The Union-Find Partition in Evolutionary Operators for Spanning Trees Encoded as Lists of Edges

Bryant A. Julstrom
Department of Computer Science
St. Cloud State University
720 Fourth Avenue South
St. Cloud, MN 56301
julstrom@eeyore.stcloudstate.edu

Abstract

Students generally encounter the union-find partition in an efficient implementation of Kruskal's algorithm, which identifies a minimum spanning tree in a weighted undirected graph. The union-find structure can also be applied in other algorithms involving spanning trees. In particular, lists of edges can encode candidate spanning trees in evolutionary algorithms that search spaces of spanning trees. Effective and efficient crossover and mutation operators for this coding use union-find partitions of the target graph's vertices, and exhibit additional applications of the union-find structure.

1 Introduction

A *graph* is a non-empty set V of *vertices* and a set E of pairs of vertices called *edges*. If the pairs in E are not ordered—that is, if no direction is associated with edges—the graph is *undirected*, and if a function $w : E \to R^+$ associates a numerical value with each edge, the graph is *weighted*. Figure 1(a) shows a weighted undirected graph on ten vertices. Undergraduate courses in data structures, discrete math, and algorithms introduce students to graphs, graph representations, and graph algorithms.

A *spanning tree* of an undirected graph G is a maximal acyclic subgraph of G; a spanning tree connects all of G's vertices and contains no cycles. In a weighted graph, the weight of a spanning tree is the sum of the weights of its edges. A standard problem is to find a spanning tree of minimum weight, called a minimum spanning tree (MST), on a weighted undirected graph. Students generally investigate and implement two eponymous polynomial-time greedy algorithms that find a MST, due to Kruskal (1956) and to Prim (1957). Figure 1(b) shows a minimum spanning tree on the graph of Figure 1(a); it has weight 408.

The *union-find partition* is an abstract data type whose values are partitions of a finite set of objects and whose three operations initialize a partition to singletons, identify the subset in a partition to which an object belongs, and form the union of two subsets. Kruskal's algorithm, which builds a MST by repeatedly including in it the lowest-weight edge that connects two currently unconnected groups of vertices, motivates a discussion of the union-find partition and its efficient implementation.

Given a weighted undirected graph G, many interesting and useful problems seek a minimum-weight spanning tree on G that satisfies additional constraints. For example, the degree-constrained minimum spanning tree problem (d-MSTP)

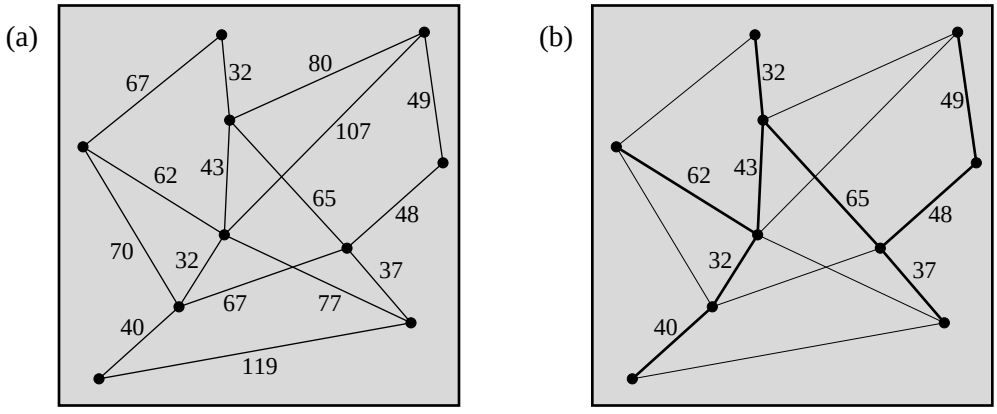


Figure 1: (a) A weighted undirected graph on ten vertices. (b) A minimum spanning tree on the graph; it has weight 408.

bounds the number of edges in which each vertex may participate. Finding an unconstrained minimum spanning tree is computationally easy, but many such problems, including the d-MSTP, are NP-hard. It is unlikely that polynomial-time algorithms exist to solve these problems exactly, and for approximate solutions we turn to heuristics, including evolutionary algorithms.

