Parallel Shortest Paths Using Radius Stepping

Guy E. Blelloch
Carnegie Mellon University
guyb@cs.cmu.edu

Yan Gu
Carnegie Mellon University
yan.gu@cs.cmu.edu

Yihan Sun
Carnegie Mellon University
yihans@cs.cmu.edu

Kanat Tangwongsan*
Mahidol University
International College
kanat.tan@mahidol.edu

ABSTRACT

The single-source shortest path problem (SSSP) with nonnegative edge weights is notoriously difficult to solve efficiently in parallel—it is one of the graph problems said to suffer from the transitive-closure bottleneck. Yet, in practice, the Δ-stepping algorithm of Meyer and Sanders (*J. Algorithms, 2003*) often works efficiently but has no known theoretical bounds on general graphs. The algorithm takes a sequence of steps, each increasing the radius by a user-specified value Δ. Each step settles the vertices in its annulus but can take $\Theta(n)$ substeps, each requiring $\Theta(m)$ work (n vertices and m edges).

Building on the success of Δ-stepping, this paper describes Radius-Stepping, an algorithm with one of the best-known tradeoffs between work and depth bounds for SSSP with nearly-linear ($\tilde{O}(m)$) work. The algorithm is a Δ-stepping-like algorithm but uses a variable instead of a fixed-size increase in radii, allowing us to prove a bound on the number of steps. In particular, by using what we define as a vertex k-radius, each step takes at most $k+2$ substeps. Furthermore, we define a (k,ρ)-graph property and show that if an undirected graph has this property, then the number of steps can be bounded by $O(n/\rho \cdot \log \rho L)$, for a total of $O(kn/\rho \cdot \log \rho L)$ substeps, each parallel. We describe how to preprocess a graph to have this property. Altogether, for an arbitrary input graph with n vertices and m edges, Radius-Stepping, after preprocessing, takes $O((m+n\rho)\log n)$ work and $O(n/\rho \cdot \log n \log(\rho L))$ depth per source. The preprocessing step takes $O(m \log n + n\rho^2)$ work and $O(\rho \log \rho)$ depth, adding no more than $O(n\rho)$ edges.

1. INTRODUCTION

The single-source shortest path problem (SSSP) is a fundamental graph problem that is extremely well-studied and has numerous practical and theoretical applications. For a weighted graph $G = (V, E, w)$ with $n = |V|$ vertices and $m = |E|$ edges, and a source vertex $s \in V$, the SSSP problem with nonnegative edge weights is to find the shortest (i.e., minimum weight) path from s to every $v \in V$, according to the weight function $w\colon E \to \mathbb{R}_+$, which assigns to every edge a real-valued nonnegative weight ("distance"). We assume without loss of generality that the lightest nonzero edge has weight 1, i.e, $\min_{e:w(e)>0} w(e) = 1$. Throughout, we let $L = \max_e w(e)$ denote the heaviest edge in the graph.

In the sequential setting, Dijkstra's algorithm [8] implemented with the Fibonacci heap [10] solves this problem in $O(m + n\log n)$ time. This is the best theoretical running time for general nonnegative edge weights although faster algorithms exist for certain special cases. Thorup [27], for example, gives an $O(m+n)$-time algorithm for undirected graphs when the edge weights are positive integers.

In the parallel setting, the holy grail of parallel SSSP is an algorithm with Dijkstra's work bound (i.e., work efficient) that runs in small depth. Although tens of algorithms for the problem have been proposed over the last several decades, none of the existing algorithms that take the same amount of work as Dijkstra's have polylogarithmic or even $o(n)$ depth. This has led Karp and Ramachandran to coin the term *transitive closure bottleneck* [15].

Apart from the theoretical quest for a polylogarithmic-depth, work-efficient algorithm, one could aim for an algorithm with a high degree of parallelism ($\mathbb{P} = W/D$) that is work efficient or nearly work-efficient. From a performance standpoint, both factors are important because work efficiency is a prerequisite for the algorithm to quickly gain speedups over the sequential algorithm, and the parallelism factor $\mathbb{P}$ indicates how well it will scale with processors.

This observation has fueled the design of SSSP algorithms with substantial parallelism that are nearly work-efficient. Spencer [26] shows that SSSP can be solved in $O(n/\rho \cdot \log n \log(\rho L))$ depth and $O((n\rho^2 \log \rho + m)\log(n\rho L))$ work, using limited path-doubling to determine the shortest-path distances to about ρ vertices in each round. Later, by precomputing about ρ closest neighbors of every vertex, Shi and Spencer [25] solve SSSP in $O(n/\rho \cdot \log n)$ depth, and $O(n\rho^2 \log n \log \rho + m \log n)$ or $O((\rho^3 + m\rho)\log n)$ work; the preprocessing takes $O(n\rho^2 \log n \log \rho + m)$ work and $O(\log n \log \rho)$ depth. We further discuss these results, as well as other related algorithms, in Section 6.

On the empirical side, the Δ-stepping algorithm of Meyer and Sanders [19] works well on many kinds of graphs. The algorithm has been analyzed for random graphs, but no theoretical guarantees are known for the general case. Δ-stepping is a hybrid of Dijkstra's algorithm and the Bellman-Ford algorithm. It determines the correct distances from s in increments of Δ, settling in step i all the vertices whose shortest-path distances are between $i\Delta$ and $(i+1)\Delta$. Within each step, the algorithm resorts to running Bellman-Ford as substeps to determine the distances to such vertices. Altogether, the algorithm performs up to a total of LK/Δ rounds of substeps, where K is the shortest-path distance to the farthest vertex from s.

*Corresponding author.

SPAA '16, July 11-13, 2016, Pacific Grove, CA, USA

 ISBN 978-1-4503-4210-0/16/07…$15.00

DOI: http://dx.doi.org/10.1145/2935764.2935765

Our Contributions

In this paper, we investigate the single-source shortest-path problem for weighted, undirected graphs. We begin by introducing the notion of a (k, ρ)-graph to capture structural properties that enable computing SSSP in a few parallel steps. Informally, a (k, ρ)-graph is a graph in which every vertex can reach its ρ closest vertices within k hops. Based on this notion, our main results are as follows:

THEOREM 1.1 (SHORTEST PATH). *There is an algorithm, RADIUS-STEPPING, that on input a (k, ρ)-graph with m edges and n vertices and a source vertex, computes the shortest path to every vertex in $O(km \log n)$ work and $O(kn/\rho \cdot \log n \log(\rho L))$ depth.*

However, not all graphs are (k, ρ)-graphs as given. We prove that by adding a small number of extra edges ("shortcuts"), every graph can be turned into a $(1, \rho)$-graph at a modest cost:

THEOREM 1.2 (PREPROCESSING). *There is an algorithm that on input an undirected, weighted graph G with n vertices and m edges, computes a $(1, \rho)$-graph out of G in $O(m \log n + n\rho^2)$ work and $O(\rho \log \rho)$ depth by adding at most $n\rho$ extra edges.*

Together, this means that given as input an *arbitrary* undirected, weighted graph with n vertices and m edges, we can make it a $(1, \rho)$-graph in $O(m \log n + n\rho^2)$ work and $O(\rho \log \rho)$ depth. The graph then has $m' = m + n\rho$ edges, whose weight is at most ρL each. Hence, after preprocessing, the shortest path from any given source can be computed in $O((m + n\rho) \log n)$ work and $O(n/\rho \cdot \log n \log(\rho L))$ depth[1].

Compared to a standard Dijkstra's implementation, the algorithm, excluding preprocessing, is work-efficient up to only a $\log n$ factor. Using $\rho = \text{polylog}(n)$, the algorithm offers at least polylogarithmic parallelism. Using $\rho = \Omega(n^{\varepsilon})$, the algorithm has sublinear depth. As far as we know, this is one of the best tradeoffs between work and depth for the problem and has the potential to perform better on real graphs than what the theoretical bounds suggest.

In broad strokes, our algorithm RADIUS-STEPPING (Section 3) can be seen as a hybrid variant of Δ-stepping and Spencer's algorithm, though simpler. It also bears some similarity to Shi and Spencer's algorithm but is again simpler. Like Δ-stepping, it performs Dijkstra's-like steps in the outer loop, finding and settling multiple vertices at a time using (restricted) Bellman-Ford as substeps. Unlike Δ-stepping, however, the algorithm judiciously picks a new step size (Δ) every time. The step size is chosen so that, like in Spencer's algorithm, about ρ new vertices are settled in every step, or otherwise it doubles the distance explored. Yet, unlike Spencer's algorithm, RADIUS-STEPPING does not perform path doubling in every iteration, leading to a conceptually simpler algorithm with fewer moving parts. Ultimately, the steps of RADIUS-STEPPING mirror Dijkstra's execution steps, and the only invariant necessary is a natural generalization of Dijkstra's invariant.

Provided that the input is a (k, ρ)-graph, RADIUS-STEPPING runs in $O(\frac{n}{\rho} \log(\rho L))$ steps, each running up to $O(k)$ substeps, leading to the bounds in Theorem 1.1. For unweighted graphs, it only takes $O(k(m + n))$ work and $O(k\frac{n}{\rho} \log \rho \log^* \rho)$ depth.

