Intelligent Retrieval of Solid Models

A Thesis

Submitted to the Faculty

of

Drexel University

by

Vincent A. Cicirello

in partial fulfillment of the

requirements for the degree

of

Master of Science

in

Computer Science

June 1999

Dedication

“What is now proved was once only imagin’d.” — William Blake

–

To my parents Vincent J. and Loretta Cicirello, my sisters Deborah Cicirello and Donna Fitzstephens, my grandmother Regina Mingarino Sr., and my entire family for all of your guidance and encouragement throughout the years. The future grows from what we imagine in the present. Thank you for encouraging me to strive to reach my goals and dreams.

–

Also to my grandfather Eugene Mingarino. You are remembered with love.

Acknowledgements

Thanks are extended to Dr. William Regli, director of the Geometric and Intelligent Computing Laboratory (GICL) in Drexel University's Department of Mathematics and Computer Science, for his advisory role over this work for without which this work would not be possible. Thanks also to the other members of my thesis committee, Dr. Lloyd Greenwald, Dr. Spiros Mancoridis, and Dr. Ljubomir Perkovic for their time, knowledge, and opinions.

Thanks are extended to Dr. Steve Brooks of Allied Signal Corporation, Federal Manufacturing Technologies Program, in Kansas City for providing the National Design Repository with the ACIS models for the TEAM parts. Thanks also to Alexei Elinson for providing the code to generate random TEAM-like solid models.

This work was supported in part by National Science Foundation (NSF) CAREER Award CISE/IIS-9733545 and Grant ENG/DMI-9713718. Additional support was provided by the National Institute of Standards and Technology (NIST) under Grant 60NANB7D0092.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation or the other supporting government and corporate organizations.

Table of Contents

List of Tables

List of Figures

Abstract

Intelligent Retrieval of Solid Models

Vincent A. Cicirello

Advisor: William C. Regli

Nearly all major commercial computer-aided design systems have adopted a feature-based design approach to solid modeling. Models are created via a sequence of operations that apply design features to incremental versions of a design model. Even surfacing, free-form surface shaping, and deformation operations are internally represented in modeling systems as features in a "history tree" that generates the final design. Much in the same manner that Constructive Solid Geometry (CSG) trees for an individual model can be non-unique, these design feature histories for solid models might be ordered in a number of ways and still result in the same final geometry and topology. Manufacturing features, easily obtained from the use of a feature recognition system, often map simply to manufacturing operations such as milling operations for some machine tool.

This problem is formulated symbolically and geometric reasoning techniques are presented to generate a representation of features and feature dependencies that deals with the non-uniqueness problem encountered in design feature histories. It is shown that this representation is not limited to design features and can be used with manufacturing features as well. The representation defined is termed the Model Dependency Graph (MDG) and alternatively the Undirected Model Dependency Graph (UMDG) and is used as a basis for developing techniques for managing databases of solid models. Using the MDG, algorithms are introduced that can assess the similarity of solid models based on design or manufacturing features and can be used in the retrieval of these models. One of these algorithms computes an approximation to the subgraph isomorphism and graph isomorphism

problems using a random restart gradient descent approach. Another of these algorithms uses the search method known as A* to detect subgraph isomorphism. It is believed that these techniques can be used to build intelligent CAD knowledge bases and to identify meaningful part families from large sets of designs. Lastly, experimental results and performance metrics for these approaches are described. It is shown empirically that although the worst case complexity of solutions to the subgraph isomorphism problem is exponential the described algorithms' performance on random graphs and on the UMDG is tractable in practice.

Chapter 1

Introduction

1.1 Problem Statement

CAD databases and knowledge-bases are at the core of the modern engineering enterprise. These emerging digital libraries store all information relevant over a product's life-cycle (geometry, topology, features, revisions, etc.). An overview of the problem can be seen in Figure 1.1 which shows a database of solid models. Given this database, a designer may need to determine if a given design is contained in this database. Or a design engineer might encounter a problem of case-based design: how can this designer find previous design cases based on how similar they are to some new solid model? Given the solid model of some new part, an engineer might need to design a plan for the manufacture of this new part. There might be other parts similar to this new part in design and structure stored in the CAD repository. How can the engineer find the manufacturing plans for these similar parts efficiently? Or how can a case-based planning system find similar parts from which plans for some new part can be derived? These are all important questions.

The goal of this research is to develop algorithmic techniques to manage databases of CAD and Solid Models. To accomplish this goal, techniques for the intelligent retrieval of solid models will be described. Data structures for the representation of features are developed. A *feature* is a structural property or a volumetric property of the solid model.

The lack of standard representation schemes for CAD data and features data has been under significant study. A representation of features data and feature interactions that allows for the efficient retrieval of CAD and Solid Models from knowledge-bases is important. The representation developed in the present work is a graph based representation of the dependencies between features of the CAD model.

Why is a representation of features and feature interactions relevant to solid model retrieval? Features, whether they are design features or manufacturing features, represent structural properties of the solid model. So in this way, a representation of features data is a representation of the structure of the model. If two solid models have similar features and feature interactions, then in some way the two solid models are similar.

Given this representation scheme, efficient algorithms for performing this retrieval and for comparing solid models based on this representation becomes key to solving this problem. Algorithms for the comparison of solid models are described. These algorithms apply the search techniques of A* and gradient descent to the problems of graph isomorphism and subgraph isomorphism. These algorithms are used to compare the graph representation of features and feature interactions that is developed in this work.

In stating that I will incorporate a representation of features and feature interactions in my approach, you may ask “what is meant by features?” I already stated that a feature is some structural property of the solid model. But this is a very general definition of a feature. Many people have many different views of what a feature is. This work attempts to abstract itself from these varying definitions and to be independent on the class of feature in question. That is, the representation discussed may be applied to design features obtained from the design feature history for a given model and alternatively the representation may be applied to manufacturing features that may be obtained by running a feature recognizer over a collection of CAD models. It may also be desirable to represent both the design features and the manufacturing features and have multiple views of the data in the given

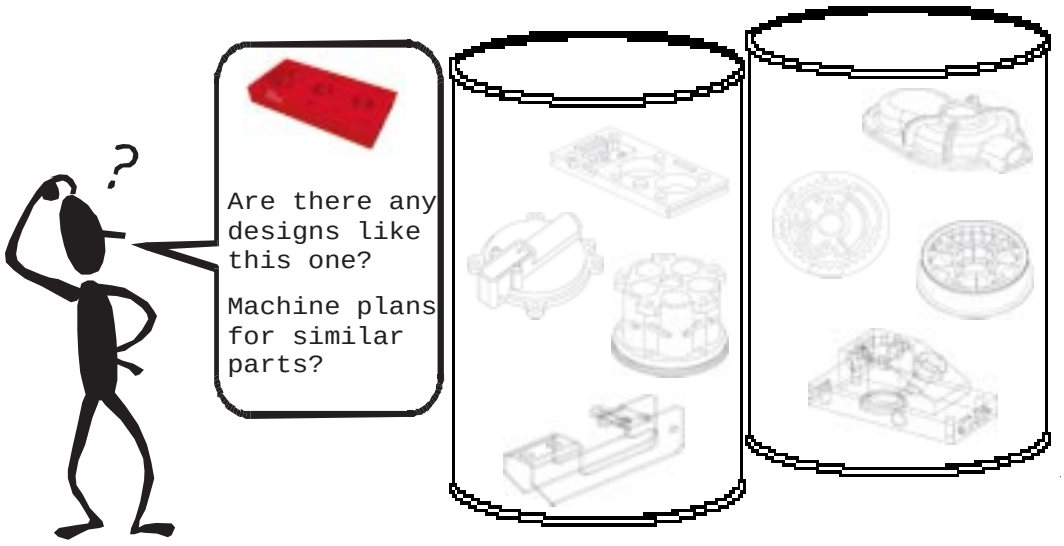


Figure 1.1: Overview of the problem of solid model retrieval.

CAD knowledge-base.

Nearly all major commercial computer-aided design systems have adopted a feature-based design approach to solid modeling. Thus design feature data is easily obtained from the CAD system in use. Models are created via a sequence of operations that apply design features to incremental versions of a design model. Surfacing, free-form surface shaping, and deformation operations are internally represented in modeling systems as features in a "history tree" that generate the final design. However, much in the same manner that Constructive Solid Geometry (CSG) trees for an individual model can be non-unique, these design feature histories for CAD models might be ordered in a number of ways and result in the same final geometry and topology.

In order to efficiently store and retrieve solid models from a CAD knowledge-base, one requires a more uniform representation for the feature information used to describe the artifact. Following, techniques are presented for dealing with ambiguity and variation that are independent of feature definition. Given that, with a fixed feature library, one might

be able to design an artifact in several alternative ways, techniques are presented to convert these orderings into a representation that removes the ambiguity that results from the ordering inherent in the feature history. This representation may also be constructed from manufacturing features obtained from a feature recognition stage along with an interaction analysis. Once reduced to this form, solid models can be more efficiently hashed or indexed for storage.

1.2 Overview of Approach

I present geometric reasoning techniques to generate an abstraction of a CAD model's design feature history that deals with the ambiguity and non-uniqueness inherent in the ordering of the design feature history. These same techniques may also be applied to manufacturing features to result in an alternative view of the CAD knowledge-base. The representation described is called the *Model Dependency Graph* (MDG) and alternatively the *Undirected Model Dependency Graph* (UMDG). These schemes represent features as nodes and feature interactions as edges between the nodes representing the features of the interaction. Based on the MDG and UMDG, I introduce algorithms that can assess the similarity of solid models based on features; index models for database storage; and identify meaningful part families from large sets of designs, such as are stored in engineering databases. The algorithms described include an inexact solution to the subgraph isomorphism and graph isomorphism problems using a gradient descent approach that allows for a measure of similarity. Also described is a fast A* algorithm for the subgraph isomorphism problem that I call the *A* Subgraph Isomorphism Checker* (ASIC).

1.3 Outline of Thesis

This paper is organized as follows: Chapter 2 provides an overview of related work and background in graph matching algorithms, solid modeling and feature-based modeling, and search algorithms. Chapter 3 presents the formulation of the problem of ambiguous design history trees and introduces an approach to addressing it based on constraint and graph algorithms, including defining the MDG and UMDG and describing the gradient descent approach to the problem and the ASIC algorithm. Chapter 4 describes experimental results on both randomly generated graphs and on UMDGs of randomly generated CAD models. Chapter 5 presents conclusions and plans for future work.

Chapter 2

Background

This chapter presents a summary of previous research work in the areas related to the work of this thesis. In this thesis, I reduce the problem of retrieving solid models based on similarity to various graph matching problems including graph isomorphism, subgraph isomorphism, and directed graph D-morphism. This chapter begins with a history and overview of previous algorithmic solutions to the graph isomorphism and subgraph isomorphism problems. Next, I present relevant research and background of solid modeling and feature-based design, including the topics of solid model representations and feature recognition. Finally, I present various search techniques that I will later use in presenting my solutions to the problem. These include A* search and hill-climbing or gradient descent search.

2.1 Graph Matching

A graph matching problem is a problem involving some form of comparison between graphs. Graph matching problems of varying types are important in a wide array of application areas. Some of the many application areas of such problems include information retrieval, sub-circuit identification, chemical structure classification, and networks. Problems of efficient graph matching arise in any field that may be modeled with graphs. For example, any problem that can be modeled with binary relations between entities in the

domain is such a problem. The individual entities in the problem domain become nodes in the graph. And each binary relation becomes an edge between the appropriate nodes.

Graph matching is a very difficult problem. The *graph isomorphism* problem is to determine if there exists a one-to-one mapping from the nodes of one graph to the nodes of a second graph that preserves adjacency. Similarly, the *subgraph isomorphism* problem is to determine if there exists a one-to-one mapping from the nodes of a given graph to the nodes of a subgraph of a second graph that preserves adjacency. The *largest common subgraph* problem is to find the largest subgraphs of two given graphs such that the subgraphs are isomorphic. The *digraph D-morphism* problem is to determine if there exists a one-to-one mapping from the nodes of one directed graph to the nodes of a second directed graph that preserves adjacency if you disregard the directions of the arcs. The closely related problems of *subgraph isomorphism*, *largest common subgraph*, and *digraph D-morphism* are known to be NP-complete [13]. Whether or not the *graph isomorphism* problem is in the class of NP-complete problems is an open question [13]. Although there do exist special cases of each of these problems that can be solved in polynomial time, there do not exist known algorithms of polynomial complexity to solve these problems in the general case. Therefore, the search for more efficient solutions to these problems is of great importance.

The best known algorithm for the graph isomorphism problem runs in $2^{O(n*\log n)}$ time in the worst case [4, 42]. One class of graphs that poses particular problems for graph isomorphism algorithms is the class of strongly regular graphs. A special case that is solvable in polynomial time is planar graph isomorphism. An $O(n * \log n)$ algorithm for planar graph isomorphism can be found in [21] and a linear time solution in [22]. There are other special cases that are solvable in polynomial time that involve a bound on some nodal property of the graphs. However, the degree of the polynomial time complexity of these algorithms is typically dependent on the value of the bound on the given nodal property. So therefore unless the bound on the given nodal property is small, these approaches do not seem

very practical. A few examples of such cases include graphs of bounded valence [26], k-contractible graphs [33], and graphs that are pairwise k-separable [32].

2.1.1 Definitions and Background

Graph Isomorphism. The graphs $G = (V_1, E_1)$ and $H = (V_2, E_2)$ are *isomorphic* if there exists a one-to-one mapping between their node sets V_1 and V_2 that preserves adjacency. The *graph isomorphism problem* asks whether or not there exists such a mapping between a given pair of graphs. The *graph isomorphism problem* can be formally defined as in [13]: "is there a one-to-one onto function $f : V_1 \rightarrow V_2$ such that $\{u, v\} \in E_1$ if and only if $\{f(u), f(v)\} \in E_2$?" Isomorphism is an equivalence relation on graphs.

Subgraph Isomorphism. The *subgraph isomorphism problem*, given graphs $G = (V_1, E_1)$ and $H = (V_2, E_2)$, asks whether G contains a subgraph isomorphic to H. It is formally defined in [13] by the question of the existence of a subset $V \subseteq V_1$ and a subset $E \subseteq E_1$ such that $|V| = |V_2|$, $|E| = |E_2|$, and there exists a one-to-one function $f : V_2 \rightarrow V$ satisfying $\{u, v\} \in E_2$ if and only if $\{f(u), f(v)\} \in E$.

Largest Common Subgraph. The *largest common subgraph problem* asks if there exist subsets $E_1' \subseteq E_1$ and $E_2' \subseteq E_2$ with $|E_1'| = |E_2'| \geq K$ for some positive integer K such that the two subgraphs $G' = (V_1, E_1')$ and $H' = (V_2, E_2')$ are isomorphic [13].

Digraph D-morphism. A related problem relevant to directed graphs is that of *digraph D-morphism*. For a given pair of directed graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ a *D-morphism* is formally defined in [13] as a function $f : V_1 \rightarrow V_2$ such that for all $(u, v) \in E_1$ either $(f(u), f(v)) \in E_2$ or $(f(v), f(u)) \in E_2$ and such that for all $u \in V_1$ and $v' \in V_2$ if $(f(u), v') \in E_2$ then there exists a $v \in f^{-1}(v')$ for which $(u, v) \in E_1$.

Adjacency Lists. One of the more common ways of representing graphs makes use of what are termed *adjacency lists* [9, 1]. An *adjacency list* is a list of the nodes that are adjacent to a given node. *Adjacency lists* are commonly implemented as an array of linked lists. Each element of the array represents a node of the graph and each linked list is the *adjacency list* for the given node. The *adjacency list* representation of a graph is often advantageous for representing sparse graphs as it only requires $O(|V| + |E|)$ storage space.

Adjacency Matrix. Another common representation of a graph is the *adjacency matrix* [19, 9, 1, 25]. The *adjacency matrix* of a graph with n nodes is an $n \times n$ matrix $A = [a_{ij}]$ in which $a_{ij} = 1$ if node v_i is adjacent to v_j and $a_{ij} = 0$ otherwise. A major drawback to using *adjacency matrices* is that they require $O(|V|^2)$ space. The *adjacency matrix* of a directed graph is defined similarly with $a_{ij} = 1$ if the directed edge $(v_i, v_j) \in E$. The *adjacency matrix* for undirected graphs is symmetric with a 0 diagonal.

Strongly Regular Graph. A *strongly regular graph* is defined in [42] as a graph having parameters (n, k, λ, μ) where n is the number of nodes, k is the degree of each node, λ is the number of common neighbors of each pair of neighbors in the graph, and μ is the number of common neighbors of each pair of non-neighbors of the graph.

2.1.2 Common Approaches

Due to its applicability to such a diverse set of problem domains, the problem of finding more efficient solutions to the graph isomorphism and subgraph isomorphism problems have occupied researchers for over 30 years. The simplest complete solution to the problem of graph isomorphism is a brute force search of the space of all possible orderings over the nodes of the graphs. For example, arbitrarily order the N nodes of one graph. Next, iterate over the $N!$ possible orderings of the nodes of the second graph. The orderings of the nodes

of the two graphs represent a one-to-one mapping. On each iteration check this mapping and determine if it represents an isomorphism between the graphs.

Over the history of the graph isomorphism problem, many approaches have been tried. One such approach is the search for a graph-theoretical property or some set of properties that form a sufficient condition to classify graphs through isomorphism [44, 18]. Although this research direction has yet to arrive at its goal, it has turned up some useful conditions that must necessarily exist for an isomorphism to exist. These conditions, known as graph invariants, are often incorporated into solutions to the graph isomorphism problem to reduce the search space immensely.

