

Project Report

An Implementation of Single-Pair and Single-Source Shortest Beer Path Algorithm in Maximal Outerplanar Graph

Charles Nguyen

Table of Contents

- I. Introduction
- II. Algorithms
 - 1. Preprocess
 - 1.1 (Lemma 3 – 4): Queries in Trees
 - 1.2 (Lemma 7): Beer Distance
 - 1.3 (Lemma 12 - 13 – 14): Report weight and instance of shortest path on the v -chain
 - 1.4 (Lemma 15): Report beer weight and instance of shortest beer path on the v -chain
 - 2. Answer Shortest Beer Path Queries
 - 2.1 Single-Pair Shortest (Beer) Path - *SPSP* and *SPSBP*
 - 2.2 Single-Source Shortest (Beer) Path - *SSSP* and *SSSBP*
 - 3. Verification
- III. Implementation
 - 1. Construct Maximal Outerplanar Graph
 - 2. Data Structure
 - 2.1 Lowest Common Ancestor
 - 2.2 DCEL
 - 3. Beer Distance
 - 4. Answer Shortest Beer Path Query
 - 4.1 Build DAG
 - 4.2 Dynamic Programming
- IV. Analysis
 - 1. Compare Dual of Graph
 - 2. Compare Beer Probability on Graph

I. Introduction

The project is a faithful attempt to explain and implement the shortest (beer) path algorithms in the work of J. Bacic, S. Mehrabi and M. Smid [?] as well as report and record its practical efficiency and behavior. The organization and content of this project report is fully in reference to the paper, specifically the figures, lemmas, theorems and notations. Additionally, much of the technical information also belongs to the original draft of the paper (by courtesy of J. Bacic).

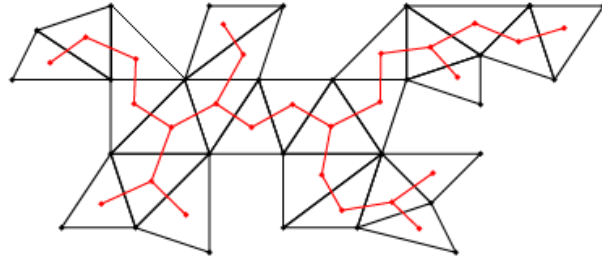
Given an undirected outerplanar graph $G(V, E)$ with positive edge weight and some vertices being beer store, we want to report the shortest path between any 2 given vertices that passes through at least 1 beer store. The result of the problem is fully stated in the following theorem:

Theorem 2: *Let G be an outerplanar beer graph with n vertices. We can preprocess G in $O(n)$ time into a data structure of size $O(n)$, such that for any two vertices u and v , the shortest beer path from u to v can be reported in $O(L)$ time, where L is the number of vertices on this beer path.*

We denote such a path as $SP_B(u, v, G)$ and its weight as $dist_B(u, v, G)$; but we prefer to use the shorthand $SP_B(u, v)$ and $dist_B(u, v)$ when we are referring to our original beer graph G . Similarly, we denote the shortest path from u to v and its weight as $SP(u, v)$ and $dist(u, v)$, respectively.

For a subgraph H of the beer graph G containing some vertices u and v , the shortest path between u and v must reside entirely in H (due to the outerplanarity of the graph G , but it is not necessary the case for the shortest beer path), and we denote such shortest path as $SP(u, v, H)$ and its weight as $dist(u, v, H)$. Likewise, we also have $SP_B(u, v, H)$ and $dist_B(u, v, H)$.

Note that the result is for arbitrary outerplanar graph, but it would require a linear preprocessing step of triangulation to convert it to maximal outerplanar graph. The implementation currently focuses on the maximal outerplanar graph and throughout the rest of the paper, G is assumed to have been fully triangulated. The algorithm exploits a property of a maximal outerplanar graph, namely its weak dual, denoted as $D(G)$, which is a tree:



II. Algorithms

1. Preprocessing

1.1 Queries on Trees (Lemma 3 – 4)

These queries/lemmas are mainly for the *SPSP* and *SPSBP* algorithm which heavily uses the dual $D(G)$ as we will later see in section 2.2.

Lemma 3: *Let T be a rooted tree of n nodes. We can preprocess T in linear time such that each of the following queries can be answered in constant time:*

- i) Given nodes u and v of T , report their lowest common ancestor, denoted as $lca(u, v)$
- ii) Given nodes u and v of T , decide whether u is in the subtree rooted at v
- iii) (Second-node Query) Given distinct nodes u and v of T , report the second node on the path from u to v in T

i) is a static data structural problem and follows from the work of Bender and Farach-Colton [?] where after a reduction to a specific range-minimum-query (RMQ) problem and a linear-time and space precomputation, $lca(u, v)$ can be looked up in constant time. *ii)* simply follows from the fact that u is in the subtree rooted at v if and only if $lca(u, v) = v$. *iii)* follows from an observation:

Algorithm 1 Second-node query

```

1: if  $v$  in subtree of  $left(u)$  then
2:   return  $left(u)$ 
3: if  $v$  in subtree of  $right(u)$  then
4:   return  $right(u)$ 
5: if  $u = v$  then
6:   return  $v$ 
7: else
8:   return  $parent(u)$ 

```

(courtesy of M. Smid)

Lemma 4: (Closest-color Query) *Given nodes u and v of tree T and a color c , such that u is on the path P_c , report the node on P_c that is closest to v . (refer to the paper for original details and proof)*

As part of the preprocessing step, for each color c and its path P_c in T , we store pointers to endpoints ccw_c and cw_c as well as $h_c = lca(ccw_c, cw_c)$ – the closest node to root of T .

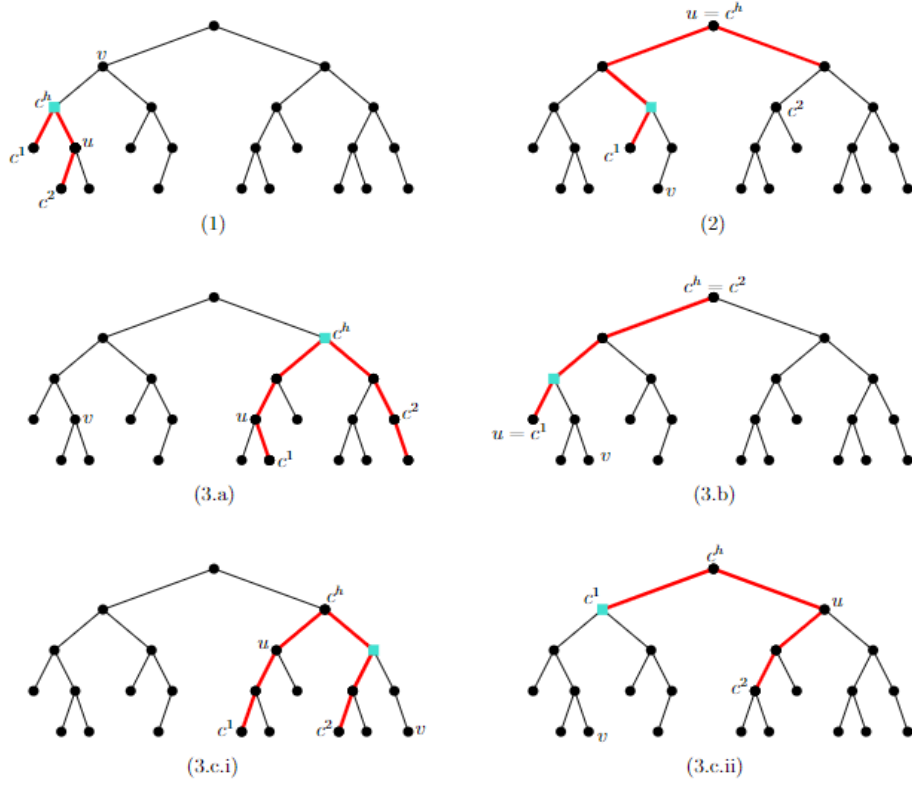
Algorithm 2 Closest-color query

```

1: if  $u$  is in subtree of  $v$  then return  $h_c$                                 ▷ Case 1
2:
3: if  $v$  is in the subtree of  $u$  then                                       ▷ Case 2
4:    $u' =$  child node of  $u$  where  $v$  is in subtree of  $u'$ 
5:    $w_1 = lca(v, ccw_c)$ 
6:    $w_2 = lca(v, cw_c)$ 
7:
8:   if  $u' = lca(v, w_1)$  then return  $w_1$ 
9:   if  $u' = lca(v, w_2)$  then return  $w_2$ 
10:
11: if  $u$  and  $v$  is in different subtrees of  $lca(u, v)$  then                ▷ Case 3
12:    $x = lca(u, v)$ 
13:    $y =$  child node of  $x$  where  $u$  is in subtree of  $y$ 
14:    $z =$  child node of  $x$  where  $v$  is in subtree of  $z$ 
15:
16:   if  $h_c$  is in subtree of  $y$  then return  $h_c$                             ▷ Case 3.a
17:
18:   if  $x$  is strictly in subtree of  $h_c$  then return  $x$                     ▷ Case 3.b
19:
20:   if  $x = h_c$  then                                                       ▷ Case 3.c
21:     if  $h_c = ccw_c$  or  $h_c = cw_c$  then return  $x$ 
22:
23:     if  $ccw_c$  is in subtree of  $z$  then return  $lca(v, ccw_c)$               ▷ Case 3.c.i
24:
25:     if  $cw_c$  is in subtree of  $z$  then return  $lca(v, cw_c)$               ▷ Case 3.c.ii

```

(courtesy of M. Smid)



1.2 Beer Distance (Lemma 7)

A crucial part of finding the shortest beer path is, of course, having already computed beer weight/distance for each edge in G . The full result is stated in the following lemma:

Lemma 7: After a linear-time computation,

1. we will obtain $dist_B(u, u) \forall u \in V$, $dist_B(v, v) \forall v \in V$ and $dist_B(u, v) \forall (u, v) \in E$ of G .
2. for any queried edge $(u, v) \in E$ of G , $SP_B(u, v)$ can be reported in $O(L)$ time where L is the number of vertices on the path.
3. for any queried vertex $u \in V$ of G , $SP_B(u, u)$ can be reported in $O(L)$ time where L is the number of vertices on the path.

The first claim follows from an observation where for each edge $(u, v) \in E$ of G :

1. $dist_B(u, v) = \min(dist_B(u, v, G_{uv}^R), dist_B(u, v, G_{uv}^{\neg R}))$
2. $dist_B(u, u) = \min(dist_B(u, u, G_{uv}^R), dist_B(u, u, G_{uv}^{\neg R}))$

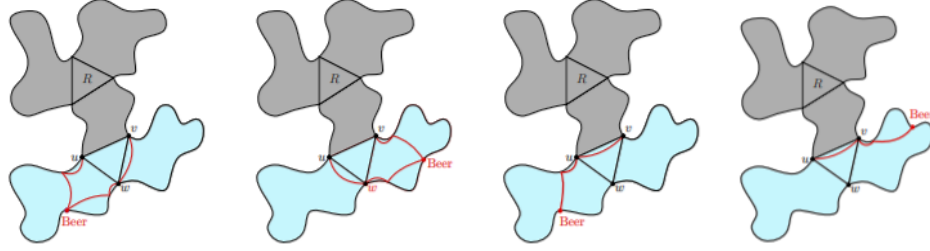
, likewise, with $dist_B(v, v)$ by replacing u with v

G_{uv}^R is a subgraph containing edge (u, v) and the face, or rather, the root R of $D(G)$; and $G_{uv}^{\neg R}$ is the other subgraph that doesn't contain this 'root' face R .

1. Compute $dist_B(u, v, G_{uv}^{\neg R})$, $dist_B(u, u, G_{uv}^{\neg R})$ and $dist_B(v, v, G_{uv}^{\neg R})$

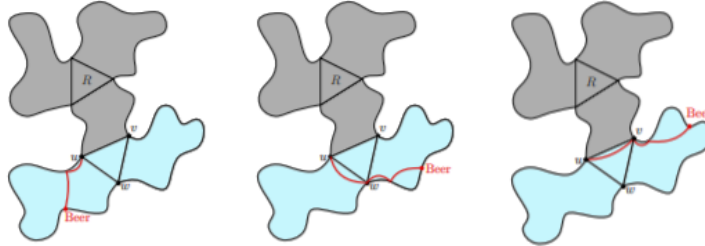
$dist_B(u, v, G_{uv}^{\neg R})$ is computed via the following recurrence:

$$(R1) \quad dist_B(u, v, G_{uv}^{\neg R}) = \min \begin{cases} dist_B(u, w, G_{uw}^{\neg R}) + weight(w, v) \\ weight(u, w) + dist_B(w, v, G_{vw}^{\neg R}) \\ dist_B(u, u, G_{uw}^{\neg R}) + weight(u, v) \\ weight(u, v) + dist_B(v, v, G_{vw}^{\neg R}) \end{cases}$$



Likewise, with $dist_B(u, u, G_{uv}^{\neg R})$ (and $dist_B(v, v, G_{uv}^{\neg R})$ by swapping u and v):

$$(R2) \quad dist_B(u, u, G_{uv}^{\neg R}) = \min \begin{cases} dist_B(u, u, G_{uw}^{\neg R}) \\ 2 \cdot weight(u, w) + dist_B(w, w, G_{vw}^{\neg R}) \\ 2 \cdot weight(u, v) + dist_B(v, v, G_{vw}^{\neg R}) \end{cases}$$



For each vertex v and each edge (u, v) in G , we maintain a tentative beer weight attribute δ_B and a tentative predecessor attribute ω_B .

