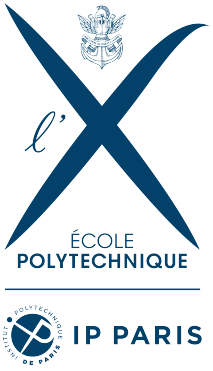


École Polytechnique

BACHELOR THESIS IN COMPUTER SCIENCE

Algorithmic Approaches for the 1D Distance Geometry Problem

Author:

Ha Duy Nguyen, École Polytechnique

Advisor:

Leo Liberti, LIX

Academic year 2024/2025

Abstract

The Distance Geometry Problem (DGP) involves determining whether a valid realization of a set of points exists, given pairwise distance constraints in Euclidean space. This problem has significant applications in fields such as sensor network localization, molecular conformation, and robotics. While DGP is NP-hard in general, its one-dimensional variant (DGP1) is NP-complete, and it exhibits unique structural properties that can be leveraged for efficient solution methods.

In this paper, we explore various algorithmic approaches to solving DGP1, focusing on branch-and-prune methods, combinatorial optimizations, and tree-based search techniques. We present a modular solver written in C++, implementing key optimizations such as bridge decomposition, triangle inequality checks, subset sum pruning, and randomized branching. We also examine the impact of different spanning tree structures on solver performance. The complete implementation is made available in an open-source repository for further study and application.

Contents

1 Introduction

1.1 The distance geometry problem

Distance geometry is a branch of mathematics concerned with characterizing and studying sets of points based only on distances.

The modern theory of distance geometry began in 19th century with work by Arthur Cayley, then Karl Menger. The results found by Menger was eventually completed by Leonard Blumenthal, who gave a general overview for distance geometry at the graduate level [1].

A fundamental problem in distance geometry is to determine if there exists a realization of points satisfying the given distances between some pairs of points. We refer to it as the *Distance Geometry Problem*. Formally [2],

> **DISTANCE GEOMETRY PROBLEM (DGP):** Given an integer $K > 0$ and a simple undirected graph $G = (V, E)$ whose edges are weighted by a nonnegative function $d : E \to \mathbb{R}^+$, determine whether there is a function $x : V \to \mathbb{R}^K$ such that
>
> $$\forall \{u, v\} \in E, \quad \|x(u) - x(v)\| = d(\{u, v\}). \tag{1}$$
>
> The function x satisfying (1) is called a realization of G in $\mathbb{R}^K$

Distance geometry problems arise whenever one needs to infer the shape of a configuration of points (relative positions) from the distances between them. DGP is typically studied in many applications of distributed sensor networks, where there arises the problem of determining the locations of sensors from incomplete (and possibly errorful) information about their distances from each other and from fixed landmarks.

In 1979, J. Saxe [3] introduced DGP as the K-embeddability problem, and showed that it is *NP-complete* for $K = 1$ fixed, and *NP-hard* otherwise. In this paper, we will examine specifically the case of DGP in 1-dimension, i.e with K = 1. For the sake of brevity, we will refer to this problem as *DGP1*. See more details on different difficulty classes of decision problems in A.

1.2 Goals and Contributions

In this paper, we will develop a DGP1 solver. At heart, the DGP1 solver will determine the answer to DGP1 (which is YES/NO), as well as finding a certificate in case the answer is YES. However, as DGP does not impose any structure on the graph G (see (1)), there might be multiple realizations satisfying the distance constraints, which the user might want to know. Thus, our solver should also support listing all (or many) possible realizations associated with the input. Overall, the program should support:

- Answering the DGP1 instance associated with the input (YES/NO).
- Giving one or multiple certificates if the output is YES.
- Implementing various optimizations that this paper will study, and allowing the user to freely select the subset of optimizations that will be enabled (i.e the implementation is modular). As a result,
- Each optimization should be implemented independently, and be as fast as possible.

Our main contributions are:

- Exploring algorithmic approaches for solving DGP1 in practical settings.
- Identifying structural properties of DGP1 that influence runtime.

- Providing experimental results on instances of DGP1.

The program will be written in C++ to prioritize runtime, due to it being a compiled, highly-optimized language. The complete DGP1 solver, equipped with all optimizations mentioned in this paper section, is available on my GitHub repository [4], with code and instructions on how to use the program.

2 Building a crude DGP1 solver

2.1 Separating connected components

Our solver will take AMPL .dat graph files as input. See A for example and explanation of AMPL .dat files.

As DGP does not enforce any structure on its input graph G, the graph G might be disconnected, i.e there might be multiple *connected components* in the graph. A connected component of an undirected graph is a connected subgraph that is not part of any larger connected subgraph. The components of any graph partition its vertices into disjoint sets, and are the induced subgraphs of those sets.

Since the only constraints in DGP are distance constraints (given by edge weights), different connected components are completely independent from each other. Formally, if the graph G has n connected components, and let X_i be the realization set of the i-th component, where each realization is defined on a domain D_i which is a subset of $\mathbb{N}$. The characteristics of connected components imply that $D_i \cap D_j = \varnothing$ for all i, j in $1..n$. The realization set of G is given by

$$X_G = \{F(x_1, \ldots, x_n) \mid x_i \in X_i\} \tag{2}$$

where

$$F(x_1, \ldots, x_n) = \sum_{1}^{n} x_i \cdot \mathbf{1}_{D_i} \tag{3}$$

It can be easily shown from 3 by induction that $\forall A, B, C$ functions with pairwise-disjoint domains,

$$F(A, B, C) = F(F(A, B), C) \tag{4}$$

In short, we can solve DGP independently for each connected component, and combine the solutions afterwards. As the process is basically the same for each component, we'd rather be interested in figuring out how to solve the problem for connected graphs (i.e graphs with only one connected component). We will use *Disjoint-Set Union (DSU)* to detect if the input graph is not connected, and use the utility program `splitutil` to separate each connected component into a different, standalone graph. `splitutil` works by identifying and compressing the range of value of vertices for each connected components (see appendix).

Algorithm Check graph connectivity

Require: Undirected weighted graph $G = (V, E, d)$
Ensure: **true** if G is connected, otherwise **false**

```
Initialize n ← |V|                                  ▷ Number of vertices
Initialize DSU structure for n elements
Initialize components ← n                           ▷ Initially, each vertex is its own component
for each edge (u, v) ∈ E do
    if Find(u) ≠ Find(v) then
        Union(u, v)                                 ▷ Merge components
        components ← components − 1
    end if
end for
return components = 1                               ▷ Graph is connected if only one component remains
```

From now on, we assume that the input graph G is connected.

2.2 Non-uniqueness of "YES" certificates

DGP1 takes place in the Euclidean space of dimension 1, i.e a (Euclidean) line. The solution to DGP1 is equivalent to finding the placement of points on the real line to satisfy all distance constraints.

Take the example graph $G = (V, E, d)$ given as below:

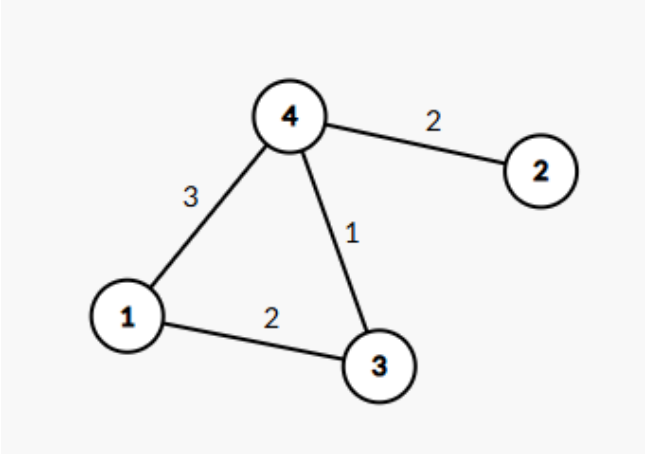


Figure 1: Illustration of the example DGP1 input

In particular, we have $V = \{1, 2, 3, 4\}$, $E = \{(1, 3), (3, 4), (4, 1), (4, 2)\}$ and

$$d(1, 3) = 2,\ d(3, 4) = 1,\ d(1, 4) = 3,\ d(2, 4) = 2$$

It is possible to construct several realizations satisfying the distance constraints given in d. One such realization is

$$x(1) = 0,\ x(2) = 1,\ x(3) = 2,\ x(4) = 3$$

Observe that in DGP, only the relative distance between the vertices matters. Thus, we can shift any realization to DGP1 by a constant real value to produce a new realization (e.g $x(1) = 1,\ x(2) = 2,\ x(3) = 3,\ x(4) = 4$). Similarly, as the distance is absolute, we can reverse the sign of any realization to create a new one (we will call this *mirror realizations*). Formally, if the function x is a realization of DGP1, then

$$\forall c \in \mathbb{R}, \quad y_{+_c} : \mathbb{R} \to \mathbb{R}, \quad y_{+_c}(a) \mapsto x(a) + c$$

and

$$\forall c \in \mathbb{R}, \quad y_{-_c} : \mathbb{R} \to \mathbb{R}, \quad y_{-_c}(a) \mapsto -x(a) + c$$

are also valid realizations.

