Shortest Beer Path Queries in Digraphs with Bounded Treewidth

Joachim Gudmundsson
The University of Sydney, Australia

Yuan Sha
The University of Sydney, Australia

Abstract

A beer digraph G is a real-valued weighted directed graph where some of the vertices have beer stores. A beer path from a vertex u to a vertex v in G is a path in G from u to v that visits at least one beer store.

In this paper we consider the online shortest beer path query in beer digraphs with bounded treewidth t. Assume that a tree decomposition of treewidth t on a beer digraph with n vertices is given. We show that after $O(t^3 n)$ time preprocessing on the beer digraph, (i) a beer distance query can be answered in $O(t^3\alpha(n))$ time, where $\alpha(n)$ is the inverse Ackermann function, and (ii) a shortest beer path can be reported in $O(t^3\alpha(n)L)$ time, where L is the number of edges on the path. In the process we show an improved $O(t^3\alpha(n)L)$ time shortest path query algorithm, compared with the currently best $O(t^4\alpha(n)L)$ time algorithm [Chaudhuri & Zaroliagis, 2000].

We also consider queries in a dynamic setting where the weight of an edge in G can change over time. We show two data structures. Assume t is constant and let β be any constant in $(0, 1)$. The first data structure uses $O(n)$ preprocessing time, answers a beer distance query in $O(\alpha(n))$ time and reports a shortest beer path in $O(\alpha(n)L)$ time. It can be updated in $O(n^\beta)$ time after an edge weight change. The second data structure has $O(n)$ preprocessing time, answers a beer distance query in $O(\log n)$ time, reports a shortest beer path in $O(\log n + L)$ time, and can be updated in $O(\log n)$ time after an edge weight change.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms

Keywords and phrases Graph algorithms, Shortest Path, Data structures, Bounded treewidth

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2023.35

1 Introduction

A *beer digraph* is a real-valued weighted directed graph $G = (V(G), E(G))$ in which some of the vertices have beer stores. For any two vertices $u, v \in V(G)$, a *shortest beer path* is a shortest path from u to v that visits at least one beer store. The *beer distance* from u to v is the weight of a shortest beer path from u to v. In this paper, we consider the problem of shortest beer path and beer distance queries for beer digraphs in both static and dynamic settings. In the dynamic setting, the weight of an edge in the graph can change.

The shortest path and distance queries are fundamental graph problems. There are numerous works on the subject in the literature. Thorup and Zwick [22] used the term “distance oracle” to refer to a compact data structure that can efficiently answer distance query between any two vertices. Ideally one would like a distance oracle with linear preprocessing time and space, and constant query time. However, it is well known that there are graphs for which no distance oracle with $o(n^2)$ bits of space and $O(1)$ query time exists. Because of this, researchers have focused their attention on restricted classes of graphs.

There has been extensive research on the class of planar graphs [5, 8, 9, 10, 11, 13, 18, 19]. We briefly highlight some of the most recent results for planar graphs. In [5], Charalampopoulos et al. showed a distance oracle with space $O(n^{1+\epsilon})$ and polylogarithmic query-time for any constant $\epsilon > 0$. Long and Pettie [18] showed (i) an oracle with space

$O(n \log^{2+o(1)} n)$ and query-time $O(n^{\epsilon})$ for any constant $\epsilon > 0$, and (ii) an oracle with space $O(n^{1+o(1)})$ and query-time $n^{o(1)}$. For a comparison of the existing distance oracles for planar graphs, refer to Table 1 in [18].

For the class of graphs with bounded treewidth, the best shortest path and distance query structure is attributed to Chaudhuri and Zaroliagis [6]. Let t be the treewidth of the input graph G and assume that a tree decomposition of G with treewidth t is given. They showed that after $O(t^3 n)$ preprocessing, distance queries can be answered in time $O(t^3\alpha(n))$, and shortest path queries can be answered in time $O(t^4\alpha(n)L)$, where L is the number of edges on the path and $\alpha(n)$ is the inverse Ackermann function.

Not surprisingly, answering shortest path or distance queries efficiently in a dynamic setting where the graph undergoes changes is hard. Roditty and Zwick [21] showed that the innocent looking decremental (only edge deletions) or incremental (only edge insertions) *single-source* shortest path (SSSP) problem is, in a strong sense, as hard as APSP (all-pairs shortest path). It is conjectured that APSP has no truly subcubic time ($O(n^{3-\epsilon})$) solution [23].

The shortest beer path problem is a generalization of the shortest path problem. Bacic et al. [2] considered the shortest beer path and beer distance queries for undirected outerplanar graphs with non-negative edge weights. They showed that after $O(n)$ time preprocessing a beer distance query can be answered in $O(\alpha(n))$ time, and a shortest beer path query can be answered in $O(L)$ time. Hanaka et al. [15] considered the shortest beer path and beer distance queries for graphs with bounded triconnected component size (which they extends to graphs with bounded treewidth). In this paper we study shortest beer path and beer distance queries for digraphs with bounded treewidth.

We first show an improved $O(t^3\alpha(n)L)$ time shortest path query algorithm, compared with the $O(t^4\alpha(n)L)$ time algorithm in [6]. Pettie [20] proved that for the *online MST (minimum spanning tree) verification* problem, answering m queries requires $\Omega(m\alpha(m,n))$ time. Bacic et al. [2] reduced the online MST verification problem to the beer distance query problem on trees, which implies that answering m beer distance queries on beer trees requires $\Omega(m\alpha(m,n))$ time. Thus our result of beer distance query in Table 1 is optimal for graphs with constant treewidth. Note that our shortest beer path and beer distance query structure answers shortest beer path queries in the same asymptotic time as answering shortest path queries, and answers beer distance queries in the same asymptotic time as answering distance queries (see Table 1).

We also consider the shortest beer path and beer distance queries in the dynamic setting where edge weights can change over time. Edge deletion (setting weight to infinity) and edge re-insertion are special cases of edge weight change. We do not consider general edge insertion, since our approach is based on tree decompositions of graphs while general edge insertion can greatly change the treewidths and break the tree decompositions. For this dynamic setting we give two dynamic shortest beer path query structures. See Table 2 for a comparison of the structures.

2 Preliminaries

Let $G = (V(G), E(G))$ be a weighted beer digraph with n vertices where some of the vertices have a beer store. The edge weights are specified by a weight function $wt : E(G) \longrightarrow \mathbb{R}$. Let $wt(u, v)$ be the weight of edge $\langle u, v\rangle$. The weight of a path in G is the sum of the weights of the edges on the path. For any two vertices $u, v \in E(G)$, a shortest path in G from u to v is a path from u to v with the minimum weight and is denoted as $\pi_G(u, v)$. The distance from

Table 1 Comparison of shortest path query structures. In the table t is the treewidth, L is the number of edges on the path, and $\alpha(n)$ is the inverse Ackermann function. Assume a tree decomposition of treewidth t is given for a bounded treewidth graph. Entries in boldface are new.

Source	Graph	Preproc.	Distance	Shortest path	Beer dist.	Beer path
[6]	directed, bounded treewidth	$O(t^3 n)$	$O(t^3\alpha(n))$	$O(t^4\alpha(n)L)$	–	–
[2]	undirected, outer-planar	$O(n)$	$O(\alpha(n))$	$O(L)$	$O(\alpha(n))$	$O(L)$
Theorem 9	directed, bounded treewidth	$O(t^3 n)$	$O(t^3\alpha(n))$	$\mathbf{O(t^3\alpha(n)L)}$	$\mathbf{O(t^3\alpha(n))}$	$\mathbf{O(t^3\alpha(n)L)}$

Table 2 Comparison of dynamic shortest path query structures on directed graphs with constant treewidth. β is any constant in $(0,1)$, and L is the number of edges on the path. Entries in boldface are new.

Source	Preproc.	Distance	Shortest path	Beer dist.	Beer path	Edge weight update
[6]	$O(n)$	$O(\alpha(n))$	$O(L\alpha(n))$	–	–	$O(n^\beta)$
Theorem 13	$O(n)$	$O(\alpha(n))$	$O(L\alpha(n))$	$\mathbf{O(\alpha(n))}$	$\mathbf{O(L\alpha(n))}$	$\mathbf{O(n^\beta)}$
Theorem 23	$\mathbf{O(n)}$	$\mathbf{O(\log n)}$	$\mathbf{O(\log n + L)}$	$\mathbf{O(\log n)}$	$\mathbf{O(\log n + L)}$	$\mathbf{O(\log n)}$

u to v in G, denoted as $\delta_G(u,v)$, is the weight of $\pi_G(u,v)$. A shortest *beer path* in G from u to v is denoted as $\pi_{BG}(u,v)$ and the *beer distance* from u to v in G, denoted as $\delta_{BG}(u,v)$, is the weight of $\pi_{BG}(u,v)$.