ω_B is a tuple of 2 elements, the first is a vertex w that tentative beer path of edge (u, v) (or of vertex u back to itself) passes through; and the second is a bit 0 if a beer store is visited before passing through w (somewhere in the tentative beer path from u to w) or bit 1 if a beer store is visited after passing w (somewhere in the tentative beer path from w to v). Otherwise, set ω_B to NIL .

Looking at the recurrences and thinking in terms of tree $D(G)$, recursing on subgraph $G^{\neg R}$ means we are “going away” from root R of $D(G)$, i.e. recursively traverse the subtrees before processing the root node (or the face of G) and obtaining the δ_B and ω_B values. Thus, the values of those recurrence terms in $R1$ and $R2$ are “deeper in” the tree $D(G)$. By doing the computation of $R1$ and $R2$ in a post-order fashion during the traversal of $D(G)$, in $O(n)$ time we will obtain the results:

Algorithm 4 Post-order Traversal Beer Distance Computation

```
1: procedure BEERDISTNOTROOT( $u, v, F$ )
2:   if  $F = \text{NULL}$  then return
3:
4:   BEERDISTNOTROOT( $u', v', \text{left}(F)$ )           $\triangleright (u', v') \neq (u, v)$  shared by  $F$  and  $\text{left}(F)$ 
5:   BEERDISTNOTROOT( $u', v', \text{right}(F)$ )          $\triangleright (u', v') \neq (u, v)$  shared by  $F$  and  $\text{right}(F)$ 
6:
7:    $w = \text{vertex} \neq u \neq v$  in  $F$ 
8:    $\delta_B(u, v) = \min(\text{weight}(u, v) + \delta_B(u, u), \text{weight}(u, v) + \delta_B(v, v),$ 
9:                      $\text{weight}(u, w) + \delta_B(w, v), \delta_B(u, w) + \text{weight}(w, v))$ 
10:  if  $\delta_B(u, v) = \text{weight}(u, v) + \delta_B(u, u)$  then
11:     $\omega_B(u, v) = (u, 0)$ 
12:  else if  $\delta_B(u, v) = \text{weight}(u, v) + \delta_B(v, v)$  then
13:     $\omega_B(u, v) = (v, 1)$ 
14:  else if  $\delta_B(u, v) = \text{weight}(u, w) + \delta_B(w, v)$  then
15:     $\omega_B(u, v) = (w, 1)$ 
16:  else
17:     $\omega_B(u, v) = (w, 0)$ 
18:
19:  for all  $v \in F$  do
20:     $u, w = \text{distinct vertices} \neq v$  in  $F$ 
21:     $\delta_B(v, v) = \min(\delta_B(v, v), 2 \cdot \text{weight}(v, u) + \delta_B(u, u), 2 \cdot \text{weight}(v, w) + \delta_B(w, w))$ 
22:    if  $\delta_B(v, v) = 2 \cdot \text{weight}(v, u) + \delta_B(u, u)$  then
23:       $\omega_B(v, v) = (u, 0)$ 
24:    else if  $\delta_B(v, v) = 2 \cdot \text{weight}(v, w) + \delta_B(w, w)$  then
25:       $\omega_B(v, v) = (w, 0)$ 
```

(courtesy of J. Bacic)

Line 7: (in reference to R1)

At each recursive call of $\text{BeerDistNotRoot}(u, v, F)$, we are computing the tentative beer weight $\delta_B(u, v)$, specifically $\text{dist}_B(u, v, G_{uv}^{-R})$, for edge $(u, v) \in E$ shared by face $F (\in G_{uv}^{-R})$ and $\text{parent}(F) (\in G_{uv}^R)$. Furthermore, because we are doing post-order traversal, by the time we recursively return back to function call $\text{BeerDistNotRoot}(u, v, F)$ after previous recursive calls in Line 3-4, $\delta_B(w, v)$ and $\delta_B(u, w)$ will have been computed (specifically $\text{dist}_B(w, v, G_{vw}^{-R})$ with previous recursive call to $\text{BeerDistNotRoot}(v, w, \text{right}(F))$ and $\text{dist}_B(u, w, G_{uw}^{-R})$ with $\text{BeerDistNotRoot}(u, w, \text{left}(F))$).

Similarly, the value of $\delta_B(u, u)$ and $\delta_B(v, v)$, or rather $\text{dist}_B(u, u, G_{uw}^{-R})$ and $\text{dist}_B(v, v, G_{vw}^{-R})$, will also have been computed after previous recursive calls in Line 3-4 return ($\text{BeerDistNotRoot}(u, w, \text{left}(F))$ and $\text{BeerDistNotRoot}(v, w, \text{right}(F))$), respectively)

Line 20: (in reference to R2)

Again, because of the computation done in post-order fashion and after Line 3-4 return, $\delta_B(w, w)$ and $\delta_B(u, u)$ will have been computed (or rather $\text{dist}_B(w, w, G_{uw}^{-R})$ and $\text{dist}_B(u, u, G_{uw}^{-R})$). Therefore, we are able to compute $\delta_B(v, v) \forall v \in F$, or rather $\text{dist}_B(v, v, G_{vw}^{-R})$.

As we recurse on smaller subgraph, in this case it's subgraph G^{-R} where we are going away from the "root" face R of G , eventually we will reach the external edges of G . Additionally, because it is post-order traversal, δ_B and ω_B of external edges will be, in fact, the first to be computed, and subsequent computation for internal edges and vertices will depend on these values. Therefore, we handle the following base case:

Algorithm 3 Base Case Post-order Traversal Beer Distance Computation

```

1: for all external edges  $(u, v) \in E$  do
2:   if  $u$  is beer store then
3:      $\delta_B(u, v) = \text{weight}(u, v)$ 
4:      $\omega_B(u, v) = (u, 0)$ 
5:
6:   else if  $v$  is beer store then
7:      $\delta_B(u, v) = \text{weight}(u, v)$ 
8:      $\omega_B(u, v) = (v, 1)$ 
9:
10:  else
11:     $\delta_B(u, v) = \infty$ 
12:
13: for all  $v \in V$  do
14:   if  $v$  is beer store then
15:      $\delta_B(v, v) = 0$ 
16:      $\omega_B(v, v) = \text{NIL}$ 
17:
18:   else
19:      $\delta_B(v, v) = \infty$ 
20:

```

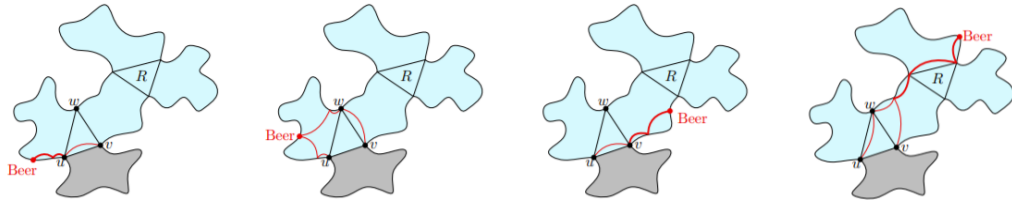
(courtesy of J. Bacic)

Note that the base case acts as a step of initialization and because of the aforementioned reason regarding the nature of post-order traversal, we would like to handle the base case first, i.e. run Algorithm 3 before Algorithm 4. Otherwise, δ_B and ω_B would still be ∞ and *NIL* after everything.

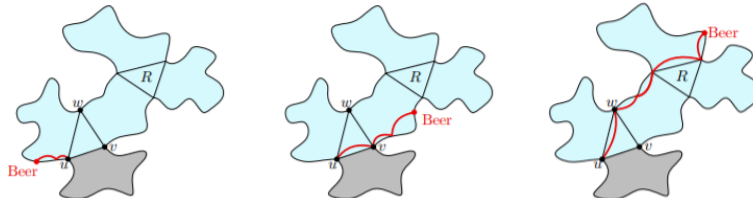
2. Compute $\text{dist}_B(u, v, G_{uv}^R)$, $\text{dist}_B(u, u, G_{uv}^R)$ and $\text{dist}_B(v, v, G_{uv}^R)$

Similar to part 1. above, we have the following recurrence:

$$(R3) \quad \text{dist}_B(u, v, G_{uv}^R) = \min \begin{cases} \text{dist}_B(u, u, G_{uw}^R) + \text{weight}(u, v) \\ \text{dist}_B(u, w, G_{uw}^R) + \text{weight}(w, v) \\ \text{weight}(u, v) + \text{dist}_B(v, v, G_{vw}^R) \\ \text{weight}(u, w) + \text{dist}_B(w, v, G_{vw}^R) \end{cases}$$



$$(R4) \quad \text{dist}_B(u, u, G_{uv}^R) = \min \begin{cases} \text{dist}_B(u, u, G_{uw}^R) \\ 2 \cdot \text{weight}(u, v) + \text{dist}_B(v, v, G_{vw}^R) \\ 2 \cdot \text{weight}(u, w) + \text{dist}_B(w, w, G_{vw}^R) \end{cases}$$



We maintain additional attributes, namely the beer weight d_B and predecessor π_B . Their use is the same as δ_B and ω_B , respectively. However, *roughly speaking*, δ_B attribute corresponds to values of subgraph $G^{\neg R}$ whereas d_B attribute corresponds to values of remaining subgraph G^R .

After Algorithm 3-4, $\delta_B(u, v)$ is computed relative to subgraph $G_{uv}^{\neg R} \forall (u, v) \in E$ of G , meaning the current beer path and distance computed for edge (u, v) is within subgraph $G_{uv}^{\neg R}$ only. And, so is the case for $\delta_B(u, u) \forall u \in V$. Now, we want to compute $d_B \forall v \in V$ and $(u, v) \in E$.

Since we did a post-order traversal of $D(G)$ for the computation on $G^{\neg R}$, it's intuitive that we now do the computation in the opposite fashion, i.e. a pre-order traversal for the computation on G^R . Furthermore, recursing on subgraph G^R means we are “approaching” the face R , and in terms of traversal on a rooted tree $D(G)$ at face R , that means processing the root node first and obtain the values before traversing the subtrees. As such, $R3$ and $R4$ are possible since the recurrence terms involving subgraph G^R are “higher up” in the tree $D(G)$ and will have been computed, and the values of terms involving subgraph $G^{\neg R}$ are, of course, already computed from previous post-order traversal in Algorithm 3-4. We have the following:

Algorithm 6 Pre-order Traversal Beer Distance Computation

```

1: procedure BEERDISTR00T( $u, v, F$ )
2:    $w = \text{vertex} \neq u \neq v$  in  $F$ 
3:
4:    $d_B(u, w) = \min(\delta_B(u, w), \text{weight}(u, w) + \delta_B(u, u), \text{weight}(u, w) + d_B(w, w),$ 
5:                   $\delta_B(u, v) + \text{weight}(v, w), \text{weight}(u, v) + d_B(v, w))$ 
6:   if  $d_B(u, w) = \delta_B(u, w)$  then
7:      $\pi_B(u, w) = \omega_B(u, w)$ 
8:   else if  $d_B(u, w) = \text{weight}(u, w) + \delta_B(u, u)$  then
9:      $\pi_B(u, w) = (u, 0)$ 
10:  else if  $d_B(u, w) = \text{weight}(u, w) + d_B(w, w)$  then
11:     $\pi_B(u, w) = (w, 1)$ 
12:  else if  $d_B(u, w) = \delta_B(u, v) + \text{weight}(v, w)$  then
13:     $\pi_B(u, w) = (v, 0)$ 
14:  else
15:     $\pi_B(u, w) = (v, 1)$ 
16:
17:   $d_B(v, w) = \min(\delta_B(v, w), \text{weight}(v, w) + \delta_B(v, v), \text{weight}(v, w) + d_B(w, w),$ 
18:                   $\delta_B(v, u) + \text{weight}(u, w), \text{weight}(v, u) + d_B(u, w))$ 
19:  if  $d_B(v, w) = \delta_B(v, w)$  then
20:     $\pi_B(v, w) = \omega_B(v, w)$ 
21:  else if  $d_B(v, w) = \text{weight}(v, w) + \delta_B(v, v)$  then
22:     $\pi_B(v, w) = (v, 0)$ 
23:  else if  $d_B(v, w) = \text{weight}(v, w) + d_B(w, w)$  then
24:     $\pi_B(v, w) = (w, 1)$ 
25:  else if  $d_B(v, w) = \delta_B(v, u) + \text{weight}(u, w)$  then
26:     $\pi_B(v, w) = (u, 0)$ 
27:  else
28:     $\pi_B(v, w) = (u, 1)$ 
29:
30:   $d_B(w, w) = \min(\delta_B(w, w), 2 \cdot \text{weight}(w, u) + d_B(u, u), 2 \cdot \text{weight}(w, v) + d_B(v, v))$ 
31:  if  $d_B(w, w) = \delta_B(w, w)$  then
32:     $\pi_B(w, w) = \omega_B(w, w)$ 
33:  else if  $d_B(w, w) = 2 \cdot \text{weight}(w, u) + d_B(u, u)$  then
34:     $\pi_B(w, w) = (u, 0)$ 
35:  else
36:     $\pi_B(w, w) = (v, 0)$ 
37:
38:  BEERDISTR00T( $u', v', \text{left}(F)$ )  $\triangleright (u', v') \neq (u, v)$  shared by  $\text{left}(F)$  and  $F$ 
39:  BEERDISTR00T( $u', v', \text{right}(F)$ )  $\triangleright (u', v') \neq (u, v)$  shared by  $\text{right}(F)$  and  $F$ 

```

Lines 4 & 17 are in reference to R3.

