VOLKER KAIBEL MATTHIAS A.F. PEINHARDT

On the Bottleneck Shortest Path Problem

Supported by the DFG Research Center MATHEON in Berlin

ZIB-Report 06-22 (May 2006)

ON THE BOTTLENECK SHORTEST PATH PROBLEM

VOLKER KAIBEL AND MATTHIAS A.F. PEINHARDT

ABSTRACT. The Bottleneck Shortest Path Problem is a basic problem in network optimization. The goal is to determine the limiting capacity of any path between two specified vertices of the network. This is equivalent to determining the unsplittable maximum flow between the two vertices. In this note we analyze the complexity of the problem, its relation to the Shortest Path Problem, and the impact of the underlying machine/computation model.

1. INTRODUCTION

The Bottleneck Shortest Path Problem (BSP) is at the core of a number of network optimization problems. The performance of algorithms for it is sometimes crucial for the running times of algorithms for higher level problems in which it occurs as a subproblem. Two examples are the k–splittable flow problem [4] (where the running time of the underlying BSP algorithm appears as a factor in the worst case running time bound of the presented algorithm) and the Max Flow Problem [2, Chapter 7]. As outlined in [2], the asymptotical worst case behaviour of the Edmonds–Karp algorithm cannot be improved by using a BSP algorithm for finding augmenting paths, but it might still improve the practical performance.

We start with a formal definition of the problem. Let $G = (V, E)$ be a graph, either directed or undirected, with integral edge weights $c_e \in \mathbb{N}$ for all edges $e \in E$. The *capacity* b_p *of a path* p (viewed as a set of edges) is given by $b_p := \min_{e\in p} c_e$. (For the empty path, we define $b_\varnothing = \infty$.) With respect to some fixed *start vertex* vertex $s \in V$, the *bottleneck* b_v *of a vertex* $v \in V$ is $b_v := \max_p b_p$, where p ranges over all (directed) paths starting in s and ending in v. An edge that determines the capacity of a path or the bottleneck of a vertex (i.e., an edge for which the corresponding minimum is attained) is called *critical* for the path or vertex, respectively.

The Bottleneck Shortest Path Problem (BSP) is to determine, for a given graph $G = (V, E)$, edge weights $c_e \in \mathbb{N}$ $(e \in E)$, and a start vertex $s \in V$, the bottleneck b_t of some specified target vertex t. If we are given the bottleneck b_t of some vertex t then we can construct an s–t–path with capacity b_t in linear time by a simple search in the graph obtained by removing all edges of capacity less than b_t.

Viewing the BSP as the problem to find a maximal unsplittable s–t–flow, it may not be too surprising that one has a duality relation via cuts: Given a BSP instance with source s, target t, we have

$$\max_p b_p = \min_q \max_{e\in q} c_e,$$

Supported by the DFG Research Center MATHEON in Berlin.

where p ranges over all s–t–paths and q over all s–t–cuts [8], see also [12, Chap. 8].

Nevertheless, for our purposes it will turn out to be more important that the BSP is closely related to the (Single Source) Shortest Path Problem (SP) by replacing the capacity b_p of a path p by the length $\ell_p = \sum_{e \in p} c_e$, and, instead of maximizing b_p, minimizing ℓ_p over all paths that connect s and v. Indeed, a slightly modified Dijkstra algorithm solves the BSP in time $O(m + n \log n)$ (i.e., as fast as the SP can be solved this way, see, e.g., [7, Chap. 24.3]).

However, looking a bit more closely to the BSP one soon gets the impression that it should be solvable much easier (and faster?) than the SP. The most intriguing reason is that it is trivial to solve the decision problem "Is there an s–t–path of capacity at least k?" in linear time: Just remove all edges with weights less than k and check whether there is any s–t–path left. This is in contrast to the SP, for which it is not at all clear, why the decision problem "Is there an s–t–path of length at most k?" should be easier than SP itself.

We discuss a linear time algorithm for the BSP in undirected graphs in Section 2. In Section 3, we describe an algorithm for the BSP in directed graphs with m edges that runs in time $O(m \log \log m)$. The role played by the machine models in these discussions is treated briefly in Section 4.

2. The BSP in Undirected Graphs

It seems to be folklore for a long time that a linear time algorithm for the BSP in undirected graphs exists. Nevertheless, we could not find any explicit reference to that in the literature. Therefore, we describe such an algorithm below (see Algorithm 1).

