UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

Graph Isomorphisms and Homeomorphisms for Part and Assembly Matching: Distributed System and Provenance with Blockchain Technology

A THESIS

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

Degree of

MASTER OF SCIENCE

By

Dwaine Kenney
Norman, Oklahoma
2020

Graph Isomorphisms and Homeomorphisms for Part and Assembly

Matching: Distributed System and Provenance with Blockchain Technology

A THESIS APPROVED FOR THE
SCHOOL OF COMPUTER SCIENCE

BY THE COMMITTEE CONSISTING OF

Dr. Sridhar Radhakrishnan, Chair

Dr. Christan Grant

Dr. Shivakumar Raman

Abstract

In recent times, the democratization of manufacturing through various developments in the industry, such as 3-D printing and crowdsourcing, has led to increasing levels of innovation and a greater ability for smaller organizations and even individual entrepreneurs to participate in the manufacturing process. One problem to be solved with the advent of the democratization of manufacturing is how a manufacturer can determine if a required part is already being manufactured by another manufacture exactly or with sufficient similarity without having to manually search through existing parts.

In this thesis, we present algorithms for searching existing parts and assemblies for a pattern specification, and we propose some distance measures for determining how similar two matched part or assembly trees are based on node attributes. Furthermore, we present a distributed system that can be used to search parts and assemblies created by other manufacturers, make purchases after receiving results with respective distances, and track provenance of parts used in assemblies by third-party manufacturers using blockchain technology, all without requiring a centralized authority or database.

Acknowledgements

First, I wish to thank my research advisor, Dr. Sridhar Radhakrishnan, for his guidance and support throughout the development of this thesis, as well as for his love of computer science and dedication to his students.

Additionally, I would like to thank my students in his research group, Sudhindra Gopal, Aditya Narasimhan, Jonathan Leslie, Aaron Moris, Addison Womack, Michael Nelson, and Dorian Selimovic, for providing their insight and support throughout my time doing research.

Finally, I would like to thank my family for their support and love throughout my education at the University of Oklahoma.

Contents

Chapter 1

Introduction

1.1 Motivation

As of 2018, manufacturing was a $14.17 trillion industry worldwide in terms of value added [6]. However, the industry is distributed quite unevenly geographically; four countries, China, the United States, Japan, and Germany alone account for more than half of this figure ($7.99 trillion). The concentration continues when one looks to individual corporations; in 2017, the 100 largest manufacturing companies worldwide generated a total of $7.07 trillion in revenue, though it is worth noting that some of these companies, such as Apple and Dell, provide services in addition to manufacturing products which is not accounted for in this figure [4]. In addition, concentration of the manufacturing industry in the United States has increased steadily since 2000 with respect to the market share of the largest eight firms in the industry [3].

This concentration of the manufacturing industry both in the United States and globally presents several problems. As noted by former chair of Council of Economic Advisors Jason Furman, high concentration within an industry can reflect "high economic rents and barriers to competition." [5]. In addition to these problems, he further notes that greater concentration can lead to stifled innovation within an industry. In the manufacturing industry, lower barriers for entrepreneurs into the industry could serve to lessen the effects of these problems.

A recent trend in the manufacturing industry has come with developments in areas such as crowd-funding, crowdsourcing, and 3D-printing. The term *democratization of manufacturing* refers to this trend of increasing ability for smaller organizations and individuals unaffiliated with larger manufacturing companies to enter into the

manufacturing market, both in the sense of being able to compete in the industry and having the ability to obtain increasingly specialized products from other manufacturers [7].

With the advent of the democratization of manufacturing, an interesting problem is to determine how a manufacturer or other purchaser may determine if a part they need is already being manufactured by another manufacturer, and if not, determining how similar existing parts are to the necessary specification. Extending this problem further, it would be useful for a manufacturer looking to create some part from smaller, less complex components to determine which of the smaller components can be sourced from other manufacturers and combined together. Furthermore, once a manufacturer has agreed to sell a part they have created to some other party for use in their own manufactured component, the problem arises of how to guarantee that provenance of the original part is honored in the new manufactured component without requiring a centralized authority to maintain that information.

1.2 Parts and Assemblies as a Tree

Consider the assembly pictured in Figure 1.1, created using SolidWorks. Observe that the assembly has been visually exploded to allow for an observer to see constituent parts and sub-assemblies, many of which can be further decomposed into smaller assemblies. We can imagine that a manufacturer or designer may wish to search for assemblies that are similar to any of these sub-assemblies (or possibly the entire assembly). With this in mind, the first problem is to consider how the assembly can be represented to more intuitively allow it to be searched by a user.

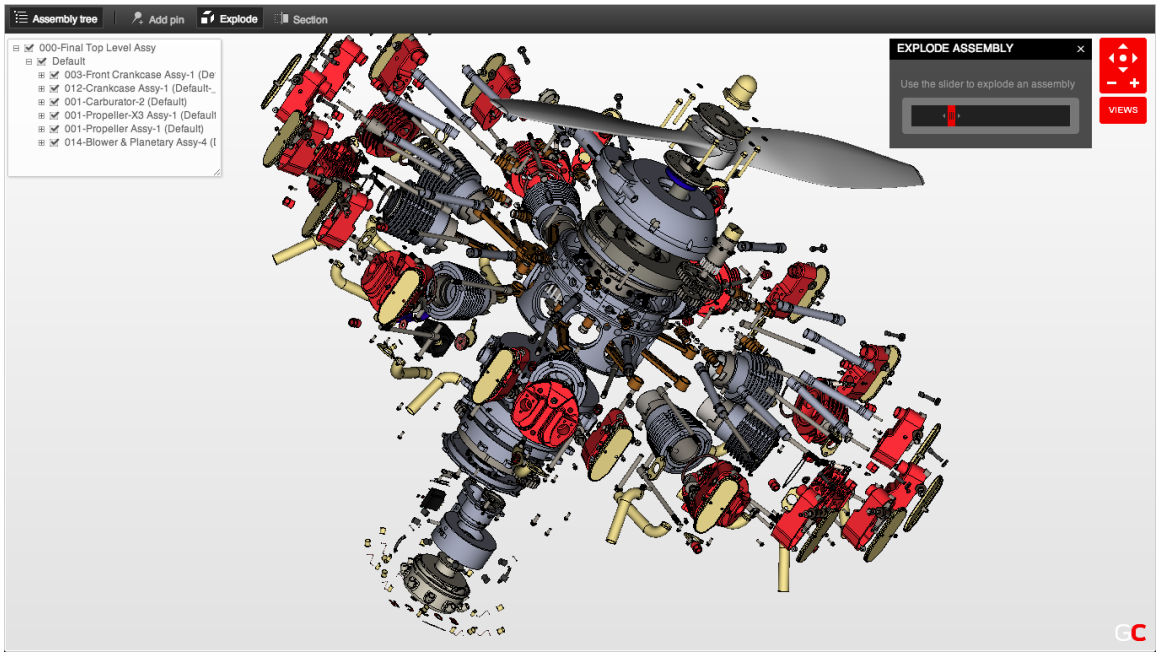

Figure 1.1: A complex assembly with many constituent parts and sub-assemblies. Sourced from `https://blog.grabcad.com/blog/2013/04/15/finally-your-favorite-3d-files-wherever-you-go//`. Accessed April 20, 2020.

To solve this problem, observe in Figure 1.2 that assemblies can be represented as a rooted tree. Note that a complete assembly is represented at the root node of this tree, with children nodes representing smaller, constituent assemblies, with indivisible parts at the leaf nodes of the tree. Consider again the radial engine shown in Figure 1.1. If we are to consider this device as a rooted tree, we can imagine that the root node of the tree corresponds to the entire device, perhaps with some node attributes containing consumer information about the device, such as its name or price, as well as possibly information about the construction and makeup of device, such as its weight, density, heat tolerance, etc. The children of the root nodes then will correspond to smaller components that may constitute parts in their own right, with each node having some unstructured set of attributes that give further information about the part. This hierarchy of increasingly smaller parts as children of larger ones can continue down to the smallest components of the engine, such as screws or wires, which themselves may have children to outline their features, such as the thread and head of a screw.

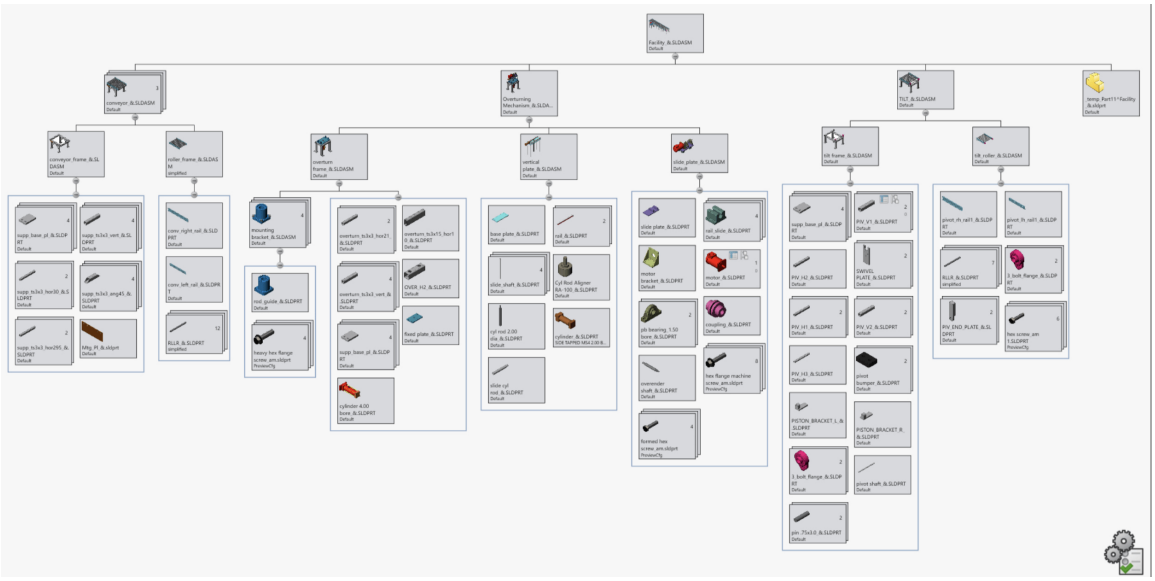

Figure 1.2: An assembly represented as a rooted tree, with the complete assembly represented by the root node of the tree. Sourced from `https://www.innova-systems.co.uk/solidworks-treehouse-tutorial/`. Accessed April 19, 2020.

Any part or assembly can be represented as such a rooted tree, which we will refer to as an *assembly tree*. Assembly trees should have some information to describe the characteristics of the assembly as well as its constituent parts and sub-assemblies. In order to provide this information, each node will contain some unstructured attributes; the attributes are unstructured in that any two nodes may not have the same number of attributes, and attributes may describe different characteristics of their respective parts and assemblies. It is also important in this research that attributes be useful for someone who may be searching for a part using specific constraints; each node's attributes should demonstrate tolerances, limits, operating requirements, and other specifications that may be important to someone looking for a specific part or sub-assembly to use in their own assembly.

Once a manufacturer has created a tree structure for some set of parts and assemblies that they have currently available to be manufactured, a searcher can create their own tree that sets up the requirements they have for a part or assembly they are looking to use and query that manufacturer's database of part trees to determine if any part, assembly, or section of a part or assembly matches what they are looking for to a near enough degree.

1.3 Assembly Search in a Distributed System

Once manufacturers have created their database containing part trees for parts and assemblies they manufacture, we can image a system that allows them to search and reuse each other's parts and assemblies. Such a system, called GrabCAD, already exists. As shown in Figure 1.3, GrabCAD connects a community of millions of professional designers and manufacturers with a library consisting of millions of assemblies. However, this system presents two problems. The first is a limited search capability, where searching for assemblies can be done using a few simple properties, such as keywords and category; adding deeper search functionality can be accomplished by searching through rooted tree representations of assemblies, as discussed in Section 1.2.

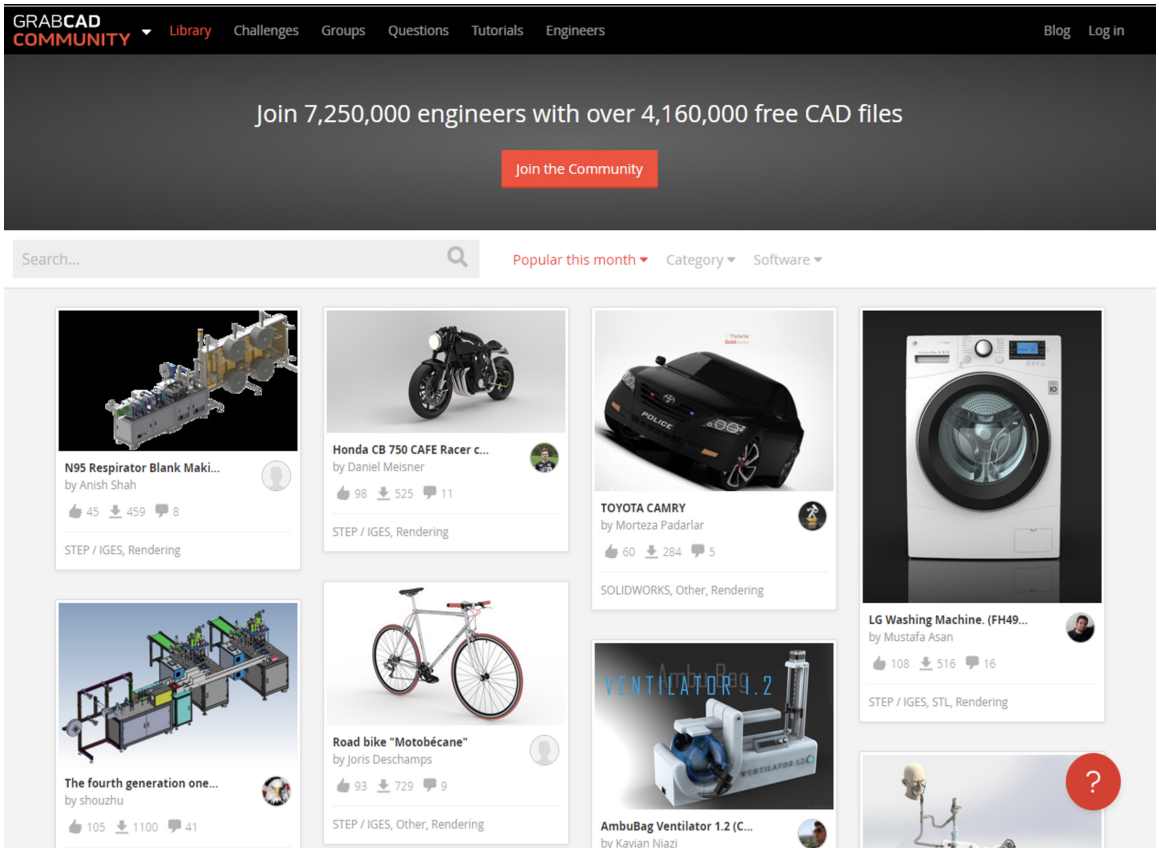

Figure 1.3: The GrabCAD library. Sourced from `https://grabcad.com/library`. Accessed April 20, 2020.

A second problem is that GrabCAD acts as a centralized database, where manufacturers and designers must share designs through a single database, rather than

communicating with each other directly; this presents a single point of failure for the system. To solve this problem, a peer-to-peer network consisting of a node for each participating manufacturer can be used. In this network, each participant, whether they be a manufacturer, designer, or engineer, stores their design files at their own database, as shown in Figure 1.4; as such, each participant node acts as both a client and a file server. When manufacturer A wants to buy a part or assembly based on some part tree specification, they can query the database of part trees at each other node, which in turn will respond with results containing matching part trees and some measure of similarity of each resulting part tree to the specification provided by A.