A second common approach to solving the graph isomorphism problem involves partitioning the nodes of the given graphs based on various graph invariants. A handful of such approaches to the problem are described in [46, 10, 45, 8, 15, 5, 38, 48]. Upon arriving at this partition, the simple brute force search is then applied to the reduced search space. This search can be a depth-first search or a breadth-first search and in some cases is a combination of the two. Some of these methods also incorporate heuristics to either guide the search or to trim away branches of the search space. An example of such a heuristic that will be discussed later is Ullmann's neighborhood consistency check [45].

There have been other approaches to the problem over the years. Some of these include reductions to other problems. For example, in [2] Almohamad and Duffuaa describe a linear programming approach. And in [49], Yang shows how a state machine may be generated from a graph and the resulting state machines then compared for isomorphism. In [39], an attempt at finding a canonical representation of the adjacency matrix of a graph is described. Other approaches include the massively parallel structure matcher described in [3], the decision tree technique described in [30], and the graph decomposition approach described in [29].

2.1.3 Invariants

A graph *invariant* is a number or a property of a graph that has the same value for any graph to which it is isomorphic. Another type of graph invariant is a property on the individual nodes of a graph that must have the same value for the node in the second graph to which it is mapped in an isomorphism. This type of invariant is sometimes referred to as a nodal function. A *complete set of invariants* determines a graph through isomorphism [19]. No complete set of invariants for a graph that can be computed in polynomial time is known to exist. Two simple examples of graph invariants are the number of nodes and the number of edges. A list of graph invariants and a list of nodal functions follow.

Graph Invariants.

- Number of nodes of a graph.
- Number of edges of a graph.
- The determinant of the adjacency matrix of a graph equals the determinant of any graph to which it is isomorphic [18]. The proof is trivial. Any graph isomorphic to a graph A can be represented as $P \cdot A \cdot P^{-1}$ for some permutation matrix P. Note that $det(P \cdot A \cdot P^{-1}) = det(P) \cdot det(A) \cdot det(P^{-1}) = det(A)$.
- The characteristic polynomial of the adjacency matrix of a graph is equal to that of any graph to which it is isomorphic [44]. The characteristic polynomial of an adjacency matrix A is defined as $det(A \Leftrightarrow \lambda \cdot I)$. A proof similar to that above may be found in [44].

Nodal Functions.

- Attribute consistency. An *attributed graph* is a graph for which there is a function mapping each of the nodes of the graph to a subset of a set of possible attributes or labels. A node of one graph is attribute consistent with the node to which it is mapped in the second graph if these nodes have the same set of attributes or labels associated with them.

- The degree of the nodes of the graph [45]. If the graphs are directed this would include both the in-degree and the out-degree of the nodes [46].

- The number of *nth generation descendents* of the nodes [46]. An nth generation descendent of a node i is a node reachable from i along a path of length n. And similarly, the number of *nth generation ancestors* of the nodes [46]. An nth generation ancestor of a node i is a node from which i can be reached along a path of length n. Given an algorithm for finding the nth generation descendents, the nth generation ancestors of a directed graph can be found by finding the nth generation descendents of the complement of the graph [46].

- The number of nodes in the *n-shell of descendents* of the nodes [46, 38]. A node is in the n-shell of descendents of a node i if it can be reached along a directed path of length n from node i but not by any other path shorter than length n. In other words, the length of the shortest path from a node i to any node in its n-shell is n. And similarly, the number of nodes in the *n-shell of ancestors* [46, 38]. A node j is in the n-shell of ancestors of a node i if node i can be reached along a directed path of length n from node j but not by any other shorter path. Given an algorithm for finding the n-shell of descendents, the n-shell of ancestors of a directed graph can be found by finding the n-shell of descendents of the complement of the graph [46].

- A function equal to 1 on nodes included in n-length *unrestricted circuits* and 0 otherwise [46]. Similarly, A function equal to 1 on nodes included in n-length *simple circuits* and 0 otherwise [46]. An unrestricted circuit may include an edge more than once whereas a simple circuit may not include any edge more than once.

2.1.4 Conventional Approaches

The conventional approach to the graph isomorphism algorithm is a modified brute force algorithm. These approaches make use of one or more invariants or nodal functions to partition the nodes of the given graphs into sets. Only nodes in corresponding sets may be mapped to each other. Upon partitioning the nodes of the graphs, either a depth-first search or breadth-first search can be used. Each search technique has its advantages. If there is no isomorphism between the graphs then a breadth-first search may determine that the graphs are not isomorphic more quickly. But breadth-first search requires more space as it keeps around all of the states of the search space. If there are many isomorphisms between the graphs, then a depth-first search may find one relatively quickly compared to a breadth-first search.

GIT: Graph Isomorphism Tester. In [46], Unger attempts to partition the nodes of the given graphs to as fine a partition as possible. The algorithm he describes begins by generating a PNPL (possible node pairing list) composed of one partition containing all of the nodes and then iteratively refines the PNPL into groups of smaller and smaller partitions representing the nodes that may be paired to each other. On each iteration the algorithm checks the current ordering to see if it represents an isomorphism. The nodes are first partitioned by in-degree and further partitioned by out-degree. The partitions are then refined by computing the nth generation descendents, nth generation ancestors, the n-shell of descendents, the n-shell of ancestors, a function equal to 1 on nodes included in n-length

unrestricted circuits and 0 otherwise, and a function equal to 1 on nodes included in n-length simple circuits and 0 otherwise.

Unger next describes an EXTEND method of generating additional nodal functions. Assign each partition a unique number. Then for each node assign it the value that is the sum of the set values to which each of its descendents belong. And use these values to further refine the partition. Any symmetric function of the set numbers can be used and is not limited to sum. The EXTEND method can be used also on the ancestors, n-th generation descendents, and n-th generation ancestors.

The GIT algorithm works better with graphs with a smaller numbers of edges. Therefore, if the graphs have more than $\frac{n(n+1)}{2}$ arcs, GIT first takes their complements. It can do this because two graphs are isomorphic if and only if their complements are isomorphic.

Using K-formulas. The algorithm described by Berztiss in [5] uses K-formulas. The K-operator $*$ is a binary prefix operator. A K-formula represents an arc in the digraph and consists of the K-operator followed by the node names of the originating node and the terminal node. A K-formula can represent all of the n arcs originating from a given node by beginning the K-formula with n K-operators followed by the originating node name and the n terminal node names. K-formulas can also be used in place of a single node name. A K-formula can be defined recursively as 1) a node symbol, or 2) if a and b are K-formulas then $*ab$ is a K-formula. Berztiss describes a procedure for generating a set of K-formulas that represent a given digraph. The isomorphism algorithm works by generating a minimal set of K-formulas for one graph and fixing it. It then attempts to generate a K-formula for the second graph corresponding to this K-formula (having the same pattern) using a backtracking procedure.

Ullmann's Algorithm. Ullmann's algorithm for subgraph isomorphism is perhaps still one of the most widely used algorithms for graph and subgraph isomorphism. Even today, researchers often compare the performance of their algorithms to that of Ullmann's. The complete description of Ullmann's algorithm can be found in [45].

Ullmann first describes a simple enumeration algorithm for subgraph isomorphism using a depth first tree search. He then presents a refinement procedure to reduce the search space and incorporates it into the algorithm. The simple enumeration algorithm works as follows. Let the adjacency matrices of graphs G_α and G_β be $A = [a_{ij}]$ and $B = [b_{ij}]$. G_α has p_α nodes and q_α edges (and similarly G_β). $M' = [m'_{ij}]$ is a matrix with p_α rows and p_β columns. Each row has exactly one 1. No column has more than one 1. Let $C = [c_{ij}] = M'(M'B)^T$ where ^{T} is transpose. If $\forall i, j(a_{ij} = 1) \Rightarrow (c_{ij} = 1)$ then M' specifies an isomorphism between G_α and a subgraph of G_β (labeled condition 1 in [45]). If $m'_{ij} = 1$ then the jth point of G_β is mapped to the ith point of G_α. At the start of the algorithm $M^0 = [m^0_{ij}]$ is constructed. $m^0_{ij} = 1$ if the degree of the jth point of G_β is $\geq$ the ith point of G_α and is 0 otherwise. For isomorphism testing change the $\geq$ condition to $=$. The simple enumeration algorithm generates all possible matrices M' such that for all m'_{ij} of M', $(m'_{ij} = 1) \Rightarrow (m^0_{ij} = 1)$. For each such matrix, condition 1 is applied to determine if it is an isomorphism. The M' are generated by systematically changing all but one 1 in each row of M^0 to a 0 subject to the constraint.

Ullmann's refinement procedure is then described. It is referred to in [8] as Ullmann's neighborhood consistency check. The idea is to eliminate some of the 1's from the matrices M thus eliminating some successor nodes from the tree search. It tests each 1 in M to find whether, $\forall x((a_{ix}) \Rightarrow \exists y(m_{xy} * b_{yj} = 1))$. That is for every neighbor x of node i, there must exist a node y of G_β such that y is a neighbor of vertex j and x is allowed to be mapped to y. It changes the 1 to a 0 if this condition is not satisfied. It iterates until there is an iteration in which none of the 1's are changed to 0. If M satisfies the condition for being an

M' matrix (that is, each row of M contains exactly one 1 and each column of M contains no more than one 1), then if the refinement procedure does not alter M, M specifies an isomorphism between the graphs. This refinement procedure is then incorporated into the simple depth-first enumeration algorithm.

Using Distance Matrices. In [38], Schmidt and Druffel propose using distance matrices as an improvement over using the degree sequence of the nodes of the graphs to reduce the search space of the traditional backtracking approach. The distance matrix D is an n by n matrix in which element d_{ij} represents the length of the shortest path between nodes v_i and v_j. If $i = j$, then $d_{ij} = 0$. If there is no path between i and j then $d_{ij} = \infty$. By using the distance matrix of a graph, it is possible to obtain an initial partition of the nodes of the graph that is finer than simply using the degree of the nodes as a partition.

The authors in [38] describe a characteristic matrix. The row characteristic matrix XR is an N by $(N \Leftrightarrow 1)$ matrix. xr_{im} is the number of vertices a distance m away from v_i. The column characteristic matrix XC is an N by $(N \Leftrightarrow 1)$ matrix. xc_{im} is the number of vertices from which v_i is a distance m. A characteristic matrix X is formed by composing the corresponding rows of XR and XC. An initial partition may be obtained from X. v_i^1 will map to v_r^2 in an isomorphism if and only if $x_{im}^1 = x_{rm}^2$ for all m. An initial partition based on the distance matrix may be more refined than that based on the adjacency matrix, and can never be less refined.

The algorithm is a backtracking algorithm that selects possible vertex mappings. It checks each mapping for consistency using the distance matrix. The mapping v_i^1 to v_r^2 is consistent if every element $d_{ij}^1 = d_{rs}^2$ and $d_{ji}^1 = d_{sr}^2$ for all j, s such that v_j^1 has been mapped to v_s^2 and if every element d_{ik}^1 (where v_k^1 has not been previously mapped) has a corresponding d_{rp}^2 (where v_p^2 has not been previously mapped) such that $c_k^1 = c_p^2$ (that is they are in the same partition). If the partition does not consist of consistent mappings

then the mapping is not an isomorphism and it's necessary to backtrack and try another mapping.

There are some classes of graphs for which the distance matrix does not refine the initial partition any more than the degree sequences. For example, if there was a single node attached to all other nodes of the graph the shortest path between any two nodes of the graph will be 2. But the authors offer a possible solution for some cases. If the two graphs have an equal number of such nodes they may be removed from the graphs. In the same way if the graphs have an equal number of 0 degree nodes they may also be removed.

2.1.5 Other Approaches

Canonical Adjacency Matrix. In [39], an attempt at finding a canonical representation of the adjacency matrix of a graph is described. The algorithm described is for undirected linear graphs. The idea is to generate what it terms an"optimum code" from an adjacency matrix as a sort of canonical representation of the set of graphs isomorphic to the graph represented by the adjacency matrix. The graphs are undirected so the upper triangle of the adjacency matrix represents the entire graph. The algorithm attempts to"relabel" a graph uniquely so that upon this relabeling the binary number obtained by concatenating the rows of the upper triangle of the adjacency matrix of the relabeled graph is of greatest possible magnitude. The worst case complexity of this approach was exponential.

Reduction to Isomorphism of Finite State Machines. In [49], Yang shows how a state machine may be generated from a graph and the resulting state machines then compared for isomorphism. Algorithms for determining the transition preserving morphisms (endomorphism, homomorphism, isomorphism, and automorphism) of state machines using nontrivial closed partitions over their state sets are described. These algorithms are then extended to determine the structural preserving morphisms of finite automata by adding a

constraint of output-consistency to the partitions of their state sets. It is then shown that a Moore-type sequential machine may be constructed from a directed graph by performing 3 steps: constructing a non-deterministic state machine corresponding to the graph, transforming this to an equivalent deterministic state machine, and defining the outputs of the states. The isomorphism algorithm for finite state machines is then used on the resulting Moore-type sequential machines.

Linear Programming. In [2] Almohamad and Duffuaa describe a linear programming approach to the weighted graph matching problem. The problem of matching two weighted graphs can be formulated as finding an optimum permutation matrix that minimizes a distance measure between the two graphs. The weighted graph matching problem includes the graph isomorphism problem as a special case. This paper formulates the problem as a linear programming problem and uses a simplex-based algorithm to solve it. The idea behind the weighted graph matching problem is to find the permutation matrix P as to minimize $\parallel A_g \Leftrightarrow P * A_h * P^T \parallel$. $\parallel A \parallel$ is the sum of all of the elements of the matrix A. This is also equivalent to minimizing $\parallel A_g * P \Leftrightarrow P * A_h \parallel$. They formulate a linear programming problem, solve the linear programming problem using the simplex method and then obtain approximate 0-1 integer solutions from the real solution of the linear program. The simplex method is exponential but in practice will find the solution quickly. Approximate 0-1 integer solutions can be found in polynomial time. There is no known algorithm for finding the exact 0-1 integer solutions in polynomial time. All known methods for exact 0-1 integer solutions such as branch and bound have exponential time complexity.

Graph Decomposition Approach. In [29], an approach to the graph isomorphism problem based upon decomposing the graphs into common subgraphs is proposed. The idea of the algorithm is to search for a graph among a collection of model graphs for one that is

isomorphic to some given query graph. The algorithm described builds a network from the set of model graphs. It decomposes the model graphs into subgraphs, then subgraphs of the subgraphs, etc. Thus common subgraphs of the larger graphs can be represented once in the network. The network algorithm (NA) will then take the input graph and propagate it through the network to determine a subgraph isomorphism. The authors also describe an inexact network based algorithm (INA). It is based on their exact algorithm. The authors compare NA to Ullmann's. Ullmann's algorithm improves with more diversity among the labels. But too many different labels is actually harmful to NA when it breaks the model graphs down into subgraphs. The network will not be as compact as it would be with less diversity among the labels.

Decision Tree Approach. In [30], the authors attempt to solve the problem of given a database of model graphs known a priori and an input graph known only at run-time, find any of the model graphs for which the input graph is either isomorphic to or isomorphic to a subgraph of. Their algorithm runs in $O(M^2)$ time if you neglect preprocessing of the model graphs and does not depend on the number of model graphs. M is the maximum number of nodes in any given model graph. They arrive at this polynomial time by building a decision tree from all permutations of the adjacency matrices of the model graphs. This decision tree in the worst case is exponential in size.

The authors propose techniques for pruning the decision tree. The first is a breadth-pruned decision tree. It will no longer support subgraph-isomorphism but the runtime is still polynomial although now $O(M^3)$. One type of breadth-pruning involves transforming the input graph by ordering the vertices so that each vertex is connected to at least one other vertex that appears earlier in the ordering. For connected graphs this is equivalent to finding the spanning tree of the graph (quadratic time). Now any permutations of the adjacency matrices of the model graphs for which this condition does not hold may be

removed from the decision tree. The algorithm is still quadratic and still works for both graph and subgraph isomorphism but the decision tree is greatly reduced in size. If the graph is completely connected this pruning will not save any space. A second breadth-pruning technique increases the runtime to $O(M^3)$ but no longer guarantees that subgraph isomorphism may be detected as some of the subgraphs of a given model graph may no longer be present in the decision tree. But graph isomorphism can still be detected in polynomial time.

The authors next present depth-pruning the decision tree to use the decision tree as an index into the collection of model graphs for further testing by a conventional algorithm such as Ullmann's. The idea is instead of representing all subgraphs and permutations of the model graphs in the decision tree, only represent all subgraphs and permutations of size $k < n$ in the decision tree. When an input graph is now classified against the decision tree, all graphs associated with the result decision tree node must now be further tested with a conventional algorithm such as Ullmann's but Ullmann's may be initialized based on the decision tree greatly reducing the search space. Polynomial time is no longer guaranteed but this technique will reduce the size of the decision tree and make it of practical use for larger sized graphs.

A technique based on the decision tree approach to the graph and subgraph isomorphism problem to find what is termed error-correcting graph isomorphism is described in [31]. The authors begin by defining error-correcting graph isomorphism. The idea is to develop a measure of distance between graphs by the cost of making a sequence of edit operations to transform one graph to the other. The possible edit operations are changing a nodes label, changing an edge's label, adding an edge, and removing an edge. The definition can be extended to include adding and removing nodes. Costs are assigned to each type of operation and the distance between two graphs is taken to be the minimum cost to transform one graph into the other. There may be more than one error-correcting

isomorphism but the problem is to find the one of minimum cost.

To compute the error-correcting isomorphisms the authors make use of the decision tree approach described in [30]. They compute all of the error-correcting isomorphisms of the model graphs and classify them by the decision tree. Then at run-time they use the decision tree algorithm to find an exact match in the tree. Alternatively, they generate all of the error-correcting isomorphisms of the input graph some distance away and use the decision tree attempting to match each in order of distance.