As not all graphs are (k, ρ)-graphs as given, Section 4 describes preprocessing strategies to make a graph a (k, ρ)-graph. They involve running restricted Dijkstra's to discover the ρ-closest vertices for all vertices in parallel. To derive a $(1, \rho)$-graph, the preprocessing step effectively adds up to $n\rho$ shortcut edges to the input graph. The number of shortcut edges can be reduced if we use a larger k. Although this saves some memory, RADIUS-STEPPING will have to do slightly more work when computing the shortest paths. For $k > 1$, it is less trivial to generate a (k, ρ)-graph that adds only a small number of edges. We present some heuristics in Section 4 and empirically study their quality in Section 5.

Our experiments study the number of edges added by preprocessing and the number of steps taken by RADIUS-STEPPING. This is a good proxy for measuring the depth of the algorithm experimentally. The results confirm our theory: the number of steps taken (that is, roughly the depth) is inversely proportional to ρ for both the weighted and unweighted cases. They further show that with an appropriate heuristic (such as the dynamic programming heuristic from Section 4.2) and choice of parameters, preprocessing will add a reasonably small number of edges (at most $O(m)$), thereby increasing the total work by at most a constant.

[1]The heaviest edge is technically heavier by a ρ-factor, affecting the $\log(\rho L)$ term only by a constant.

2. PRELIMINARIES

Let $G = (V, E, w)$ be an undirected, simple, weighted graph. We assume without loss of generality that G is a connected, simple graph (i.e., no self-loops nor parallel edges). For $S \subseteq V$, define $N(S) = \cup_{u \in S}\{v \mid (u, v) \in E\}$ to be the ***neighbor set*** of S. During the execution of standard breadth-first search (BFS) or Dijkstra's algorithm, the ***frontier*** is the neighbor set of all visited vertices. Let $d(u, v)$ denote the shortest-path distance in G between two vertices u and v. The ***enclosed ball*** of a vertex u is $B(u, r) = \{v \in V \mid d(u, v) \leqslant r\}$.

A ***shortest-path tree*** rooted at vertex u is a spanning tree T of G (a subtree if disconnected) such that the path distance from the root u to any other vertex $v \in T$ is the shortest-path distance from v to u in G. A graph is ***unweighted*** if $w(e) = 1$ for all $e \in E$ (so $L = 1$).

We assume the standard PRAM model that allows for concurrent reads and writes, and analyze the performance of our parallel algorithms in terms of work and depth [14], where ***work*** W is equal to the total number of operations performed and ***depth (span)*** D is equal to the longest sequence of dependent operations. Nonetheless, all of our algorithms can also be implemented on machines with exclusive writes. An algorithm is ***work efficient*** if it does the same amount of work as the best sequential counterpart, up to constants.

Consider two ordered sets A and B stored in balanced BSTs with $|A| \leqslant |B|$. Recent research work shows that set union and set difference can be computed either in $O(|A| \log |B|)$ work and $O(\log |B|)$ depth [24, 22, 23], or in $O(|A| \log |B|/|A|)$ work and $O(\log |A| \log |B|)$ depth [3, 2]. For this paper, it suffices that both set operations take $O(|A| \log |B|)$ work and $O(\log |B|)$ depth. Also, to split a tree A by a certain key costs $O(\log |A|)$ work and depth [2].

Finally, we need a few definitions for our algorithms:

DEFINITION 1 (HOP DISTANCE). *Let $\hat{d}(u, v)$ be the number of edges on the shortest (weighted) path between u and v that uses the fewest edges.*

DEFINITION 2 (k-RADIUS). *For $u \in V$, the k-radius of u, denoted by $\bar{r}_k(u)$, is $\min_{v \in V, \hat{d}(u,v) > k} d(u, v)$—that is, the closest distance to u which is more than k hops away.*

The k-radii of vertices are widely used in our algorithm analysis. However, computing them can be costly, so instead, we provide the concepts of ρ-nearest distance and (k, ρ)-graph, and our algorithm is based on both the k-radius and ρ-nearest distance of each vertex.

DEFINITION 3 (ρ-NEAREST DISTANCE). *For $v \in V$, the ρ-nearest distance of v, denoted by $r_\rho(v)$, is the distance from v to the ρ-th closest vertex to v.*

Algorithm 1: The RADIUS-STEPPING Algorithm.

Input: A graph $G = (V, E, w)$, vertex radii $r(\cdot)$, and a source vertex s.
Output: The graph distances $\delta(\cdot)$ from s.

```
1  δ(·) ← +∞, δ(s) ← 0
2  foreach v ∈ N(s) do δ(v) ← w(s, v)
3  S_0 ← {s}, i ← 1
4  while |S_{i−1}| < |V| do
5      d_i ← min_{v∈V\S_{i−1}} {δ(v) + r(v)}
6      repeat
7          foreach u ∈ V\S_{i−1} s.t. δ(u) ⩽ d_i do
8              foreach v ∈ N(u)\S_{i−1} do
9                  δ(v) ← min{δ(v), δ(u) + w(u, v)}
10     until no δ(v) ⩽ d_i was updated
11     S_i = {v | δ(v) ⩽ d_i}
12     i = i + 1
13 return δ(·)
```

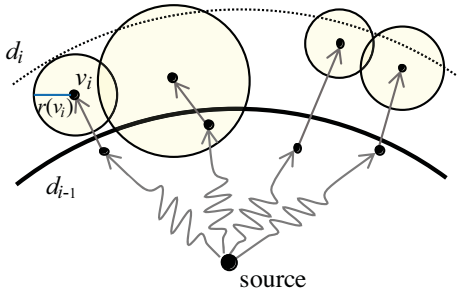


Figure 1: Illustration of a step of RADIUS-STEPPING: For all vertices whose neighbors are within d_i, we pick v_i, the lead node in the i-th step, that minimizes the tentative distance plus the vertex radius of this vertex. The step distance d_i is set to $\delta(v_i) + r(v_i)$.

DEFINITION 4 ((k,ρ)-BALL AND (k,ρ)-GRAPH). *We say that a vertex $v \in V$ has a (k,ρ)-ball if $r_\rho(v) \leqslant \bar{r}_k(v)$. A graph is a (k,ρ)-graph if each vertex in the graph forms a (k,ρ)-ball.*

The following lemma then follows directly from these definitions:

LEMMA 2.1. *For a graph G, if we let $r(v) = r_\rho(v)$ for all $v \in V$ and G is a (k,ρ)-graph, then $r(v) \leqslant \bar{r}_k(v)$ and $|B(v, r(v))| \geqslant \rho$.*

3. THE RADIUS-STEPPING ALGORITHM

This section describes our algorithm for parallel SSSP called RADIUS-STEPPING and discusses its performance analysis. We begin with a high-level description, which we analyze for the number of steps. Following that, we give a detailed implementation, which we analyze for work and depth.

The RADIUS-STEPPING algorithm is presented in Algorithm 1. At a high level, it works as follows: The input to the algorithm is a weighted, undirected graph, a source vertex, and a target radius value for every vertex, given as a function $r : V \to \mathbb{R}_+$. In the algorithm we use $\delta(\cdot)$ to denote the tentative distance. The algorithm has the same basic structure as the Δ-stepping algorithm; both are effectively a hybrid of Dijkstra's algorithm and the Bellman-Ford algorithm. Like Dijkstra's algorithm, they visit vertices in increasing distance from the source s, settling each vertex v—i.e., determining its correct distance $d(s,v)$. However, instead of visiting one vertex at a time, the algorithms visit vertices in ***steps*** (the while loop in Line 5–12). In each step i, RADIUS-STEPPING increments the radius centered at s from d_{i-1} to d_i, and settles all vertices v in the annulus $d_{i-1} < d(s,v) \leqslant d_i$. This is illustrated in Figure 1.

Settling these vertices involves multiple ***substeps*** (the do loop in Lines 6–10) of what is effectively Bellman-Ford's algorithm. Each substep is easily parallelized. In Δ-stepping, the step distance $d_i = d_{i-1} + \Delta$ increases the radius by a fixed amount on each step. This can require $\Theta(n)$ substeps in the worst case, which is not efficient since each substep will process the same set of vertices and their edges. In the worst case, it could take $O(nm)$ work.

In the RADIUS-STEPPING algorithm, a new step distance d_i is decided on each step with the goal of bounding the number of substeps. The algorithm takes a radius $r(v)$ for each vertex and selects a d_i on step i by taking the minimum of $\delta(v) + r(v)$ over all v in the frontier (Line 5). Lines 6–10 then run the Bellman-Ford substeps until all vertices with radius less than d_i are settled. Vertices within d_i are then added to the visited set S_i.

The algorithm is correct for any radii $r(\cdot)$, but by setting it appropriately, it can be used to control the number of substeps per step and the number of steps. If $r(v) = 0$, then the algorithm is effectively Dijkstra's, and the inner step is run only once. It may, however, visit multiple vertices with the same distance. If $r(v) = \infty$, then the algorithm is effectively the Bellman-Ford algorithm, and the substeps will run until all vertices are settled, and hence there will be a single step. If $r(v) = \Delta$, then the algorithm is almost Δ-stepping, but not quite since Δ is added to the distance of the nearest frontier vertex instead of to d_{i-1}.

In this paper, we aim to set $r(v)$ to be close, but no more, than the k-radius $\bar{r}_k(v)$. By making $r(v)$ no more than the k-radius, it guarantees that the algorithm will run at most $(k+2)$ Bellman-Ford substeps. For our theoretical bounds, we set $k = 1$ so the algorithm only does constant substeps.