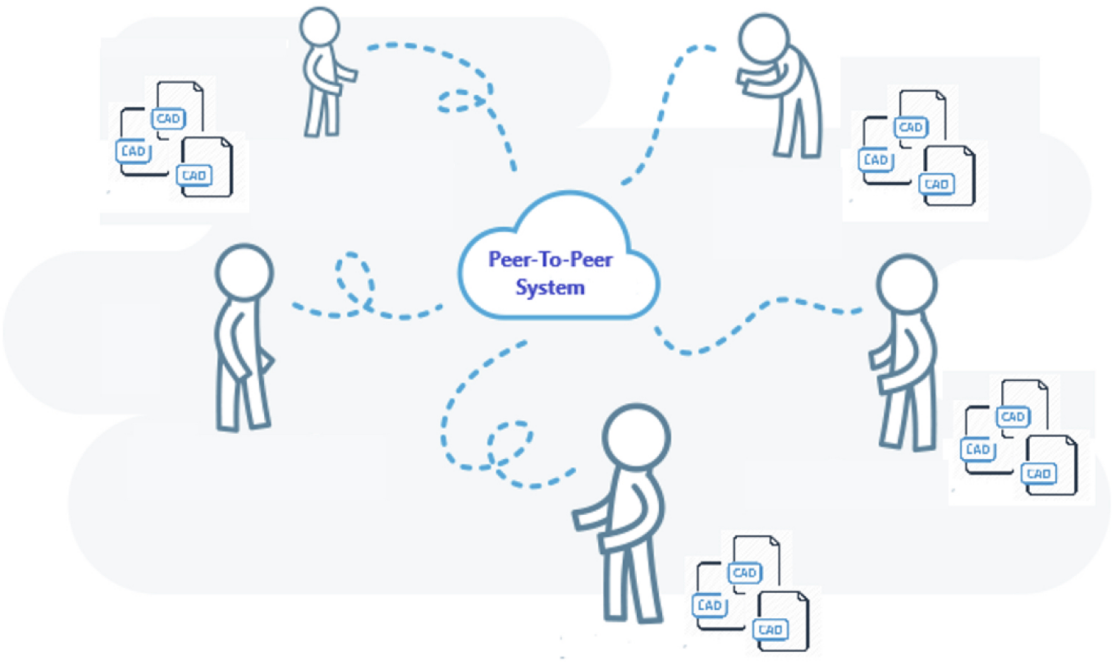

Figure 1.4: A peer-to-peer network with design files stored at each participant node. Modified from `https://grabcad.com/workbench`. Accessed April 20, 2020.

The use of a peer-to-peer network rather than a network with a central server presents problems with tracking provenance, however. For example, consider that manufacturer A manufactures some assembly a, while manufacturer B manufactures some assembly b. Now suppose that manufacturer C needs assemblies a and b to develop their part c. After searching the databases of manufacturers A and B and determining that a and b meet their specifications, manufacturer C purchases a and b from manufacturers A and B to use in their new assembly c. Now suppose that

manufacturer D has similarly determined that they need to use c in order to create their assembly d. It is necessary that manufacturer D compensate not only manufacturer C for the use of c, but also manufacturers A and B since they own the rights to sub-assemblies contained in c. This information may not be published in the part tree for c, and it must be stored somewhere without requiring a central authority. This problem can be solved by the use of blockchain technology, where each manufacturer and designer in the peer-to-peer network also participates on the blockchain.

1.4 Contributions

In this thesis we make several contributions:

1. We present an algorithm to search for part tree matches or subtree matches within a query tree.
2. We consider methods for determining distances between matched subtrees.
3. We present a distributed system that utilizes the algorithm and distance measures for searching within any number of manufacturer's databases.
4. We demonstrate how blockchain technology can be used to store information about available parts and honor provenance of individual parts and sub-assemblies used in assemblies across manufacturers.

Chapter 2

Theory

In order to query a database of part trees with some pattern tree provided as a search parameter, there are a mathematical concepts and algorithms that must be introduced, which we will outline in this chapter.

2.1 Definitions

Suppose that a manufacturer is searching a database to determine if a part tree they have created to specify requirements for a part or assembly; we will refer to this tree as a *pattern tree*. Any part tree in the database that is being searched to determine if it or one of its subtrees matches the pattern tree will be referred to as a *query tree*. Note that both the pattern tree and the query tree are rooted trees.

In order to search for a pattern tree within a query tree, we must first determine structural matches of the query tree or one of its subtrees to the pattern tree. Let T_1 and T_2 be rooted trees with respective vertex sets V_1 and V_2 and respective edge sets E_1 and E_2. T_2 is an isomorphism of T_1 (or rather, T_1 and T_2 are isomorphic) if there exists a bijection between V_1 and V_2

$$f : V_1 \to V_2$$

where for any vertices $u, v \in V_1$, u and v are adjacent if and only if $f(u)$ and $f(v)$ are adjacent in T_2. As such, searching for isomorphisms of a pattern tree within a query tree will provide a set of strict structural matches. In figure 2.2, observe that Trees A and B are isomorphic, with one possible bijection of nodes shown with node labels.

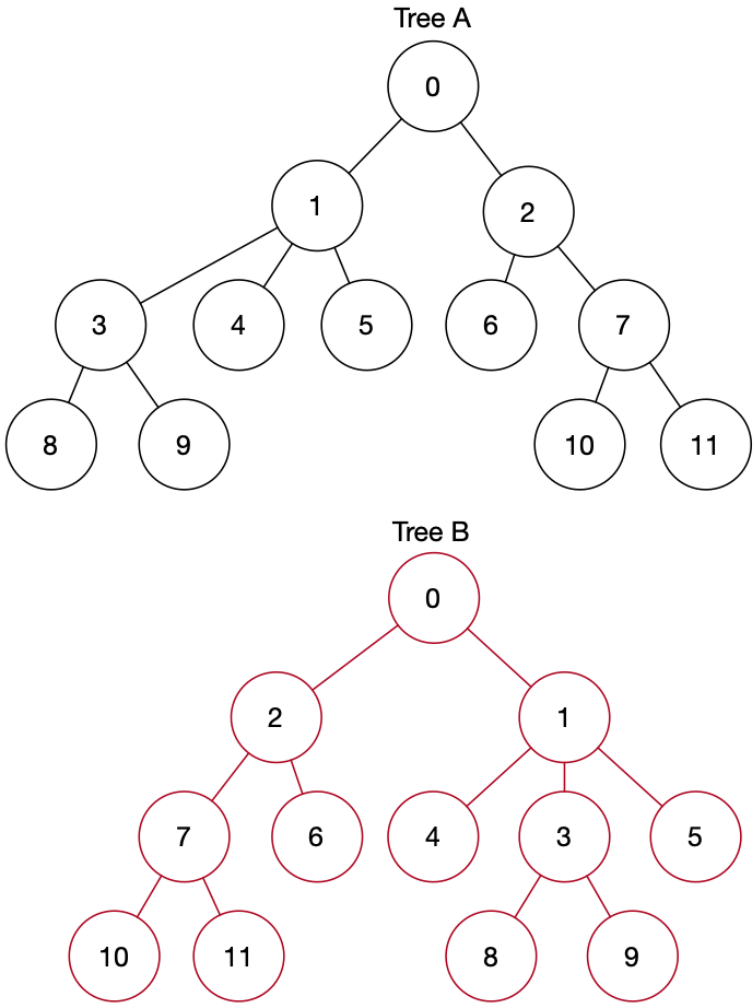

Figure 2.1: Two isomorphic trees.

However, it may be the case that the manufacturer searching for a pattern tree in a database may not want such a strict structural match; it is impossible to know exactly what procedure any given manufacturer may use for organizing nodes in a part tree, and they may define parts using more or fewer nodes than the manufacturer might place in the pattern tree, with sub-components placed lower or higher than expected. As such, a looser mechanism for structural matching is useful.

Suppose in some tree T with vertex set V and edge set E there is an edge between vertices $u, v \in V$. We can *subdivide* the edge between u and v by adding a vertex w to V, removing the edge between u and v from E, and adding edges between u and

w and between v and w. A tree T' is a *subdivision* of T if T' can be obtained by subdividing edges in T. We say that T_2 is an homeomorphism of T_1 (or rather, T_1 and T_2 are homeomorphic) if there exists some subdivision of T_1 that is isomorphic to some subdivision of T_2. As such, searching for homeomorphisms of a pattern tree within a query tree will provide a set of structural matches that allow for more flexibility in the way the two trees are defined by their respective manufacturers.

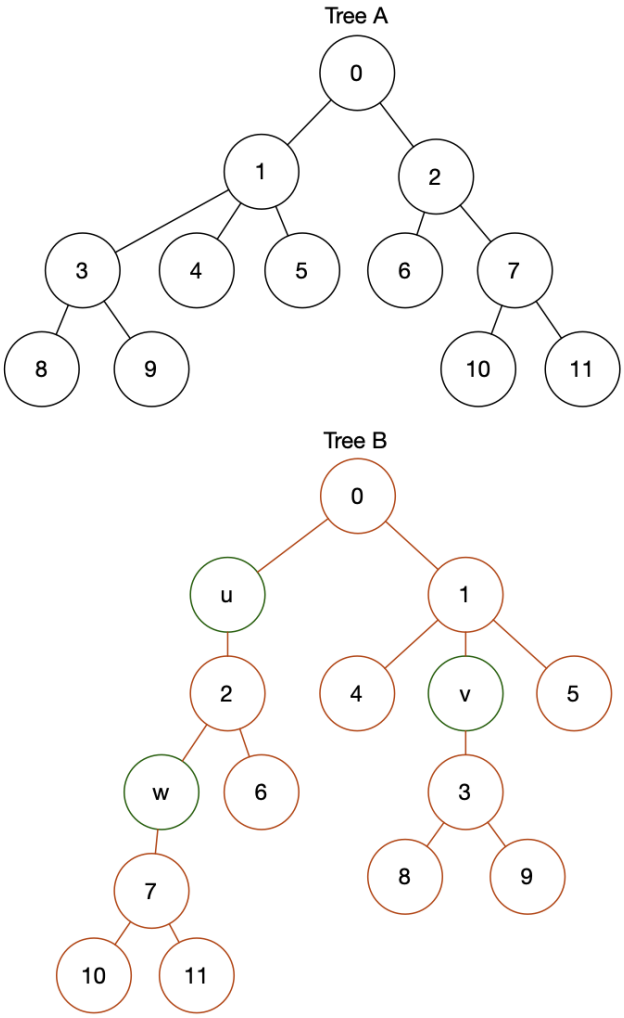

Figure 2.2: Two homeomorphic trees. If nodes u, v, and w are removed, then Trees A and B will be isomorphic.

Note that when searching for isomorphisms and homeomorphisms of a pattern tree within a query tree in this context, only subtrees of the query tree such that the entire

rooted subtree, including all the root's descendents, are isomorphic or homeomorphic to the pattern tree will be included in the set of resulting matches.

In this context, each node of a given tree will have a set of attributes that describe it. These attributes can be of any type, though this work largely considers only numerical attributes. The attributes that each node has has no guaranteed structure, and as such, each attribute set at a node cannot be considered tuple. Therefore, each node will have its attributes organized as set of key-value pairs. We refer to the *attribute set* of a node as the set of key-value pairs, while we refer to the *attribute key set* to be the set of keys in the attribute set.

2.2 Rooted Tree Isomorphism Search

Algorithms for determining if two rooted trees are isomorphic have been demonstrated in previous research, and this research builds upon one of them; however, a few modifications have been made. In searching to determine if a pattern tree is contained within a query tree, we must check all rooted subtrees of the query tree to determine which are isomorphic to the pattern tree. However, we also want to determine if any rooted subtrees of the pattern tree can be matched within the query tree as well; as noted earlier, it can be equally useful in this context to combine some set of part trees to form the pattern tree even if the entire pattern tree cannot be matched to a query tree or one of its rooted subtrees; if a subtree of the query tree T' is matched to a subtree of the pattern tree P', then the tuple (T', P') will be added to the list of isomorphisms.

Algorithm 1 has been adapted, with the aforementioned modifications, from an algorithm described in *The Design and Analysis of Computer Algorithms* [1] using improvements described by Alexander Smal [8], which assigns each node of the the query tree and pattern tree a canonical name that contains the complete history of their descendants; the two trees are isomorphic if their roots have the same label. In the following pseudocode, let T and P be the query tree and pattern tree, with root nodes r and q respectively, and let $T(u)$ (resp. $P(u)$) be the rooted subtree of T (resp. P) with root node u. Note that if a $T'(v)$ and $P'(v)$ are homeomorphic and, without loss of generality, if v had been combined with its single child n in the process of removing subdivisions from T, it is also true that $T(n)$ is homeomorphic to $P'(v)$. In order to have the most information available with respect to node attributes and

computing distance measures, this thesis uses the highest possible node in a tree as root when returning homeomorphic subtrees.

Algorithm 1 $GetIsomorphisms(T, P)$

```
isomorphisms ← ∅
AssignCanonicalNames(r)
AssignCanonicalNames(q)
for vertex u in P do
  for vertex v in T do
    if v.name = u.name then
      isomorphisms = isomorphisms ∪ (T(v), P(u))
    end if
  end for
end for
return isomorphisms
```

Algorithm 2 $AssignCanonicalNames(u)$

```
if u is a leaf then
  u.name ←"10"
else
  for v in u.children do
    AssignCanonicalNames(v)
  end for
end if
childrenNames ← sorted names of u.children
u.name ← "1"
for name in childrenNames do
  u.name ← u.name + name
end for
u.name ← u.name+"0"
```

2.3 Rooted Tree Homeomorphism Search

Algorithms exist for determining if one tree contains a subtree that is homeomorphic to a pattern tree, such as the one presented by Moon Jung Chung [2]. However, many of these algorithms are capable of finding homeomorphisms of a pattern tree within a query tree where the entire matched subtree is not guaranteed to be included in the homeomorphism. The use of such an algorithm would require us to check each returned homeomorphism to verify that it includes an entirety of the subtree corresponding to its root.

A simpler algorithm can actually be adapted from Algorithm 1. Note that homeomorphisms form an equivalence class, and as such, satisfy the transitivity property; that is, if rooted subtrees T_1 and T_2 are homeomorphic and T_2 and T_3 are homeomorphic, then T_1 and T_3 must also be homeomorphic. With this in mind, the problem of determining whether two trees are homeomorphic becomes as simple as removing any possible subdivisions from both trees and checking if the resultant trees are isomorphic. Algorithm 3 modifies Algorithm 1 to accomplish this. Again, let T and P be the query tree and pattern tree, with root nodes r and q respectively, and let $T(u)$ (resp. $P(u)$) be the rooted subtree of T (resp. P) with root node u.

Algorithm 3 $GetHomeomorphisms(T, P)$

```
homeomorphisms ← ∅
T' ← Compress(T)
P' ← Compress(P)
AssignCanonicalNames(r)
AssignCanonicalNames(q)
for vertex u in P' do
  for vertex v in T' do
    if v.name = u.name then
      homeomorphisms ← homeomorphisms ∪ {T(v), P(u)}
    end if
  end for
end for
return homeomorphisms
```

Algorithm 4 *Compress*(T)

for vertex v in T such that v has one child u **do**
  $v.children \leftarrow u.children$
  Remove v from T
end for
return T

2.4 Node Matching for Matched Subtrees

Once a pattern tree (or one of its subtrees) P' has been matched to an isomorphic subtree of a query tree T', in order to determine how similar the two matched trees are based on the attributes at each of their nodes, it is necessary to determine a map

$$g : V(P') \to V(T')$$

Once this map has been discovered, we can apply some distance function to the map $\delta(g)$ in order to determine how dissimilar the two matched trees are.