This implies that if there exists a realization to DGP1, there are infinitely many other realizations. Thus, our solver should be able to unite all realizations that are formed by shifting the original realization by a constant. To do this, we will fix the value of vertex 1 at 0. This creates an anchor that prevent a realization to shift, while not creating any loss of generality. For now, we don't have to worry about mirror realizations that much, as for every realization there exists only one mirror realization (even though this will be discussed later).

2.3 The case with acyclic graphs

We start by examining the case where $G = (V, E, d)$ does not contain any cycle. As it is connected, it must be a tree. Let (u, v, w) signify that there is a distance constraint of value w between the vertices u and v. If one vertex' value is already known (WLOG, assume we know the value of $x(u)$), there are only two options for the value of v: at distance w either to the left of u or to the right of u on the Euclidean line, i.e $x(v) = x(u) - w$ or $x(v) = x(u) + w$. This means that every distance constraint introduces two possibilities (or *branches*) of assignment. By starting a *depth-first search (DFS)* at vertex 1 (whose value is always fixed to 0) and branching the search tree into 2 every time we visit a new vertex, we will be able to enumerate through all possible assignments. Let $n = |V|$. As the search space gets doubled every time we travel down an edge in E, the size of the search space will be exactly 2^{n-1}.

G is an undirected tree, and let it be rooted at 1. Having a tree structure means that the value of a vertex is only dependent on the value of its parent vertex. Thus, any assignment created by DFS from 1 is a valid realization. The answer to the decision problem is always "YES", and it only takes $O(n)$ (the time complexity of DFS on tree) to find a realization in this case. To list all solutions (which should be supported by the solver), we only need to implement the function F in 2 and 3.

Algorithm Function F

Require: Two disjoint partial assignments A and B
Ensure: Returns $F(A, B)$ where F is specified in 2 and 3

```
function F(A, B)
    Create cpy a copy of A
    for each key-value pair in B do
        Insert the key-value pair to cpy
    end for
    return cpy
end function
```

Algorithm DGP1 solver for tree graphs

Require: undirected weighted tree graph $G = (V, E, d)$

```
function DFS_Naive(v, val)                    ▷ Recursive-style implementation of DFS.
    Create an empty realization set X_v = {}
    Create a mapping x'
    Assign x'(v) ← val
    Add x' to X_v
    for each edge (v, u) ∈ E where u is not the parent of v do
        Let w ← d(v, u)
        Let rs ← {}
        Let X_+ ← DFS_Naive(u, val + w)
        Add X_+ to rs
        if list-all-solutions mode enabled then
            Let X_- ← DFS_Naive(u, val - w)
            Add X_- to rs
        end if
        X_v ← merge(X_v, rs)
    end for
    return X_v
end function
```

The above algorithm returns the list of one/all solutions depending on the user preference when calling `DFS_Naive(1, 0)`.

2.4 The case of cyclic graphs

Now let's try to understand what happens when we add cycles to the graph G. With the introduction of cycles, the value of a node might no longer depends solely on the value assigned to its parent node.

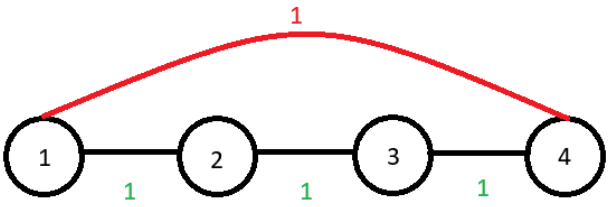


Figure 2: Illustration of a cyclic graph. A possible realization is $x(1) = 0, x(2) = 1, x(3) = 2, x(4) = 1$

In this graph, the addition of edge $(1, 4)$ with $d(1, 4) = 1$ (colored red) to the original tree graph (colored black) introduces a cycle. Note that the assignment $x(1) = 0,\ x(2) = 1,\ x(3) = 2,\ x(4) = 3$ is no longer valid, as the new constraint enforce the distance between node 1 and 4 is 1. If we run DFS from 1 and follow the edges in black, the value of node 4 now depends on both 3 and 1, not just on its parent as in a tree graph. Similarly, there are now another way to reach node 3 from the DFS root, either through node 2 or through node 4, thus its value is constrained from two directions. The same case goes for node 2. In other words, adding cycles to an acyclic graph will non-trivialize the number of possible realizations to the DGP1 problem associated with it.

We could still brute force through all possible assignments and eliminate the invalid ones. Let's now assume G is cyclic, and let G' the *spanning tree* of G. A spanning tree of an undirected graph G is a subgraph that is a tree which includes all of the vertices of G. Knowing that G' is a tree, we can call `DFS_Naive` on G' to find all 2^{n-1} potential realizations. We then check each realization against all the edges that are not in G', as they are the only source of potential conflicts. Unfortunately, this method runs in $\Theta(2^n)$ to find an answer, as it always need to gather all possible assignments first (since we cannot limit the search space by knowing how many realizations we should take from a sub-tree).

Luckily, an invalid partial assignment can be detected and pruned early before forming the full assignment. For example, if a partial assignment gathered from a sub-tree contradicts a constraint, we can immediately stop and try to find another partial assignment for that sub-tree (or backtrack and change the value of its parent node), without having to continue exploring and gathering partial assignments from the sub-trees of its *siblings*. Two nodes in a tree are siblings if they have the same immediate parent.

Figure 3 shows a possible opportunity for pruning. Assume the DFS algorithm has currently traversed the path (using black edges) from node 1 to 5, with the current partial assignment for each node colored in orange above the node label. When detecting a contradiction at node 5, pruning happens immediately. Now, instead of having to gather all assignments from the sub-tree of node 6, we try to backtrack and change the value of the nodes on the DFS path until the contradiction is resolved (in this case, only node 5 needs to be modified to contain the value 2). Only then we will start the exploration of the sub-tree at node 6. We call the edge $(5, 1)$ a *back edge*, i.e an edge that connects a node to its ancestor in the graph's spanning tree. In fact, every back edge in a graph presents an opportunity for pruning. This is because for DFS to reach a node, all its ancestors on the path from the root to itself must have been explored. We can then check for any back edge that leads to one of its already assigned ancestors to detect contradiction early.

