Improved Algorithms for Orienteering and Related Problems

Chandra Chekuri* Nitish Korula † Martin Pál‡

Abstract

In this paper, we consider the orienteering problem in undirected and directed graphs and obtain improved approximation algorithms. The point to point-orienteering problem is the following: Given an edge-weighted graph $G = (V, E)$ (directed or undirected), two nodes $s, t \in V$ and a time limit B, find an s-t *walk* in G of total length at most B that maximizes the number of distinct nodes visited by the walk. This problem is closely related to tour problems such as TSP as well as network design problems such as k-MST. Orienteering with *time-windows* is the more general problem in which each node v has a specified time-window $[R(v), D(v)]$ and a node v is counted as visited by the walk only if v is visited during its time-window. We design new and improved algorithms for the orienteering problem and orienteering with time windows. Our main results are the following:

- A $(2 + \varepsilon)$ approximation for orienteering in undirected graphs, improving upon the 3-approximation of [6].
- An $O(\log^2 \mathrm{OPT})$ approximation for orienteering in directed graphs, where $\mathrm{OPT} \leq n$ is the number of vertices visited by an optimal solution. Previously, only a quasi-polynomial time algorithm due to [12] achieved a poly-logarithmic approximation (a ratio of $O(\log \mathrm{OPT})$).
- Given an α approximation for orienteering, we show an $O(\alpha \cdot \max\{\log \mathrm{OPT}, \log \frac{L_{\max}}{L_{\min}}\})$ approximation for orienteering with time windows, where $L_{\max}$ and $L_{\min}$ are the lengths of the longest and shortest time windows respectively.

1 Introduction

The traveling salesperson problem (TSP) and its variants have been an important driving force for the development of new algorithmic and optimization techniques. This is due to several reasons. First, the problems have many practical applications. Second, they are often simple to state and intuitively appealing. Third, for historical reasons, TSP has been a focus for trying new ideas. See [23, 21] for detailed discussion on various aspects of TSP. In this paper, we consider some TSP variants in which the goal is to find a tour or a walk that maximizes the number of nodes visited, subject to a strict time limit (also called budget) requirement. The main problem of interest is the ORIENTEERING problem [20][1] which we define formally below. The input to the problem consists of an edge-weighted graph $G = (V, E)$ (directed or undirected), two vertices $s, t \in V$ and a non-negative time limit B. The goal is to find an s-t walk of total length at most B so as to maximize the number of *distinct* vertices visited by the walk. Note that a vertex may be visited multiple times by the walk,

*Dept. of Computer Science, University of Illinois, Urbana, IL 61801. chekuri@cs.illinois.edu. Partially supported by an NSF grant CCF 0728782.

†Dept. of Computer Science, University of Illinois, Urbana, IL 61801. nkorula2@uiuc.edu. Partially supported by an NSF grant CCF 0728782.

‡Google Inc., 76 9th Avenue, New York, NY 10011. mpal@google.com

[1]The problems we describe are referred to by several different names in the literature, one of which is prize-collecting TSP.

but is only counted once in the objective function. (Alternatively, we could work with the metric completion of the given graph.) One could consider weighted versions of ORIENTEERING, where the goal is to maximize the sum of the weights of visited vertices; using standard scaling techniques, one can reduce the weighted version to the unweighted problem at the loss of a factor of $(1 + o(1))$ in the approximation ratio. Hence, we focus on the unweighted version throughout this paper. We use OPT to denote the number of vertices in an optimal solution; OPT can be as large as n, the number of vertices in the graph, but may be much smaller.

We also study the more general problem of orienteering with *time-windows*. In this problem, we are additionally given a time-window (or interval) $[R(v), D(v)]$ for each vertex v. A vertex is counted as visited only if the walk visits v at some time $t \in [R(v), D(v)]$. (If a vertex v is reached before $R(v)$, we may choose to "wait" at v until $R(v)$, so the walk can obtain credit for v, and then resume the walk. The time spent "waiting" is included in the length of the walk.) For ease of notation, we use ORIENT-TW to refer to the problem of orienteering with time-windows. A problem of intermediate complexity is the one in which $R(v) = 0$ for all v. We refer to this problem as orienteering with deadlines (ORIENT-DEADLINE); it has also been called the Deadline-TSP problem by [6]. The problem where vertices have release times but not deadlines (that is, $D(v) = \infty$ for all v) is essentially equivalent to ORIENT-DEADLINE.

One of the main motivations for budgeted/time-limited TSP problems comes from real world applications under the umbrella of vehicle routing; a large amount of literature on this topic can be found in operations research. Problems in this area arise in transportation, distribution of goods, scheduling of work, etc. Most problems that occur in practice have several constraints, and are often difficult to model and solve exactly. A recent book [25] discusses various aspects of vehicle routing. Another motivation for these problems comes from robot motion planning where typically, the planning problem is modeled as a Markov decision process. However there are situations where this does not capture the desired behaviour and it is more appropriate to consider Orienteering-type objective functions in which the reward at a site expires after the first visit; see [8], which discussed this issue and introduced the discounted-reward TSP problem. In addition to the practical motivation, budgeted TSP problems are of theoretical interest.

ORIENTEERING is NP-hard via a straightforward reduction from TSP and we focus on approximation algorithms; it is also known to be APX-hard to approximate [8]. The first non-trivial approximation algorithm for ORIENTEERING is due to Arkin, Mitchell and Narasimhan [2], who gave a $(2 + \varepsilon)$ approximation for points in the Euclidean plane. For points in arbitrary metric spaces, which is equivalent to the undirected case, [8] gave the first approximation algorithm with a ratio of 4; this was shortly improved to a ratio of 3 by [6]. More recently, [14] obtained a PTAS for points in fixed-dimensional Euclidean space. The basic insights for approximating ORIENTEERING were obtained in [8], where a related problem called the minimum-excess problem was defined. It was shown in [8] that an approximation for the min-excess problem implies an approximation for ORIENTEERING. Further, the min-excess problem can be approximated using algorithms for the k-stroll problem. In the k-stroll problem, the goal is to find a minimum length walk from s to t that visits at least k vertices. Note that the k-stroll problem and the ORIENTEERING problem are equivalent in terms of exact solvability but an approximation for one does not immediately imply an approximation for the other. Still, the clever reduction of [8] (via the intermediate min-excess problem) shows that an approximation algorithm for k-stroll implies a corresponding approximation algorithm (losing a small constant factor in the approximation ratio) for ORIENTEERING. The results in [8, 6] are based on existing approximation algorithms for k-stroll [18, 10] in undirected graphs. In directed graphs, no non-trivial algorithm is known for the k-stroll problem and the best previously known approximation ratio for ORIENTEERING was $O(\sqrt{\text{OPT}})$. A different approach was taken for the directed ORIENTEERING problem by [12]; the authors use a recursive greedy algorithm to obtain a $O(\log \text{OPT})$ approximation for ORIENTEERING and for several generalizations, but unfortunately the running time is *quasi-polynomial* in the input size.

In this paper, we obtain improved algorithms for ORIENTEERING and related problems in both undirected and directed graphs. Our main results are encapsulated by the following theorems.

Theorem 1.1 *For any fixed $\varepsilon > 0$, there is an algorithm with running time $n^{O(1/\varepsilon^2)}$ achieving a $(2+\varepsilon)$-approximation for* ORIENTEERING *in undirected graphs.*

Theorem 1.2 *There is an $O(\log^2 \mathrm{OPT})$-approximation for* ORIENTEERING *in directed graphs.*[2]

Orienteering with Time Windows: ORIENT-DEADLINE and ORIENT-TW are more difficult problems; in fact ORIENT-TW is NP-hard even on the line [26]. The recursive greedy algorithm of [12] mentioned previously applies to ORIENTEERING even when the reward function is a given monotone submodular set function[3] f on V, and the objective is to maximize $f(S)$ where S is the set of vertices visited by the walk. Several non-trivial problems, including ORIENT-TW, can be captured by using different submodular functions. Thus, the algorithm from [12] provides an $O(\log \mathrm{OPT})$ approximation for ORIENT-TW in directed graphs, but it runs in *quasi-polynomial* time. Therefore, we make the following natural conjecture:

Conjecture 1.3 *There is a polynomial time $O(\log \mathrm{OPT})$ approximation for* ORIENT-TW *in directed (and undirected) graphs.*

As can be seen from Table 1, even in undirected graphs the best ratio known previously for ORIENT-TW was $O(\log^2 \mathrm{OPT})$. Our primary motivation is to close the gap between the ratios achievable in polynomial and quasi-polynomial time respectively. We remark that the quasi-polynomial time algorithm in [12] is quite different from all the other polynomial time algorithms, and it does not appear easy to find a polynomial time equivalent. In this paper we make some progress in closing the gap, while also obtaining some new insights. An important aspect of our approach is to understand the complexity of the problem in terms of the maximum and minimum time-window lengths. Let $L(v) = D(v) - R(v)$ be the length of the time-window of v. Let $L_{\max} = \max_v L(v)$ and $L_{\min} = \min_v L(v)$. Our results depend on the ratio $L = L_{\max}/L_{\min}$[4]; our main result in this setting is the following theorem:

Theorem 1.4 *In directed and undirected graphs, there is an $O(\alpha \max\{\log \mathrm{OPT}, \log L\})$ approximation for* ORIENT-TW*, where α denotes the approximation ratio for* ORIENTEERING*.*

Our results for ORIENT-TW are stated in more detail in Section 5; note that for polynomially-bounded instances, Theorem 1.4 implies an $O(\log n)$ approximation. We define the parameter L following the work of [15]; they showed that a constant factor approximation is achievable in undirected graphs if all time-windows are of the same length (that is, $L = 1$) and the end points of the walk are not specified. We believe this is a natural parameter to consider in the context of time-windows. In many practical settings L is likely to be small, and hence, algorithms whose performance depends on L may be better than those that depend on other parameters. In [6] an $O(\log D_{\max})$ approximation is given for ORIENT-TW in undirected graphs where $D_{\max} = \max_v D(v)$ and only the start vertex s is specified (here it is assumed that all the input is integer valued). We believe that $L_{\max}$ is a better measure than $D_{\max}$ for ORIENT-TW; $L_{\max} \leq D_{\max}$ for all instances, and $L_{\max}$ is considerably smaller in many instances. Further, our algorithm applies to directed graphs while the algorithm in [6] is applicable only for undirected graphs. Finally, our algorithm is for the point-to-point version while the one in [6] does not guarantee that the walk ends at t.

[2] A similar result was obtained concurrently and independently by [24]. See related work for more details.

[3] A function $f : 2^V \to \mathcal{R}^+$ is a montone submodular set function if f satisfies the following properties: (i) $f(\emptyset) = 0$, $f(A) \leq f(B)$ for all $A \subseteq B$ and (ii) $f(A) + f(B) \geq f(A \cup B) + f(A \cap B)$ for all $A, B \subseteq V$.

[4] If $L_{\min} = 0$, consider the set of vertices which have zero-length time-windows. If this includes a significant fraction of the vertices of an optimal solution, use dynamic programming to get a $O(1)$-approximation. Otherwise, we can ignore these vertices and assume $L_{\min} > 0$ without losing a significant fraction of the optimal reward.

	$\mathbb{R}^d$	Undirected Graphs	Directed Graphs
ORIENTEERING	$1+\varepsilon$	$2+\varepsilon$	$O(\log^2 \mathrm{OPT})$
ORIENT-DEADLINE	$*$	$O(\log \mathrm{OPT})$†	$O(\log^3 \mathrm{OPT})$
ORIENT-TW	$*$	$O(\log^2 \mathrm{OPT})$† $O(\max\{\log \mathrm{OPT}, \log L\})$	$O(\log^4 \mathrm{OPT})$ $O(\log^2 \mathrm{OPT} \cdot \max\{\log \mathrm{OPT}, \log L\})$

Table 1: The best currrently known polynomial-time approximation ratios for ORIENTEERING and ORIENT-TW. The $1+\varepsilon$-approximation for ORIENTEERING in Euclidean Space ($\mathbb{R}^d$ for fixed dimension d) is due to [14]; the other entries for Euclidean space marked '$*$' have the same approximation ratio as the more general undirected graph problems. The two entries marked '†' in undirected graphs are due to [6]; all remaining entries are from this paper. The quasi-polynomial time algorithm of [12] gives an $O(\log \mathrm{OPT})$-approximation for *all* problems in this table.