Let V_1, V_2 and U be disjoint subsets of $V(G)$. U is a separator for V_1 and V_2 if every path in G from a vertex in V_1 (V_2) to a vertex in V_2 (V_1) goes through a vertex in U.

Let X be a subset of $V(G)$ and let $G[X]$ be the subgraph of G induced by X.

A *tree decomposition* of a graph G (directed or undirected) is a pair (X,T) in which $T = (I = V(T), E(T))$ is a tree and $X = \{X_i | i \in I\}$ is a family of subsets of $V(G)$ such that:

1. $\bigcup_{i\in I} X_i = V(G)$;
2. for each edge $e = (u,v) \in E(G)$ there exists a node $i \in I$ such that both u and v belong to X_i; and
3. for all $v \in V$, the set of nodes $\{i \in I | v \in X_i\}$ forms a connected subtree of T.

The set X_i is called the *bag of* node i. The *treewidth* of a tree decomposition is $\max_{i\in I} |X_i| - 1$. The treewidth of G is the minimum treewidth over all possible tree decompositions of G.

▶ **Theorem 1** ([4]). *For every constant $t \in N$, there exists an $O(n)$ time algorithm which tests whether a given n-vertex graph G has treewidth at most t and if so, outputs a tree-decomposition (X,T) of G with treewidth at most t, where $|V(T)| = n - t$. In $O(n)$ time, the tree decomposition (X,T) can be converted into a binary tree decomposition of $2(n-t)$ nodes and treewidth t.*

For a tree decomposition (X,T) of G, every edge $e = (i,j) \in E(T)$ corresponds to a separator of G. Let T_1 and T_2 be the subtrees of T obtained by removing e from T, then $X_i \cap X_j$ separates $\bigcup_{m\in V(T_1)} X_m - (X_i \cap X_j)$ from $\bigcup_{m\in V(T_2)} X_m - (X_i \cap X_j)$.

3 Shortest path and distance queries

The most efficient algorithms for shortest path and distance queries on digraphs with bounded treewidth to date are given by Chaudhuri and Zaroliagis [6] (see Table 1). We give some preliminaries in Section 3.1, before giving an outline of the algorithms in [6] in Section 3.2. Finally we show how to improve the shortest path query algorithm in Section 3.3.

3.1 Preliminaries

Call (a, b, c) a *distance tuple* if a, b are symbols and $c \in \mathbb{R}$. Define a composition operator $\circ$ on two distance tuples as $(a_1, b_1, c_1) \circ (a_2, b_2, c_2) = (a_1, b_2, c_1 + c_2)$ if $b_1 = a_2$, otherwise it is undefined. For a set of distance tuples, say M, define $minmap(M)$ to be the operation on M such that among all distance tuples in M with the same first and second components, only the distance tuple with the smallest third component is retained.

Let A and B be two sets of distance tuples. Define a composition operator $\circ$ on A and B as $A \circ B = minmap\{x \circ y | x \in A,\ y \in B\}$. For a node $i \in V(T)$ where (X, T) is a tree decomposition of G, define $\gamma(i) = \{(u, v, \delta_G(u, v)) | u, v \in X_i\}$ to be the set of distance tuples for pairs of vertices in X_i (u and v can be the same vertex).

▶ **Definition 2** ([6]). *Let H be a digraph, with V_1, V_2 and U be a partition of $V(H)$ such that U is a separator for V_1 and V_2. Let H_1 and H_2 be subgraphs of H such that $V(H_1) = V_1 \cup U$ and $V(H_2) = V_2 \cup U$. We say that H_1' is a graph obtained by absorbing H_2 into H_1, if H_1' is obtained from H_1 by adding edges $\langle u, v\rangle$, with weight $\delta_{H_2}(u, v)$ or $\delta_H(u, v)$, for each pair $u, v \in U$. In case multiple edges are created, retain the one with minimum weight.*

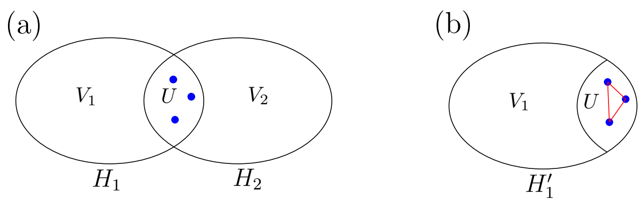

Figure 1 (a) Illustrating Definition 2. (b) H_1' where the red edges have weights $\delta_{H_2}(\cdot,\cdot)$ or $\delta_H(\cdot,\cdot)$.

Absorption preserves distances in a digraph. Let H, H_1, H_2 and H_1' be as in Definition 2, then for all $x, y \in V(H_1')$, $\delta_{H_1'}(x, y) = \delta_H(x, y)$.

3.2 Shortest path and distance query algorithms

Let G be a weighted digraph with bounded treewidth t. Here we will briefly present the key ideas of the algorithms by Chandhuri and Zaroliagis [6]. From Theorem 1, one can obtain a tree decomposition (X, T) of G with constant treewidth in $O(n)$ time.[1] One can obtain $\gamma(i)$ for all nodes $i \in V(T)$ in $O(t^3 n)$ time by the following lemma. The algorithm uses an *absorption and expansion processes*. $Int_G(u, v)$ is an intermediate vertex (neither u nor v) on a shortest path from u to v.

[1] The hidden constant of the $O(n)$ time in Theorem 1 is $2^{O(t^3)}$. Recently, Korhonen [17] gave a 2-approximation algorithm that runs in $O(2^{O(t)} n)$ time, which suffices for our application. Interested readers are also referred to [12] for an $O(t^7 n \log n)$ time, $O(t)$-approximation algorithm.

▶ **Lemma 3.** *Let G be an n-vertex weighted digraph and let (X, T) be the tree decomposition of G, of treewidth t. For each pair u, v such that $u, v \in X_i$ for some $i \in V(T)$, let $Dist(u, v) = \delta_G(u, v)$ and $Int_G(u, v) = x$, where x is some intermediate vertex (neither u nor v) on a shortest path from u to v. If $wt(u, v) = \delta_G(u, v)$, then $Int_G(u, v) = null$. Then in $O(t^3n)$ time, we can either find a negative cycle in G, or compute the values $Dist(u, v)$ and $Int_G(u, v)$ for each such pair u, v.*

After using the algorithm in Lemma 3, all $\gamma(i)$ for $i \in V(T)$ have been computed. One can define a semigroup $(\Gamma, \circ)$ where Γ is the set of sets of distance tuples and $\circ$ is the composition operator defined on two sets of distance tuples. Label node $i \in V(T)$ with $\gamma(i)$. For any two node $i, j \in V(T)$, the composition of the labels along the path in T from i to j, which is $\gamma(i) \circ \ldots \circ \gamma(j)$, gives the set of distance tuples $P(X_i, X_j) = \{(a, b, \delta_G(a, b)) | a \in X_i, b \in X_j\}$. This is true because each edge along the path from i to j corresponds to a separator of G (see the text below Theorem 1. Thus for any two vertices $u, v \in V(G)$, if $u \in X_i$ and $v \in X_j$, then the composition of the labels along the path in T from i to j gives $\delta_G(u, v)$. The following theorem is proved in [1] and [7].

▶ **Theorem 4** ([1, 7])**.** *Let $(S, \circ)$ be a semigroup such that for $q, r \in S$, $q \circ r$ can be computed in $O(m)$ time. Let T be a tree with n nodes where each node is labeled with an element from the semigroup. After $O(mn)$ time preprocessing, the composition of the labels along any path in T can be computed in $O(m\alpha(n))$ time.*

Plug in the result in Theorem 4, after $O(t^3n)$ time preprocessing, the composition of the labels along any path in T of (X, T) can be computed in $O(t^3\alpha(n))$ time. Therefore the distance between any two vertices in $V(G)$ can be computed in $O(t^3\alpha(n))$ time.