Algorithm DGP1 solver with back edge pruning

Require: undirected weighted tree graph $G = (V, E, d)$, and G' a spanning graph of G

```
function DFS(v, val)
    Assign x(v) ← val
    for each back edge (v, u) ∈ E do
        if |x(v) − x(u)| ≠ d(v, u) then
            return ∅
        end if
    end for
    Create an empty realization set X_v = {}
    Create a mapping x'
    Assign x'(v) ← val
    Add x' to X_v
    for each edge (v, u) ∈ G' where u is not parent of v do
        Let w ← d(v, u)
        Let rs ← {}
        Let X_+ ← DFS(u, val + w, x)
        if X_+ ≠ ∅ then
            Add X_+ to rs
        end if
        if rs is empty or list-all-solutions mode enabled then
            Let X_− ← DFS(u, val − w, x)
            if X_− ≠ ∅ then
                Add X_− to rs
            end if
        end if
        if rs is empty then
            return ∅
        end if
        X_v ← merge(X_v, rs)
    end for
    return X_v
end function
```

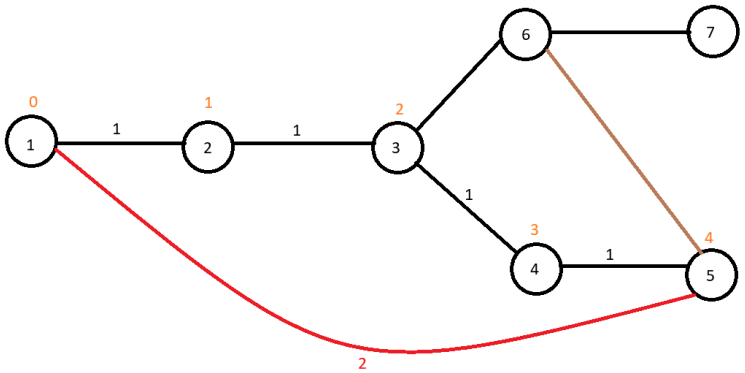


Figure 3: Possible pruning opportunity

The above algorithm details how to implement DFS with back edge pruning, which, in comparison to `DFS_Naive`, also includes the current assignment of nodes x, which stores the currently assigned value of all nodes from the root 1 to the current node v.

The only other type of non-spanning-tree edge that could exist in a connected undirected graph is *cross edge*. A cross edge is an edge between two nodes in which no one is the ancestor of another (informally, an edge between two different branches). In figure 3, the edge $(5, 6)$ is a cross edge. In the scope of this research, this type of edge does not provide any pruning opportunity. It is a tremendous headache to manage DFS progress on different branches, as all implementations of DFS only provides a natural way to keep track of the path from the root to the current node. Luckily, we won't have to deal with cross edges at all.

2.5 The DFS Tree of a graph

To maximize the efficiency of pruning, we ideally want our graph to have as many back edges as possible. The choice of G' can affect the number of back edges the graph will have. The goal of this section is to find a spanning tree structure that maximizes the number of back edges (thus minimizing the number of cross edges).

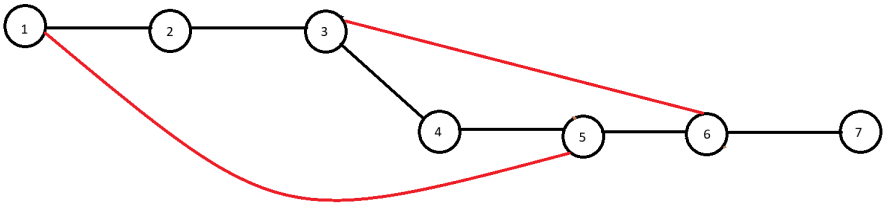


Figure 4: The graph in figure 3 can be re-structured to contain no cross edge

The DFS order produces a spanning tree of the graph, known as the *DFS tree*. In particular, the DFS tree of a graph contains only the edges used to advance the DFS exploration (i.e edges that lead a visited node to an unvisited node during DFS). Depending on the choice of root and the ordering of neighbors, the DFS tree could take many different shapes for the same graph.

However, interestingly, it has been shown that the DFS tree of an undirected graph never contains cross edges [5]. For this reason, we will choose G' to be a DFS tree of G, as it tends to maximize the efficiency of our pruning method. The effect of DFS tree structure will be study later. For now, let's assume that our DFS tree is rooted at 1 (whose value is fixed at 0). The algorithm for building the DFS tree is standard and can be found at [4].

3 Optimizing the crude DGP1 Solver

In this section, we will examine some optimizations that can be applied to our DGP1 Solver.

3.1 Bridges optimization

A *bridge* in a graph satisfies that when we remove it, the number of connected components in the graph increases (by 1). Equivalently, (u, v) is a bridge of G if all paths from u to v must pass through (u, v). For our graph G which is connected, removing a bridge will effectively make the graph disconnected. The algorithm to find bridges in an undirected graph is standard. We decided to use Tarjan's algorithm [6] in our implementation, which runs in $O(|V| + |E|)$.

Let B be the list of all bridges of $G = (V, E)$ where $V = \{1, \ldots, N\}$. Let $G'' = (V, E \setminus B)$. Let $M = |B|+1$, then G'' has M connected components. Number these connected components from 1 to M. Let I_v

be the number of the connected component that contains the node v in G''. Define a new graph $G_B = (V_B, E_B)$, where $V_B = \{1, \ldots, M\}$, E_B is such that

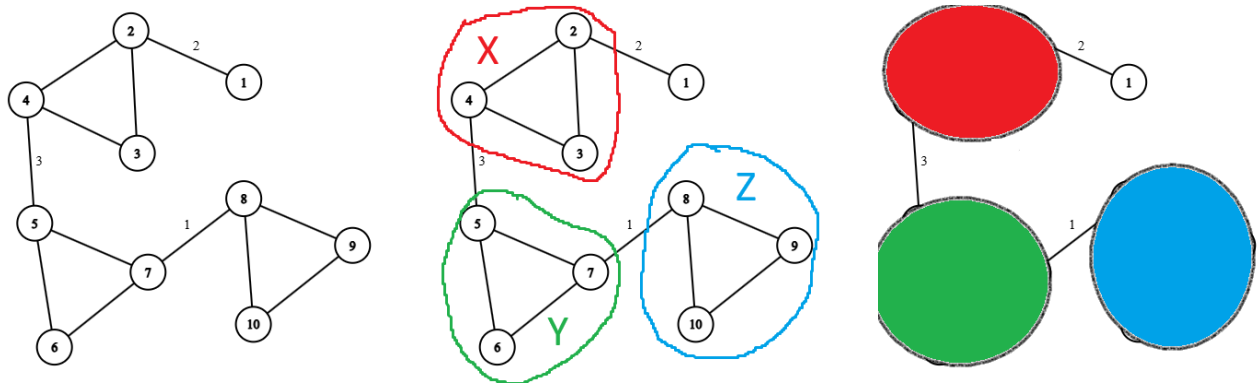
$$\forall u_B, v_B \in 1..M, (u_B, v_B) \in E_B \Leftrightarrow \exists (u, v) \in E \; s.t \; I_u = u_B \; and \; I_v = v_B \tag{5}$$

In other words, we define a new graph G_B that is the graph formed by compressing the vertices of G into its representative component number, keeping the same connectivity between components as the graph G would have provided. We notice that the edge list $E \setminus B$ provides no connection between different connected components by definition, so (7) can be further simplified to

$$\forall u_B, v_B \in 1..M, (u_B, v_B) \in E_B \Leftrightarrow \exists (u, v) \in B \; s.t \; I_u = u_B \; and \; I_v = v_B \tag{6}$$

Let d be the edge weight function of G, and d_B be that of G_B. We define d_B based on d as follows:

$$\forall (u, v) \in B, I_u = u_B, I_v = v_B \Leftrightarrow d_B(u_B, v_B) = d(u, v)$$



(a) Example graph G (only relevant edge weights will be shown)

(b) Identifying components created by removing bridges

(c) Constructing G_B

Figure 5: Example of the construction of G_B

We will prove that G_B is a tree by showing that it is connected and acyclic.

- **Connectivity**: Suppose G_B is disconnected. Then, there exists a vertex v ($1 \leq v \leq M$) that is unreachable from 1. By construction of G_B, this means that in the graph G, every node in the component number v of G'' is unreachable from every node in the component number 1 of G''. This implies that G is not connected, which contradicts our assumption. Thus G_B is connected.
- **Acyclicity**: Suppose G_B contains a cycle. Then, by construction of G_B, this implies that there exists a cycle in G that contains a bridge. This contradicts the definition of bridge, as there exists another path between two endpoints of the bridge that goes around the cycle, which does not pass through the bridge. Thus, G_B is acyclic.