An *evolutionary algorithm* (EA) is a probabilistic search procedure inspired by biological evolution. It maintains a population of data structures, called *chromosomes*, that encode candidate solutions to the target problem instance, and it assigns each chromosome a numerical *fitness* indicating the quality of the solution it represents. The EA selects chromosomes to survive or to reproduce probabilistically: the more fit chromosomes are more likely to be selected. Operators inspired by genetic recombination and mutation generate novel chromosomes. *Crossover* (or *recombination*) combines the genetic material of two parents; *mutation* randomly modifies one parent chromosome. Successive generations of such chromosomes yield codings of better and better solutions. Bäck, Fogel, and Michalewicz (2000) provide an excellent, current introduction to evolutionary algorithms.

The most important design choice confronting the author of an evolutionary algorithm is the coding by which the EA's chromosomes will represent candidate solutions. An effective coding of spanning trees for evolutionary search is simply as lists of edges, as in recent EAs for the d-MSTP (Raidl, 2000) and the rectilinear Steiner problem, which seeks a shortest spanning tree made up of horizontal and vertical line segments (Julstrom, 2001). When lists of edges encode spanning trees, effective genetic operators can be based on union-find partitions of the target graph's vertices.

This paper describes these operators, which provide novel examples of the use of this well known data structure. The following sections review the union-find partition and its efficient implementation, describe the list-of-edges coding of spanning trees in evolutionary algorithms, and describe crossover and mutation operators for this coding that use union-find partitions.

2 The Partition Abstract Data Type

Given a set U of objects, a *partition* P of U is a set of disjoint subsets of U whose union is equal to U. We say that the subsets are mutually exclusive and exhaustive; every element of U belongs to exactly one subset in P. For example, if $U = \{0, 1, 2, \ldots, 9\}$, then $P = \{\{0, 6, 7\}, \{1, 4\}, \{2, 5, 8, 9\}, \{3\}\}$ is a partition of U.

The partition abstract data type contains only three operations: `Create(P)` initializes the partition `P` to singletons; `Find(P,x)` identifies the subset in `P` to

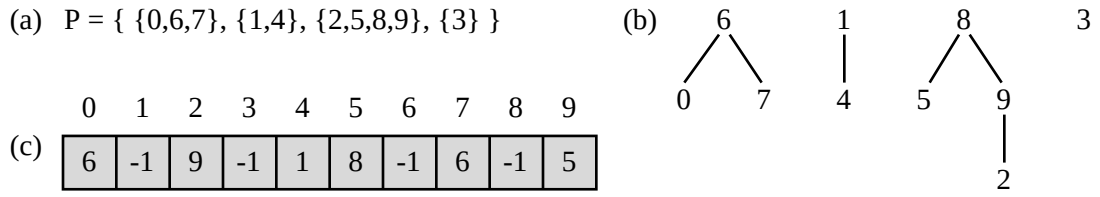


Figure 2: (a) A partition P of the set $U = \{0, 1, 2, \ldots, 9\}$; (b) trees that represent the partition's subsets; (c) and an array that represents the trees. In the array, -1 marks roots.

which the element `x` belongs; and `Union(P,x,y)` modifies `P` by forming the union of the subsets identified by `x` and `y`.

Any implementation of sets can be extended to represent a partition, but when the elements of U can index an array, we have a concise implementation of the partition ADT that elegantly supports the three operations. In it, a partition is represented by a forest in which each tree represents one subset, and the entire forest is represented in an array whose indices *and* elements are members of U. In particular, each element's entry in the array identifies its *parent* in the tree that represents its subset. Marker values indicate roots. Figure 2 illustrates a partition, the trees that represent its sets, and the array that represents the trees.