In order to for a vertex v in P' to be mapped to a vertex u in T', then $P'(v)$ must be isomorphic to $T'(u)$; knowing this makes it relatively simple to determine the mapping of nodes that can be used to calculate distance. However, it is possible that there exists more than one isomorphism between P' and T'; in this case, it is undesirable to find only one of the isomorphisms, as it is possible that using a different mapping of nodes could result in a lower distance between P' and T'.

If the P' is homeomorphic to T' but the two trees are not isomorphic, then determining which nodes of P' to map to corresponding nodes in T' becomes more difficult; in addition to the challenges with matching nodes in isomorphic trees, P' and T' may not even have the same number of nodes. However, by the definition of a homeomorphism, there exist subdivisions P'' and T'' of P' and T' respectively. That is to say, the map

$$h : V(P'') \to V(T'')$$

can be used to find the distance between the two trees similarly to if the trees were isomorphic by calculating $\delta(h)$.

In order to determine these subdivisions, some nodes of P' and/or T' will need

to be removed. However, when nodes are removed, the attributes defining that node will be removed as well; in order to avoid losing information that could be potentially useful in calculating $\delta(h)$, attributes in nodes that are removed will be attached to their parent nodes before removal, except for in cases where the parent has an attribute with the same key.

Algorithm 5 can be used on T' and P' whether they are isomorphic or only homeomorphic, and results in some set of mappings between either P' and T' or some subdivisions of them. Note that for each pair of nodes matched between P' and T', the level of the tree that the nodes appear on is included as well; while the nature of δ has been left ambiguous, it can be seen how in some contexts, it may be desirable to weight the distances between individual pairs of nodes differently depending on their distance from the root node. Also note that the $IsHomeomorphic$ function is used to determine whether two nodes can be mapped together, and works in the same way as Algorithm 3, but only checks the root nodes of two trees to determine if they are homeomorphic; as all isomorphisms are also homeomorphisms, this function will work whether T' and P' were determined to be isomorphisms or only homeomorphisms.

Algorithm 5 $GetMatchedNodes(T, P, level)$

```
if T has no children or P has no children then
  return ∅
end if
possibleMatches ← ∅
for vertex u in T.children do
  partMatches ← ∅
  for vertex v in P.children do
    checkForMatches ← false
    if level = 0 then
      checkForMatches ← true
    else if IsHomeomorphic(T(u), P(v)) then
      checkForMatches ← true
      while T(u) and P(v) have different numbers of children do
        if T(u) has 1 child then
          CombineChild(u))
        else if P(v) has 1 child then
          CombineChild(v))
        end if
      end while
    end if
    if checkForMatches then
      childMatches ← GetMatchedNodes(T(u), P(v), level + 1)
    end if
    for match in childMatches do
      match ← match ∪ (u, v, level)
    end for
    partMatches ← partMatches ∪ childMatches
  end for
  possibleMatches ← possibleMatches ∪ {partMatches}
end for
finalMatches ← ReduceMatches(possibleMatches)
return finalMatches
```

Algorithm 6 *CombineChild*(v)

$u \leftarrow$ child of v
for attribute key k in u not already in v **do**
  $v.k \leftarrow$ attribute value of u with key k
end for
$v.children \leftarrow u.children$

Algorithm 7 *ReduceMatches*($matches$)

$finalMatches \leftarrow \emptyset$
$checkedMatches \leftarrow \emptyset$
for $matchesX$ in $matches$ **do**
  $checkedMatches \leftarrow checkedMatches \cup \{matchesX\}$
  for $matchesY$ in $possibleMatches - checkedMatches$ **do**
    for $matchX$ in $matchesX$ **do**
      for $matchY$ in $matchesY$ **do**
        if $matchX$ and $matchY$ share no vertices **then**
          $matchX \leftarrow matchX \cup matchY$
        end if
      end for
    end for
  end for
  $finalMatches \leftarrow finalMatches \cup \{matchesX\}$
end for
return $finalMatches$

2.5 Distance Measures

Once nodes of a subtree of a query tree have been matched to nodes of a pattern tree, some measure of distance, or dissimilarity, should be applied to the pairs of nodes to determine the distance between the two trees. However, determining a measure of distance between two trees poses some challenges.

The first challenge that must be considered is that, as each node in a given part or assembly may correspond to an arbitrary assembly, part, or feature, the set of at-

tributes at any given node is not guaranteed to have a determined structure. As such, any two nodes that have been matched together by Algorithm 5 are not guaranteed to have all the same attributes, and may have no attributes in common.

The second challenge in determining the distance between two trees is that the context of search can vary. There may be situations where certain attributes must have exact or nearly exact values, or where specific attributes may be more important than others. There may also be situations where the distance between two nodes should be weighted more heavily in the overall distance between the two trees if it appears closer to the root node.

The third challenge is that there may not be a sensible way to determine the similarity between two values of a particular attribute. For example, if an attribute is listed on two matched nodes defining the material used to create the corresponding component, there may not be a way to translate this into a numerical distance. Furthermore, many distance measures use some knowledge about the distribution of values of a particular attribute, such as mean and standard deviation, but in this context, a distribution may not be useful. Note that not all nodes will have all attributes, and furthermore, an attribute may not mean the same thing at all nodes that have a value defined for that attribute. For example, it does not make sense to include the mass of a screw on the battery cover of a TV remote and the mass of a screw used to hold together pieces of an earth mover.

Given these challenges, it is important to note that there can be no distance function that will work properly in all situations. As such, we propose only a few simple measures, each having some advantages and disadvantages; note that each measure proposed deals only with numerical attributes. Given that two matched nodes may have not have all attributes in common, a user may wish to consider two matched nodes as more dissimilar if they have different attributes; with this in mind, for each measure we also provide a modification based on the Jaccard index that considers two nodes more dissimilar the more attributes there are in their union of attribute keys but not in their intersection. Note that while there exist scenarios where a user may wish to weight certain nodes' importance with respect to calculating the distance of two isomorphic or homeomorphic trees, in our implementation, the distance between two trees is calculated to be the average distance between their matched nodes using one of the measures proposed.

2.5.1 Euclidean distance

Euclidean distance is a common distance measure used to compare two data points; the formula for comparing nodes x and y is as follows with attribute key sets A_x and A_y respectively:

$$distance = \sqrt{\sum_{a \in A_x \cap A_y} (x(a) - y(a))^2}$$

In order to account for nodes with different attributes, the following measure can also be used:

$$distance = \left(2 - \left(\frac{|A_x \cap A_y|}{|A_x \cup A_y|}\right)\right) \sqrt{\sum_{a \in A_x \cap A_y} (x(a) - y(a))^2}$$

In determining the distance of two matched nodes, Euclidean distance can be useful in ruling out matched trees that have significant distance between the values of one or many attributes. However, given that the mean and standard deviation are likely to be meaningless for any given attribute key across the dataset of all parts, attribute values cannot be normalized; thus, Euclidean distance is sensitive to attribute values that exist on larger scales, which may be a problem in some contexts.

2.5.2 Cosine distance

The formula for comparing nodes x and y with attribute key sets A_x and A_y, respectively, using Cosine distance is as follows:

$$distance = 1 - \frac{\sum_{a \in A_x \cap A_y} x(a)y(a)}{\left(\sum_{a \in A_x \cap A_y} x(a)^2\right)\left(\sum_{a \in A_x \cap A_y} y(a)^2\right)}$$

In order to account for nodes with different attributes, the following measure can also be used:

$$distance = \left(2 - \left(\frac{|A_x \cap A_y|}{|A_x \cup A_y|}\right)\right)\left(1 - \frac{\sum_{a \in A_x \cap A_y} x(a)y(a)}{\left(\sum_{a \in A_x \cap A_y} x(a)^2\right)\left(\sum_{a \in A_x \cap A_y} y(a)^2\right)}\right)$$

Cosine distance can be particularly useful in situations where proportionality is more important than having exact values for particular attributes at matched nodes,

and unlike Euclidean distance, is not sensitive to attributes values existing using different scales across different nodes. However, in situations where attribute values at a particular node have entirely different contexts and by extension where proportionality is not meaningful, Cosine distance is not useful.

2.5.3 Ratio distance

Ratio distance is a simple measure that utilizes the proportional difference between values at a particular attribute to, in a sense, normalize attribute values without requiring any kind of structure to the data. The formula for comparing nodes x and y with attribute key sets A_x and A_y, respectively, using ratio distance is as follows:

$$distance = 1 - \left(\frac{\sum_{a \in A_x \cap A_y} \frac{min(x(a), y(a))}{max(x(a), y(a))}}{|A_x \cap A_y|} \right)$$

In order to account for nodes with different attributes, the following measure can also be used; note that this measure has the same value if the two nodes have the same set of attribute keys:

$$distance = 1 - \left(\frac{\sum_{a \in A_x \cap A_y} \frac{min(x(a), y(a))}{max(x(a), y(a))}}{|A_x \cup A_y|} \right)$$

Ratio distance is useful in situations where it is important that attribute values be similar to each other on the respective scale of the attribute; unlike Euclidean distance, it is not sensitive to differing scales of attributes across nodes. However, it is not particularly useful in situations where the minimum and maximum values at an attribute have different signs.

Chapter 3

Distributed System Using Blockchain

In this chapter we will describe a distributed system that provides capabilities for searching for a pattern tree across any number of databases of parts and assemblies and return a set of matches with corresponding distances, for purchasing a matched query tree from a manufacturer, and for tracking provenance of a part or assembly once it has been purchased and placed into a larger assembly.

3.1 Distributed System

The distributed system implemented follows a peer-to-peer model with an arbitrary number of participants, and in this model, no centralized authority is required at any step. Each node in the distributed system can act as a server processing requests to search for a pattern tree; in this case, the node will search through its database for tree matches, calculate a distance for each match, and return the results to the source of the request. Acting as a server, the node can also return the data for a tree, structured as an XML file, that a client has purchased after conducting a search.

Naturally, each node can also act as a client. In order to perform a full search for a pattern tree, the node must send a request to all nodes whose corresponding databases it wishes to search. Upon receiving results, the node can then be used to make a purchase of a part or assembly that the user deems to have a low enough level of dissimilarity to the pattern tree.

3.2 Blockchain and Smart Contracts

While the simple peer-to-peer system described in Section 3.1 can be used to process searches and initiate purchases, each node may not properly track information about purchases. Furthermore, if an assembly containing parts and/or assemblies from other manufacturers is purchased, the corresponding manufacturers should be appropriately compensated as well; in Figure 3.1, consider the propagation of small parts being used in larger and larger assemblies until such a assembly as the radial engine in Figure 1.1. Unfortunately, this peer-to-peer model alone is unsuitable for maintaining provenance of a part or assembly once it has been purchased and added to a larger assembly by another party.

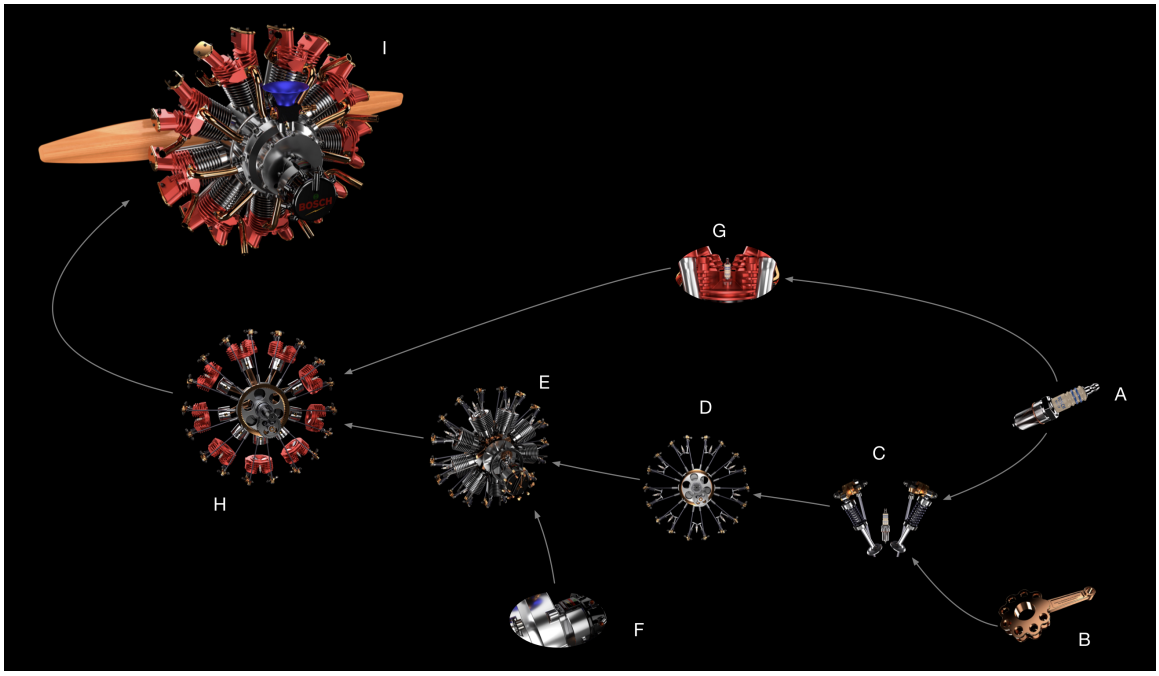

Figure 3.1: The propagation of smaller parts, such as A and B, into larger assemblies. Modified from `https://grabcad.com/library/karl-erik-olsryd-9-cylinder-radial-wright-j-5-whirlwind?viewer=ab88242f5eb28702abba41afc69f60c5`. Accessed April 21, 2020.

The use of blockchain technology in the peer-to-peer system can alleviate these issues. A blockchain consists of a list of blocks, each containing a hash of the previous block in the blockchain, a timestamp, and the block's data. As a block in a blockchain cannot be altered without also altering every other block following, a blockchain can be used to securely store information in a decentralized system. A variety of protocols

exist for communicating between nodes in a decentralized system with respect to the blockchain and validating new blocks placed on the blockchain.

In this context, a blockchain with each manufacturer as a participant can be used to store information about which parts and assemblies are available for purchase and transactions between manufacturers. In order to carry out both transactions between manufacturers and the publishing of new parts or assemblies available for purchase, we can use smart contracts. A smart contract refers to some computation that can be done on a blockchain. As such, when a smart contract is used to carry out a transaction or publish a new part, the computation is verified and stored on the blockchain and cannot be altered, similar to any other block residing on the blockchain. The immutability of smart contracts can be useful in tracking purchases; in particular for this context, its usefulness is in its application for tracking provenance of parts and assemblies and making appropriate payments when assemblies containing smaller parts and assemblies are purchased.

For this system, two different kinds of a smart contracts are used. One, called *Publish*, is used to place information about a new part or assembly to the blockchain, including its manufacturer, price, and any parts or assemblies that it makes use of, referenced by the addresses of their corresponding *Publish* contracts. Once a *Publish* contract has been placed on the blockchain, no further processing is required except to view the information stored. The second, called *Purchase*, is used to make a transaction between two manufacturers when one purchases a part or assembly created by the other. The contract stores information pertaining to the purchase, such as the two manufacturers and the part involved. More importantly, however, the *Purchase* smart contract utilizes information in the *Publish* smart contract corresponding to the part being purchased to pay any other manufacturers whose parts or assemblies were used in the part being purchased.

Chapter 4

Implementation

In this chapter, we provide in more detail implementation details for the algorithms detailed in chapter 2 and chapter 3, including tools necessary to use this implementation. Furthermore, we provide details of how datasets were generated for use by the system.