Parallel Methods. The PARKA structure matching algorithm is described in [3, 37]. The authors describe an algorithm for efficient associative matching of relational structures in large semantic networks. The goal is to allow for efficient and flexible access to large knowledge bases for case-based reasoning systems. The algorithm uses PARKA, a massively parallel knowledge representation system which runs on the Connection Machine. The algorithm uses parallel search for knowledge structures. Both the retrieval probe and the stored cases are represented as graph structures in a semantic network. The algorithm relies on massively parallel hardware (the CM-2) to match knowledge structures in memory against the retrieval probe.

A knowledge base (KB) defines a set of unary and binary relations. Given a conjunctive expression of a subset of these relations, the task is to retrieve all structures from memory that match this expression. This problem of matching knowledge structures can be viewed in two ways: a subgraph isomorphism problem or a problem of unification or constraint satisfaction. The authors take the subgraph matching view. Seen this way, case memory is represented as a graph structure, where cases consist of a set of concepts (nodes) connected by relations on the concepts (edges). The problem of finding similar cases is reduced to a problem of structural matching, or of identifying subgraphs in the semantic network that are isomorphic to the query graph. The structure matching algorithm operates by comparing

the query case against a KB to find all structures in the KB which are consistent with the query. This match process occurs in parallel across the entire KB.

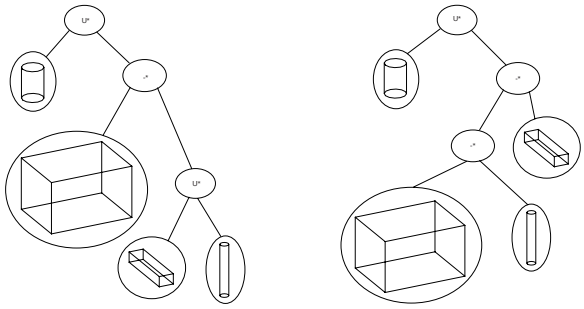
2.2 Solid Modeling and Feature Based Design

2.2.1 Constructive Solid Geometry (CSG)

Constructive Solid Geometry (CSG) is a volumetric representation scheme for three-dimensional solid geometric models. Solids are represented as a set-theoretic boolean expression of primitive solid objects, of a simpler structure [20]. Regularized set boolean operations and motion operations are used to represent a composition of primitive geometric shapes. The standard primitives that are used in the CSG representation scheme are the parallelepiped or block, the triangular prism, the sphere, the cone, the cylinder, and the torus [20]. The set boolean operations that may be used are regularized union, regularized difference, and regularized intersection. The regularized boolean set operations are extensions of the typical boolean operations that prevent dangling edges and faces from resulting. A CSG representation of a solid model can be viewed as a tree. The primitive shapes used in representing the solid are the leaves of the tree; the boolean set operations and motion operations are the interior nodes, as shown in Figure 2.1 (a). A CSG representation of a three-dimensional solid model lacks uniqueness. There may be several ways to represent a single solid model with multiple CSG representations. One solid may be representable by several valid CSG representations, as shown in Figure 2.1 (b).

2.2.2 Boundary Representation (B-rep)

A solid can unambiguously be represented by describing its surface and its topological orientation. The boundary representation (B-Rep) consists of a topological description of the



(a) (b)

Figure 2.1: Examples of CSG trees: two different trees that create the same solid model.

connectivity and orientation of the faces, edges, and vertices and a geometric description for embedding these surface elements in space. The vertices, edges, and faces are specified abstractly with their incidences and adjacencies indicated in the topological description. And in the geometric description, the equations of the surfaces of which the faces are a subset are specified [20].

The boundary representation (B-Rep) scheme represents three-dimensional solid objects by a hierarchical description of the faces, edges, and vertices that form the boundary of the model. A face is specified by the edges that it is bounded by. An edge is specified by the curve it lies on and its vertices. Vertices are points in three-dimensional coordinate space. The B-Rep representation scheme is unique unlike that of the CSG representation [23].

2.2.3 Feature-based Modeling

A feature can be defined differently depending on the context in which it is to be used. Machining features may differ from forging features [24]. Features of a solid geometric model are dependent on the use of the model. In the application of machining, some example

features are holes, slots, pockets. Each such feature of the solid model may correspond to some manufacturing procedure or step of the design process.

Modeling and design are typically performed by the addition and subtraction of primitive shape components from the solid model. Using this approach, features would be extracted later using a feature recognition process. However, this is not the only approach that may be used. Modeling by using design features is an alternate approach. This is known as feature-based modeling [36, 14].

In [36], it is pointed out that feature-based design has the advantage of keeping relevant information for applications during the design process. It is also pointed out that manufacturing concerns can be considered early in the design process. Using feature recognition, this may not have been possible. A model may have been designed with "features" that would be difficult to actually manufacture. In feature-based design, functional meaning is assigned to the parts of an object during the design phase rather than during the feature recognition [6].

[11] discusses a combined approach of feature-based design and design recognition. The feature-based design part of the described approach incorporates a feature library, consisting of predefined design features and user defined design features. The predefined features consist of features such as cylindrical holes, rectangular pockets, and slots. User-defined features can be created by the designer to make up for a deficiency in the features in the library. These user-defined features can be created with either the feature modeler or a solid modeler.

2.2.4 Feature Recognition From Solid Models

Much research has been done in the area of automatic feature recognition from three-dimensional solid models [17, 34, 27, 28, 47]. Although there are other representation

schemes, the most commonly used representations in systems that perform automatic feature recognition are the B-Rep and the CSG. This is in part due to the fact that a majority of solid modeling and CAD systems make use of either the B-Rep or the CSG in their representations. Some systems may incorporate both into the representation of the solid models. Boundary representations are often used for rendering and display purposes, while CSG-like structures supply the history of the operations performed in designing an artifact.

One technique used for feature recognition makes use of the attributed adjacency graph (AAG) that is generated from the B-Rep of the solid model and is described in detail in [24]. In an AAG, each node represents a face of the solid model. Each edge in the solid model becomes an arc in the AAG where the endpoints are the nodes that represent the faces that share the edge. Each arc in the AAG is attributed with an attribute that specifies if the faces corresponding to the edge are concave or convex. The technique for feature recognition described in [24] uses graph-based techniques to search the AAG of the solid model in question for subgraphs which correspond to the AAG representations of primitive elements, and incorporates some special techniques for detecting interacting features. The use of AAGs for feature recognition is limited to polyhedral parts with polyhedral features [24].

Another technique for feature recognition similar to the use of AAG is presented in [27]. The approach presented incorporates what is termed a cavity graph. A cavity graph consists of nodes representing the faces of the solid. Links between two nodes represent nonconvexity of the corresponding faces. And each node is labeled to show the relative orientation of the faces in space. The method proposed uses a hypothesis generation and elimination approach. The hypotheses are generated by decomposing the cavity graph of the object into maximal subgraphs and searching searching these subgraphs for the known cavity graphs of primitive components. Rule-based methods are used to eliminate incorrect hypotheses and generate new hypotheses. The methods described also incorporate the idea of using "virtual links" to aid in finding interacting features (i.e., additional links are added to the

cavity graph) [27].

Convex-hull techniques use volumetric properties of solid models rather than surface features to extract features. The convex-hull technique relies on finding the materials that must be removed from a solid to form the model of the part. The feature extraction process uses convex decompositions. An object is represented as a set of convex components with alternating addition and subtraction of volumes. The convex decompositions are sometimes known as alternating sum of volumes (ASV) [23]. This decomposition represents an object by a series of convex volumes with alternating additions and subtractions. The technique first finds the convex hull of the object and then finds the set difference between the object and its convex hull. The technique is applied recursively to find the full decomposition of the object. The domain of geometric objects that ASV can handle is limited as ASV will not always terminate. The removed volumes also do not always represent features. Volumes that may be shared by two or more interacting features will only be applied to one, for example [23].

In addition to techniques for feature recognition that use B-rep as input, there exist techniques that use the CSG representation of the solid model. These techniques must first overcome the problem that the CSG representation is not unique. A single solid model may be represented by several different CSG trees. Another problem is that nearby nodes in the CSG tree do not necessarily correspond to features. The set difference operation also does not necessarily correspond to the removal of manufacturing material; and some removal operations may be implicit without the use of a set difference operation. Most methods of performing feature recognition from the CSG representation begin by converting it to some other representation. Although the potential exists for the CSG representation to more closely resemble machining operations, in practice there appears to be a lack of a general relationship between the primitives of a CSG and the features of the design [23].

2.3 Search Techniques

Many problems can be solved by the application of a search strategy. These search strategies begin at some initial problem state and search through the state space for the given problem looking for a goal state. This search of the state space can be visualized as a tree search. Nodes of this search tree represent various states of the problem. The root node of this tree represents the initial state. The children of a search tree node, or the successors of the node, represent the states that result from performing successor operations to transform the problem from one state to the next. Making such a transformation is termed expanding a search node. The number of new search tree nodes that result from applying the successor functions to a search node is referred to as the *branching factor*. Not all search strategies involve searching the state space in a type of tree search. There are other algorithms known as iterative improvement algorithms. These algorithms simply maintain the current state and iteratively make changes to the state that improve some evaluation of the state. These algorithms tend to halt in polynomial time whereas tree search strategies are often exponential in the worst case. But they are not complete and suffer from an inability to guarantee an exact solution. There are several techniques for searching a search space. Each technique has its advantages and disadvantages. Here I describe a few search strategies.

2.3.1 Depth-first Search

Depth-first search proceeds through the search tree by always expanding one of the search nodes at the deepest level of the tree. It continues in this way until it either reaches a goal node or until it reaches a goal node with no expansion (i.e. a node for which the successor functions produce no new search nodes). When the search reaches a dead end such as this it backs up to a shallower level and continues the *depth-first search* down another branch of the search tree. The space requirements of *depth first search* is only $O(b \times d)$ where b is

the branching factor an d is the maximum depth of the search tree. The time complexity in the worst case is $O(b^d)$. But for problems with many solutions *depth-first search* may take much less time than this. For problems that have very deep or infinite search trees *depth-first search* may get stuck going down a wrong path, and thus for such problems another search strategy should be used [35].

2.3.2 A* Search

A search* is a form of informed search [35, 43]. Informed search methods make use of problem-specific information to guide the search and to help obtain more efficient solutions. The idea behind *A* search* is to minimize the total path cost $f(n) = g(n) + h(n)$ where $g(n)$ is the cost of the path so far and $h(n)$ is the estimated cost to the goal. The function $f(n)$ can be looked at as the estimated cost of the cheapest solution through n. If $h(n)$ never overestimates the cost to the goal, then it is said to be *admissible*. If $h(n)$ is *admissible* then *A* search* is both optimal and complete [35]. *A* search* chooses the successor node as to minimize $f(n)$.

2.3.3 Constraint Satisfaction

The states of a *constraint satisfaction problem* are represented by a set of values for a set of variables. A goal state for such a problem must satisfy a set of constraints on the variables. These problems may be solved using search techniques such as those described. There are some other techniques that may be applied in such situations. *Arc consistency checking* verifies that every variable has a value in its domain that is consistent with the constraint set [35]. Values that are inconsistent with any constraint are removed from the domain of the given variable. *Arc consistency checking* may in some cases result in a solution to the problem if the domain of each variable is reduced to a single value.

One form of *arc consistency checking* is known as *forward checking* [35]. When *forward checking* is used, whenever a variable is instantiated, any value in the domain of a variable that has not yet been instantiated that is inconsistent with the variables that have been assigned values is removed from the domain of that variable. If the domain of a variable becomes empty the search backtracks immediately.

2.3.4 Hill-climbing Search

Hill-climbing or *gradient descent* is an example of an iterative improvement algorithm [35, 43]. These algorithms work by making iterative changes that always improve the current state. The value of the current state is determined by an evaluation function. The *hill-climbing algorithm* does not maintain a search tree and only stores the current state of the problem. There are two variations. *Best ascent* will always choose the best successor of the current state. *Next ascent* chooses the first successor it finds that is an improvement on the current state.

There are problems with *hill-climbing* algorithms. The first is the problem of *local maxima*. The algorithm may reach a peak in the state space that is lower than the highest peak and halt with a far from accurate answer. Another problem is that of *plateaux*. A plateau is a flat area of the search space. The search will wander around randomly on such a *plateau*. *Random restart hill-climbing* attempts to combat these problems by starting at a randomly chosen new initial state when the search reaches a point where it can make no further progress. It can use a fixed number of restarts or can continue until the best result found so far has not been improved for a specified number of iterations [35].

Chapter 3

Problem and Approach

3.1 Problem Formalization

A design, D, is defined as a tuple $D = \langle F, P, T \rangle$ where: F is a finite set $F = \{f_0, \ldots, f_n\}$ of features (these may be design features or manufacturing features); P is the geometric and topological model of the artifact (including the boundary representation of the component); T is a possibly empty set of dependencies or orderings imposed on the features (for example, T can be the design history of the artifact). If the features in F are design features then T can be thought of as a type of CSG tree and design features are local or global operations on part geometry. In general, the features in F are volumetric representations and are themselves CAD models, although more primitive then the CAD model as a whole.

The boundary representation (B-rep) of a component (or part), P, is a representation of the geometric and topological model of the artifact. T is another representation of the geometric model—related in some way to the steps that were involved in the design. Most CAD and three-dimensional geometric/solid modeling packages use either B-rep or CSG representations, and in some cases both. There also exist techniques for converting (1) CSG to B-Rep; (2) certain classes of solids from B-Rep and CSG; and (3) feature identification from solid models. Hence, history information is either readily available or can be produced, to a degree, via automated feature identification techniques [41, 16, 40].

The set F of features is a finite set defined as $F = \{f_0, \ldots, f_n\}$. These features can include any volumetric or surface operation typically used in a commercial CAD environment (i.e., holes, pockets, slots, bosses, etc.) and alternatively they can include any manufacturing features typically extracted using a feature recognizer (i.e., holes, pockets, slots, etc.). A feature ordering T, can be either a tree structured representation of the design phase of an engineering artifact or it can simply be a linear ordering of the steps taken to design the artifact. The nodes of this tree represent the primitive elements added to or removed from the component during the design phase, such as blocks, cones, and cylinders, operations on those primitives, and operations on the component as a whole entity. Some of the possible operations include blends, chamfers, fillets, extrusions, contouring, and free-form surface modeling. T can also represent alternatively some ordering on the manufacturing features, perhaps associated with some sequence of manufacturing operations. T can also be an empty set.

3.2 Design Histories

The same artifact may be designed in several different ways. One designer may do things in one order, and another designer in a different order. These different orderings of operations will result in history trees that can be drastically different and yet represent the same thing. For example, Figure 3.1 shows a solid model for a torpedo motor housing consisting of about one hundred design feature instances. Figure 3.1 also shows one possible design feature history tree for this model that can be used to define this part in a commercial CAD environment (there may be many other ways of designing this part). The non-uniqueness of the history tree poses a problem as to how to effectively index and retrieve CAD and solid model data based on feature information.

This problem is similar to that of the non-uniqueness of CSG models. In the feature

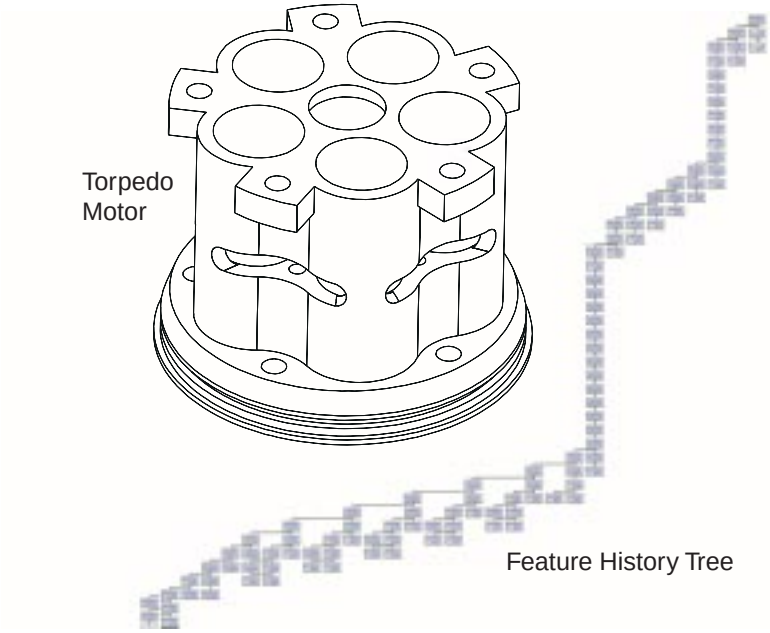


Figure 3.1: An illustration of a model of a torpedo motor housing and a snapshot of the design feature history tree for the torpedo motor (each box is a feature or operation on the model). This history tree was generated when the motor was modeled using Bentley Systems' MicroStation Modeler. The over one hundred features and operations make the history tree difficult to present in detail—for requiring more detail, this model is available through the National Design Repository at http://repos.mcs.drexel.edu/.

recognition techniques that use CSG models rather than the B-reps, methods of converting the CSG to another representation or an ordered representation are usually incorporated to get around this disadvantage [40]. I will incorporate similar techniques in the use of history trees for similarity comparisons.

Some operations performed on the design are dependent on other previous operations and must be performed in a specific order. For example, it is not possible to create a hole in a block that does not yet exist. But there may be other operations that may be independent of all other operations that have been performed on a design. For example, if you had a block and wanted to remove a hole in one side and create a slot in the other side with no interaction occurring between this slot and hole, then it would make no difference to the

final product which of these two operations occurred first.

The history tree is initially sorted in the relative order of when each operation was performed in time relation to each other. This ordering has no bearing on the order of operations performed during the manufacturing of the component. For example, a designer may take a block and remove a hole in one side and then a hole in the other side. The designer may then remove a second hole in the first side. When the manufacturing plan is later devised by a machinist, he or she may decide that the two holes on the first side can be drilled at the same time, rather than following the exact steps taken in the design phase.

Hence, to retrieve engineering data from knowledge-bases using the design history as part of the retrieval probe, it becomes necessary to transform the design history tree in such a way that it is now ordered solely on the basis of dependencies rather than on an order based on temporal position.