In addition to bounding the number of substeps, we want to bound the number of steps. To do this, we can make use of properties of the input graph. In particular, if every vertex v has ρ vertices that are within a distance $r(v)$, i.e. $r_\rho(v) \leqslant r(v)$ (indicating $|B(v, r(v))| \geqslant \rho$), then we can bound the number of steps by $O(\frac{n}{\rho}\log(\rho L))$, where L is the longest edge in the graph.

THEOREM 3.1 (CORRECTNESS). *After the i-th step of RADIUS-STEPPING, each vertex v with $d_{i-1} < d(s,v) \leqslant d_i$ will have $\delta(v) = d(s,v)$ and will be included in S_i.*

Correctness of the algorithm is straightforward since it is essentially the Δ-stepping algorithm with a variable step size: At the end of the inner loop, all vertices within the step distance are settled, and all of their direct neighbors are relaxed.

To analyze the parallelism of this algorithm, we observe that the inner-most loops for each Bellman-Ford step (Lines 7 and 8) are readily parallelizable, so the longest chain of dependencies (i.e., the depth) is bounded by the number of steps (Line 5) and substeps (Line 6), as given below:

THEOREM 3.2 (THE NUMBER OF SUBSTEPS). *The repeat-until loop (Lines 6–10) runs at most $k+2$ times provided that $r(v) \leqslant \bar{r}_k(v)$ for all $v \in V$.*

THEOREM 3.3 (THE NUMBER OF STEPS). *The while loop (Lines 5–12) requires no more than*

$$\left\lceil \frac{n}{\rho} \right\rceil (1 + \lceil \log_2 \rho L \rceil) = O\left(\frac{n}{\rho}\log(\rho L)\right)$$

steps provided that $r_\rho(v) \leqslant r(v)$ for all $v \in V$.

We prove these theorems in Sections 3.1 and 3.2 that follow, and give implementation details of the algorithm in Section 3.3.

In the following analysis, we define the ***lead node*** in step i as the vertex v that attains d_i in Line 5 and denote it by v_i.

3.1 Number of Substeps

Let k be such that $r(v) \leqslant \bar{r}_k(v)$ for all $v \in V$. To bound the number of substeps, we use the fact that vertices with distances no more than the step distance in the previous step (d_{i-1}) are all correctly computed, as given by Theorem 3.1. Observe that their neighbors are also relaxed, since the distances to none of these vertices are updated in the last substep of the previous step (the termination condition in Line 10).

LEMMA 3.4. *For a vertex $v \in V$, if $d_{i-1} < d(s,v) \leqslant d_i$, then its distance is correctly computed after $k+1$ substeps of the i-th step, i.e. $\delta(v) = d(s,v)$.*

PROOF. Consider the shortest path from s to v that uses the fewest hops. Let u be the first vertex on this path with distance larger than d_{i-1}. By Theorem 3.1, we have that $\delta(u)$ is relaxed to $d(s,u)$ at the end of the $(i-1)$-th step.

Now we prove that v must be within $(k+1)$ hops from u. Assume to the contrary that the hop distance $\hat{d}(u,v) > k+1$. Let v' be the vertex on the shortest path from s to v which appears right before v, then $d(u,v') < d(u,v) \leqslant d_i - \delta(u) \leqslant r(u) \leqslant \bar{r}_k(u)$, which contradicts the condition that $\hat{d}(u,v') = \hat{d}(u,v) - 1 > k$ (the definition of k-radius $\bar{r}_k(u)$ prevents this from happening). Hence, we have $\hat{d}(u,v) \leqslant k+1$, and the distance of v can be computed within $k+1$ substeps along the shortest path from u to v. □

By Lemma 3.4, all vertices with distances no more than d_i can be computed within $k+1$ substeps. We need an extra substep to relax all of their direct neighbors, and to ensure no $\delta(v) \leqslant d_i$ was updated. This proves Theorem 3.2.

3.2 Number of Steps

Let $\rho > 0$ be such that the condition $r(v) \geqslant r_\rho(v)$ (or $B(v, r(v)) \geqslant \rho$) holds for all $v \in V$. We prove Theorem 3.3 by showing the following lemma:

LEMMA 3.5. *In any $t = 1 + \lceil \log_2 \rho L \rceil$ consecutive steps, except possibly in the last step of the algorithm, we have $|S_{i+t} - S_i| \geqslant \rho$.*

The lemma says that RADIUS-STEPPING will visit at least ρ vertices in any $t = 1 + \lceil \log \rho L \rceil$ consecutive steps, except perhaps in the last one. Hence, all vertices will be visited in no more than $\lceil n/t \rceil = \lceil n/\rho \rceil (1 + \lceil \log_2 \rho L \rceil)$ steps.

To prove this lemma, we show that in a step $j, i < j \leqslant i+t$, either ρ vertices have already been visited in the j-step (i.e. $|S_{j+1} - S_i| \geqslant \rho$), or the difference in the step distance doubles (i.e. $d_j - d_i \geqslant 2(d_{j-1} - d_i)$). In the latter case, if that happens for t consecutive steps, then $d_t - d_i \geqslant 2^{t-1} > \rho L$, which has at least ρ nodes in this range.We begin the proof by showing some basic facts about the algorithm: The lemma below shows a property of the enclosed ball of the lead node v_j.

LEMMA 3.6. *Let $i < j$. For a lead node v_j, if $r(v_j) < d(s,v_j) - d_i$, then $B(v_j, r(v_j)) \subseteq (S_j - S_i)$.*

PROOF. Consider a vertex $v \in B(v_j, r(v_j))$. First, v cannot be visited after the j-th step, since $d(s,v) \leqslant d(s,v_j) + d(v_j,v) \leqslant \delta(v_j) + r(v_j) = d_j$. Similarly, v cannot be visited in the first i steps since $d(s,v) \geqslant d(s,v_j) - d(v_j,v) > d_i + r(v_j) - d(v_j,v) \geqslant d_i$. □

The next property states that starting from the i-th step, either we reach ρ vertices, or the distance we explore in each step doubles.

LEMMA 3.7. *Let i be given. If $|B(v_j, r(v_j))| \geqslant \rho$, then $\forall j > i$, either $|S_j - S_i| \geqslant \rho$, or $d_j - d_i \geqslant 2(d_{j-1} - d_i) \geqslant 2^{j-i-1}$.*

PROOF. Lemma 3.6 already shows that if $r(v_j) < d(s,v_j) - d_i$ then we have $|S_j - S_i| \geqslant |B(v_j, r(v_j))| \geqslant \rho$. Otherwise, since $\delta(v_j) \geqslant d(s,v_j)$,

$$\begin{aligned} d_j - d_i &= \delta(v_j) + r(v_j) - d_i \\ &\geqslant d(s,v_j) + \big(d(s,v_j) - d_i\big) - d_i \\ &= 2(d(s,v_j) - d_i) \geqslant 2(d_{j-1} - d_i), \end{aligned}$$

where the last step used the fact that $d(s,v_j) \geqslant d_{j-1}$ (Line 5). Since the minimum edge weight is 1, we have that $d_{i+1} - d_i \geqslant 1$ and further that $d_j - d_i \geqslant 2^{j-i-1}$. □

With Lemma 3.7, we now prove Lemma 3.5, which is equivalent to proving Theorem 3.3.

PROOF OF LEMMA 3.5. If $|B(v_j, r(v_j))| \geqslant \rho$ and $|S_{i+t} - S_i| < \rho$, then based on Lemma 3.7, $d_{i+t} - d_i \geqslant 2^{t-1} \geqslant \rho L$. Consider the node $u^* = \arg\min_{u \in V \setminus S_{i+t}} d(s,u)$. Let $\{u_1 = s, u_2, \cdots, u_k = u^*\}$ be the ancestors to reach u^* on the shortest-path tree. From the definition of u^* we have $\{u_1 = s, u_2, \cdots, u_{k-1}\} \subseteq S_{i+t}$. Meanwhile, since the longest edge in G has edge weight L, for any j that $k - \rho \leqslant j \leqslant k-1$, the shortest-path distance to u_j satisfies that $d(s,u_j) \geqslant d(s,u^*) - (k-j)L \geqslant d(s,u^*) - \rho L > d_{i+t} - \rho L \geqslant d_i$, which means $u_j \notin S_i$. Therefore, $\{u_{k-\rho}, u_{k-\rho+1}, \cdots, u_{k-1}\} \in S_{i+t} - S_i$, and $|S_{i+t} - S_i| \geqslant \rho$. □

3.3 Work and Depth

We now analyze the work and depth of RADIUS-STEPPING. Algorithm 1 is a high-level description, and an efficient implementation with priority queues is shown in Algorithm 2. Throughout the complexity analysis, we assume $r_\rho(v) \leqslant r(v) \leqslant \bar{r}_k(v)$ for all $v \in V$ (we show how to prepare the graph to guarantee this property in the next section).

There are mainly two steps in Algorithm 1 that are costly and require efficient solutions: the calculation of the step distance d_i (Line 5) and the discovery of all unvisited vertices with tentative distances less than d_i (Line 7).

To efficiently support these two types of queries, we use two ordered sets Q and R, implemented as balanced binary search trees (BSTs) to store the tentative distance ($\delta(u)$) for each unvisited vertex and the tentative distance plus the vertex radius ($\delta(u) + r(u)$). With these BSTs, the selection of step distance corresponds to finding the minimum in the tree R, and picking vertices within a certain distance is a split operation on the BST Q. Both operations require $O(\log n)$ work.