4.1 Node Implementation

The distributed system used in this project consists of ten nodes. Each node consists of a server and a client, communicating with each other via HTTP POST requests and responses. The client side of each node implements a simple user interface, created using React.js, that allows a user to upload a pattern tree specification, select a distance measure to use, initiate a search of the other manufacturers' databases, and make purchases of any returned matches. When initiating a search, the client makes requests to each manufacturer and combines the results once each corresponding server has responded with results from its respective database.

The server side of each node is implemented as a Flask server in Python. The API has only one endpoint, accepting a string containing the pattern tree represented as an XML tree and a distance measure to use when searching. Because homeomorphism search can also return isomorphic matches to a pattern tree as well, by default only Algorithm 3 is used. The search algorithm and node matching is implemented in Python.

4.2 Blockchain Implementation

Given that currency is required to participate on a live blockchain, this project was implemented using a local blockchain. This was a local version of the programmable Ethereum blockchain, a popular platform for building distributed applications. Ethereum provides its own native currency, similar to Bitcoin, known as Ether; transactions on the Ethereum blockchain are done using Ether, and some amount of Ether is required in order to carry out a smart contract on the blockchain, depending on the amount of memory and processing power the smart contract requires. A local Ethereum blockchain can be created and interfaced with using the Ganache tool.

In order to connect to the local Ethereum blockchain, the JavaScript web3 library is used. This required the addition of an Express server running on Node.js with endpoints that can be used to publish or purchase a part or assembly; this server is accessed both by the client side of a node when making purchases and during dataset generation when parts are published, described in Section 4.3.

The Ethereum blockchain provides a Turing-complete language, called Solidity, for use in creating smart contracts. Each smart contract in Solidity takes the form of an class that can be instantiated, with instance methods that can be used to do calculations on the object's fields, view the object's fields, or perform Ether transactions between participants on the blockchain. Two classes, implementations of the *Publish* and *Purchase* smart contracts, were created using the Solidity language.

4.3 Dataset Generation

For the purposes of this project, finding a real-world dataset proved difficult; finding parts or assemblies represented as a tree structure presented a challenge, and most manufacturers naturally do not have the specifications for their products freely available. As such, it was necessary to generate a synthetic dataset that provided the necessary tree structure for testing.

To begin with, a set of a simple parts is generated from several templates. Each template provides some kind of structure to a part; for example, the template for a screw describes the screw's pin, thread, and head in numerical parameters. Each template provides some constraints for the numerical parameters; for example, the head of a screw cannot be wider than the diameter of the corresponding pin. The

generator accepts a template and some user-defined number of parts to generate, and for each generated part fills the template in with randomly-generated fields for each parameter, rejecting a field if it does not meet the constraints laid out by the template.

Once some set of simple parts have been generated from any number of templates, a set of assemblies can be created by combining some random subset of simple parts and adding additional simple parameters to each such as placement and rotation. Any number of assemblies can be generated this way. Additionally, assemblies can be generated in the same manner by combining both assemblies and simple parts. Once a set of assemblies of sufficient depth to satisfy the user has been created, the dataset can be used for testing.

This part generation mechanism interfaces with the blockchain as well. Each time it is run, each part or assembly is randomly assigned to any of the participating nodes and a price strictly greater than the sum of prices of children parts/assemblies is randomly generated, and the generator keeps track of which parts and assemblies are used as children for each assembly it creates. At the end of the run, a *Publish* contract is created on the blockchain, storing the part name, manufacturer, children parts, and price of each. The contract address must be stored at the top-level tag of the XML file for the respective tree in order to look up this information later.

Chapter 5

Analysis

In this chapter, we analyze the speed of the system in performing search and examine the performance of the proposed distance measures given changes against a baseline part.

5.1 Speed

The search algorithm was performed in each case on a 2015 Macbook Pro with a 2.9 GHz Dual-Cord Intel Core i5 processor and 8 GB of 1867 MHz DDR3 RAM. For each proposed distance measure, the search algorithm was performed on randomly generated datasets consisting of 10, 25, 50, 100, 200, 500, and 1,000 query trees varying from 10 to 86 nodes in size and a pattern tree with 26 nodes. Figure 5.1 shows the average speed of performing the algorithm with each distance measure across five runs.

As expected, the algorithm appears to increase runtime linearly as parts increase; given that the time taken to search for homeomorphisms within a part is not affected by the number of additional parts that must be searched. Also note that the distance measure used does not affect the speed; this is also expected, as searching a tree for homeomorphisms and matching nodes is more complex than performing simple calculations using node attributes.

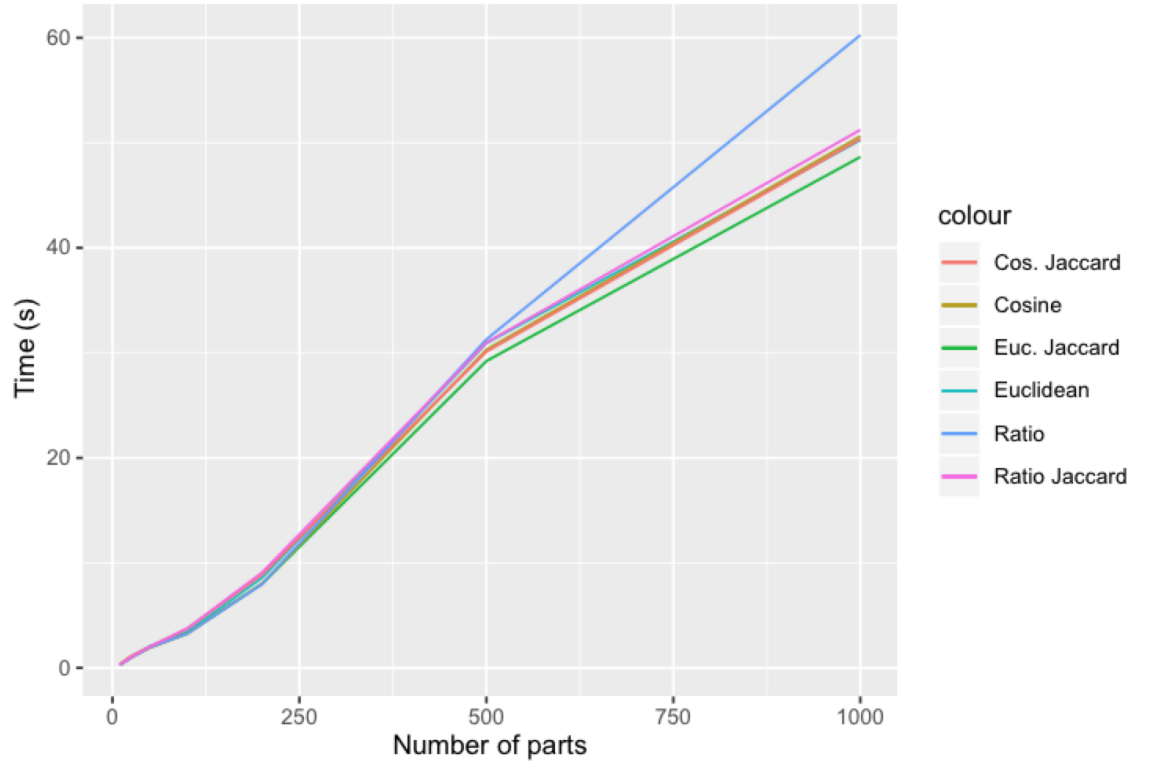

Figure 5.1: The speed of the algorithm with each distance measure

5.2 Distance Measures

In order to compare distance measures, the same pattern tree used to test the speed of the algorithm was modified into five different parts. For each of five probabilities $p = 0.1, 0.25, 0.5, 0.9, 1$, each attribute at each node of the pattern tree was mutated randomly by a factor of 0.05 to 2 with probability p. The original pattern tree was compared to each mutated pattern tree using the three distance measures (without the additions based on the Jaccard index, as no attributes were removed in any mutated tree), with the results shown in Table 5.1. Note that for all distance measures proposed, the distance between any pattern tree and itself is 0.

Formula	Measure	$p = 10\%$	$p = 25\%$	$p = 50\%$	$p = 90\%$	$p = 100\%$
$\sqrt{\sum_{a \in A_x \cap A_y} (x(a) - y(a))^2}$	Euclidean	3.75	2.55	4.69	7.46	7.65
$1 - \frac{\sum_{a \in A_x \cap A_y} x(a)y(a)}{\left(\sum_{a \in A_x \cap A_y} A(a)^2\right)\left(\sum_{a \in A_x \cap A_y} A(a)^2\right)}$	Cosine	$1.58 * 10^{-8}$	$1.52 * 10^{-5}$	$1.12 * 10^{-3}$	$7.26 * 10^{-5}$	$4.35 * 10^{-3}$
$1 - \left(\frac{\sum_{a \in A_x \cap A_y} \frac{min(x(a),y(a))}{max(x(a),y(a))}}{\lvert A_x \cap A_y \rvert}\right)$	Ratio	0.0390	0.0410	0.0767	0.119	0.168

Table 5.1: The distance generated by each distance measure against mutated trees with various probabilities of mutation.

Observe that both Euclidean and Cosine distance do not strictly increase as the probability of mutation increases, while Ratio distance does. However, as mentioned in Section 2.5, the utility of a particular distance is dependent upon the user and the context, and as such, and objective measure for determining usefulness of a particular distance measure is not achievable.

Chapter 6

Conclusion

In this thesis, we have noted that the ability for parts and assemblies to be searched by their hierarchical structure and characteristics and shared or purchased in a distributed manner rather than through a single central database is amenable to the democratization of the manufacturing industry. We further noted that these issues could be accomplished by use of tree search algorithms and a peer-to-peer system with an associated blockchain, respectively.

In order to provide search functionality for parts and assemblies, we proposed algorithms for discovering all isomorphisms and homeomorphisms in a query tree from a pattern tree and matching each node in a pattern tree or one of its subtrees to a node in the matched subtree of the query tree. We also have proposed some potential distance measures that can be used to determine the distance between two matched trees based on their attributes, each with associated advantages and disadvantages, and we have demonstrated that each distance measures performs differently on mutated trees, but they perform similarly in speed.

We have also demonstrated that these algorithms and distance measures can be used to search for a desired part or assembly within a manufacturer's database and provide some information on how closely the part or assembly matches the searcher's specifications. Additionally, we have implemented a peer-to-peer system that allows manufacturers to publish parts and assemblies, search for assemblies based on some specification at the other nodes in the system, and purchase assemblies given an associated distance from a pattern assembly tree, all without requiring a centralized authority or database. Finally, we have demonstrated that the use of blockchain can alleviate the need to store information on manufacturer, price, or information

on children parts and assemblies within an assembly tree file while still maintaining provenance when a part or assembly is used in an assembly created by a different designer or manufacturer.

6.1 Future Work

One issue to address in future research is the creation of more context-suited distance measures. The measures used in this thesis are simple, and while they attempt to account for some industry-appropriate contexts, they are only usable with numerical attributes, and they do not account for some possible factors that a user may wish to address, such as the level at which a two matched nodes appear in their trees. Furthermore, these distance measures do not allow for attributes that a user may consider more important in calculating distance to be weighted.

Another issue that should be addressed in future research is guaranteeing that once a part a published by manufacturer A has been purchased for use by manufacturer B, part a should be recorded as originating from manufacturer A in any subsequent assemblies that it appears in. While this system does track the provenance of part a when publishing an assembly that part a has been included in, this requires that manufacturer B not deliberately exclude that information when publishing a new assembly. One potential solution to this problem is to add a fingerprinting mechanism that allows a manufacturer to include some information in trees they publish that is not easily found and removed by any subsequent purchasers, and as such allows them to verify that nobody uses their trees without giving proper credit.

Bibliography

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Publishing Company, 1974.

[2] Moon Jung Chung. $o(n^{2.5})$ time algorithms for the subgraph homeomorphism problem on trees. *Journal of Algorithms*, 8(1):106–112, 1987.

[3] Matías Covarrubias, Germán Gutiérrez, and Thomas Philippon. From good to bad concentration? u.s. industries over the past 30 years. 2019.

[4] Fortune. Global 500. Online.

[5] Jason Furman. Business investment in the united states: Facts, explanations, puzzles, and policies, 2015.

[6] The World Bank Group. Manufacturing, value added (current us$). Online.

[7] Adam Robinson. The democratization of manufacturing. Online, Feb 2015.

[8] Alexander Smal. Explanation for ‘tree isomorphism‘ talk. Online, 2008.

Appendix A

Source Code

The source code for the distributed system and tree search is spread over three Python sections, two JavaScript sections, and one Solidity section. The Python sections contain code for creating and publishing new parts and assemblies, searching query trees for a pattern tree, and accepting requests from clients to search query trees. The JavaScript sections include code for the user interface and for making purchases. The Solidity section contains code for the *Publish* and *Purchase* smart contracts; note that application binary interfaces (ABIs) must also be generated from Solidity code before smart contracts can be used by web3.