The shortest path query algorithm uses the distance query algorithm. It works recursively and finds one intermediate vertex on the shortest path at a time. In a recursive step, the algorithm checks t vertices and for each vertex it makes two distance queries. Since a distance query is answered in $O(t^3\alpha(n))$ time, an intermediate vertex is found in $O(t^4\alpha(n))$ time. The shortest path is reported in $O(t^4\alpha(n)L)$ time, where L is the number of edges on the path. In summary:

▶ **Theorem 5** ([6])**.** *Let G be an n-vertex weighted digraph of treewidth t and assume a tree decomposition of G with treewidth t is given. After $O(t^3n)$ time preprocessing, distance queries in G can be answered in time $O(t^3\alpha(n))$, and shortest path queries in G can be answered in time $O(t^4\alpha(n)L)$, where L is the number of edges on the path.*

3.3 Our improved shortest path query algorithm

In this section we show how to improve the shortest path query algorithm. The shortest path query algorithm in Section 3.2 computes an intermediate vertex by checking t vertices and making $O(t)$ distance queries. However, we can define *augmented* distance tuples which contain intermediate vertex information, and define the composition operator $\circ$ on two augmented distance tuples such that the composition not only gives distance but also gives intermediate vertex. Using the augmented distance tuples and the composition operator on them, we can compute an intermediate vertex by making just one distance query.

An augmented distance tuple (a, b, c, d) has a fourth component d which is the intermediate vertex information. The composition operator $\circ$ on two augmented distance tuples is defined as $(a_1, b_1, c_1, d_1) \circ (a_2, b_2, c_2, d_2) = (a_1, b_2, c_1 + c_2, d')$ if $b_1 = a_2$, otherwise it is undefined. When $b_1 = a_2$, the fourth component d' is determined as follows. If $a_1 = b_1$ or $a_2 = b_2$, then $d' = d_2$ or $d' = d_1$, respectively. Otherwise, $a_1 \neq b_1$ and $a_2 \neq b_2$. Then if $a_1 \neq b_2$,$d' = b_1$. If $a_1 = b_2$, $d' = null$.

Now we redefine the semigroup $(\Gamma, \circ)$. An element in the redefined Γ is a set of augmented distance tuples. Let A and B be two sets of augmented distance tuples. The composition operator $\circ$ on A and B is defined as $A \circ B = minmap\{x \circ y | x \in A, y \in B\}$, where the *minmap* operation is as defined before. For a node $i \in V(T)$, let $\bar{\gamma}(i) = \{(u, v, \delta_G(u, v), Int_G(u, v)) | u, v \in X_i\}$. The values $\delta_G(u, v)$ and $Int_G(u, v)$ have been computed by Lemma 3. Relabel node $i \in V(T)$ with $\bar{\gamma}(i)$. The labels are elements in the redefined $(\Gamma, \circ)$. For any two nodes $i, j \in V(T)$, the composition of the labels along the path in T from i to j, $\bar{\gamma}(i) \circ \ldots \circ \bar{\gamma}(j)$, gives the set of augmented distance tuples $\bar{P}(X_i, X_j) = \{(a, b, \delta_G(a, b), Int_G(a, b)) | a \in X_i, b \in X_j\}$. For any two vertices $u, v \in V(G)$, if $u \in X_i$ and $v \in X_j$, then the composition of the labels along the path in T from i to j gives $\delta_G(u, v)$ and $Int_G(u, v)$.

The composition of the labels along a path in T gives the claimed set of augmented distance tuples, since each edge along the path corresponds to a separator of G.

For two labels $\bar{\gamma}(i)$ and $\bar{\gamma}(j)$, one can compute $\bar{\gamma}(i) \circ \bar{\gamma}(j)$ in $O(t^3)$ time. If we plug in the data structure in Theorem 4, then after $O(t^3 n)$ time preprocessing, the composition of the labels $\bar{\gamma}(\cdot)$ along any path in T can be computed in $O(t^3 \alpha(n))$ time. We have the following theorem.

▶ **Theorem 6.** *Let G be an n-vertex weighted digraph of treewidth t and assume a tree decomposition of G with treewidth t is given. After $O(t^3 n)$ time preprocessing, distance queries in G can be answered in time $O(t^3 \alpha(n))$, and shortest path queries in G can be answered in time $O(t^3 \alpha(n) L)$, where L is the number of edges on the path.*

4 Shortest beer path and beer distance queries

Throughout this section, let G be a beer digraph. In this section we extend the approach discussed in Section 3 to shortest beer path and beer distance queries. Given a tree decomposition (X, T) of the input digraph G, we apply the *absorption and expansion* processes similar to those in Section 3 on (X, T), which compute information on beer paths and beer distances. Then we define a semigroup whose elements are sets of augmented distance tuples and augmented beer distance tuples. Finally we use the data structure in Theorem 4 to answer shortest beer path and beer distance queries efficiently. Before describing the algorithms, we give some preliminaries.

4.1 Preliminaries

Let H be a beer digraph. We call a vertex $v \in V(H)$ a beer vertex if v has a beer store. Let $u, v \in V(H)$. Recall that we use $\pi_{BH}(u, v)$ to denote a shortest beer path in H from u to v and use $\delta_{BH}(u, v)$ to denote the the weight of $\pi_{BH}(u, v)$. Let $Int_{BH}(u, v)$ denote an intermediate vertex (neither u nor v) on $\pi_{BH}(u, v)$. If $wt(u, v) = \delta_{BH}(u, v)$, then $Int_{BH}(u, v) = null$. If either u or v is a beer vertex, a shortest path from u to v is a shortest beer path from u to v and we define $Int_{BH}(u, v)$ to be an intermediate vertex (neither u nor v) on $\pi_H(u, v)$. If neither u nor v is a beer vertex, we define $Int_{BH}(u, v)$ to be a beer vertex that is on $\pi_{BH}(u, v)$.

Corresponding to a shortest beer path $\pi_{BH}(u, v)$, we define a beer edge $\langle u, v \rangle_B$ which has weight $\delta_{BH}(u, v)$ and the intermediate vertex $Int_{BH}(u, v)$. Note that a beer edge is different from a normal edge. We extend Definition 2 to beer graphs.

▶ **Definition 7.** *Let H be a beer digraph, possibly with beer edges. Let V_1, V_2 and U be a partition of $V(H)$ such that U is a separator for V_1 and V_2. Let H_1 and H_2 be subgraphs of H such that $V(H_1) = V_1 \cup U$ and $V(H_2) = V_2 \cup U$. We say that H_1' is a beer graph* *obtained by absorbing H_2 into H_1, if H_1' is obtained from H_1 by adding edges $\langle u, v\rangle$, with weight $\delta_{H_2}(u, v)$ or $\delta_H(u, v)$, and beer edges $\langle u, v\rangle_B$, with weight $\delta_{BH_2}(u, v)$ or $\delta_{BH}(u, v)$, for each pair $u, v \in U$. In case multiple edges or multiple beer edges are created, retain the one with minimum weight.*

Absorption preserves distances and beer distances in a beer digraph. Let H, H_1, H_2 and H_1' be as in Definition 7, then for all $x, y \in V(H_1')$, $\delta_{H_1'}(x, y) = \delta_H(x, y)$ and $\delta_{BH_1'}(x, y) = \delta_{BH}(x, y)$.

For a node $i \in V(T)$ where (X, T) is a tree decomposition of G, define $\bar{\gamma}_B(i) = \{(u, v, \delta_G(u, v), Int_G(u, v))|u, v \in X_i\} \cup \{(u, v, \delta_{BG}(u, v), Int_{BG}(u, v))|u, v \in X_i\}$ to be the set of augmented distance tuples and augmented beer distance tuples for pairs of vertices in X_i.

Given a tree decomposition (X, T) of treewidth t of G, we use the absorption and expansion processes with the absorbing procedure defined in Definition 7, to compute $\bar{\gamma}(i)$ and $\bar{\gamma}_B(i)$ for all $i \in V(T)$. We have the following lemma.