It is straightforward to implement the three operations in this representation. Initialization sets all the array's components to the root marker, representing a forest of singletons. The element at a tree's root names its subset; to identify the subset that contains an element x, follow the parent indices from x to the root of its tree and report that root. To form the union of two distinct subsets, make the root of one a child of the root of the other. Figure 3 illustrates a `Union()` operation on the partition of Figure 2.

Each `Union()` operation makes two calls to `Find()`, which traverses a path in one tree. In the worst case, the sequence of $(n-1)$ `Union()`s required to merge n singletons into one set takes time that is $\Theta(n^2)$, but two changes to the operators' implementations can improve this time significantly. *Weight balancing* modifies the union operator so that it always makes the root of the *smaller* tree a child of the root of the larger. This is easy to implement if the elements of U are non-negative integers and roots are marked with -1 times the size of the subset rather than with

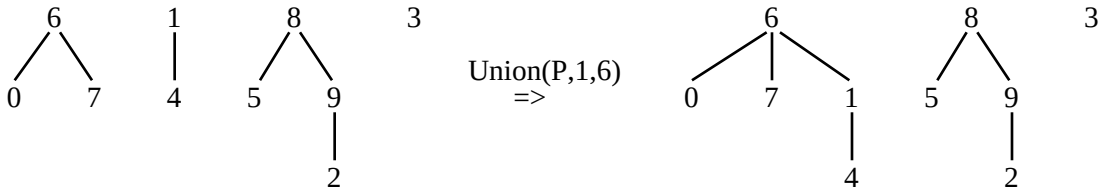


Figure 3: The effect on the partition of the operation `Union(P,1,6)`; the subsets/trees identified by 1 and 6 are merged by making 1 a child of 6.

a fixed marker value. *Path compression* extends `Find(x)` to retrace the path from `x` to its root. On the second traversal, it redirects all the parent indexes to point directly to the root, thus shortening paths and reducing the number of steps in subsequent traversals.

Tarjan (1975) and more recently Harfst and Reingold (2000) have shown that, with weight balancing and path compression, a sequence of m `find()` and `union()` operations on a partition of n elements ($m \geq n \geq 1$) requires time that is $O(m\ \alpha(m, n))$, where $\alpha(m, n)$ is an inverse of Ackermann's function $A(x, y)$:

$$\alpha(m, n) = \min\{i > 0 \mid A(i, \lfloor m/n \rfloor) > \lg n\}.$$

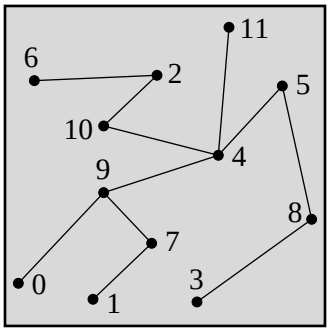
As Ackermann's function grows quickly, this function grows slowly; in general, we can regard it as never exceeding four. Thus the time of a sequence of `find()`s and `union()`s is essentially linear in their number.

3 Spanning Trees in Evolutionary Algorithms

The algorithms of Prim and Kruskal require only polynomial time to identify minimum spanning trees in weighted undirected graphs, but searching for optimal spanning trees that satisfy constraints is often NP-hard. Examples of such problems include the d-MSTP and RStP already mentioned as well as the minimum degree spanning tree problem (Crescenzi, *et al.*, 1999). the optimum communication spanning tree problem (Hu, 1974; Johnson, Lenstra, and Rinooy Kan, 1978), the minimum facility location problem (Guha and Khuller, 1998), the fixed-charge transportation problem (Gottlieb and Eckert, 2000), and many others. Finding good approximate solutions to problems such as these requires heuristic search of the space of spanning trees on the target graph. Evolutionary algorithms are often effective in such searches.

For evolutionary search, spanning trees can be encoded simply as lists of edges: the entry $(9, 4)$ represents the edge connecting vertices 9 and 4, and a list of $(n - 1)$ such entries represents a spanning tree on a graph's n vertices. Figure 4 shows a spanning tree on twelve vertices and its representation as a list of edges.