Let A_i be the ***active set*** that contains the elements visited in the i-th step (all vertices u in line 7 in Algorithm 1). This can be computed by splitting the BST Q (Line 7 in Algorithm 2). To maintain the priority queues sequentially, we relax all neighbors of vertices in A_i for $k+2$ substeps. Other than updating the tentative distance, we update each neighbor vertex (referred to as v) as follows:

(1) if the previous distance of v is already no more than d_i, v is already removed from Q and R earlier in this step, so we do nothing;
(2) if the previous distance is larger than d_i while the updated value is no more than d_i, we remove v from Q and R, and add v to the active set A_i; and
(3) if the updated distance is still larger than d_i, we decrease the keys corresponding to v in Q and R.

It can be checked that the efficient implementation with BSTs Q and R (Algorithm 2) visits the same vertices in each step as Algorithm 1 and hence computes the shortest-path distances correctly.

The key step to parallelize Algorithm 2 is to handle the inner loop (Line 10), which can be separated into two parts: (1) update the tentative distances and (2) update the BSTs. We process these two steps in turn. To update the tentative distances, we just use priority-write (WRITEMIN) to relax the other endpoints for all edges that have endpoints in A_i. After all tentative distances are updated, each edge checks whether the relaxation (priority-write) is successful. If successful, the edge will *own* this vertex (called u), and create a BST node that contains a pair of keys (lexicographical ordering): the current tentative distance of u as the first key, and the vertex label of u as the second key. Then, we create a BST that contains all these nodes using standard parallel packing and sorting process. With this BST, we first apply set DIFFERENCE to remove out-of-date keys in Q, then we SPLIT this BST into two parts by d_i, and UNION each part separately with A_i and Q. It is easy to see that in this parallel version, we apply asymptotically the same number of operations to Q compared to the sequential version. We can maintain R in a similar way.

Let us now analyze the work and depth of the parallel version. Each vertex is in the active set in only one step, so each of its neighbors is relaxed $k+2$ times, once in each substep. In the worst case (all the relaxation succeeded), there are $O(m)$ updates for Q and R; each can be done for no more than $O(k)$ times when $r(v) \leqslant \bar{r}_k(v)$.

Suppose there are t steps and the number of updates in step i is a_i. Then, we have $\sum_{i=1}^{t} a_i = m$, and $t = O(\frac{n}{\rho}\log \rho L)$ as $r_\rho(v) \leqslant r(v)$. The work and depth for set DIFFERENCE and UNION are $O(p\log q)$ and $O(\log q)$ for two sets with size p and q, $q \geqslant p$. The overall work W for all set operations is no more than $\sum_{i=1}^{t} a_i \cdot k\log n = O(km\log n)$. The depth D is no larger than

$$O\left(\sum_{i=1}^{t} k\log n\right) = O\left(k\frac{n}{\rho}\log \rho L\log n\right)$$

All other steps in Algorithm 2 are dominated by the cost to maintain the BSTs. The total work and depth can be summarized in the following lemma.

LEMMA 3.8. *Assuming $r_\rho(v) \leqslant r(v) \leqslant \bar{r}_k(v)$ for all $v \in V$, the RADIUS-STEPPING algorithm computes single-source shortest paths in $O(km\log n)$ work and $O\left(k\frac{n}{\rho}\log n\log \rho L\right)$ depth.*

3.4 Unweighted Cases

The implementation we just discussed also works on unweighted graphs. But the performance can be further improved in this case. A special property of the unweighted case is that all vertices in the frontier have the same tentative distances. This means that we do not need search trees or priority queues to maintain the ordering. Hence, a similar approach to parallel BFS can be directly used here, and each step takes $O(n')$ work and $O(\log^* n')$ depth on the CRCW PRAM model [2] where n' is the number of vertices and their associated edges in each step. The additional step to compute the step distance uses one priority-write. Because of concavity, the worst case appears when n' is the same in every step.

LEMMA 3.9. *The RADIUS-STEPPING algorithm computes single-source shortest paths on unweighted graph in $O(m+n)$ work and $O(n/\rho \cdot \log\rho\log^*\rho)$ depth.*

[2]The bound changes with different models, and a more practical solution is to use semisort [12].

Algorithm 2: The Shortest-path Algorithm.

Input: A graph $G=(V,E,w)$, vertex radii $r(\cdot)$ and a source node s.
Output: The graph distances $\delta(\cdot)$ from s.

```
1  δ(·) ← +∞, δ(s) ← 0
2  i ← 1
3  Q = {w(s,u) | u ∈ N(s)}
4  R = {w(s,u) + r(u) | u ∈ N(s)}
5  while |Q| > 0 do
6      d_i ← R.EXTRACT-MIN()
7      {A_i, Q} = Q.SPLIT(d_i)
8      foreach u ∈ A_i do R.REMOVE(u)
9      repeat
10         foreach u ∈ A_i, v ∈ N(u) do
11             if δ(v) > δ(u) + w(u,v) then
12                 if δ(v) > d_i and δ(u) + w(u,v) ⩽ d_i then
13                     R.REMOVE(v)
14                     Q.REMOVE(v)
15                     A_i.INSERT(v)
16                 δ(v) = δ(u) + w(u,v)
17                 if δ(v) > d_i then
18                     Q.DECREASE-KEY(v, δ(v))
19                     R.DECREASE-KEY(v, δ(v) + r(v))
20     until no δ(v) that v ∈ A_i is updated
21     i = i + 1
22 return δ(·)
```

4. SHORTCUTS AND PREPROCESSING

This section describes how to prepare an arbitrary undirected graph so that RADIUS-STEPPING can run efficiently afterward. As shown earlier, the cost of RADIUS-STEPPING is a function of k and ρ provided that the input to the algorithm is a (k,ρ)-graph. But not every graph as given meets this condition with k and ρ that one desires. Our aim here is to prepare the input graph by adding a small number of edges (shortcuts) to satisfy the condition for k and ρ that the user chooses. The process will also generate an appropriate vertex radius $r(v)$ for every $v \in V$.

Any graph can be turned into a (k,ρ)-graph by adding up to $n\rho$ edges (put shortcut edges from every vertex to its ρ-nearest vertices). For $k=1$, this strategy adds the minimum number of edges possible. For $k>1$, however, one can usually do better. Below, we discuss efficient preprocessing strategies and their cost.

4.1 Heuristics for $(1,\rho)$-graph

The simplest case of a (k,ρ)-graph is $k=1$. In this case, we have to ensure that every vertex is a $(1,\rho)$-ball, so all the ρ-closest vertices from a vertex u are added to u's neighbor list with edge weight $d(u,\cdot)$. To do so for all the vertices, we can, in parallel, start n Dijkstra's execution (or BFS for unweighted graphs) and compute the ρ-closest vertices in each run.

LEMMA 4.1. *Given a graph G, generating $(1,\rho)$-balls for all vertices takes $O(m\log n + n\rho^2)$ work and $O(\rho\log\rho)$ depth.*

PROOF. Initially, we sort all edges from each vertex by their weights, requiring $O(m\log n)$ work and $O(\log n)$ depth. This can be skipped if the edges are presorted or the graph is unweighted. Then, run parallel Dijkstra's [4] from each vertex for ρ rounds. For each vertex, we only consider the lightest ρ edges as only these edges may be needed to reach the ρ-closest vertices. So, the search from each vertex explores no more than ρ^2 edges (ρ^2 DECREASE-KEYS

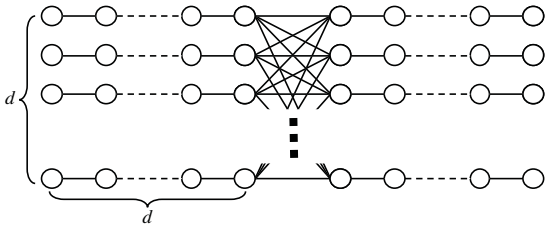


Figure 2: Example of a sparse graph that requires BFS to look at $O(d^2)$ edges from any vertex to reach $3d$ vertices.

in the Fibonacci Heap) and visits ρ vertices (ρ DELETE-MINS in the Fibonacci Heap), leading to $O(\rho^2)$ work for each source node—or in total, $O(\rho \log \rho)$ depth and $O(n\rho^2)$ work.

We can then add ρ shortcut edges from the source to all these vertices. When the algorithm stops in the ρ-th round, we also have the value of $r_\rho(v)$, the distance from the source to ρ-th nearest neighbor. Clearly, we have $r_\rho(v) \leqslant \bar{r}_1(v)$ at this time. □

For the unweighted case, a parallel BFS from the source vertex to reach ρ-nearest neighbors takes $O(\rho^2)$ work and $O(\rho \log^* \rho)$ depth.

Combining Lemmas 4.1, 2.1, and 3.9 proves the main theorems. Notie that after preprocessing, the graph now has $O(m + n\rho)$ edges.

It is worth mentioning that even when the given graph is unweighted and sparse, we may still need to look at $O(\rho^2)$ edges to reach ρ vertices. We give an example of a carefully-constructed graph with that behavior in Figure 2. If $d = \lfloor \rho/3 \rfloor - 1$, then the BFS search from any vertex has to visit $O(d^2)$ edges to reach $\rho > 3d$ vertices. However, Section 5.3 shows empirically that this scenario is not common in real-world graphs. Furthermore, if the input graph has constant-bounded degrees, the work for this step is $O(n\rho)$.

4.2 Heuristics for (k, ρ)-graphs

The construction of a $(1, \rho)$-graph can add up to $n\rho$ extra edges, potentially too many to be useful for many graphs in practice. Using a small $k > 1$ creates an opportunity to add fewer shortcut edges, by taking better advantange of the original edges.