▶ **Lemma 8.** *Let G be an n-vertex weighted digraph and let (X, T) be a tree decomposition of G, of treewidth t. Then in $O(t^3 n)$ time, we can compute the values $\delta_G(u, v)$, $Int_G(u, v)$, $\delta_{BG}(u, v)$ and $Int_{BG}(u, v)$ for each pair $u, v \in X_i$ for every $i \in V(T)$.*

4.2 Defining a semigroup

We first define a composition operator $\circ_B$ on augmented distance tuples and augmented beer distance tuples. When both operands are augmented distance tuples, the operator $\circ_B$ equals the operator $\circ$ defined in Section 3.1. The operator $\circ_B$ is undefined when both operands are augmented beer distance tuples. It remains to define $\circ_B$ between an augmented distance tuple and an augmented beer distance tuple. Let (a_1, b_1, c_1, d_1) be an augmented distance tuple and let $(a_2, b_2, \hat{c}_2, \hat{d}_2)$ be an augmented beer distance tuple. We only show the definition of $(a_1, b_1, c_1, d_1) \circ_B (a_2, b_2, \hat{c}_2, \hat{d}_2)$, since the definition of $(a_2, b_2, \hat{c}_2, \hat{d}_2) \circ_B (a_1, b_1, c_1, d_1)$ is symmetric. Define $(a_1, b_1, c_1, d_1) \circ_B (a_2, b_2, \hat{c}_2, \hat{d}_2) = (a_1, b_2, c_1 + \hat{c}_2, \hat{d})$ if $b_1 = a_2$, otherwise it is undefined. The tuple $(a_1, b_2, c_1 + \hat{c}_2, \hat{d})$ is a beer distance tuple. The fourth component $\hat{d}$ is set as follows. Here we assume that a_1, b_1 and b_2 are all different. The other cases are very similar. If a_1 or b_1 is a beer vertex, then $\hat{d} = b_1$. Otherwise neither a_1 nor b_1 is a beer vertex. Then, if a_2 or b_2 is a beer vertex, $\hat{d} = a_2$, otherwise $\hat{d} = \hat{d}_2$. It is not hard to verify that the setting of $\hat{d}$ is consistent with the definition of an augmented beer distance tuple. The setting of $\hat{d}$ takes constant time in all cases, therefore the composition $(a_1, b_1, c_1, d_1) \circ_B (a_2, b_2, \hat{c}_2, \hat{d}_2)$ takes constant time.

Let $\hat{A}$ and $\hat{B}$ be two sets of augmented distance tuples and augmented beer distance tuples. That is, $\hat{A} = A_1 \cup A_2$ and $\hat{B} = B_1 \cup B_2$ where A_1 (B_1) is a set of augmented distance tuples and A_2 (B_2) is a set of augmented beer distance tuples. Define the composition operator $\circ_B$ on $\hat{A}$ and $\hat{B}$ as $\hat{A} \circ_B \hat{B} = minmap\{x \circ_B y|x \in \hat{A}, y \in \hat{B}\}$, where the $minmap$ is the operation such that among all distance tuples (or all beer distance tuples) with the same first and second components, only the distance tuple (or the beer distance tuple) with the smallest third component is retained. The operation $\circ_B$ is associative.

Recall that

$$\bar{\gamma}_B(i) = \{(u, v, \delta_G(u, v), Int_G(u, v))|u, v \in X_i\} \cup \{(u, v, \delta_{BG}(u, v), Int_{BG}(u, v))|u, v \in X_i\}$$

for all $i \in V(T)$ can be computed in $O(t^3 n)$ time according to Lemma 8. We define a semigroup $(\Gamma_B, \circ_B)$ where Γ_B is the set of sets of augmented distance tuples and augmented beer distance tuples, $\circ_B$ is the composition operator defined on two elements in Γ_B. Label each node $i \in V(T)$ with $\bar{\gamma}_B(i)$. Since the composition $\circ_B$ of two tuples takes constant time, the composition of two labels takes $O(t^3)$ time. For any two nodes $i, j \in V(T)$, the composition of the labels along the path in T from i to j gives the set $\{(a, b, \delta_G(a, b), Int_G(a, b)) | a \in X_i, b \in X_j\} \cup \{(a, b, \delta_{BG}(a, b), Int_{BG}(a, b)) | a \in X_i, b \in X_j\}$.

Plug in Theorem 4, we have that after $O(t^3 n)$ time preprocessing, the composition of the labels along any path in T of (X, T) can be computed in $O(t^3 \alpha(n))$ time. Thus the beer distance between any two vertices $u, v \in V(G)$ and the intermediate vertex $Int_{BG}(u, v)$ can be computed in $O(t^3 \alpha(n))$ time.

The shortest beer path query algorithm is straightforward. If either u or v is a beer vertex, then we can use the shortest path query algorithm to compute the shortest beer path from u to v in $O(t^3 \alpha(n) L)$ time by Theorem 6, where L is the number of edges on the beer path. If neither u nor v is a beer vertex, we first make a beer distance query from u to v. The beer distance query returns the intermediate vertex $Int_{BG}(u, v)$, which is a beer vertex. Then we use the shortest path query algorithm to compute the shortest beer path from u to $Int_{BG}(u, v)$ and the shortest beer path from $Int_{BG}(u, v)$ to v. The concatenation of the two paths is a shortest beer path from u to v, which is computed in $O(t^3 \alpha(n) L)$ time. We have obtained the following theorem.

▶ **Theorem 9.** *Let G be an n-vertex beer digraph of treewidth t and assume a tree decomposition of G with treewidth t is given. After $O(t^3 n)$ time preprocessing on G, beer distance queries can be answered in time $O(t^3 \alpha(n))$, and shortest beer path queries can be answered in time $O(t^3 \alpha(n) L)$, where L is the number of edges on the path.*

5 A dynamic shortest beer path query structure

In this section we give a dynamic shortest beer path query structure, which is an extension of the dynamic shortest path query structure in [6]. We give an outline of the dynamic shortest path query structure in the following. The dynamic shortest beer path query structure is shown in Section 5.2.

The main technical tool is a graph partitioning algorithm. Let (X, T) be a *binary* tree decomposition of G of treewidth t. One can convert a tree decomposition of treewidth t into a binary tree decomposition of treewidth t in $O(tn)$ time. Given any integer m, the graph partitioning algorithm partitions the tree T of (X, T) in $O(n)$ time into $q \leq 16n/m$ node-disjoint subtrees $\{T_i | 1 \leq i \leq q\}$ such that T_i has at most m nodes and is connected to the rest of T via at most three nodes. For each T_i, a subgraph H_i which is the subgraph of G induced by vertices in $\bigcup_{v \in V(T_i)} X_v$ is created. The subgraph H_i has T_i as its tree decomposition. A reduced graph H' is also created. Let v_1, v_2 and v_3 be the nodes of subtree T_i via which T_i is connected to the rest of T. The set $C(H_i) = X_{v_1} \cup X_{v_2} \cup X_{v_3}$ is called the *cut set* of H_i and contains at most $3t$ vertices. By shrinking each subtree T_i into a node with $C(H_i)$ as its bag of vertices, one creates a reduced tree T' with $q \leq 16n/m$ nodes. The reduced graph H' has T' as its tree decomposition and includes edges in G joining pairs of vertices in $C(H_i)$, $1 \leq i \leq q$.

The input graph G is partitioned into edge-disjoint components $\{G_i | 1 \leq i \leq q\}$ where G_i is H_i with edges joining pairs of vertices in $C(H_i)$ deleted. Construct graph G' from H' by adding edges $\langle x, y \rangle$ weighted $\delta_{G_i}(x, y)$ for each pair x, y in $C(H_i)$, $1 \leq i \leq q$. Multiple edges in G' are replaced by the edge of minimum weight. Note that G' is the graph obtained by absorbing $G_1, \ldots, G_q$ into the rest of G. It follows that $\delta_{G'}(u, v) = \delta_G(u, v)$, for any $u, v \in V(G')$. By setting $m = 8\sqrt{n}$, H_i has at most $8t\sqrt{n}$ vertices and H' has at most $3tq \leq 6t\sqrt{n}$ vertices.

When the shortest path/distance query structures have been built for G' and G_i, $1 \leq i \leq q$, we can compute the shortest path/distance between any two vertices $u, v \in V(G)$ by querying the structures. If $u \in V(G_i)$ and $v \in V(G_j) \setminus V(G_i)$ for some i and j, we have

$$\delta_G(u,v) = \min\{\delta_{G_i}(u,x) + \delta_{G'}(x,y) + \delta_{G_j}(y,v) | x \in C(G_i), y \in C(G_j)\}. \tag{1}$$

If $u, v \in V(G_i)$ for some i, we have

$$\delta_G(u,v) = \min\{\delta_{G_i}(u,v), \min\{\delta_{G_i}(u,x) + \delta_{G'}(x,y) + \delta_{G_i}(y,v) | x, y \in C(G_i)\}\}. \tag{2}$$

Replacing the distances realizing $\delta_G(u,v)$ in Equation (1) or Equation (2) by the corresponding shortest paths gives the shortest path from u to v.