We have shown that G_B is a tree. Furthermore, each edge in G_B signifies a translation constant between two connected components in G''. With this observation, we remark that DGP1 can be solved independently for each connected component of G'', which will then only need to be translated according to the acyclic distance constraints introduced by G_B. Let $x_1, \ldots, x_M$ be the realizations of the connected components. Let $x = F(x_1, \ldots, x_M)$.

Finding a valid translations is equivalent to finding the function t such that

$$\forall (u,v) \in B,\ |x(u) + t(I_u) - (x(v) + t(I_v))| = d_B(I_u, I_v)$$

$x(u)$ and $x(v)$ are already known. Let $c_{uv} = x(v) - x(u)$ be some real number. Note that $\forall (u,v),\ c_{uv} = -c_{vu}$. The function t satisfies $|t(I_u) - t(I_v) + c_{vu}| = d_B(I_u, I_v)$. This introduces two possibilities: $t(I_u) = d_B(I_u, I_v) - c_{vu} + t(I_v)$, or $t(I_u) = -d_B(I_u, I_v) - c_{vu} + t(I_v)$. The introduction of the term c_{vu} puts additional constraints to the problem of finding t, in comparison to the typical DGP1 problem.

Let r_B be a function such that

$$\forall (u,v) \in B, I_u = u_B, I_v = v_B \Leftrightarrow r_B(u_B, v_B) = c_{uv},\ r_B(v_B, u_B) = c_{vu}$$

Then the problem can be viewed as follow: each edge (u_B, v_B) in G_B specifies the desired distance between two components, while the value $r_B(u_B, v_B)$ indicates the current (signed) distance from u_B to v_B. If $t(u_B)$ is fixed, then either $t(v_B) = t(u_B) + (d_B(u_B, v_B) - r_B(u_B, v_B))$ or $t(v_B) = t(u_B) + (-d_B(u_B, v_B) - r_B(u_B, v_B))$. Thus, this introduces two possibilities for the value of $t(v_B)$ based on $t(u_B)$. `DFS_Naive` can be modified to solve the problem of finding valid translations. The exact algorithm is presented below:

Algorithm Finding valid translations in Bridges optimization

Require: graphs $G_B = (V_B, E_B, d_B)$ function r_B
  function DFS_TRANSLATIONS(v_B, val)
    Create an empty realization set $T_v = \{\}$
    Create a mapping t'
    Assign $t'(v) \leftarrow val$
    Add t' to X_v
    for each edge $(v_B, u_B) \in E_B$ where u_B is not the parent of v_B **do**
      Let $w \leftarrow d_B(v_B, u_B)$
      Let $cur\_dist \leftarrow r_B(v_B, u_B)$
      Let $rs \leftarrow \{\}$
      Let $X_+ \leftarrow$ `DFS_Translations(u, val + w - cur_dist)`
      Add X_+ to rs
      if `list-all-solutions` mode enabled **then**
        Let $X_- \leftarrow$ `DFS_Translations(u, val - w - cur_dist)`
        Add X_- to rs
      end if
      $X_v \leftarrow$ `merge`(X_v, rs)
    end for
    return X_v
  end function

As seen in section 2.3, the number of valid translations is 2^{M-1}, or $2^{|B|}$. If only one realization of DGP1 is needed, the algorithm at 3.1 will find a valid translation in $O(M)$. If many/all realizations are needed, we apply the functions `MERGE` (see pseudocode below) and `F` (see 2.3) to gather all possible solutions.

Once a valid translation t have been found, the final combined realization is produced by summing the evaluation of x with the corresponding translation value. Let x_{ans} be the combined realization, then

$$x_{ans}(u) = x(u) + t(I_u)$$

Algorithm Merge Realization Sets

Require: A set of realization maps `base` and another set `to_merge`
Ensure: A merged set (`merged`) containing all possible combinations of realizations

```
function MERGE(base, to_merge)
    Initialize empty list merged
    for each realization b ∈ base do
        for each realization tm ∈ to_merge do
            Assign cpy ← F(b, tm)
            Append cpy to merged
        end for
    end for
    return merged
end function
```

The bridges optimization enables the graph to be split into multiple independent components, on each of which one DGP1 solver instance will run. This reduces the size of the input for our DGP1 solver, as each solver instance will have to deal with lower-depth graphs with less nodes. The independence of the components also means that solving them can be *parallelized*, leading to faster runtime in multi-core systems. In a single-threaded environment (as with our current implementation), it can still be very helpful, especially if the graph contains large connected components chained together by bridges, and one component C_{fail} cannot be realized (so the answer to the DGP1 problem is "NO"). Without the bridges optimization, when trying to find an assignment for the nodes in C_{fail}, the solver will keep pruning and backtracking to the nodes in the connected components leading to C_{fail} in the DFS tree to try different values, until it backtracks to the root and decides that the graph is unsolvable. With the bridges optimization, the infeasibility of the input can be detected much earlier depending on the order that the components are solved.

Figure 6 shows an example of this, where the component $\{8, 9, 10\}$ is infeasible (the edge weights cannot correspond to any realization due to the triangle inequality). Calling DFS from 1 without the bridges optimization will lead to the solver detecting this infeasibility at the end of the DFS process, and try to backtrack up to the root only to get the same conclusion.

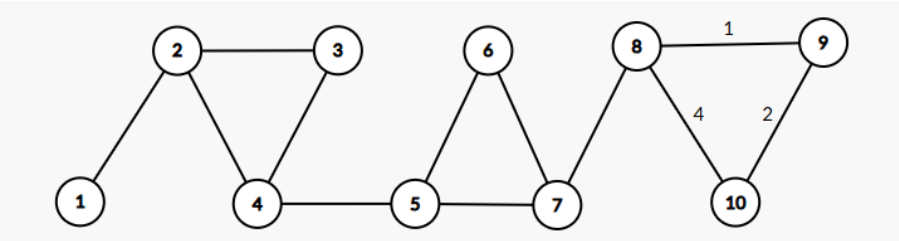


Figure 6: Example of infeasible component located at the end of DFS tree

3.2 Detecting infeasibility early with triangle inequality

Recall the extended triangle inequality in 1D Euclidean space: $\forall x_1, \ldots, x_n \in \mathbb{R}$,

$$|\sum_{i=1}^{n} x_i| \leq \sum_{i=1}^{n} |x_i| \tag{7}$$

This inequality can be used to detect quickly if the graph is infeasible. In particular, let there be a cycle of

length m in $G = (V, E, d)$, formed by the edges $e_1, \ldots, e_m$. If there exists $1 \leq i \leq m$ such that

$$2 \cdot d(e_i) > \sum_{j=1}^{m} d(e_j) \tag{8}$$

then the function d violates the triangle inequality, i.e there does not exist a valid realization for G, and we could immediately terminate the solver program.

We will study how the triangle inequality detection can be implemented efficiently. Observe that if a cycle violates the inequality, then e_i from (10) can always be chosen as the edge with the largest weight on that cycle (otherwise, we have $\forall l,\ 2 \cdot d(e_l) \leq 2 \cdot d(e_i) \leq \sum_{j=1}^{m} d(e_j)$, meaning that the cycle does not violate the inequality). With this observation, we only need to consider the largest weighted edge on any cycle to know if it satisfies the triangle inequality or not.

Given the DFS tree G' of our input graph G, every added edge (u, v) to G' creates a cycle. To check the validity of this cycle, it is sufficient to go through the edges on the simple path from u to v in G', taking the sum of their weight and keeping the largest weighted edge recorded. Finally, add $d(u, v)$ to the tracked sum and update the maximum weight edge with $d(u, v)$ if necessary. We obtain $d(e_i)$ and $\sum_{j=1}^{m} d(e_j)$, and now can check the inequality (10).

