Finding t-dominators in Linear Time

Ruijie Fang

Preliminaries. Let $G = (V, E)$ be a directed graph of n vertices and m edges. A t-dominator vertex is a vertex v such that v is on every $s - t$ path in G; in other words, every path from s to t must pass through v. We describe an algorithm based on incremental search for finding the set of all t-dominators in G in $O(m + n)$-time.

The algorithm. For each vertex $v \in V$, we maintain an extra bit $u(v)$ that tells us if the vertex is usable; a directed edge (u, v) is usable if and only if u is usable; if an edge (u, v) is unusable, then we treat it as deleted. Initially, mark all vertices as usable. Let $s = u_1u_2...u_k = t$ be an arbitrary st-path. Let the vertices u_i be denoted as a *path vertex*. Let the edges on this st-path be denoted as a *path edge*. We can find this path using BFS or DFS in $O(m + n)$-time. During the search, we maintain a global variable hi which records the largest i for which a path vertex u_i is visited. Initially, set $hi = 1$ and mark s as the first (trivial) t-dominator. Afterwards, mark all the vertices $u_iu_{i+1}, 1 \leq i < k$ as unusable. Next, start searching from s, ignoring all unusable edges and only visiting the usable edges. After the search terminates, mark the vertex u_2 as usable; and examine hi; if $hi = 2$, then 2 is a t-dominator, otherwise, 2 can be ruled out as a t-dominator.

We can iteratively repeat the search process incrementally at every path vertex u_i; we augment the search by going through the newly available vertex $u_{i-1}u_i$ for $i > 1$, and visit every usable outgoing edge of u_i, if unexplored. Before visiting each path vertex u_{i+1} after the search terminates, we examine hi; if $u_{i+1} = hi$ at any stage of the algorithm, then u_{i+1} will be marked as a t-dominator, otherwise it is ruled out.

Each vertex is examined at most once by the aforementioned procedure, hence the algorithm works in $O(m + n)$-time. The incremental search may be implemented using either breath-first or depth-first search; we simply maintain an array $visited[\cdot]$ that is marked as true whenever we visited a vertex, so as to not use it again in the future.

Proof of correctness. Via induction on each iteration of the algorithm. For the base case, $hi = 1$ and $s = u_1$ is a trivial t-dominator.

Next, assume we have found the t-dominators among the path vertices $u_1...u_i$. After marking the vertex u_{i+1} as usable, there are two possibilities to consider for u_{i+1}:

1. $hi \neq i + 1$; in this case, we can reach some u_l further down the path with $l > i$ from some previous path vertex u_j with $j < i$, without using u_iu_{i+1} as an edge. This implies that $u_1u_2...u_j \to^* u_lu_{l+1}...u_k$ is a valid st-path; this path does not include u_i as a vertex, hence u_i is not a t-dominator.

2. $hi = i + 1$; in this case, the deepest path vertex reachable from s without using any path edges after u_{i-1} is exactly u_i. Assume, for sake of contradiction, that u_i is not a t-dominator. Then by definition there exists a u_i-free st-path $u_1...u_j \to^* u_l...u_k$, with $j < i < l$; such a path coincides with the initial st-path on the first j and the last $k - l$ vertice. However, this necessarily means the vertex u_l is reachable from u_j without using $u_i u_{i+1}$ as an edge, implying that $hi = l \neq i + 1$; contradiction.

Application to SAT solving. A practical backtracking algorithm for solving SAT that is quite popular nowadays is the CDCL (Conflict-Driven Clause Learning) Solver. The core idea of the CDCL algorithm is to iteratively maintain, at each recursion level, an *implication graph*. An implication graph is a directed graph where vertices are literal assignments and the directed edges denote implications between literal assignments. The idea is, of course, that whenever we assign a value to some literals in a certain clause, it might implicate others during *boolean constraint propagation* (BCP). Then we can record such implications in the graph.

If BCP returns a conflict, then we return from graph-building and analyze the implication graph. The graph will be directed, with multiple sources (the vertices we pick arbitrarily at the start of each iteration without implications) and a single sink (the conflict node). Here comes the next great idea in CDCL: doing *non-chronological backtracking*; we don't just backtrack to the *second-deepest BFS level* of the implication graph. Instead, we seek to remove an entire part of the subgraph enclosing the sink node, defined by a cut. Now how do we find such a cut? Our cut has to have certain nice properties, such as being minimal. What we need is a *unique implication point* (UIP), which is essentially a t-dominator that is closest to sink (but not the sink) on the implication graph. We can now use the above algorithm to compute the UIP in linear time.

One might argue that using a nice linear-time algorithm as a subroutine to speed-up an exponential-time backtracking algorithm is purposeless. It might not be as purposeless as it seems; for instance, it is conventional knowledge that much of the time during SAT solving is spent on doing BCP, so even practical improvements to BCP performance that does not change the theoretical worst-case bounds could lead to a good, practical speed-up.

Acknowledgements. This was originally presented as an initial problem in Tarjan's graph algorithms seminar at Princeton University. Two solutions were thought up, both running in linear time: this one, and one based on a shortest-path characterization, the idea due to Antonio Molina Lovett and refined by Tarjan. I had discussions with Kexin Jin and Henry Tang on this problem.