Line 30 is in reference to R4.

At each recursive call $\text{BeerDistRoot}(u', v', \text{child}(F))$ at Lines 38 - 39 where edge (u', v') is shared by face $\text{child}(F)$ ($\in G_{uv'}^R$) and F ($\in G_{uv'}^R$), the face F will have been processed already at Lines 1-37 in previous recursive call to $\text{BeerDistRoot}(u, v, F)$. Values of d_B and π_B will have been computed for all vertex v and internal edge $(u, v) \in F$.

As such, at this current face, only the vertex w as well as edges (u, w) and (v, w) are yet to be processed. Note that for each edge $e \in E$, when we compute d_B , we deal with recurrence R3 but we also take the minimum of R3 with $\delta_B(e)$. Recall that R3 is to compute $\text{dist}_B(e, G_e^R)$ and $\delta_B(e) = \text{dist}_B(e, G_e^{\neg R})$ (R2) has already been computed during post-order traversal. As the result, by the end of Algorithm 6, $d_B(e) = \text{dist}_B(e)$ is the true shortest beer weight $\forall e \in E$.

As explained above, because of pre-order traversal, computation on a root node/face of G will be performed first and subsequent computation done on other faces while traversing the subtrees of $D(G)$ depends on these already computed values from the parent faces. In fact, the “root” face R will be processed first and this is our base case where v and $(u, v) \in R$:

$$(R5) \quad \text{dist}_B(u, v, G_{uv}^R) = \min \begin{cases} \text{dist}_B(u, u, G_{uw}^{\neg R}) + \text{weight}(u, v) \\ \text{dist}_B(u, w, G_{uw}^{\neg R}) + \text{weight}(w, v) \\ \text{weight}(u, v) + \text{dist}_B(v, v, G_{vw}^{\neg R}) \\ \text{weight}(u, w) + \text{dist}_B(w, v, G_{uv}^{\neg R}) \end{cases}$$

$$(R6) \quad \text{dist}_B(u, u, G_{uv}^R) = \min \begin{cases} \text{dist}_B(u, u, G_{uw}^{\neg R}) \\ 2 \cdot \text{weight}(u, w) + \text{dist}_B(w, w, G_{vw}^{\neg R}) \\ 2 \cdot \text{weight}(u, v) + \text{dist}_B(v, v, G_{vw}^{\neg R}) \end{cases}$$

Algorithm 5 Base Case Pre-order Traversal Beer Distance Computation

```

1: for all  $v \in R$  do
2:    $u, w = \text{unique vertices } \neq v \in R$ 
3:    $d_B(v, v) = \min(\delta_B(v, v), 2 \cdot \text{weight}(v, u) + \delta_B(u, u), 2 \cdot \text{weight}(v, w) + \delta_B(w, w))$ 
4:   if  $d_B(v, v) = \delta_B(v, v)$  then
5:      $\pi_B(v, v) = \omega_B(v, v)$ 
6:   else if  $d_B(v, v) = 2 \cdot \text{weight}(v, u) + \delta_B(u, u)$  then
7:      $\pi_B(v, v) = \omega_B(u, 0)$ 
8:   else
9:      $\pi_B(v, v) = \omega_B(w, 0)$ 
10:
11: for all internal edge  $(u, v) \in R$  do
12:    $w = \text{vertex } \neq u \neq v \in R$ 
13:    $d_B(u, v) = \min(\delta_B(u, v), \text{weight}(u, v) + \delta_B(u, u), \text{weight}(u, v) + \delta_B(v, v),$ 
14:      $\delta_B(u, w) + \text{weight}(w, v), \text{weight}(u, w) + \delta_B(w, v))$ 
15:   if  $d_B(u, v) = \delta_B(u, v)$  then
16:      $\pi_B(u, v) = \omega_B(u, v)$ 
17:   else if  $d_B(u, v) = \text{weight}(u, v) + \delta_B(u, u)$  then
18:      $\pi_B(u, v) = \omega_B(u, 0)$ 
19:   else if  $d_B(u, v) = \text{weight}(u, v) + \delta_B(v, v)$  then
20:      $\pi_B(u, v) = \omega_B(v, 1)$ 
21:   else if  $d_B(u, v) = \delta_B(u, w) + \text{weight}(w, v)$  then
22:      $\pi_B(u, v) = \omega_B(w, 0)$ 
23:   else
24:      $\pi_B(u, v) = \omega_B(w, 1)$ 

```

(courtesy of J. Bacic)

As always, we want to handle the base case first, i.e. running Algorithm 5 before Algorithm 6.

The second and third claims of Lemma 7 are very simple once we obtained predecessor attribute $\pi_B \forall v \in V$ and $(u, v) \in E$:

Algorithm 7 Print beer path

```

1: procedure PRINTBEERPATH( $u, v$ )
2:   return  $\{u\} \cup \text{PRINTBEERSUBPATH}(u, v)$ 

```

(courtesy of J. Bacic)

Algorithm 8 Print beer subpath

```

1: procedure PRINTBEERSUBPATH( $u, v$ )
2:   if  $u$  is beer store ||  $v$  is beer store ||  $\pi_B(u, v) = \text{NIL}$  then
3:     return  $\{u\}$ 
4:
5:    $(w, \text{beerLoc}) = \pi_B(u, v)$ 
6:   if  $\text{beerLoc} = 0$  then
7:     return  $\text{PRINTBEERSUBPATH}(u, w) \cup \{v\}$ 
8:   if  $\text{beerLoc} = 1$  then
9:     return  $\{u, w\} \cup \text{PRINTBEERSUBPATH}(w, v)$ 
10:

```

(courtesy of J. Bacic)

1.3: Report weight and instance of shortest path on the v -chain

Refer to the Lemma 12, 13 and 14, the result can be summed up as follows:

After an $O(n)$ -time preprocessing, we can answer the following queries, for any three vertices v, u and w , where u and $w \in G[P_v]$:

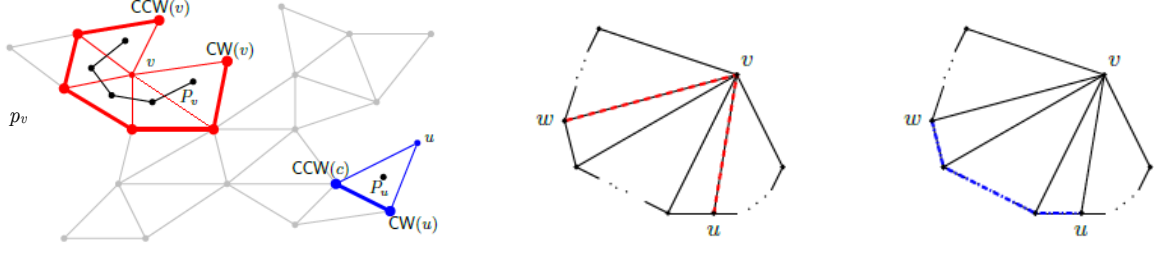
1. Report $\text{dist}(u, w)$, i.e. shortest distance between u & w in G in $O(1)$ time.
2. Report $SP(u, w)$ in $O(L)$ time where L is number of vertices on the path.

Algorithm 9 Report $\text{dist}(u, w, G[P_v])$

```

1: if  $u = w$  then
2:    $\text{dist}(u, w, G[P_v]) = 0$ 
3:    $\pi(u, w) = \text{NIL}$ 
4:
5: else if  $u = v$  ||  $w = v$  then
6:    $\text{dist}(u, w, G[P_v]) = \text{weight}(u, w)$ 
7:    $\pi(u, w) = \text{NIL}$ 
8:
9: else
10:   During preprocessing step, for any apex vertex  $v$ , store an array of size  $N$  and for any
       vertex  $u$  on apex's chain  $p_v$ , store the weight of path from  $u$  to  $CW(p_v)$  along  $p_v$ . As such,
11:    $\text{dist}(u, w, p_v) = |\text{dist}(u, CW(p_v), p_v) - \text{dist}(w, CW(p_v), p_v)|$ 
12:
13:    $\text{dist}(u, w, G[P_v]) = \min(\text{dist}(u, w, p_v), \text{weight}(u, v) + \text{weight}(v, w))$ 
14:   if  $\text{dist}(u, w, G[P_v]) = \text{dist}(u, w, p_v)$  then
15:      $\pi(u, w) = (v, 1)$ 
16:   else
17:      $\pi(u, w) = (v, 0)$ 

```



The algorithm is a simple case-by-case analysis and follows from the observation as shown in the figures above.

By now, the pseudocode notation is very familiar. While computing the distance, we also encode the information so that we can later reconstruct the shortest path easily. If u and w are just endpoints of an edge, then $\pi(u, w) = NIL$ since the shortest path is simply $\{u, w\}$. Otherwise, store the apex vertex v and one additional bit 1 if the shortest path is on the chain p_v or bit 0 if the path crosses apex v instead. From there, constructing $SP(u, w)$ is quite trivial.

1.4: Report beer weight and instance of shortest beer path on the v -chain

Lemma 15: After $O(n)$ -time preprocessing, we can report, for any three vertices v , u and w where $u, w \in G[P_v]$, beer distance $dist_B(u, w)$ in $O(1)$ time and the corresponding $SP_B(u, w)$ in $O(L)$ time.

Algorithm 10 Report $dist_B(u, w, G[P_v])$

```

1: if  $u = w$  then
2:    $dist_B(u, w, G[P_v]) = dist_B(u, u)$ 
3:    $\pi_B(u, w) = NIL$ 
4:
5: else if  $u = v \parallel w = v$  then
6:    $dist_B(u, w, G[P_v]) = weight_B(u, w)$ 
7:    $\pi_B(u, w) = NIL$ 
8:
9: else
10:   During preprocessing step, for any apex vertex  $v$ , store an array  $A_v$  of size  $N - 1$ .
11:   Let  $p_v = (u_1, u_2, \dots, u_N)$ 
12:    $A_v[i] = dist_B(u_i, u_{i+1}) - weight(u_i, u_{i+1})$  for  $i = [1, N - 1]$ 
13:   tree  $C = \text{BUILD CARTESIAN TREE}(A_v)$ 
14:    $\text{PRECOMPUTE LCA}(C)$ 
15:
16:    $dist_B(u, w, G[P_v]) = \min(weight_B(u, v) + weight(v, w), weight(u, v) + weight_B(v, w),$ 
17:                                $dist(u, w, p_v) + A_v[i]) \triangleright A_v[i] = \min(A_v[j], \dots, A_v[k - 1])$ 
18:                                $= \text{LCA}(A[j], A[k - 1])$ 
19:   if  $dist_B(u, w, G[P_v]) = weight_B(u, v) + weight(v, w)$  then
20:      $\pi_B(u, w) = (v, -1)$ 
21:   else if  $dist_B(u, w, G[P_v]) = weight(u, v) + weight_B(v, w)$  then
22:      $\pi_B(u, w) = (v, -2)$ 
23:   else
24:      $\pi_B(u, w) = (v, i)$ 

```

Note that by now, we have computed $dist_B(u, v) \forall (u, v) \in E$ and $dist_B(u, u) \forall u \in V$ of G . The first 2 cases in the pseudocode are trivial and are handled similarly to Algorithm 9 above. The last case, however, is more interesting where u, w and v are distinct vertices in $G[P_v]$.

Lines 10 - 14 are part of the preprocessing step where we maintain an additional array A_v (similar to the last case in Algorithm 9) and preprocess it in $O(n)$ time to answer the RMQs in $O(1)$ time. [?] shows that we can reduce the problem to LCA-queries in Cartesian tree of the array. We already know how to answer LCA-queries in $O(1)$ after linear precomputation time (from Lemma 3.i)), and the following algorithm shows how we can build the Cartesian tree in $O(n)$ time and space:

Algorithm 11 Build Cartesian tree from a given array

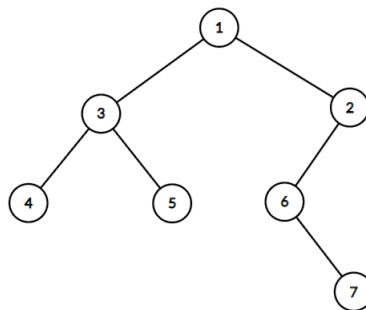
```

1: procedure BUILD_CARTESIAN_TREE(array  $A[N]$ )
2:    $LNSV[ ]$  = new array holding all nearest smaller values from the left of  $A$ 
3:    $stack\ S = [null]$ 
4:   for all index  $i$  from 0 to  $N - 1$  do
5:     while  $!S.empty()$  and  $S.top() \geq A[i]$  do
6:        $S.pop()$ 
7:     if  $S.empty()$  then
8:        $LNSV[i] = null$ 
9:     else
10:       $LNSV[i] = S.top()$ 
11:     $S.push(A[i])$ 
12:
13:    $RNSV[ ]$  = new array holding all nearest smaller values from the right of  $A$ 
14:    $S = [null]$ 
15:   for all index  $i$  from  $N - 1$  to 0 do
16:     Repeat the same procedure as above but for  $RNSV[ ]$ 
17:
18:   for all index  $i$  from 0 to  $N - 1$  do
19:     if  $LNSV[i] = RNSV[i] = null$  then
20:       Cartesian root node =  $A[i]$ 
21:     else if  $RNSV[i] > LNSV[i]$  then
22:        $A[i]$  = left child of  $RNSV[i]$ 
23:     else
24:        $A[i]$  = right child of  $LNSV[i]$ 
25:
26:   return Cartesian root node

```

We reduce the problem of building Cartesian tree to a much simpler problem of finding the nearest smaller value for each element in an array from left to right and vice versa (formerly known as all nearest smaller values problem).