We already know all non-tree edges in G' are back edges (leading one vertex to its ancestor). Denote $par(u)$ the parent node of u if we fix the tree root at 1. Then, for every back edge (u, v) that introduces a cycle (assume u is v's ancestor), we can perform the steps above by repeatedly going along $par(v)$ and update until we reach u:

Algorithm Triangle inequality check for back edge (u, v) - naive

```
max_weight ← d(u, v)
sum ← d(u, v)
p ← par(v)
while true do
    max_weight ← max(max_weight, d(v, p))
    sum ← sum + d(v, p)
    if p = u then
        break
    end if
    v ← p
    p ← par(p)
end while
if max_weight * 2 > sum then
    Graph is infeasible
end if
```

For every back edge, the above algorithm runs in $O(|V|)$. The number of back edges G has is the number of edges of G that are not in G', which is $|E| - |V| + 1$. For large $|E|$, the above algorithm runs in $O(|V| \cdot |E|)$. We can reduce it to $O(|V|^2 + |E|) \equiv O(|V|^2)$ by pre-computing the functions sum and max_w.

$sum(u, h)$ is the sum of edges along the path from node u to its h-th ancestor (the ancestor at distance h from u; $par(u)$ is the 1-st ancestor of u). $max_w(u, h)$ is the largest edge weight along the path from node u to its h-th ancestor. This way, for every back edge (u, v), knowing the distance from u to v, we can perform triangle inequality check in $O(1)$.

For sparse graphs where $|E| \ll |V|^2$, we can use an alternate method for better runtime. Let $ancestor(u, h)$ denote the h-th ancestor of u. As taking sum and maximum is associative (i.e $max(a, b, c) = max(max(a, b), c) =$

$max(a, max(b, c)))$, we have $\forall u \in V, h_1, h_2 \in \mathbb{N}\ s.t\ ancestor(u, h_1 + h_2)$ is well defined:

$$max_w(u, h_1 + h_2) = max(max_w(u, h_1), max_w(ancestor(u, h_1), h2))$$

and

$$sum(u, h_1 + h_2) = sum(u, h_1) + sum(ancestor(u, h_1), h_2)$$

The same property applies to $ancestor$, as $ancestor(u, h_1 + h_2) = ancestor(ancestor(u, h_1), h_2)$.

With this observation in mind, let's now define $ancestor_2$, max_{w2}, sum_2 similar to before, except that the distance is now exponential. $ancestor_2(u, h)$ **is now the** 2^h**-th ancestor of** u, i.e $ancestor_2(u, h) = ancestor(u, 2^h)$. $max_{w2}(u, h)$ is the largest edge weight along the path from u to $ancestor_2(u, h)$. $sum_2(u, h)$ is the sum of edges along the same path. With this definition, the range of possible values for h is now from 0 to $log_2(|V|)$, i.e in $O(log_2(|V|))$.

For every node u, $ancestor_2(u, 0)$ is the direct parent of u. Then, for $h > 0$, we have

$$ancestor_2(u, h) = ancestor_2(ancestor_2(u, h - 1), h - 1)$$

The path-saving property of DFS allows us to compute $ancestor_2$ efficiently. We can add this calculation at the beginning of any DFS function:

Algorithm Calculate $ancestor_2$

```
function GET_ANCESTOR_2(v)
    if par(v) exists then
        ancestor_2(v, 0) ← par(v)
    end if
    for i from 1 to log_2(|V|) do
        if ancestor_2(v, i - 1) is well-defined then
            ancestor_2(v, i) ← ancestor_2(ancestor_2(v, i - 1), i - 1)
        end if
    end for
    for edges (v, u) in G' where u is not parent of v do
        get_ancestor_2(u)
    end for
end function
```

Once $ancestor_2$ is computed, sum_2 and max_{w2} are straightforward.

For sum_2, assume that all terms are well-defined:

$$sum_2(u, 0) = d(u, par(u))$$
$$sum_2(u, h) = sum_2(u, h - 1) + sum_2(ancestor_2(u, h - 1), h - 1)\ \ (h > 0)$$

For max_{w2}, assume that all terms are well-defined:

$$max_{w2}(u, 0) = d(u, par(u))$$
$$max_{w2}(u, h) = max(max_{w2}(u, h - 1), max_{w2}(ancestor_2(u, h - 1), h - 1))\ \ (h > 0)$$

We can modify any DFS function to calculate sum_2 and max_{w2}, similar to how we calculated $ancestor_2$. Refer to [4] for the detailed implementation. Overall, computing $ancestor_2$, sum_2 and max_{w2} takes $O(|V|log_2(|V|))$.

Now, let (u, v) be a back edge where u is the k-th ancestor of v (i.e, $u = ancestor(v, k)$, not to be confused with the function $ancestor_2$). We can decompose k into sum of powers of 2, as if converting it to binary representation. Let $p_1, \ldots, p_l \in \mathbb{N}$ and pairwise distinct such that

$$k = \sum_{i=1}^{l} 2^{p_i}$$

Then the maximum weight edge and the sum of weight of edges along the path $u \to v$ can be obtained from this decomposition. In general, for any associative operation $\otimes$, let $\bigotimes(u, k)$ be the application of $\otimes$ on all edges on the path from u to its k-th ancestor. Then,

$$\begin{aligned}
\bigotimes(u, k) &= \bigotimes(u, \sum_{i=1}^{l} 2^{p_i}) \\
&= \bigotimes(u, 2^{p_1}) \\
&\otimes \bigotimes(ancestor(u, 2^{p_1}), 2^{p_2}) \\
&\otimes \bigotimes(ancestor(u, 2^{p_1} + 2^{p_2}), 2^{p_3}) \\
&\otimes \cdots \otimes \bigotimes(ancestor(u, \sum_{i=1}^{l-1} 2^{p_i}), 2^{p_l})
\end{aligned}$$

Let $\bigoplus(u, p)$ be the application of $\otimes$ on all edges on the path from u to its 2^p-th ancestor. We can rewrite the terms in the expressions above as

$$\begin{aligned}
\bigotimes(u, k) &= \bigoplus(u, p_1) \\
&\otimes \bigoplus(ancestor_2(u, p_1), p_2) \\
&\otimes \bigoplus(ancestor_2(ancestor_2(u, p_1), p_2), p_3) \\
&\otimes \cdots \otimes \bigoplus(ancestor_2(\ldots(ancestor_2(u, p_{l-1}))), p_l)
\end{aligned}$$

As any natural number k can be represented using at most $\lfloor log_2(k) + 1 \rfloor$ bits, we have $l \leq \lfloor log_2(k) + 1 \rfloor$. Thus, there are $O(log_2(k))$ operands in the right hand side of the expression above. Furthermore, each operand can be evaluated in $O(1)$ by knowing the operand before it, since $ancestor_2$ is already pre-computed. This implies that calculating $sum(u, k)$ or $max_w(u, k)$ only takes $O(log_2(k))$ applications of sum_2 or max_{w2}, which is consequently $O(log_2(k))$. This technique is known as *binary lifting*. Refer to [4] for the detailed implementation.

Finally, a back edge (u, v) can be checked for triangle inequality using sum_2 and max_{w2}. This detection method runs in $O(|E|log_2(|V|))$. For dense graph with a lot of edges, the $O(|V|^2)$ method mentioned earlier might run faster. However, we decided to use this method in our implementation, as it works better for most input graphs, and naive branch-and-prune already works very well on very dense graphs (where there are many back edges).

Algorithm Triangle inequality check for back edge (u, v) - optimized

1: Assume v is the k-th ancestor of u (if not, simply swap the two vertices)
2: $max\_weight \leftarrow max(d(u, v), max_w(u, k))$
3: $path\_sum \leftarrow d(u, v) + sum(u, k)$
4: **if** $max\_weight \cdot 2 > path\_sum$ **then**
5: Graph is infeasible
6: **end if**

3.3 General branch-and-prune method for detecting infeasibility

Another way to examine the feasibility of an input graph G to DGP1 is to examine it cycles. If there exists a cycle in G that is infeasible, then the entire graph is infeasible. However, due to the selection of the DFS tree root and DFS order, such a cycle might be detected very late into the DFS process. Even when the cycle is detected, our current implementation will try all 2^n assignments for the cycle (let n be the size of the cycle), before backtracking to the node leading to the cycle, trying a different value for that node, and trying to hopelessly find a suitable assignment for the cycle again.