3.3 Model Dependency Graph

Let's begin by developing a representation to handle design features. Upon doing so, I will discuss how this same representation may be used for manufacturing features. As discussed earlier, design history trees, like CSG trees, are non-unique. That is, for a given solid model, there may be several ways to design it and result in the same final product, and thus there may be different design history trees that represent the same design.

The **Model Dependency Graph** (MDG) attempts to deal with this problem. This graph is a directed acyclic graph which has some unique characteristics. The model history, M, is defined as $M = \{m_0, \ldots, m_n\}$. The m_i is the complete model at stage i of the design. That is, m_i represents the solid model after feature f_i is applied to the model. There is an ordering inherent in the design history graph. In the case where it is not clear which operation or feature came before the other, simply impose an arbitrary order on

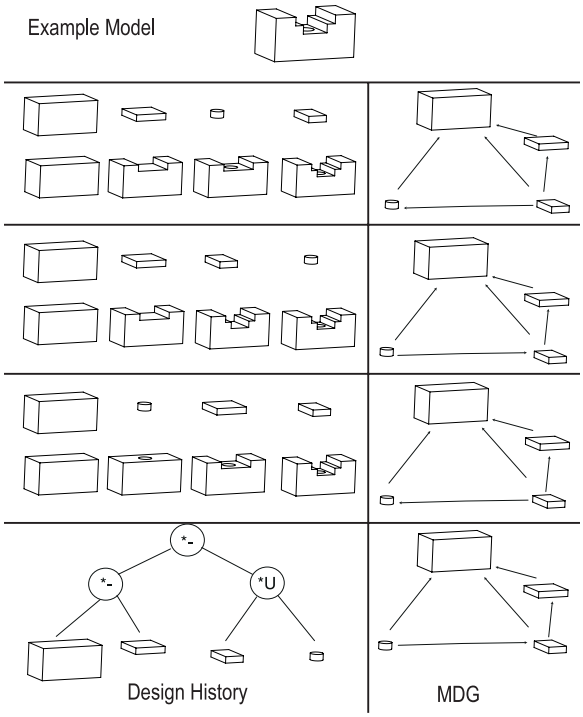


Figure 3.2: Pictured is a single solid model and several alternative design feature histories, and one possible CSG tree, that can produce it. On the right are the MDGs for each of these alternatives—note that they are all D-morphic to one another.

these operations. The m_i may be generated and stored at design time. Or they may be easily generated from the design history. Let $vol(f_i)$ represent the "solid" volume that is either added or removed from the complete model by the application of feature f_i.

Definition 1: *Model Dependency Graph* - basic definition

A *Model Dependency Graph* (MDG) is defined as $G = (V, E)$. The vertex set is defined as $V = \{f_0, \ldots, f_n\}$. The indices on the f_i represent the order that the features were applied during the design process. The edge set can be defined as $E = \{(f_i, f_j)$ such that $i > j, vol(f_i) \cap vol(f_j) \neq \emptyset\}$. Note that $\cap$ is not a regularized intersection.

One limitation with the MDG as it has been defined in Definition 1 is that it assumes an

explicit ordering on the features or design operations. In many cases this may be captured in the solid modeling application in the form of a design history. But can the MDG be used when dealing with CSG trees? The answer is yes and can be obtained by extending the definition of the MDG to work recursively down the CSG tree.

Definition 2: *Model Dependency Graph* - non-linear definition

Let $T = (op\ left\ right)$ be a CSG tree or some non-linear design history where op is an operation and $left$ and $right$ are CSG subtrees or primitives shapes. Let $G_1 = (V_1, E_1)$ be the MDG of $left$ that results from either the basic definition or the non-linear definition. Let $G_2 = (V_2, E_2)$ be the MDG of $right$ that results from either the basic definition or the non-linear definition. Then the MDG of T can be defined as $G = (V, E)$ such that $V = V_1 \cup V_2$ and $E = E_1 \cup E_2 \cup E_3$ where $E_3 = \{(v_2, v_1), v_1 \in V_1, v_2 \in V_2$ such that $vol(v_1) \cap vol(v_2) \neq \emptyset\}$. Note that $\cap$ is not a regularized intersection.

An example of a solid model with different possible design feature histories is shown in Figure 3.2. There is a property of the MDG that I will exploit in our similarity assessment of solid models: *digraph D-morphism.* For a given pair of graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ a *D-morphism* is formally defined in [13] as a function $f : V_1 \rightarrow V_2$ such that for all $(u, v) \in E_1$ either $(f(u), f(v)) \in E_2$ or $(f(v), f(u)) \in E_2$ and such that for all $u \in V_1$ and $v' \in V_2$ if $(f(u), v') \in E_2$ then there exists a $v \in f^{-1}(v')$ for which $(u, v) \in E_1$.

Theorem 1: *D-morphisms of Model Dependency Graphs.* Let G_1 and G_2 be two MDGs for the same solid model resulting from different orderings of a feature set $F = \{f_0, \ldots, f_n\}$ (such as shown in Figure 3.2). G_1 and G_2 are D-morphic.

Proof: Pick any two orderings of the set $F = \{f_0, \ldots, f_n\}$ arbitrarily. Let these orderings be $L = \{l_0, \ldots, l_n\}$ and $H = \{h_0, \ldots, h_n\}$ where $\forall f_i \in F,\ \exists l_j \in L, h_k \in H$ such that

$f_i = l_j = h_k$ and $\exists i, 0 <= i <= n$ such that $l_i \neq h_i$. Let $G_1 = (V_1, E_1)$ be the MDG that results from L and let $G_2 = (V_2, E_2)$ be the MDG that results from H. It is clear that $V_1 = V_2$. By the definition of the MDG, these vertex sets must be equal to the set F. Now take any two vertices $v_k, v_l \in V_1$. Pick out the vertices $v_m, v_p \in V_2$ such that $v_k = v_m = f_i$ and $v_l = v_p = f_j$. Note that $vol(v_k) = vol(v_m)$ and $vol(v_l) = vol(v_p)$. Therefore, $vol(v_k) \cap vol(v_l) = vol(v_m) \cap vol(v_p)$. Hence, from the definition of the MDG, if there is an edge $(v_k, v_l) \in E_1$ where $k > l$ then either $(v_m, v_p) \in E_2$ where $m > p$ or $(v_p, v_m) \in E_2$ where $p > m$. Therefore, G_1 and G_2 are D-morphic.

Some questions may arise given the definition of MDG and the proof of D-morphism. One such question is how to generate the MDG of a given model. A possibility is to generate the MDG at design time. Upon the addition of a feature to the design, a node must be added to the MDG. Along with this new node, edges must be added from the newly added node to any previously added node corresponding to any features for which there is a non-empty intersection with the newly added feature or some interaction with the newly added feature.

Another question that will arise is how to handle the possibility of the same model being designed two different ways or with different feature sets. The same model designed with different feature sets will have MDGs that are not necessarily D-morphic. And related to this question is the question of what to do if a design history is not available for a given model. The answer to these questions is to use the manufacturing features rather than the design features. Use a feature extraction system such as F-Rex [34] or Allied Signal's FBMach [7] to extract the features. In this way you can use one common set of features across the entire collection of models. Performing this feature extraction will result in a unique set of features for the given model. This set of features will become the node set of the MDG. You can then "order" this set arbitrarily and generate the edge set of the MDG

by making use of feature interactions detected during the feature extraction phase.

The MDG as defined is a directed graph. What exactly is the significance of the direction of the edges? Well, the direction is an indication of the order the features were performed. In the case of using design features, the direction represents which design features were added to the design earlier in the design phase. If manufacturing features have been used in the construction of the MDG, then the direction possibly will represent which operations were performed before which others (for example, if the features are related to some manufacturing plan). But what meaning does the direction on the edges have to features obtained through feature extraction? In the grand scheme of things, there is no clear meaning. In fact, as stated above the extracted features are "ordered" arbitrarily. Therefore these directions may result in unintended ambiguity. So instead, define an *Undirected Model Dependency Graph* (UMDG). The UMDG is defined similarly to the MDG. The node set consists of the set of features, either design or manufacturing features, and the edge set consists of edges between any two features between which there is an interaction.

Definition 3: *Undirected Model Dependency Graph*

An *Undirected Model Dependency Graph* (UMDG) is defined as an undirected graph $G = (V, E)$. The vertex set is defined as $V = \{f_0, \ldots, f_n\}$. The edge set can be defined as $E = \{\{f_i, f_j\}$ such that $vol(f_i) \cap vol(f_j) \neq \emptyset\}$. Note that $\cap$ is not a regularized intersection.

Theorem 2: *Isomorphisms of Undirected Model Dependency Graphs.* Let G_1 and G_2 be two UMDGs for the same solid model resulting from the use of the features resulting from two executions of a consistent and unambiguous feature extraction system. G_1 and G_2 are Isomorphic.

Proof: Let $F = \{f_0, \ldots, f_n\}$ be the feature set resulting from the first execution of the feature extraction system. Let $L = \{l_0, \ldots, l_n\}$ be the feature set resulting from the second execution. The feature extraction system in question is assumed to be consistent and therefore, $F = L$. Therefore, $\forall f_i \in F,\ \exists l_j \in L$ such that $f_i = l_j$ and $\forall l_i \in L,\ \exists f_j \in F$ such that $l_i = f_j$. Therefore, the node sets V_1 and V_2 of G_1 and G_2 are the same. Now take any two vertices $f_k, f_l \in V_1$. Pick out the vertices $f_m, f_p \in V_2$ such that $f_k = f_m$ and $f_l = f_p$. Note that $vol(f_k) = vol(f_m)$ and $vol(f_l) = vol(f_p)$. Therefore, $vol(f_k) \cap vol(f_l) = vol(f_m) \cap vol(f_p)$. Hence, from the definition of the UMDG, $\exists$ an undirected edge $\{f_k, f_l\} \in E_1 \Leftrightarrow \exists$ an undirected edge $\{f_m, f_p\} \in E_2$. Therefore, the edge sets E_1 and E_2 of G_1 and G_2 are the same and therefore G_1 and G_2 are Isomorphic.

Theorem 3: *Subgraph Isomorphisms of Undirected Model Dependency Graphs.* Let G_1 be the UMDG for a solid model M. Let G_2 be the UMDG for the solid model M' that results from adding a feature f' to solid model M. G_1 is isomorphic to a subgraph of G_2.

Proof: Let $F = \{f_0, \ldots, f_n\}$ be the feature set for M. The feature set for M' is $F' = \{f_0, \ldots, f_n\} \cup \{f'\}$. The node set of G_1 is $V_1 = F = \{f_0, \ldots, f_n\}$ and the node set of G_2 is $V_2 = F' = \{f_0, \ldots, f_n\} \cup \{f'\}$. The edge set of G_2 is therefore $E_2 = E_1 \cup \{\{f', f_i\}$ such that $vol(f_i) \cap vol(f') \neq \emptyset\}$. Note that $V_1 \subset V_2$. To put this stronger $V_1 = V_2 \Leftrightarrow \{f'\}$. Also note that $E_1 \subset E_2$ and to say this stronger $E_1 = E_2 \Leftrightarrow \{\{f', f_i\}$ such that $vol(f_i) \cap vol(f') \neq \emptyset\}$. Remove the node f' from V_2 and remove all edges that have f' as an endpoint from E_2. You have now obtained G_1. Therefore, G_1 is isomorphic to a subgraph of G_2.

3.4 Comparison and Retrieval

To compare the similarity of 2 solid models, test the MDGs of the models for a D-morphism or for a subgraph D-morphism. Accomplish this by testing the corresponding UMDGs for isomorphism or subgraph isomorphism. The general problem of determining if there exists a subgraph isomorphism for a given pair of graphs is NP-complete and the graph isomorphism problem is an open question. Therefore, there is currently no known polynomial time solution for these problems [13]. However, there are two aspects of this problem domain that can be exploited to significantly reduce this complexity:

- First, it is not necessary to completely solve the isomorphism and subgraph isomorphism problems: Since we are only concerned with similarity, knowing if two UMDG's are "almost" isomorphic is sufficient. Hence, we can use a heuristic method for the Isomorphism test. Specifically, an algorithm that is a variant of gradient descent (or hill-climbing) that exploits the feature information we have in the design feature history will be described.

- Second, there is a great deal of domain knowledge present in the CAD model and in the feature history that can reduce the search space. For example, we will only consider mappings that compare similar feature types (i.e., holes map to holes, not to pockets). Additional constraints about vertex degree and size, location, and orientation can also be considered.

In the following subsections, two graph isomorphism algorithms will be described. First, I will describe a gradient descent approach to the problem. This approach is not guaranteed to find an isomorphism if one exists, but allows for a measure of similarity based on the best result obtained from executing some number of restarts of the algorithm. Next, I describe Ullmann's algorithm for subgraph isomorphism [45]. And finally I describe a

variation of Ullmann's algorithm that employs an A* search and incorporates more nodal invariants in the initialization. I call this variation of Ullmann's algorithm the *A* Subgraph Isomorphism Checker* or ASIC.

3.4.1 Gradient Descent

In testing for an isomorphism, first, arbitrarily choose an initial mapping between the nodes of the two graphs (i.e., for each node of G_1 choose at random a node of G_2 such that no two nodes of G_1 are mapped to the same node of G_2). Next, swap the mappings of the two nodes that reduce the value of the evaluation function the most. If there is no swap that reduces the value of the evaluation function, but there are swaps that result in the same value (i.e., a plateau has been reached), choose one of those at random. The algorithm ends when either every possible swap increases the value of the evaluation function or it makes P random moves on the plateau. Values of P ranging from constant values to $P = |V_1|^2$ (where V_1 is the vertex set in the smaller graph) have been experimented with.

The evaluation function is the count of the number of mismatched edges. That is, the evaluation function, $H = |E|$ such that $G_1 = (V_1, E_1)$ is the smaller of the two graphs being compared, $G_2 = (V_2, E_2)$ is the larger of the two graphs, and $E = \{(u, v) \in E_1$ such that $(((paired(u), paired(v)) \notin E_2 \wedge (paired(v), paired(u)) \notin E_2)) \vee label(u) \neq label(paired(u)) \vee label(v) \neq label(paired(v)\}$. As a measure of similarity employ the value $H^* = \frac{min\{H_1,\ldots,H_n\}}{|E_1|}$ where $H_1, \ldots, H_n$ are the values of H from up to n random restarts of the algorithm and E_1 is the edge set of the smaller graph. The function "paired(x)" above returns the node $y \in V_2$ that is currently mapped to the node $x \in V_1$. This value may be null in the case of subgraph isomorphism testing as not all nodes in the larger graph may be mapped to a node if the number of nodes of the graphs differs. The function "label(x)" used above returns the label, or attributes, of the node x.

Algorithm 3.1: Subgraph Isomorphism Approximation (Gradient Descent)
Input: $G_1 = (V_1, E_1), G_2 = (V_2, E_2)$, the two graphs being tested. P is the number of moves to make on a plateau before giving up.
Output: $H = 0$ if the graphs are found to be isomorphic or if one is found to be subgraph isomorphic to the other. Otherwise, H is returned where H is the number of mismatched edges when the algorithm halts.

```
ISOMORPHISMAPPROXIMATIONGRADIENTDESCENT(G1, G2, P)
(1)   Pairings = GETRANDOMPAIRINGS(G1, G2)
(2)   i = 0
(3)   BestResult = H(G1, G2,Pairings)
(4)   while (BestResult > 0) ∧ (i < P)
(5)     if H(G1, G2,APPLYSWAP(Pairings,BestSwap)) < BestRe-
        sult
(6)       Pairings = APPLYSWAP(Pairings,BestSwap)
(7)       i = 0
(8)       BestResult = H(G1, G2, Pairings)
(9)     else
(10)      if H(G1, G2, APPLYSWAP(Pairings,BestSwap)) = BestRe-
          sult
(11)        Pairings = APPLYSWAP(Pairings, BestSwap)
(12)        i = i + 1
(13)      else
(14)        i = P
(15)  return BestResult
```

The node labels may contain as little or as much information as you choose. For the experiments that are described later, the node labels were simply the type of feature, such as “hole” or “pocket”. However, by incorporating more information into the node labels such as dimensions or orientation, you may restrict allowable mappings which will increase the algorithm’s performance by reducing the search space. Incorporating more information in the node labels will also obtain a more meaningful similarity measure. For example, if some notion of dimension was incorporated into the labels then a really large block with a tiny hole will not be found similar to a little block with a larger hole.