To fix some notation: For a subset S of nodes, we denote by $\delta(S)$ the set of all edges with one end point in S and the other end point not in S. For directed graphs we further distinguish the sets of outgoing edges $\delta^{\text{out}}(S)$ and incoming edges $\delta^{\text{in}}(S)$. We also write $\delta(v)$ for a single vertex v meaning $\delta(\{v\})$ Here are some remarks on Algorithm 1 some of which will also be relevant later.

- We always assume that the graphs are given by means of adjacency lists.
- We can assume that the edges have pairwise distinct weights; e.g., by numbering the edges and breaking ties in favor of the lower numbers.
- It is well known (see, e.g. [5, 6]) that a *median* of m numbers (a number k out of the given numbers with the property that $\lfloor m/2 \rfloor$ numbers are less than or equal to k and $m - \lfloor m/2 \rfloor$ are greater than or equal to k) can be found in linear time, i.e., in $O(m)$ steps (see, e.g., [7, Chap. 9.3]). This can be done both on a pointer machine as well as on a RAM machine.
- *Shrinking* a set of nodes can be done in linear time. More precisely, given a set $S \subseteq V$ of nodes of an (undirected) graph $G = (V, E)$, one can construct in linear time another graph with nodes $(V \setminus S) \cup \{v_{\text{new}}\}$ (where v_{new} represents the shrunken set S), where $v, w \in V \setminus S$ are adjacent if and only if v and w are adjacent in G (in this case,

the edge keeps its weight), and v_{new} and $w \in V \setminus S$ are adjacent if and only if there is some $v \in S$ such that v and w are adjacent in G (in which case the edge receives the biggest weight of any edge connecting S and w in G).

Algorithm 1 A BSP algorithm for undirected graphs

```
1: INPUT: an undirected graph $G = (V, E)$ with edge weights $c_e \in \mathbb{N}$ for
   all $e \in E$,
   and source and target vertices $s, t \in V$;
   w.l.o.g. all edge weights are different, and there is an $s$–$t$–path.
2: Initialize Iterationcount $\leftarrow 0$
3: while Iterationcount $< \lceil \log m \rceil$ do
4:   Determine the median value $M$ of the edge weights of the edges cur-
     rently in the graph.
5:   Remove all edges $e$ with small weight $c_e < M$.
6:   if the graph is not $s$–$t$–connected then
7:     Let $V_1, \dots, V_q$ be the connected components.
8:     Reinsert all edges removed in this iteration.
9:     Shrink $V_1, \dots, V_q$.
10:  end if
11:  Iterationcount $\leftarrow$ Iterationcount $+ 1$
12: end while
13: OUTPUT: the last remaining edge as a critical edge
```

The correctness of Algorithm 1 (for undirected graphs) is rather obvious: If in line 6 it turns out that the graph still is s–t–connected, than hiding the edges of weights less than M in line 5 did not affect the optimal solution. Otherwise, in each subgraph to be shrunken in line 9, every vertex can be connected to every other one by a path with capacity at least M; thus the bottleneck of t (being smaller than M) will remain the same in the shrunken graph.

Furthermore, the algorithm runs in time $O(m)$, where $m = |E|$ denotes the number of edges: The crucial observation is that in every iteration of the loop in lines 3–11 half of the edges are removed from the graph, either by shrinking all "thick" edges (in line 9) or by dropping all "thin" edges (in line 5). As all steps inside that loop can be done in linear time in the size of the current graph, the total running time is bounded by

$$O(m + \tfrac{m}{2} + \tfrac{m}{4} + \cdots) = O(m) \ .$$

This shows the following result.

Theorem 1. *Algorithm 1 solves the BSP in undirected graphs with m edges in time $O(m)$.*

As mentioned above, the result described in this theorem apparently has been known before, but it seems that it has not been stated explicitly in the literature.

Comparing this simple algorithm to Thorup's linear time algorithms for SP in undirected graphs [14, 15], we observe that the BSP algorithm even works on simple machine models like comparison machines, while Thorup's algorithm heavily uses the capabilities of the more powerful RAM model. Additionally, Thorup's algorithm utilizes complicated data structures, e.g., multilevel buckets, while the above algorithm for the BSP is very basic.