In an evolutionary algorithm, the genetic operators of crossover and mutation act on chromosomes that encode candidate solutions to generate novel chromosomes and explore the search space. Crossover should replicate in the structures that offspring represent features of the structures that their parents represent; when chromosomes encode spanning trees, offspring should be composed of parental edges. Mutation should make a small change in a chromosome that corresponds to a small change in the structure the chromosome represents. The next two sections describe operators for spanning trees encoded as lists of edges that satisfy these



{ (6,2), (0,9), (2,10), (7,9), (3,8), (7,1),
(4,11), (4,5), (10,4), (9,4), (5,8) }

Figure 4: A spanning tree on twelve vertices and its representation as a list of edges.

requirements. Each uses a union-find partition of the graph's vertices as it builds an offspring spanning tree.

4 Crossover

Without loss of generality, we can assume that the graph's vertices are labeled with the integers from 0 to $(n-1)$. Crossover uses a union-find partition of these integers to keep track of connected vertices as it builds an offspring spanning tree from two parents. This outline describes the algorithm:

1. Initialize the partition to singletons.
2. Sort the edges in each parent. Compare two edges by comparing their lower-numbered vertices. If these match, compare their higher-numbered vertices.
3. Scan the two sorted lists of edges. Copy identical edges into the offspring. In the partition, merge components joined by each edge included in the offspring. Copy all other edges into a separate list.
4. Select randomly from the remaining edges, including in the offspring those that join previously unconnected components, until a spanning tree is completed.

Figure 5 illustrates the crossover of two spanning trees. In problems such as the degree-constrained minimum spanning tree problem, the inclusion of parental edges may violate the problem's constraints. In this case, the last step may require the inclusion of non-parental edges; the partition indicates which edges may join the spanning tree.

The sorting steps require time that is $O(n \log n)$, but they simplify what follows. In step 4 in the worst case, the parent trees have no edges in common and every edge

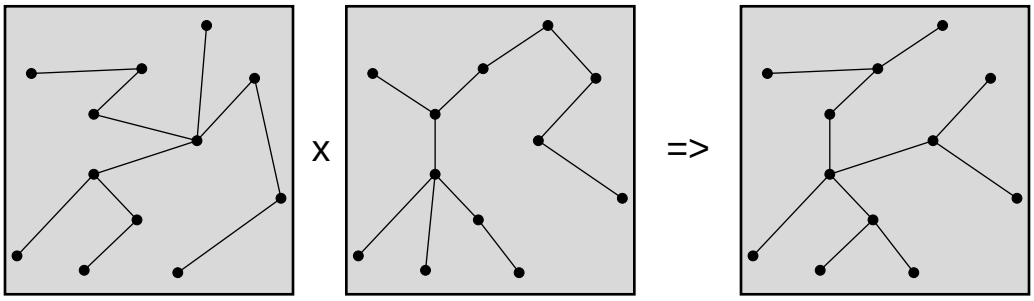


Figure 5: Crossover of two parental spanning trees according to the union-find-based crossover operator.

must be examined to build the offspring, so the number m of calls to `find()` and `union()` is $2 \cdot 2(n-1) + (n-1) = 5(n-1)$. Thus the time for this step is $O(5(n-1), \alpha(5(n-1), n))$, and the sorting steps determine crossover's time: $O(n \log n)$.

Some time can be saved by attaching to each chromosome a flag that indicates whether or not its edges have been sorted; if a chromosome is a parent more than once, the sorting step need not be repeated.

5 Mutation

Mutation deletes a random edge from a parent chromosome and replaces it with a new random one that reconnects the tree. The modified list of edges is the offspring. Mutation uses a union-find partition of the graph's vertices to identify the two components created when an edge is removed. It creates a new edge by choosing one point from each component, according to this outline:

1. Initialize the partition to singletons.
2. Copy the parent chromosome into the offspring.
3. Select one edge at random in the offspring and remove it.
4. Scan the remaining edges. Use the partition to identify the two components of the resulting graph.
5. Scan the partition to list the vertices in each component.
6. Select one point at random from each component. Write this edge into the offspring.