A.1 Part and Assembly Generation

A.1.1 createpart.py

```
from .part import AbstractPart as ap
from .part import CompoundPart as cp
import xml.etree.ElementTree as et
import requests
import random
import glob
import math
import json

# For generating simple parts
```

```
class PartGenerator:

  def __init__(self, path, output=None, debug=False):
    self.abstractpart = ap(path, debug=debug)
    self.checks = self.abstractpart.checks
    self.params = self.abstractpart.params
    self.reqparams = self.abstractpart.getrequiredparams()
    self.optparams = self.abstractpart.getoptionalparams()
    self.output = output
    self.debug = debug

  # Randomly generate a value for a given attribute of a
  # certain type
  def generateattribvalue(self, key, value, values):
    t = 'float'
    if 'type' in self.params[key].keys():
      t = self.params[key]['type']

    if t == 'string':
      return 'randomstring'
    elif t == 'int':
      l = 0
      u = 0
      try:
        l = int(self.checks.getlowerbound(key, values))
      except OverflowError:
        l = -10000000000000000000
      try:
        u = int(self.checks.getupperbound(key, values))
      except OverflowError:
        u = 10000000000000000000
      return random.randint(l, u)
    elif t == 'boolean':
      r = random.random()
```

```
    return r < 0.5
  else:
    l = self.checks.getlowerbound(key, values)
    u = self.checks.getupperbound(key, values)
    return random.uniform(l, u)

# Verify that parameter values follow their listed type
def validatevalue(self, key, value):
  t = 'float'
  if 'type' in self.params[key].keys():
    t = self.params[key]['type']

  try:
    if t == 'string':
      v = str(value)
    elif t == 'int':
      v = int(value)
    elif t == 'boolean':
      v = bool(value)
    else:
      v = float(value)
  except Exception:
    return False

  return True

# Create n parts from template
def generateparts(self,
                  n=1,
                  params={},
                  groupname=None,
                  changeoptionalparams=False,
                  createcontracts=False,
                  contractlocation=''):
```

```
name = groupname
if name is None:
  name = self.abstractpart.getname()

for k, v in params.items():
  if k in self.params.keys():
    validated = self.validatevalue(k, v)
    if not validated:
      return False

i = 1
index = 0
parts = []
outnames = []
partnames = []
# Create one part per loop
while i <= n:
  index += 1
  if self.debug:
    print('Creating␣part␣"' + name + str(i) +
          '",␣try␣' + str(index))
  p = {}
  # Set values of attributes
  for k, v in self.reqparams.items():
    if k in params.keys():
      p[k] = params[k]
    else:
      p[k] = self.generateattribvalue(k, v, p)
  for k, v in params.items():
    if k not in p.keys():
      if k in self.params.keys():
        p[k] = v
  if changeoptionalparams:
    for k, v in self.optparams.items():
```

```
        if k not in p.keys():
          p[k] = self.generateattribvalue(k, v, p)

    # Write part to file
    outname = None
    if self.output is not None:
      partname = name
      outname = self.output + '/' + name
      if n != 1:
        outname += str(i)
        partname += str(i)
      outname += '.xml'
    if not createcontracts:
      result = self.abstractpart.createpart(
              p,
              outname,
              name=partname)
    else:
      result = self.abstractpart.createpart(p,
                                             None,
                                             partname)
      outnames.append(outname)
      partnames.append(partname + '.xml')
    if result != False:
      i += 1
      index = 0
      parts.append(result)

# Publish parts to blockchain
if createcontracts:
  partstosend = []
  for i in range(len(parts)):
    parttosend = {
        'owner': int(parts[i].getroot().get('owner')),
```

```
            'cost': float(parts[i].getroot().get('price')),
            'children': []}
        partstosend.append(parttosend)

      response = requests.post(
              'http://localhost:3001/publish',
              json.dumps({'parts': partstosend}),
              headers={'Content-Type': 'application/json'})

      # Write parts with contract addresses to files
      addresses = response.json()['addresses']
      for i in range(len(parts)):
        part = parts[i]
        owner = part.getroot().get('owner')
        part.getroot().attrib.pop('owner')
        part.getroot().set('address', addresses[i])
        part.write(outnames[i])
        part.write(contractlocation + '/' + owner + '/' +
                    partnames[i])

    return parts

# For generating assemblies
class CompoundPartGenerator:

  def __init__(self, folder, output=None, debug=False):
    if folder[-1] is not '/':
      folder += '/'
    folder += '*.xml'
    files = glob.glob(folder)
    self.parts = []
    for f in files:
      self.parts.append(et.parse(f))
    self.output = output
```

```
  self.debug = debug

# Generate position of a part in the assembly
def getrandomposition(self, unit):
  pos = [0, 0, 0]
  for i in range(3):
    if unit == 'millimeter':
      pos[i] = random.uniform(-100, 100)
    elif unit == 'centimeter':
      pos[i] = random.uniform(-50, 50)
    elif unit == 'meter':
      pos[i] = random.uniform(-10, 10)
    elif unit == 'kilometer':
      pos[i] = random.uniform(-.5, .5)
  return tuple(pos)

# Generate the rotation of a part in the assembly
def getrandomrotation(self, unit):
  rot = [0, 0, 0]
  for i in range(3):
    if unit == 'degree':
      rot[i] = random.uniform(0, 360)
    if unit == 'radian':
      rot[i] = random.uniform(0, 2 * math.pi)
  return tuple(rot)

# Generate n assemblies
def generatecompoundparts(self,
                          n=1,
                          groupname='compoundpart',
                          maxparts=3,
                          minparts=1,
                          units={},
                          createcontracts=False,
```

```
                          contractlocation=''):
    i = 1
    index = 0
    parts = []
    outnames = []
    partnames = []
    compoundparts = []

    # Generate one assembly per loop
    while i <= n:
      index += 1
      if self.debug:
        print('Creating␣compound␣part␣"' + groupname +
              str(i) + '",␣try␣' + str(index))

      c = cp(units, (groupname + str(i)))
      # Use random number of parts/assemblies
      # within constraints
      num = random.randint(minparts, maxparts)
      for a in range(num):
        pos = self.getrandomposition(c.units['length'])
        rot = self.getrandomrotation(c.units['angle'])
        partindex = random.randrange(0, len(self.parts))
        c.addpart(self.parts[partindex].getroot(),
                  pos,
                  rot)
      outname = None
      if self.output is not None:
        partname = groupname
        outname = self.output + '/' + groupname
        if n != 1:
          outname += str(i)
          partname += str(i)
        outname += '.xml'
```

```
    result = c.tree
    if not createcontracts:
      result = c.write(outname)
    else:
      outnames.append(outname)
      partnames.append(partname + '.xml')
      compoundparts.append(c)
    i += 1
    index = 0
    parts.append(result)

# Publish assemblies to blockchain
if createcontracts:
    partstosend = []
    for i in range(len(parts)):
      parttosend = {
          'owner': int(parts[i].getroot().get('owner')),
          'cost': compoundparts[i].cost,
          'children': compoundparts[i].children}
      partstosend.append(parttosend)

    response = requests.post(
          'http://localhost:3001/publish',
          json.dumps({'parts': partstosend}),
          headers={'Content-Type':
              'application/json'})

    addresses = response.json()['addresses']
    # Write assemblies to files with contract addresses
    for i in range(len(parts)):
      part = parts[i]
      owner = part.getroot().get('owner')
      part.getroot().attrib.pop('owner')
      part.getroot().set('address', addresses[i])
```

```
        compoundparts[i].write(outnames[i])
        compoundparts[i].write(contractlocation + '/' +
                               owner + '/' + partnames[i])

    return parts
```

A.1.2 part.py

```
from .units import transformunits
import xml.etree.ElementTree as et
from xml.etree.ElementTree import TreeBuilder as tb
from xml.etree.ElementTree import Element
import xml.dom.minidom as md
import copy
import random as rand

class ParameterException(Exception):
  pass


class CheckException(Exception):
  pass


# Used to store any constraints listed in the part
# template file
class CheckList:

  def __init__(self, params):
    self.checks = []
    self.params = params

  def add(self, c):
    self.checks.append(c)
```

```
  # Verify that all constraints are met
  def evaluate(self, values):
    for check in self.checks:
      if not check.evaluate(values):
        raise CheckException('Check␣failed:␣' +
                  str(check))

  # Determine lowest possible value for a value based
  # on constraints
  def getlowerbound(self, k, values, minimum=-100):
    lower = minimum
    for c in self.checks:
      l = c.getlowerbound(k, values)
      if l > lower:
        lower = l
    return lower

  # Determine highest possible value for a value based
  # on constraints
  def getupperbound(self, k, values, maximum=100):
    higher = maximum
    for c in self.checks:
      u = c.getupperbound(k, values)
      if u < higher:
        higher = u
    return higher

# Used to store and validate information on one constraint
class Check:

  def __init__(self, expr, params):
    self.expr = expr
    tok = expr.split('␣')
```

```
self.tokens = []
subexpr = []
self.subexpr = []
left = True
lastcomp = ''
for t in tok:
  token = {'value': t, 'type': 'unknown'}
  if t in params.keys():
    token['type'] = 'parameter'
    subexpr.append(token)
  elif t in ['<', '>', '>=', '<=', '==']:
    token['type'] = 'comparison'
    lastcomp = t
    s = {'expr': subexpr, 'comparison': t}
    if left:
      s['position'] = 'left'
    else:
      s['position'] = 'right'
    p = []
    for subtoken in subexpr:
      if subtoken['type'] == 'parameter':
        p.append(subtoken['value'])
    s['params'] = p
    s['tokens'] = subexpr.copy()
    self.subexpr.append(s.copy())
    subexpr = []
    left = False
  elif t in ['+', '-', '/', '*', '^', ')', '(']:
    token['type'] = 'operator'
    subexpr.append(token)
  else:
    token['type'] = 'constant'
    subexpr.append(token)
  self.tokens.append(token)
```

```
    s = {'expr': subexpr, 'comparison': lastcomp}
    if left:
      s['position'] = 'left'
    else:
      s['position'] = 'right'
    p = []
    for subtoken in subexpr:
      if subtoken['type'] == 'parameter':
        p.append(subtoken['value'])
    s['params'] = p
    s['tokens'] = subexpr.copy()
    self.subexpr.append(s.copy())

# Get lower bound of one constraint
def getlowerbound(self, k, values):
    expr = None
    for s in self.subexpr:
      if k in s['params']:
        expr = s
    if expr is None:
      return -float('inf')
    if expr['position'] == 'left':
      otherexpr = self.subexpr[1]
      if expr['comparison'] in ['<', '<=']:
        return -float('inf')
    else:
      otherexpr = self.subexpr[0]
      if expr['comparison'] in ['>', '>=']:
        return -float('inf')
    returnval = self.evaluatesubexpr(otherexpr, values)
    if returnval is None:
      return -float('inf')
    else:
      return returnval
```

```
# Get upper bound of one constraint
def getupperbound(self, k, values):
  expr = None
  for s in self.subexpr:
    if k in s['params']:
      expr = s
  if expr is None:
    return float('inf')
  if expr['position'] == 'left':
    otherexpr = self.subexpr[1]
    if expr['comparison'] in ['>', '>=']:
      return float('inf')
  else:
    otherexpr = self.subexpr[0]
    if expr['comparison'] in ['<', '<=']:
      return float('inf')
  returnval = self.evaluatesubexpr(otherexpr, values)
  if returnval is None:
    return float('inf')
  else:
    return returnval

# Determine if the constraint has been met
def evaluate(self, values):
  ex = ''
  for tok in self.tokens:
    if tok['value'] in values.keys():
      v = values[tok['value']]
      if isinstance(v, str):
          ex += '\'' + v + '\' '
      else:
        ex += str(v) + ' '
    else:
```

```
        ex += tok['value']
    return eval(ex)

  # Evaluate any subexpressions that occur in a constraint
  # statement
  def evaluatesubexpr(self, subexpr, values):
    ex = ''
    for tok in subexpr['tokens']:
      if tok['value'] in values.keys():
        v = values[tok['value']]
        if isinstance(v, str):
            ex += '\'' + v + '\' '
        else:
          ex += str(v) + ' '
      else:
        ex += tok['value']
    try:
      return eval(ex)
    except Exception:
      return None

  def __str__(self):
    return self.expr


# Contains information on a template
class AbstractPart:

  def __init__(self, path, debug=False):
    self.path = path
    self.tree = et.parse(path)
    root = self.tree.getroot()
    self.params = {}
    for param in root.iter('parameter'):
```

```
    p = param.attrib
    n = p['name']
    p.pop('name')
    self.params[n] = p
  self.debug = debug
  self.checks = CheckList(self.params)
  for param in self.params.values():
    if 'check' in param.keys():
      c = param['check']
      c = c.split(';')
      for check in c:
        self.checks.add(Check(check, self.params))

# Get parameters that must be met (have no default
# value)
def getrequiredparams(self):
  required = {}
  for k, v in self.params.items():
    if 'default' not in v.keys():
      required[k] = v
  return required

# Get parameters that are optional (have a default
# value)
def getoptionalparams(self):
  optional = {}
  for k, v in self.params.items():
    if 'default' in v.keys():
      optional[k] = v
  return optional

# Get the name of the template
def getname(self):
  t = self.tree.getroot().attrib
```

```
  return t['name']

# Change string information in XML tree to value of
# required type
def transformvalue(self, key, value):
  param = self.params[key]
  if 'type' in param.keys():
    t = param['type']
    if t == 'string':
      return str(value)
    elif t == 'int':
      return int(value)
    elif t == 'boolean':
      return bool(value)
    else:
      return float(value)
  else:
    return float(value)

# Create a part based on the associated template
def createpart(self, params, output=None, name=None):
  tree = copy.deepcopy(self.tree)
  tree.getroot().remove(tree.find('parameters'))

  paramstoadd = []
  for k in self.params.keys():
    paramstoadd.append(k)
  values = {}

  tree.getroot().set('owner', str(rand.randint(0, 9)))
  tree.getroot().set('price', str(rand.randint(1, 8)))

  # Substitute values for parameters
  try:
```

```
    while len(paramstoadd) > 0:
      param = paramstoadd.pop(0)
      if param in params.keys():
        values[param] = self.transformvalue(
                            param,
                            params[param])
      else:
        if 'default' not in self.params[param]:
          raise ParameterException(
                'There is no value for parameter: ' +
                param)
        else:
          if self.params[param]['default'][0] == '$':
            lookup = self.params[param]['default'][1:]
            if lookup in values.keys():
              values[param] = values[lookup]
            else:
              paramstoadd.append(param)
          else:
            values[param] = self.transformvalue(param,
                        self.params[param]['default'])

    # Check constraints
    self.checks.evaluate(values)

    for tag in tree.getroot().iter():
      for k, v in tag.attrib.items():
        if v[0] == '$':
          value = values[v[1:]]
          tag.set(k, str(value))

  except (ParameterException, CheckException) as e:
    if self.debug:
      print(e)
```

```
      return False

    tree.getroot().tag = 'part'

    if name is not None:
      tree.getroot().set('name', name)

    if output is not None:
      tree.write(output)

    return tree


# An assembly
class CompoundPart:

  def __init__(self, units={}, name=None):
    self.units = {'length': 'meter',
                  'angle': 'degree',
                  'mass': 'kilogram'}
    for k, v in units:
      self.units[k] = v
    self.tree = et.ElementTree()
    self.tree._setroot(Element('compoundpart'))
    root = self.tree.getroot()
    ###########################################
    root.set('owner', str(rand.randint(0, 9)))
    root.set('price', str(rand.randint(1, 8)))
    self.cost = float(root.get('price'))
    self.children = []
    ###########################################
    if name is not None:
      root.set('name', name)
    meta = Element('meta')
```

```
  units = Element('units')
  units.append(Element('mass',
                   {'unit': self.units['mass']}))
  units.append(Element('length',
                   {'unit': self.units['length']}))
  units.append(Element('angle',
                   {'unit': self.units['angle']}))
  meta.append(units)
  root.append(meta)
  self.parts = Element('parts')
  root.append(self.parts)

# Add a part to the assembly
def addpart(self,
            part,
            position=(0,0,0),
            rotation=(0,0,0)):
  part = copy.deepcopy(part)
  pos = ''
  for i in position:
    pos += str(i) + ','
  self.tree.getroot().set('price',
            str(int(self.tree.getroot().get('price')) +
                   int(part.get('price'))))
  part.set('position', pos[0:-1])
  self.children.append(part.get('address'))
  part.attrib.pop('address')
  part.attrib.pop('price')
  rot = ''
  for i in rotation:
    rot += str(i) + ','
  part.set('rotation', rot[0:-1])
  self.parts.append(part)
  transformunits(part, self.units)
```

```
  for element in part.iter('meta'):
    part.remove(element)

# Print assembly to file in readable format
def prettyprint(self,
                element,
                indentlevel,
                hasmorechildren):
  indent = "\n"
  if indentlevel > 0:
    indent += (indentlevel - 1) * '␣␣'
  numchildren = len(element)
  if numchildren > 0:
    if not element.text:
      element.text = indent + '␣␣'
      if indentlevel > 0:
        element.text += '␣␣'
      x = 0
      for child in element:
        self.prettyprint(
            child,
            indentlevel + 1,
            x < numchildren - 1)
        x += 1
      if not element.tail:
        element.tail = indent
        if hasmorechildren:
          element.tail += '␣␣'
  else:
    if indentlevel > 0 and not element.tail:
      element.tail = indent
      if hasmorechildren:
        element.tail += '␣␣'
```

```
  # Strip assembly XML of whitespace
  def strip(self, element):
    element.tail = ''
    if element.text:
      element.text = element.text.strip()
    for c in element:
      self.strip(c)

  # Write assembly to XML file at path
  def write(self, path):
    self.strip(self.tree.getroot())
    self.prettyprint(self.tree.getroot(), 0, False)
    self.tree.write(path)
    return self.tree
```