Our results for ORIENT-TW are obtained using relatively simple ideas. Nevertheless, we believe that they are interesting, useful and shed more light on the complexity of the problem. In particular we are optimistic that some of these ideas will lead to an $O(\log n)$ approximation for the time-window problem in undirected graphs even when L is not poly-bounded.

Table 1 above summarizes the best known approximation ratios for ORIENTEERING and ORIENT-TW.

We now give a high level description of our technical ideas.

Overview of Algorithmic Ideas: For ORIENTEERING we follow the basic framework of [8], which reduces ORIENTEERING to k-stroll via the min-excess problem (formally defined in Section 2). We thus focus on the k-stroll problem.

In undirected graphs, [10] give a $2+\varepsilon$ approximation for the k-stroll problem. To improve the 3-approximation for ORIENTEERING via the method of [8], one needs a 2-approximation for the k-stroll problem with some additional properties. Unfortunately it does not appear that even current advanced techniques can be adapted to obtain such a result (see [18] for a more technical discussion of this issue). We get around this difficulty by giving a *bi-criteria* approximation for k-stroll. For k-stroll, let L be the length of an optimal path, and D be the shortest path in the graph from s to t. (Thus, the excess of the optimal path is $L-D$.) Our main technical result for k-stroll is a polynomial-time algorithm that, for any given $\varepsilon \geq 0$, finds an s-t walk of length at most $\max\{1.5D, 2L-D\}$ that contains at least $(1-\varepsilon)k$ vertices. For this, we prove various structural properties of near optimal k-strolls via the algorithm of [10], which in turn relies on the algorithm of [3] for k-MST. We also obtain a bi-criteria algorithm for min-excess.

For directed graphs, no non-trivial approximation algorithm is known for the k-stroll problem. In [12] the $O(\log \mathrm{OPT})$ approximation for ORIENTEERING is used to obtain an $O(\log^2 k)$ approximation for the k-TSP problem in quasi-polynomial time. In the k-TSP problem, the goal is to find a walk containing at least k vertices that begins *and ends* at a given vertex s; that is, k-TSP is the special case of k-stroll where $t=s$. Once again we focus on a bi-criteria approximation for k-stroll and obtain a solution of length 3OPT that visits $\Omega(k/\log^2 k)$ nodes. Our algorithm for k-stroll is based on an algorithm for k-TSP for which we give an $O(\log^3 k)$ approximation - for this we use simple ideas inspired by the algorithms for asymmetric traveling salesperson problem (ATSP) [16, 22] and an earlier poly-logarithmic approximation algorithm for k-MST [4].

For ORIENT-TW, we scale time window lengths so $L_{\min}=1$; our main insight is that (with a constant-factor loss in approximation ratio), the problem can be reduced to *either* a collection of ORIENT-DEADLINE instances (for which we use an $O(\log \mathrm{OPT})$-approximation), or an instance in which all release times and deadlines are integral and in which the longest window has length $L = L_{\max}/L_{\min}$. In the latter case, we note that windows of length at most L can be partitioned into $O(\log L)$ smaller windows whose lengths are powers

of 2, such that a window of length 2^i begins and ends at a multiple of 2^i. This allows us to decompose the instance into $O(\log L)$ instances of ORIENTEERING.

Related Work: We have already mentioned some of the related work in the discussion so far. The literature on TSP is vast, so we only describe some other work here that is directly relevant to the results in this paper. We first discuss undirected graphs. The ORIENTEERING problem seems to have been formally defined in [20]. [19] considered the prize-collecting Steiner tree and TSP problems (these are special cases of the more general version defined in [5]); in these problems the objective is to minimize the cost of the tree (or tour) plus a penalty for not visiting nodes. They used primal-dual methods to obtain a 2-approximation. This influential algorithm was used to obtain constant factor approximation algorithms for the k-MST, k-TSP and k-stroll problems [9, 17, 3, 18, 10], improving upon an earlier poly-logarithmic approximation [4]. As we mentioned already, the algorithms for k-stroll yield algorithms for ORIENTEERING [8]. ORIENT-TW was shown to be NP-hard even when the graph is a path [26]; for the path, [7] give an $O(\log \mathrm{OPT})$ approximation. The best known approximation for general undirected graphs is $O(\log^2 \mathrm{OPT})$, given by [6]; the ratio improves to $O(\log \mathrm{OPT})$ for the case of deadlines only [6]. A constant factor approximation can be obtained if the number of distinct time windows is fixed [11].

In directed graphs, the problems are less understood. For example, we have no non-trivial approximation for the k-stroll problem, though it is only known to be APX-hard. In [12] a simple recursive greedy algorithm that runs in quasi-polynomial time was shown to give an $O(\log \mathrm{OPT})$ approximation for ORIENTEERING and for ORIENT-TW. The algorithm also applies to the problem where the objective function is any given sub-modular functions on the vertices visited by the walk; several more complex problems can be captured by this generalization. Motivated by the lack of algorithms for the k-stroll problem, in [13] the asymmetric traveling salesperson path problem (ATSPP) was studied. ATSPP is the special case of k-stroll with $k = n$. Although closely related to the well studied ATSP problem, an approximation algorithm for ATSPP does not follow directly from that for ATSP. In [13] an $O(\log n)$ approximation is given for ATSPP.

In concurrent and independent work, [24] obtained an $O(\log^2 n)$ approximation for ORIENTEERING in directed graphs. They also use a bi-criteria approach for the k-stroll problem and obtain results essentially similar to those in this paper for directed graph problems, including rooted k-TSP. However their algorithm for (bi-criteria) k-stroll is based on an LP approach while we use a simple combinatorial greedy merging algorithm. Our ratios depend only on OPT or k while theirs depend also on n. On the other hand, the LP approach has some interesting features; in particular, an improved upper bound on the integrality gap of a natural LP relaxation for ATSP would directly lead to an improved approximation ratio for ORIENTEERING in directed graphs. (See [24] for more details.)

2 Preliminaries and Notation

Recall that in the k-stroll problem, we are given a graph $G(V, E)$, two vertices $s, t \in V$, and a target integer k; the goal is to find a minimum-length walk from s to t that visits at least k vertices. In [8], ORIENTEERING was reduced to the k-stroll problem; for completeness, we provide a brief description of this reduction. We adapt some of their technical lemmas for our setting.

Given a (directed or undirected) graph G, for any path P that visits vertices u, v (with u occurring before v on the path), we define $d^P(u, v)$ to be the distance along the path from u to v, and $d(u, v)$ to be the shortest distance in G from u to v. We define $\mathit{excess}^P(u, v)$ (the excess of P from u to v) to be $d^P(u, v) - d(u, v)$. We simplify notation in the case that $u = s$, the start vertex of the path P: we write $d^P(v) = d^P(s, v)$, $d(v) = d(s, v)$, and $\mathit{excess}^P(v) = \mathit{excess}^P(s, v)$.

If P is a path from s to t, the *excess of path* P is defined to be $\mathit{excess}^P(t)$. That is, the excess of a path is the difference between the length of the path and the distance between its endpoints. (Equivalently,

$length(P) = d(t) + excess^P(t)$.) In the min-excess path problem, we are given a graph $G = (V, E)$, two vertices $s, t \in V$, and an integer k; our goal is to find an s-t path of minimum-excess that visits at least k vertices. The path that minimizes excess clearly also has minimum total length, but the situation is slightly different for approximation. If x is the excess of the optimal path, an α-approximation for the minimum-excess problem has length at most $d(t) + \alpha x \leq \alpha(d(t) + x)$, and so it gives us an α-approximation for the minimum-length (i.e. the k-stroll) problem; the converse is not necessarily true. Below, we reduce the min-excess problem to k-stroll, and then reduce ORIENTEERING to min-excess.

2.1 From k-stroll to ORIENTEERING, via min-excess:

We first describe the algorithm due to [8] for the min-excess problem, given one for the k-stroll problem. If an optimal path P visits vertices in increasing order of their distance from s, we say that it is *monotonic*. The best monotonic path can be found via dynamic programming. In general, however, P may be far from monotonic; in this case, we break it up into continuous segments that are either monotonic, or have large excess. An optimal path in monotonic sections can be found by dynamic programming, and we use an algorithm for k-stroll in the large-excess sections. Intuitively, in these large-excess sections, the length of the path is comparable to its excess; therefore, a good approximation for k-stroll in these sections yields a good approximation for the min-excess problem. We formalize this intuition below.

For each real r, define $f(r)$ as the number of edges on the optimal path P with one endpoint at distance from s less than r, and the other endpoint at distance at least r from s. We partition the real line into maximal intervals with $f(r) = 1$ and $f(r) > 1$. (See Figure 1 below, essentially similar to that of [8].) Let b_i denote the left endpoint of the ith interval: An interval from b_i to b_{i+1} is of *type 1* (corresponding to a monotonic segment) if, for each r between b_i and b_{i+1}, $f(r) = 1$. The remaining intervals are of *type 2* (corresponding to segments with large excess).

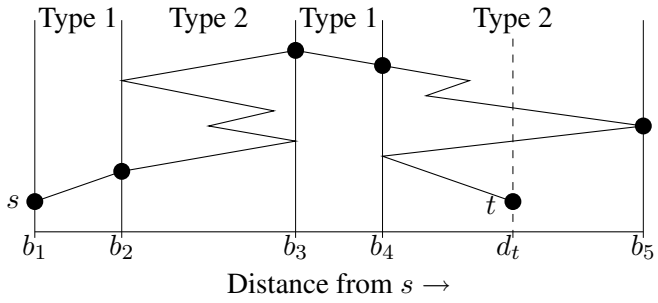

Figure 1: A breakdown of a path P into type 1 (monotonic) and type 2 (large-excess) segments. The solid vertical lines indicate segment boundaries, with dots corresponding to s and t, and the first and last vertex of each segment.

For each interval i, from vertex u (at distance b_i from s) to vertex v (at distance b_{i+1} from s), we define $ex(i)$ as the *increase* in excess that P incurs while going from u to v. (That is, $ex(i) = excess^P(v) - excess^P(u)$.) Also, we let ℓ_i be the length of P contained in interval i, and d_i be the length of the shortest path from u to v contained entirely in interval i. From our definitions, the overall excess of the optimal path P is given by $excess^P(t) = \sum_i ex(i)$. In [8], it is shown that in undirected graphs, for any type-2 interval i, $\ell_i \geq 3(b_{i+1} - b_i)$. (For the last interval, we instead obtain $\ell_i \geq 3(d(t) - b_i)$.) To see that this is true, note from Figure 1 that ℓ_i is at least the integral of $f(d)$ for each d between b_i and b_{i+1}. Since i is an interval of type 2, $f(d) \geq 2$; further, one can observe using a parity argument that $f(d) \geq 3$, since if P crosses distance d only twice, it must end at distance less than d. For the results of [8], it suffices to prove that the global excess, $excess(P)$, is at least $\frac{2}{3} \sum_{i \text{ of type } 2} \ell_i$, which follows from the previous argument. We need to refine this slightly in the following

lemma by bounding the local excess in each interval, instead of the global excess.