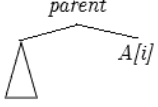
$A[]$	4	3	5	1	6	7	2	
$LNSV[]$	<i>null</i>	<i>null</i>	3	<i>null</i>	1	6	1	
	3	1	1	<i>null</i>	2	2	<i>null</i>	$RNSV[]$



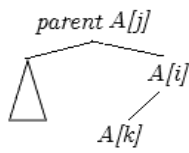
Naively, we can compute the nearest smaller value of all elements in the array by having nested loops, which results in $O(n^2)$ time. By maintaining only the values of A that have been processed so far (Line 11) and are smaller than later values in A (Lines 5-6) in most recent order, we then have the aforementioned stack-based algorithm. Note that each element in A is pushed onto stack exactly once and is pop off the stack at most once, therefore, all nearest smaller values of A can be computed in linear time and space.

We now proof why the reduction from building Cartesian tree to all nearest smaller values from both sides of array A works via case-by-case analysis and contradiction:

1. We first prove that if $A[i]$ is the right child of node *parent*, then *parent* = $LNSV$ of $A[i]$ (denoted as $LNSV_{A[i]}$) and also, if $A[i]$ is the left child of node *parent*, then *parent* = $RNSV$ of $A[i]$ (denoted as $RNSV_{A[i]}$)
2. We then prove if $LNSV_{A[i]} > RNSV_{A[i]}$, then the node *parent* of $A[i]$ is $LNSV_{A[i]}$. Otherwise, if $RNSV_{A[i]} > LNSV_{A[i]}$, then node *parent* of $A[i]$ is $RNSV_{A[i]}$.

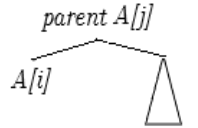
1. Suppose we have Cartesian tree where $A[i]$ is the right child of node *parent* -

, then *parent* must be $A[j]$ for some index $j < i$ (because of inorder traversal). Note that Cartesian tree, in our case, is a min-heap, and therefore $A[j] < A[i]$. As such, *parent* = $A[j]$ = $LNSV_{A[i]}$

But suppose $\exists k$ such that $j < k < i$ and $A[k] < A[i]$. That is, $A[k]$ = $LNSV_{A[i]}$ instead of $A[j]$ (note that $A[j]$ is still the parent of $A[i]$ but it is now supposed that $A[j] \neq LNSV_{A[i]}$). Since $j < k < i$, we have some subsequence $(A[j], \dots, A[k], \dots, A[i])$ in A , and the Cartesian tree must be of the structure:



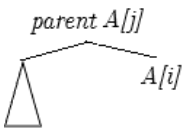
Due to min-heap property, it must be that $A[i] < A[k]$ which is a contradiction. Therefore, if $A[i]$ is right child of its parent $A[j]$, then it must be the case that $A[j] = LNSV_{A[i]}$.

Similarly, by proof of contradiction, if $\forall i < j$, $A[j]$ is the parent of $A[i]$ (i.e. $A[i]$ is the left child of its parent $A[j]$), then it must be that $A[j] = RNSV_{A[i]}$.



2. If $LNSV_{A[i]} \neq null$ or $RNSV_{A[i]} \neq null$ for an entry $A[i]$, then $A[i]$ is not the root since there is some smaller value than $A[i]$ that should be the parent of $A[i]$ (according to min-heap property)

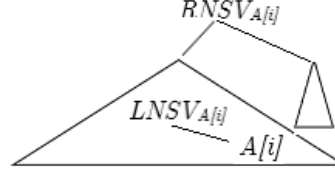
Suppose now that $A[i]$ is the right child of its parent $A[j]$, i.e. $A[i]$'s *parent* node is $LNSV_{A[i]}$.



We must have some subsequence $(LNSV_{A[i]}, \dots, A[i], \dots, RNSV_{A[i]})$ in array A . Since $RNSV_{A[i]} < A[i]$, $RNSV_{A[i]}$ is not in the subtree of $A[i]$ (min-heap property).

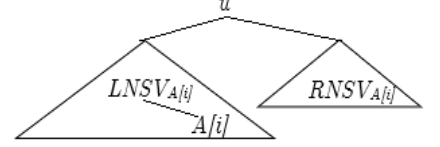
Let $u = lca(LNSV_{A[i]}, RNSV_{A[i]})$.

If $u = RNSV_{A[i]}$, then $LNSV_{A[i]}$ is in subtree of $RNSV_{A[i]}$ and due to in-order traversal, $LNSV_{A[i]}$ must specifically be in left subtree of $RNSV_{A[i]}$. Again, due to min-heap property, $A[j] = LNSV_{A[i]} > RNSV_{A[i]}$



Otherwise, $LNSV_{A[i]}$ and $RNSV_{A[i]}$ are in two different subtrees. Because of in-order traversal, it must be the case that $RNSV_{A[i]}$ is in right subtree of u and $LNSV_{A[i]}$ is in left subtree of u .

Thus, $u < LNSV_{A[i]} < A[i]$ and $u < RNSV_{A[i]}$. This is impossible since $LNSV_{A[i]}$ and $RNSV_{A[i]}$ are already the nearest values that are smaller than $A[i]$ and there can't be any other "better" values. Hence, this is a contradiction and therefore it must be the case that if $LNSV_{A[i]} > RNSV_{A[i]}$, then $A[j] = LNSV_{A[i]}$ is the parent of $A[i]$.



Similarly, if $RNSV_{A[i]} > LNSV_{A[i]}$, then $A[j] = RNSV_{A[i]}$ is the parent of $A[i]$.

2. Answer Shortest Beer Path Queries

2.1 Single-Pair Shortest (Beer) Path - SPSP and SPSBP

By now, we should have completed the entire $O(n)$ -time preprocessing step in the above section I.1.

Given two vertices s and t of G , we want to compute $SP(s, t)$ and $SP_B(s, t)$ and of course, their distance as well, in $O(L)$ time. There are edge cases that are trivial and can be handled separately:

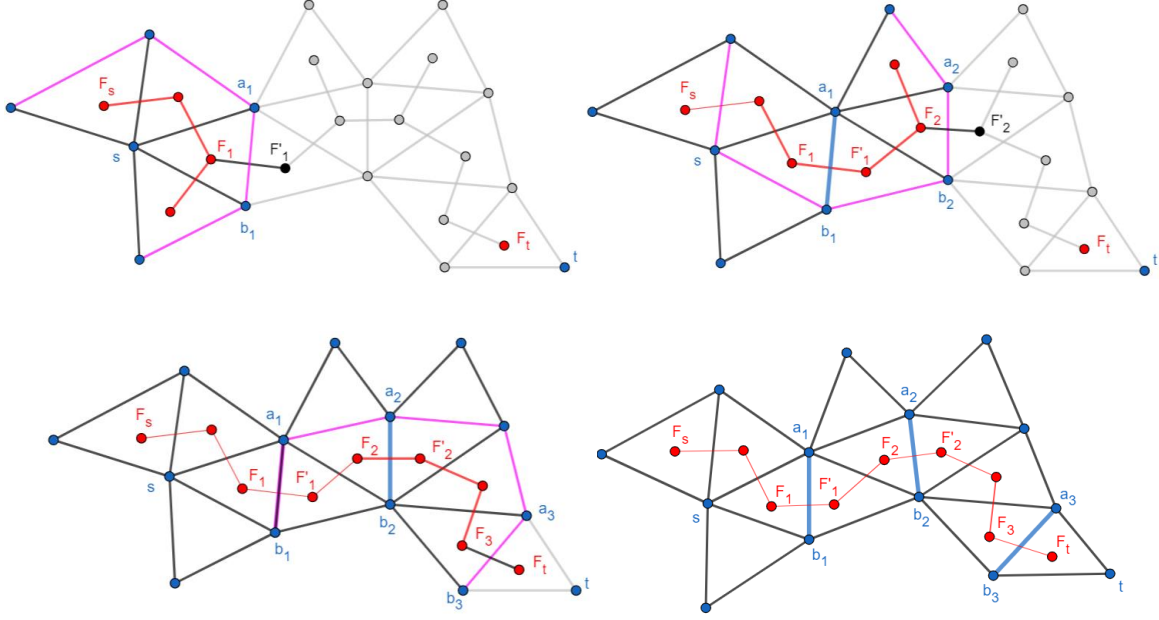
Algorithm 12 Edge Cases of SPSP and SPSBP

```

1: if  $s = t$  then
2:    $SP(s, t) = \{s\}$ 
3:    $dist(s, t) = 0$ 
4:    $SP_B(s, t) = \text{PRINTBEERPATH}(s, t)$  ▷ Algorithm 7
5:    $dist_B(s, t)$  is already computed  $\forall v \in V$  of  $G$  ▷ During  $O(n)$  preprocessing step
6:
7: else if  $(s, t) \in E$  (i.e.  $t \in G[P_s]$ ) then
8:    $SP(s, t) = \{s, t\}$ 
9:    $dist(s, t) = weight(s, t)$ 
10:   $SP_B(s, t) = \text{PRINTBEERPATH}(s, t)$  ▷ Algorithm 7
11:   $dist_B(s, t)$  is already computed  $\forall (u, v) \in E$  of  $G$  ▷ During  $O(n)$  preprocessing step

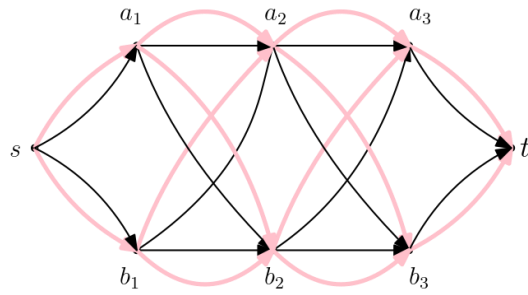
```

Otherwise, we build a directed-acyclic-graph (DAG), called H . In this DAG, vertices are arranged in column of constant size and edges go from left to right between the vertices in consecutive columns. The intuition for building the DAG is illustrated below (for formal details, refer to the original paper):



We use the dual $D(G)$ to navigate towards a face F_t containing the destination vertex t , starting from the face F_s containing the source vertex s . For each vertex $v \in V$, we have its v -chain (colored in magenta above) which contains the p_v path $\in G$ along the chain as well as P_v path $\in D(G)$. We use the Lemma 4 – Closest-color Query (Algorithm 2) and Second-node Query (Algorithm 1) on this P_v path in order to get to the correct face F_i and F'_i , respectively. Thus, we obtain the 2 vertices a_i and b_i (shared by F_i and F'_i) for the $(i + 1)^{th}$ DAG column.

We repeat the entire procedure again but for either vertex a_i or b_i , whichever vertex can get us furthest away, i.e. the next face F_{i+1} only contains either a_i or b_i . We terminate the procedure when we've reached some face F_t that contain the final vertex t , and obtain the following DAG:



Assuming WLOG that we are in the a_i -chain, note that a_{i+1} and $b_{i+1} \in G[P_{a_i}]$, and thus we can compute the edges from 1 column to the next consecutive column in DAG via Algorithm 9-10. Between each consecutive DAG columns, for each vertex u in the current i^{th} column, we have DAG *shortest path* edges (colored in black above) towards the next two vertices in the $(i + 1)^{th}$ column. These can be computed with Algorithm 9. Along with the DAG *shortest path* edge, we also add a corresponding DAG *shortest path beer* edge (colored in pink above) and it can also be computed via Algorithm 10.

The following recurrence computes the shortest path and the shortest beer path in G using Dynamic Programming approach:

$$dist(s, u) = \min \begin{cases} dist(s, a_{i-2}) + dist(a_{i-2}, u) \\ dist(s, b_{i-2}) + dist(b_{i-2}, u) \end{cases}$$

$$dist_B(s, u) = \min \begin{cases} dist(s, a_{i-2}) + dist_B(a_{i-2}, u) \\ dist_B(s, a_{i-2}) + dist(a_{i-2}, u) \\ dist(s, b_{i-2}) + dist_B(b_{i-2}, u) \\ dist_B(s, b_{i-2}) + dist(b_{i-2}, u) \end{cases}$$

Again, to reconstruct the actual instance of the shortest path, we use the predecessor attributes π and π_B and use the same idea of storing the information as with the beer distance computation.

2.2 Single-Source Shortest (Beer) Path - SSSP and SSSBP

For both SSSP and SSSBP problem, the idea is very simple as we simply perform a pre-order traversal. Below is the pseudo-code for computing SSSP:

```

1: procedure PRETRAVERSAL( $a, b, F$ )
2:    $c :=$  the vertex of  $F$  that is not  $a$  and not  $b$ 
3:    $d(c) := \min(d(a) + \text{dist}(a, c), d(b) + \text{dist}(b, c))$ 
4:   if  $d(c) = d(a) + \text{dist}(a, c)$  then
5:      $\pi(c) := a$ 
6:   else
7:      $\pi_B(c) := b$ 
8:   end if
9:   for all interior edges  $(u, v) \neq (a, b)$  in  $F$  do
10:     $F' :=$  the face that shares  $(u, v)$  with  $F$ 
11:    PRETRAVERSAL( $u, v, F'$ )
12:  end for
13: end procedure

```

(courtesy of J. Bacic)

```

1: procedure SSSP( $s, G$ )
2:    $F_s :=$  a face of  $G$  such that  $s \in F_s$ 
3:    $a, b :=$  the vertices of  $F_s$  that are not  $s$ 
4:    $d(s) := 0$ 
5:    $d(a) := \text{dist}(s, a)$ 
6:    $\pi(a) := s$ 
7:    $d(b) := \text{dist}(s, b)$ 
8:    $\pi(b) := s$ 
9:   for all interior edges  $(u, v)$  of  $F_s$  do
10:     $F :=$  the face that shares the edge  $(u, v)$  with  $F_s$ 
11:    PRETRAVERSAL( $u, v, F$ )
12:  end for
13: end procedure

```

(courtesy of J. Bacic)

Reconstructing the path is very simple for the SSSP problem since it is only the matter of following the attribute π until we reach the source vertex s .