A.1.3 units.py

```
attributelookup = {
  'position': 'position',
  'rotation': 'rotation',
  'radius': 'length',
  'height': 'length',
  'lead': 'length',
  'pitch': 'length',
  'major_diameter': 'length',
  'minor_diameter': 'length',
  'pitch_diameter': 'length',
  'thread_angle': 'angle',
  'thread_height': 'length',
}

massconstants = {}
massconstants['milligram'] = {
  'milligram': 1,
```

```
  'gram': 1000,
  'kilogram': 1000000
}
massconstants['gram'] = {
  'milligram': 0.001,
  'gram': 1,
  'kilogram': 1000
}
massconstants['kilogram'] = {
  'milligram': 0.000001,
  'gram': 0.001,
  'kilogram': 1
}
lengthconstants = {}
lengthconstants['millimeter'] = {
  'millimeter': 1,
  'centimeter': 10,
  'meter': 1000,
  'kilometer': 1000000
}
lengthconstants['centimeter'] = {
  'millimeter': 0.1,
  'centimeter': 1,
  'meter': 100,
  'kilometer': 10000
}
lengthconstants['meter'] = {
  'millimeter': 0.001,
  'centimeter': 0.01,
  'meter': 1,
  'kilometer': 1000
}
lengthconstants['kilometer'] = {
  'millimeter': 0.000001,
```

```
  'centimeter': 0.00001,
  'meter': 0.001,
  'kilometer': 1
}

angleconstants = {}
angleconstants['degree'] = {
  'radian': 57.2958,
  'degree': 1
}
angleconstants['radian'] = {
  'radian': 1,
  'degree': 0.0174533
}

# Transform measurements in a file to another unit
def transformunits(part, newunits):
  currunits = {
    'mass': 'gram',
    'length': 'meter',
    'angle': 'degree'}
  for meta in part.iter('meta'):
    for units in meta.iter('units'):
      for unit in units:
        name = unit.tag
        value = unit.get('unit')
        currunits[name] = value

  for parts in part.iter('parts'):
    for element in parts.iter():
      for k, v in element.attrib.items():
        if k in attributelookup:
          typ = attributelookup[k]
          if typ == 'position':
```

```
    positions = [float(s) for s in v.split(',')]
    for i in range(len(positions)):
      newlength = newunits['length']
      lengthconstant = lengthconstants[newlength]
      u = currunits['length']
      positions[i] *= lengthconstant[u]
    pos = ''
    for i in positions:
      pos += str(i) + ','
    element.set('position', pos[0:-1])
    rot = ''
  elif typ == 'rotation':
    rotations = [float(s) for s in v.split(',')]
    for i in range(len(rotations)):
      newangle = newunits['angle']
      angleconstant = angleconstants[newangle]
      u = currunits['angle']
      rotations[i] *= angleconstant[u]
    rot = ''
    for i in rotations:
      rot += str(i) + ','
    element.set('rotation', rot[0:-1])
  elif typ == 'mass':
    value = float(v)
    newmass = newunits['mass']
    massconstant = massconstants[newmass]
    value *= massconstant[currunits['mass']]
    element.set(k, str(value))
  elif typ == 'length':
    value = float(v)
    newlength = newunits['length']
    lengthconstant = lengthconstants[newlength]
    value *= lengthconstant[currunits['length']]
    element.set(k, str(value))
```

```
        elif typ == 'angle':
          value = float(v)
          newangle = newunits['angle']
          angleconstant = angleconstants[newangle]
          value *= angleconstant[currunits['angle']]
          element.set(k, str(value))

for features in part.iter('features'):
  for element in features.iter():
    for k, v in element.attrib.items():
      if k in attributelookup:
        typ = attributelookup[k]
        if typ == 'position':
          positions = [float(s) for s in v.split(',')]
          for i in range(len(positions)):
            newlength = newunits['length']
            lengthconstant = lengthconstants['length']
            u = currunits['length']
            positions[i] *= lengthconstant[u]
          pos = ''
          for i in positions:
            pos += str(i) + ','
          element.set('position', pos[0:-1])
          rot = ''
        elif typ == 'rotation':
          rotations = [float(s) for s in v.split(',')]
          for i in range(len(rotations)):
            newangle = newunits['angle']
            angleconstant = angleconstants[newangle]
            u = currunits['angle']
            rotations[i] *= angleconstant[u]
          rot = ''
          for i in rotations:
            rot += str(i) + ','
```

```
            element.set('rotation', rot[0:-1])
          elif typ == 'mass':
            value = float(v)
            u = currunits['mass']
            value *= massconstants[newunits['mass']][u]
            element.set(k, str(value))
          elif typ == 'length':
            value = float(v)
            lc = lengthconstants[newunits['length']]
            value *= lc[currunits['length']]
            element.set(k, str(value))
          elif typ == 'angle':
            value = float(v)
            ac = angleconstants[newunits['angle']]
            value *= ac[currunits['angle']]
            element.set(k, str(value))

# Transform measurements in part to match units in
# parts_with_units
def transformunitsfrompart(part, part_with_units):
  newunits = {'mass': 'gram',
              'length': 'meter',
              'angle': 'degree'}
  for meta in part_with_units.iter('meta'):
    for units in meta.iter('units'):
      for unit in units:
        name = unit.tag
        value = unit.get('unit')
        newunits[name] = value
  transformunits(part, newunits)
```

A.2 Tree Search

A.2.1 parttree.py

```
# One node in a tree
class PartNode:

  def __init__(self,
               element,
               levels={},
               parent=None,
               frompart=False):
    # Set up necessary information for nodes for both
    # isomorphism and homeomorphism search and attribute
    # information for node matching and distance measure
    # calculation
    self.discovered = False
    # If node is being created from XML
    if not frompart:
      self.id = ''
      self.element = element
      self.parent = parent
      self.levels = levels
      if self.parent is not None:
        self.level = parent.level + 1
      else:
        self.level = 1
      if self.level not in levels.keys():
        levels[self.level] = []
      levels[self.level].append(self)
      self.tag = self.element.tag
      self.children = self.generate_children(levels)
      self.attributes = self.element.attrib
      if len(self.children) == 0:
```

```
      self.label = 0
      self.s = set()
      self.h = set()
      self.marked = True
      self.child_labels = [0]
    else:
      self.label = None
      self.s = set()
      self.h = set()
      self.marked = False
      self.child_labels = None
  # If node is being copied from an already existing
  # node
  else:
    part = element
    self.id = part.id
    if part.ids is not None:
      self.ids = part.ids.copy()
    else:
      self.ids = None
    self.element = part.element
    self.parent = parent
    self.levels = levels
    if self.parent is not None:
      self.level = parent.level + 1
    else:
      self.level = 1
    if self.level not in levels.keys():
      levels[self.level] = []
    levels[self.level].append(self)
    self.tag = self.element.tag
    self.children = self.generate_children(
                levels, True, part.children)
    self.attributes = self.element.attrib
```

```
    self.label = part.label
    self.s = part.s.copy()
    self.h = part.h.copy()
    self.marked = part.marked
    if part.child_labels is not None:
      self.child_labels = part.child_labels.copy()
    else:
      self.child_labels = None

# Merge a node with its child and absorb child's
# attributes
def merge_child(self):
  if len(self.children) != 1:
    return
  child = self.children[0]
  self.children = child.children

  self.attributes = self.attributes.copy()
  for k, v in child.attributes.items():
    if k not in self.attributes.keys():
      self.attributes[k] = v

  self.ids = self.ids.copy()
  if self.ids is not None:
    self.ids.extend(child.ids)

  if len(self.children) == 0:
    self.label = 0
    self.s = set()
    self.marked = True
    self.child_labels = [0]
  else:
    self.label = None
    self.s = set()
```

```
    self.marked = False
    self.child_labels = None

  for c in self.children:
    c.decrement_level()

# Lower level of all lower nodes during child merging
def decrement_level(self):
  self.level -= 1
  for c in self.children:
    c.decrement_level()

# Create children from XML
def generate_children(self,
                      levels,
                      frompart=False,
                      children=[]):
  if not frompart:
    node_with_children = []

    if self.tag == 'part':
      node_with_children = self.element.find('features')
    elif self.tag == 'compoundpart':
      node_with_children = self.element.find('parts')
    else:
      node_with_children = self.element

    return [PartNode(c, levels, self, False) for
                  c in node_with_children]
  else:
    return [PartNode(c, levels, self, True) for
                  c in children]

# Assign identification to a node
```

```
def identify(self, prefix, i, nodes):
  self.id = prefix + str(i)
  self.ids = [self.id]
  nodes[self.id] = self
  i += 1
  for c in self.children:
    i, nodes = c.identify(prefix, i, nodes)
  return i, nodes

# Assign edintification to node and children
def identify_nodes(self, prefix):
  i, nodes = self.identify(prefix, 0, {})
  return nodes

# Remove all nodes with one children for use in
# determining if two trees are homeomorphic
def compress(self):
  while len(self.children) == 1:
    self.merge_child()
  for c in self.children:
    c.compress()

# Get all ids of current node and any children
# that have been compressed
def getids(self, ids=set()):
  ids = ids.union(set(self.ids))
  for c in self.children:
    ids = c.getids(ids)
  return ids

def tostring(self, level=0):
  s = ''
  s += '␣␣'*level + self.tag + '␣' +
              self.element.get('name', '') + '\n'
```

```
    for c in self.children:
      s += c.tostring(level + 1)
    return s


# Representation of part or assembly as tree
class PartTree:

  def __init__(self, part, frompart=False):
    if not frompart:
      self.levels = {}
      self.root = PartNode(part, self.levels)
      self.height = len(self.levels.keys())
      self.element = self.root.element
      self.level = self.root.level
      self.tag = self.root.tag
      self.children = self.root.children
      self.attributes = self.root.attributes
    else:
      self.root = part
      self.levels = part.levels
      self.height = len(self.levels.keys())
      self.element = self.root.element
      self.level = self.root.level
      self.tag = self.root.tag
      self.children = self.root.children
      self.attributes = self.root.attributes

  # Assign identity to all nodes of tree
  def identify_nodes(self, prefix):
    i, nodes = self.root.identify(prefix, 0, {})
    return nodes

  def tostring(self):
```

```
    return self.root.tostring()
```