Lemma 2.1 *For any type-2 interval i of path P in an undirected graph, $ex(i) \geq \max\{\ell_i - d_i, \frac{2\ell_i}{3}\}$.*

Proof: We have:

$$\begin{aligned} ex(i) &= \big(d^P(v) - d(v)\big) - \big(d^P(u) - d(u)\big) \\ &= \big(d^P(v) - d^P(u)\big) - (d(v) - d(u)) \\ &= \ell_i - (b_{i+1} - b_i). \end{aligned}$$

(In the case of the last segment, containing t, the last equality should be $\ell_i - (d(t) - b_i)$.) For any type-2 segment, $\ell_i \geq 3(b_{i+1} - b_i)$ (or $3(d_t - b_i)$), so we have $ex(i) \geq \frac{2\ell_i}{3}$. Also, the shortest-path distance d_i from u to v contained in interval i is at least $b_{i+1} - b_i$. Therefore, $ex(i) \geq \ell_i - d_i$. □

We now briefly describe the dynamic-programming algorithm of [8] for min-excess: A vertex v belongs to interval i if its distance from s is greater than b_i and at most b_{i+1}. (Note that v may be any vertex of G, not necessarily one on an optimal path P.) For each interval that might be in an optimal solution, and for each reward that might be collected in this interval, find a short path using vertices of this interval that collects at least the desired reward. For each interval, find paths assuming that it is both a type-1 interval and a type-2 interval. In the former case, the optimal path is monotonic, so we can easily find it using a dynamic programming subroutine. In the latter case, we use an approximation algorithm for k-stroll to find a short path that collects at least the desired reward. Having found a good solution for each possible interval, one can "guess" the intervals of the optimal solution and stitch them together using a master dynamic program. Thus, the following lemma is proved in [8]; we omit the proof here, but we note that it is very similar to that of Lemma 2.4 for directed graphs which we prove subsequently, the only difference being that we can use Lemma 2.1 instead of the weaker Lemma 2.3 for directed graphs.

Lemma 2.2 ([8]) *In undirected graphs, a β-approximation to the k-stroll problem implies a $(\frac{3\beta}{2} - \frac{1}{2})$-approximation to the min-excess problem.*

Using very similar arguments, we can prove an analogous result for directed graphs. First, we need the equivalent of Lemma 2.1. In directed graphs, for each real r, we let $f(r)$ be the number of arcs a on the optimal path P such that the tail of a is at distance less than r from s, and the head of a is at distance at least r from s. All other definitions are identical to those in the undirected case. Now, we can only observe that $f(d)$ is at least 2 for all d in type 2 intervals. (As before, a parity argument implies that path P must cross distance d at least 3 times, but on one of these occasions, the tail of the arc will have distance at least d from s, while the head has distance less than d. Hence, this does not contribute to $f(d)$.) It is now easy to prove the required lemma:

Lemma 2.3 *For any type-2 interval i of path P in a directed graph, $ex(i) \geq \max\{\ell_i - d_i, \frac{\ell_i}{2}\}$.*

Proof: Exactly as in Lemma 2.1, we obtain $ex(i) = l_i - (b_{i+1} - b_i)$. Again, the shortest path distance d_i from the first vertex of the interval to the last is at least $b_{i+1} - b_i$, and so $ex(i) \geq l_i - d_i$. However, we can now only conclude that $l_i \geq 2(b_{i+1} - b_i)$. Therefore, $ex(i) \geq l_i/2$. □

We now reduce min-excess to k-stroll in directed graphs, similar to Lemma 2.2 in undirected graphs.

Lemma 2.4 *In directed graphs, a β-approximation to the k-stroll problem implies a $(2\beta - 1)$-approximation to the min-excess problem.*

Proof: Recall that for every type 1 interval, we can find the optimal path using dynamic programming, and for every type 2 interval, we use our approximation algorithm for k-stroll to find a short path that collects the reward we desire. Let L denote the total length of an optimal path P, L_1 denote the total length of P in type-1 segments, and L_2 the total length in type-2 segments. The path P' we find has total length at most $L_1+\beta L_2$. The excess of the optimal path P is $L-d(t)$, while the excess of our path P' is at most $L_1+\beta L_2-d(t) = L-d(t)+(\beta-1)L_2$. From Lemma 2.1, $\frac{L_2}{2} \leq \mathit{excess}(P)$. Hence, the excess of P' is at most $\mathit{excess}(P) + 2(\beta-1)\mathit{excess}(P)$. □

We now reduce ORIENTEERING to the min-excess problem. The following lemma, due to [6], applies to both directed and undirected graphs.

Lemma 2.5 ([6]) *A γ-approximation to the min-excess problem implies a $\lceil\gamma\rceil$-approximation for* ORIENTEERING.

Proof: Consider an optimal path P that visist OPT vertices, and break it into $h = \lceil\gamma\rceil$ consecutive segments $P_1, P_2, \ldots P_h$, each containing OPT/h vertices. Guess (that is, try all possible choices for) the first and last vertex of each segment. For each i, let s_i, t_i be the first and last vertices of segment P_i, and let ex_i, the local excess of P_i, be the difference between the length of P_i and the shortest-path distance from s_i to t_i. Let P_j be the segment with least excess; $h \cdot ex_j \leq \sum_{i=1}^{h} ex_i = d^P(s,t) - \sum_{i=1}^{h} d(s_i,t_i)$. Now, use the min-excess approximation algorithm to find a new s_j-t_j path P'_j that visits at least OPT/h vertices, and with excess at most $\gamma \leq h$ times that of P_j.

Finally, construct a path P' by going directly from s to s_j, follow P'_j from s_j to t_j, and then go directly from t_j to t. The total length of this path is at most $\sum_{i=1}^{h} d(s_i,t_i) + h \cdot ex_j \leq d^P(s,t) = length(P)$. Therefore, we have an $s-t$ path of length at most the given time limit, that visits at least $\mathrm{OPT}/\lceil\gamma\rceil$ vertices. □

The way in which our algorithms differ from those of [8] and [6] is that we use *bi-criteria approximations* for k-stroll. We say that an algorithm is an (α,β)-approximation to the k-stroll problem if, given a graph G, vertices $s,t \in V(G)$, and a target integer k, it finds a path which visits at least k/α vertices, and has length at most β times the length of an optimal path that visits k vertices.

Lemmas 2.4 and 2.5 can be easily extended to show that an (α,β)-approximation to the k-stroll algorithm for directed graphs gives an $(\alpha\lceil 2\beta-1\rceil)$-approximation for the ORIENTEERING problem in directed graphs. In Section 4, we use this fact, with a $(O(\log^2 k), 3)$-approximation for the k-stroll problem in directed graphs, to get an $O(\log^2 \mathrm{OPT})$-approximation for directed ORIENTEERING.[5] For undirected graphs, one might try to use Lemmas 2.2 and 2.5 with a $(1+\varepsilon, 2)$-approximation for the k-stroll problem, but this leads to a $((1+\varepsilon)\times\lceil 2.5\rceil) = (3+\varepsilon)$ approximation for ORIENTEERING. To obtain the desired ratio of $(2+\varepsilon)$, we need a refined analysis to take advantage of the particular bi-criteria algorithm that we develop for k-stroll; the details are explained in Section 3.

3 A $(2+\varepsilon)$-approximation for Undirected ORIENTEERING

In the k-stroll problem, given a metric graph G, with 2 specified vertices s and t, and a target integer k, we wish to find an s-t path of minimum length that visits at least k vertices. Let L be the length of an optimal such path, and D the shortest-path distance in G from s to t. In this section, we describe a *bi-criteria* approximation algorithm for the k-stroll problem, as guaranteed by the following theorem:

Theorem 3.1 *For any $\varepsilon > 0$, there is an algorithm with running time $O(n^{O(1/\varepsilon^2)})$ that, given a graph G, two vertices s and t and a target integer k, finds an s-t walk of length at most $\max\{1.5D, 2L-D\}$ that visits at least*

[5] When we use the k-stroll algorithm as a subroutine, we call it with $k \leq$ OPT, where OPT is the number of vertices visited by an optimum ORIENTEERING solution.

$(1-\varepsilon)k$ vertices, where L is the length of the optimal s-t path that visits k vertices and D is the shortest-path distance from s to t.

We prove Theorem 3.1 in Section 3.2; first, in Section 3.1, we describe the desired $(2+\varepsilon)$-approximation for ORIENTEERING in undirected graphs, assuming Theorem 3.1.

3.1 From k-stroll to minimum-excess

We solve the minimum-excess problem using essentially the algorithm of [8]; as explained in Section 2.1, the key difference is that instead of calling the k-stroll algorithm of [10] as a subroutine, we use the algorithm of Theorem 3.1 that returns a bi-criteria approximation. In addition, the analysis is slightly different, making use of the fact that our algorithm returns a path of length at most $\max\{1.5D, 2L-D\}$. In the arguments below, we fix an optimum path P, and chiefly follow the notation of [8].

Theorem 3.2 *For any fixed $\varepsilon > 0$, there is a polynomial-time algorithm to find an s-t path visiting at least $(1-\varepsilon)k$ vertices, with excess at most twice that of an optimal path P visiting k vertices.*

Proof: As described in Section 2, the algorithm uses dynamic programming similar to that in [8] with our bi-criteria k-stroll algorithm of Theorem 3.1 in place of an approximate k-stroll algorithm. Let P' be the path returned by our algorithm. Roughly speaking, P' will be at least as good as a path obtained by replacing the segment of P in each of its intervals by a path that the algorithm finds in that interval. In type-1 intervals the algorithm finds an optimum path because it is monotonic. In type-2 intervals we have a bi-criteria approximation that gives a $(1-\varepsilon)$ approximation for the number of vertices visited. This implies that P' contains at least $(1-\varepsilon)k$ vertices. To bound the excess, we sum up the lengths of the replacement paths to obtain:

$$\begin{aligned}
length(P') &\leq \sum_{i \text{ of type 1}} \ell_i + \sum_{i \text{ of type 2}} \max\{1.5d_i, 2\ell_i - d_i\} \\
&\leq \sum_i \ell_i + \sum_{i \text{ of type 2}} \max\{0.5\ell_i, \ell_i - d_i\} \\
&\leq \sum_i \ell_i + \sum_{i \text{ of type 2}} ex(i) \\
&\leq length(P) + excess^P(t) \\
&= d(t) + 2excess^P(t)
\end{aligned}$$

where the second inequality comes from rearranging terms and the fact that $d_i \leq \ell_i$, and the third inequality follows from Lemma 2.1. Therefore, the excess of P' is at most twice that of P, the optimal path. □

For completeness, we restate Lemma 2.5, modified for a bi-criteria excess approximation: An (α, β)-approximation to the min-excess problem gives an $\alpha\lceil\beta\rceil$-approximation to the ORIENTEERING problem.

Proof of Theorem 1.1. For any constant $\varepsilon > 0$, to obtain a $(2+\varepsilon)$-approximation for the undirected ORIENTEERING problem, first find ε' such that $2+\varepsilon = \frac{2}{1-\varepsilon'}$. Theorem 3.2 implies that there is a $(\frac{1}{1-\varepsilon'}, 2)$-bi-criteria approximation algorithm for the min-excess problem that runs in $n^{O(1/\varepsilon^2)}$ time. Now, we use (the bi-criteria version of) Lemma 2.5 to get a $\frac{2}{1-\varepsilon'} = (2+\varepsilon)$-approximation for ORIENTEERING in undirected graphs. □

It now remains only to prove Theorem 3.1, to which we devote the rest of this section.

3.2 Proof of Theorem 3.1

Given graph G, vertices s, t, and integer k, for any fixed $\varepsilon > 0$, we wish to find an s-t path that visits at least $(1-O(\varepsilon))k$ vertices, and has total length at most $\max\{1.5D, 2L-D\}$. Our starting point is the following

theorem on k-Stroll, proved by [10]:

Theorem 3.3 ([10]) *Given a graph G,two vertices s and t and a target integer k, let L be the length of an optimal path from s to t visiting k vertices. For any $\delta > 0$, there is a polynomial-time algorithm to find a tree of length at most $(1+\delta)L$ and containing at least k vertices, including both s and t.*

The algorithm of [10] *guesses* $O(1/\delta)$ vertices $s = w_1, w_2, w_3,$
$\ldots, w_{m-1}, w_m = t$ such that an optimal path P visits the guessed vertices in this order, and for any i, the distance from w_i to w_{i+1} along P is $\leq \delta L$. It then uses the k-MST algorithm of [3] with the given set of guessed vertices to obtain a tree satisfying Theorem 3.3; this tree is also guaranteed to contain all the guessed vertices. We can assume that all edges of the tree have length at most δL; longer edges can be subdivided without adding more than $O(1/\delta)$ vertices.