To construct a (k, ρ)-graph, we will still run the algorithm in Lemma 4.1. But for each vertex s, instead of adding a shortcut edge for each of the ρ closest vertices, we derive a shortest-path tree T_s (at no additional cost) from s that reaches out to its ρ nearest vertices. We then work with these trees with the goal of adding as few edges as possible while ensuring that every $v \in V(T_s)$ can be reached within k hops using the combination of the original edges and the added edges E'.

We propose two heuristics below, both relying on the following claim, which shows that from the perspective of a single source vertex, the most beneficial shortcut edge goes directly from the source itself. The proof is straightforward and is omitted.

CLAIM 4.2. *Let s be a source vertex. A shortcut edge (u, v), $u \neq s$, can be exchanged for (s, v) with an appropriate weight without increasing the overall hop count.*

Shortcut using Greedy: The greedy heuristic eliminates vertices that are more than k hops away in T_s by adding edges from s to all $(ki + 1)$-th hop vertices for $i \in \mathbb{Z}_+$. It is simple and easy to implement, but may end up adding far too many edges. For example, consider a shortest-path tree that first forms a chain of length k from the source, then the remaining $\rho - k - 2$ vertices all lie in the $(k + 1)$-st level. The greedy heuristic will shortcut to all these $\rho - k - 2$ vertices, while we can only add one edge from the source to the node in the k-th layer to include all vertices in k hops.

Shortcut using Dynamic Programming (DP): The DP heuristic solves a dynamic program that adds the optimal number of edges from the perspective of one vertex. Consider, among all shortest-path trees from s, one where for every $v \in T_s$, the path $s \to v$ on T has the smallest hop count possible. The optimal number of edges needed for a certain node can be computed using the following dynamic program. Let $F(u, t)$ be the number of edges into the subtree of T_s rooted at u so that every node in the subtree is at most k hops away from s provided that parent(u) is at t hops away from s. Then, for all $u \neq s$, $F(u, t)$ is given by

$$\begin{cases} 1 + \sum\limits_{w \in N^+(u)} F(w, 1) & \text{if } t = k \\ \min\left(1 + \sum\limits_{w \in N^+(u)} F(w, 1), \sum\limits_{w \in N^+(u)} F(w, t+1)\right) & \text{if } t < k \end{cases}$$

The number of edges needed in the end is $\sum_{u \in N^+(s)} F(u, 0)$. The actual edges to add can be traced from the recurrence. We note that this dynamic program can be solved in $O(k\rho)$ (as $|T_s| = \rho$) work and $O(\rho \log n)$ depth by resolving the recurrence bottom up. We apply this to all n vertices in parallel, requiring $O(nk\rho)$ work and $O(\rho \log n)$ depth overall.

Note that although the dynamic programming solution gives an optimal solution for each shortest-path tree individually, the overall solution is not necessary the global optimal. Adding globally smallest number of edges to construct (k, ρ)-ball appears to be much more involved. Yet, we show empirically in Section 5 that the dynamic programming (DP) heuristic performs very well on a wide variety of graphs—and even the greedy heuristic does well on well-structured graphs.

5. EXPERIMENTAL ANALYSIS

This section empirically investigates how the settings of k and ρ, and the choice of shortcutting heuristics affect the number of shortcut edges added, as well as the number of steps RADIUS-STEPPING requires. These quantities provide insights into how a well-engineered implementation would perform in practice and how much parallelism RADIUS-STEPPING is able to derive.

5.1 Experiment Setup

We use graphs of various types from the SNAP datasets [18], including the road networks in Pennsylvania and Texas (real-world planer graphs) and the web graphs of the University of Notre Dame and Stanford University (real-world networks). In the case of web graphs, each edge represents a hyperlink between two web pages. We also use synthetic graphs of 2D and 3D grids (structured and unstructured grids). For each of the graphs used, the number of vertices and the number of edges are given in the table below.

Graph	# Vertices	# Edges
Road map: Pennsylvania	1.09M	3.08M
Road map: Texas	1.39M	3.84M
Web graph: Notre Dame	325K	2.20M
Web graph: Stanford	281K	3.98M
Grid: 2D	1M	2.00M
Grid: 3D	1M	5.94M

In practical applications, such graphs are used both in the weighted and unweighted settings. In the weighted setting, for instance, the distances in road networks and grids can represent real distances, while the edge weights in network graphs (e.g., social networks) also have real-world meanings, such as the time required to pass a message between two users, or (the logarithm of) the probability

to pass a message. In our experiments, if a graph does not come equipped with weights, we assign to every edge a random integer between 1 and 10,000.

We construct (k, ρ)-balls using the heuristics described in Section 4.2, except that the implementation has the following modifications: instead of breaking ties arbitrarily and taking exactly ρ neighbors, we continue until all vertices with distance $r_\rho(\cdot)$ are visited. This has the same implementation complexity as the theoretical description but is more deterministic. Using this, our results are a pessimistic estimate of the original heuristics as more than ρ edges may be found for some sources. However, in all our experiments, we found the difference to be negligible in most instances.

To improve our confidence in the results, two of the authors have independently implemented and conducted the experiments, and arrived at the same results, even without introducing a tie-breaker.

5.2 The Number of Shortcut Edges

Of the heuristics in Section 4.2 for making a (k, ρ)-graph, *how many extra edges are generated by each heuristic?* To answer this question, we use 3 representative graphs in our analysis: (1) road networks in Pennsylvania, (2) a webgraph of Stanford University, and (3) a synthetic 1000-by-1000 2D grid. All these graphs have about 1 million vertices, and between 2 and 3 million edges. We show experimental results for unweighted graphs since the performance of the heuristics is independent of edge weights.

Figure 3 shows the number of added edges in terms of the fraction of the original edges for $k = 3$ as the value of ρ is varied between 10 and 1,000. More detailed results are given in Tables 8 and 9. Evidently, both heuristics achieve similar results on the road map and 2D grid. This is because road maps and grids are relatively regular, in fact almost planar. Thus, even naïvely shortcutting to the $(ki + 1)$-hop performs well: for small k, both heuristics add only a small number of edges. As k becomes larger, the gap between the greedy and the DP heuristic increases. When ρ reaches 1000, both DP and greedy add a large number of edges (more than 100x edges). This is because in road maps and grids, the degree of a vertex is usually a small constant, making the shortest-path tree deep.

On web graphs, which is less regular, DP only adds 4x the number of original edges even when ρ is as large as 1000 while greedy still adds 100x the original edges. This is because web graphs not only are far more irregular but also have a very skewed degree distribution. As a known scale-free network [1], it has some "super stars" (or more precisely, the "hubs") in the network. In this case, the bad example in Section 4.2 occurs frequently when these hubs are not at the exact $(ki + 1)$-th layer in the shortest-path tree, while the DP heuristic can discover the hubs accurately. This also explains the phenomena that only a few edges are added on web graphs by the DP heuristic even when ρ is large, since the hubs already significantly reduce the depth of the shortest path tree, and it only takes a few edges to shortcut to the hubs. This property holds for many kinds of real-world graphs such as social networks, airline networks, protein networks, and so on. In such graphs, a relatively optimal heuristic is necessary to construct the enclosed balls; a naïve method often has bad performance. As can be seen, the DP solution achieves satisfactory performance on web graphs, adding only about 10% more edges with $k = 3$ and $\rho = 100$.

5.3 The Number of Visited Edges

Theoretically, the preprocessing step has to examine $O(\rho^2)$ edges per source vertex (Section 4.1). This bound is indeed tight even on sparse unweighted graphs (Figure 2). In practice, the number of visited edges tends to be much smaller: *on all real-world graphs studied, we find that the number of visited edges appears proportional to ρ for $\rho \leqslant 1000$.*

ρ	Road Maps		Web Graphs		Grids	
	Penn	*Texas*	*Notre Dame*	*Stanford*	*2D*	*3D*
10	1.47	1.47	1.06	1.18	1.24	1.20
20	1.72	1.71	1.10	1.27	1.84	1.66
50	1.99	1.96	1.20	1.52	2.61	2.32
100	2.16	2.12	1.29	1.70	2.94	2.90
200	2.29	2.24	1.35	1.72	3.26	3.45
500	2.43	2.38	1.34	1.97	3.52	4.00
1000	2.51	2.46	1.73	1.98	3.66	4.40

Table 1: The average number of edges visited from each source node divided by $n\rho$ for different unweighted graphs.

ρ	Road Maps		Web Graphs		Grids	
	Penn	*Texas*	*Notre Dame*	*Stanford*	*2D*	*3D*
10	1.69	1.68	1.49	1.74	1.76	1.84
20	2.01	1.99	1.80	2.00	2.19	2.20
50	2.31	2.28	2.29	2.73	2.72	2.63
100	2.47	2.44	2.67	2.85	3.04	2.95
200	2.59	2.55	2.99	3.20	3.28	3.27
500	2.69	2.66	3.41	3.89	3.51	3.68
1000	2.75	2.72	4.02	4.73	3.64	3.97

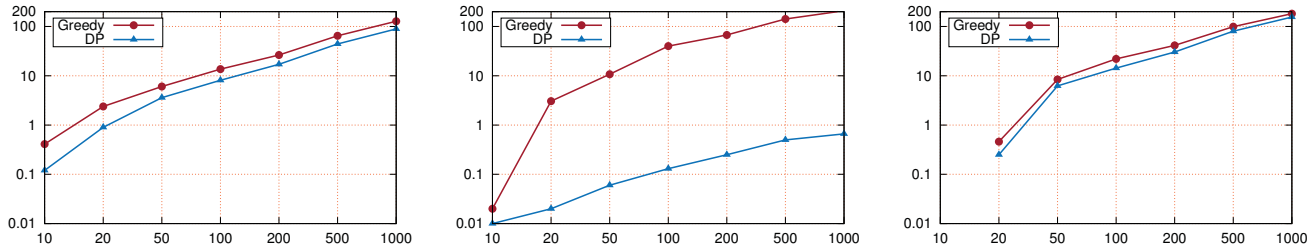
Table 2: The average number of edges visited from each source node divided by $n\rho$ for different weighted graphs.