For SSSBP problem, the algorithm is very much similar to the SSSP algorithm:

```

1: procedure PRETRAVERSALSBP( $a, b, F$ )
2:    $c :=$  the vertex of  $F$  that is not  $a$  and not  $b$ 
3:    $d(c) := \min(\text{dist}(s, a) + \text{dist}_B(a, c), \text{dist}(s, b) + \text{dist}_B(b, c), d_B(a) + \text{dist}(a, c), d_B(b) + \text{dist}(b, c))$ 
4:   if  $d(c) = \text{dist}(s, a) + \text{dist}_B(a, c)$  then
5:      $\pi_B(c) := (a, 1)$ 
6:   else if  $\text{dist}(s, b) + \text{dist}_B(b, c)$  then
7:      $\pi_B(c) := (b, 1)$ 
8:   else if  $d_B(a) + \text{dist}(a, c)$  then
9:      $\pi_B(c) := (a, 0)$ 
10:  else
11:     $\pi_B(c) := (b, 0)$ 
12:  end if
13:  for all interior edges  $(u, v) \neq (a, b)$  in  $F$  do
14:     $F' :=$  the face that shares  $(u, v)$  with  $F$ 
15:    PRETRAVERSAL( $u, v, F'$ )
16:  end for
17: end procedure

```

(courtesy of J. Bacic)

```

1: procedure SBP( $s, G$ )
2:    $F_s :=$  a face of  $G$  such that  $s \in F_s$ 
3:    $a, b :=$  the vertices of  $F_s$  that are not  $s$ 
4:    $d_B(s) := \text{dist}_B(s, s)$ 
5:    $\pi_B(s) := (s, 1)$ 
6:    $d_B(a) := \text{dist}_B(s, a)$ 
7:    $\pi_B(a) := (s, 1)$ 
8:    $d_B(b) := \text{dist}_B(s, b)$ 
9:    $\pi_B(b) := (s, 1)$ 
10:  for all interior edges  $(u, v)$  of  $F_s$  do
11:     $F :=$  the face that shares the edge  $(u, v)$  with  $F_s$ 
12:    PRETRAVERSALSBP( $u, v, F$ )
13:  end for
14: end procedure

```

(courtesy of J. Bacic)

```

1: procedure PRINTOUTERPLANARSINGLESOURCEBEERPATH( $v, w$ )
2:   return  $\{v\} \cup \text{PRINTOUTERPLANARSINGLESOURCEBEERSUBPATH}(v, w)$ 

```

(courtesy of J. Bacic)

```

1: procedure PRINTOUTERPLANARSINGLESOURCEBEERSUBPATH( $v$ )
2:    $(w, \text{beerPathLoc}) = \pi_B(v)$ 
3:
4:   if  $\text{beerPathLoc} = 1$  then
5:     PRINTBEERPATH( $v, w$ )
6:     PRINTOUTERPLANARSINGLESOURCESUBPATH( $w$ )
7:
8:   else
9:      $\{v, w\} \cup \text{PRINTOUTERPLANARSINGLESOURCEBEERSUBPATH}(w)$ 

```

(courtesy of J. Bacic)

3. Verification

For SSSP problem, the computed predecessor attribute π induces a shortest path tree. We can verify, if the implementation is correct by checking whether the ‘shortest path tree’ it computed is indeed a tree and contains the true shortest paths from source s to every other vertices, in $O(|V| + |E|)$ time with the following conditions:

1. $\pi(\pi(\dots\pi(v)\dots)) = s \quad \forall v \in V$
2. $d(v) \leq d(k) + \omega(k, v) \quad \forall k \in N_v, \forall v \in V$
3. $d(v) = d(\pi(v)) + \omega(\pi(v), v) \quad \forall v \in V$

1. The first condition is to verify if the computed shortest path tree is indeed a tree. By doing a DFS on the “tree”, we can detect if there is any cycle or disconnected component in $O(|V| + |E|)$ time. However, it is not easy to perform a DFS when each vertex only has one pointer to its parent. Fortunately, we can still verify in linear time by marking vertices we’ve visited so far that has a path to root s . We can move on at the moment new vertices are connected to one of those marked vertices. Additionally, we observe that when there is a loop at some vertex v , we will visit that vertex again in the same ‘round’. By also keeping track of which round a vertex was marked, we can identify at termination the vertex inside a loop:

Algorithm 19 Detect Loop

```

1: HASH MAP visited
2: count = 0
3:
4: visited[s] = count
5: count = count + 1
6: for all  $v \in V$  do
7:   while  $v \neq s$  and  $v \notin \text{visited}$  do
8:     visited[v] = count
9:      $v = \pi(v)$ 
10:
11:   if  $v \neq s$  and visited[v] = count then
12:     Cycle Detected...
13:     return False
14:
15:   count = count + 1

```

(courtesy of P.Bose and M.Smid)

2. The second condition is to verify whether we’ve computed the corrected shortest distance for every vertex, that is, by examining all neighbor k of vertex v , we check that the shortest distance $d(v)$ is indeed the minimum and cannot be reduced further. Since all the term in the inequality is already computed, checking this condition for all vertices takes $O(|V| + |E|)$ time.

3. The final condition is to verify if we've got the correct predecessor of vertex v that gives the correct distance $d(v)$, that is, $\pi(v)$ is some neighbor vertex k of v that minimizes the inequality in condition 2. Again, like condition 2, we can check this equality for all vertices in $O(|V| + |E|)$ time.

With the computed shortest path tree given to the verification algorithm as input, if any of those conditions is not satisfied, return false and we can also identify at termination which vertex at which condition is incorrect. Assume now that the first condition has been met, we attempt to show that the later 2 conditions are sufficient and correct in verifying that the functions d and π match those of the true shortest path tree rooted at s .

Suppose that with the 2 later conditions, the verification algorithm terminates and doesn't return false even though the input is wrong. Let v be the vertex closest to the root s whose $d(v)$ is the smallest incorrectly computed distance, and let $\delta(v)$ be the true shortest distance from s to v .

Suppose $d(v) < \delta(v)$, that is, somehow the computed distance $d(v)$ is smaller than the true shortest distance $\delta(v)$. By condition 2, $d(v)$ can no longer be reduced and $\pi(v)$ is the best neighbor, otherwise we would have detected a better one at termination. Since v is the vertex of smallest value $d(v)$ that has been computed incorrectly, the shortest distance value of $\pi(v)$, i.e. $d(\pi(v))$, must be correct (because $\pi(v)$ is higher up and closer to root s in the tree, and every edge $(k, v) \in E, k \in N_v$ is non-negative so $\delta(\pi(v))$ can only be $\leq \delta(v)$). Thus, we obtain:

$$\begin{aligned} d(v) &< \delta(v) \\ &\leq d(\pi(v)) + \omega(\pi(v), v) \\ &= d(\pi(v)) + \omega(\pi(v), v) \\ \Rightarrow d(v) &< d(\pi(v)) + \omega(\pi(v), v) \end{aligned}$$

This contradicts to the fact that $\pi(v)$ is the predecessor of v and that $d(v) = d(\pi(v)) + \omega(\pi(v), v)$ by condition 3.

The argument above implies that you can only overestimate the distance when $d(v)$ is computed incorrectly, that is, $\delta(v) < d(v)$. We show that such a case is impossible, as we have the following:

$$\begin{aligned} d(v) &= d(\pi(v)) + \omega(\pi(v), v) \\ &= \delta(\pi(v)) + \omega(\pi(v), v) \\ &\leq \delta(v) + \omega(\pi(v), v) \\ &< d(v) + \omega(\pi(v), v) \end{aligned}$$

, which implies that $\omega(\pi(v), v) < 0$ (a contradiction). Therefore, by condition 2, if there exists a better neighbor k than $\pi(v)$, we would have detected it at termination. And by condition 3, we've made sure $d(v)$ is computed correctly with the right predecessor $\pi(v)$.

III. Implementation

Many pieces and parts of the algorithms have been provided with pseudo-code, therefore only some of the implementation will be discussed briefly here.

1. Data Structure

1.1 Lowest Common Ancestor

As we've seen, it's fair to say that the heart and soul of the algorithms is the *LCA* data structural problem on the dual $D(G)$. The data structure that Theorem 2 refers to is not just the dual $D(G)$ but also the *LCA* data structure as well. The topic is worthy of a complete separate lecture/project, and for now we will take it for granted and simply use this as a black box for our main purpose. The implementation is thankfully provided by [?].

1.2 DCEL

At its core, our data structure is a variation of edge list, namely the doubly-connected edge list (DCEL) where we have *vertex*, *halfedge* and *node* objects/structures closely linked to each other.

A *vertex* object has:

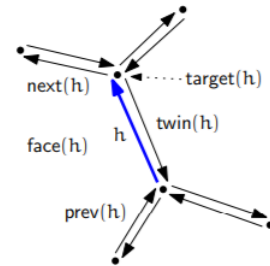
- value/label
- coordinate

A *node* object has:

- value/label
- pointer to any of its incident *halfedge*

A *halfedge* object has:

- weight
- pointer to its target *vertex*
- pointer to its twin *halfedge*
- pointer to its incident *node*
- pointer to its next *halfedge*
- pointer to its previous *halfedge*



The data structure is essentially a collection of linked lists closely intertwine with each other. As such, checking if there's an edge between 2 given vertices is not possible in constant time. We perform data structure augmentation by adding an adjacency list of constant size to each vertex. The following lemma allows us to do such task:

Lemma: For any simple connected outerplanar graph of size $|V| \geq 2$, there must exists a vertex with degree at most 2.

In our implementation, while building our maximal outerplanar graph (as we will see in the next section), every new vertex we are about to add will have degree 2. Therefore, we can immediately add to adjacency

list the 2 *halfedge* edges incident to and directed towards that vertex, rather than spending time searching for such a vertex. When checking if (u, v) is an edge, check if vertex u is in vertex v 's adjacency list or vice versa. The correctness of this constant size adjacency lists follows from the fact that once we've removed such a vertex with degree at most 2, along with its incident edges, we have a subgraph which is still an outerplanar graph. The above lemma still holds and we can continuously build our adjacency list by "removing" such vertex. In the end, we will have maintained the connectivity of the entire graph with this small adjacency list in each vertex, and thus we can check if 2 vertices are adjacent or not.

Of course, there are also additional book-keeping information required from preprocessing steps (beer distance, Lemma 4, 12-15) as well as for our shortest path algorithms (DAG edges, predecessor and shortest distance attributes). The complete detail of the data structure can be found in:

<https://github.com/charlespnh/shortest-beer-path/blob/master/include/graph/dcel.h>

The DCEL data structure supports the following operation:

```
struct vertex* add_vertex(int val, int radius, float beer_prob);
struct halfedge* add_edge(struct halfedge* h, struct vertex* dest, struct node* f1, struct node* f2);
struct halfedge* split_face(struct halfedge* h, struct vertex* dest, struct node* f1, struct node* f2);
```