Algorithm 3.1 is the algorithm developed and described for the Subgraph Isomorphism Test using gradient descent. In the algorithm, `Pairings` refers to the mapping between

Algorithm 3.2: Similarity
Input: $G_1 = (V_1, E_1), G_2 = (V_2, E_2)$, the two graphs being compared. R is the number of restarts.
Output: $S = 0$ if the smaller of the two graphs is the largest common subgraph. Otherwise, S is returned where S is the smallest result of R restarts of IsomorphismApproximationGradientDescent divided by the number of edges in the smaller of the two input graphs.
SIMILARITY(G_1, G_2, R)
(1) $i = 1$
(2) `BestResultThusFar` = ISOMORPHISMAPPROXIMATIONGRADIENTDESCENT(G_1, G_2, P)
(3) **while** (`BestResultThusFar` > 0) $\wedge$ ($i < R$)
(4) `BestResultThusFar` = $\min$ { `BestResultThusFar`, ISOMORPHISMAPPROXIMATIONGRADIENTDESCENT(G_1, G_2, P) }
(5) $i = i + 1$
(6) **return** $\frac{BestResultThusFar}{\min\{|E_1|,|E_2|\}}$

the nodes of the two graphs. And GETRANDOMPAIRINGS returns a random mapping as described above. H is the evaluation function that counts the number of mismatched edges given two graphs and a mapping between the nodes in these two graphs. `BestSwap` is the swap from the set of all possible swaps between pairings that results in a mapping with the smallest value for H. APPLYSWAP returns the mapping that results from applying the given swap to the given mapping. The algorithm is of polynomial time complexity. It takes $O(N^2)$ time to choose the best swap. In the worst possible case, by choosing the best swap at each step the evaluation function is simply reduced by one and therefore can look for the best swap as many as $|E|$ times. It takes time in $O(|E|)$ to compute the evaluation function. Also in this worst case, the algorithm reaches a plateau as often as possible and takes P random moves on each of these plateaux before finding the swap that reduces the evaluation function. So therefore the worst case complexity of the algorithm is $O(P * E^2 * N^2)$. If P is a constant then the complexity is simply $O(E^2 * N^2)$. To obtain a similarity measure, the smallest result of r executions of this algorithm is divided by the number of edges

Algorithm 3.3: Ullmann Subgraph Isomorphism
Input: $G_1 = (V_1, E_1), G_2 = (V_2, E_2)$, the two graphs being tested.
Output: true if the graphs are found to be isomorphic or false otherwise.
ULLMANN(G_1, G_2)
(1) M = INITULLMANN(G_1, G_2)
(2) **if** REFINE(M, G_1, G_2) $= 0$
(3) **return** false
(4) **else**
(5) N = SORT(V_1)
(6) **return** ULLMANNDFS(M, G_1, G_2, N, LENGTH(N)-1)

in the smaller of the graphs. Algorithm 3.2 is the random restart algorithm for similarity assessment. The similarity algorithm simply calls the gradient descent algorithm r times. Since r is constant the complexity is $O(E^2 * N^2)$.

3.4.2 Ullmann's Algorithm

Ullmann's algorithm for subgraph isomorphism is perhaps still one of the most widely used algorithms for graph and subgraph isomorphism. Even today, researchers often compare the performance of their algorithms to that of Ullmann's. The complete description of Ullmann's algorithm can be found in [45].

Ullmann's algorithm for subgraph isomorphism is a depth first tree search. Each state in the search space is represented by a matrix M. This matrix is $n \times m$ where there are n nodes in the smaller graph and m nodes in the larger. The matrix elements m_{ij} are 0 if the corresponding nodes may not be mapped to each other in any isomorphism and 1 otherwise. The initial state is generated based on the degrees of the nodes. And although not discussed in [45], it is clear that to handle attributed graphs simply incorporate the attributes into the initialization stage. Ullmann's algorithm proceeds depth-first. Each successor state binds a node mapping by setting all but one 1 in a row of the matrix M to 0. After this binding is performed, an arc consistency check is made iteratively to each remaining 1 in the matrix

Algorithm 3.4: DFS for Ullmann Subgraph Isomorphism
Input: $G_1 = (V_1, E_1), G_2 = (V_2, E_2)$, the two graphs being tested. M is the mapping matrix M previously described. N is a sorted list of the nodes of the smaller graph G_1. $Level$ is the current level of the DFS.
Output: true if the graphs are found to be isomorphic or false otherwise.

```
ULLMANNDFS(M, G1, G2, N, Level)
(1)   forall v2 in V2 do
(2)     if M[N[Level], v2] = 1
(3)       Mnew = BIND(N[Level], M, v2)
(4)       if REFINE(Mnew, G1, G2) = 0
(5)         DO NOTHING
(6)       else
(7)         if Level = 0
(8)           return true
(9)         else
(10)          if ULLMANNDFS(Mnew, G1, G2, N, Level − 1 ) = true
(11)            return true
(12)  return false
```

M. This check is referred to as Ullmann's neighborhood consistency check. Basically, for each $m_{ij} = 1$ in M it checks to ensure that for all neighbors x of i in graph G_1 there must exist a neighbor y of j in graph G_2 such that $m_{xy} = 1$. If this condition does not hold, then m_{ij} is changed to 0. If all rows of M have exactly one 1 and all columns of M have no more than one 1, and if the neighborhood consistency check does not alter M then M represents an isomorphism. Ullmann's algorithm is described in algorithms 3.3, 3.4, 3.5, 3.6, and 3.7.

The worst case complexity of Ullmann's algorithm is exponential. This occurs if all or a large number of the elements of the M matrix are 1 and if the refinement procedure fails to reduce the search space. In practice, this worst case is far from typical as can be seen later in chapter 4. The initialization stage takes $O(|V_1| * |V_2|)$ to initialize the M matrix.

Algorithm 3.5: Initialization Ullmann Subgraph Isomorphism
Input: $G_1 = (V_1, E_1), G_2 = (V_2, E_2)$, the two graphs being tested.
Output: The matrix M of allowable mappings between the nodes of the graphs.

INITULLMANN(G_1, G_2)
(1) **forall** v_i **in** V_1 **do**
(2) **forall** v_j **in** V_2 **do**
(3) **if** $|V_1| = |V_2|$ and $|E_1| = |E_2|$
(4) **if** DEGREE(v_i) = DEGREE(v_j) and ATTRIBUTES(v_i) = ATTRIBUTES(v_j)
(5) $M[v_i, v_j] = 1$
(6) **else**
(7) $M[v_i, v_j] = 0$
(8) **else**
(9) **if** DEGREE(v_i) $\leq$ DEGREE(v_j) and ATTRIBUTES(v_i) = ATTRIBUTES(v_j)
(10) $M[v_i, v_j] = 1$
(11) **else**
(12) $M[v_i, v_j] = 0$

Algorithm 3.6: Bind for Ullmann Subgraph Isomorphism
Input: $Node$ is the node being bound to a mapping. M is the mapping matrix. $Node2$ is the node of G_2 to which $Node$ is being bound.
Output: Returns the new M

BIND($Node, M, Node2$)
(1) **forall** v_j **in** V_2 **do**
(2) **if** $v_j! = Node2$
(3) $M[Node, v_j] = 0$
(4) **forall** v_j **in** V_1 **do**
(5) **if** $v_j! = Node$
(6) $M[v_j, Node2] = 0$
(7) **return** M

Algorithm 3.7: Neighborhood Consistency
Input: $G_1 = (V_1, E_1), G_2 = (V_2, E_2)$, the two graphs being tested. The matrix M of allowable mappings between the nodes of the graphs.
Output: false if there is an all zero row and true otherwise
REFINE(M, G_1, G_2)

```
(1)   do
(2)     Changed = false
(3)     forall v_i in V_1 do
(4)       ZeroRow = true
(5)       forall v_j in V_2 do
(6)         if M[v_i, v_j] = 1
(7)           forall x adjacent to v_i do
(8)             GoodOne = false
(9)             forall y adjacent to v_j do
(10)              if M[x, y] = 1
(11)                GoodOne = true
(12)                break
(13)            if GoodOne = false
(14)              M[v_i, v_j] = 0
(15)              Changed = true
(16)            else
(17)              ZeroRow = false
(18)      if ZeroRow = true
(19)        return false
(20)  while Changed
(21)  return true
```

Algorithm 3.8: A* for ASIC
Input: $G_1 = (V_1, E_1), G_2 = (V_2, E_2)$, the two graphs being tested. M is the mapping matrix M previously described. H is a sorted list of the not yet bound nodes of the smaller graph G_1.
Output: true if the graphs are found to be isomorphic or false otherwise.

```
ASTAR(M, G1, G2, H)
(1)   S = CREATESTATE(M, H)
(2)   h(S) = LENGTH(H)
(3)   g(S) = 0
(4)   INITPQUEUE(Q)
(5)   ADDTOPQUEUE(Q, S, h(S) + g(S))
(6)   while NOTEMPTY(Q)
(7)     S = REMOVEMIN(Q)
(8)     M = GETM(S)
(9)     H = GETH(S)
(10)    forall v2 in V2 do
(11)      if M[H[h(S) - 1], v2] = 1
(12)        Mnew = BIND(H[h(S) - 1], M, v2)
(13)        if REFINE2(Mnew, G1, G2, Hnew) = 0
(14)          DO NOTHING
(15)        else
(16)          if LENGTH(Hnew) = 0
(17)            return true
(18)          else
(19)            Snew = CREATESTATE(Mnew, Hnew)
(20)            h(Snew) = LENGTH(Hnew)
(21)            g(Snew) = g(S) + 1
(22)            ADDTOPQUEUE(Q, Snew, h(Snew) + g(Snew) )
(23)  return false
```

3.4.3 A* Subgraph Isomorphism Checker (ASIC)

As stated previously, Ullmann's algorithm for subgraph isomorphism is still widely used today. In the worse case, it requires exponential time. But in practice on graphs encountered in everyday applications, the time complexity is tractable. However, can we do better?

In an attempt to do better than Ullmann's algorithm for subgraph isomorphism, I describe the *A* Subgraph Isomorphism Checker* (ASIC). ASIC is a variation of Ullmann's algorithm that incorporates an A* search rather than a depth-first search. It includes some

Algorithm 3.9: A* Subgraph Isomorphism Checker (ASIC)
Input: $G_1 = (V_1, E_1), G_2 = (V_2, E_2)$, the two graphs being tested.
Output: true if the graphs are found to be isomorphic or false otherwise.
ASIC(G_1, G_2)
(1) M = INITASIC(G_1, G_2)
(2) **if** REFINE2(M, G_1, G_2, H) = 0
(3) **return** false
(4) **else**
(5) **if** LENGTH(H) = 0
(6) **return** true
(7) **else**
(8) H = SORT(H)
(9) **return** ASTAR(M, G_1, G_2, H)

other modifications as well. ASIC is described in algorithms 3.8, 3.9, 3.10, 3.11, 3.12, and 3.13.

The first of these modifications is in the initialization of the matrix M. Ullmann's algorithm initializes this matrix solely on the basis of the degrees of the nodes and the attributes of the nodes. ASIC will instead initialize M based on *n-Region Density*. The n-region density of a node v is the number of nodes reachable from v along a path no longer than n. If the graphs are of the same size and isomorphism is being tested then for all $n = 1, 2, \ldots, N \Leftrightarrow 1$ the n-region density must be the same for any two nodes that are mapped to each other. And for subgraph isomorphism testing, a node in the larger graph must have at least as many nodes in its n-region density as does the node in the smaller graph to which it is to be mapped for all $n = 1, 2, \ldots, N \Leftrightarrow 1$. The n-region density for all n and for all nodes of a graph is easily calculated in $O(N^3)$ time where N is the number of nodes in the graph. Also incorporated into the initialization stage is the sum of the degrees of the adjacent nodes of a node. For isomorphism testing this value must be equal and for subgraph isomorphism testing this value for a node in the smaller graph must be no larger than that of a node in the larger graph to which it is mapped. This new initialization stage is described in algorithm 3.11.

The next modification is the use of A* search rather than depth-first. Algorithm 3.8 shows this modification. Rather than choosing a successor state at the deepest level of the search space, ASIC chooses a successor state s as to minimize the function $f(s) = g(s) + h(s)$ where $g(s)$ is the depth the state s is in the search space and $h(s)$ is the number of yet unbound nodes in the graph. This heuristic $h(s)$ is not admissible as the refinement procedure will remove some 1s from the matrix M and possibly result in nodes being bound as a side effect, but this is not important. We do not care particularly whether or not we took the optimal path to find the isomorphism as we are only interested in finding an isomorphism. To calculate $h(s)$, the refinement procedure of Ullmann's algorithm has been modified and both refinement and this calculation of $h(s)$ are computed simultaneously. The modified refinement algorithm is described in algorithm 3.10.

The complexity of ASIC in the worst case is still exponential and occurs under the same conditions as the worst case of Ullmann's algorithm. I will explore in chapter 4 experimentally how ASIC performs compared to Ullmann's algorithm. ASIC does however suffer from a higher complexity initialization stage. To calculate the neighbor degree sums, if the graph was completely connected, would take $O(N)$ time. To compute the n-region density, you must first compute all pairs shortest paths in $O(N^3)$ time. The initialization stage is therefore $O(N^3)$.

Algorithm 3.10: Neighborhood Consistency with Heuristic Calculation
Input: $G_1 = (V_1, E_1), G_2 = (V_2, E_2)$, the two graphs being tested. The matrix M of allowable mappings between the nodes of the graphs. H is an output variable and will upon completion contain a list of the not yet bound nodes of graph G_1.
Output: false if there is an all zero row and true otherwise

```
REFINE2(M, G1, G2, H)
(1)   do
(2)     Changed = false
(3)     CLEAR(H)
(4)     forall v_i in V_1 do
(5)       ZeroRow = true
(6)       NumberOfOnes = 0
(7)       forall v_j in V_2 do
(8)         if M[v_i, v_j] = 1
(9)           forall x adjacent to v_i do
(10)            GoodOne = false
(11)            forall y adjacent to v_j do
(12)              if M[x, y] = 1
(13)                GoodOne = true
(14)                break
(15)            if GoodOne = false
(16)              M[v_i, v_j] = 0
(17)              Changed = true
(18)            else
(19)              ZeroRow = false
(20)              NumberOfOnes = NumberOfOnes + 1
(21)              if NumberOfOnes = 2
(22)                APPEND(H, v_i)
(23)       if ZeroRow = true
(24)         return false
(25)  while Changed
(26)  return true
```

Algorithm 3.11: Initialization for ASIC
Input: $G_1 = (V_1, E_1), G_2 = (V_2, E_2)$, the two graphs being tested.
Output: The matrix M of allowable mappings between the nodes of the graphs.

```
INITASIC(G1, G2)
(1)   RegionDensity1 = COMPUTEREGIONDENSITY(G1)
(2)   RegionDensity2 = COMPUTEREGIONDENSITY(G2)
(3)   DegreeSums1 = CALCNEIGHBORDEGREESUMS(G1)
(4)   DegreeSums2 = CALCNEIGHBORDEGREESUMS(G2)
(5)   forall vi in V1 do
(6)     forall vj in V2 do
(7)       if |V1| = |V2| and |E1| = |E2|
(8)         if DegreeSums1[vi] = DegreeSums2[vj] and ATTRIBUTES(vi) =
            ATTRIBUTES(vj)
(9)           M[vi, vj] = 1
(10)        else
(11)          M[vi, vj] = 0
(12)          break
(13)      else
(14)        if DegreeSums1[vi] ≤ DegreeSums2[vj] and ATTRIBUTES(vi) =
            ATTRIBUTES(vj)
(15)          M[vi, vj] = 1
(16)        else
(17)          M[vi, vj] = 0
(18)          break
(19)      for k from 1 to |V1| − 1 do
(20)        if |V1| = |V2| and |E1| = |E2|
(21)          if RegionDensity1[vi, k] = RegionDensity2[vj, k]
(22)            DO NOTHING
(23)          else
(24)            M[vi, vj] = 0
(25)            break
(26)        else
(27)          if RegionDensity1[vi, k] ≤ RegionDensity2[vj, k]
(28)            DO NOTHING
(29)          else
(30)            M[vi, vj] = 0
(31)            break
```

Algorithm 3.12: Neighbor Degree Sum Calculation
Input: $G_1 = (V_1, E_1)$ a graph.
Output: an array DegreeSums containing the sum of the degrees of the adjacent nodes for each node.

```
CALCNEIGHBORDEGREESUMS(G_1)
(1)   forall v_i in V_1 do
(2)     DegreeSums[v_i] = 0
(3)   forall v_i in V_1 do
(4)     forall x adjacent to v_i do
(5)       DegreeSums[v_i] = DegreeSums[v_i] + DEGREE(x)
(6)   return DegreeSums
```

Algorithm 3.13: n-Region Density Calculation
Input: $G_1 = (V_1, E_1)$ a graph.
Output: an matrix RegionDensity containing the n-Region densities as described.

```
COMPUTEREGIONDENSITY(G_1)
(1)   Paths = ALLPAIRSSHORTESTPATHS(G_1)
(2)   forall v_i in V_1 do
(3)     for k from 0 to |V_1| - 1 do
(4)       RegionDensity[v_i, k] = 0
(5)   forall v_i in V_1 do
(6)     forall v_j in V_1 do
(7)       if v_i ≠ v_j
(8)         for k from Paths[v_i, v_j] to |V_1| - 1 do
(9)           RegionDensity[v_i, k] = RegionDensity[v_i, k] + 1
(10)  return RegionDensity
```

Chapter 4

Experiments

In this chapter, I will report the results of experimental studies of the algorithms. In section 4.1, I present the results of experiments on randomly generated graphs. These experiments were for isomorphism. That is, only pairs of graphs of the same size were tested. In section 4.2, I present experiments conducted on pairs of randomly generated graphs of varying sizes. These experiments test each pair of graphs for a subgraph isomorphism. In section 4.3, I present experiments on MDG graphs. The algorithms have been implemented in C++ using the LEDA graph library. The MDG experiments were performed on a Sun UltraSPARC 30 workstation running Sun Solaris 2.6. The isomorphism and subgraph isomorphism experiments were performed on a Sun UltraSPARC 10 workstation running Sun Solaris 2.7.

The randomly generated graphs used for the experiments of section 4.1 and section 4.2 were generated for varying numbers of nodes and edge densities. The edge density of the graph is the probability that an edge exists between any given pair of nodes. These random graphs follow the G_{np} model.

4.1 Isomorphism Experiments

Figure 4.1 shows surface plots of the timing results for the ASIC algorithm, Ullmann's algorithm, and two variations for pairs of isomorphic graphs. Along the x-axis of each are number of nodes. Along the y-axis is the edge density. The edge density is a value between 0 and 1. Edge density 1 is a complete graph. Edge density 0 is a completely disconnected graph. CPU time is on the z-axis. As can be seen, Ullmann's algorithm performs terribly for isomorphic graphs, especially for very dense graphs. For example, for 150 node graphs with edge density 0.8, Ullmann's algorithm takes over 600 seconds. ASIC performs much better. The plot for ASIC does not go up as steep as Ullmann's. The timing results are more uniform across edge densities as well, whereas with Ullmann's algorithm, denser graphs take an enormous amount of extra time. Ullmann's algorithm with the addition of ASIC's initialization stage performs roughly the same as ASIC. ASIC has a slight timing advantage not clearly visible in the plots. Both ASIC and Ullmann's plus ASIC's initialization take approximately 30 seconds of CPU time for the 150 node 0.8 edge density case. One particularly interesting thing to note is the performance of ASIC with relaxed initialization. The initialization was relaxed to only use node degrees, but the algorithm still uses A* search. For this variation of ASIC, graphs of 300 nodes and edge density 0.8 only require approximately 60 seconds of CPU time. The surface plot for ASIC with relaxed initialization is far less steep, although like Ullmann's denser graphs require more time.