For the unweighted case, our implementation of preprocessing is just a standard BFS, and it terminates as soon as ρ vertices are added to the visited set. For the weighted case, it is slightly more complex, and we use a BST to maintain the priority queue with size no more than ρ. Then, a standard Dijkstra's is run from every source node. Since the edges from each vertex are sorted by weights, if the current relaxed distance is larger than the ρ-th nearest node from the source (can be checked in $O(1)$ cost), then we can skip all the remaining edges. Therefore, it is possible that even fewer edges are visited in the preprocessing for the weighted case, compared to the unweighted case on the same graph.

Tables 1 and 2 show the average number of edges visited from each source node divided by ρ on both unweighted and weighted cases. We can see that on all input instances, the average number of visited edges is never more than 5x compared to ρ, so it is much closer to $O(\rho)$ than $O(\rho^2)$. The ratio slightly increases with larger ρ, but the growth is much slower.

5.4 The Number of Steps

How many steps does Radius-Stepping take for each setting of ρ? We ran our Radius-Stepping on both weighted and unweighted graphs and counted the number of steps as we change ρ. Notice that the number of steps is independent of k and is only affected by ρ. (The value of k only affects the number of substeps within a step.) We discuss the performance of our algorithm on all six graphs after preprocessing the graphs with the corresponding ρ. Since the cost of SSSP potentially varies with the source and we cannot afford to try all possible sources, we take 1000 random sample sources for each graph. We use the same 1000 sources for all our experiments for both the weighted and unweighted cases. We report the arithmetic mean over the sampled sources.



(a) Road map of Pennsylvania **(b) Web graph of Stanford** **(c) 2D-grid**

Figure 3: The ratio of extra edges to the original edges (y-axis) to generate $(k = 3, \rho)$-graph using greedy and dynamic programming (DP) for different types of graphs as ρ (x-axis) is varied.

ρ	Road Maps		Web Graphs		Grids	
	Penn	*Texas*	*Notre Dame*	*Stanford*	*2D*	*3D*
1	619.12	761.06	28.09	108.92	1504.00	223.50
2	309.32	380.31	13.77	54.23	751.76	111.50
5	308.47	379.34	13.44	43.27	751.74	111.50
10	206.30	253.71	13.32	31.29	501.14	74.50
20	165.73	196.30	13.17	21.67	375.62	74.48
50	123.01	151.13	12.38	14.13	250.32	55.48
100	101.41	124.07	9.78	10.63	187.46	44.08
200	78.61	96.92	8.47	8.56	136.24	36.48
500	58.44	70.75	6.63	7.30	87.86	27.36
1000	45.95	55.39	5.69	7.18	64.88	21.74
2000	35.66	42.58	5.27	6.72	44.82	17.94
5000	24.95	29.17	4.14	5.84	28.82	12.50
10000	18.54	21.33	3.83	5.76	20.18	10.00

Table 3: The average number of steps with different settings of ρ for different unweighted graphs.

ρ	Road Maps		Web Graphs		Grids	
	Penn	*Texas*	*Notre Dame*	*Stanford*	*2D*	*3D*
2	2.00	2.00	2.04	2.01	2.00	2.00
5	2.01	2.01	2.09	2.52	2.00	2.00
10	3.00	3.00	2.11	3.48	3.00	3.00
20	3.74	3.88	2.13	5.03	4.00	3.00
50	5.03	5.04	2.27	7.71	6.01	4.03
100	6.11	6.13	2.87	10.25	8.02	5.07
200	7.88	7.85	3.32	12.72	11.04	6.13
500	10.59	10.76	4.24	14.92	17.12	8.17
1000	13.47	13.74	4.94	15.17	23.18	10.28
2000	17.36	17.87	5.33	16.21	33.56	12.46
5000	24.81	26.09	6.79	18.65	52.19	17.88
10000	33.39	35.68	7.33	18.91	74.53	22.35

Table 4: The number steps required by RADIUS-STEPPING unweighted graphs divided by the number of BFS steps.

Unweighted Graphs (BFS): Figure 4 shows, for the unweighted case, the average number of steps taken by RADIUS-STEPPING as ρ is varied. More results appear in Tables 3 and 4, which compare the number of steps taken by RADIUS-STEPPING with that of a conventional BFS implementation.

Several things are clear: on a log-log scale[3], the trends are downward linear as ρ increases except for the Notre Dame web graph (not completely regular but shows a similar trend). This suggests that the average number of steps is inversely proportional to ρ, which is consistent with our theoretical analysis.

The web graph has a relatively smoother slope. The reason is that once the "super stars" are included in the enclosed balls, which usually only need a few hops, then most of the vertices will be visited in a few steps (20–100 steps vs. 200–1,500 steps on the other graphs when $\rho = 1$). However, we will later see that in the weighted case, the performances on these graphs are even better than the other graphs. For the road maps and grids, the number of steps reduces steadily. Furthermore, the number of steps shown in the experiments is much less than the theoretical upper bound $(\frac{n}{\rho} \log \rho)$ because most real-world graphs tend to have a hop radius that is much smaller than n.

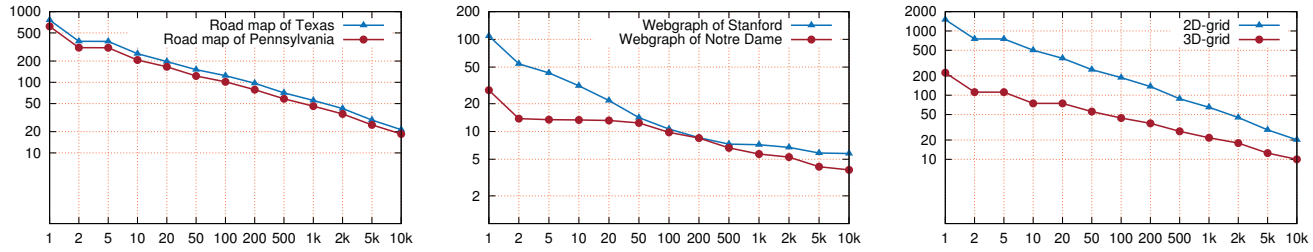
On all the graphs studied, ρ can be as small as 10 and already reduces the number of steps by 3x. When $\rho = 100$, the reduction is about 10x. As the results show, we can obtain more than 20x reduction when thousands of vertices are in the balls. This experimentally supports our theoretical analysis.

What should an unweighted graph look like so that RADIUS-STEPPING can significantly reduce the steps after adding no more than m edges? At first thought, the answer might be a grid or a road map with a large diameter, so that there is more space to be reduced. However, our experiments give the opposite answer: RADIUS-STEPPING, in fact, performs better on web graphs with smaller diameters. On web graphs, RADIUS-STEPPING can reduce the number of steps by 15x by adding no more than m edges (choosing $\rho = 1000$ and $k = 3$), while on road maps and grids, a 5x reduction in steps requires $4m$ to $6m$ edges. Even though the number of steps reduces more steadily and quickly on road maps and grids with larger ρ, the number of added edges in turn increases more rapidly (100x times more edges added to achieve 20x reduction on the number of steps). On scale-free networks, however, RADIUS-STEPPING improves standard BFS by more than 10x without adding many extra edges, and an efficient implementation of RADIUS-STEPPING on these networks might be worthwhile in the future.

Weighted Graphs: Figure 5 shows, for the weighted case, the average number of steps taken by RADIUS-STEPPING as ρ is varied. More results appear in Tables 5 and 6, which compare the number of steps taken by RADIUS-STEPPING to when $\rho = 1$. Notice that when $\rho = 1$, RADIUS-STEPPING becomes essentially Dijkstra's except vertices with the same distance are extracted together.

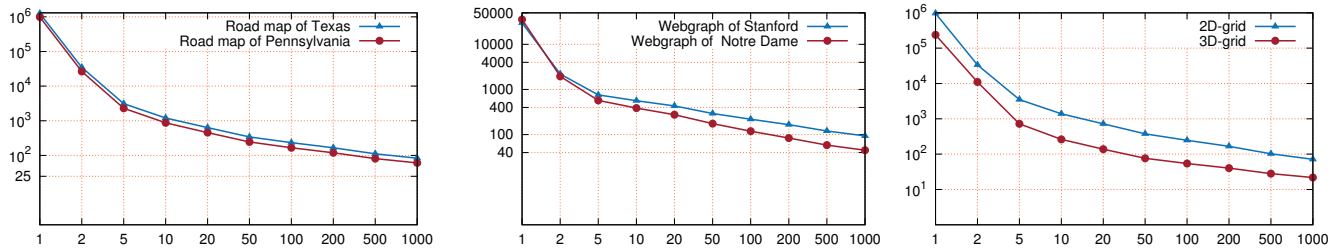
Similar to the unweighted case, the trends in Figure 5 are also nearly-linear, indicating an inverse-proportion relationship between

[3]The vertical axis is drawn on a log scale and the horizontal axis closely approximates a log scale.