A.2.2 partisomorphism.py

```
from .parttree import PartTree as pt
from .parttree import PartNode as pn
from partstree.units import transformunitsfrompart
import xml.etree.ElementTree as et
import math
import networkx as nx

# Update labels of parents
def updateparents(l):
  for n in l:
    if n.parent.child_labels is None:
      n.parent.child_labels = []
    n.parent.child_labels.append(n.label)
    n.parent.child_labels.sort()

# Create labels for a node
def createchildlabelsdict(nodes):
  labels = {}
  for n in nodes:
    if n.label == 0:
      continue
    if tuple(n.child_labels) not in labels.keys():
      labels[tuple(n.child_labels)] = []
    labels[tuple(n.child_labels)].append(n)

  return labels

# Create the list of nodes at a given level of the tree
def createnewlevellist(s, scontents, level):
  l = []
```

```
  k = 0
  for childlabels in s:
    k += 1
    for n in scontents[childlabels]:
      n.label = k
      l.append(n)
  for n in level:
    if n.label == 0:
      l.insert(0, n)
  return l

# Determine if two trees are isomorphic
def structuralisomorphism(querytree, patterntree):
  if not isinstance(querytree, pt):
    querytree = pt(querytree.element)
  if not isinstance(patterntree, pt):
    patterntree = pt(patterntree.element)

  is_isomorphic = False

  if querytree.height is patterntree.height:
    is_isomorphic = True
    l1 = querytree.levels[querytree.height]
    l2 = patterntree.levels[patterntree.height]
    for level in range(querytree.height - 1, 0, -1):
      updateparents(l1)
      updateparents(l2)

      s1contents = createchildlabelsdict(
                   querytree.levels[level])
      s2contents = createchildlabelsdict(
                   patterntree.levels[level])

      s1 = sorted(set(s1contents.keys()))
```

```
      s2 = sorted(set(s2contents.keys()))

      if s1 != s2:
        is_isomorphic = False
        break
      else:
        l1 = createnewlevellist(s1,
                                s1contents,
                                querytree.levels[level])
        l2 = createnewlevellist(s2,
                                s2contents,
                                patterntree.levels[level])

    if is_isomorphic:
      if (len(querytree.root.child_labels) ==
            len(patterntree.root.child_labels)):
        if (querytree.root.child_labels !=
            patterntree.root.child_labels):
          is_isomorphic = False

  return is_isomorphic

# Find structural isomorphisms of pattern tree
# in query tree
def structuralisomorphisms(querytree,
                           patterntree,
                           isomorphisms):
  is_isomorphic = structuralisomorphism(querytree,
                        patterntree)

  if is_isomorphic:
    isomorphisms.append(querytree)
  else:
    for c in querytree.root.children:
```

```
      structuralisomorphisms(pt(c.element),
                             pt(patterntree.root.element),
                             isomorphisms)

# Find the lowest location of a homeomorphism on a branch
def lowesthomeomorphisms(tree, root, homeomorphisms):
  if root not in tree.s:
    return False

  lowest = True
  for c in tree.children:
    if root in c.s:
      lowest = False
      lowesthomeomorphisms(c, root, homeomorphisms)
  if lowest:
    homeomorphisms.append(tree)

# Determine if two trees are isomorphic
def isisomorphic(tree, pattern):
  if isinstance(tree, pt):
    tree = tree.root
  if isinstance(pattern, pt):
    pattern = pattern.root

  if len(tree.children) != len(pattern.children):
    return False
  for c in tree.children:
    for d in pattern.children:
      if isisomorphic(c, d) and not d.discovered:
        c.discovered = True
        d.discovered = True
        break

  i = True
```

```
  for c in tree.children:
    if not c.discovered:
      i = False
      break
  for c in tree.children:
    c.discovered = False
  for d in pattern.children:
    d.discovered = False
  return i

# Determine if two trees are homeomorphic
def ishomeomorphic(tree, pattern):
  if isinstance(tree, pt):
    tree = tree.root
  if isinstance(pattern, pt):
    pattern = pattern.root

  tree = pt(pn(tree, fromquery=True), True).root
  tree.compress()
  pattern = pt(pn(tree, fromquery=True), True).root
  pattern.compress()
  return isisomorphic(tree, pattern)

# Find all structurual homeomorphisms of a pattern tree
# within a query tree
def structuralhomeomorphisms(querytree,
                             patterntree,
                             homeomorphisms=[]):
  T = querytree.identify_nodes('T')
  P = patterntree.identify_nodes('P')

  # Step 1: get all leaves of P and set them to Sr(v)
  # for all leaves in T
  parents = set()
```

```
tleaves = set()
pleaves = set()
for k,v in P.items():
  if len(v.children) == 0:
    pleaves.add(k)
for k, v in T.items():
  if len(v.children) == 0:
    tleaves.add(k)
    parents.add(v.parent.id)
    v.s = pleaves.copy()

# Step 2 already done; step 3: use set of nodes to look
# into to see if they have all children marked
while len(parents) > 0:
  childrenmarked = []
  for v in parents:
    allmarked = True
    for c in T[v].children:
      if not c.marked:
        allmarked = False
        break
    if allmarked:
      childrenmarked.append(v)
  for v in childrenmarked:
    #compute Sr(v)
    # Step 1
    vchildren = [c.id for c in T[v].children]
    for c in T[v].children:
      T[v].s = T[v].s.union(c.s)

    # Step 2
    for k, u in P.items():
      uchildren = [c.id for c in u.children]
      bpgnodes = vchildren + uchildren
```

```
        bpg = nx.Graph()
        for i in range(len(bpgnodes)):
          bpg.add_node(i)
        for y in uchildren:
          for x in vchildren:
            if y in T[x].s:
              bpg.add_edge(bpgnodes.index(x),
                           bpgnodes.index(y))
        if (len(
            nx.algorithms.matching.maximal_matching(bpg))
                == len(uchildren)):
          T[v].s.add(k)

      T[v].marked = True
      if T[v].id[1:] != '0':
        parents.add(T[v].parent.id)
      parents.remove(v)

  # Final step: determine lowest nodes such
  # that the root of P is in Sr(v)
  root = querytree
  if isinstance(querytree, pt):
    root = querytree.root
  lowesthomeomorphisms(root, 'P0', homeomorphisms)
  if len(homeomorphisms) > 0:
    return True

# Get set of all possible node matchings between two
# matched isomorphic or homemorphic trees
def getmatchednodes(querychildren,
                    patternchildren,
                    level,
                    debug=False):
```

```
if not isinstance(querychildren, list):
  querychildren = [querychildren]
if not isinstance(patternchildren, list):
  patternchildren = [patternchildren]

if ((querychildren is None or len(querychildren) == 0)
      and (patternchildren is None
      or len(patternchildren) == 0)):
  return [[]]

allpossiblematches = []
for pc in querychildren:
  if isinstance(pc, pt):
    pc = pc.root
  matches = []
  for sc in patternchildren:
    if isinstance(sc, pt):
      sc = sc.root
    querychild = pn(pc, {}, fromquery=True)
    patternchild = pn(sc, {}, fromquery=True)
    checkformatches = False
    if level == 0:
      checkformatches = True
    elif ishomeomorphic(querychild, patternchild):
      checkformatches = True
      if (len(patternchild.children) !=
              len(querychild.children)
              and (len(patternchild.children) == 1
              or len(querychild.children) == 1)):
        querychild = pn(querychild, {}, fromquery=True)
        patternchild = pn(patternchild,
                          {}, fromquery=True)
        while (len(patternchild.children) !=
                  len(querychild.children)):
```

```
            if (len(patternchild.children) == 1
                    and len(querychild.children) != 1):
              patternchild.merge_child()
            elif (len(patternchild.children) != 1
                    and len(querychild.children) == 1):
              querychild.merge_child()
            else:
              checkformatches = False
              break
      else:
        checkformatches = False

      if checkformatches:
        match = getmatchednodes(
                        querychild.children,
                        patternchild.children,
                        level+1,
                        debug=debug)
        for i in range(len(match)):
          if match[i] is None:
            match[i] = []
          match[i].insert(0,
                  {'querynode': querychild,
                  'patternnode': patternchild,
                  'level': level})
        matches.extend(match)
    allpossiblematches.append(matches)

finalmatches = []
for i in range(len(allpossiblematches)):
  m = allpossiblematches[i]
  for j in range(len(allpossiblematches)):
    if i != j:
      for listofnodesx in m:
```

```
          for listofnodesy in allpossiblematches[j]:
            xids = set()
            yids = set()
            for n in listofnodesx:
              xids = xids.union(set(n['patternnode'].ids))
              xids = xids.union(set(n['querynode'].ids))
            for n in listofnodesy:
              yids = yids.union(set(n['patternnode'].ids))
              yids = yids.union(set(n['querynode'].ids))
            if len(xids.intersection(yids)) == 0:
              listofnodesx.extend(listofnodesy)
    finalmatches.extend(m)

  return finalmatches

# Calculate ratio distance
def calculateratios(matchednodes, jaccard, debug=False):
  totalsum = 0
  for m in matchednodes:
    intersection = 0
    union = 0
    ratiosum = 0
    samekeys = []
    # Determine size of intersection and union of
    # attribute keys
    for key in m['querynode'].attributes.keys():
      if not (key == 'name' or
              key == 'owner' or
              key == 'price' or
              key == 'address' or
              (m['level'] == 0 and
              (key == 'rotation' or key == 'position'))):
        union += 1
        if key in m['patternnode'].attributes.keys():
```

```
      intersection += 1
      samekeys.append(key)
for key in m['patternnode'].attributes.keys():
  if (key not in m['querynode'].attributes.keys() and
      not (key == 'name' or
           key == 'owner' or
           key == 'price' or
           key == 'address' or
           (m['level'] == 0 and
           (key == 'rotation' or key == 'position')))):
    union += 1

# Add to total sum of ratios for numerical attributes
for key in samekeys:
  a = m['querynode'].attributes[key]
  b = m['patternnode'].attributes[key]
  try:
    if (key == 'name' or
        key == 'owner' or
        key == 'price' or
        key == 'address' or
        (m['level'] == 0 and
        (key == 'rotation'
        or key == 'position'))):
      raise Exception()
    a = float(a)
    b = float(b)
  except:
    union -= 1
    intersection -= 1
    continue
  ratiosum += min(a, b) / max(a, b)

if intersection != 0:
```

```
      if jaccard:
        totalsum += (1 - (ratiosum / intersection)
                     * (intersection / union))
      else:
        totalsum += 1 - (ratiosum / intersection)
    elif union != 0:
      totalsum += 1

  return totalsum / len(matchednodes)

# Calculate cosine distance
def calculatecosine(matchednodes, jaccard, debug=False):
  totalsum = 0
  for m in matchednodes:
    intersection = 0
    union = 0
    a_ = 0
    b_ = 0
    samekeys = []
    # Determine size of union and intersection of
    # attribute keys
    for key in m['querynode'].attributes.keys():
      if not (key == 'name' or
              key == 'owner' or
              key == 'price' or
              key == 'address' or
              (m['level'] == 0 and
              (key == 'rotation' or
              key == 'position'))):
        union += 1
        if key in m['patternnode'].attributes.keys():
          intersection += 1
          samekeys.append(key)
    for key in m['patternnode'].attributes.keys():
```

```
  if (key not in m['querynode'].attributes.keys() and
      not (key == 'name' or
           key == 'owner' or
           key == 'price' or
           key == 'address' or
           (m['level'] == 0 and
           (key == 'rotation' or
           key == 'position')))):
    union += 1

ab = 0
# Calculate dot product of vector of numerical
# attributes for matched nodes
for key in samekeys:
  a = m['querynode'].attributes[key]
  b = m['patternnode'].attributes[key]
  try:
    if (key == 'owner' or
        key == 'price' or
        key == 'address' or
        (m['level'] == 0 and
        (key == 'rotation' or
        key == 'position'))):
      raise Exception()
    a = float(a)
    b = float(b)
  except:
    union -= 1
    intersection -= 1
    continue
  a_ += math.pow(a, 2)
  b_ += math.pow(b, 2)
  ab += a * b
```

```
    a_ = math.sqrt(a_)
    b_ = math.sqrt(b_)

    if intersection != 0 and union != 0:
      if jaccard:
        totalsum += ((2 - (intersection / union)) *
                     (1 - (ab / (a_ * b_))))
      else:
        totalsum += 1 - (ab / (a_ * b_))

  return totalsum / len(matchednodes)

# Calculate euclidean distance
def calculateeuclidean(matchednodes, jaccard, debug=False):
  totalsum = 0
  for m in matchednodes:
    # if m['level'] == 0:
    #   continue
    intersection = 0
    union = 0
    s = 0
    samekeys = []
    # Determine intersection and union for attribute
    # keys
    for key in m['querynode'].attributes.keys():
      if not (key == 'name' or
              key == 'owner' or
              key == 'price' or
              key == 'address' or
              (m['level'] == 0 and
              (key == 'rotation' or key == 'position'))):
        union += 1
        if key in m['patternnode'].attributes.keys():
          intersection += 1
```

```
      samekeys.append(key)
for key in m['patternnode'].attributes.keys():
  if (key not in m['querynode'].attributes.keys() and
      not (key == 'name' or
           key == 'owner' or
           key == 'price' or
           key == 'address' or
           (m['level'] == 0 and
           (key == 'rotation' or
           key == 'position')))):
    union += 1

for key in samekeys:
  a = m['querynode'].attributes[key]
  b = m['patternnode'].attributes[key]
  try:
    if (key == 'owner' or
        key == 'price' or
        key == 'address' or
        (m['level'] == 0 and
        (key == 'rotation' or
        key == 'position'))):
      raise Exception()
    a = float(a)
    b = float(b)
  except:
    union -= 1
    intersection -= 1
    continue
  s += math.pow(a - b, 2)

s = math.sqrt(s)

if intersection != 0 and union != 0:
```

```
      if jaccard:
        totalsum += (2 - (intersection / union)) * s
      else:
        totalsum += s

  return totalsum / len(matchednodes)

# Calculate distance between sets of matched nodes
def calculatedistance(matchednodes,
                      measure,
                      jaccard,
                      debug=False):

  if measure == 'ratio':
    return calculateratios(matchednodes,
                           jaccard, debug), measure
  elif measure == 'euclidean':
    return calculateeuclidean(matchednodes,
                            jaccard, debug), measure
  elif measure == 'cosine':
    return calculatecosine(matchednodes,
                            jaccard, debug), measure
  else:
    return 1, measure

# Calculate distances on attributes given set of
# isomorphisms/homeomorphisms
def attributeisomorphisms(isomorphisms,
                          pattern,
                          measure,
                          jaccard,
                          debug=False):
  isomorphisms_with_distances = []
  x = []
```

```
for i in isomorphisms:
  a = pn(i, fromquery=True)
  b = pn(pattern.root, fromquery=True)
  a.compress()
  b.compress()

  if not isisomorphic(pt(a, fromquery=True),
                      pt(b, fromquery=True)):
    continue

  i.identify_nodes('P')
  pattern.identify_nodes('S')
  matchednodes = getmatchednodes(i,
                                 pattern,
                                 0,
                                 debug=debug)

  iids = i.getids()
  patternids = pattern.root.getids()

  if debug:
    print('Matchings: ' + str(len(matchednodes)))

  distance = float('inf')
  # Determine lowest distance for possible matchings
  # of nodes
  for m in matchednodes:

    sids = set()
    pids = set()
    for n in m:
      sids = sids.union(set(n['querynode'].ids))
      pids = pids.union(set(n['patternnode'].ids))
```

```
      if (len(sids) != len(iids) or
          len(pids) != len(patternids)):
        continue
      d, y = calculatedistance(m,
                  measure, jaccard, debug=debug)
      x.append(y)
      if d < distance:
        distance = d
    if distance != float('inf'):
      isomorphisms_with_distances.append(
                {'distance': distance, 'part': i})

  return isomorphisms_with_distances, x

# Get isomorphisms/homeomorphisms of a pattern tree
# within a query tree
def partisomorphisms(query,
                     pattern,
                     measure,
                     jaccard,
                     debug=False,
                     verbose=False):
  isomorphisms = []

  querytree = pt(query)
  patterntree = pt(pattern)

  structuralhomeomorphisms(querytree,
                           patterntree,
                           isomorphisms)

  if debug:
    print('Found', len(isomorphisms), 'isomorphisms')
```

```
  isomorphismswithdistances, m = attributeisomorphisms(
                      isomorphisms,
                      patterntree,
                      measure,
                      jaccard,
                      debug=debug)

  if debug:
    for d in isomorphismswithdistances:
      print('Distance:␣' + str(d['distance']))

  for i in isomorphismswithdistances:
    i['owner'] = int(
          querytree.root.element.get('owner', '-1'))
    i['price'] = int(
          querytree.root.element.get('price', '0'))

  return isomorphismswithdistances, m

# Get isomorphisms/homeomorphisms from query and pattern
# file paths
def isomorphisms(queryfile,
                 patternfile,
                 measure,
                 jaccard,
                 debug=False,
                 verbose=False):
  query = et.parse(queryfile).getroot()
  pattern = et.parse(patternfile).getroot()
  transformunitsfrompart(pattern, query)

  return partisomorphisms(query,
                          pattern,
```

```
                              measure,
                              jaccard,
                              debug=debug,
                              verbose=verbose)

# Get isomorphisms/homeomorphisms from query filepath
# and pattern in XML string
def isomorphismsfromstring(queryfile,
                              pattern,
                              measure,
                              jaccard,
                              debug=False,
                              verbose=False):
  query = et.parse(queryfile).getroot()
  pattern = et.fromstring(pattern)
  transformunitsfrompart(pattern, query)

  return partisomorphisms(query,
                             pattern,
                             measure,
                             jaccard,
                             debug=debug,
                             verbose=verbose)

# Get isomorphisms/homeomorphisms from query and pattern
# XML strings
def isomorphismsfromstrings(query,
                              pattern,
                              measure,
                              jaccard,
                              debug=False,
                              verbose=False):
  query = et.fromstring(query)
  pattern = et.fromstring(pattern)
```

```
  transformunitsfrompart(pattern, query)

  return partisomorphisms(query,
                          pattern,
                          measure,
                          jaccard,
                          debug=debug,
                          verbose=verbose)

# Get isomorphisms/homeomorphisms of subtrees of a pattern
# tree within a query tree
def subtreeisomorphisms(isomorphisms,
                        query,
                        patternchildren,
                        measure,
                        jaccard,
                        debug=False,
                        verbose=False):
  m = []
  for c in patternchildren:
    i, m = partisomorphisms(query,
                            c.element,
                            measure,
                            jaccard,
                            debug,
                            verbose)
    for subi in i:
      subi['partialmatchedpattern'] = c.element
    isomorphisms.extend(i)
    subtreeisomorphisms(isomorphisms,
                        query,
                        c.children,
                        measure,
                        jaccard,
```

```
                          debug,
                          verbose)
  return isomorphisms, m

# Get isomorphisms/homeomorphisms of all subtrees of a
# pattern tree within a query tree from filepaths
def isomorphismspartial(queryfile,
                          patternfile,
                          measure,
                          jaccard,
                          debug=False,
                          verbose=False):
  query = et.parse(queryfile).getroot()
  pattern = et.parse(patternfile).getroot()
  transformunitsfrompart(pattern, query)

  pattern = pt(pattern).root

  isomorphisms = []
  i, m = subtreeisomorphisms(isomorphisms,
                             query,
                             pattern.children,
                             measure,
                             jaccard,
                             debug=debug,
                             verbose=verbose)
  return isomorphisms, m

# Get isomorphisms/homeomorphisms of all subtrees of a
# pattern tree from XML string within a query tree from
# filepath
def isomorphismspartialfromstring(queryfile,
                                  pattern,
                                  measure,
```

```
                                    jaccard,
                                    debug=False,
                                    verbose=False):
  query = et.parse(queryfile).getroot()
  pattern = et.fromstring(pattern)
  transformunitsfrompart(pattern, query)

  pattern = pt(pattern).root

  isomorphisms = []
  i, m = subtreeisomorphisms(isomorphisms,
                             query,
                             pattern.children,
                             measure,
                             jaccard,
                             debug=debug,
                             verbose=verbose)
  return isomorphisms, m

# Get isomorphisms/homeomorphisms of all subtrees of a
# pattern tree in a query tree from XML strings
def isomorphismspartialfromstrings(query,
                                   pattern,
                                   measure,
                                   jaccard,
                                   debug=False,
                                   verbose=False):
  query = et.fromstring(query)
  pattern = et.fromstring(pattern)
  transformunitsfrompart(pattern, query)

  pattern = pt(pattern).root

  isomorphisms = []
```

```
  i, m = subtreeisomorphisms(isomorphisms,
                             query,
                             pattern.children,
                             measure, jaccard,
                             debug=debug,
                             verbose=verbose)
  return isomorphisms, m
```