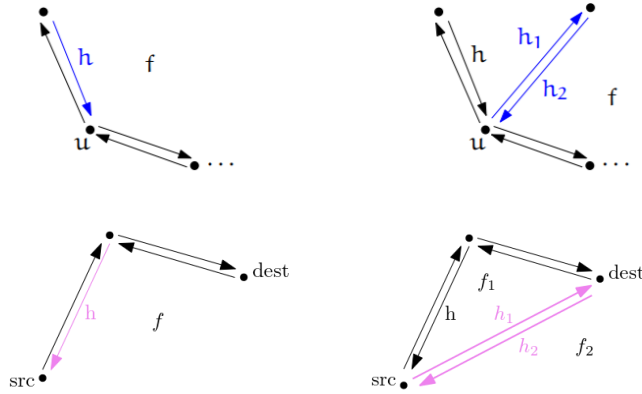
We will discuss the *add_vertex()* operation later. Operations *add_edge()* and *split_face()* each takes constant time and is only a matter of manipulating constant number of pointers:

```
struct halfedge* add_edge(struct halfedge* h, struct vertex* dest, struct node* f1, struct node* f2){
    struct vertex* src = h->target;
    struct halfedge* h1 = new halfedge();
    struct halfedge* h2 = new halfedge();
    h1->weight = h2->weight = euclidean_dist(src, dest);
    h1->beer_edge = h2->beer_edge = new b_edge(src, dest);

    h1->twin = h2;
    h2->twin = h1;
    h1->target = dest;
    h2->target = src;
    h1->incident_face = f1;
    h2->incident_face = f2;
    if (f2 != NULL) f2->incident_edge = h2;

    h1->next = h2;
    h2->next = h->next;
    h1->prev = h;
    h2->prev = h1;
    h->next = h1;
    h2->next->prev = h2;

    return h1;
}
```



```

struct halfedge* split_face(struct halfedge* h, struct vertex* dest, struct node* f1, struct node* f2){
    struct vertex* src = h->target;
    struct halfedge* h1 = new halfedge();
    struct halfedge* h2 = new halfedge();
    h1->weight = h2->weight = euclidean_dist(src, dest);
    h1->beer_edge = h2->beer_edge = new b_edge(src, dest);
    h1->incident_face = f1;
    h2->incident_face = f2;
    if (f2 != NULL) f2->incident_edge = h2;
    h1->twin = h2;
    h2->twin = h1;
    h1->target = dest;
    h2->target = src;

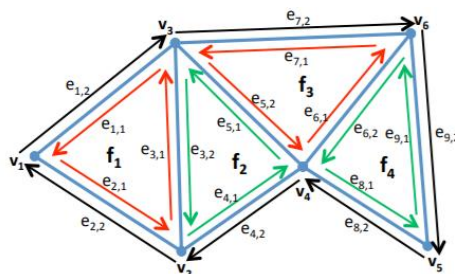
    h2->next = h->next;
    h2->next->prev = h2;
    h1->prev = h;
    h->next = h1;

    h->prev->prev->next = h2;
    h2->prev = h->prev->prev;
    h1->next = h->prev;
    h1->next->prev = h1;

    return h1;
}

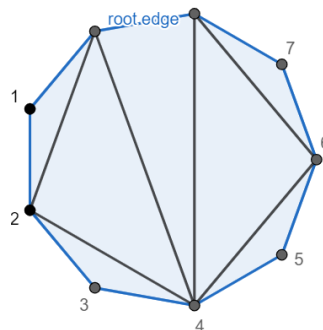
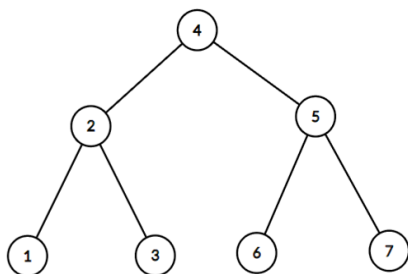
```

In addition to the operations above, other trivial operations are possible, such as getting source vertex of a *halfedge*, getting parent of a *node*, walking along *v*-chain (or traversing the incident edges to *v*) and traversing outer edges (boundary of *G*).



2. Construct Maximal Outerplanar Graph

Intuitively, one can start with some maximal outerplanar graph and compute its dual. In our implementation, we start with the dual first and build our graph based on that. We observe that there is a bijection between a binary tree and a triangulated convex polygon [?].



We first start by generating a binary tree. The implementation includes 3 types of binary tree (courtesy of [?]) which are random, balance and skew-right. We then build and recursively triangulate the polygon based on the binary tree. While traversing the binary tree, at each node we construct the corresponding triangular face in the polygon.

Although our binary tree can be random, we still want some structure to it so that we can determine which side of the polygon to recurse and triangulate. In order to implement this bijection, we label the binary tree with values (in this case, numbers) in an in-order fashion and have our polygon labeled counter-clockwise. Now, we can build and triangulate the polygon as follow:

```
void graph::build_polygon(){
    triangulate_start = clock();
    struct vertex* u = dcel::add_vertex(-1, mV, mbeer_probability);
    struct vertex* v = dcel::add_vertex(0, mV, mbeer_probability);
    struct halfedge* h1 = new halfedge();
    struct halfedge* h2 = new halfedge();
    h1->weight = h2->weight = euclidean_dist(u, v);
    h1->beer_edge = h2->beer_edge = new b_edge(u, v);

    h1->twin = h2;
    h2->twin = h1;
    h1->target = v;
    h2->target = u;
    h1->incident_face = mroot_node;
    h2->incident_face = NULL;
    h1->incident_face->incident_edge = h1;
```

```

        h1->next = h2;
        h2->next = h1;
        h1->prev = h2;
        h2->prev = h1;

        mroot_edge = h1;
        triangulate_polygon(mroot_edge, mroot_node);
        triangulate_duration = (clock() - triangulate_start) / (double) CLOCKS_PER_SEC;

        u->adj = {dcel::twin(mroot_edge), dcel::prev(mroot_edge)};
        v->adj = {mroot_edge, dcel::twin(dcel::next(mroot_edge))};
        return;
    }

void graph::triangulate_polygon(struct halfedge* root_edge, struct node* root_node){
    if (root_node == NULL) return;

    struct vertex* v = dcel::add_vertex(dcel::iData(root_node), mV, mbeer_probability);
    struct halfedge* new_root_edge;

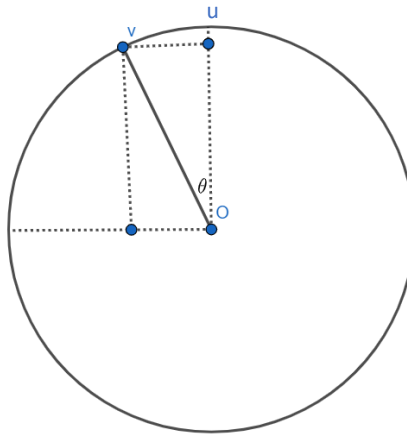
    new_root_edge = dcel::add_edge(root_edge, v, root_node, dcel::left(root_node));
    triangulate_polygon(dcel::twin(new_root_edge), dcel::left(root_node));

    new_root_edge = dcel::split_face(new_root_edge, dcel::origin(root_edge), root_node, dcel::right(root_node));
    triangulate_polygon(dcel::twin(new_root_edge), dcel::right(root_node));

    // create adj list for vertex v
    v->adj = {dcel::next(root_edge), dcel::twin(new_root_edge)};
    return;
}

```

We also want to make sure our polygon is big and “balance” enough so that no 2 vertices are too close to each other, which can result in arithmetic error when computing edge weight. By imagining that we place our vertices evenly on a circle whose radius $r = |V| = n$, we make sure that each arc length between 2 neighbor vertices (u, v) on the circle is 2π and $\theta = \frac{2\pi}{n}$.



Note that there's an interesting property if we place the vertices this way. For every 2 neighbor vertices u, v on the circle, observe that

$$\begin{aligned}
 v_x &= r \cdot \sin(\theta) = n \cdot \sin\left(\frac{2\pi}{n}\right) \\
 \lim_{n \rightarrow \infty} n \cdot \sin\left(\frac{2\pi}{n}\right) &= \lim_{n \rightarrow \infty} \frac{\sin\left(\frac{2\pi}{n}\right)}{\frac{1}{n}} \\
 &\stackrel{L'H}{=} \lim_{n \rightarrow \infty} \frac{\frac{2\pi}{n^2} \cdot \cos\left(\frac{2\pi}{n}\right)}{\frac{1}{n^2}} \\
 &= \lim_{n \rightarrow \infty} 2\pi \cdot \cos\left(\frac{2\pi}{n}\right) \\
 &= 2\pi \cdot 1 = 2\pi
 \end{aligned}$$

Since the arc length uv on the circle is also 2π , the end point of the segment v_x will eventually coincide with vertex u as $n \rightarrow \infty$. In other word, $\forall e = (u, v) \in E$ where u, v are neighbor vertices on the circle, $\omega(e) \rightarrow 2\pi$ as the graph G grows larger in terms of number of vertices. The remark is useful for the process of debugging since we can quickly recognize/approximate the edge weight of our graph G just by sketching out on paper rather than having to examine the coordinate of the vertices.

```

struct vertex* add_vertex(int val, int radius, float beer_prob){
    struct vertex* new_vertex = new vertex(val);
    new_vertex->point.first = radius * sin((val + 1) * (2*M_PI/radius));
    new_vertex->point.second = radius * cos((val + 1) * (2*M_PI/radius));
    new_vertex->is_beer = random_bernoulli(beer_prob);
    new_vertex->beer_edge = new b_edge(new_vertex, new_vertex);
    return new_vertex;
}

```

We want to randomly assign a vertex to be a beer store, given the input probability as well as the probability distribution (Bernoulli distribution).

3. Beer Distance

There are some notable differences in the implementation compared to the pseudo-code due to the choice of our data structure. We define a separated beer edge *b_edge* structure where each *vertex* and *halfedge* object will point to.

```

struct b_edge {
    struct vertex* u;
    struct vertex* v;
    double distB;
    pair<struct vertex*, bool> pathB;

    b_edge(struct vertex* a, struct vertex* b){
        u = a;
        v = b;
        distB = INF;
        pathB = NIL;
    }

    b_edge(){
    }
};

```

Computing the distance is still the same via the recurrence relations, but we only need to store δ_B and ω_B during the computation (*distB* and *pathB* attributes in the *b_edge* structure, respectively). The other attributes d_B and π_B is not necessary. It is intuitive that each edge in G should only point to its corresponding *b_edge* structure. However, in our DCEL data structure, an edge in G is represented by 2 *halfedges*. With 2 *halfedges* involve, computing δ_B is still the same since these *halfedges* share the same weight so they should share the same beer distance as well. But when it comes to setting ω_B , things are a lot more complicated.

Note that the *b_edge* structure has its vertices u and v correspond to only one of the *halfedge* twins. We have to be aware of the fact that the other twin *halfedge* is in the opposite direction. When setting ω_B (or *pathB* attribute), we are processing some face F of G and the computation will be respect to some *halfedge* twin incident to F , not the other twin edge. This causes ambiguity when we later attempt to reconstruct the actual instance of the shortest beer path for each edge using *b_edge* structure and its *pathB* attribute.

One approach is to make each *halfedge* points to a separate *b_edge* structure, even though they share the same beer distance *distB* attribute. In addition, we'll have to go through all the complex recurrence relations twice, one for each *halfedge* twin just so we can set *pathB* attributes correctly. It's a waste of memory and constant amount of more workload. The problem has been safely handled in the implementation where some swappings of u and v in the *b_edge* structure are needed during the computation and when printing out the beer path.

<https://github.com/charlespnh/shortest-beer-path/blob/master/utils/preprocess/beer.cpp>

4. Answer Shortest Beer Path Queries

4.1 DAG

We store the DAG using adjacency list data structure where the vertices are in a dynamic array and each vertex store another adjacency list of constant size containing the DAG edges (since each column in DAG only has at most 2 vertices). The vertices in DAG will be the same *vertex* structure as those previously created. A DAG edge is implemented as a separated first-class object:

```
typedef vector<pair<struct d_edge*, struct d_edge*> > DAGEdges;

struct d_edge {
    double weight;
    struct vertex* dest;
    pair<struct vertex*, int> subpath;

    d_edge(struct vertex* d){
        double weight;
        dest = d;
        subpath = NIL;
    }

    d_edge(){
    }
};
```

Much of our preprocessing steps is for building DAG efficiently by computing its vertices and edges in constant time. All that's left to do is to combine all the pieces together:

```
void dag::build_dag(graph& G, struct vertex* src, struct vertex* sink){
    struct node* u = dcel::face(src);
    struct node* v = dcel::face(sink);
    struct vertex* col_vtx1 = src;
    struct vertex* col_vtx2 = src;
    struct halfedge* col_edge = NULL;
    struct halfedge* sink2vtx = NULL;

    clock_t start = clock();
    do {
        // Answer last label query and second node query
        struct node* lnq = G.last_node_query(col_vtx1, u, v);
        if (col_vtx1 != col_vtx2 && graph::incident_vertex2face(col_vtx2, lnq))
            lnq = G.last_node_query(col_vtx2, u, v);
        struct node* snq = G.second_node_query(lnq, v);
        u = snq;

        // Get the next (i+1)th DAG column edge
        if (dcel::face(dcel::twin(dcel::edge(lnq))) == snq)
            col_edge = dcel::edge(lnq);
        else col_edge = dcel::twin(dcel::edge(snq));
    } while (col_vtx1 != col_vtx2);
}
```

```

    // Build ith DAG column with col_vtx1 & col_vtx2
    if (col_vtx1 == col_vtx2){ // edge case: 0th col in DAG
        src->dag_edges = dag::compute_dag_edges_off_chain(src, col_edge);
        mDAG.push_back(src);
    }
    else dag::compute_dag_edges(col_vtx1, col_vtx2, col_edge); // ith col in DAG

    // Update col_vtx1 & col_vtx2: Get the next (i+1)th DAG column vertices
    col_vtx1 = dcel::target(col_edge);
    col_vtx2 = dcel::origin(col_edge);
} while(! (sink2vtx = graph::get_edge(sink, col_vtx1)) && ! (sink2vtx = graph::get_edge(sink, col_vtx2)));

// Build (L-1)th DAG column with col_vtx1 & col_vtx2
if (col_vtx1 != dcel::target(sink2vtx)){
    swap(col_vtx1, col_vtx2);
    col_edge = dcel::twin(col_edge);
}

// DAG shortest path edge
struct d_edge* sp_e1 = new d_edge(sink);
struct d_edge* sp_be1 = new d_edge(sink);
sp_e1->weight = dcel::weight(sink2vtx);
sp_be1->weight = beer::weightB(sink2vtx);
sp_e1->subpath = sp_be1->subpath = NIL;

// DAG shortest path beer edge
struct d_edge* sp_e2 = dag::compute_spedge_on_chain(col_vtx1, col_vtx2, sink);
struct d_edge* sp_be2 = dag::compute_spedge_on_chain(col_vtx1, col_vtx2, sink);

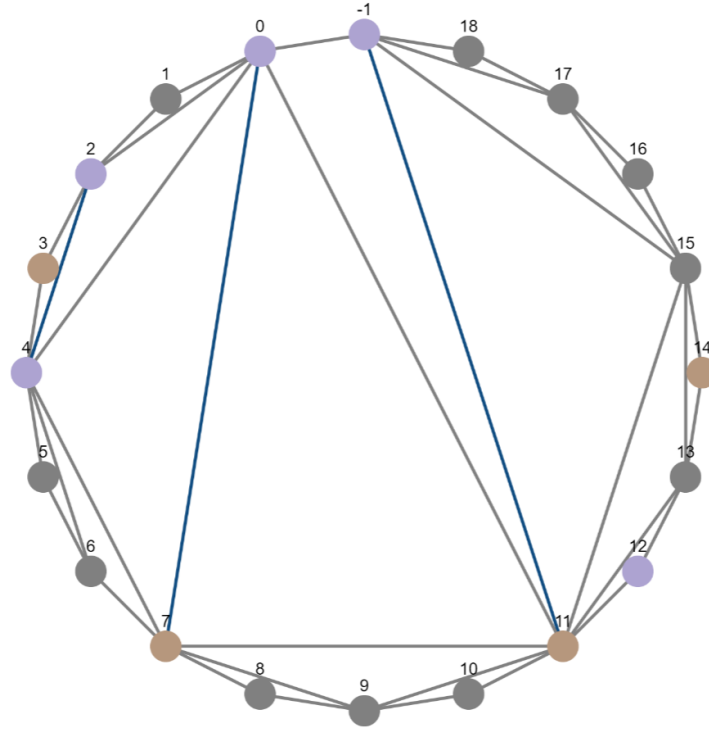
col_vtx1->dag_edges = {make_pair(sp_e1, sp_be1)};
col_vtx2->dag_edges = {make_pair(sp_e2, sp_be2)};
mDAG.push_back(col_vtx1);
mDAG.push_back(col_vtx2);

// Add last Lth DAG column with sink
mDAG.push_back(sink);

build_duration = (clock() - start) / (double) CLOCKS_PER_SEC;
return;
}