Our bi-criteria approximation algorithm for k-stroll begins by setting $\delta = \varepsilon^2$, and using the algorithm of Theorem 3.3 to obtain a k-vertex tree T containing s and t. We are guaranteed that $length(T) \leq (1+\delta)L$ (recall that L is the length of a shortest s-t path P visiting k vertices). Let $P^T_{s,t}$ be the path in T from s to t; we can double all edges of T not on $P^T_{s,t}$ to obtain a path P_T from s to t that visits at least k vertices. The length of the path P_T is $2length(T) - length(P^T_{s,t}) \leq 2length(T) - D$.
If either of the following conditions holds, the path P_T visits k vertices and has length at most $\max\{1.5D, 2L - D\}$, which is the desired result:

- The total length of T is at most $5D/4$. (In this case, P_T has length at most $3D/2$.)
- $length(P^T_{s,t}) \geq D + 2\delta L$. (In this case, P_T has length at most $2(1+\delta)L - (D + 2\delta L) = 2L - D$.)

We refer to these as the *easy doubling conditions*. Our aim will be to show that if neither of the easy doubling conditions applies, we can use T to find a new tree T' containing s and t, with length at most L, and with at least $(1 - O(\varepsilon))k$ vertices. Then, by doubling the edges of T' that are not on the s-t path (in T'), we obtain a path of length at most $2L - D$ that visits at least $(1 - O(\varepsilon))k$ vertices.

In the next subsection, we describe the structure the tree T must have if neither of the easy doubling conditions holds, and in Section 3.2.2, how to use this information to obtain the tree T'.

3.2.1 Structure of the Tree

If neither of the easy doubling conditions holds, then since D is at most $4/5$ of the length of T, and the length of $P^T_{s,t}$ is less than $D + 2\delta L$, the total length of the edges of $T \setminus P^T_{s,t}$ is greater than $(1/5 - 2\delta)L$. In this section, we describe how to construct the desired tree T' by removing a small piece of $T \setminus P^T_{s,t}$.

Say that a set of edges S in $T \setminus P^T_{s,t}$ is an *isolated* component if the total length of S is less than εL, and S is a connected component of $T \setminus P^T_{s,t}$.

Proposition 3.4 *We can greedily decompose the edge set of $T \setminus P^T_{s,t}$ into $\Omega(1/\varepsilon)$ disjoint pieces such that:*

- *Each piece is either a connected subgraph of or the union of isolated components of $T \setminus P^T_{s,t}$.*
- *Each piece has length in $[\varepsilon L, 3\varepsilon L)$, unless it is the union of* all *isolated components of $T \setminus P^T_{s,t}$ and has length less than εL.*

Proof: Consider the following greedy algorithm: root T at s, and consider a deepest node v in $T \setminus P^T_{s,t}$ such that the total length of edges in the subtree rooted at v is at least εL. If the total length of all edges in the subtree is

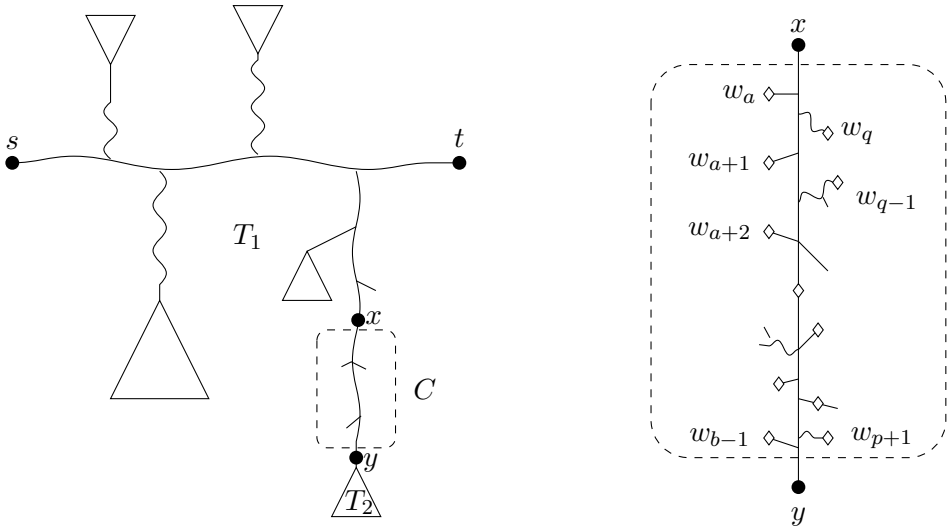

Figure 2: To the left is the tree T; a constant fraction of its length is not on $P^T_{s,t}$. We break these parts into pieces; the path-like piece C of degree 2, with fewer than $32\varepsilon k$ vertices, is shown in the box with the dashed lines. The right shows C in more detail, with vertices x and y at the head and foot of the spine, and guessed vertices shown as diamonds.

at most $2\varepsilon L$, this forms a piece that is connected and has the desired size. Otherwise, (arbitrarily) select enough children of v such that the total size of all their subtrees, together with their edges to v, is between εL and $2\varepsilon L$. (Since the subtree rooted at each child has size $< \varepsilon L$ and each edge has length $\leq \delta L \ll \varepsilon L$, this is always possible.) Again, this forms a piece that is connected and has the required size.

Now delete the edges of the piece just found from T, and recurse. When no more such pieces can be found, we may be left with parts of length $< \varepsilon L$ hanging off the s-t path. For any such part that has a further piece hanging off it, connect it to that piece, increasing its length to less than $3\varepsilon L$. The remaining parts are isolated components, and unless their total size is less than εL, it is easy to combine them arbitrarily into groups with total length in $[\varepsilon L, 3\varepsilon L]$. □

Let $\mathcal{T}$ be the tree formed as follows: We have one vertex s' for $P^T_{s,t}$ and one vertex for each of the pieces of Proposition 3.4. (Thus, $\mathcal{T}$ has $\Omega(1/\varepsilon)$ vertices.) There is an edge between vertices $v_1, v_2 \in V(\mathcal{T})$ corresponding to edge sets S_1, S_2 iff S_1 contains the parent edge in T of a minimum-depth edge in S_2, or vice versa. (In the special case that $v_1 = s'$, and the minimum-depth edge in S_2 is incident in T to s, we add the edge between $v_1 = s'$ and v_2.) Note that any piece containing isolated components becomes a leaf of $\mathcal{T}$ adjacent to s'.

Proposition 3.5 *The tree $\mathcal{T}$ contains a vertex of degree 1 or 2 that corresponds to a piece with length in $[\varepsilon L, 3\varepsilon L)$, and containing at most $32\varepsilon k$ vertices of the original tree T that are not contained in other pieces.*

Proof: The number of vertices in $\mathcal{T}$ (not including s'), is at least $\frac{(1/5-2\delta)L}{3\varepsilon L} = \frac{1}{15\varepsilon} - \frac{2\varepsilon}{3} \geq \frac{1}{16\varepsilon}$. At least one more than half these vertices have degree 1 or 2, since $\mathcal{T}$ is a tree. If the union of all isolated components has size less than εL, we discard the vertex corresponding to this piece; we are left with at least $1/(32\varepsilon)$ vertices of degree 1 or 2. If each of them corresponds to a piece that has more than $32\varepsilon k$ vertices not in other pieces, the total number of vertices they contain is more than k, which is a contradiction. □

If $\mathcal{T}$ has a leaf that corresponds to a piece with at most $32\varepsilon k$ vertices, we delete this piece from T, giving us a tree T' with length at most $(1+\delta)L - \varepsilon L < L$, with at least $(1-32\varepsilon)k$ vertices. Doubling the edges of T' not on its s-t path, we obtain an s-t walk that visits $(1-32\varepsilon)k$ vertices and has length at most $2L - D$, and we are done.

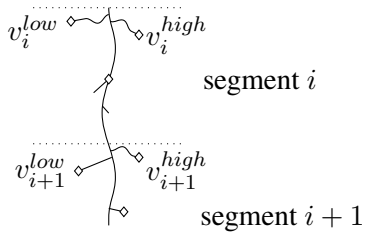

Figure 3: Two consecutive segments.

If there does not exist such a leaf, we can find a vertex of degree 2 in $\mathcal{T}$, corresponding to a connected subgraph/piece C of $T \setminus P_{s,t}^T$, with length ℓ in $[\varepsilon L, 3\varepsilon L)$, and at most $32\varepsilon k$ vertices. Deleting C from T gives us two trees T_1 and T_2; let T_1 be the tree containing s and t. We can reconnect the trees using the shortest path between them. If the length of this path is at most $\ell - \delta L$, we have a new tree T' with length at most L, and containing at least $(1 - 32\varepsilon)k$ vertices. In this case, as before, we are done.

Therefore, we now assume that the shortest path in G that connects T_1 and T_2 has length greater than $\ell - \delta L$, and use this fact repeatedly. (Recall that the total length of piece C is ℓ.) One consequence of this fact is that the piece C is *path-like*. That is, if x and y are the two vertices of $T - C$ with edges to C, the length of the path in C from x to y is more than $\ell - \delta L$; we refer to this path from x to y as the *spine* of the piece. (See Figure 2.) It follows that the total length of edges in C that are not on the spine is less than δL. We also refer to the vertex $x \in T_1$ adjacent to C as the *head* of the spine, and $y \in T_2$ adjacent to C as the *foot* of the spine. Finally, we say that for any vertices $p, q \in C$, the *distance along the spine* between vertices p and q is the length of those edges on the path between p and q that lie on the spine.

We assume for the moment that T_2 contains at least one vertex that was guessed by the algorithm of Theorem 3.3. Consider the highest-numbered guessed vertex w_p in T_2; where is the next guessed vertex w_{p+1}? It is not in T_2 by definition, nor in T_1 because the shortest path from T_2 to T_1 has length at least $\ell - \delta L$, and the edge $w_p w_{p+1}$ has length $\leq \delta L$. Therefore, it must be in C. Similarly, since $\delta L \ll l - \delta L$, the guessed vertices $w_{p+2}, w_{p+3}, \ldots$ must be in C. (In fact, there must be at least $\frac{\ell - \delta L}{\delta L} = \Omega(1/\varepsilon)$ such consecutive guessed vertices in C.) Let w_q be the highest-numbered of these consecutive guessed vertices in C.

By an identical argument, if w_b is the lowest-numbered guessed vertex in T_2, $w_{b-1}, w_{b-2}, \ldots$ must be in C. Let w_a be the lowest-numbered of these consecutive guessed vertices, so $w_a, w_{a+1}, \ldots w_{b-2}, w_{b-1}$ are all in C.

Remark 3.6 *If T_2 does not contain any guessed vertices, the procedure above is to be modified by finding the guessed vertex w nearest the foot of the spine. Remove from C the path from w to the foot, and those branches off the spine adjacent to this path; add these edges to the tree T_2. Now, T_2 contains a guessed vertex and we may continue; this does not change our proof in any significant detail.*

We now break up the piece C into *segments* as follows: Starting from x, the head of the spine, we cut C at distance $10\delta L$ along the spine from x. We repeat this process until the foot of the spine, obtaining at least $\frac{\ell - \delta L}{10\delta L} \geq \frac{1}{10\varepsilon} - \frac{1}{10}$ segments. We discard the segment nearest x and the two segments nearest y, and number the remaining segments from 1 to r consecutively from the head; we have at least $\frac{1}{10\varepsilon} - \frac{1}{10} - 3 \geq \frac{1}{15\varepsilon}$ segments remaining. For each segment, we refer to the end nearer x (the head of the spine) as the *top* of the segment, and the end nearer y as the *bottom* of the segment.

We now restrict our attention to guessed vertices in the range w_a to w_{b-1} and w_{p+1} through w_q. For each segment i, define v_i^{low} to be the lowest-numbered guessed vertex in segments i through r, and v_i^{high} to be the highest-numbered guessed vertex in segments i through r. (See Figure 3.)

