

# Simple Linear Time Approximation Algorithm for Betweenness

Yury Makarychev

Toyota Technological Institute at Chicago

## Abstract

We present a simple linear time combinatorial algorithm for the Betweenness problem. In the Betweenness problem, we are given a set of vertices and *betweenness* constraints. Each betweenness constraint of the form  $x \rightsquigarrow \{y, z\}$  requires that vertex  $x$  lies between vertices  $y$  and  $z$ . Our goal is to find a linear ordering of vertices that maximizes the number of satisfied constraints. In 1995, Chor and Sudan designed an SDP algorithm that satisfies half of all constraints if the instance is satisfiable. Our algorithm has the same approximation guarantee.

## 1 Introduction

In this paper, we present a simple combinatorial algorithm for the Betweenness problem. In the Betweenness problem, we are given a set  $V$  of  $n$  vertices and a set  $\mathcal{C}$  of  $m$  *betweenness* constraints. Every betweenness constraint is a pair  $(x, S)$  where  $x \in V$  and  $S$  is a 2 element subset of  $V \setminus \{x\}$ . We denote this constraint by  $x \rightsquigarrow S$ . We say that a vertex ordering  $\psi : V \rightarrow \{1, \dots, n\}$  satisfies a betweenness constraint  $x \rightsquigarrow \{y, z\}$  if either  $\psi(y) < \psi(x) < \psi(z)$  or  $\psi(z) < \psi(x) < \psi(y)$  (that is,  $\psi(x)$  lies *between*  $\psi(y)$  and  $\psi(z)$ ). Our goal is to find a vertex ordering that maximizes the number of satisfied constraint.

In 1979, Opatrny [3] proved that the decision version of the problem is NP-hard. Chor and Sudan [2] showed that moreover it is MAX SNP hard. The approximability of the problem crucially depends on whether the instance is completely satisfiable or not. There is a trivial 3-approximation algorithm, which just chooses a random ordering. In general, one cannot get a better approximation factor as was recently shown by Charikar, Guruswami, and Manokaran [1] (assuming the Unique Games Conjecture). However, Chor and Sudan [2] proved that if an instance is satisfiable, it is possible to satisfy at least half of all constraints. Their approximation algorithm is based on semidefinite programming and thus it is not very fast. In this paper, we design a very simple combinatorial algorithm with running time  $O(m + n)$  that achieves the same approximation guarantee.

## 2 Overview

In this section, we give a brief overview of our algorithm. Let us say that a vertex  $u$  is a free vertex if there is no constraint of the form  $u \rightsquigarrow \{v, w\}$  in  $\mathcal{C}$ . Note that if the instance is satisfiable then there is at least one free vertex since the first vertex w.r.t. the optimal ordering must be a free vertex. Moreover, it is clear that we can find a free vertex in  $G$  efficiently.

Our algorithm uses the following recursive approach.

1. First, it chooses a free vertex  $u$ .
2. Then it removes  $u$  and all constraints that involve  $u$  from the instance.
3. It recursively solves the obtained sub-instance and gets an ordering  $\psi$ .
4. Finally, it inserts  $u$  either at the beginning or at the end of  $\psi$ ; the algorithm chooses the better among these two options.

Since every sub-instance of a completely satisfiable instance is completely satisfiable, the algorithm is always able to find a free vertex at the first step.

We prove by induction on the size of the instance that the algorithm satisfies at least half of all constraint if the input instance is satisfiable. Indeed, let  $\mathcal{C}_1 \subset \mathcal{C}$  be the set of constraints involving  $u$ . By the induction hypothesis, at step 3, the algorithm finds an ordering  $\psi$  that satisfies at least half of constraints in  $\mathcal{C} \setminus \mathcal{C}_1$ . On the other hand, every constraint  $c$  in  $\mathcal{C}_1$  is of the form  $x \rightsquigarrow \{u, y\}$  (for some  $x$  and  $y$ ). Thus either if we put  $u$  before all other vertices or if we put  $u$  after all vertices, we will satisfy  $c$ . So the better of these two options satisfies at least half of constraints in  $\mathcal{C}_1$ . Therefore, the algorithm finds an ordering that satisfies at least half of constraints in  $\mathcal{C}$ .