An edge in $E(G)$ corresponds to an edge in exactly one graph in $\{G_i | 1 \leq i \leq q\} \cup G'$. If the weight of the edge is updated and the edge is in G', one only needs to update the structure for G'. If the edge is in G_i, one needs to update the structure for G_i. Since an edge $\langle x, y\rangle$ with weight $\delta_{G_i}(x,y)$ was added in G' for each pair $x, y \in C(H_i)$, the change of an edge weight in G_i can incur at most $(3t)^2$ edge weight changes in G'. Thus one needs to update the structure for G' for these edge weight updates. To make an edge weight update efficient, the above procedure of graph partitioning and construction of a reduced graph is repeated *recursively* for each of $G_1, \ldots, G_q, G'$ until the component subgraphs at the deepest recursion level have small sizes. A *static* query structure in Section 3 is built for each component subgraph at the deepest recursion level. The dynamic data structure can be thought of as a tree structure where the root is G and every other node is $\bar{G}_i$ or $\bar{G}'$ of its parent $\bar{G}$. A static query structure is built and associated with each leaf node.

Since the dynamic data structure is built recursively, distance query or shortest path query is answered by recursively querying lower recursion level structures. Eventually queries are made and answered at the static query structures at the bottom level. An edge weight update is accommodated by recursively updating lower level structures. Eventually an update at a static query structure at the bottom level is accommodated by rebuilding the static query structure.

Let $r-1$ denote the number of recursion steps. The dynamic shortest path query structure is summarized in the following theorem.

▶ **Theorem 10** ([6]). *Let G be an n-vertex weighted digraph and assume a binary tree decomposition of G with treewidth t is given. For any positive integer constant r, one can build a data structure in $O(C_r t^{r+3} n)$ time such that the structure answers distance queries in $O(C_r t^{2r+2}\alpha(n))$ time, answers shortest path queries in $O(C_r t^{2r+2}\alpha(n)L)$ time where L is the number of edges on the shortest path, and accommodates an edge weight update in $O(C_r t^{2r+2} n^{(1/2)^{r-1}})$ time, where $C_r = 3^{r(r+2)}$.*

5.1 On the shortest path query time

We observe that the shortest path query algorithm only needs to compute the shortest path from u to v after making all the distance queries used to determine the shortest path from u to v. The shortest path from u to v is computed by making shortest path queries to static query structures at the bottom level and concatenating the computed shortest subpaths together, where a shortest subpath is a subpath of the shortest path from u to v. A shortest path query to a static query structure is answered in $O((3^{r-1}t)^4\alpha(n)L_i)$ time, where $3^{r-1}t$ is the maximum treewidth of a graph at the bottom recursion level and L_i is the number of edges on the computed shortest subpath. Therefore the shortest path from u to v can be computed in $O(\sum_i (3^{r-1}t)^4\alpha(n)L_i) = O(3^r t^4 \alpha(n) L)$ time where L is the number of edges on the shortest path from u to v, besides the $O(C_r t^{2r+2}\alpha(n))$ time spent on making all the distance queries. If we use the shortest path query algorithm in Theorem 6 of Section 3.3, the $O(3^r t^4 \alpha(n) L)$ time is improved to $O(3^r t^3 \alpha(n) L)$ time. We have the following corollary.

▶ **Corollary 11.** *With the static shortest path query structure in Theorem 6, the shortest path query time in Theorem 10 can be improved to $O(C_r t^{2r+2}\alpha(n) + 3^r t^3 \alpha(n) L)$.*

5.2 Dynamic shortest beer path and beer distance queries

Equipped with the graph partitioning technique and the shortest beer path query structure in Theorem 9, we are ready to devise a dynamic shortest beer path and beer distance query structure, which is an extension of the dynamic shortest path query structure in Theorem 10. In the following we only focus on the differences.

We use the graph partitioning algorithm to partition the input beer graph G into edge-disjoint components $\{G_i | 1 \le i \le q\}$ where $q \le 2\sqrt{n}$, the same as was done by the dynamic shortest path query structure. The reduced graph H' is the same as in the dynamic shortest path query structure. A reduced beer graph G' is constructed from H' by adding edges $\langle x, y\rangle$ weighted $\delta_{G_i}(x, y)$ and beer edges $\langle x, y\rangle_B$ weighted $\delta_{BG_i}(x, y)$ for each pair $x, y \in C(H_i)$, $1 \le i \le q$. Edges and beer edges are dealt with separately. Multiple edges in G' are replaced by the edge of minimum weight, and multiple beer edges in G' are replaced by the beer edge of minimum weight. The beer graph G' is obtained by absorbing $G_1, \ldots, G_q$ into the rest of G using the absorption defined in Definition 7. It follows that $\delta_{G'}(u, v) = \delta_G(u, v)$ and $\delta_{BG'}(u, v) = \delta_{BG}(u, v)$, for any $u, v \in V(G')$.

When the shortest path query structures and the shortest beer path query structures have been built for G' and G_i, $1 \le i \le q$, we can compute the shortest beer path or beer distance between any two vertices $u, v \in V(G)$. If $u \in V(G_i)$ and $v \in V(G_j) \setminus V(G_i)$ for some i and j, we have

$$\delta_{BG}(u, v) = \min\{M_1, M_2, M_3\}, where \tag{3}$$
$$M_1 = \min\{\delta_{BG_i}(u, x) + \delta_{G'}(x, y) + \delta_{G_j}(y, v) | x \in C(G_i), y \in C(G_j)\}, \tag{4}$$
$$M_2 = min\{\delta_{G_i}(u, x) + \delta_{BG'}(x, y) + \delta_{G_j}(y, v) | x \in C(G_i), y \in C(G_j)\}, \tag{5}$$
$$M_3 = min\{\delta_{G_i}(u, x) + \delta_{G'}(x, y) + \delta_{BG_j}(y, v) | x \in C(G_i), y \in C(G_j)\}. \tag{6}$$

If $u, v \in V(G_i)$ for some i, we have

$$\delta_{BG}(u, v) = \min\{M_1, M_2, M_3\}, where \tag{7}$$
$$M_1 = \min\{\delta_{BG_i}(u, v), \min\{\delta_{BG_i}(u, x) + \delta_{G'}(x, y) + \delta_{G_i}(y, v) | x, y \in C(G_i)\}\}, \tag{8}$$
$$M_2 = \min\{\delta_{BG_i}(u, v), \min\{\delta_{G_i}(u, x) + \delta_{BG'}(x, y) + \delta_{G_i}(y, v) | x, y \in C(G_i)\}\}, \tag{9}$$
$$M_3 = \min\{\delta_{BG_i}(u, v), \min\{\delta_{G_i}(u, x) + \delta_{G'}(x, y) + \delta_{BG_i}(y, v) | x, y \in C(G_i)\}\}. \tag{10}$$

Replacing the beer distances realizing $\delta_{BG}(u, v)$ in Equation (3) or Equation (7) by the corresponding shortest beer paths gives the shortest beer path from u to v.

An edge in $E(G)$ corresponds to an edge in exactly one graph in $\{G_i | 1 \le i \le q\} \cup G'$. If an edge weight is updated and the edge is in G', one only needs to update the structure for G'. If the edge is in G_i, one needs to update the structure for G_i and G'. Since an edge $\langle x, y\rangle$ with weight $\delta_{G_i}(x, y)$ and a beer edge $\langle x, y\rangle_B$ with weight $\delta_{BG_i}(x, y)$ were added in G' for each pair $x, y \in C(H_i)$, the change of an edge weight in G_i can incur at most $2(3t)^2$ edge or beer edge weight changes in G'. The procedure of graph partitioning and construction of a reduced beer graph is repeated recursively for each of $G_1, \ldots, G_q, G'$ until the component subgraphs at the deepest recursion level have small sizes. A static shortest beer path query structure in Theorem 9 and a static shortest path query structure in Section 3 are built for each component subgraph at the deepest recursion level.

Since the dynamic data structure is built recursively, a beer distance query or a shortest beer path query is answered by recursively querying lower recursion level structures. Eventually queries are made and answered at the static query structures at the bottom level. An edge weight update is accommodated by recursively updating lower level structures. An update at a static shortest path (or shortest beer path) query structure at the bottom level is accommodated by rebuilding the static query structure.

Let $r-1$ denote the number of recursion steps. We omit the details of calculating the preprocessing/query/update times of the dynamic data structure, since the calculations are similar to the calculations for the dynamic shortest path structure in Theorem 10 and Corollary 11. The dynamic data structure is summarized in the following theorem.