Lemma 3.7 *For each i:*

1. *v_i^{low} occurs before v_{i+1}^{low} in the optimal path, and v_i^{high} occurs after v_{i+1}^{high} in the optimal path.*
2. *the distance along the spine from the top of segment i to each of v_i^{low} and v_i^{high} is at most $2\delta L$.*
3. *the distance between v_i^{low} and v_{i+1}^{low}, is at least $7\delta L$; the distance between v_i^{high} and v_{i+1}^{high} is at least $7\delta L$.*

Proof: We prove the statements for v_i^{low} and v_{i+1}^{low}; those for v_i^{high} and v_{i+1}^{high} are symmetric. Our proofs repeatedly use the fact (referred to earlier) that the shortest path from x to y does not save more than δL over ℓ, the length of C.

First, we claim that each segment contains some guessed vertex between w_a and w_{b-1}. Suppose some segment i did not; let c be the first index greater than or equal to a such that w_c is not above segment i in the tree. (Since w_a is above segment i, and w_b below it, we can always find such an index c.) Therefore, w_{c-1} is above segment i, and w_c below it. We can now delete segment i, and connect the tree up using the edge between w_{c-1} and w_c; this edge has length at most δL. But this gives us a path from x to y of length at most $\ell - 10\delta L + \delta L$, which is a contradiction.

Now, let v_i^{low} be the guessed vertex w_j; we claim that it is in segment i. Consider the location of the guessed vertex w_{j-1}. By definition, it is not in segments i through r; it must then be in segments 1 through $i-1$. If w_j were not in segment i, we could delete segment i (decreasing the length by $10\delta L$) and connect x and y again via the edge between w_j and w_{j-1}, which has length at most δL. Again, this gives us a path that is shorter by at least $9\delta L$, leading to a contradiction. Therefore, for all i, v_i^{low} is in segment i.

Because the lowest-numbered guessed vertex in segments i through r is in segment i, it has a lower number than the lowest-numbered guessed vertex in segments $i+1$ through r. That is, v_i^{low} occurs before v_{i+1}^{low} on the optimal path, which is the first part of the lemma.

We next prove that for all i, the distance along the spine from v_i^{low} to the top of segment i is at most $2\delta L$. If this is not true, we could delete the edges of the spine from v_i^{low} to the top of segment i, and connect v_i^{low} to the previous guessed vertex, which must be in segment $i-1$. The deletion decreases the length by at least $2\delta L$, and the newly added edge costs at most δL, giving us a net saving of at least δL; as before, this is a contradiction.

The final part of the lemma now follows, because we can delete the edges of the spine from v_i^{low} to the bottom of the segment (decreasing our length by at least $8\delta L$), and if the distance from v_i^{low} to v_{i+1}^{low} were less than $7\delta L$, we would save at least δL, giving a contradiction. □

Now, for each segment i, define $gain(i)$ to be the sum of the reward collected by the optimal path between v_i^{low} and v_{i+1}^{low} and the reward collected by the optimal path between v_{i+1}^{high} and v_i^{high}. Since these parts of the path are disjoint, $\sum_i gain(i) \leq k$, and there are at least $\frac{1}{15\varepsilon}$ such segments, there must exist some i such that $gain(i) \leq 15\varepsilon k$. By enumerating over all possibilities, we can find such an i.

3.2.2 Contracting the Graph

We assume we have found a segment numbered i such that $gain(i) \leq 15\varepsilon k$. Consider the new graph H formed from G by contracting together the 4 vertices v_i^{low}, v_i^{high}, v_{i+1}^{low} and v_{i+1}^{high} of G to form a new vertex v'; we prove the following proposition.

Proposition 3.8 *The graph H has a path of length at most $L - 14\delta L$ that visits at least $(1-15\varepsilon)k$ vertices.*

Proof: Consider the optimal path P in G, and modify it to find a path P_H in H by shortcutting the portion of the path between v_i^{low} and v_{i+1}^{low}, and the portion of the path between v_{i+1}^{high} and v_i^{high}. Since $gain(i) \leq 15\varepsilon k$,

the new path P_H visits at least $(1-15\varepsilon)k$ vertices. Further, since the shortest-path distance from v_i^{low} to v_{i+1}^{low} and the shortest-path distance from v_i^{high} to v_{i+1}^{high} are each $\geq 7\delta L$, path P_H has length at most $L - 14\delta L$. □

Using the algorithm of [3], we can find a tree T_H in H of total length at most $L - 13\delta L$ with at least $(1-15\varepsilon)k$ vertices. This tree T_H may not correspond to a tree of G (if it uses the new vertex v'). However, we claim that we can find a tree T_i in G of length at most $13\delta L$, that includes each of v_i^{low}, v_i^{high}, v_{i+1}^{low}, v_{i+1}^{high}. We can combine the two trees T_H and T_i to form a tree T' of G, with total length L.

Proposition 3.9 *There is a tree T_i in G containing v_i^{low}, v_i^{high}, v_{i+1}^{low} and v_{i+1}^{high}, of total length at most $13\delta L$.*

Proof: We use all of segment i, and enough of segment $i+1$ to reach v_{i+1}^{low} and v_{i+1}^{high}. The edges of segment i along the spine have length $\leq 10\delta L$, v_{i+1}^{low} and v_{i+1}^{high} each have distance along the spine at most $2\delta L$ from the top of segment $i+1$ (by Lemma 3.7). Finally, the total length of *all* the edges in the piece C not on the spine is at most δL. Therefore, to connect all of v_i^{low}, v_i^{high}, v_{i+1}^{low} and v_{i+1}^{high}, we must use edges of total length at most $(10+2+1)\delta L = 13\delta L$. □

We can now complete the proof of Theorem 3.1:

Proof of Theorem 3.1. Set $\varepsilon' = \varepsilon/32$ and run the algorithm of [10] with $\delta = \varepsilon'^2$ to obtain a k-vertex tree T of length at most $(1+\delta)L$. If either of the easy doubling conditions holds, we can double all the edges of T not on its s-t path to obtain a new s-t walk visiting k vertices, with length at most $\max\{1.5D, 2L-D\}$.

If neither of the easy doubling conditions holds, use T to obtain T' containing s and t, with length at most L and at least $(1-32\varepsilon')k$ vertices. Doubling edges of T' not on its s-t path, we find a new s-t path visiting $(1-32\varepsilon')k = (1-\varepsilon)k$ vertices, of length at most $2L - D$. □

4 ORIENTEERING in Directed Graphs

We give an algorithm for ORIENTEERING in directed graphs, based on a bi-criteria approximation for the (rooted) k-TSP problem: Given a graph G, a start vertex s, and an integer k, find a cycle in G of minimum length that contains s and visits k vertices. We assume that G always contains such a cycle; let OPT be the length of a shortest such cycle. We assume knowledge of the value of OPT, and that G is complete, with the arc lengths satisfying the asymmetric triangle inequality.

Our algorithm finds a cycle in G containing s that visits at least $k/2$ vertices, and has length at most $O(\log^2 k) \cdot \text{OPT}$. The algorithm gradually builds up a collection of strongly connected components. Each vertex starts as a separate component, and subsequently components are merged to form larger components. The main idea of the algorithm is to find low density cycles that visit multiple components, and use such cycles to merge components. (The density of a cycle C is defined as its length divided by the number of vertices that it visits; there is a polynomial-time algorithm to find a minimum-density cycle in directed graphs.) While merging components, we keep the invariant that each component is strongly connected and *Eulerian*, that is, each arc of the component can be visited exactly once by a single closed walk.

We note that this technique is similar to the algorithms of [16, 22] for ATSP; however, the difficulty is that a k-TSP solution need not visit all vertices of G and the algorithm is unaware of the vertices to visit. We deal with this using two tricks. First, we force progress by only merging components of similar size, hence ensuring that each vertex only participates in a logarithmic number of merges — when merging two trees or lists, one can charge the cost of merging to the smaller side, however when merging multiple components via a cycle, there is no useful notion of a smaller side. Second, we are more careful about picking representatives for each component; picking an arbitrary representative vertex from a component does not work. A variant that does work is to contract each component to a single vertex, however, this loses an additional logarithmic factor in the

BUILDCOMPONENTS:
for (each i in $\{0, 1, \ldots, \lfloor \log_2(k/4\log_2 k)\rfloor\}$) do:
  For each component in tier i
    (Arbitrarily) assign each vertex a distinct color in $\{1, \ldots, 2^{i+1} - 1\}$.
  Let $\{\mathcal{V}_j^i \mid j = 1, \ldots, 2^{i+1} - 1\}$ be the resulting color classes.
  Let H_j^i be the subgraph of G induced by the vertex set $\mathcal{V}_j^i$.
  While (there is a cycle C of density at most $\alpha \cdot 2^i$ in some graph H_j^i)
    Let $v_1, \ldots, v_l$ be the vertices of H_j^i visited by C
    Let v_p belong to component $\mathcal{C}_p$, $1 \leq p \leq l$
    (Two vertices of H_j^i never share a component, so $\mathcal{C}_1, \ldots, \mathcal{C}_l$ are distinct.)
    Form a new component $\mathcal{C}$ by merging $\mathcal{C}_1, \ldots, \mathcal{C}_l$ using C
    ($\mathcal{C}$ must belong to a higher tier)
    Remove all vertices of $\mathcal{C}$ from the graphs $H_{j'}^i$ for $j' \in \{1, \ldots, 2^{i+1} - 1\}$.

approximation ratio since an edge in a contracted vertex may have to be traversed a logarithmic number of times in creating a cycle in the original graph. To avoid this, our algorithm ensures components are Eulerian. One option is to pick a representative from a component *randomly* and one can view our coloring scheme described below as a derandomization.

We begin by pre-processing the graph to remove any vertex v such that the sum of the distances from s to v and v to s is greater than OPT; such a vertex v obviously cannot be in an optimum solution. Each remaining vertex initially forms a component of size 1. As components combine, their sizes increase; we use $|X|$ to denote the size of a component X, i.e. the number of vertices in it. We assign the components into tiers by size; components of size $|X|$ will be assigned to tier $\lfloor \log_2 |X| \rfloor$. Thus, a tier i component has at least 2^i and fewer than 2^{i+1} vertices; initially, each vertex is a component of tier 0. For ease of notation, we use α to denote the quantity $4 \log k \cdot \text{OPT}/k$.

In the main phase of the algorithm, we will iteratively push components into higher tiers, until we have enough vertices in large components, that is, components of size at least $k/4 \log k$. The procedure BUILDCOMPONENTS (above) implements this phase. Once we have amassed at least $k/2$ vertices belonging to large components, we finish by attaching a number of these components to the root s via direct arcs. Before providing the details of the final phase of the algorithm, we establish some properties of the algorithm BUILDCOMPONENTS.

Lemma 4.1 *Throughout the algorithm, all components are strongly connected and Eulerian. If any component X was formed by combining components of tier i, the sum of the lengths of arcs in X is at most $(i+1)\alpha|X|$.*

Proof: Whenever a component is formed, the newly added arcs form a cycle in G. It follows immediately that every component is strongly connected and Eulerian. We prove the bound on arc lengths by induction.

Let C be the low-density cycle found on vertices $v_1, v_2, \ldots v_l$ that connects components of tier i to form the new component X. Let $\mathcal{C}_1, \mathcal{C}_2, \ldots \mathcal{C}_l$ be the components of tier i that are combined to form X. Because the density of C is at most $\alpha 2^i$, the total length of the arcs in C is at most $\alpha 2^i l$. However, each tier i component has at least 2^i vertices, and so $|X| \geq 2^i l$. Therefore, the total length of arcs in C is at most $\alpha|X|$.

Now, consider any component $\mathcal{C}_h$ of tier i; it was formed by combining components of tier at most $i-1$, and so, by the induction hypothesis, the total length of all arcs in component $\mathcal{C}_h$ is at most $i\alpha|\mathcal{C}_h|$. Therefore, the total length of all arcs in all the components combined to form X is $i\alpha \sum_{h=1}^{l} |\mathcal{C}_h| = i\alpha|X|$. Together with the newly added arcs of C, which have weight at most $\alpha|X|$, the total weight of all arcs in component X is at most $(i+1)\alpha|X|$. □

Let O be a fixed optimum cycle, and let $o_1, \ldots, o_k$ be the vertices it visits.