Figure 6 illustrates the union-find-based mutation operator.

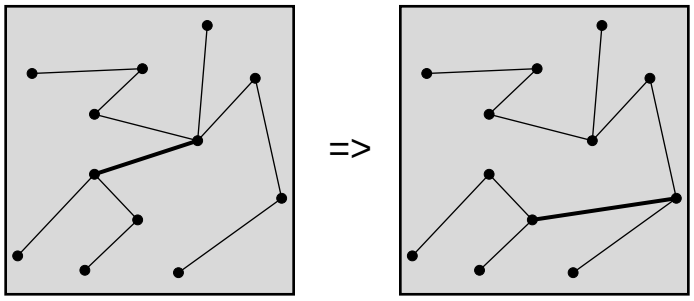


Figure 6: Mutation of a spanning tree. An edge is chosen at random, removed, and replaced with another edge that reconnects the tree.

Mutation calls `find()` twice and `union()` once for each of $(n-1)$ edges, then `find()` n times to identify the two components that must be joined, so $m = 3(n-1) + n = 4n - 6$. Thus the time of mutation is $O(4n-6, \alpha(4n-6, n))$, just more than linear.

6 Conclusion

While CS students usually encounter the union-find partition in the context of Kruskal's algorithm for minimum spanning trees, it can also be used in efficient implementations of other algorithms that manipulate spanning trees. This paper has presented two such: a crossover operator and a mutation operator for spanning trees encoded as lists of edges in evolutionary algorithms that search spaces of spanning trees. This paper has summarized the union-find partition and its operations, introduced a coding of spanning tree as lists of edges for evolutionary search, and described efficient implementations of a crossover and a mutation operator for this coding. The operators use union-find partitions of integers that label vertices. They should provide effective search for the computationally hard problems that evolutionary algorithms address.

References

Thomas Bäck, David B. Fogel, and Zbigniew Michalewicz (Eds.). (2000). *Evolutionary Computation 1: Basic Algorithms and Operators*. Bristol and Philadelphia: Institute of Physics Publishing.

P. Crescenzi, V. Kann, R. Silvestri, and L. Trevisan (1999). Structure in approximation classes. *SIAM Journal on Computing*, **28**, 1759–1782.

J. Gottlieb and C. Eckert (2000). A comparison of two representations for the fixed charge transportation problem. In Kalyanmoy Deb, Günther Rodolph, Xin Yao, and Hans-Paul Schwefel (Eds.), *Parallel Problem Solving from Nature – PPSN VI*, 330–335. Berlin: Springer-Verlag.

S. Guha and S. Khuller (1998). Greedy strikes back: Improved facility location algorithms. In *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms*, 649–657. ACM-SIAM.

Gregory C. Harfst and Edward M. Reingold (2000). A potential-based amortized analysis of the union-find data structure. *SIGACT NEWS*, **31**(3), 86–95.

T. C. Hu (1974). Optimum communication spanning trees. *SIAM Journal of Applied Mathematics*, **30**, 188–195.

D. S. Johnson, J. K. Lenstra, and A. H. G. Rinooy Kan (1978). The complexity of the network design problem. *Networks*, **8**, 279–285.

Bryant A. Julstrom (2001). Encoding rectilinear Steiner trees as lists of edges. In *Proceedings of the 2001 ACM Symposium on Applied Computing*. ACM Press, 2001. March 11–14, Las Vegas, Nevada.

J. B. Kruskal (1956). On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematics Society*, **7**(1), 48–50.

R. C. Prim (1957). Shortest connection networks and some generalizations. *Bell System Technical Journal*, **36**, 1389–1401.

Günther R. Raidl (2000). An efficient evolutionary algorithm for the degree-constrained minimum spanning tree problem. In *Proceedings of the 2000 Congress on Evolutionary Computation*, 104–111. Piscataway, NJ: IEEE Press.

R. E. Tarjan (1975). Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, **22**, 215–225.