```

Inside the loop, we run the Closest-color Query (Algorithm 2) as well as Second-node Query (Algorithm 1) to get the edge *col_edge* whose vertices are in the next DAG column. As an actual example, given our graph G with 20 vertices and $D(G)$ is a balance binary tree, a DAG for queried vertices $src = 3$, $sink = 12$ looks as follow:



The brown vertices are the beer stores and the purple vertices are on the DAG. The edges colored in blue are the *col_edges* that we obtain as we go on in the loop. *col_edge* is merely an implementation detail that make it easier to compute for the DAG (beer) edges and (column) vertices. We now compute the DAG edges going from the current i^{th} column's vertices (*col_vtx1* and *col_vtx2*) to the next $(i + 1)^{th}$ column vertices (vertices in edge *col_edge*) as follows:

```
void dag::compute_dag_edges(struct vertex* col_vtx1, struct vertex* col_vtx2, struct halfedge* col_edge){
    if (col_vtx1 == dcel::apex(col_edge)){           // halfedge col_edge now faces vertex col_vtx1
        col_vtx1->dag_edges = dag::compute_dag_edges_off_chain(col_vtx1, col_edge);
        col_vtx2->dag_edges = dag::compute_dag_edges_on_chain(col_vtx2, col_edge);
    }
    else{
        col_vtx1->dag_edges = dag::compute_dag_edges_on_chain(col_vtx1, col_edge);
        col_vtx2->dag_edges = dag::compute_dag_edges_off_chain(col_vtx2, col_edge);
    }

    mDAG.push_back(col_vtx1);
    mDAG.push_back(col_vtx2);
}
```

We identify whether the vertices in edge *col_edge* belongs to $G[P_{col_vtx1}]$ or $G[P_{col_vtx2}]$. If *col_edge* belongs to $G[P_{col_vtx}]$ for some vertex *col_vtx*, then there is an edge between *col_vtx* and each of the vertices in edge *col_edge*. This corresponds to the trivial edge cases handled by Algorithm 9 and 10, and we have the following:

```

DAGEdges dag::compute_dag_edges_off_chain(struct vertex* col_vtx, struct halfedge* col_edge){
    // DAG shortest path edges
    struct d_edge* sp_e1 = new d_edge(dcel::target(col_edge));
    struct d_edge* sp_e2 = new d_edge(dcel::origin(col_edge));
    sp_e1->weight = dcel::weight(dcel::next(col_edge));
    sp_e2->weight = dcel::weight(dcel::prev(col_edge));

    // DAG shortest path beer edges
    struct d_edge* sp_be1 = new d_edge(dcel::target(col_edge));
    struct d_edge* sp_be2 = new d_edge(dcel::origin(col_edge));
    sp_be1->weight = beer::weightB(dcel::next(col_edge));
    sp_be2->weight = beer::weightB(dcel::prev(col_edge));

    return {make_pair(sp_e1, sp_be1), make_pair(sp_e2, sp_be2)};
}

```

Otherwise, *col_edge* does not belongs to $G[P_{col_vtx}]$ for some vertex *col_vtx*. But both edge *col_edge* and vertex *col_vtx* still belong to the chain of some other apex vertex, which is the remaining DAG vertex in the current i^{th} column. We first compute the DAG shortest path edge as well as the corresponding DAG shortest path beer edge going from *col_vtx* to the vertices in edge *col_edge*.

```

DAGEdges dag::compute_dag_edges_on_chain(struct vertex* col_vtx, struct halfedge* col_edge){
    struct vertex* apexv = dcel::apex(col_edge);

    struct d_edge* sp_e1 = compute_spedge_on_chain(apexv, col_vtx, dcel::target(col_edge));
    struct d_edge* sp_be1 = compute_spbedge_on_chain(apexv, col_vtx, dcel::target(col_edge));

    struct d_edge* sp_e2 = compute_spedge_on_chain(apexv, col_vtx, dcel::origin(col_edge));
    struct d_edge* sp_be2 = compute_spbedge_on_chain(apexv, col_vtx, dcel::origin(col_edge));

    return {make_pair(sp_e1, sp_be1), make_pair(sp_e2, sp_be2)};
}

```

Below is the implementation of the final case of Algorithm 9:

```

struct d_edge* dag::compute_spedge_on_chain(struct vertex* apex, struct vertex* u, struct vertex* v){
    int index1 = dcel::apex_index(graph::get_edge(u, apex));
    int index2 = dcel::apex_index(graph::get_edge(v, apex));
    struct d_edge* sp_e = new d_edge(v);

    double path_weight_on_chain, path_weight_cross_apex;

    /* DAG shortest path edge */
    path_weight_on_chain = abs(dcel::dist2cw(apex, index1) - dcel::dist2cw(apex, index2));
    path_weight_cross_apex = dcel::weight(graph::get_edge(u, apex)) + dcel::weight(graph::get_edge(apex, v));
    sp_e->weight = min(path_weight_on_chain, path_weight_cross_apex);

    if (sp_e->weight == path_weight_on_chain)
        sp_e->subpath = make_pair(apex, 1); // 1 if SP is on apex's chain
    else sp_e->subpath = make_pair(apex, 0); // 0 if SP is crossing apex

    return sp_e;
}

```

And likewise for the implementation of the final case of Algorithm 10. Again, direction is implied when we are trying to go from a vertex u to v . The pseudo-code provided was for the case of going clockwise along the apex's chain, but the opposite case can be handled quite easily. Every vertex v is an apex of its own v -chain, and as stated in the pseudo-code, during the preprocessing step, we store the path p_v along the v -chain with the vertices in counter-clockwise order. As such, if vertex v is stored later in the array than vertex u (index $u < \text{index } v$), then we go counter-clockwise from u to v along p_v of the chain.

```

struct d_edge* dag::compute_spbedge_on_chain(struct vertex* apex, struct vertex* u, struct vertex* v){
    struct halfedge* u2apex = graph::get_edge(u, apex);
    struct halfedge* v2apex = graph::get_edge(v, apex);
    int index1 = dcel::apex_index(u2apex);
    int index2 = dcel::apex_index(v2apex);

    struct d_edge* sp_be = new d_edge(v);

    /* DAG shortest path beer edge */
    struct lca* lca_chain = apex->lca_Av_chain;
    struct node* Av_smallest;
    // cw on chain
    if (index1 < index2)
        Av_smallest = get_lca(lca_chain, index1, index2 - 1);
    // ccw on chain
    else Av_smallest = get_lca(lca_chain, index1 - 1, index2);

    double path_weight_on_chain, path_weight_cross_apex;
    path_weight_on_chain = abs(dcel::dist2cw(apex, index1) - dcel::dist2cw(apex, index2)) + dcel::dData(Av_smallest);
    path_weight_cross_apex = min(beer::weightB(u2apex) + dcel::weight(v2apex),
                                dcel::weight(u2apex) + beer::weightB(v2apex));
    sp_be->weight = min(path_weight_on_chain, path_weight_cross_apex);

    // if SP is on chain
    if (sp_be->weight == path_weight_on_chain)
        sp_be->subpath = make_pair(apex, dcel::iData(Av_smallest));

    // else SP is crossing apex
    else{
        if (sp_be->weight == beer::weightB(u2apex) + dcel::weight(v2apex))
            sp_be->subpath = make_pair(apex, -1); // 1st edge "cross apex" is the beer path
        else
            sp_be->subpath = make_pair(apex, -2); // 2nd edge "cross apex" is the beer path
    }

    return sp_be;
}

```

4.2 Dynamic Programming

Our DAG data structure's layout is a flat array, rather than column by column. As such, indexing to a vertex in the previous column requires some trivial indexing trick. Below is the implementation of the recurrences for the shortest (beer) path. We use the values *dist* and *pred* to store the computed shortest distance d and predecessor π attributes, as well as values *distB* and *predB* to store d_B and π_B attributes for each vertex.

0	1	2	3	4	5	6	7
s	a_1	b_1	a_2	b_2	a_3	b_3	t

```

void dag::shortest_path_dag(){
    clock_t start = clock();
    mDAG[1]->dist = dag::weight(mDAG[0], mDAG[1]);
    mDAG[1]->pred = mDAG[0];
    // edge case: an edge... no pun intended
    if (mDAG.size() == 2)
        return;
    mDAG[2]->dist = dag::weight(mDAG[0], mDAG[2]);
    mDAG[2]->pred = mDAG[0];

    for (int i = 3, j = 0; i < mDAG.size(); i++, j = !j){
        mDAG[i]->dist = min(dcel::dist(mDAG[i - j - 1]) + dag::weight(mDAG[i - j - 1], mDAG[i]),
                           dcel::dist(mDAG[i - j - 2]) + dag::weight(mDAG[i - j - 2], mDAG[i]));

        if (mDAG[i]->dist == dcel::dist(mDAG[i - j - 1]) + dag::weight(mDAG[i - j - 1], mDAG[i]))
            mDAG[i]->pred = mDAG[i - j - 1];
        else mDAG[i]->pred = mDAG[i - j - 2];
    }

    spsp_duration = (clock() - start) / (double) CLOCKS_PER_SEC;
    return;
}

void dag::shortest_beer_path_dag(){
    clock_t start = clock();
    mDAG[1]->distB = dcel::dist(mDAG[0]) + dag::weightB(mDAG[0], mDAG[1]);
    mDAG[1]->predB = make_pair(mDAG[0], 1);
    // edge case: an edge... no pun intended
    if (mDAG.size() == 2)
        return;
    mDAG[2]->distB = dcel::dist(mDAG[0]) + dag::weightB(mDAG[0], mDAG[2]);
    mDAG[2]->predB = make_pair(mDAG[0], 1);

    for (int i = 3, j = 0; i < mDAG.size(); i++, j = !j){
        mDAG[i]->distB = min(min(dcel::distB(mDAG[i - j - 1]) + dag::weight(mDAG[i - j - 1], mDAG[i]),
                                dcel::dist(mDAG[i - j - 1]) + dag::weightB(mDAG[i - j - 1], mDAG[i])),
                                min(dcel::distB(mDAG[i - j - 2]) + dag::weight(mDAG[i - j - 2], mDAG[i]),
                                    dcel::dist(mDAG[i - j - 2]) + dag::weightB(mDAG[i - j - 2], mDAG[i]))));

        if (dcel::distB(mDAG[i]) == dcel::distB(mDAG[i - j - 1]) + dag::weight(mDAG[i - j - 1], mDAG[i]))
            mDAG[i]->predB = make_pair(mDAG[i - j - 1], 0);
        else if (dcel::distB(mDAG[i]) == dcel::distB(mDAG[i - j - 2]) + dag::weight(mDAG[i - j - 2], mDAG[i]))
            mDAG[i]->predB = make_pair(mDAG[i - j - 2], 0);
        else if (dcel::distB(mDAG[i]) == dcel::dist(mDAG[i - j - 1]) + dag::weightB(mDAG[i - j - 1], mDAG[i]))
            mDAG[i]->predB = make_pair(mDAG[i - j - 1], 1);
        else mDAG[i]->predB = make_pair(mDAG[i - j - 2], 1);
    }

    spsbsp_duration = (clock() - start) / (double) CLOCKS_PER_SEC;
    return;
}