Lemma 4.2 *At the end of iteration i of* BUILDCOMPONENTS*, at most $\frac{k}{2\log k}$ vertices of O remain in components of tier i.*

Proof: Suppose that more than $\frac{k}{2\log k}$ vertices of O remain in tier i at the end of the ith iteration. We show a low-density cycle in one of the graphs H^i_j, contradicting the fact that the while loop terminated because it could not find any low-density cycle: Consider the color classes $\mathcal{V}^i_j$ for $j \in \{1, \ldots, 2^{i+1} - 1\}$. By the pigeonhole principle, one of these classes has to contain more than $k/(2\log k \cdot 2^{i+1})$ vertices of O.[6] We can "shortcut" the cycle O by visiting only these vertices; this new cycle has cost at most OPT and visits at least two vertices. Therefore, it has density less than $(2^{i+2} \cdot \text{OPT} \log k)/k$, which is $2^i \cdot \alpha$. Hence, the while loop would not have terminated. □

We call a component *large*, if it has at least $k/4\log k$ vertices. Since we lose at most $\frac{k}{2\log k}$ vertices of O in each iteration, and there are fewer than $\log k$ iterations, we must have at least $k/2$ vertices of O in large components after the final iteration.

Theorem 4.3 *There is an $O(n^4)$-time algorithm that, given a directed graph G and a vertex s, finds a cycle with $k/2$ vertices rooted at s, of length $O(\log^2 k)$*OPT*, where* OPT *is the length of an optimum k-TSP tour rooted at s.*

Proof: Run the algorithm BUILDCOMPONENTS, and consider the large components; at least $k/2$ vertices are contained in these components. Greedily select large components until their total size is at least $k/2$; we have selected at most $2\lfloor(\log k)\rfloor$ components. For each component, pick a representative vertex v arbitrarily, and add arcs from s to v and v to s; because of our pre-processing step (deleting vertices far from s), the sum of the lengths of newly added arcs for each representative is at most OPT. Therefore, the total length of newly added arcs (over all components) is at most $2\log k$OPT. The large components selected, together with the newly added arcs, form a connected Eulerian component H, containing s. Let $k' \geq k/2$ be the number of vertices of H. From Lemma 4.1, we know that the sum of the lengths of arcs in H (not counting the newly added arcs) is at most $(\log k - 1)\alpha k'$. With the newly aded arcs, the total length of arcs of H is at most $4\log^2 k$OPT $\times\, k'/k$. Since H is Eulerian, there is a cycle of at most this length that visits each of the k' vertices of H.

If, from this cycle, we pick a segment of $k/2$ consecutive vertices uniformly at random, the expected length of this segment will be $2\log^2 k$OPT. Hence, the shortest segment containing $k/2$ vertices has length at most $2\log^2 k$OPT. Concatenate this with the arc from s to the first vertex of this segment (paying at most OPT), and the arc (again of cost $\leq$ OPT) from the last vertex to s; this gives us a cycle that visits at least $k/2$ vertices, and has cost less than $3\log^2 k \cdot$ OPT.

The running time of this algorithm is dominated by the time to find minimum-density cycles, each of which takes $O(nm)$ time [1], where n and m are the number of vertices and edges respectively. The algorithm makes $O(n)$ calls to the cycle-finding algorithm which implies the desired $O(n^4)$ bound. □

By using the algorithm from Theorem 4.3 greedily $\log k$ times, we obtain the following corollary.

Corollary 4.4 *There is an $O(\log^3 k)$ approximation for the rooted k-TSP problem in directed graphs.*

Theorem 4.5 *There is an $O(n^4)$-time algorithm that, given a directed graph G and nodes s, t, finds an s-t path of length* 3OPT *containing $\Omega(k/\log^2 k)$ vertices, where* OPT *is the length of an optimal k-stroll from s to t.*

Proof: We pre-process the graph as before, deleting any vertex v if the sum of the distance from s to v and the distance from v to t is greater than OPT. In the remaining graph, we consider two cases: If the distance from

[6] The largest value of i used is such that $k/2\log k \cdot 2^{i+1} \geq 1$, so there are always at least 2 vertices in this color class.

t to s is at most OPT, we leave the graph unmodified. Otherwise, we add a 'dummy' arc from t to s of length OPT. Now, there is a cycle through s that visits at least k vertices, and has length at most 2OPT. We use the previous theorem to find a cycle through s that visits $k/2$ vertices and has length less than $6\log^2 k\text{OPT}$. Now, break this cycle up into consecutive segments, each containing $\lfloor k/(12\log^2 k)\rfloor$ vertices (except possibly the last, which may contain more). One of these segments has length less than OPT; it follows that this part cannot use the newly added dummy arc. We obtain a path from s to t by beginning at s and taking the shortest path to the first vertex in this segment; this has length at most OPT. We then follow the cycle until the last vertex of this segment (again paying at most OPT), and then take the shortest path from the last vertex of the segment to t. The total length of this path is at most 3OPT, and it visits at least $\lfloor k/(12\log^2 k)\rfloor$ vertices. □

We can now complete the proof of Theorem 1.2, showing that there is an $O(\log^2 \text{OPT})$ approximation for ORIENTEERING in directed graphs.

Proof of Theorem 1.2. As mentioned in Section 2, Lemmas 2.4 and 2.5 can be extended to show that an (α, β)-bi-criteria approximation to the directed k-stroll problem can be used to get an $(\alpha \cdot \lceil 2\beta - 1\rceil)$-approximation to the ORIENTEERING problem on directed graphs. Theorem 4.5 gives us a $(O(\log^2 k, 3)$-approximation to the the directed k-stroll problem, which implies that there is a polynomial-time $O(\log^2 \text{OPT})$-approximation algorithm for the directed ORIENTEERING problem. □

5 ORIENTEERING with Time Windows

Much of the prior work on ORIENT-TW, following [6], can be cast in the following general framework: Use combinatorial methods to reduce the problem to a collection of sub-problems where the time-windows can be ignored. Each sub-problem has a subset of vertices V', start and end vertices $s', t' \in V'$, and a time-interval I in which we must travel from s' to t', visiting as many vertices of V' within their time windows as possible. However, the sub-problem is constructed such that the time-window for every vertex in V' entirely contains the interval I. Therefore, the sub-problem is really an instance of ORIENTEERING (without time-windows). An approximation algorithm for ORIENTEERING can be used to solve each sub-problem, and these solutions can be pasted together using dynamic programming. The next subsection describes this framework in more detail.

We use the same general framework; as a consequence, our results apply to both directed and undirected graphs; while solving a sub-problem we use either the algorithm for ORIENTEERING on directed graphs, or the algorithm for undirected graphs. Better algorithms for either of these problems would immediately translate into better algorithms for ORIENT-TW.

Subsequently, we use α to denote the approximation ratio for ORIENTEERING, and state our results in terms of α; from the previous sections, α is $O(1)$ for undirected graphs and $O(\log^2 \text{OPT})$ for directed graphs.

Recall that $L_{\max}$ and $L_{\min}$ are the lengths of the longest and shortest time time-windows respectively, and L is the ratio $\frac{L_{\max}}{L_{\min}}$. We first provide two algorithms with the following guarantees:

- $O(\alpha \log L_{\max})$, if the release time and deadline of every vertex are integers.
- $O(\alpha \log \text{OPT})$, if $L \leq 2$.

We note that the first algorithm is already an improvement over the $O(\log D_{\max})$-approximation of [6], and as mentioned in the introduction, our algorithm has the advantages that it can also be used in directed graphs, and is for the point-to-point version of the problem. The second algorithm immediately leads to an $O(\alpha \cdot \log \text{OPT} \times \log L)$-approximation for the general time-window problem, which is already an improvement on $O(\alpha \log^2 \text{OPT})$ when the ratio L is small. However, we can combine the first and second algorithms to obtain an $O(\alpha \max\{\log \text{OPT}, \log L\})$-approximation for ORIENT-TW.

Throughout this section, we use $R(v)$ and $D(v)$ to denote (respectively) the release time and deadline of a vertex v. We also use the word *interval* to denote a time window; $I(v)$ denotes the interval $[R(v), D(v)]$. Typically, we use 'time-window' when we are interested in the start and end points of a window, and 'interval' when we think of a window as an interval along the 'time axis'. For any instance X of ORIENT-TW, we let $\text{OPT}(X)$ denote the reward collected by an optimal solution for X. When the instance is clear from context, we use OPT to denote this optimal reward.

5.1 The General Framework

As described at the beginning of this section, the general method to solve ORIENT-TW is to reduce the problem to a set of sub-problems without time-windows. Given an instance of ORIENT-TW on a graph $G(V, E)$, suppose $V_1, V_2, \ldots V_m$ partition V, and we can associate times R_i and D_i with each V_i such that each of the following conditions holds:

- For each $v \in V_i$, $R(v) \leq R_i$ and $D(v) \geq D_i$.
- For $1 \leq i < m$, $D_i < R_{i+1}$.
- An optimal solution visits any vertex in V_i during $[R_i, D_i]$.

Then, we can solve an instance of ORIENTEERING in each V_i separately, and combine the solutions using dynamic programming. The approximation ratio for such "composite" solutions would be the same as the approximation ratio for ORIENTEERING. We refer to an instance of ORIENT-TW in which we can construct such a partition of the vertex set (and solve the sub-problems separately) as a *modular instance*. Subsection 5.1.1 describes a dynamic program that can solve modular instances.

Unfortunately, given an arbitrary instance of ORIENT-TW, it is unlikely to be a modular instance. Therefore, we define restricted versions of a given instance:

Definition 5.1 *Let A and B be instances of* ORIENT-TW *on the same underlying graph (with the same edge-weights), and let $I_A(v)$ and $I_B(v)$ denote the intervals for vertex v in instances A and B respectively. We say that B is a* restricted version *of A if, for every vertex v, $I_B(v)$ is a sub-interval of $I_A(v)$.*

Clearly, a walk that gathers a certain reward in a restricted version of an instance will gather at least that reward in the original instance. We attempt to solve ORIENT-TW by constructing a set of restricted versions that are easier to work with. Typically, the construction is such that the reward of an optimal solution in at least one of the restricted versions is a significant fraction of the reward of an optimal solution in the original instance. Hence, an approximation to the optimal solution in the 'best' restricted version leads us to an approximation for the original instance.

This idea leads us to the next proposition, the proof of which is straightforward, and hence omitted.

Proposition 5.2 *Let A be an instance of* ORIENT-TW *on a graph $G(V, E)$. If $B_1, B_2, \ldots B_\beta$ are restricted versions of A, and for all vertices $v \in V$, $I_A(v) = \bigcup_{1 \leq i \leq \beta} I_{B_i}(v)$, there is some B_j such that* $\text{OPT}(B_j) \geq \frac{\text{OPT}(A)}{\beta}$.

The restricted versions we construct will usually be modular instances of ORIENT-TW. Therefore, the general algorithm for ORIENT-TW is:

1. Construct a set of β restricted versions of the given instance; each restricted version is a modular instance.
2. Pick the best restricted version (enumerate over all choices), find an appropriate partition, and use an α-approximation for ORIENTEERING together with dynamic programming to solve that instance.

It follows from the previous discussion that this gives a $(\alpha \times \beta)$-approximation for ORIENT-TW. We next describe how to solve modular instances of ORIENT-TW.

5.1.1 A dynamic program for modular instances

Recall that a modular instance is an instance of ORIENT-TW on a graph $G(V, E)$ in which the vertex set V can be partitioned into $V_1, V_2, \ldots V_m$, such that an optimal solution visits vertices of V_i after time R_i and before D_i. For any vertex $v \in V_i$, $R(v) \leq R_i$ and $D(v) \geq D_i$. Further, vertices of V_i are visited before vertices of V_j, for all $j > i$.

To solve a modular instance, for each V_i we could 'guess' the first and last vertex visited by an optimal solution, and guess the times at which this solution visits the first and last vertex. If α is the approximation ratio of an algorithm for orienteering, we find a path in each V_i that collects an α-fraction of the optimal reward, and combine these solutions.