▶ **Theorem 12.** *Let G be an n-vertex beer digraph with treewidth t and assume a binary tree decomposition of G with treewidth t is given. For any positive integer constant r, after $O(C(r)t^{r+3}n)$ time preprocessing, beer distance queries can be answered in $O(C(r)t^{2r+2}\alpha(n))$ time, shortest beer path queries can be answered in $O(C(r)t^{2r+2}\alpha(n) + 3^r t^3 \alpha(n) L)$ time, where $C(r) = 3^{r(r+3)}$ and L is the number of edges on the shortest beer path. The data structure updates in $O(C(r)t^{2r+2}n^{(1/2)^{r-1}})$ time after an edge weight change.*[2]

Combined with Theorem 1, we obtain the following theorem by setting $r = 1 - \log\beta$.

▶ **Theorem 13.** *Let G be an n-vertex beer digraph with constant treewidth, and let β be any constant in $(0,1)$. After $O(n)$ time preprocessing, beer distance queries can be answered in $O(\alpha(n))$ time, shortest beer path queries can be answered in $O(\alpha(n)L)$ time where L is the number of edges on the shortest beer path. Edge weight updates (including edge deletions and edge re-insertions) can be performed in $O(n^\beta)$ time.*

6 Another dynamic shortest path query structure

In this section we give another dynamic shortest path query structure. The basic structure is a balanced tree $BST(G)$ which represents a *balanced separator decomposition* of G. We build compressed graphs for the associated subgraphs of nodes in $BST(G)$ so that the distances between vertices in different separators can be quickly computed. Finally, we partition edges in $E(G)$ among nodes in $BST(G)$ so that edge weight update can be handled efficiently.

We give some preliminaries in Section 6.1, and describe the structure in Section 6.2, Section 6.3 and Section 6.4.

6.1 Preliminaries

Let (X,T) be a binary tree decomposition of G. Recall that an edge $e=(i,j)$ in $E(T)$ corresponds to $X_i \cap X_j$, which is a separator of G. We can get a decomposition of G alongside a decomposition of T. Let ϕ denote the separator that an edge e in $E(T)$ corresponds to. The removal of e from T divides T into two subtrees T_1 and T_2, the removal of ϕ from G divides G into two disjoint subgraphs $G[V_1]$ and $G[V_2]$. We add the separator ϕ and the edges in $E(G)$ between vertices in ϕ and vertices in V_1 to $G[V_1]$, and get the subgraph $G_1 = G[V_1] \cup \phi \cup E_1$

[2] One can detect whether the edge weight change incurs a negative cycle, by using Lemma 3 at the partitioned component subgraphs.

where E_1 is the set of edges in $E(G)$ joining vertices in ϕ and vertices in V_1. We add the separator ϕ and the edges in $E(G)$ between vertices in ϕ and vertices in V_2 to $G[V_2]$, and get the subgraph $G_2 = G[V_2] \cup \phi \cup E_2$ where E_2 is the set of edges in $E(G)$ joining vertices in ϕ and vertices in V_2. Thus G is divided into G_1, G_2 and $G[\phi]$ where $G[\phi]$ is the subgraph of G induced by vertices in ϕ. See Figure 2(a). We call $G[\phi]$ a *separator subgraph.* We have that T_1 is a tree decomposition of G_1 and T_2 is a tree decomposition of G_2. Note that G_1, G_2 and $G[\phi]$ partitions edges in $E(G)$.

If we repeat the division step for T_1 and G_1, and for T_2 and G_2 recursively until the subtrees have $O(1)$ nodes, we obtain a decomposition of G. We use a tree DT to represent the decomposition. Each node p of DT corresponds to a subtree of T and is associated with a subgraph of G, denoted as G_p. If p is an internal node of DT, p is also associated with an edge e_p in $E(T)$ and a separator ϕ_p which corresponds to e_p. Recall that $G_p[\phi_p]$ is called a separator subgraph, and we associate p with $G_p[\phi_p]$. Since the division stops at a leaf node of DT, a leaf node has no associated separator or separator subgraph.

Let Φ denote the set of separators associated with nodes of DT. The separators of Φ that separate G_p from the rest of G are called the *cut separators* of G_p. See Figure 2(b) for an illustration.

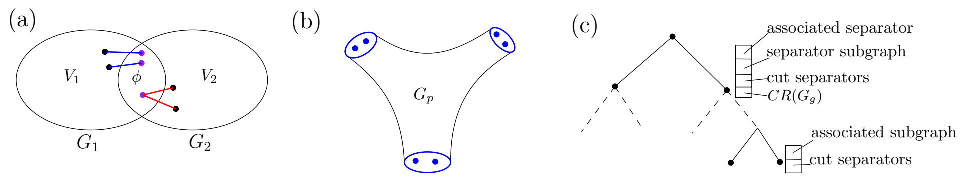

Figure 2 (a) $G_1 = G[V_1] \cup \phi \cup E_1$ where E_1 is the set of edges in $E(G)$ joining vertices in ϕ and vertices in V_1. The blue edges are in E_1. $G_2 = G[V_2] \cup \phi \cup E_2$ where E_2 is the set of edges in $E(G)$ joining vertices in ϕ and vertices in V_2. The red edges are in E_2. G_1, G_2 and $G[\phi]$ partitions edges in $E(G)$. (b) The cut separators of G_p are in blue. (c) The $BST(G)$ structure.

Constructing a balanced separator decomposition

We give an algorithm that outputs a decomposition of G that will be used in the construction of the data structure in Section 6.2. The decomposition is the construction of a DT using balanced separators. A balanced separator corresponds to an edge of a tree decomposition whose removal partitions the tree decomposition into two subtrees of proportional sizes. The decomposition fulfills two goals: (1) the height of the DT is $O(\log n)$ and (2) any subgraph associated with a node of DT has constant size separators. If in the decomposition of T we successively use an edge of a subtree whose removal partitions the subtree into two subtrees of proportional sizes, we get a *balanced decomposition* of T which fulfills the first goal.

Guibas et al. [14] showed an algorithm that computes a balanced decomposition of a binary tree in linear time. They called the balanced decomposition a *centroid decomposition.* In their algorithm the binary tree is decomposed by removing a *centroid edge* which partitions the binary tree into two subtrees, each of size at least $(|T| + 1)/3$. A centroid edge is found by finding a node that is a centroid of the tree. When each of the subtrees is partitioned similarly and this is repeated recursively until the subtrees are single nodes, we obtain a balanced binary tree structure which is the balanced decomposition of the binary tree. For our purpose, we perform a centroid decomposition on T until the subtrees have at most 5 nodes.

To fulfill the second goal, we modify the centroid decomposition procedure by using the following step when choosing the edge used to partition a subtree. Edges in $E(T)$ which are used for partitioning subtrees are called *partition edges*. If a subtree is connected to the rest of T through at most three partition edges, we just choose an centroid edge of the subtree as the partition edge (used to partition the subtree). If a subtree is connected to the rest of T through four partition edges, we choose an edge of the subtree whose removal partitions the subtree into two subtrees, either of which is connected to the rest of T through two of the four partition edges. Thus in every two levels of recursion, the size of a subtree is reduced by a factor of at least $1/3$.

If we use the above modified centroid decomposition on T to get a decomposition of G, we get a balanced DT which we call a *balanced separator tree* of G, denoted as $BST(G)$.

$BST(G)$ is a balanced decomposition of G which meets the two goals. The second goal is fulfilled since any subgraph associated with a node of $BST(G)$ has at most four cut separators, due to the fact that any subtree in the modified centroid decomposition on T is connected to the rest of T through at most four partition edges. The two goals will support the shortest path and distance queries in Section 6.3. For an internal node of $BST(G)$, we store with it its associated separator and its separator subgraph. For a leaf node of $BST(G)$, we store with its associated subgraph. For any node p of $BST(G)$, we store with it pointers to edges in $E(T)$ that correspond to the cut separators of G_p. See Figure 2(c) for an illustration.