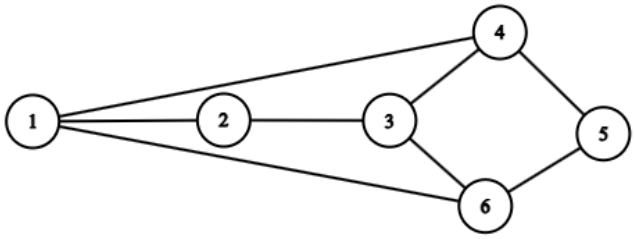


Figure 7: If the DFS starts from 1 and the cycle $3 \to 4 \to 5 \to 6 \to 3$ is infeasible, the current implementation would backtrack and re-assign the value of nodes 2 and 1 before concluding that the graph is unsolvable. We would like to terminate the solver as soon as the infeasibility of the cycle is confirmed.

This issue is similar to the one mentioned in 3.1, although it can exist for graphs with no bridges as well. Figure 7 gives an example of such graph. This section discusses a general optimization for this case.

The scenario we want to detect is when the DFS solver wants to assign a value to a node, but we already know that there exists an infeasible cycle somewhere in the sub-tree of that node, i.e an infeasible cycle where the node with least depth still lies below the current node . In that case, the solver can terminate immediately with answer "NO".

Fortunately, we can use a global or static variable to efficiently keep track of this information. Let's conventionally assume that the DFS tree root's depth is 0. Let there be a global variable called low, initialized to -1. Every time we try to prune the search tree with a back edge, we record the contradictory back edge that leads to the deepest node possible, and update low to be the depth of that node. This node corresponds to the node with least depth in an infeasible cycle, as described in the previous paragraph.

Now, every time the DFS manages to find a valid partial assignment for the current sub-tree, we know that the sub-tree is not infeasible and thus reset low to -1. However, if the DFS enters a node with depth not greater than low, we know that this node will eventually lead to an infeasible cycle anyway, so the answer is immediately "NO".

Algorithm General infeasibility detection

```
function DFS(v, val)
    Declare static variable low ← −1
    if dep(v) ≤ low then
        Graph is infeasible
    end if
    ...
    if X_v ≠ ∅ then
        low ← −1
    end if
    return X_v
end function
```

3.4 Speeding up the discovery of mirror realizations

At each step in the DFS process, either the positive branch ($x(u) = x(v) + w$) or the negative branch ($x(u) = x(v) - w$) is chosen.

Once we find a realization to the DGP1 instance, we can find its mirror realization by flipping the sign of the each value in the assignment. Analogously, this corresponds to taking the opposite branch at every point in the DFS process. We can thus assume that the first branch taken in the DFS process is always the positive one. This does not affect how long it takes to find the one solution, but halves the time it takes to list all solutions.

From now on, we assume that the first branch taken from the DFS root is positive, i.e the first node that is visited after node 1 is always assigned a value greater than 0.

3.5 Using randomized branching to add variance to runtime

Thanks to the possibility of pruning, our DGP1 solver in practice works much faster than its theoretical $O(2^{|V|})$ upper bound runtime. However, it still struggles a lot when there are little to no pruning opportunities. Take for example graph that consists of a single cycle of the form $1 \to 2 \to \cdots \to n \to 1$. The validity of an assignment can only be checked when the DFS reaches the final node of the graph (assuming it starts from 1, then the final node is n).

Our algorithm 2.4 is currently always trying the positive branch first, then the negative branch after. If the valid realization can only be obtained by choosing the negative branch at some early point in the DFS process, the solver will fail to find a valid assignment until it backtracks way up to near the root. This leads to a practical $O(2^n)$ runtime.

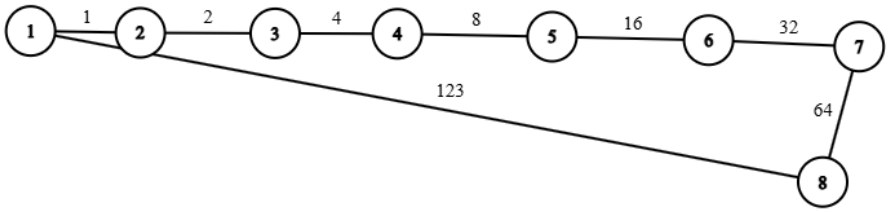


Figure 8: Starting the DFS at node 1, the only way to make a valid assignment is taking the negative branch when moving from node 2 to node 3.

To counter this behavior, we can modify the algorithm such that branch selection is randomized, i.e there is a 50% chance that the positive branch is taken, and 50% chance the negative branch is taken. This modification essentially makes the runtime of the solver a random variable, with potentially very high variance. With the example in figure 8, it would run in $O(n)$ for the best case (the randomization picks the correct branch every time), and $O(2^n)$ for the worst case (the wrong branch is picked every time).

Assume the input graph is feasible. If multiple DGP1 solvers with branch randomization are launched (e.g. by multi-processing), the possibility that one solver makes decent branching decisions increases. We could consider the following problem to gauge this increase:

> Given a target string in $\{+, -\}^n$ and an algorithm that generates a random string in $\{+, -\}^n$, how many strings we should generate to be confident enough that there exists at least one string that mismatches the target string at no more than m positions?

Note that this problem is not equivalent to the one we are solving, but similar in the sense that good branching decisions will generally result in less mismatches.

The number of mismatches in a string follows a binomial distribution: $X \sim Bin(n, 0.5)$. The probability of having no string satisfying the mismatches condition after k trial is $(1 - P(X \leq m))^k$. Assuming 95%

confidence is enough, we simply have to find the smallest k satisfying $(1 - P(X \leq m))^k \leq 0.05$, which gives

$$k \geq \frac{ln(0.05)}{ln(1 - P(X \leq m))}$$

Combining this optimization with the next one will tremendously help solving DGP1 on cycle graphs like the one in figure 8.

Algorithm DGP1 solver with back edge pruning

function DFS(v, val)
...
Let $w \leftarrow d(v, u)$
Let $rs \leftarrow \{\}$
Flip the sign of w **with a chance of** 0.5
Let $X_+ \leftarrow$ `DFS(u, val + w, x)`
...
end function

3.6 Pruning the search tree with subset sum optimization

Consider a path from u to v where u is the ancestor of v. The set of possible values for v is directly a function of the set of all possible sums formed from edges on the path $u \to v$. Specifically, let $w_1, \ldots, w_k$ be the edge weights on this path. By inserting either $+$ or $-$ before each term w_i, we form one such sum*. This intuition leads to an idea of using the set of reachable values from one node to quickly detect infeasibility of the (partial) assignment.

This detection can be separated into two parts: pre-DFS and during-DFS. For pre-DFS detection, we check all cycles in the graph (formed by adding back edges to the DFS tree). For each cycle, we check if the weight of the back edge in the cycle can be obtained from the remaining edges using the same method described in *.

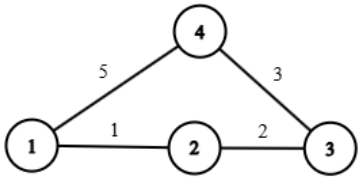


Figure 9: There is no placement of $+$ and $-$ to form the value 5 from 1, 2 and 3, so pre-DFS detection can detect that this cycle is infeasible, and thus the answer is "NO"

For during-DFS detection, we keep track of all cycles the current node is a part of during DFS, and check if it is still possible to fulfill the constraints created by the cycles from the current assignment by considering the set of all sums reachable. To do this, we can consider only the back edge (u, v) of each cycle. As we already have the value of the ancestor node u, node v can have two possible values. We can check if the difference between the value of the current node in DFS and the target value of v can be formed from the edges on the path from the current node to v.

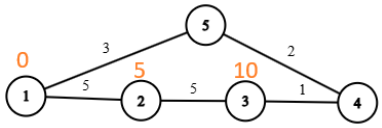


Figure 10: The DFS is at node 3 with the current node assignment in orange. Node 5 can only take value -3 or 3. The remaining edges would have to form a sum of either -13 or 7 for this assignment to be feasible. But there is no placement of $+$ and $-$ to form either of them, so the during-DFS detection discards the current assignment.