In the next section, we will formally analyze this algorithm, and we will show how to implement it in linear time. In particular, we will explain how to find a free vertex  $u$  in  $V$  in amortized constant time. For the sake of analysis, it will be convenient to state this algorithm as an iterative algorithm. We will call the order in which the algorithm chooses free vertices (at step 1) a “*relaxed ordering*.” First, in Lemma 3.6, we will show how to find a relaxed ordering in linear time. Then, in Lemma 3.7, we will show how to find a solution that satisfied at least half of all constraints given a relaxed ordering.

### 3 Algorithm

**Definition 3.1.** Consider a constraint  $x \rightsquigarrow \{y, z\}$ . Let us call vertex  $x$  the middle vertex of the constraint; let us call vertices  $y$  and  $z$  the end vertices of the constraint.

**Definition 3.2.** Suppose that we are given an instance of the Betweenness problem. We say that an ordering  $\varphi : V \rightarrow \{1, \dots, n\}$  is a relaxed ordering or relaxed solution if for every constraint  $x \rightsquigarrow \{y, z\}$  in  $\mathcal{C}$ ,  $\varphi(y) < \varphi(x)$  or  $\varphi(z) < \varphi(x)$ .

Note that if an ordering  $\psi$  satisfies all betweenness constraints in  $\mathcal{C}$  then  $\psi$  is a relaxed solution. Hence, the following claim holds.

**Claim 3.3.** Every satisfiable instance has a relaxed solution.

**Definition 3.4.** Given a subset of vertices  $A \subset V$ , we say that a vertex  $x \in A$  is free in  $A$  if there is no constraint of the form  $x \rightsquigarrow \{y, z\}$  with  $y, z \in A$ .

**Claim 3.5.** If an instance has a relaxed solution then there is a free vertex in every non-empty subset  $A$  of vertices.

*Proof.* Consider a relaxed solution  $\varphi$ . Let  $x = \arg \min_{x \in A} \varphi(x)$  (the leftmost vertex in  $A$  w.r.t. the ordering  $\varphi$ ). Consider a constraint  $x \rightsquigarrow \{y, z\}$  in  $\mathcal{C}$ . Since  $\varphi$  is a relaxed solution, either  $\varphi(y) < \varphi(x)$  or  $\varphi(z) < \varphi(x)$ . Therefore, either  $y \notin A$  or  $z \notin A$ . We conclude that there is no constraint  $x \rightsquigarrow \{y, z\}$  in  $\mathcal{C}$  with  $y, z \in A$ .  $\square$

**Lemma 3.6.** *There is a linear time algorithm that given a Betweenness instance finds a relaxed solution  $\varphi$  if a relaxed solution exists. In particular, the algorithm finds a relaxed solution  $\varphi$  if the instance is satisfiable.*

*Proof.* The algorithm first finds a vertex  $u_1$  that is free in  $V$ . Then it finds a vertex  $u_2$  that is free in  $V \setminus \{u_1\}$ . At step  $i \in \{1, \dots, n\}$ , the algorithm finds a vertex  $u_i$  that is free in  $V \setminus \{u_1, \dots, u_{i-1}\}$ . It returns ordering  $\varphi$  defined by  $\varphi(u_i) = i$ .

Note that by Claim 3.5, the algorithm is able to find a vertex  $u_i$  that is free in  $V \setminus \{u_1, \dots, u_{i-1}\}$  at step  $i$  if there is a relaxed solution. Let us check that the algorithm returns a relaxed solution  $\varphi$ . Indeed consider a constraint  $x \rightsquigarrow \{y, z\}$ . Let  $i = \varphi(x)$ . Note that  $x = u_i$  is a free vertex in  $V \setminus \{u_1, \dots, u_{i-1}\} = \{u_i, \dots, u_n\}$ . Thus either  $y \notin \{u_i, \dots, u_n\}$  or  $z \notin \{u_i, \dots, u_n\}$ . That is, either  $\varphi(y) < i = \varphi(x)$  or  $\varphi(z) < i = \varphi(x)$ . Hence  $\varphi$  is a relaxed ordering.