Assume that a binary tree decompostion (X, T) of G with treewidth t is given. The modified centroid decomposition on T takes $O(n)$ time. Storing the associated separators and separator subgraphs at internal nodes of $BST(G)$ takes $O(t^2 n)$ time. The associated subgraph of a leaf node in $BST(G)$ has $O(t)$ vertices, so storing the associated subgraphs at leaf nodes of $BST(G)$ takes $O(t^2 n)$ time. In summary:

▶ **Lemma 14.** *Given a binary tree decomposition (X, T) of G with treewidth t, we can build $BST(G)$ in $O(t^2 n)$ time. The depth of $BST(G)$ is $O(\log n)$. The associated subgraph of a node in $BST(G)$ has at most four cut separators.*

6.2 The preprocessing

Given a binary tree decompostion (X, T) of G, we preprocess T and G to output the following data structures:

1. The $BST(G)$ with $CR(G_g)$ for each internal node g in $BST(G)$. $CR(G_g)$ is a compressed representation of g's associated subgraph G_g, which will be described below.
2. An array $Loc[\cdot]$ for vertices in $V(G)$, where $Loc[v]$ is a leaf node of $BST(G)$ whose associated subgraph contains vertex v in $V(G)$.
3. An LCA (lowest common ancestor) structure for $BST(G)$.

We have the following definition.

▶ **Definition 15.** *Let $BST(G)$ be the balanced separator tree of G and let p be a node of $BST(G)$. If p is an internal node g, let $CR(G_g)$ be a clique for G_g with respect to vertices in ϕ_g and the cut separators of G_g. The weight of an edge in the clique is the distance in G_g between the vertices of the edge. We call $CR(G_g)$ the compressed representation of G_g. If p is a leaf node, we have $CR(G_p) = G_p$.*

Given $BST(G)$, we compute $CR(G_g)$ for each internal node g of $BST(G)$ in a bottom-up fashion. Let t_1 and t_2 be g's children. We take the union of $CR(G_{t_1})$, $CR(G_{t_2})$ and g's associated separator subgraph $G_g[\phi_g]$, run the Floyd-Warshall algorithm on the union graph, and build a clique on vertices in ϕ_g and the cut separators of G_g where the weight of an edge is the distance computed for the vertices of the edge.

The correctness of the clique construction procedure is supported by the following lemma.

▶ **Lemma 16.** *lemmaCRunion Let g be an internal node of $BST(G)$ and let ϕ_g be its associated separator. Let t_1 and t_2 be g's children in $BST(G)$. Let G_{t_1} be t_1's associated subgraph and let G_{t_2} be t_2's associated subgraph. The distance between any two vertices in $CR(G_{t_1}) \cup CR(G_{t_2}) \cup G_g[\phi_g]$ equals the distance between the two vertices in G_g.*

Note that the associated subgraphs of leaf nodes in $BST(G)$ and the separator subgraphs of internal nodes in $BST(G)$ *partition* edges in $E(G)$. This property is used for efficiently updating an edge weight in Section 6.4.

The preprocessing consists of two steps: building the $BST(G)$, and constructing the $CR(G_g)$ for each internal node g of $BST(G)$. The associated subgraph of an internal node g in $BST(G)$ has at most four cut separators, each of which contains at most t vertices. Thus $CR(G_g)$ has $O(t)$ vertices and computing $CR(G_g)$ takes $O(t^3)$ time, dominated by the Floyd-Warshall algorithm. There are $O(n)$ internal nodes, so the second step takes $O(t^3n)$ time. Combined with Lemma 14, we have the following lemma.

▶ **Lemma 17.** *Given a binary tree decomposition (X,T) of G with treewidth t, we can build $BST(G)$ and $CR(G_g)$ for each internal node g of $BST(G)$ in $O(t^3n)$ time.*

The array $Loc[\cdot]$ for vertices in $V(G)$ can be constructed by scanning the associated subgraphs of leaf nodes of $BST(G)$. There are $O(n)$ leaf nodes. The associated subgraph of a leaf node has $O(t)$ vertices. Thus $Loc[\cdot]$ can be constructed in $O(tn)$ time.

We can build an LCA structure for $BST(G)$ in $O(n)$ time [3, 16].

▶ **Corollary 18.** *The preprocessing outputs the $BST(G)$ with $CR(G_g)$, $Loc[\cdot]$ and an LCA structure for $BST(G)$ in $O(t^3n)$ time.*

6.3 Answering shortest path and distance queries

We describe the distance query algorithm first. Let u and v be any two vertices of G. The distance query algorithm consists of three steps: (1) find the node s of $BST(G)$ whose associated separator separates u from v, (2) compute the distances in G between any two vertices in ϕ_a, where ϕ_a is the associated separator of an ancestor node a of s, and (3) compute the distances in G from u to vertices in ϕ_s and the distances in G from vertices in ϕ_s to v, from which the distance from u to v is computed.

In step 1 we use the array $Loc[\cdot]$ to locate the leaf node $l(u)$ of $BST(G)$ whose associated subgraph contains u and locate the leaf node $l(v)$ whose associated subgraph contains v. We use the LCA structure on $BST(G)$ to find the vertex s, which is the lowest common ancestor of $l(u)$ and $l(v)$ in $BST(G)$.

Step 2 makes use of the compressed representations. Let g be an internal node of $BST(G)$ and let $D(\phi_g)$ denote the distances in G between any two vertices in ϕ_g. Let r be the root of $BST(G)$ and let a be an ancestor of s. We compute the distances top-down, for nodes on the path from r to s. $CR(G)$ at r is a clique containing the distances in G between any two vertices in ϕ_r, thus $D(\phi_r)$ has already been computed. Next let b be a's child that is on the path from r to s and assume we have computed $D(\phi_r), \ldots, D(\phi_a)$. We can use $CR(G_b)$ and $D(\phi_r), \ldots, D(\phi_a)$ to compute $D(\phi_b)$. The associated subgraph G_b of b is separated from the rest of G by at most four cut separators, which are among $\{\phi_r, \ldots, \phi_a\}$. Recall that we have stored, with each node p of $BST(G)$, pointers to the cut separators of G_p. If we add to $CR(G_b)$ edges with weights equal to distances in $D(\phi_x)$, where ϕ_x is a cut separator of G_b, the distance between any two vertices in the resulting graph equals the distance in G between the two vertices. This is because absorption preserves distances in G (see Definition 2). Thus we can run the Floyd-Warshall algorithm on the resulting graph to compute $D(\phi_b)$ in $O(t^3)$ time, recalling that any compressed representation has $O(t)$ vertices. Therefore we can compute $D(\phi_r), \ldots, D(\phi_s)$ top-down from r to s, where each $D(\cdot)$ is computed in $O(t^3)$ time.

In step 3 we compute the distances in G from u to vertices in ϕ_s recursively. Computing the distances in G from vertices in ϕ_s to v is symmetric so we only discuss the former. Let s_1 be the child of s that is on the path from s to $l(u)$. We have computed $D(\phi_r), \ldots, D(\phi_q)$ in step 2, where q is the parent of s. Let $D(u, \phi_s)$ denote the distances in G from u to vertices in ϕ_s. We compute $D(u, \phi_s)$ at s_1 and differentiate between two cases:

Case 1: ϕ_{s_1} separates u from ϕ_s in G_{s_1}. See Figure 3(a). For a vertex v_s in ϕ_s, we have

$$d_G(u, v_s) = \min_{v_{s_1} \in \phi_{s_1}} \{d_G(u, v_{s_1}) + d_G(v_{s_1}, v_s)\}. \tag{11}$$

Let $D(u, \phi_{s_1}) = \{d_G(u, v_{s_1}) | v_{s_1} \in \phi_{s_1}\}$ and let $D(\phi_{s_1}, \phi_s) = \{d_G(v_{s_1}, v_s) | v_{s_1} \in \phi_{s_1}, v_s \in \phi_s\}$. Given $D(\phi_r), \ldots, D(\phi_s)$, we add edges to $CR(G_{s_1})$ with weights equal to distances in $D(\phi_x)$, where ϕ_x is a cut separator of G_{s_1}. Then we run the Floyd-Warshall algorithm on the resulting graph to compute $D(\phi_{s_1})$ and $D(\phi_{s_1}, \phi_s)$. Thus in this case, computing $D(u, \phi_s)$ is reduced to computing $D(u, \phi_{s_1})$, which is computed recursively at s' where s' is the child of s_1 on the path from s to $l(u)$.

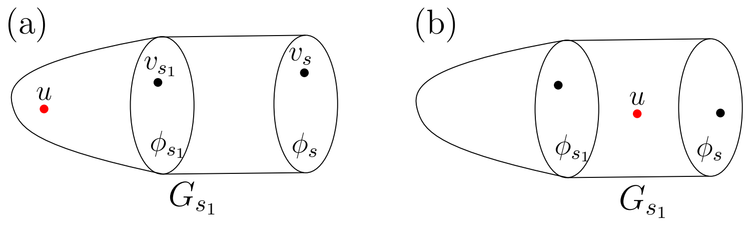

Figure 3 (a) ϕ_{s_1} separates u from ϕ_s in G_{s_1}. (b) ϕ_{s_1} does not separate u from ϕ_s in G_{s_1}.