By considering all sums reachable, we can efficiently determine if the graph or the current assignment is infeasible, without having to do any further DFS! This section will study how that can be done efficiently.

Let $ss(u, k)$ denote the set of all possible sums formed from the path from u to its k-th ancestor. We have

$$ss(u, 1) = \{d(u, par(u)), -d(u, par(u))\}$$
$$ss(u, k) = \{a \pm d(ancestor(u, k-1), ancestor(u, k)) \mid a \in ss(u, k-1)\}$$

So ss can be easily computed during DFS. We have $u \in V$, $k \in 1..|V|$, and $ss(u, k)$ can contain up to 2^k elements. So if we were to implement ss this way, it would take $O(|V|^2 \cdot 2^{|V|})$ time and space, which is impractical.

Due to the exponential size of $ss(u, k)$, this optimization can only be limit the value of each node to a reasonable range (or if we decide to only look for solutions in such range). Let's assume that each value in our solution can only fall into the range $[-M, M]$, the calculation of ss above will then take $O(|V|^2 M)$ time and space.

Let H be a function that combines two subsets of $\mathbb{R}$ as follows:

$$H(A, B) = \{a + b \mid a \in A,\ b \in B\}$$

Then we can implement H by iterating through all pairs in $A \times B$, resulting in an $O(|A| \cdot |B|)$ runtime.

Using *binary lifting* (as seen in 3.2), we can define the function ss_2, which is similar to ss but works with exponential height: $ss_2(u, k)$ **is the set of all reachable sums on the path from** u **to its** 2^k**-th ancestor**. We have

$$ss_2(u, 0) = \{d(u, par(u)), -d(u, par(u))\}$$
$$ss_2(u, k) = H(ss_2(u, k-1), ss_2(ancestor_2(u, k-1), k-1))$$

So ss_2 takes $O(|V| log_2(|V|) M)$ space. We establish a very loose (and obvious) bound of $O(|V| log_2(|V|) M^2)$ for its time complexity, but real-life testing shows that it is much faster. It remains a topic of study to find a tighter bound of its time complexity, but running this calculation across multiple tests shows that calculating ss_2 is at least as fast as ss, which is $O(|V|^2 M)$.

Once ss_2 is computed, we can use ss_2 to compute any value in ss in $O(log_2(|V|))$ rounds of applying H on two subsets, resulting in an upper bound of $O(log_2(|V|) M^2)$ time to compute $ss(u, k)$, but practical testing again shows that this is much faster. The formula is very similar to 3.2. For example, if $k = 11 = 2^0 + 2^1 + 2^3$, then

$$ss(u, 11) = H(H(ss_2(u, 0), ss_2(ancestor_2(u, 0), 1)), ss_2(ancestor_2(ancestor_2(u, 0), 1), 3))$$

Now, observe that for low values of k, the set $ss(u, k)$ will very likely contain the full 2^k values, as there are simply not enough summands for repetitions to happen. In such case, calculating ss will take no less than $O(2^k)$ time. Therefore, it would actually be better to just ignore this optimization for low values of k, as $O(2^k)$ is already the worst case runtime for DFS, while DFS also gives the valid assignment rather than just whether

the assignment can be found. Especially if randomized branching is also used, the worst case of DFS will almost never happen in all the solver instances.

Our implementation thus set a `MIN_LOOKAHEAD_DEPTH` variable, which dictates the minimum value of k to apply this optimization. We left the choice to the user, but suggest taking `MIN_LOOKAHEAD_DEPTH` at around $log_2(|V|)$.

In theory, this optimization works for floating point values, but floating point arithmetics are much slower than integer ones. Furthermore, C++ enables a very interesting optimization option for integer weights using `std::bitset`. Instead of storing the set of all reachable sums for $ss_2(u, k)$, we store a bit array where the i-th bit is 1 if it is possible to obtain sum i, and 0 otherwise. Now, we only need one bit to indicate a reachable sum, compared to (typically) 32 bits to represent the sum itself. This effectively saves the memory usage for the same M by 32 times, and increases the performance substantially, as the function H can be implemented using bit-shifts (which are highly optimized) instead of having to iterate through all pairs from both sets. As bit arrays can only be indexed from 0, we use a value $OFFSET$ to support negative values, as in the pseudocode below. We also use a value $WINDOW$ (set to $4 \cdot M$) to specify the length of the range that our bit array should keep track of.

Algorithm Function H using `std::bitset`

```
function H(A, B)
    if Number of set bits in A is less than B then
        Swap A and B
    end if
    Initialize res as an empty bitset
    idx ← First set bit in B
    while idx is a valid index in bitset do
        shift ← idx − OFFSET
        if 0 ≤ shift < WINDOW then
            res ← res ∨ (A ≪ shift)
        else if shift < 0 and |shift| < WINDOW then
            res ← res ∨ (A ≫ |shift|)
        end if
        idx ← Next set bit in B
    end while
    return res
end function
```

4 Choosing the optimal structure for the DFS Tree

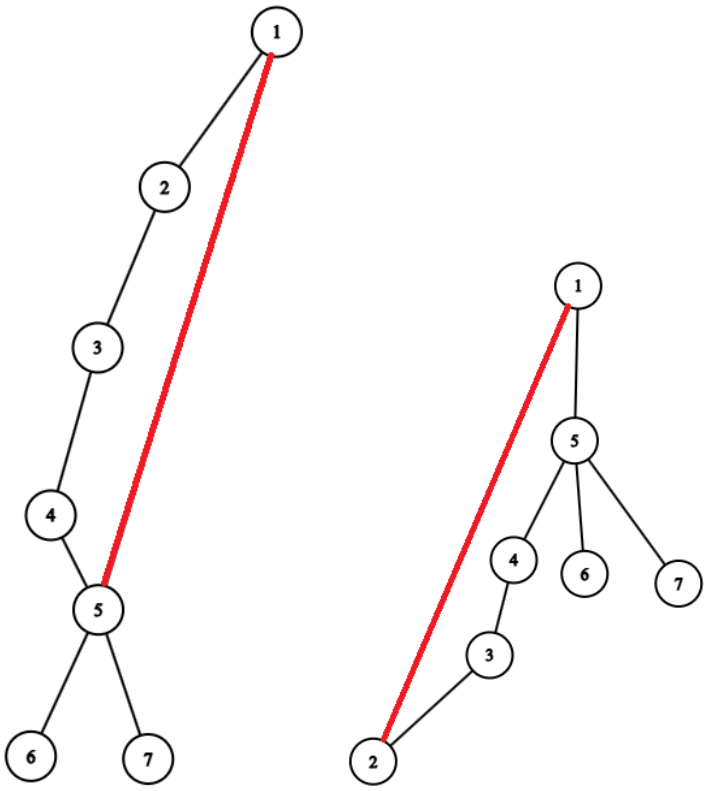
The choice of the root and neighbor visiting order drastically changes the structure of the DFS tree, which directly affects the performance of our DGP1 solver. This section studies how to optimize the shape of the DFS tree to facilitate our program.

The ideal DFS tree should have:

- Minimum height possible. A shallow DFS tree means less DFS recursions. This leads to less recursion overhead, shorter DFS chains, and directly improve the performance of depth-based optimizations in the previous part, such as 3.2 and 3.6.
- Many pruning opportunities around the root. The DFS root should be in the cycle-rich region of the graph. Finding new cycles deep down in the graph should be avoided.

There is currently no known deterministic method to find the minimum height DFS tree, except for trying all roots and all neighbor traversal orders. This takes a significant amount of time, so heuristics are usually preferred. A common heuristic is to prioritize nodes with higher degree, i.e nodes that has more connections. Figure 11 shows the DFS trees of the same graph, choosing 1 as the root. Observe that the tree that prioritizes higher degree neighbor (on the right; visiting node 5 from 1 first) is more shallow than the one that does not prioritize higher degree neighbor (on the left; visiting node 2 from 1 first). Higher-degree node heuristic is very straightforward to implement: we only need to go through all nodes in V, choosing the one with the highest degree as the root. Then, to determine the neighbor traversal order, for every node, we sort the list of nodes adjacent to itself also based on degree, putting the ones with higher degree first.