A.2.3 partsearch.py

```
from .parttree import PartTree as pt
from partstree.partisomorphism import (
        isomorphisms as partisomorphisms,
        isomorphismsfromstring,
        isomorphismspartialfromstring)
from partstree.units import transformunitsfrompart
import xml.etree.ElementTree as et
from glob import glob

# Search for isomorphisms/homeomorphisms given a filepath
# for pattern tree and a directory to search for query
# trees
def searchisomorphisms(directory,
                       partpath,
                       measure="euclidean",
                       jaccard=True):
    parts = glob(directory)
    isomorphisms = []

    for part in parts:
        returnedisomorphisms, m = partisomorphisms(part,
                                                   partpath,
                                                   measure,
                                                   jaccard)
```

```
        for p in returnedisomorphisms:
            p['file'] = part
        isomorphisms.extend(returnedisomorphisms)

    return sorted(isomorphisms,
            key=lambda x: x['distance'], reverse=True), m

# Search for isomorphisms/homeomorphisms given an XML
# string for pattern tree and a directory to search for
# query trees
def searchisomorphismsfromstring(directory,
                                 partstring,
                                 measure="euclidean",
                                 jaccard=True):
    parts = glob(directory)
    isomorphisms = []

    for part in parts:
        returnedisomorphisms, m = isomorphismsfromstring(
                    part, partstring, measure, jaccard)
        for p in returnedisomorphisms:
            p['file'] = part
        isomorphisms.extend(returnedisomorphisms)

    return sorted(isomorphisms,
                key=lambda x: x['distance']), m

# Search for isomorphisms/homeomorphisms given an XML
# string for pattern tree and a directory to search for
# query trees looking at all subtrees of pattern tree as
# well
def searchisomorphismsfromstringpartial(
                            directory,
                            partstring,
```

```
                            measure="euclidean",
                            jaccard=True):
    parts = glob(directory)
    isomorphisms = []
    partialisomorphisms = []

    for part in parts:
        returnedisomorphisms, m = isomorphismsfromstring(
                    part, partstring, measure, jaccard)
        partial, m = isomorphismspartialfromstring(
                    part, partstring, measure, jaccard)
        for p in returnedisomorphisms:
            p['file'] = part
        for p in partial:
            p['file'] = part
        isomorphisms.extend(returnedisomorphisms)
        partialisomorphisms.extend(partial)

    return (sorted(isomorphisms,
                key=lambda x: x['distance']),
                sorted(partialisomorphisms,
                        key=lambda x: x['distance']), m)
```

A.3 Server - Search

A.3.1 server.py

```
from flask import Flask, request, jsonify
import json
import partstree.partsearch as partsearch
import xml.etree.ElementTree as et
from glob import glob
import time
```

```
app = Flask(__name__)

# Determine owner of a subtree
def getowner(result):
    if result.element.get('owner') is not None:
        return result.element.get('owner')
    return getprice(result.parent)

# Determine price of a subtree
def getprice(result):
    if result.element.get('price') is not None:
        return result.element.get('price')
    return getprice(result.parent)

# Determine contract address of a subtree
def getaddress(result):
    if result.element.get('address') is not None:
        return result.element.get('address')
    return getaddress(result.parent)

# Determine children parts of a node
def getchildren(result):
    children = []
    for c in result.children:
        if c.tag == 'part' or c.tag == 'compoundpart':
            children.append({'price': getprice(c),
                             'owner': getowner(c),
                             'children': getchildren(c)})
        else:
            children.extend(getchildren(c))
    return children


@app.route('/search', methods=['POST'])
```

```
def search():
    data = request.get_json()

    directory = 'Data/Parts/Test/Compound/*.xml'
    partstring = data.get('part')
    measure = data.get('measure')

    start = time.time()
    results, partialresults, m = (
        partsearch.searchisomorphismsfromstringpartial(
                directory, partstring, measure))
    print(time.time() - start)

    parts = [{'part': et.tostring(r['part'].element,
                                  encoding='unicode'),
              'address': getaddress(r['part']),
              'price': getprice(r['part']),
              'distance': r['distance'],
              'file': r['file'][(len(directory) - 5):]}
                                      for r in results]
    partialparts = [{'part': et.tostring(r['part'].element,
                                 encoding='unicode'),
              'address': r['part'].element.get(
                                 'address', ''),
              'price': getprice(r['part']),
              'distance': r['distance'],
              'file': r['file'][(len(directory) - 5):],
              'partialmatchedsubpart': et.tostring(
                        r['partialmatchedsubpart'],
                        encoding='unicode')}
                        for r in partialresults]

    return jsonify({'results': parts,
                    'partialresults': partialparts})
```

A.4 User Interface

A.4.1 index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import * as serviceWorker from './serviceWorker';

ReactDOM.render(<App />, document.getElementById('root'));

serviceWorker.unregister();
```

A.4.2 App.js

```
import React, { useState } from 'react';
import Web3 from 'web3';
import { BuyABI } from './PurchaseABI'
import { bc } from './bc'
import Header from './components/Header'
import Upload from './components/Upload'
import Results from './components/Results'
import axios from 'axios'

import "./App.css";

const web3 = new Web3(new
    Web3.providers.HttpProvider("HTTP://127.0.0.1:7545"));
web3.eth.defaultAccount = web3.eth.accounts[0];

const acc = [
  '0x1d0ef33691fd358A9E3289842935E48946f52b8d',
  '0x081BCa6b9D81cca11E2b4e3D7f00E205b19fE769',
```

```
  '0xb7b0Fd435DbFb7616523Eb3a41a31525802B7507',
  '0x6051517c02e5B2fB3EA9f705a6DF4645E4AD64d4',
  '0x9a6BBD72dC73782f53692615B44300Ae62741e80',
  '0x480F12fcfb966a344bd90704f957ebaDa7bd034C',
  '0x73B9a084F104dCca7e5a318216e4300bC5234B5c',
  '0xC54Ad0e63d5Fc3571222CCa9401Edf008B3eF933',
  '0x8D2D3eb9CB5c288BD203f0B230B029C962ef33aC',
  '0xB071F65dbbE83895D6ce5f6D19c3C077421783E7'
]

const getPrice = (owner, price, children, prices) => {
  children.forEach((child) => {
    price -= getPrice(child.owner, child.price,
                      child.children, prices)
  })
  prices[owner] += price
  return price
}

class App extends React.Component {
  constructor() {
    super()

    this.state = {
      results: undefined,
      path: '',
      file: null,
      searching: false,
      purchasing: false
    }
  }

  buy = async result => {
    const accounts = await window.ethereum.enable()
```

```
  const account = accounts[0];

  const BuyContract = new web3.eth.Contract(
    BuyABI
  );

  const address = result.address

  this.setState({purchasing: true})
  var gas = await BuyContract.deploy({data: bc,
          arguments: [address]}).estimateGas()
  var newContract = await BuyContract.deploy({data:
          bc, arguments: [address]}).send(
                {from: account, gas })

  const cost = web3.utils.toWei(result.price.toString(),
                                    'ether')
  gas = 6721975
  const res = await newContract.methods
    .transfer()
    .send({ from: account, gas: gas,
            value: cost.toString() }).then(() => {
      alert('Purchased successfully!')
      this.setState({purchasing: false})
  }, res => {
      alert('Transaction unsuccessful')
      this.setState({purchasing: false})
  })
}

submitSearch = () => {
  const measure = 'ratio'
  const part = this.state.file
  this.setState({searching: true, results: undefined})
```

```
  axios.post('/search', {part, measure}).then(res => {
    console.log(res.data)
    this.setState({results: res.data.results,
                   searching: false})
  }, res => {
    this.setState({searching: false})
  })
}

setFile = (path, file) => {
  var reader = new FileReader();
  reader.onload = e => {
    this.setState({
      path,
      file: reader.result
    })
  };
  reader.readAsText(file);
}

resetFile = () => {
  this.setState({
    path: '',
    file: null
  })
}

render() {
  return (
    <div className="App">
      <Header />
      <Upload file={this.state.path}
              setFile={this.setFile}
              submitSearch={this.submitSearch} />
```

```
        {
          this.state.searching ? <h6>Searching...</h6> :
          this.state.purchasing ? <h6>Purchasing...</h6> :
          ''
        }
        {
          typeof this.state.results !== 'undefined' ?
          <Results path={this.state.path}
                   results={this.state.results}
                   buy={this.buy} /> :
            ''
        }
      </div>
    );
  }
}

export default App;
```

A.4.3 Header.jsx

```
import React from 'react'
import { AppBar,
         Typography, Toolbar } from '@material-ui/core'

class Header extends React.Component {

  render() {

    return (
      <div>
        <AppBar position="static">
          <Toolbar>
            <Typography variant="h6">
```

```
              Part Search
            </Typography>
          </Toolbar>
        </AppBar>
      </div>
    )
  }
}

export default Header
```

A.4.4 Results.jsx

```
import React from 'react'
import { Button, Divider,
     List, ListItem, ListItemText,
     ListItemSecondaryAction } from '@material-ui/core'

var format = require('xml-formatter')

class Results extends React.Component {

    openWindow = result => {
        var myWindow = window.open("",
                result.file, "width=500,height=300");
        const p =
            format(result.part).replace(/&/g, '&amp;')
        .replace(/</g, '&lt;')
        .replace(/>/g, '&gt;')
        .replace(/"/g, '&quot;')
        .replace(/'/g, '&apos;');
        myWindow.document.write(p);
    }
```

```
render() {
    const { results, path } = this.props;
    const displayResults =
            results.map((result, index) => {
        return (<div key={index}>
            <ListItem button
                    onClick={e => {e.preventDefault();
                    this.openWindow(result)}}>
            <ListItemText primary={result.file} />
            <ListItemText
                secondary={'Distance: ' +
                            result.distance} />
            <ListItemText
                secondary={'Price: ' + result.price} />
            <ListItemSecondaryAction>
                <Button
                    onClick={e =>
                        this.props.buy(result)}>Buy
                </Button>
            </ListItemSecondaryAction>
        </ListItem><Divider /></div>)
    })

    const content = results.length === 0 ?
            <h2>No results</h2> :
    <List>
        {displayResults}
    </List>
    return (
        <div>
            {content}
        </div>
    )
}
```

```
}

export default Results
```

A.4.5 Upload.jsx

```
import React from 'react'
import { makeStyles } from '@material-ui/core/styles';
import { AppBar, Button, Typography, Toolbar }
            from '@material-ui/core'
import Dropzone from 'react-dropzone'

const style = {
    backgroundColor: '#e0e0e0',
    cursor: 'pointer',
    width: 'fit-content',
    height: 'fit-content',
    padding: '5px',
    margin: 'auto',
    borderRadius: '3px',
    marginTop: 10
}

class Upload extends React.Component {
    constructor() {
        super()
    }

    onDrop = files => {
        this.props.setFile(files[0].name, files[0])
    }

    render() {
        const file = this.props.file
```

```
        return (
            <div>
                <Dropzone onDrop={this.onDrop}>
                    {({getRootProps, getInputProps}) => (
                        <section>
                        <div {...getRootProps()}
                                style={style}>
                            <input {...getInputProps()} />
                            {file === '' ?
                                <h3>Upload part</h3> :
                            <h3>{file}</h3>}
                        </div>
                        </section>
                    )}
                </Dropzone>
                <div style={{marginTop: 10}}>
                    <Button
                        onClick={this.props.submitSearch}
                        disabled={file === ''}>
                        Search
                    </Button>
                </div>
          </div>
        );
    }
}

export default Upload
```

A.5 Server - Publish

A.5.1 index.js

```
const express = require('express')
const app = express()
const Web3 = require('web3')

const PublishABI = require('PublishABI')
const bc = require('bc')

const web3 = new Web3(new Web3.providers.HttpProvider
        ("HTTP://127.0.0.1:7545"));
web3.eth.defaultAccount = web3.eth.accounts[0];

const acc = [
        '0x1d0ef33691fd358A9E3289842935E48946f52b8d',
        '0x081BCa6b9D81cca11E2b4e3D7f00E205b19fE769',
        '0xb7b0Fd435DbFb7616523Eb3a41a31525802B7507',
        '0x6051517c02e5B2fB3EA9f705a6DF4645E4AD64d4',
        '0x9a6BBD72dC73782f53692615B44300Ae62741e80',
        '0x480F12fcfb966a344bd90704f957ebaDa7bd034C',
        '0x73B9a084F104dCca7e5a318216e4300bC5234B5c',
        '0xC54Ad0e63d5Fc3571222CCa9401Edf008B3eF933',
        '0x8D2D3eb9CB5c288BD203f0B230B029C962ef33aC',
        '0xB071F65dbbE83895D6ce5f6D19c3C077421783E7'
  ]

const defaultAccount = acc[0]

app.use(express.json())
app.use(express.urlencoded())

app.post('/publish', (req, res) => {
```

```
    const PartContract = new web3.eth.Contract(
      PublishABI
    );
    const addresses = []

    const parts = req.body.parts
    parts.forEach(async (part, index) => {
        const cost = web3.utils.toWei(
                        part.cost.toString(), 'ether')
        var owner = part.owner
        if (typeof owner === 'number') {
            owner = acc[owner]
        }
        const args = [owner, part.children, cost]

        var gas = await PartContract.deploy(
        {data: bc, arguments: args}).estimateGas()
        var newContract = await PartContract
            .deploy({data: bc, arguments: args})
            .send({ from: acc[0], gas })
            .then(x => {
                addresses.push(x._address)
                if (addresses.length === parts.length) {
                    res.json({addresses})
                }
            })
    })
})

app.listen(3001, () =>
        console.log('Listening on port 3001'));
```

A.6 Smart Contracts

A.6.1 Publish.sol

```
pragma solidity ^0.5.11;
contract Publish {

    address public owner;
    address[] public childrenCons;
    mapping (address => uint) public contractNums;
    mapping (address => bool) public contractExists;
    uint public cost;

    constructor(address _owner,
                address[] memory _childrenCons,
                uint _cost) public {
        owner = _owner;
        cost = _cost;

        for (uint i=0; i < _childrenCons.length; i++) {
            childrenCons.push(_childrenCons[i]);
            contractExists[_childrenCons[i]] = true;
            contractNums[_childrenCons[i]]++;

            Part child = Part(_childrenCons[i]);
            address[] memory grandchildren =
                            child.getChildrenCons();
            for (uint j=0; j < grandchildren.length; j++) {
                if (!contractExists[grandchildren[j]]) {
                    contractExists[grandchildren[j]] =
                                true;
                    childrenCons.push(grandchildren[j]);
                }
                contractNums[grandchildren[j]]++;
```

```
            }
        }
    }

    function getOwner() view public returns (address) {
        return owner;
    }

    function getChildrenCons() view public returns
                            (address[] memory) {
        return childrenCons;
    }

    function getNums() view public returns
                        (uint[] memory) {
        uint[] memory nums = new
                    uint[](childrenCons.length);
        for (uint i=0; i < childrenCons.length; i++) {
            nums[i] = contractNums[childrenCons[i]];
        }
        return nums;
    }

    function getCost() view public returns (uint) {
        return cost;
    }
}
```

A.6.2 Purchase.sol

```
pragma solidity ^0.5.11;
contract Purchase {

    address public contractOwner;
```

```
address[] public toPay;
uint[] public costs;

constructor(address contractAddress) public {
    contractOwner = msg.sender;

    Part part = Part(contractAddress);
    address owner = part.getOwner();
    address[] memory childrenCons =
                    part.getChildrenCons();
    uint[] memory nums = part.getNums();
    uint cost = part.getCost();
    toPay = new address[](childrenCons.length + 1);
    costs = new uint[](childrenCons.length + 1);
    toPay[0] = owner;
    costs[0] = cost;

    for (uint i=1; i <= childrenCons.length; i++) {
        part = Part(childrenCons[i-1]);
        owner = part.getOwner();
        cost = part.getCost();
        toPay[i] = owner;
        costs[i] = cost * nums[i-1];
    }
}

function transfer() payable public returns (bool) {

    address payable owner =
            address(uint160(toPay[0]));
    for (uint i=0; i < toPay.length; i++) {
        owner = address(uint160(toPay[i]));
        owner.transfer(costs[i]);
    }
```

```
        return true;
    }
}
```