More formally, one could use the following dynamic program: For any $u, v \in V_i$, consider the graph induced by V_i, and let $\text{OPT}(u, v, t)$ denote the optimal reward collected by any walk from u to v of length at most t (ignoring time-windows). Now, define $\Pi_i(v, T)$ for $v \in V_i, R_i \leq T \leq D_i$ as the optimal reward collected by any walk in G that begins at s at time 0, and ends at v by time T. Given $\text{OPT}(u, v, t)$, the following recurrence allows us to easily compute $\Pi_i(v, T)$:

$$\Pi_i(v, T) = \max_{u \in V_i, w \in V_{i-1}, t \leq T - R_i} \text{OPT}(u, v, t) + \Pi_{i-1}(w, T - t - d(w, u)).$$

Of course, we cannot exactly compute $\text{OPT}(u, v, t)$; instead, we use an α-approximation algorithm for orienteering to compute an approximation to $\text{OPT}(u, v, t)$ for all $u, v \in V_i$, $t \leq D_i - R_i$. This gives an α-approximation to $\Pi_i(v, T)$ using the recurrence above.

Unfortunately, the running time of this algorithm depends polynomially on T; this leads to a pseudo-polynomial algorithm. To obtain a polynomial-time algorithm, we use a standard technique of dynamic programming based on reward instead of time (see [8, 12]). Using standard scaling tricks for maximization problems, one can reduce the problem with arbitrary rewards on the vertices to the problem where the reward on each vertex is 1; the resulting loss in approximation can be made $(1 + o(1))$. Thus, the maximum reward is n.

To construct a dynamic program based on reward instead of time, we wish to find, for each $u, v \in V_i$ and each $k_i \in [0, |V_i|]$, an optimal (shortest) walk from u to v that collects reward at least k_i. However, we cannot do this exactly. One could try to find an *approximately* shortest $u - v$ walk collecting reward k_i, but this does not lead to a good solution overall: taking slightly too much time early on can have bad consequences for later groups V_j. Instead, we "guess" the length (using binary search over the maximum walk length in G) of an optimal walk that obtains reward k_i, and for each guess use the α-approximate ORIENTEERING algorithm. This guarantees that if there is a u-v walk of length B that collects reward k_i, then we find a u-v walk of length at most B that collects reward at least k_i/α. Finally, to obtain the desired approximation for the entire instance, we stitch together the solutions from each V_i using a dynamic program very similar to the one described above based on time.

5.2 The Algorithms

Using the framework described above, we now develop algorithms which achieve approximation ratios depending on the lengths of the time-windows. We first consider instances where all time-windows have integral end-points, and then instances for which the ratio $L = \frac{L_{\max}}{L_{\min}}$ is bounded. Finally, we combine these ideas to obtain an $O(\alpha \max\{\log \text{OPT}, \log L\})$-approximation for all instances of ORIENT-TW.

5.2.1 An $O(\alpha \log L_{\max})$-approximation

We now focus on instances of ORIENT-TW in which, for all vertices v, $R(v)$ and $D(v)$ are integers. Our algorithm is based on the following simple lemma:

Lemma 5.3 *Any interval of length $M > 1$ with integral endpoints can be partitioned into at most $2\lfloor \log M \rfloor$ disjoint sub-intervals, such that the length of any sub-interval is a power of 2, and any sub-interval of length 2^i begins at a multiple of 2^i. Further, there are at most 2 sub-intervals of each length.*

Proof: Use induction on the length of the interval. The lemma is clearly true for intervals of length 2 or 3. Otherwise, use at most 2 sub-intervals of length 1 at the beginning and end of the given interval, so that the *residual* interval (after the sub-intervals of size 1 are deleted) begins and ends at an even integer. To cover the residual interval, divide all integers in the (residual) problem by 2, and apply the induction hypothesis; we use at most $2+(2\lfloor \log M/2 \rfloor) \leq 2\lfloor \log M \rfloor$ sub-intervals in total. It is easy to see that we use at most 2 sub-intervals of each length; intervals of length 2^i are used at the $(i+1)$th level of recursion. □

For ease of notation, we let ℓ denote $\lfloor \log L_{\max} \rfloor$ for the rest of this sub-section, and assume for ease of exposition that $L_{\max} \geq 2$. Given an instance of ORIENT-TW, for each vertex v with interval $I(v)$, we use Lemma 5.3 to partition $I(v)$ into at most 2ℓ sub-intervals. We label the sub-intervals of $I(v)$ as follows: For each $1 \leq i \leq \ell$, the first sub-interval of length 2^i is labeled $I_i^1(v)$ and the second sub-interval $I_i^2(v)$. (Note that there may be no sub-intervals of length 2^i.)

We now construct a set of at most 2ℓ restricted versions of the given instance. We call these restricted versions $B_1^1, B_2^1, \ldots B_\ell^1$ and $B_1^2, B_2^2, \ldots B_\ell^2$, such that the interval for vertex v in B_i^b is $I_i^b(v)$. If $I_i^b(v)$ was not an interval used in the partition of $I(v)$, v is not present in the restricted version. (Equivalently, it has reward 0 or an empty time-window.)

Consider an arbitrary restricted instance B_i^b. All vertices in this instance of ORIENT-TW have intervals of length 2^i, and all time-windows begin at an integer that is a multiple of 2^i. Hence, any 2 vertices either have time-windows that are identical, or entirely disjoint. This means that B_i^b is a modular instance, so we can break it into sub-problems, and use an α-approximation to orienteering in the sub-problems to obtain an α-approximation for the restricted instance.

By Proposition 5.2, one of the restricted versions has an optimal solution that collects reward at least $\frac{\text{OPT}}{2\ell}$. Using an α-approximation for this restricted version gives us an $\alpha \times 2\ell = O(\alpha \log L_{\max})$-approximation for ORIENT-TW when all interval endpoints are integers.

5.2.2 An $O(\alpha \log \text{OPT})$-approximation when $L \leq 2$

For an instance of ORIENT-TW when $L = \frac{L_{\max}}{L_{\min}} \leq 2$, we begin by scaling all release times, deadlines, and edge lengths so that $L_{\min} = 1$ (and so $L_{\max} \leq 2$). Note that even if all release times and deadlines were integral prior to scaling, they may not be integral in the scaled version; after scaling, all interval lengths are in $[1, 2]$.

For each vertex v, we partition $I(v) = [R(v), D(v)]$ into 3 sub-intervals: $I_1(v) = [R(v), a]$, $I_2(v) = [a, b]$, and $I_3(v) = [b, D(v)]$, where $a = \lfloor R(v)+1 \rfloor$ (that is, the next integer strictly greater than the release time) and $b = \lceil D(v)-1 \rceil$ (the greatest integer strictly less than the deadline). The figure below illustrates the partitioning of intervals. Note that $I_2(v)$ may be a point, and in this case, we ignore such a sub-interval.

We now construct 3 restricted versions of the given instance — B_1, B_2, and B_3 — such that the interval for any vertex v in B_i is simply $I_i(v)$. By Proposition 5.2, one of these has an optimal solution that collects at least a third of the reward collected by an optimal solution to the original instance. Suppose this is B_2. All time-windows have length exactly 1, and start and end-points are integers. Therefore, B_2 is a modular instance, and we can get an α-approximation to the optimal solution in B_2; this gives a 3α-approximation to the original instance.

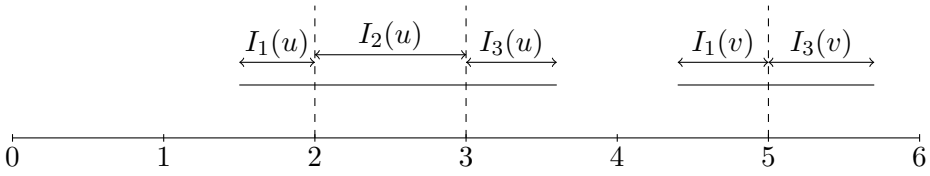

Figure 4: We illustrate the partitioning of 2 intervals into sub-intervals. Note that on the right, $I_2(v)$ is empty.

Dealing with B_1 and B_3 is not nearly as easy; they are not quite modular. Every interval in B_1 has length at most 1, and ends at an integer; for B_3, intervals have length at most 1 and start at an integer. We illustrate how to approximate a solution for B_3 within a factor of $O(\alpha \log \text{OPT})$; the algorithm for B_1 is identical except that release times and deadlines are to be interchanged.

For B_3, we can partition the vertex set into $V_1, V_2, \ldots V_m$, such that all vertices in V_i have the same (integral) release time, and any vertex in V_i is visited before any vertex in V_j for $j > i$. Figure 5 shows such a partition. The deadlines for vertices in V_i may be all distinct. However, we can solve an instance of ORIENT-DEADLINE in each V_i separately, and paste the solutions together using dynamic programming. The solution we obtain will collect at least $\Omega(1/\alpha \log \text{OPT})$ of the reward of an optimal solution for B_3, since there is a $O(\log \text{OPT})$-approximation for ORIENT-DEADLINE ([6]). Therefore, this gives us a $3 \times O(\alpha \log \text{OPT}) = O(\alpha \log \text{OPT})$-approximation to the original instance.

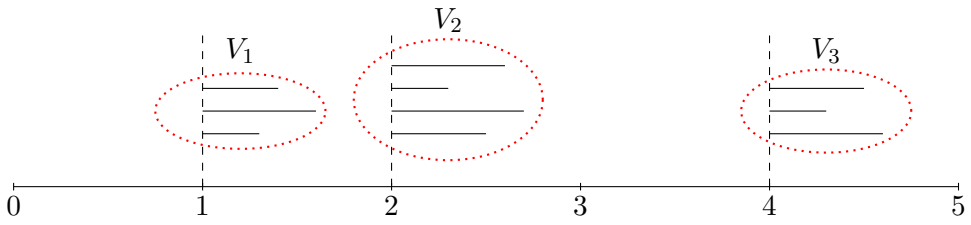

Figure 5: In B_3, all time-windows start at an integer and have length at most 1. Each set of vertices whose windows have a common beginning corresponds to a sub-problem that is an instance of orienteering with deadlines.

Similarly, we can obtain an $O(\alpha \log \text{OPT})$-approximation for B_1 using the $O(\log \text{OPT})$-approximation algorithm for orienteering with release times. Therefore, when $L \leq 2$, we have an $O(\alpha \log \text{OPT})$-approximation for ORIENT-TW.

5.2.3 Putting the pieces together

An arbitrary instance of ORIENT-TW may have $L > 2$, and interval end-points may not be integers. However, we can combine the algorithms from the two preceding sections to deal with such instances. We begin by scaling release times, deadlines, and edge lengths such that the shortest interval has length 1; the longest interval now has length $L = \frac{L_{\max}}{L_{\min}}$, where $L_{\max}$ and $L_{\min}$ are the lengths of the longest and shortest intervals in the original instance.

We now construct 3 restricted versions of the scaled instance: B_1, B_2, and B_3. For any vertex v with interval $[R(v), D(v)]$ in the scaled instance, we construct 3 sub-intervals. If the interval for v has length less than 2, we set $I_1(v) = [R(v), D(v)]$, and $I_2(v) = I_3(v) = \emptyset$. Otherwise, $I_1(v) = [R(v), a]$, $I_2(v) = [a, b]$, and $I_3(v) = [b, D(v)]$, where $a = \lceil R(v) + 1 \rceil$ and $b = \lfloor D(v) - 1 \rfloor$. As before, the interval for v in the instance B_i

is $I_i(v)$.

One of the restricted versions collects at least a third of the reward of the original instance. Suppose this is B_1 or B_3. All intervals in B_1 and B_3 have length between 1 and 2 by our construction. Therefore, we can use the $O(\alpha \log \text{OPT})$-approximation algorithm from section 5.2.2 to collect at least $\Omega(1/\alpha \log \text{OPT})$ of the reward of an optimal solution to the original instance. It now remains only to consider the case that B_2 collects more than a third of the reward. In B_2, the end-points of all time-windows are integral, and the longest interval has length less than L. We can now use the algorithm of section 5.2.1 to obtain an $O(\alpha \log L)$-approximation.