We now show how to implement the algorithm in linear time. The algorithm is presented in Figure 1. The algorithm finds vertices  $u_1, \dots, u_n$  in a loop (the **while** loop in Figure 1) — in iteration  $i$  it finds vertex  $u_i$ . The algorithm keeps the list of vertices  $\mathcal{F}$  that are free in  $V \setminus \{u_1, \dots, u_{i-1}\}$ . Consider iteration  $i$  of the loop. Let us say that a constraint  $x \rightsquigarrow \{y, z\}$  is active if neither of its end vertices lies in  $\{u_1, \dots, u_i\}$ . The active degree of a vertex  $x$  is the number of active constraints of the form  $x \rightsquigarrow \{y, z\}$ . Note that a vertex  $x \in V \setminus \{u_1, \dots, u_{i-1}\}$  is free in  $V \setminus \{u_1, \dots, u_{i-1}\}$  if and only if the active degree of  $x$  equals 0. For every constraint  $c_j \in \mathcal{C}$ , the algorithm has a flag  $a_j$  that says whether the constraint is active or passive. The algorithm also stores the active degree  $d_x$  of every vertex  $x$ . Finally, it keeps the list  $\mathcal{L}_y$  of constraints of the form  $x \rightsquigarrow \{y, z\}$  for every vertex  $y$  (every constraint  $x \rightsquigarrow \{y, z\}$  from  $\mathcal{C}$  appears in exactly two lists,  $\mathcal{L}_y$  and  $\mathcal{L}_z$ ).

During the initialization step, the algorithm marks all vertices as active, initializes all lists  $\mathcal{L}_y$ , computes active degrees  $d_x$  of all vertices, and pushes all vertices of active degree 0 on stack  $\mathcal{F}$ . This step takes time  $O(m + n)$ .

In the main loop, the algorithm pops a vertex  $x$  from  $\mathcal{F}$  and sets  $\varphi(x) = i$ . Then it goes over all constraints in  $\mathcal{L}_x$ , marks them as inactive, and updates the degrees of vertices that appear in them. Finally, it adds vertices of degree 0 to  $\mathcal{F}$ . Thus  $\mathcal{F}$  contains all free vertices in  $\{u_i, \dots, u_n\}$  at every iteration  $i$ .

The outside **while** loop is executed once for every vertex  $x$ ; thus it is executed at most  $n$  times. The inner **for** loop is executed at most twice for every constraint  $c_j$ ; thus it is executed at most  $2m$  times. Therefore, the running time of the algorithm is  $O(m + n)$ .  $\square$

**Lemma 3.7.** *There is a linear time algorithm that given a relaxed solution  $\varphi$  finds an ordering  $\psi$  that satisfies at least  $m/2$  constraints.*

*Proof.* It will be more convenient for us to construct first a map  $\psi$  from  $V$  to some set of  $n$  consecutive integers  $\{l, l + 1, \dots, l + n - 1\}$  rather than to the set  $\{1, \dots, n\}$ . Once we find such  $\psi$ , we will define a valid solution by  $\psi'(x) = \psi(x) - (l - 1)$ .

Let the rank of a constraint  $x \rightsquigarrow \{y, z\}$  be the minimum of  $\varphi(y)$  and  $\varphi(z)$ . Note that since  $\varphi$  is a relaxed solution  $\varphi(x)$  is strictly greater than the rank of the constraint  $x \rightsquigarrow \{y, z\}$ . Let  $\mathcal{C}_i \subset \mathcal{C}$  be the set of constraints of rank  $i$ . Let  $u_i = \varphi^{-1}(i)$  for every  $i \in \{1, \dots, n\}$ .

First the algorithm computes all sets  $\mathcal{C}_i$  in linear time. Then it goes over all vertices  $u_i$  from  $u_n$  to  $u_1$  and defines  $\psi(u_i)$ . The algorithm assigns values to  $\psi(u_n), \dots, \psi(u_1)$  in such way that after iteration  $t$ ,  $\psi$  maps vertices  $u_n, u_{n-1}, \dots, u_{n+t-1}$  to  $t$  consecutive integer numbers. Specifically, at iteration  $t = 1$ , the algorithm lets  $\psi(u_n) = 1$ . At iteration  $t > 1$ , it lets either

---

**Figure 1** The function finds a relaxed solution for a given instance.

---

**input:** number of vertices  $n$ ,

a set of constraints  $\mathcal{C} = \{c_1, \dots, c_m\}$  on  $V = \{1, \dots, n\}$