Figure 4.3 shows the ASIC algorithm, ASIC with relaxed initialization, Ullmann's algorithm, and Ullmann's algorithm with the addition of ASIC's initialization for three different edge densities: 0.8, 0.5, and 0.2. The data presented is for pairs of isomorphic graphs. In all cases, ASIC with relaxed initialization far out-performs the other three algorithms by a large margin. Ullmann's algorithm performs the worst in all cases for all edge densities.

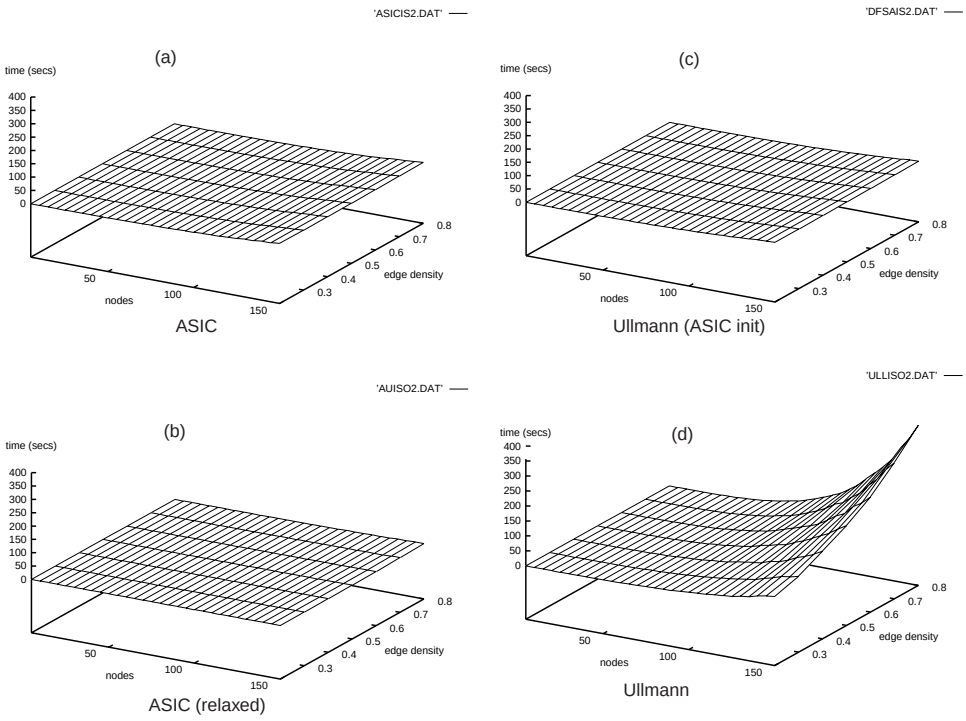


Figure 4.1: Surface plots of timing results of (a) ASIC; (b) ASIC with relaxed initialization; (c) Ullmann's with ASIC's initialization; (d) Ullmann's for pairs of isomorphic graphs.

So it appears that the true power of ASIC is from using A* search. The extra initialization adds time to the comparison of isomorphic graphs.

Figure 4.2 shows surface plots of the timing results for the ASIC algorithm, Ullmann's algorithm, and two variations for pairs of non-isomorphic graphs. Along the x-axis of each are number of nodes. Along the y-axis is the edge density. CPU time is on the z-axis. The ASIC algorithm and Ullmann's algorithm with the additional initialization of ASIC appear to behave roughly the same. These plots are roughly uniform across edge densities. Graphs of 150 nodes take approximately 25 seconds to compare. For non-isomorphic graphs, however, Ullmann's algorithm and ASIC with relaxed initialization are much faster. Their surface plots increase at a very slow gradual rate with a few odd cases here and there

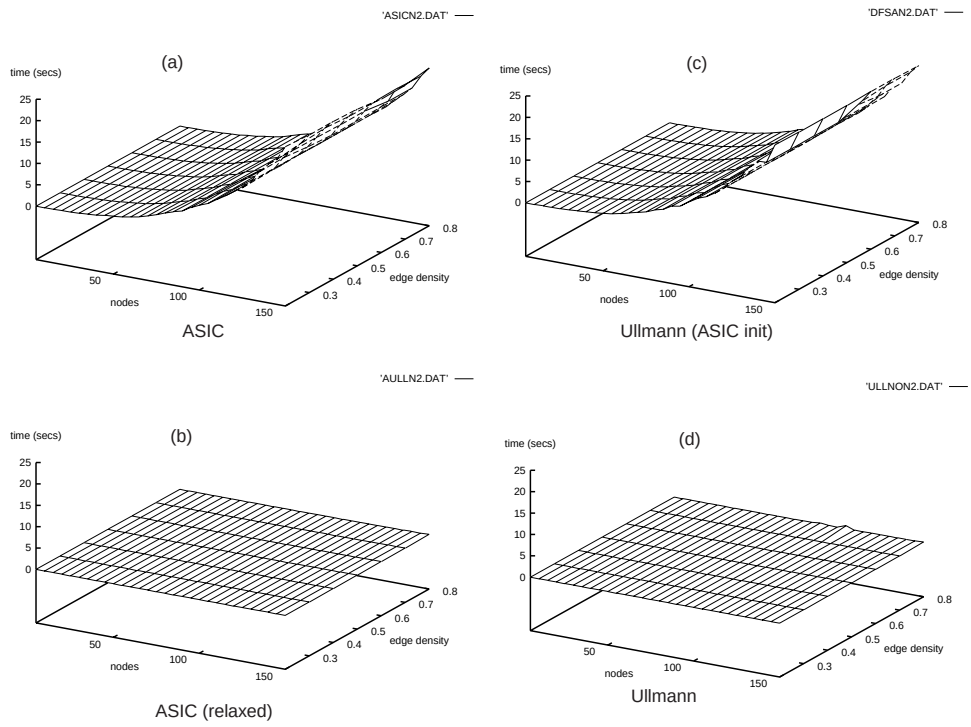


Figure 4.2: Surface plots of timing results of (a) ASIC; (b) ASIC with relaxed initialization; (c) Ullmann's with ASIC's initialization; (d) Ullmann's for pairs of non-isomorphic graphs.

"jumping off the surface." The odd cases take in the range of 8-10 seconds. Most cases however, up to 400 node graphs of all densities, require far less time.

Figure 4.4 shows the ASIC algorithm, ASIC with relaxed initialization, Ullmann's algorithm, and Ullmann's algorithm with the addition of ASIC's initialization for three different edge densities: 0.8, 0.5, and 0.2. The data presented is for pairs of non-isomorphic graphs. In all cases, ASIC performs roughly the same as Ullmann's with the addition of ASIC's initialization. Also in all cases, ASIC with relaxed initialization performs roughly the same as Ullmann's algorithm. These results are to be expected. Considering that the test graphs in these cases are non-isomorphic and assuming that the neighborhood consistency check effects both Ullmann's and ASIC in the same way, both of these algorithms should ex-

amine the same number of search states regardless of using A* or depth-first search. The initialization stage should be expected to be the sole cause of timing differences for the algorithms for non-isomorphic graphs as we have seen.

From these experiments, we can see that ASIC with the relaxed initialization stage is the best choice for graphs of the same number of nodes and edges. For isomorphic pairs, it tends to find the isomorphism far faster than the other algorithms examined. This is due to the guidance of the heuristic in the A* search. And for non-isomorphic pairs, ASIC with relaxed initialization performs no worse than the other algorithms examined. So is there any benefit to using the additional initialization stage? This question will be answered in the next section.

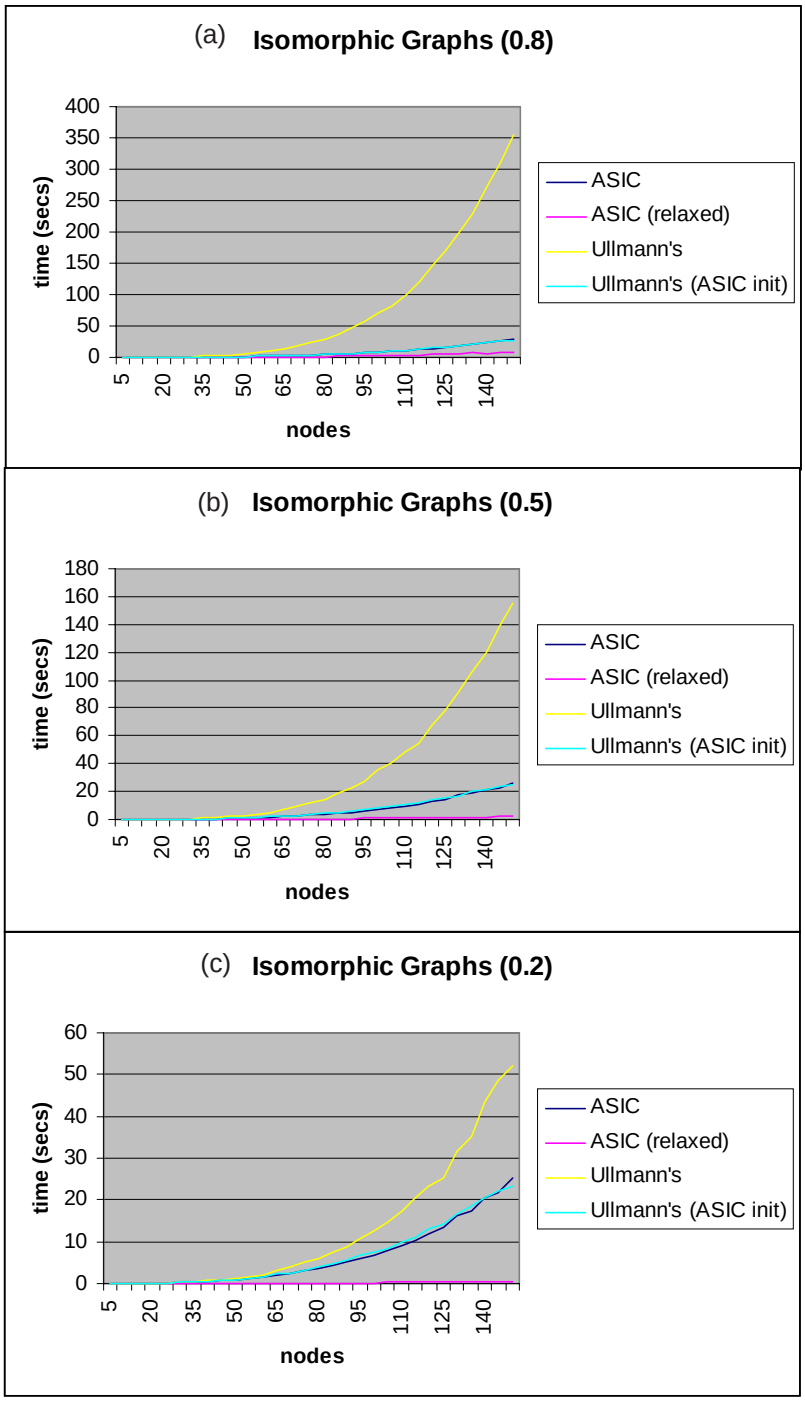


Figure 4.3: Timing results for pairs of isomorphic graphs of edge densities (a) 0.8; (b) 0.5; (c) 0.2.

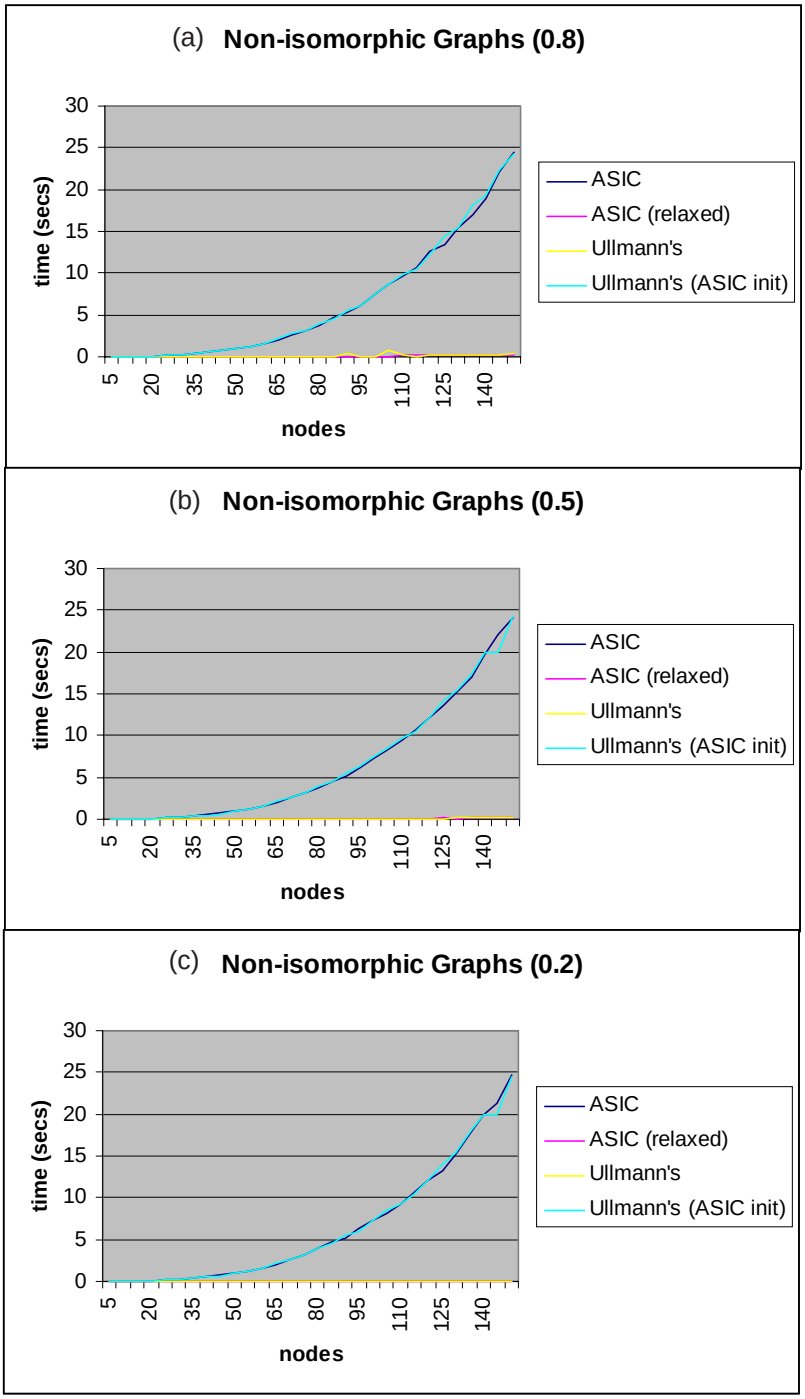


Figure 4.4: Timing results for pairs of non-isomorphic graphs of edge densities (a) 0.8; (b) 0.5; (c) 0.2.

4.2 Subgraph Isomorphism Experiments

Figure 4.5 shows surface plots of the timing results for the ASIC algorithm, Ullmann's algorithm, and two variations for pairs of subgraph isomorphic graphs of differing sizes. These experiments search for a subgraph of the larger graph that is isomorphic to the smaller graph. Along the x-axis of each are the number of nodes of the larger graph. Along the y-axis is the difference in the number of nodes of the graphs. The edge density of both graphs is 0.5. CPU time is on the z-axis in seconds. These four plots appear to be very similar to each other and do not appear to suggest any one algorithm having a large advantage over the others. The plots are flat for the particularly small test cases and then as the graphs increase in size, this flat area begins to have sharp peaks. For all four algorithms, these peaks appear as the difference in size of the pair of graphs gets further apart. This is to be expected as the nodes of the smaller graph have a larger number of nodes in the larger that they may map to. One significant difference in the performance of the algorithms is that for Ullmann's algorithm these peaks in the surface begin appearing when the graphs are closer in size to each other as compared to when they appear in the ASIC algorithm and in the ASIC with relaxed initialization algorithm. Also note that ASIC appears to be the best choice of algorithm as the graphs get further apart in size. The peaks in the plot for ASIC are not as steep as they are for the other algorithm variations. This suggests that for subgraph isomorphism testing, as the graphs get further apart in size, ASIC with full initialization is the best choice of algorithm.