Figure 12 shows two DFS trees of the same graph, with the one on the right choosing to prioritize nodes that belong to more cycles. Prioritizing nodes in more cycle-rich regions of the graph, on the other hand, can be done using deterministic methods. Take an arbitrary DFS tree of the graph. Let $c(u)$ denote how many cycles the node u is part of. Then, for each added cycle (by a back edge (u, v)), we add 1 to the value $c(i)$ of all nodes i on the path $u \to v$. After having full information about c, we can discard the current DFS tree and build a new one satisfying the desired properties. We simply take the node u that maximizes $c(u)$ as the root. To determine the neighbor traversal order, for every node, we also sort the list of nodes adjacent to itself based on c.



(a) No priority (b) Prioritizing higher degree neighbor

Figure 11: DFS trees of the same graph. Rooted at 1 but with different neighbor traversal order. Back edges are marked in red.

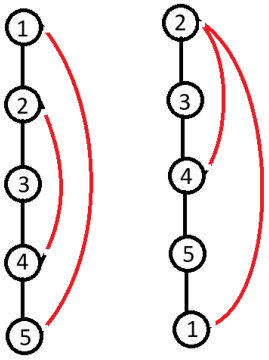


Figure 12: The DFS tree on the right is rooted at the node that is part of more cycles. Choosing this root makes pruning happen earlier during DFS (first pruning at depth 2), compared to the root choice of the DFS tree on the left (first pruning happens at depth 3).

A naive algorithm for updating $c(i)$ takes $O(|E| \cdot |V|)$, similar to the naive approach for detecting triangle inequality in 3.2. We can improve this runtime to $O(|E|(log_2(|V|))^2)$ using *heavy-light decomposition* and *segment tree* for range update [7]. However, the updates to $c(i)$ need not to be done online, i.e we can save all the back edges creating cycles and group them to perform the updates efficiently. With this observation, we propose a simple but very interesting algorithm to calculate c in just $O(|E|log_2(|V|))$:

Algorithm Calculate c

Require: DFS tree adjacency list `dfs_tree_adj`, list of back edges `back_edges`, parent array `parent`, depth array `dep`, DFS tree root node `dfs_root`, number of nodes $n = |V|$
Ensure: Full information of c
  `psum` $\leftarrow$ array of size n, initialized to 0 (1-indexed)
  `c` $\leftarrow$ array of size n, initialized to 0 (1-indexed)
  `leaves` $\leftarrow$ empty set, prioritize nodes by depth.
  for $i = 1$ to n **do**
    if size(`dfs_tree_adj`$[i]) = 1$ and $i \neq$ `dfs_root` **then**
      Insert i into `leaves`
    end if
  end for
  for each (i, j) in `back_edges` **do**
    $psum[i] \leftarrow psum[i] + 1$
    if $parent[j] \neq -1$ **then**
      $psum[parent[j]] \leftarrow psum[parent[j]] - 1$
    end if
  end for
  while `leaves` $\neq \emptyset$ **do**
    Select and remove u with highest $dep[u]$ from `leaves` (this takes $O(log_2(n))$)
    $c[u] \leftarrow c[u] + psum[u]$
    if $parent[u] \neq -1$ **then**
      $c[parent[u]] \leftarrow c[parent[u]] + c[u]$
      Insert $parent[u]$ into `leaves`
    end if
  end while
  return c

5 References

[1] Wikipedia contributors. Distance Geometry – Wikipedia, The Free Encyclopedia.

[2] Leo Liberti, Carlile Lavor, Nelson Maculan, and Antonio Mucherino. Euclidean distance geometry and applications. *SIAM Review*, 56:3–69, 2014.

[3] James B. Saxe. Embeddability of weighted graphs in k-space is strongly np-hard. In *Proceedings of the 17th Allerton Conference on Communication, Control, and Computing*, pages 480–489. University of Illinois, Urbana-Champaign, 1979. Also available as Carnegie-Mellon University Technical Report CMU-CS-79-115.

[4] Ha Duy Nguyen. Dgp1 solver. https://github.com/nhdtxdy/DGP1-solver/blob/main/solver.cpp, 2025.

[5] James Aspnes. Depth-first search notes. https://www.cs.yale.edu/homes/aspnes/pinewiki/DepthFirstSearch.html, 2024.

[6] CP-Algorithms. Finding bridges in a graph. https://cp-algorithms.com/graph/bridge-searching.html, 2025.

[7] CP-Algorithms. Heavy-light decomposition, 2024.

A Appendix

A.1 P, NP, NP-Complete, NP-hard

In computational complexity theory, decision problems are classified based on the resources required to solve them. The most relevant classes for our discussion are:

- **P (Polynomial Time)**: The class of decision problems that can be solved by a deterministic Turing machine in polynomial time, i.e., in $O(n^k)$ for some constant k.
- **NP (Nondeterministic Polynomial Time)**: The class of decision problems for which a given solution can be verified in polynomial time by a deterministic Turing machine. Formally, a language L is in NP if there exists a polynomial-time verifier $V(x, y)$ such that:

$$x \in L \iff \exists y, |y| = O(|x|^c) \text{ such that } V(x, y) \text{ runs in } O(n^k). \tag{9}$$

- **NP-Complete (NPC)**: A problem L is NP-complete if:
  1. $L \in$ NP.
  2. Every problem $L' \in$ NP can be reduced to L in polynomial time, i.e., there exists a polynomial-time computable function f such that:

$$x \in L' \iff f(x) \in L. \tag{10}$$

  NP-complete problems are the hardest problems in NP, meaning that if any NP-complete problem can be solved in polynomial time, then P = NP.
- **NP-Hard**: A problem is NP-hard if it is at least as hard as the hardest problems in NP, meaning every NP problem can be reduced to it in polynomial time. However, NP-hard problems may not themselves belong to NP (i.e., they may not be decision problems or have efficiently verifiable solutions).

A.2 The AMPL .dat format

Inputs to the (general) DGP problem are given as AMPL .dat files. As a result, our DGP1 solver takes in the same file format.

For a graph $G = (V, E, d) with$V = {1,2,3,4},E = {{1,2}, {2,3}, {3,4}, {1,4}}, and $d(1, 2) = 1, d(2, 3) = 1, d(3, 4) = 1, d(1, 4) = 3$, we have the file graph.dat as follows:

```
# DGP1 instance in AMPL .dat format
# graph.dat
param n := 4 ;
param : E : c I :=
  1 2  1.000 1
  2 3  1.000 1   # this denotes the fact that the edge {2,3} has weight 1.000
  3 4  1.000 1
  1 4  3.000 1
;
```

Figure 13: An example AMPL .dat file

- "#" introduces comments: part of lines including and after the character "#" should be ignored, note that a comment may come mid-line, as in the example

- the parameter n is the number of vertices in the graph, always labeled as $\{1, ..., n\}$;
- d is encoded as c
- there is a parameter "I" (capital "i") which is always set to 1: it is unused for DGP1.
- tokens are space-delimited.

A.3 Graph splitting algorithm in `splitutil`

Algorithm Graph splitting

Require: Undirected weighted graph $G = (V, E, d)$
Ensure: A set of files, each containing a connected component of G, relabeled from 1.
Read G from the input file, extracting $n = |V|$ and edge set E.
Initialize a DSU structure for n elements.
for each edge $(u, v) \in E$ **do**
`Union`(u, v) in DSU
end for
Group vertices by their connected component (DSU root).
Initialize `compNum` $\leftarrow 0$ ▷ Component counter
for each component C_i in the DSU grouping **do**
`compNum` $\leftarrow$ `compNum` $+ 1$
Sort the vertices in C_i.
Create a mapping `newId`(v) from old vertex IDs to 1-based new IDs.
Extract edges within C_i, replacing old IDs with `newId`.
Generate output filename `graph_comp_compNum.dat`.
Write C_i to the output file in AMPL .dat format.
end for
return A set of component files {`graph_comp1.dat`, `graph_comp2.dat`, `...`}