(a) Road maps **(b) Web graphs** **(c) Grids**

Figure 4: Unweighted graphs—the average number of RADIUS-STEPPING steps (y-axis) as ρ (x-axis) is varied.



(a) Road maps **(b) Web graphs** **(c) Grids**

Figure 5: Weighted graphs—the average number of RADIUS-STEPPING steps (y-axis) as ρ (x-axis) is varied.

ρ	Road Maps		Web Graphs		Grids	
	Penn	*Texas*	*Notre Dame*	*Stanford*	*2D*	*3D*
1	986K	1252K	35.6K	30.0K	965K	239K
2	26479.9	34673.4	1953.7	2203.3	33592.2	11046.1
5	2294.5	3123.5	571.3	759.2	3495.8	722.4
10	872.6	1206.5	387.2	562.3	1385.0	261.9
20	455.0	634.1	274.9	432.2	722.9	137.8
50	245.0	343.0	174.6	293.7	375.1	76.1
100	167.2	233.7	118.8	219.3	246.9	54.1
200	119.8	166.9	83.7	166.0	166.9	40.2
500	81.1	111.3	58.4	120.0	102.1	28.1
1000	61.1	83.2	45.0	93.6	71.1	21.7

Table 5: The average number of steps with different settings of ρ for weighted graphs.

ρ	Road Maps		Web Graphs		Grids	
	Penn	*Texas*	*Notre Dame*	*Stanford*	*2D*	*3D*
2	37.2	36.1	18.2	13.6	28.7	21.6
5	429.7	400.8	62.3	39.6	276.1	330.6
10	1130.0	1037.7	92.0	53.4	696.8	912.0
20	2167.0	1974.5	129.5	69.5	1334.9	1733.3
50	4024.5	3650.1	203.9	102.3	2572.7	3138.5
100	5897.1	5357.3	299.7	137.0	3908.6	4414.9
200	8230.4	7501.5	425.4	180.9	5782.0	5941.4
500	12157.8	11248.9	609.7	250.3	9451.7	8499.8
1000	16137.5	15048.1	791.2	320.9	13572.8	11006.6

Table 6: The number of steps taken by RADIUS-STEPPING on weighted graphs divided by the number of steps taken by RADIUS-STEPPING itself when using $\rho = 1$.

the number of steps and ρ, which is consistent with our theory. In the weighted case, however, we can visit much fewer vertices in each step than those in the unweighted case because most vertices have different distances to the source. Thus, Dijkstra's algorithm (or when $\rho = 1$) requires almost n steps to finish. As a result, considering the inverse-proportion relationship, even a small ρ can greatly reduce the number of steps. Indeed, the number of steps taken by RADIUS-STEPPING is far fewer than the predicted theoretical upper bound ($O(\frac{n}{\rho} \log \rho L)$ with $L = 10^4$ here). We also notice that the number of steps decreases much faster when ρ is small compared to larger values of ρ.

The reduction in the number of steps is significant. Even $\rho = 10$ leads to about 1000x fewer steps on road maps and grids, and about 50–100x on the web graphs. When $\rho = 100$, we only need a few hundreds of steps on all graphs, which can already provide considerable parallelism. The number of steps further decreases to 20–80 when $\rho = 1000$. This provides some evidence that the algorithm has the potential to deliver substantial speedups when many more cores on a shared-memory machine are made available.

Another trend is that the reduction factors on the web graphs are always less than the other two types of graphs. The reason is that the web graphs are scale-free, and even when we do not use enclosed balls to traverse the graph, not many steps are required. The "hubs" substantially bring down the diameter of the graph. For more uniformly distributed graphs, such as the road maps and grids, increasing ρ to a big value reduces the number of steps required to perform SSSP more rapidly.

5.5 How to choose the parameters?

What is the best combination of k and ρ? The choice of k and ρ offers a tradeoff between the number of added edges (hence additional space and work) and the parallelism of the algorithm. In general, we do not want to increase the number of edges substantially; the total number of edges should be around $O(m)$. On every graph tested, $k = 3$ or 4 works reasonably well whereas ρ in the range 50–100 for weighted graph yields the best bang for the buck. A larger ρ can reduce the number of steps in RADIUS-STEPPING but in turn, increases the preprocessing time and the number of edges

required to be added. A larger k will reduce the number of added edges but at the same time, increases the number of visited edges, as well as the overall depth[4].

On unweighted graphs, Radius-Stepping performs better on web graphs, but less efficient on grids and road maps. Finally, since preprocessing is only run once, if SSSP will be run from multiple sources, we suggest increasing ρ and decreasing k: the cost for preprocessing is amortized over more sources.

6. PRIOR AND RELATED WORK

This secton discusses prior results on parallel single-source shortest paths that are most relevant to ours (for a more comprehensive survey, see [19] and the references therein). Throughout, we assume the input is an *arbitrary* undirected graph with n vertices and m nonnegative edges.

Some early algorithms achieve a high degree of parallelism (polylogarithm depth) but with substantially more work than Dijkstra's algorithm. Using matrix multiplications over semirings, Han et al. [13] gives an algorithm with $O(\log^2 n)$ depth and $O(n^3(\log\log n/\log n)^{1/3})$ work that, in fact, solves the *all-pairs* shortest-path problem. With randomization, this algorithm can be implemented in $O(\log n)$ depth and $O(n^3 \log n)$ work [11].

In another line of work, Driscoll et al. [9], refining the approach of Paige and Kruskal [21], present an $O(n \log n)$-depth algorithm with Dijkstra's work bound. Later, Brodal et al. [4] improve the depth to $O(n)$; however, the algorithm needs $O(m \log n)$ work. We summarize in Table 7 the cost bounds for exact SSSP algorithms that have subcubic work.

By allowing a slight increase in work in exchange for a better depth bound, many algorithms have been proposed. Ullman and Yannakakis [29] describe a randomized parallel breadth-first search (BFS) that solves unweighted SSSP in $\widetilde{O}(m + n\rho)$ work and $\widetilde{O}(n/\rho)$. The algorithm works by performing limited searches from a number of locations and adding shortcut edges with appropriate distances to speed up later traversal. Extending this idea to weighted graphs, Klein and Subramanian [17] derive an algorithm that runs in $O(\sqrt{n} \log L \log^2 n)$ depth and $O(m\sqrt{n} \log L \log^2 n)$ work. For undirected graphs, Cohen [6] presents an algorithm with $O(n/\rho \cdot \text{polylog}(n))$ depth and $O(n^2 \log L + n\rho^2)$ work. Shi and Spencer [25] show an algorithm with $\Theta(n/\rho \cdot \log n)$ depth, and $O(n\rho^2 \log n \log \rho + m \log n)$ or $O((\rho^3 + m\rho) \log n)$ work. This not only has a smaller depth bound than our algorithm but also is the only algorithm in Table 7 with strongly-polynomial depth. However, since the search is based on vertices instead of distances, the work and depth bounds here are tight, promising $O(\rho^2)$ parallelism on any input instances but leaving no room for more parallelism on "nice" input graphs. By contrast, the experiments in this paper show that our algorithm can attain a much higher degree of parallelism on many real-world graphs.

More recently, Meyers and Sanders [19] describe an algorithm called Δ-stepping and analyze it for various random graph settings. Although the algorithm works well on general graphs in practice, no theoretical guarantees are known.

In addition to these results, better algorithms have been developed for special classes of graphs such as planar graphs (e.g., [28, 16]) and separator-decomposable graphs [5].

There have also been approximation algorithms for SSSP. Cohen [7] describes a $(1+\varepsilon)$-algorithm for undirected graphs that runs in $O(\text{polylog}(n))$ depth and $O((m+n)n^{\alpha})$ work for $\alpha > 0$. Using an alternative hopset construction of Miller et al. [20], Cohen's algorithm can be made to run in $O(m \cdot \text{polylog}(n))$ work and $O(n^{1-\alpha})$ depth for $\alpha > 0$.

[4]As mentioned at the beginning of Section 5.4, a larger k will not increase the number of steps, but the number of the overall substeps, which is the inner loop of Radius-Stepping, hence an increase in the overall depth.

7. CONCLUSION

We presented a parallel algorithm for single-source shortest paths that runs in two phases. The preprocessing requires $O(n\rho^2)$ work and $O(\rho \log \rho)$ depth, adding $O(n\rho)$ edges (shortcuts) where ρ is a user-defined parameter. Then, the single-source shortest paths can be computed with $O((m+n\rho)\log n)$ work and $O(\frac{n}{\rho} \log n \log \rho L)$ depth for any arbitrary graph with nonnegative weights. The algorithm is simple and has the potential to be practical. Our experiments further show that the theoretical bounds are pessimistic for real-world graphs; the actual costs are often much less.

Acknowledgments

This research was supported in part by NSF grants CCF-1314590 and CCF-1533858, and the Intel Science and Technology Center for Cloud Computing.

8. REFERENCES

[1] A.-L. Barabási and R. Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.

[2] G. E. Blelloch, D. Ferizovic, and Y. Sun. Parallel ordered sets using join. In *ACM Symposium on Parallel Algorithms and Architectures (SPAA)*. ACM, 2016.