**output:** a relaxed solution  $\varphi : V \rightarrow \{1, \dots, n\}$ .

**data structures:**

stack  $\mathcal{F}$  of capacity  $n$  (*stores all free vertices*)

array of integers  $d = (d_1, \dots, d_n)$  ( $d_x$  is the active degree of vertex  $x$ )

array of flags  $a = (a_1, \dots, a_m)$  ( $a_j$  indicates whether  $j$  is active or not)

a list  $\mathcal{L}_x$  for every vertex  $x$

**begin**

set all  $a_x = \text{active}$

scan all constraints in  $\mathcal{C}$ :

add each constraint  $c_j$  with end vertices  $y$  and  $z$  to lists  $\mathcal{L}_y$  and  $\mathcal{L}_z$

compute the degree  $d_x$  of each vertex  $x$

push all vertices of degree 0 on stack  $\mathcal{F}$

$i = 1$

**while**  $\mathcal{F}$  is not empty **do** //iterate over all free vertices

pop  $x$  from stack  $\mathcal{F}$

$\varphi(x) = i$

$i = i + 1$

**for each**  $j$  **in**  $\mathcal{L}_x$  **such that**  $a_j = \text{active}$  **do**

let  $w$  be the middle vertex of  $c_j$

$a_j = \text{passive}$

$d_w = d_w - 1$

**if**  $d_w = 0$  **then** push  $w$  on stack  $\mathcal{F}$

**end for**

**end while**

**if**  $\varphi$  is defined on all vertices **then**

**return**  $\varphi$

**else**

**return** “There is no relaxed solution.”

**end if**

**end.**

---

1.  $\psi(u_i) = \min(\psi(u_{i+1}), \dots, \psi(u_n)) - 1$ , or
2.  $\psi(u_i) = \max(\psi(u_{i+1}), \dots, \psi(u_n)) + 1$ ,

where  $i = n+1-t$ . We now describe how the algorithm chooses between these two options. Consider a constraint  $c \in \mathcal{C}_i$ . Constraint  $c$  is of the form  $x \rightsquigarrow \{u_i, y\}$  where  $x, y \in \{u_{i+1}, \dots, u_n\}$ . Note that  $\psi(x)$  and  $\psi(y)$  are already defined by the algorithm. Now if  $\psi(x) < \psi(y)$  and the algorithm chooses the *first* option then  $\psi(u_i) < \psi(x) < \psi(y)$ , so  $c$  is satisfied; similarly, if  $\psi(x) > \psi(y)$  and the algorithm chooses the *second* option, then  $\psi(y) < \psi(x) < \psi(u_i)$ , so  $c$  is satisfied. The algorithm scans all constraints in  $\mathcal{C}_i$  and counts the number of constraints with  $\psi(x) < \psi(y)$ . If the number of constraints with  $\psi(x) < \psi(y)$  is greater than  $|\mathcal{C}_i|/2$ , the algorithm chooses the first option; otherwise, it chooses the second option. That guarantees that  $\psi$  satisfies at least  $|\mathcal{C}_i|/2$  constraints in  $\mathcal{C}_i$ . The algorithm performs iteration  $t$  in time  $O(|\mathcal{C}_i| + 1)$ . (The algorithm stores the values of  $l \equiv \min(\psi(u_{i+1}), \dots, \psi(u_n))$  and  $r \equiv \max(\psi(u_{i+1}), \dots, \psi(u_n)) = l + (n - i)$ , and does not compute them in each iteration.)

The algorithm finds a solution  $\psi$  that satisfies at least  $\sum_{i=1}^n |\mathcal{C}_i|/2 = m/2$  constraints. The running time is  $O(\sum_{i=1}^n (|\mathcal{C}_i| + 1)) = O(m + n)$ .  $\square$

Lemma 3.6 and Lemma 3.7 immediately imply Theorem 3.8.

**Theorem 3.8.** *There is a deterministic algorithm that given a satisfiable instance of the Betweenness problem with  $n$  vertices and  $m$  constraints, finds a vertex ordering  $\psi$  that satisfies at least  $m/2$  constraints. The running time of the algorithm is  $O(m + n)$ .*

## References

- [1] M. Charikar, V. Guruswami, and R. Manokaran. *Every Permutation CSP of arity 3 is Approximation Resistant*, to appear in IEEE Conference on Computational Complexity, 2009.
- [2] B. Chor and M. Sudan. *A geometric approach to betweenness*, SIAM Journal on Discrete Mathematics, 11(4):511-523, Nov. 1998.
- [3] J. Opantry. *Total Ordering Problem*, SIAM Journal on Computing, 8(1):111-114, Feb. 1979.