Figure 4.6 shows surface plots of the timing results for the ASIC algorithm, Ullmann's algorithm, and two variations for pairs of non-isomorphic graphs of differing sizes. These experiments search for a subgraph of the larger graph that is isomorphic to the smaller graph. Along the x-axis of each are the number of nodes of the larger graph. Along the y-axis is the difference in the number of nodes of the graphs. The edge density of both

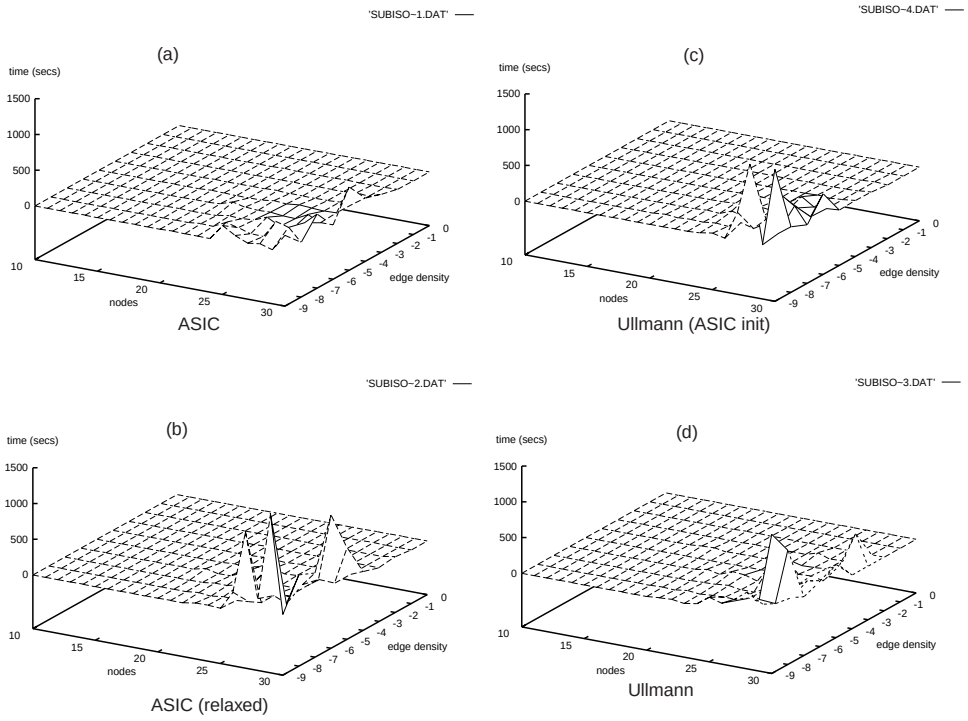


Figure 4.5: Surface plots of timing results of (a) ASIC; (b) ASIC with relaxed initialization; (c) Ullmann's with ASIC's initialization; (d) Ullmann's for pairs of random subgraph isomorphic graphs of different sizes.

graphs is 0.5. CPU time is on the z-axis in seconds. Ullmann's algorithm appears to be the slowest for these experiments. Just as with pairs of subgraph isomorphic graphs, there is significant improvement in the use of the ASIC algorithm for pairs of non-subgraph isomorphic graphs.

Although we saw that for graphs of the same size, ASIC with relaxed initialization performed the best, it has been shown here that the true power of the added initialization lies in subgraph isomorphism testing. This is especially true as the size of the graphs being tested become further apart. Subgraph isomorphism is a harder problem than graph isomorphism in terms of complexity [13]. So the added initialization does not overwhelm the complexity of the overall algorithm for subgraph isomorphism as it does for graph

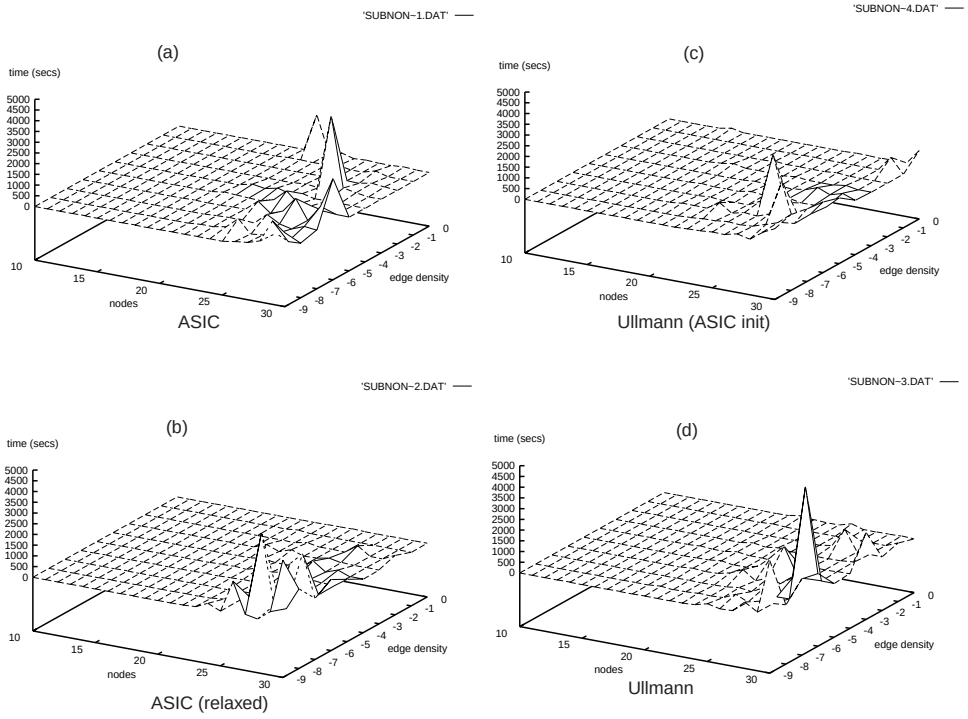
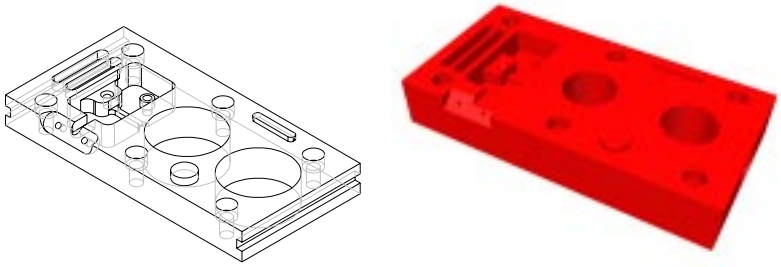


Figure 4.6: Surface plots of timing results of (a) ASIC; (b) ASIC with relaxed initialization; (c) Ullmann's with ASIC's initialization; (d) Ullmann's for pairs of random non-isomorphic graphs of different sizes.

isomorphism. From examining these results and those of the previous section, we can conclude that for graphs of the same size or close to the same size, the best choice of algorithm is ASIC using relaxed initialization. And for graphs of drastically different sizes, the best choice of algorithm is ASIC with the full initialization stage.



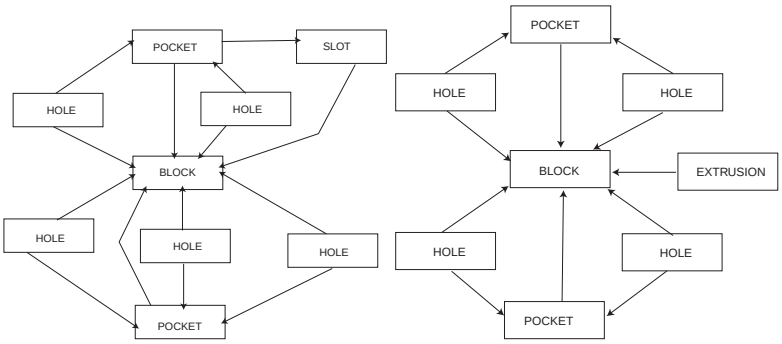
(a) TEAM (b) TEAM-2

Figure 4.7: Two of the test parts from the DOE TEAM Project. Both of these parts are available from the National Design Repository at http://repos.mcs.drexel.edu.

4.3 MDG Experiments

A family of 10,002 solid models were generated using the ACIS 3D Toolkit running on 450MHz Pentium II running Microsoft Windows NT 4.0. These models were pseudo-random variations on the US Department of Energy's Technologies Enabling Agile Manufacturing (TEAM) Project test parts pictured in Figure 4.7. These parts have a variety of standard feature types, such as pockets, slots, holes, counterbore holes, and bosses; in addition, many of the features interact and intersect, leading to a variety of different possible orderings for design feature histories and manufacturing process plans. The two parts pictured have several subtle differences that make them a useful target domain for experimentation.

The random "TEAM part" generator is based on the work of Alexei Elinson at the University of Maryland at College Park [12]. It operates by varying the number of features, the location features, and the number of different feature types over the part (depressions and protrusions, pockets, holes). For each of the 10,002 models generated, the design feature history of each model was stored. Using this feature information, the MDG for



(a) Query Model 1 (b) Query Model 2

Figure 4.8: The MDGs for the randomly generated Query Models.

each model was generated and converted to UMDG form.

Next, two arbitrary Query Models were selected from the set of 10,002 random models—these are shown in Figure 4.9. The figure shows the design histories of these parts; their MDG graphs are shown in Figures 4.8 (a) and 4.8 (b). Each of the query parts was compared to each part from the set of randomly generated parts. To perform MDG comparison, the random restart gradient descent algorithm was used with number of restarts fixed at 1000. These matching tests searched for a subgraph of the larger of the query UMDG and the given UMDG from the set of 10,002 that was isomorphic to the smaller. The matching algorithms are implemented in C++ using the LEDA graph library. The tests were performed on a Sun UltraSPARC 30 workstation running Sun Solaris 2.6.

Figure 4.10 shows the results of these two queries. The histograms show that each query model partitioned the set of 10,002 random parts into distinct subsets, based on the result of the subgraph isomorphism test. For both query parts, there was a high percentage of parts found to be "similar." This is to be expected, since the set of parts consist of a family of parts generated at random from a limited set of operations based on the TEAM parts. This

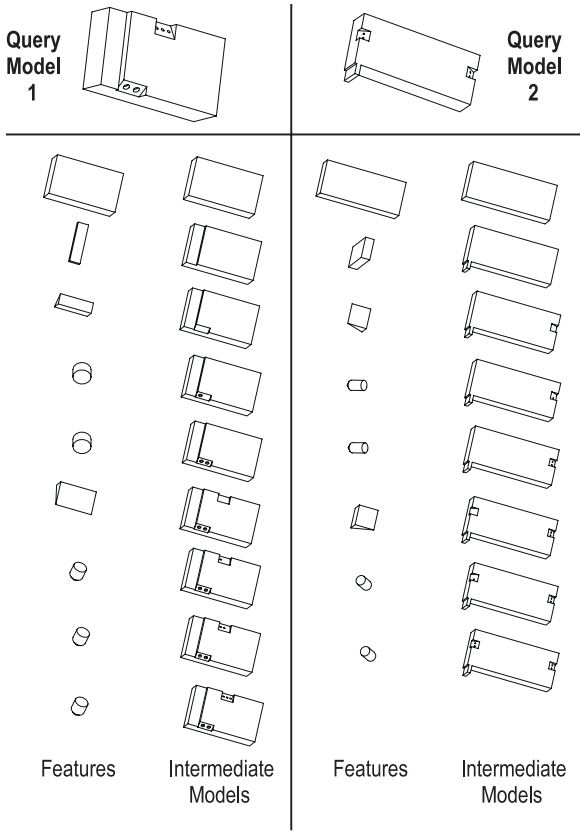


Figure 4.9: Two randomly generated query models with their design feature histories.

is also to be expected because the nodes of the UMDGs were attributed only with feature type. If other attributes such as dimension, orientation, tolerances, and so forth had been used then it could be expected that the models would have been distributed more uniformly across similarity classes or possibly more heavily clustered toward the less similar range. For both queries, the query models were successfully retrieved.

Results for Query Model 1. For **Query Model 1**, 3128 models were found such that their SMDGs were subgraph isomorphic to that of the query model or that the SMDG of the query model was subgraph isomorphic to it. Among this set was the query model itself.

Also among this set was model (a) in Figure 4.10. If you look at this model you will see that, like **Query Model 1**, it consists of two pockets each cutting through two faces, one with 2 holes and the other with 3 holes. Also common to both **Query Model 1** and (a) is a slot adjacent to one of the pockets. These two parts are very much alike. In fact, in this case, the parts were not only subgraph isomorphic, but were actually isomorphic.

Next, notice model (b). This model was among 2406 models where the ratio of "mis-matched" edges to total edges at the completion of the matching test was greater than 0 but less than or equal to 0.125. The actual value of this particular case was 0.07. Aside from the interaction between one of the pockets and the slot in **Query Model 1**, the UMDG for the query model would be isomorphic to that of model (b).

Models (c), (d), (e), and (f) were in the next four groups shown on the histogram for query 1 respectively. Model (c) has an additive feature on one of its side faces while the query model had no such feature. Model (d) has 5 pockets and holes in each and lacks the slot that the **Query Model 1** has. Model (e) has two additive features on two of its side faces while the **Query Model 1** has no such additive features. None of the edges of the UMDG of model (f) matched any of that of the query model. This part has one additive feature at one end and no other features. The query model does not have an additive feature like this one.

Results for Query Model 2. For **Query Model 2**, 2440 models were found such that they were subgraph isomorphic to the UMDG of the query model or that the UMDG of the query model was subgraph isomorphic to it. Among this set was the query model itself. Also among this set was model (g) in Figure 4.10. If you examine these two models, you will see that each has an additive feature on one side face and each has two pockets each cutting through two faces with holes in each. They are very much alike.

Model (h) is one of 3923 models with a ratio of mis-matched edges to total edges

greater than 0 and less than or equal to 0.125. This ratio for model (h) was actually 0.09. The difference between these two models is that (h) has a slot while **Query Model 2** has an additive feature on on of its side faces.

Models (i), (j), (k), and (l) are in the next four groups on the histogram. Model (i) has two slots not in the query model and the query model has the additive feature on one of the side faces. Model (j) is the same model as (d). This model was about the same in dissimilarity to both query models. The UMDG of model (k) is a 50 percent match to that of the query. This model has a pocket cutting through two faces with one hole through the pocket. Similarly the query model has a pocket like this. Model (k) also has two slots, but the query model does not. Every edge in the UMDG for model (l) was mis-matched when compared to that of **Query Model 2**. Model (l) has a slot in two of its side faces and no other features. The query model has no slots.

Statistics. All 10002 models used in this experiment along with their design history are available as ACIS `.sat` files at http://repos.mcs.drexel.edu/CICIRELLO-THESISDATA.

To compare **Query Model 1** against all 10002 models took a total of 23 hours, 17 minutes, and 23 seconds of CPU time on the Sun UltraSPARC 30 (an average of 8.38 seconds per comparison). The fastest comparison took less than 0.01 seconds. The slowest comparison took 183.35 seconds. There were a few cases where the random initial starting point represented an isomorphism, but this was a rare occurrence. On average, the algorithm made 3153 swaps of node mappings with a high of 7699 node mapping swaps.

To compare **Query Model 2** against all 10002 models took a total of 14 hours, 8 minutes, and 33 seconds of CPU time on the Sun UltraSPARC 30 (an average of 5.09 seconds per comparison). The fastest comparison took less than 0.01 seconds. The slowest comparison took 104.37 seconds. There again were a few cases where the random initial starting point represented an isomorphism, but again this was a rare occurrence. On average, the

Table 4.1: Accuracy of the Gradient Descent Algorithm using 1000 random restarts.

Query	Found Isomorphic	Actual Isomorphic	Percent Accurate
1	3128	3515	88.99
2	2440	2611	93.45

algorithm made 3026 swaps of node mappings with a high of 6493 node mapping swaps.

Table 4.1 shows how accurate the gradient descent algorithm is. For 88.99 percent of the UMDGs subgraph isomorphic to the UMDG of Query model 1, the subgraph isomorphism was found. For 93.45 percent of the UMDGs subgraph isomorphic to the UMDG of Query Model 2, the subgraph isomorphism was found.

If you are willing to trade off accuracy for time, then fewer random restarts result in faster runtimes. But, the gradient descent algorithm will find the isomorphism when it exists less often. Table 4.2, table 4.3, and table 4.4 show the accuracy of the gradient descent algorithm with 100, 10, and 0 restarts, respectively. If you reduce the number of restarts to 100 then for query 1, 80.06 percent of the subgraph isomorphic pairs were found to be subgraph isomorphic and for query 2, 90.46 percent of the subgraph isomorphic pairs were found to be subgraph isomorphic. And for queries 1 and 2, respectively, with no restarts, 74.82 percent and 88.85 percent of the subgraph isomorphic pairs were found to be subgraph isomorphic. So in terms of similarity assessment, it is not really necessary to have high numbers of restarts.

ASIC and the UMDG. Next, these same experiments were executed using the ASIC algorithm, Ullmann's subgraph isomorphism algorithm, and the two variations discussed previously. The disadvantage of these algorithms is that they only detect whether or not

Table 4.2: Accuracy of the Gradient Descent Algorithm using 100 random restarts.

Query	Found Isomorphic	Actual Isomorphic	Percent Accurate
1	2814	3515	80.06
2	2362	2611	90.46

Table 4.3: Accuracy of the Gradient Descent Algorithm using 10 random restarts.

Query	Found Isomorphic	Actual Isomorphic	Percent Accurate
1	2616	3515	74.42
2	2317	2611	88.74

Table 4.4: Accuracy of the Gradient Descent Algorithm using 0 random restarts.

Query	Found Isomorphic	Actual Isomorphic	Percent Accurate
1	2630	3515	74.82
2	2320	2611	88.85

Table 4.5: CPU performance of various algorithms on query 1 in seconds.

Algorithm	Total Time	Average	Longest	Shortest
ASIC	188.72	0.019	0.37	< 0.01
ASIC (relaxed)	41.61	0.004	0.11	< 0.01
Ullmann's	61.11	0.006	0.4	< 0.01
Ullmann's (ASIC init)	180.5	0.18	0.32	< 0.01
Gradient Descent (1000)	83843.82	8.38	183.35	< 0.01
Gradient Descent (100)	9396.74	0.94	18.55	< 0.01
Gradient Descent (10)	1011.42	0.101	2.07	< 0.01
Gradient Descent (0)	101.77	0.0102	0.22	< 0.01

a subgraph isomorphism exists and do not provide an easily quantifiable estimation of "similar" the graphs are. However, these algorithms perform far faster than the gradient descent approach. Timing results for both query 1 and query 2 can be seen in table 4.5 and table 4.6, respectively. For query 1, ASIC with relaxed initialization performed the fastest. For query 2, Ullmann's outperformed ASIC with relaxed initialization by a very small margin. ASIC with complete initialization for query 1, the worst case was better than Ullmann's worst case, although on average it was slower.