```


Printing the instance of shortest path is relatively easy as we only need to handle 2 cases, either shortest path $\pi(v)$ to v goes along the chain of some apex or crosses through that apex (which is kept track of while computing the DAG edges). As such, we have the following:

```
// subpath v1 to v2 - travel along apex's chain
vector<struct vertex*> print_subpath_on_chain(struct vertex* apex, struct vertex* v1, struct vertex* v2){
    vector<struct vertex*> path;
    struct vertex* u = v1;
    int index1 = dcel::apex_index(graph::get_edge(v1, apex));
    int index2 = dcel::apex_index(graph::get_edge(v2, apex));

    if (index1 < index2){                // cw
        while(u != v2){
            u = dcel::neighbour(apex, ++index1);
            path.push_back(u);
        }
    }
    else {                               // ccw
        while(u != v2){
            u = dcel::neighbour(apex, --index1);
            path.push_back(u);
        }
    }

    return path;
}

vector<struct vertex*> dag::print_subpath_dag(struct vertex* pred, struct vertex* v){
    vector<struct vertex*> path;
    struct d_edge* d_edge = dag::getDEdge(pred, v);
    auto [apex, direction] = d_edge->subpath;

    // DAG edge is an edge in graph G
    if (apex == NULL)
        path = {v};
    //
    else {
        // walk cross apex
        if (direction == 0)
            path = {apex, v};
        // walk cw along apex v_chain
        else
            path = print_subpath_on_chain(apex, pred, v);
    }

    return path;
}
```

```

vector<struct vertex*> dag::print_spsp_dag(struct vertex* v){
    vector<struct vertex*> path;

    if (dcel::pred(v) == NULL){
        path = {v};
        return path;
    }

    path = dag::print_spsp_dag(dcel::pred(v));
    vector<struct vertex*> subpath = dag::print_subpath_dag(dcel::pred(v), v);
    path.insert(path.end(), subpath.begin(), subpath.end());
    return path;
}

```

Printing the instance of shortest beer path, however, is a lot more difficult:

```

vector<struct vertex*> dag::print_spsbp_dag(struct vertex* v){
    vector<struct vertex*> pathB = dag::print_subpathB_dag(v);
    return path_union(pathB);
}

vector<struct vertex*> dag::print_subpathB_dag(struct vertex* v){
    vector<struct vertex*> pathB, subpathB;
    auto [w, beerLoc] = dcel::predB(v);
    if (w == NULL) return pathB;

    // just a normal shortest subpath
    if (beerLoc == 0){
        pathB = dag::print_spsbp_dag(w);
        subpathB = dag::print_subpath_dag(w, v);
    }

    // shortest beer subpath
    else{
        pathB = dag::print_spsp_dag(w);
        auto [b, direction] = dag::omega(dag::getDBEdge(w, v));

        // "an edge" in G
        if (b == NULL)
            subpathB = beer::print_beer_path(graph::get_edge(w, v));
    }
}

```

If a beer stored is visited before predecessor w of vertex v as indicated when $beerLoc = 0$, then we continue to recurse with w and after that, concatenate it with the DAG shortest path edge (w, v) . Otherwise, a beer store is visited now between the DAG shortest path beer edge (w, v) .

If (w, v) is an edge $\in E$ of G , then simply print the beer path of the edge (w, v) . If the sub-path taken was along the chain of apex b , then:

```

// on chain apex b
else if (direction >= 0){
    vector<struct vertex*> beer_edge;
    struct vertex* neigh1 = dcel::neighbour(b, direction);
    struct vertex* neigh2 = dcel::neighbour(b, direction + 1);
    // if ccw
    if (dcel::apex_index(graph::get_edge(w, b)) > dcel::apex_index(graph::get_edge(neigh1, b)))
        // now cw
        swap(neigh1, neigh2);

    subpathB = print_subpath_on_chain(b, w, neigh1);
    beer_edge = beer::print_beer_path(graph::get_edge(neigh1, neigh2));

    subpathB.insert(subpathB.end(), beer_edge.begin(), beer_edge.end());
    pathB.insert(pathB.end(), subpathB.begin(), subpathB.end());

    subpathB = print_subpath_on_chain(b, neigh2, v);
}

// cross apex b
else {
    // 1st edge is beer path
    if (direction == -1){
        subpathB = beer::print_beer_path(graph::get_edge(w, b));
        subpathB.push_back(v);
    }

    // 2nd edge is beer path
    else
        subpathB = beer::print_beer_path(graph::get_edge(b, v));
}

pathB.insert(pathB.end(), subpathB.begin(), subpathB.end());
return pathB;

```

Else, the sub-path crosses through the apex and such case can be handled easily. The necessary information to reconstruct the shortest path has been encoded while computing the DAG edges already, therefore it is only a matter making sure the paths are concatenated correctly.

IV. Analysis

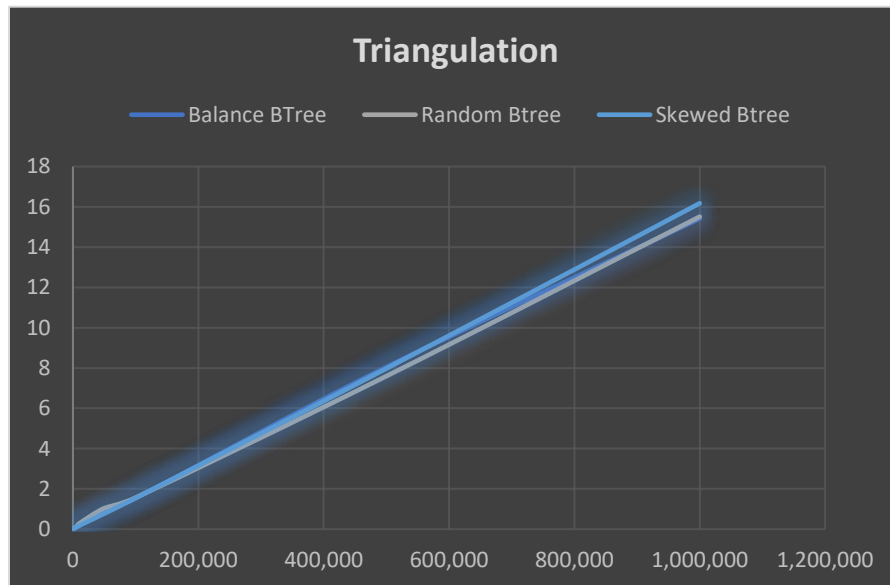
We will mainly analyze the actual runtime of the preprocessing phase. Since the query time depends on the length of the shortest path in addition to the somewhat unpredictable structure of the dual of the graph, there's no point in collecting data with regard to this. However, preprocessing and query time are stored and the user can obtain them by simply following the prompt menu.

1. Compare Duals of Graph

The runtime obtained below is in seconds:

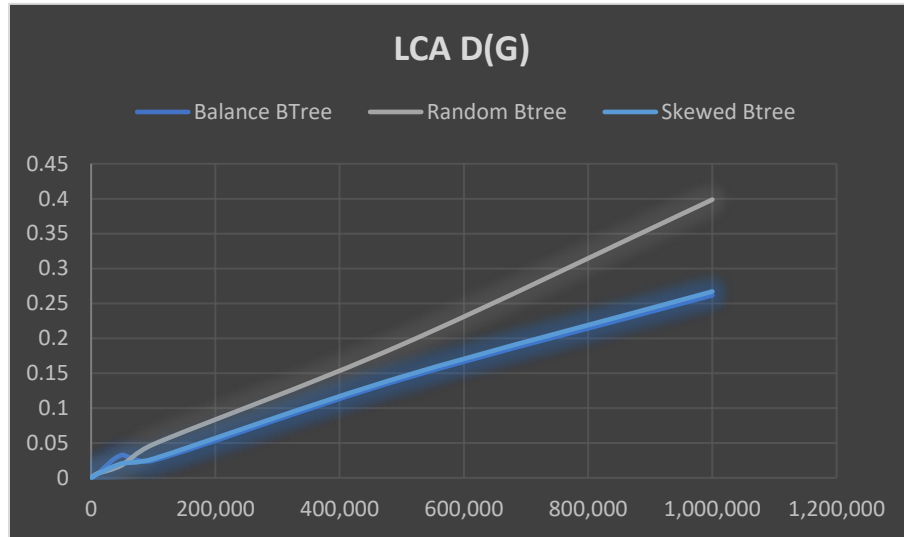
1.1 Triangulation

n	Balance BTree	Random Btree	Skewed Btree
1,000	0.025446	0.033965	0.029237
5,000	0.101882	0.101393	0.093201
10,000	0.172147	0.280461	0.179852
50,000	0.875221	1.04121	0.773405
100,000	1.53352	1.56583	1.54886
500,000	8.04199	7.596385	7.97151
1,000,000	15.4162	15.514435	16.176



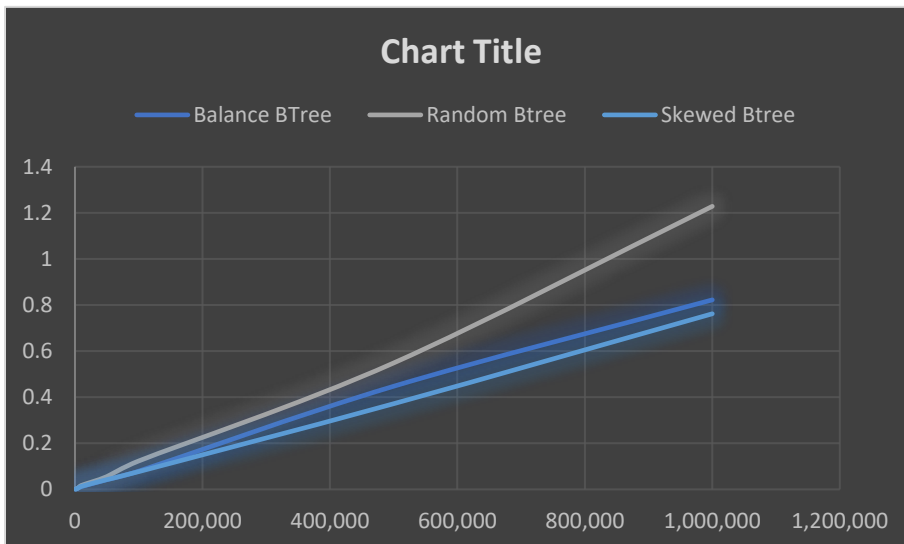
1.2 LCA on $D(G)$

n	Balance BTree	Random Btree	Skewed Btree
1,000	0.000566	0.001015	0.000655
5,000	0.002346	0.003641	0.002757
10,000	0.00572	0.006181	0.005518
50,000	0.032945	0.018421	0.020786
100,000	0.024966	0.048036	0.027323
500,000	0.141098	0.190837	0.144857
1,000,000	0.261598	0.3986465	0.266996



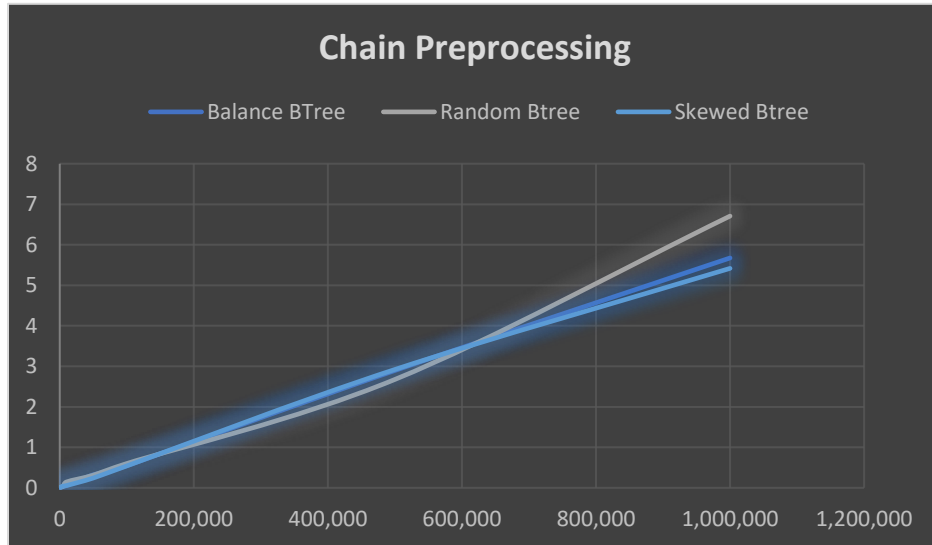
1.3 Beer Distance

n	Balance BTree	Random Btree	Skewed Btree
1,000	0.001038	0.001857	0.001222
5,000	0.00524	0.006405	0.005845
10,000	0.011939	0.017155	0.013202
50,000	0.041974	0.055537	0.042494
100,000	0.07909	0.123324	0.07705
500,000	0.44744	0.5486315	0.371756
1,000,000	0.822387	1.228323	0.762114



1.4 Chain preprocessing

n	Balance BTree	Random Btree	Skewed Btree
1,000	0.009127	0.016112	0.010087
5,000	0.033118	0.036214	0.042945
10,000	0.056399	0.14437	0.055361
50,000	0.321865	0.312217	0.241992
100,000	0.548028	0.599835	0.540999
500,000	2.8965	2.67534	2.92092
1,000,000	5.67347	6.70767	5.41717



2. Compare Beer Probability on Graph

n	0.30%	10%	45%	75%	90%
10,000	0.013815	0.010006	0.009283	0.008473	0.008616
50,000	0.051099	0.070593	0.065003	0.058368	0.051583
100,000	0.100645	0.132044	0.100269	0.099637	0.166646
500,000	0.550606	0.541539	0.550003	0.522951	0.579663
1,000,000	1.06307	1.11262	1.35795	1.13169	1.4427

