edge is assigned to exactly one of its two endpoints; we describe the assignment later.

3.1 decreaseCost and findMin

Each priority queue Q and each edge has an associated label, which is a real number. The labels are maintained so that the cost of an edge is the sum of its label, the label of the queue $Q(v,b)$ that contains it, and the label of the queue $Q(b)$ that contains its queue. These labels enable us to quickly update the costs of edges. This idea is borrowed from [3, 2] For example, to implement decreaseCost(b,δ), we simply decrease by δ the label of $Q(b)$. The keys of the elements in $Q(v,b)$ are the labels of the edges. Hence the lowest-labeled edge in $Q(v,b)$ can be found in $O(\log n)$ time. The queues $Q(v,b)$ are themselves assigned keys because these queues are elements of queues $Q(b)$. The key assigned to $Q(v,b)$ is its label plus the label of its minimum-label element. Hence the minimum-cost edge in a bicategory b can be found by a two-level search in $O(\log n)$ time; one uses a findMin operation on $Q(b)$ to find the queue $Q(v,b)$ containing the least-cost edge, and then another findMin operation on $Q(v,b)$ to find this edge.

3.2 changeCategory

Now we address the implementation of changeCategory(v,c). This operation changes the bicategory of edges incident to v. Let c_0 be the old category of v. For each category c', we must take the edges uv in bicategory (c',c_0) and reassign them to bicategory (c',c), and, symmetrically, we must take the edges vu in bicategory (c_0,c') and reassign them to bicategory (c,c'). Fix a category c', and let us consider the edges uv (the edges vu are treated similarly).

Let $b_1 = (c',c_0)$ and let $b_2 = (c',c)$. We must gather the edges uv currently in bicategory b_1 and move them to bicategory b_2. Some of these edges are assigned to v, and are therefore in $Q(v,b_1)$. Moving these edges is easy; we move the entire queue, updating its label to reflect the difference between the label of $Q(b_1)$ and that of $Q(b_2)$. We must also move those edges uv not assigned to v. In order to make this possible, we maintain a *extra-list*, extra(v, b_1), of the edges in b_1 that are incident to v but not assigned to v. To move these edges, we run down the list and move each edge uv from $Q(u, b_1)$ to $Q(u, b_2)$, updating its label to reflect the difference between the labels on $Q(u, b_1)$ and $Q(b_1)$ and the labels on $Q(u, b_2)$ and $Q(b_2)$. The time required for these moves is $O(\log n)$ times the length of extra(v, b_1). Thus it is desirable to keep the extra-lists short. We show that a proper assignment of edges to nodes ensures that every extra-list has length at most $2\sqrt{m}$ in an amortized sense. That is, we ensure that all but $2\sqrt{m}$ of the edges on an extra-list can be immediately discarded since they will never be output by a FINDMIN operation. Hence the time required for moving these edges is $O(\sqrt{m}\log n)$ plus $O(logn)$ times the number of edges discarded. Since at most m edges are discarded throughout the life of the data structure, the amortized time for discarding edges is $O(\sqrt{m}\log n)$ as long as at least $\sqrt{m}$ data-structure operations are executed. Thus CHANGECATEGORY takes $O(\sqrt{m}\log n)$ amortized time.

We now address the rule for assigning edges to nodes. Say the outdegree of a node is *high* if it exceeds $\sqrt{m}$. Note that since there are only m edges in total, the number of nodes with high outdegree is at most $m/\sqrt{m} = \sqrt{m}$. Our rule for assigning edges to nodes is as follows. When we consider the edge uv, if u has high outdegree, we assign uv to u. Otherwise, we assign it to v.

We now show that this rule ensures that extra-lists are short. Consider extra(v, b). It consists of some edges outgoing from v and some edges incoming to v. First we count the outgoing edges. If v has high outdegree, all its outgoing edges are assigned to it, so none appear on its extra-list. If v does not have high outdegree, it has at most $\sqrt{m}$ outgoing edges. Next we count the incoming edges on extra(v, b). Any such edge was not assigned to v, and hence must be outgoing from some node w with high outdegree. For each such node w, let e_w be the cheapest such edge. All other edges from w to v in extra(v, b) can be discarded they would never be chosen over e_w by a FINDMIN operation. The total number of nondiscarded edges in extra(v, b) from w to v is at most the number of nodes w with high outdegree, which is at most $\sqrt{m}$.

3.3 CONTRACTEDGE

Finally we address the implementation of CONTRACTEDGE(uv, c). First we change the categories of u and v to c. Then we merge the queues $Q(u, b)$ and $Q(v, b)$ for the various bicategories b, and similarly merge the extra-lists. If one of the endpoints of uv had high outdegree and the other didn't, we need to move edges from the second's extra-list to the first's queue. Finally, if neither endpoint had high outdegree, it may be that the new node resulting does; in this case, we must move the outgoing edges from the extra-list to the appropriate queue. Each of the last two cases involves removing edges from the queues in which they appear, and adding them to the extra-lists of the corresponding nodes. The number of edges that must be thus moved is $O(\sqrt{m})$.

Acknowledgements

Thanks to Adam L. Buchsbaum and R. Ravi for helpful discussions.

References

[1] A. K. Agrawal, P. N. Klein, and R. Ravi, "When trees collide: An approximation algorithm for the generalized Steiner problem on networks," *Proc. 23rd ACM Symp. on Theory of Computing* (1991), pp. 134-144.

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA (1974).

[3] D. Cheriton and R. E. Tarjan, "Finding minimum spanning trees, " *SIAM J. Comput. 5* (1976), pp. 724-742.

[4] M. X Goemans and D. P. Williamson, "A general approximation technique for constrained forest problems," *Proc., 3rd ACM Symposium on Discrete Algorithms*, pp. 307-316.

[5] R. M. Karp, "Reducibility among combinatorial problems", in R. E. Miller and J. W. Thatcher (eds.), *Complexity of Computer Computations.* Plenum Press, New York (1972) pp. 85-103.

[6] P. N. Klein and R. Ravi, "When cycles collapse: a general approximation technique for constrained two-connectivity problems," with R. Ravi, to

appear in *3rd Symposium on Integer Programming and Combinatorial Optimization*, 1993.

[7] L. Kou, G. Markowsky, and L. Berman, "A fast algorithm for Steiner trees", *Acta Informatica, vol. 15* (1981), pp. 141-145.

[8] K. Mehlhorn, "A faster approximation algorithm for the Steiner problem in graphs" *Information Processing Letters, vol. 27(3)* (1988), pp. 125-128.

[9] J. Plesnik, "A bound for the Steiner tree problem in graphs," *Math. Slovaca, vol. 31* (1981) pp. 155-163.

[10] H. Takahashi, and A. Matsuyama, " An approximate solution for the Steiner problem in graphs," *Math. Japonica*, vol. 24 (1980) pp. 573-577.