[3] G. E. Blelloch and M. Reid-Miller. Fast set operations using treaps. In *ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 16–26. ACM, 1998.

[4] G. S. Brodal, J. L. Träff, and C. D. Zaroliagis. A parallel priority queue with constant time operations. *Journal of Parallel and Distributed Computing*, 49(1):4–21, 1998.

[5] E. Cohen. Efficient parallel shortest-paths in digraphs with a separator decomposition. *Journal of Algorithms*, 21(2):331–357, 1996.

[6] E. Cohen. Using selective path-doubling for parallel shortest-path computations. *Journal of Algorithms*, 22(1):30–56, 1997.

[7] E. Cohen. Polylog-time and near-linear work approximation scheme for undirected shortest paths. *Journal of the ACM (JACM)*, 47(1):132–166, 2000.

[8] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[9] J. R. Driscoll, H. N. Gabow, R. Shrairman, and R. E. Tarjan. Relaxed heaps: An alternative to fibonacci heaps with applications to parallel computation. *Communications of the ACM*, 31(11):1343–1354, 1988.

[10] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3):596–615, 1987.

[11] A. M. Frieze and L. Rudolph. *A parallel algorithm for all pairs shortest paths in a random graph*. Management Sciences Research Group, Graduate School of Industrial Administration, Carnegie-Mellon University, 1984.

[12] Y. Gu, J. Shun, Y. Sun, and G. E. Blelloch. A top-down parallel semisort. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 24–34. ACM, 2015.

[13] Y. Han, V. Pan, and J. Reif. Efficient parallel algorithms for computing all pair shortest paths in directed graphs. In *ACM*

	Algorithm	Preprocessing		Shortest Path	
		Work	*Depth*	*Work*	*Depth*
Unweighted	Standard BFS	–	–	$O(m+n)$	$O(n)$
	Ullman and Yannakakis [29]	$O(m\rho+\rho^3)$	$O(\frac{n}{\rho}\log n\log^* n)$	$O(m+n\rho)$	$O(\frac{n}{\rho}\log n\log^* n)$
	Spencer [26]	–	–	$O(m\log\rho+n\rho^2\log^2\rho)$	$O(\frac{n}{\rho}\log^2\rho)$
	*This work**	$O(n\rho^2)$	$O(\rho\log^* n)$	$O(m+n\rho)$	$O(\frac{n}{\rho}\log\rho\log^* n)$
Weighted	Parallel Dijkstra's [21]	–	–	$O(m+n\log n)$	$O(n\log n)$
	Parallel Dijkstra's [4]	–	–	$O(m\log n+n)$	$O(n)$
	Klein and Subramanian [17]	–	–	$O(m\sqrt{n}\log^2 n\log L)$	$O(\sqrt{n}\log^2 n\log L)$
	Spencer [26]	–	–	$O((n\rho^2\log\rho+m)\log n\rho L)$	$O(\frac{n}{\rho}\log n\log\rho L)$
	Shi and Spencer [25]*	$O(n\rho^2\log n\log\rho+m)$	$O(\log n\log\rho)$	$O((m+n\rho)\log n)$	$O(\frac{n}{\rho}\log n)$
	Cohen [6]*	$O(m+n\rho^2\log^2 n)$	$O(\log^2 n)$	$O(n^2\log^3 n\log L)$	$O(\frac{n}{\rho}\log^3 n\log L)$
	*This work**	$O(m\log n+n\rho^2)$	$O(\rho\log\rho)$	$O((m+n\rho)\log n)$	$O(\frac{n}{\rho}\log n\log\rho L)$

* the presented bounds only apply to undirected graphs.

Table 7: Cost bounds for exact SSSP algorithms that have subcubic work, broken down into (i) the cost of preprocessing and (ii) the cost of only the shortest-path computation, where m is the number of edges in the initial input graph and n is the number of vertices in the initial input graph. The bounds already account for alterations made, if any, by preprocessing.

Symposium on Parallel Algorithms and Architectures (SPAA), pages 353–362. ACM, 1992.

[14] J. Jaja. *Introduction to Parallel Algorithms*. Addison-Wesley Professional, 1992.

[15] R. M. Karp and V. Ramachandran. *Parallel algorithms for shared-memory machines*. Handbook of theoretical computer science (vol. a): algorithms and complexity, 1991.

[16] P. N. Klein and S. Subramanian. A linear-processor polylog-time algorithm for shortest paths in planar graphs. *Symposium on Foundations of Computer Science (FOCS)*. IEEE, 1993.

[17] P. N. Klein and S. Subramanian. A randomized parallel algorithm for single-source shortest paths. *Journal of Algorithms*, 25(2):205–220, 1997.

[18] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. 2014.

[19] U. Meyer and P. Sanders. Δ-stepping: a parallelizable shortest path algorithm. *Journal of Algorithms*, 49(1), 2003.

[20] G. L. Miller, R. Peng, A. Vladu, and S. C. Xu. Improved parallel algorithms for spanners and hopsets. In *ACM Symposium on Parallelism in Algorithms and Architectures, SPAA*, pages 192–201, 2015.

[21] R. C. Paige and C. P. Kruskal. Parallel algorithms for shortest path problems. In *International Conference on Parallel Processing (ICPP)*, pages 14–20, 1985.

[22] H. Park and K. Park. Parallel algorithms for red–black trees. *Theoretical Computer Science*, 262(1):415–435, 2001.

[23] W. Paul, U. Vishkin, and H. Wagener. Parallel computation on 2-3-trees. *RAIRO, Informatique théorique*, 17(4):397–404, 1983.

[24] W. J. Paul, U. Vishkin, and H. Wagener. Parallel dictionaries in 2-3 trees. In *Intl. Colloq. on Automata, Languages and Programming (ICALP)*, pages 597–609, 1983.

[25] H. Shi and T. H. Spencer. Time-work tradeoffs of the single-source shortest paths problem. *Journal of Algorithms*, 30(1):19–32, 1999.

[26] T. H. Spencer. Time-work tradeoffs for parallel algorithms. *Journal of the ACM (JACM)*, 44(5):742–778, 1997.

[27] M. Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *Journal of the ACM (JACM)*, 46(3):362–394, 1999.

[28] J. L. Träff and C. D. Zaroliagis. A simple parallel algorithm for the single-source shortest path problem on planar digraphs. *Journal of Parallel and Distributed Computing* 60.9 (2000): 1103-1124.

[29] J. D. Ullman and M. Yannakakis. High-probability parallel transitive-closure algorithms. *SIAM Journal on Computing*, 20(1):100–125, 1991.

ρ	Road map of Pennsylvania					Web graph of Stanford					2D-grid				
	k				reduced steps	k				reduced steps	k				reduced steps
	2	**3**	**4**	**5**		**2**	**3**	**4**	**5**		**2**	**3**	**4**	**5**	
10	1.67	0.41	0.05	0.01	3.00	3.11	0.02	0.01	0.00	3.48	0.36	0.00	0.00	0.00	3.00
20	3.79	2.38	0.84	0.23	3.88	9.91	3.06	0.09	0.01	5.03	5.75	0.46	0.00	0.00	4.00
50	10.34	6.05	5.65	3.71	5.04	47.57	10.74	3.40	0.13	7.71	16.05	8.40	9.54	0.67	6.01
100	20.33	13.64	8.85	8.16	6.13	109.98	39.99	20.96	8.73	10.25	29.59	22.02	10.52	11.43	8.02
200	39.92	26.35	20.15	14.51	7.85	188.92	67.25	45.54	17.96	12.72	48.40	41.34	28.03	12.73	11.04
500	97.58	64.72	48.49	37.64	10.76	337.34	141.58	119.03	63.69	14.92	126.09	99.22	55.62	64.75	17.12
1000	192.00	127.45	95.55	75.84	13.74	529.14	208.66	219.21	149.20	15.17	243.12	181.50	129.26	108.37	23.18

Table 8: The ratio of created shortcut edges to the original edges when using the **greedy** heuristic as ρ and k are varied. The "reduced steps" columns are from Table 4 as a reference.

ρ	Road map of Pennsylvania					Web graph of Stanford					2D-grid				
	k				reduced steps	k				reduced steps	k				reduced steps
	2	**3**	**4**	**5**		**2**	**3**	**4**	**5**		**2**	**3**	**4**	**5**	
10	0.95	0.12	0.01	0.00	3.00	0.02	0.01	0.01	0.00	3.48	0.25	0.00	0.00	0.00	3.00
20	2.70	0.90	0.18	0.04	3.88	0.05	0.02	0.01	0.01	5.03	3.95	0.25	0.00	0.00	4.00
50	7.78	3.59	1.89	0.72	5.04	0.20	0.06	0.04	0.03	7.71	12.16	6.21	4.06	0.36	6.01
100	16.09	8.09	4.40	2.58	6.13	0.51	0.13	0.08	0.06	10.25	24.22	14.27	8.32	6.06	8.02
200	32.60	17.04	9.89	6.03	7.85	0.99	0.25	0.15	0.11	12.72	48.35	30.23	20.28	12.45	11.04
500	81.75	44.14	26.65	17.11	10.76	2.18	0.50	0.30	0.22	14.92	125.96	80.09	54.44	42.26	17.12
1000	162.91	89.30	54.82	35.95	13.74	3.92	0.66	0.34	0.24	15.17	241.30	154.97	110.87	84.87	23.18

Table 9: The ratio of created shortcut edges to the original edges when using the **DP** heuristic as ρ and k are varied. The "reduced steps" columns are from Table 4 as a reference.