Of course, one can easily adapt Algorithm 1 to work also for directed graphs. However, in order to guarantee correctness, in line 7 we then have to choose $V_1, \ldots, V_q$ as the *strongly* connected components of the graph (after hiding the "thin" edges in line 5), because we need to ensure that every vertex in some component can be reached from every other vertex of the same component by a *directed* path using "thick" edges only. This yields a correct algorithm. But, unfortunately, we do not achieve a similar bound on the running time, because, in general, the shrinking step in line 9 will not significantly reduce the number of edges, since it needs not to be true that every "thick" edge is contained in some *strongly* connected component. In fact, it is even possible that the shrinking operations in line 9 do not have any effect, because the strongly connected components might well be single vertices only.

Thus, the crucial problem remains to find a linear time algorithm for BSP in directed graphs, or, at least, to find an algorithm that runs faster than the obvious adaption of Dijkstra's method does. We are going to address this issue in the following section.

For the special case of *planar* directed graphs, one obtains a linear time algorithm by adapting ideas of Klein, Rao, Rauch, and Subramanian [10]. They use the fact that planar graphs have small separators, i.e., small node sets that separate the graph. They actually develop a shortest path algorithm for (directed) planar graphs running in linear time, provided that all arc weights/capacities are positive. In case that negative arc lengths occur, they give an $O(n^{4/3} \log nL)$ algorithm, where L is the largest absolute value of the a negative edge-length. Fortunately, we can check planarity in linear time, see, e.g., [13]. The idea of breaking the graph into pieces and following a divide-and-conquer mechanism applies to more than just planar graphs. The key point to obtain fast algorithms is that the pieces should be rather balanced in size, and that the extra work to connect pieces should not be too large. For planar graphs, the latter requirement is met by the fact that planar graphs have $O(\sqrt{n})$-size separators. As pointed out in [10], separators of size $O(n^{1-\epsilon})$ suffice for the application of their algorithm. Thus their algorithm can as well be applied to graphs with bounded genus [3] or graphs with excluded shallow minors [11]. The crucial point here is that the separation can be found in linear time, too.

3. The BSP in Directed Graphs

In this section we present an algorithm that solves the BSP for directed graphs. The algorithm relies on the availability and efficiency of a bucketing structure, for which we need direct addressing (see, e.g, [7, Chap. 11.1]).

The first observation is that we can solve the BSP in directed graphs in linear time (on machines that provide direct addressing) if an ordering of the edge weights is known, see, e.g., [2, Chapter 4]. For edge weights $c \in \mathbb{N}^E$, a *c-ordering* $\ell : E \to \{1, \ldots, m\}$ of the edges has the property that $\ell(e_1) > \ell(e_2)$ implies $c_{e_1} \geq c_{e_2}$. For convenience, we state such an algorithm as Algorithm 2. It is a reformulation of Dijkstra's algorithm. Algorithm 2 takes as input the BSP instance, a c-ordering ℓ of the edges, and an associated value table $T : \ell(E) \to \mathbb{N}$ such that $T(\ell(e)) = c_e$. As the algorithm works with order numbers only, the value table T is used to map back those order numbers to the original capacities.

Algorithm 2 A linear time algorithm for BSP with sorted edge weights

```
1: INPUT: BSP instance with $c$-ordering $\ell$, and value table $T$
2: Initialize empty buckets $B_1, \dots, B_m$
3: Initialize $b(v) \leftarrow 0$ for all $v \in V$, the bucket index of vertex $v$
4: Initialize flags that denote fixed vertex labels, i.e., vertices removed from
   the buckets: $f(v) \leftarrow 0, \forall v \neq s, f(s) \leftarrow 1$
5: for all $sv \in \delta^{\text{out}}(s)$ do
6:   $B_{\ell(sv)} \leftarrow B_{\ell(sv)} \cup \{v\}$
7:   $b(v) \leftarrow \ell(sv)$
8: end for
9: Set $U \leftarrow m$
10: while $U \geq 0$ do
11:   while $B_U \neq \emptyset$ do
12:     Choose $v \in B_U$
13:     $B_U \leftarrow B_U \setminus \{v\}$
14:     $f(v) \leftarrow 1$
15:     if $v = t$ then
16:       STOP: $T(b(t))$ is the bottleneck between $s$ and $t$
17:     else
18:       for all $vw \in \delta^{\text{out}}(v)$ with $f(w) = 0$ do
19:         Calculate $k \leftarrow \min\{b(v), \ell(vw)\}$
20:         if $k > b(w)$ then
21:           $B_{b(w)} \leftarrow B_{b(w)} \setminus \{w\}, B_k \leftarrow B_k \cup \{w\}$, $b(w) \leftarrow k$
22:         end if
23:       end for
24:     end if
25:   end while
26:   $U \leftarrow U - 1$
27: end while
```