Therefore, our combined algorithm is an $O(\alpha \max\{\log \text{OPT}, \log L\})$-approximation for ORIENT-TW, proving Theorem 1.4.

5.3 Towards a better approximation, and arbitrary endpoints

In the previous sub-section, we obtained an approximation ratio of $O(\alpha \max\{\log \text{OPT}, \log L\})$; we would like to improve this ratio to $O(\alpha \log \text{OPT})$. Unfortunately, it does not seem easy to do this directly. A natural question, then, would be to obtain a ratio of $O(\alpha \log L)$; this is equivalent to an $O(\alpha)$ approximation for the case when $L \leq 2$. However, this is is no easier than finding an $O(\alpha \log \text{OPT})$-approximation for arbitrary instances of ORIENT-TW, as we show in the next proposition.

Proposition 5.4 *An $O(\alpha)$ approximation algorithm for* ORIENT-TW *with $L \leq 2$ implies an $O(\alpha \log \text{OPT})$-approximation for general instances of* ORIENT-TW.

Proof: We show that an $O(\alpha)$ approximation when $L \leq 2$ implies an $O(\alpha)$ approximation for ORIENT-DEADLINE. It follows from an algorithm of [6] that we can then obtain an $O(\alpha \log \text{OPT})$-approximation for ORIENT-TW.

Given an arbitrary instance of ORIENT-DEADLINE on graph $G(V, E)$, we add a new start vertex s' to G. Connect s' to s with an edge of length $D_{\max} = \max_v D(v)$. The release time of every vertex is 0, but all deadlines are increased by $D_{\max}$. Observe that all vertices have time-windows of length between $D_{\max}$ and $2D_{\max}$, so $L \leq 2$. It is easy to see that given any walk beginning at s in the original instance, we can find an equivalent walk beginning at s' in the modified instance that visits a vertex in its time-window iff the original walk visited a vertex before its deadline in the given instance, and vice versa. Therefore, an $O(\alpha)$ approximation for the modified instance of ORIENT-TW gives an $O(\alpha)$ approximation for the original instance of ORIENT-DEADLINE. □

We *can*, however, obtain an $O(\alpha)$-approximation for ORIENT-TW when $L \leq 2$ if we remove the restriction that the walk must start and end at s and t, the specified endpoints. The algorithm of [15] for the case of $L = 1$ can be adapted relatively easily to give an $O(\alpha)$ approximation for $L \leq 2$. For completeness, we sketch the algorithm here.

We construct 5 restricted versions $B_1, \ldots B_5$, of a given instance A. For every vertex v, we create at most 5 sub-intervals of $I(v)$ by breaking it at every multiple of 0.5. (For instance $[3.7, 5.6]$ would be broken up into $[3.7, 4], [4, 4.5], [4.5, 5], [5, 5.5], [5.5, 5.6]$. Note that some intervals may have fewer than 5 sub-intervals.) The interval for v in $B_1(v)$ is the first sub-interval, and the interval in $B_5(v)$ is the last sub-interval, regardless of whether $I(v)$ has 5 sub-intervals. B_2, B_3, and B_4 each use one of any remaining sub-intervals.

B_2, B_3, and B_4 are modular instances, so if one of them is the best restricted version of A, we can use an α-approximation for orienteering to get reward at least $\frac{\text{OPT}(A)}{5\alpha}$. Exactly as in subsection 5.2.2, B_1 and B_5 are not quite modular instances; in B_1, all deadlines are half-integral but release times are arbitrary, and in B_5, all release times are half-integral, but deadlines are arbitrary.

Suppose that B_1 is the best restricted version. The key insight is that if the optimal walk in B_1 collects a certain reward starting at s at time 0, there is a walk in B_2 starting at s at time 0.5 that collects *the same*

reward. (This is the substance of Theorem 1 of [15].) Therefore, if B_1 is the best restricted version, we find an α-approximation to the best walk in B_2 starting at s at time 0.5; we are guaranteed that this walk collects reward at least $\frac{\text{OPT}(A)}{5\alpha}$. Note that this walk may not reach the destination vertex t by the time limit, since we start 0.5 time units late. Similarly, if B_5 is the best restricted version, we can find a walk in B_4 that collects reward $\frac{\text{OPT}(A)}{5\alpha}$ while beginning at s at time -0.5. (To avoid negative times, we can begin the walk at s' at time 0, where s' is the first vertex visited by the original walk after time 0.) This walk is guaranteed to reach t by the time limit, but does not necessarily begin at s.

Therefore, this algorithm is an $O(\alpha)$-approximation when $L \leq 2$, or an $O(\alpha \log L)$-approximation for general instances of ORIENT-TW. We note that one *cannot* use this with Proposition 5.4 to get an $O(\alpha \log \text{OPT})$-approximation for the variant of ORIENT-TW where start/end vertices are not specified: The dynamic program for modular instances crucially uses the fact that we can specify both endpoints for the sub-problems.

6 Conclusions

The algorithms presented in this paper can be combined with previously known techniques [8, 6, 11] to obtain the following results:

- A $(4 + \varepsilon)$ approximation for the following problem in undirected graphs: Find a tree rooted at a given vertex s of total length at most B that maximizes the number of vertices in the tree. This improves the 6-approximation in [8, 6].
- A $(3+\varepsilon)$ approximation for ORIENT-TW when there are a fixed number of time windows; this improves a ratio of 4 from [11].

Finally, we list a few open problems:

1. Is there a 2-approximation for ORIENTEERING in undirected graphs? In addition to matching the known ratios for k-MST and k-TSP [18], this may lead to a more efficient algorithm than the one presented in this paper.
2. Is there an $O(1)$ approximation for ORIENTEERING in directed graphs?
3. Is there a poly-logarithmic, or even an $O(1)$, approximation for the k-stroll problem in directed graphs? Currently there is only a bi-criteria algorithm.
4. Can one prove Conjecture 1.3 to obtain an $O(\log \text{OPT})$-approximation for ORIENT-TW, at least for undirected graphs?
5. The current algorithms for ORIENT-TW use the orienteering algorithm in a black-box fashion. For directed graphs the current ratio for ORIENTEERING is $O(\log^2 \text{OPT})$ and hence the ratio for ORIENT-TW is worse by additional logarithmic factors. Can one avoid using ORIENTEERING as a black-box? We note that the quasi-polynomial time algorithm of [12] has the same approximation ratio of $O(\log \text{OPT})$ for both the basic orienteering and time-window problems in directed graphs.

Acknowledgments: CC thanks Rajat Bhattacharjee for an earlier collaboration on the undirected Orienteering problem, and also thanks Naveen Garg and Amit Kumar for useful discussions. We thank Viswanath Nagarajan and R. Ravi for sending us a copy of [24]. We thank Pratik Worah for several discussions of algorithms and hardness results for orienteering with time-windows and other variants. We thank the anonymous referees for useful comments, particularly on clarifying Proposition 3.4 and the organization of Section 3.

References

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

[2] E.M. Arkin, J.S.B. Mitchell, and G. Narasimhan. Resource-constrained geometric network optimization. In *Symposium on Computational Geometry*, pages 307–316, 1998.

[3] S. Arora and G. Karakostas. A $2 + \varepsilon$ approximation algorithm for the k-MST problem. *Mathematical Programming A*, 107(3):491–504, 2006. Preliminary version in *Proc. of ACM-SIAM SODA*, 754–759, 2000.

[4] B. Awerbuch, Y. Azar, A. Blum, and S. Vempala. Improved approximation guarantees for minimum-weight k-trees and prize-collecting salesmen. *SIAM J. Computing*, 28(1):254–262, 1998. Preliminary Version in *Proc. of ACM STOC*, 277–283, 1995.

[5] E. Balas. The prize collecting traveling salesman problem. *Networks*, 19(6):621–636, 1989.

[6] N. Bansal, A. Blum., S. Chawla, and A. Meyerson. Approximation algorithms for deadline-TSP and vehicle routing with time-windows. In *Proceedings of the 36th Annual ACM Symposium on Theory of Computing*, pages 166–174. ACM New York, NY, USA, 2004.

[7] R. Bar-Yehuda, G. Even, and S. Shahar. On approximating a geometric prize-collecting traveling salesman problem with time windows. *Journal of Algorithms*, 55(1):76–92, 2005. Preliminary version in *Proc. of ESA*, 55–66, 2003.

[8] A. Blum, S. Chawla, D. Karger, T. Lane, A. Meyerson, and M. Minkoff. Approximation algorithms for orienteering and discounted-reward TSP. *SIAM Journal on Computing*, 37(2):653–670, 2007. Preliminary version in *Proc. of IEEE FOCS*, 46–55, 2003.

[9] A. Blum, R. Ravi, and S. Vempala. A Constant-Factor Approximation Algorithm for the k-MST Problem. *Journal of Computer and System Sciences*, 58(1):101–108, 1999. Preliminary version in *Proc. of ACM STOC*, 1996.

[10] K. Chaudhuri, B. Godfrey, S. Rao, and K. Talwar. Paths, trees, and minimum latency tours. In *44th Annual Symposium on Foundations of Computer Science*, pages 36–45. IEEE Computer Society, 2003.

[11] C. Chekuri and A. Kumar. Maximum coverage problem with group budget constraints and applications. *Proceedings of APPROX-RANDOM*, pages 72–83, 2004.

[12] C. Chekuri and M. Pál. A recursive greedy algorithm for walks in directed graphs. In *Proceedings of the 46th Annual Symposium on Foundations of Computer Science*, pages 245–253. IEEE Computer Society, 2005.

[13] C. Chekuri and M. Pál. An $O(\log n)$ Approximation for the Asymmetric Traveling Salesman Path Problem. *Theory of Computing*, 3:197–209, 2007. Preliminary version in *Proc. of APPROX*, 95–103, 2005.

[14] K. Chen and S. Har-Peled. The orienteering problem in the plane revisited. *SIAM J. on Computing*, 38(1):385–397, 2008. Preliminary version in *Proc. of ACM SoCG*, 247–254, 2006.

[15] G.N. Frederickson and B. Wittman. Approximation algorithms for the traveling repairman and speeding deliveryman problems with unit-time windows. *Proceedings of APPROX-RANDOM,*, LNCS 4627:119–133, 2007.

[16] A.M. Frieze, G. Galbiati, and F. Maffioli. On the worst-case performance of some algorithms for the asymmetric traveling salesman problem. *Networks*, 12(1):23–39, 1982.

[17] N. Garg. A 3-approximation for the minimum tree spanning k vertices. In *Proceedings of the 37th Annual Symposium on Foundations of Computer Science*, pages 302–309. IEEE Computer Society, 1996.

[18] N. Garg. Saving an epsilon: a 2-approximation for the k-MST problem in graphs. In *Proceedings of the 37th Annual ACM Symposium on Theory of computing*, pages 396–402. ACM, 2005.

[19] M. X. Goemans and D. P. Williamson. A general approximation technique for constrained forest problems. *SIAM Journal on Computing*, 24:296–317, 1995.

[20] B.L. Golden, L. Levy, and R. Vohra. The orienteering problem. *Naval Research Logistics*, 34(3):307–318, 1987.

[21] G. Gutin and A.P. Punnen, editors. *The traveling salesman problem and its variations*. Springer, Berlin, 2002.

[22] J. Kleinberg and D.P. Williamson. Unpublished Note, 1998.

[23] E.L. Lawler, A.H.G. Rinnooy Kan, J.K. Lenstra, and D.B. Shmoys, editors. *The Traveling salesman problem: a guided tour of combinatorial optimization*. John Wiley & Sons Inc, 1985.

[24] V. Nagarajan and R. Ravi. Poly-logarithmic approximation algorithms for directed vehicle routing problems. *Proceedings of APPROX-RANDOM,*, LNCS 4627:257–270, 2007.

[25] P. Toth and D. Vigo, editors. *The vehicle routing problem*. SIAM Monographs on Discrete Mathematics and Applications. Society for Industrial Mathematics, Philadelphia PA, 2001.

[26] J.N. Tsitsiklis. Special cases of traveling salesman and repairman problems with time windows. *Networks*, 22(3):263–282, 1992.