Case 2: ϕ_{s_1} does not separate u from ϕ_s in G_{s_1}. See Figure 3(b). For this case, we add edges to $CR(G_{s_1})$ with weights equal to distances in $D(\phi_x)$, where ϕ_x is a cut separator of G_{s_1}. Then we run the Floyd-Warshall algorithm on the resulting graph to compute $D(\phi_{s_1})$. We recursively compute $D(u, \phi_s)$ at s', since ϕ_s is a cut separator of $G_{s'}$.

Recursion stops when we arrive at $l(u)$. Since we have computed $D(\phi_r), \ldots, D(\phi_y)$ where y is the parent of $l(u)$, we can add edges and run the Floyd-Warshall algorithm to compute $D(u, \phi_y)$ or $D(u, \phi_s)$, as desired.

Step 1 takes $O(1)$ time. Step 2 computes $D(\phi_r), \ldots, D(\phi_s)$ top-down from r to s, where each $D(\cdot)$ is computed in $O(t^3)$ time. Since $BST(G)$ has depth $O(\log n)$, step 2 takes $O(t^3 \log n)$ time. Each recursion step in step 3 takes $O(t^3)$ time, dominated by running the Floyd-Warshall algorithm. The computation at $l(u)$ also takes $O(t^3)$ time. Thus step 3 takes $O(t^3 \log n)$ time. In summary:

▶ **Lemma 19.** *Let u and v be any two vertices in G. The distance in G from u to v can be computed in $O(t^3 \log n)$ time where t is the treewidth of G.*

6.3.1 Shortest path query algorithm

To answer shortest path queries efficiently, we add a preprocessing step *PathExtract_Pre*. *PathExtract_Pre* works on $BST(G)$ bottom-up and stores with each internal node g the complete APSP information when running the Floyd-Warshall algorithm. In analogy to the distance query algorithm, the shortest path query algorithm works down the path from the root to $l(u)$ and retains APSP information when running the Floyd-Warshall algorithm. The APSP information allows us to extract the shortest path in G between any two vertices in $CR(G_a)$ where a is an ancestor of $l(u)$, in time linear to the number of edges on the shortest path. We have the following lemma.

▶ **Lemma 20.** *Let u and v be any two vertices in G. One can preprocess G in $O(t^3 n)$ time and with $O(t^3 n)$ space so that the shortest path in G from u to v can be reported in $O(t^3 \log n + L)$ time where L is the number of edges on the shortest path.*

6.4 Handling edge weight update

We have the following lemma.

▶ **Lemma 21.** *The query structure can be updated in $O(t^3 \log n)$ time for an edge weight update.*

Combined with Corollary 18, Lemma 19 and Lemma 20, we obtain the main theorem of this section.

▶ **Theorem 22.** *Let G be an n-vertex digraph of treewidth t and assume a binary tree decomposition of G with treewidth t is given. One can preprocess G in $O(t^3 n)$ time and with $O(t^3 n)$ space, so that distance queries can be answered in $O(t^3 \log n)$ time, and shortest path queries can be answered in time $O(t^3 \log n + L)$, where L is the number of edges on the path. The data structure can be updated in $O(t^3 \log n)$ time for an edge weight update.*

7 A dynamic shortest beer path query structure

We can extend the structure in Section 6 to handle shortest beer paths. The general idea is to add beer edges and to compute not only APSP, but also APSBP (all pairs shortest beer path). The query structure is summarized in the following theorem.

▶ **Theorem 23.** *Let G be an n-vertex beer digraph with treewidth t and assume a binary tree decomposition of G with treewidth t is given. One can preprocess G in $O(t^3 n)$ time and with $O(t^3 n)$ space, so that beer distance queries can be answered in $O(t^3 \log n)$ time, shortest beer path queries can be answered in $O(t^3 \log n + L)$ time where L is the number of edges on the shortest beer path. The query structure can be updated in $O(t^3 \log n)$ time for an edge weight update.*

References

1 Noga Alon and Baruch Schieber. Optimal preprocessing for answering on-line product queries. Technical Report No.71/87, Tel Aviv University, 1987.

2 Joyce Bacic, Saeed Mehrabi, and Michiel Smid. Shortest beer path queries in outerplanar graphs. In *Proceedings of the 32nd International Symposium on Algorithms and Computation (ISAAC)*, volume 212 of *LIPIcs*, pages 62:1–62:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021.

3 Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In *Proceedings of the 4th Latin American Symposium on Theoretical Informatics (LATIN)*, pages 88–94, 2000.

4 Hans L. Bodlaender. A linear time algorithm for finding tree-decompositions of small treewidth. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing (STOC)*, pages 226–234, 1993.

5 Panagiotis Charalampopoulos, Pawel Gawrychowski, Shay Mozes, and Oren Weimann. Almost optimal distance oracles for planar graphs. In *Proceedings of the 51st Symposium on Theory of Computing (STOC)*, pages 138–151. ACM, 2019.

6 Shiva Chaudhuri and Christos D. Zaroliagis. Shortest paths in digraphs of small treewidth. part I: sequential algorithms. *Algorithmica*, 27(3):212–226, 2000.

7 Bernard Chazelle. Computing on a free tree via complexity-preserving mappings. *Algorithmica*, 2:337–361, 1987.

8 Danny Z. Chen and Jinhui Xu. Shortest path queries in planar graphs. In *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing (STOC)*, pages 469–478. ACM, 2000.

9 Vincent Cohen-Addad, Søren Dahlgaard, and Christian Wulff-Nilsen. Fast and compact exact distance oracle for planar graphs. In *Proceedings of the 58th IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, pages 962–973. IEEE Computer Society, 2017.

10 Hristo N. Djidjev. On-line algorithms for shortest path problems on planar digraphs. In *Proceedings of the 22nd International Workshop on Graph-Theoretic Concepts in Computer Science (WG)*, volume 1197 of *Lecture Notes in Computer Science*, pages 151–165, 1996.

11 Jittat Fakcharoenphol and Satish Rao. Planar graphs, negative weight edges, shortest paths, near linear time. In *42nd Annual Symposium on Foundations of Computer Science, (FOCS)*, pages 232–241, 2001.

12 Fedor V. Fomin, Daniel Lokshtanov, Saket Saurabh, Michal Pilipczuk, and Marcin Wrochna. Fully polynomial-time parameterized computations for graphs and matrices of low treewidth. *ACM Trans. Algorithms*, 14(3):34:1–34:45, 2018.

13 Viktor Fredslund-Hansen, Shay Mozes, and Christian Wulff-Nilsen. Truly subquadratic exact distance oracles with constant query time for planar graphs. In *Proceedings of the 32nd International Symposium on Algorithms and Computation (ISAAC)*, volume 212 of *LIPIcs*, pages 25:1–25:12. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021.

14 Leonidas J. Guibas, John Hershberger, Daniel Leven, Micha Sharir, and Robert Endre Tarjan. Linear-time algorithms for visibility and shortest path problems inside triangulated simple polygons. *Algorithmica*, 2:209–233, 1987.

15 Tesshu Hanaka, Hirotaka Ono, Kunihiko Sadakane, and Kosuke Sugiyama. Shortest beer path queries based on graph decomposition, 2023. `arXiv:2307.02787`.

16 Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984.

17 Tuukka Korhonen. A single-exponential time 2-approximation algorithm for treewidth. In *Proceedings of the 62nd IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, pages 184–192, 2021.

18 Yaowei Long and Seth Pettie. Planar distance oracles with better time-space tradeoffs. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2517–2537. SIAM, 2021.

19 Shay Mozes and Christian Sommer. Exact distance oracles for planar graphs. In *Proceedings of the 23rd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 209–222. SIAM, 2012.

20 Seth Pettie. An inverse-ackermann type lower bound for online minimum spanning tree verification. *Combinatorica*, 26(2):207–230, 2006.

21 Liam Roditty and Uri Zwick. On dynamic shortest paths problems. In *Proceedings of the 12th Annual European Symposium (ESA)*, volume 3221 of *Lecture Notes in Computer Science*, pages 580–591. Springer, 2004.

22 Mikkel Thorup and Uri Zwick. Approximate distance oracles. *J. ACM*, 52(1):1–24, 2005.

23 Virginia Vassilevska Williams and Ryan Williams. Subcubic equivalences between path, matrix and triangle problems. In *51th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 645–654. IEEE Computer Society, 2010.