Algorithm 3 uses Algorithm 2 in order to solve the BSP (without having a c-ordering at hands). Its correctness follows from the fact that throughout the algorithm, the edge set E' contains an optimal path, whose capacity cannot exceed U. The running time of Algorithm 3 depends on some function $s(w)$, where lines 3-12 altogether need $O(m \log s(m))$ steps. As the number of edges in E' with weights at most U is halved in each iteration, we have $t = O(\frac{m}{s(m)})$. Thus, if we sort N keys in $O(N \log N)$ time, the running time spent in line 13 is $O(\frac{m}{s(m)} \log \frac{m}{s(m)})$. Line 14 thus needs $O(m)$ steps. By setting $s(m) = \log m$ we obtain an $O(m \log \log m)$ algorithm for the BSP in directed graphs.

Algorithm 3 A BSP algorithm for directed graphs

```
1: INPUT: A (possibly directed) graph $G = (V, E)$ with $m = |E|$ and edge
   weights $c_e \in \mathbb{N}$ for all $e \in E$, source and target vertices $s, t \in V$, a
   number $s(m)$;
   w.l.o.g. all edge weights are different, and there is some $s$–$t$–path in $G$.
2: Initialize Iterationcount $\leftarrow 0, E' \leftarrow E, L \leftarrow \min_{e\in E} c_e, U = \max_{e\in E} c_e$
3: while Iterationcount $< \log s(m)$ do
4:    Determine the median $M$ of $\{c_e : e \in E', c_e \le U\}$.
5:    $T := \{e \in E' : c_e \le M\}$, $F := \{e \in E' : c_e > M\}$
6:    if $(V, F)$ is $s$–$t$–connected then
7:       $E' \leftarrow F, L \leftarrow M$
8:    else
9:       $U \leftarrow M$
10:   end if
11:   Iterationcount $\leftarrow$ Iterationcount $+ 1$
12: end while
13: Number the $t$ edges in $\{e \in E' : c_e \le U\}$ according to increasing
    weights: $e_1, \ldots, e_t$
14: Solve the instance by Algorithm 2 with the following $c$-ordering:
    $$\ell(e) = \begin{cases} 1 & \text{if } c_e \le L \\ i+1 & \text{if } e \in E', e = e_i \\ t+2 & \text{if } c_e > U \end{cases}$$
```

If we use the more sophisticated priority queue of [16] that performs sorting N keys in $O(N \log\log N)$ time, we end up with an $O(m \log\log\log m)$ time algorithm for BSP. In general, we have:

Theorem 2. *If A is a sorting algorithm, whose running time is bounded by $O(N \cdot s(N))$, then employing A in Algorithm 3 yields a BSP-algorithm for directed graphs, whose running time is bounded by $O(m \log s(m))$. This holds on every RAM model of computing.*

As proven in [16, Theorem 1.4] sorting N w–bit keys on a RAM in time $N \cdot s(N)$ (with a decreasing function $s(\cdot)$), implies and requires the existence of a monotone priority queue with constant time search for the minimum key and extraction of the minimum key in $s(N)+O(1)$ time. Thus, as long as the fastest algorithm for SP is a Dijkstra–type algorithm utilizing a monotone priority queue (and $s(N) = O(1)$ is impossible), the BSP in directed graphs can be solved faster than SP by Algorithm 3 on graphs with $O(n)$ edges (where n is the number of vertices). Note that for graphs with $\Omega(n \log n)$ edges Dijkstra's algorithm already yields a linear time method for both SP and BSP.

4. Discussion

In this section we briefly comment on machine models relevant for the different results concerning sorting, (monotone) priority queues, and SP.

There are a number of machine models considered in literature, reflecting the evolution of computers as well as the desire to incorporate different

model	allowed operations	further characteristics
Pointer Machine	comparison, conditional jumps, indirect addressing	equivalent to the Turing machine
RAM	standard AC^0 operations: conditional jumps, direct and indirect addressing, comparison, shift, bit–wise Boolean operations, addition, subtraction	storage is divided in w–bit word; constant number of registers; addresses are themselves words (thus $w \geq \log(\text{input size})$)
strong RAM	arbitrary AC^0 operations, i.e., any operation that can be computed in $O(1)$ time on a polynomial sized circuit	many variants with non–standard AC^0 operation set known

Table 1. Summary of capabilities of different computing models

aspects of real world machines. A short survey is given in Table 1. For further introduction, see e.g.[1].

Our Algorithm 3 works in the RAM model of computation, in particular it needs bucket processing or more specifically direct addressing. However, it does not need any sophisticated AC^0 operations, but can be implemented with the standard set of these operations. Thus, depending on the capabilities of a given RAM, we can choose the fastest sorting algorithm for this model. Although not possible on pointer machines, the use of buckets is computationally reasonable, see, e.g., the satisfying computational studies of elaborated codes for SP by Goldberg [9].

The linear time algorithm for SP by Thorup for undirected graphs can be implemented with AC^0 instructions only, although not all of them are considered standard as they are not available on todays hardware. The priority queue of [16] however either uses superlinear space, non–standard AC^0 instructions, or randomization.

Finally, we do not need any restrictions on the word size w of the RAM besides the usual assumption that the input data can be represented in single words, i.e., $\log m, \log U \leq w$. This is due to the fact that no intermediate data exceeds the size of the input data.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The design and analysis of computer algorithms.*, Addison-Wesley Series in Computer Science and Information Processing. Reading, Mass. etc.: Addison-Wesley Publishing Company. X, 470 p., 1974.

[2] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network Flows: Theory, Algorithms, and Applications*, Prentice Hall, New Jersey, 1993.

[3] L. Aleksandrov and H. Djidjev, *Linear algorithms for partitioning embedded graphs of bounded genus*, SIAM Journal on Discrete Mathematics **9**, no. 1 (1996), pp. 129–150.

[4] G. Baier, E. Köhler, and M. Skutella, *On the k-splittable flow problem*, in Algorithms - ESA 2002. 10th annual European symposium, Rome, Italy, September

17-21, 2002. Proceedings, R. M. (ed.) et al., ed., no. 2461 in Lect. Notes Comput. Sci., Berlin, 2002, Springer, pp. 101–113.

[5] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan, *Linear time bounds for median computations.*, in Proc. 4th ann. ACM Symp. Theory Comput., Denver 1972, 1972, pp. 119–124.

[6] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan, *Time bounds for selection.*, J. Comput. Syst. Sci. **7** (1973), pp. 448–461.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms. 2nd ed.*, Cambridge, MA: MIT Press., 2001.

[8] D. R. Fulkerson, *Flow networks and combinatorial operations research*, The American Mathematical Monthly **73** (1966), pp. 115–138.

[9] A. V. Goldberg, *Shortest Path Algorithms: Engineering Aspects*, in Eades, Peter (ed.) et al., Algorithms and computation. 12th international symposium, ISAAC 2001, Christchurch, New Zealand, December 19-21, 2001. Proceedings. Berlin: Springer. Lect. Notes Comput. Sci. 2223, 502-513 , 2001.

[10] P. Klein, S. Rao, M. Rauch, and S. Subramanian, *Faster shortest-path algorithms for planar graphs*, in Proc. 26th ACM Symp. on Theory of Computing, 1994, pp. 27–37.

[11] Plotkin, Rao, and Smith, *Shallow excluded minors and improved graph decompositions*, in SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms), 1994.

[12] A. Schrijver, *Combinatorial Optimization: Polyhedra and Efficiency*, no. 24 in Algorithms and Combinatorics, Springer, 2003.

[13] W.-K. Shih and W.-L. Hsu, *A new planarity test.*, Theor. Comput. Sci. **223**, no. 1-2 (1999), pp. 179–191.

[14] M. Thorup, *Undirected Single Source Shortest Paths in Linear Time*, in 38th Annual Symposium on Foundations of Computer Science, Miami Beach, Florida, USA, oct 1997, pp. 12–21.

[15] M. Thorup, *Floats, integers, and single source shortest paths.*, J. Algorithms **35**, no. 2 (2000), pp. 189–201.

[16] M. Thorup, *On RAM priority queues*, SIAM J. Comput. **30**, no. 1 (2000), pp. 86–109.

Zuse Institute Berlin, Takustr. 7, 14195 Berlin, Germany
E-mail address: [kaibel,peinhardt]@zib.de