Even with no restarts the gradient descent algorithm still does not compete with ASIC in terms of time performance. However, the gradient descent algorithm provides a measure of similarity based on the lowest value of the evaluation function. This is an advantage

Table 4.6: CPU performance of various algorithms on query 2 in seconds.

Algorithm	Total Time	Average	Longest	Shortest
ASIC	167.21	0.017	0.38	< 0.01
ASIC (relaxed)	36.74	0.004	0.11	< 0.01
Ullmann's	25.63	0.003	0.06	< 0.01
Ullmann's (ASIC init)	151.88	0.015	0.33	< 0.01
Gradient Descent (1000)	50913.13	5.09	104.37	< 0.01
Gradient Descent (100)	5467.47	0.55	11.48	< 0.01
Gradient Descent (10)	622.87	0.062	1.33	< 0.01
Gradient Descent (0)	70.59	0.007	0.14	< 0.01

over the ASIC algorithm. The ASIC algorithm determines isomorphism and subgraph isomorphism quickly but does not provide such a measure of similarity in the case when no isomorphism exists. An attempt was made to use the minimum value of the heuristic function as a measure of similarity. However, it was not found to be useful. The result was that all of the isomorphic or subgraph isomorphic graphs to the query graph were partitioned into one set. And all but a few of the non-isomorphic graphs were in another set at the highest possible value of the heuristic. The few remaining cases were scattered between. These few cases accounted for less than 1 percent of the cases. This is good and bad news. The good news is that it means that the ASIC algorithm for subgraph isomorphism testing of UMDG graphs eliminates most non-isomorphic cases almost immediately, either with the initialization stage or with the neighborhood consistency check. The bad news is that the heuristic value cannot be used as a measure of similarity.

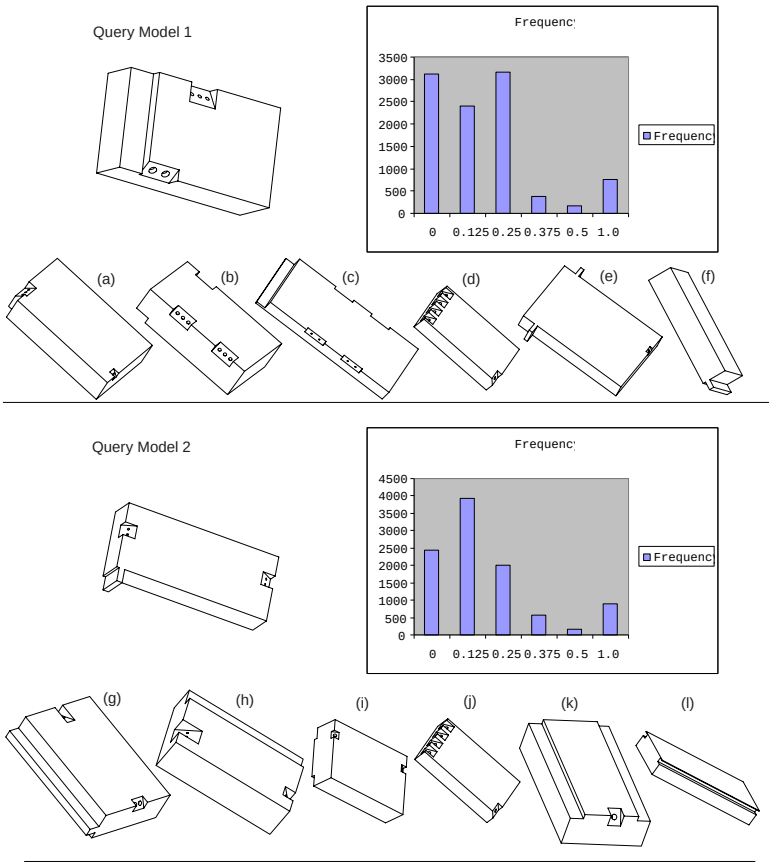


Figure 4.10: Example output data from examining subgraph isomorphism over the database of 10,002 solid models for the two query models in Figure 4.9. The histogram shows the number of models (from the 10,002 in the database) that fall into distance categories based on the subgraph isomorphism test. Read from left-to-right, the returned models are in order of decreasing similarity to the query model.

Chapter 5

Conclusions

5.1 Contributions

This thesis has presented an approach to the problem of retrieving solid models from knowledge-bases of CAD data based on the similarity of the models' structure. The idea is to enable the intelligent retrieval of solid models along with other data related to the parts that these models represent. The ultimate goal is to improve the techniques in which knowledge-bases of CAD data are managed in a positive way.

The retrieval techniques presented are based on comparing solid models for feature-based similarity. This thesis has defined similarity to mean similar in shape or structure. More specifically, it has defined similarity to mean that two models have the same features and feature interactions. To accomplish this feature-based similarity assessment, representations of features and feature interactions, termed the *Model Dependency Graph* (MDG) and alternatively the *Undirected Model Dependency Graph*, have been developed. These representations make use of a very general definition of a feature as a structural or volumetric property of the solid model and are not specific to any one definition of feature. The MDG and UMDG prove useful for representing manufacturing features and feature interactions as well as design features. Constructive solid geometry (CSG) primitives may also be seen as features under this definition of feature. The MDG and UMDG also par-

tially handle the problem of variation and non-uniqueness associated with design feature histories.

Upon defining the MDG and UMDG, algorithms for the comparison of solid models based on these representations were developed. These algorithms determine if two solid models are similar in structure based on features by checking for a graph isomorphism or a subgraph isomorphism between the UMDGs associated with the solid models in question. One of the comparison algorithms that are described is an inexact method based on a gradient descent approach to the problem. This algorithm is not guaranteed to find an isomorphism if it exists but it allows for a measure of similarity. This measure of similarity can be seen as a sort of "approximation" to the largest common subgraph problem. I put approximation in quotes as the algorithm does not guarantee the solution to any degree of certainty. If the size of the largest common subgraph found by the gradient descent algorithm is some "large" percentage of the UMDGs in question then the associated solid models are considered to be similar in structure based on features and feature interactions.

Another approach to the comparison problem that has been described is a fast subgraph isomorphism algorithm named ASIC that uses the A* search technique. This algorithm has proven to be fast in comparisons of UMDGs, but has a drawback: although it is guaranteed to find an isomorphism or a subgraph isomorphism if it exists, if an isomorphism or a subgraph isomorphism does not exist it does not allow for a measure of how "close" the graphs are to being isomorphic.

The ASIC algorithm is not limited to the CAD model comparison problem and can be used for comparing other types of graphs as well, other than the UMDG, for isomorphism and subgraph isomorphism. ASIC has proven to be far faster than Ullmann's subgraph isomorphism algorithm [45] when comparing graphs of the same size for isomorphism. For graph isomorphism testing, the relaxed initialization variant of the ASIC algorithm has proven to be far faster than ASIC itself suggesting that the true power of the ASIC

algorithm lies in the A* search technique. But as the size of the target graphs diverge, the added initialization of ASIC results in faster subgraph isomorphism comparisons. These results suggest the use of ASIC with relaxed initialization if the graphs are close in size and the use of ASIC with full initialization if the graphs vary greatly in size.

The data structure called a *Model Dependency Graph* and its variation, the *Undirected Model Dependency Graph*, along with the subgraph isomorphism algorithms described, can be used to manage knowledge-bases of CAD and Solid Modeling data. These data structures and algorithms can be used as the basis for a search and retrieval mechanism for a CAD knowledge-base. These algorithms may also prove useful in the development of case-based design and variant design systems as well as case-based manufacturing systems. It is my belief that this representation scheme and these comparison algorithms will have an impact on the way CAD models are retrieved.

Based on the UMDG and the gradient descent algorithm, it has been shown that one can create query artifacts that partition the database of solid models into different classes—based on how similar in structure each model is to the query model. It is believed that this approach can be refined to detect meaningful part classes and families in large sets of engineering models. This can form the basis for more intelligent Product Data Management (PDM) systems and tools for variational design and variant process planning.

5.2 Limitations

Non-unique Design Histories. The MDG and UMDG deal with the non-uniqueness of design histories and of the CSG representation of solid models to some degree. But it does not handle all cases of ambiguity. Any given CAD system may have its own set of design features and operations. There is no standard set of design features and design operations across all CAD systems. Individual designers may also design the same artifact in differ-

ent ways using different features even given the same CAD system. The representations described will circumvent the problem of non-uniqueness given different orderings of the same set of features or operations that produce the same artifact but is limited to this.

Consistent Feature Set. Regardless of whether design features or manufacturing features are used, it is a requirement that this is a consistent set of features across the database of solid models. If one model is represented using the feature set of one feature recognition system and a second model from the feature set of another feature recognition system then it would be essentially meaningless to compare the resulting UMDGs. The same is true with design features from CAD systems. The models under comparison must be represented using a consistent set of design features.

Strongly Regular Graphs. The UMDG does not tend to be strongly regular. So for the comparison of solid models this is not a limitation. However, it was noted that the ASIC algorithm could potentially be used for subgraph isomorphism tests of other forms of graphs. One class of graphs for which ASIC should not be used is that of strongly regular graphs. For these graphs, as with Ullmann's algorithm, the worst case exponential time complexity will be reached. In addition, the worst case exponential space complexity will be encountered as well.

Space Complexity. Although the worst case exponential space complexity of ASIC does not seem to appear in practice, constrained space may still potentially cause a problem. One possible solution may be to investigate the use of *Iterative Deepening A** (IDA*).

Scalability. The empirical results show that the ASIC algorithm is practical for graphs of up to 200 nodes or so. Also, graphs of 1000 nodes have been tested for isomorphism with the ASIC algorithm in under an hour on a Sun UltraSPARC 30. Relaxed initialization

saved some time in this case and took roughly 15 to 20 minutes. These 1000 node graphs were unattributed. With the addition of node attributes this time can be reduced. But 20 minutes for one test case is too long in terms of retrieving from a database if several of these comparisons must be performed. Perhaps simplifying the UMDG representation of a large complex model by combining multiple features into a single node may be a solution. For example, combining a group of holes in the model into a single node of the UMDG.

5.3 Future Work

Database Problem. In developing a CAD knowledge-base, it will be necessary to develop techniques to reduce the number of CAD models to examine in performing the search. Knowledge-bases of CAD data can potentially be gigabytes in size. It would be infeasible to test a query UMDG against every UMDG in the knowledge-base for subgraph isomorphism. Some possible solutions to this future problem may include using the depth-pruned decision-tree technique of [30] as an index into the knowledge-base. Although this technique generates a decision tree of exponential size, if the tree is pruned to some depth then it may be of tractable size and may prove useful as an index. It can also provide a starting point for ASIC more finer than obtained through its initialization procedure.

Another possible direction to pursue is the use of the determinant as an index. The determinant of the adjacency matrix of a graph is equal to that of any graph to which it is isomorphic [18]. This is a necessary but not a sufficient condition for isomorphism. It may prove a useful indexing technique for the UMDGs of a CAD knowledge-base.

Other possible directions to pursue with regards to the database problem include pre-computing the n-Region density of the UMDGs and storing this information in the database. This would greatly enhance the performance of the ASIC algorithm considering its biggest bottle-neck appears to be the initialization stage.

Machining Features. All experiments involving the MDG and UMDG presented in this work have been performed using design features. Future explorations will include the use of manufacturing features obtained from the use of a feature recognition system (such as FBMach from Allied Signal Inc. [7, 16]). It may also prove desirable to index the models of a CAD knowledge-base using both design features and machining features for alternative views of the data.

Node Attributes. It will be desirable to make use of more attributes on the nodes such as position, dimensions, orientation, tolerances, and materials of the feature they represent. The experiments presented in this work made use of the type of feature such as hole, slot, pocket, and so forth but did not consider any of these other possible attributes. Using more attributes on the nodes will reduce the search space of the problem resulting in faster comparisons and will also require that two models be more "similar" in structure in order for a match to occur.

Other CAD Data. The experiments presented and described in this work incorporated mechanical engineering models. But the approach is not limited to solid models of this type. Similar experiments can be performed on other forms of CAD data such as civil engineering data like bridges, buildings, roads and so forth.

Assemblies. The techniques described are not limited to solid models of individual parts. The ASIC algorithm for subgraph isomorphism as well as the gradient descent algorithm for inexact comparisons can be applied to assembly contact graphs for electro-mechanical assemblies. In this way, knowledge-bases of electro-mechanical assemblies may be developed with intelligent retrieval systems. An assembly planning system can also incorporate these algorithms for the retrieval of assembly plans for similar assemblies.

National Design Repository. It will also be desirable to perform larger-scale experiments on large knowledge-bases of real designs such as those contained in the National Design Repository (http://repos.mcs.drexel.edu), whereas the experiments presented have been performed on pseudo-randomly generated variations of a test part using design features. The UMDGs of the parts of the National Design Repository can be generated using machining features extracted using feature recognition techniques. The models may then be indexed based on these UMDGs. A World Wide Web based search engine incorporating the ASIC and gradient descent algorithms can then be developed to search the repository. This could enable researchers from around the world to search the National Design Repository in a more efficient manner.

Bibliography

Bibliography

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[2] H. A. Almohamad and S. O. Duffuaa. A linear programming approach for the weighted graph matching problem. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(5):522–525, May 1993.

[3] W. A. Andersen, J. A. Hendler, M. P. Evett, and B. P. Kettler. Massively parallel matching of knowledge structures. In H. Kitano and J. Hendler, editors, *Massively Parallel Artificial Intelligence*, pages 52–73. AAAI Press/The MIT Press, Menlo Park, California, 1994.

[4] L. Babai. Moderately exponential bound for graph isomorphism. In *Proceedings of the International Conference on Fundamentals of Computation Theory*, number 117 in Lecture Notes in Computer Science, pages 34–50. Springer-Verlag, 1981.

[5] A. T. Berztiss. A backtrack procedure for isomorphism of directed graphs. *Journal of the Association of Computing Machinery*, 20(3):365–377, July 1973.

[6] W. F. Bronsvoort and F. W. Jansen. Feature modelling and conversion - Key concepts to concurrent engineering. *Computers in Industry*, 21:61–86, 1993.

[7] S. L. Brooks and R. Bryan Greenway Jr. Using STEP to integrate design features with manufacturing features. In A. A. Busnaina, editor, *ASME Computers in Engineering Conference*, pages 579–586, New York, NY 10017, September 17-20, Boston, MA 1995. ASME.

[8] J. K. Cheng and T. S. Huang. A subgraph isomorphism algorithm using resolution. *Pattern Recognition*, 13(5):371–379, 1981.

[9] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. McGraw-Hill, 1990.

[10] D. G. Corneil and C. C. Gotlieb. An efficient algorithm for graph isomorphism. *Journal of the Association of Computing Machinery*, 17(1):51–64, Jan 1970.

[11] T. De Martino, B. Falcidieno, F. Giannini, S. Hassinger, and J. Ovtcharova. Feature-based modelling by integrating design and recognition approaches. *Computer Aided Design*, 26(8):3–13, August 1993.

[12] Alexei Elinson, Dana S. Nau, and William C. Regli. Feature-based similarity assessment of solid models. In Christoph Hoffman and Wim Bronsvoort, editors, *Fourth Symposium on Solid Modeling and Applications*, pages 297–310, New York, NY, USA, May 14-16 1995. ACM, ACM Press. Atlanta, GA.

[13] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, 1979.

[14] R. Geelink, O. W. Salomons, F. van Slooten, and F. J. A. M. van Houten. Unified feature definition for feature-based design and feature-based modeling. In A. A. Busnaina, editor, *ASME Computers in Engineering Conference*, pages 517–534, New York, NY 10017, September 17-20, Boston, MA 1995. ASME.

[15] D. E. Ghahraman, A. K. C. Wong, and T. Au. Graph monomorphism algorithms. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-10(4):189–196, April 1980.

[16] J. Han, W. C. Regli, and S. Brooks. Hint-based feature recognition. In *ASME Computers in Engineering Conference*, New York, New York, September 14-17, Sacremento, CA. 1997. ASME.

[17] J. Han and A. A. G. Requicha. Integration of feature-based design and feature recognition. In A. A. Busnaina, editor, *ASME Computers in Engineering Conference*, pages 569–578, New York, NY 10017, September 17-20, Boston, MA 1995. ASME.

[18] F. Harary. The determinant of the adjacency matrix of a graph. *SIAM Review*, 4(3):202–210, July 1962.

[19] F. Harary. *Graph Theory*. Addison Wesley, 1969.

[20] C. M. Hoffmann. *Geometric and Solid Modeling: An Introduction*. Morgan Kaufmann Publishers, Inc., California, USA, 1989.

[21] J. E. Hopcroft and R. E. Tarjan. Isomorphism of planar graphs (working paper). In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 131–152. Plenum Press, New York, 1972.

[22] J. E. Hopcroft and J. K. Wong. Linear time algorithm for isomorphism of planar graphs. In *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing*, pages 172–184, 1974.

[23] Q. Ji and M. M. Marefat. Machine interpretation of CAD data for manufacturing applications. *Computing Surveys*, 29(3):264–311, September 1997.

[24] S. Joshi and T. C. Chang. Graph-based heuristics for recognition of machined features from a 3D solid model. *Computer-Aided Design*, 20(2):58–66, March 1988.

[25] B. Kolman. *Introductory Linear Algebra*. Prentice Hall, fifth edition, 1993.

[26] E. M. Luks. Isomorphism of graphs of bounded valence can be tested in polynomial time. *Journal of Computer and System Sciences*, 25:42–65, 1982.

[27] M. Marefat and R. L. Kashyap. Geometric reasoning for recognition of three-dimensional object features. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(10):949–965, October 1990.

[28] M. Marefat and R. L. Kashyap. Automatic construction of process plans from solid model representations. *IEEE Transactions on Systems, Man, and Cybernetics*, 22(5):1097–1115, September/October 1992.

[29] B. T. Messmer and H. Bunke. A network based approach to exact and inexact graph matching. Technischer Bericht IAM 93-021, Institut für Informatik, Universität Bern, Schweiz, September 1993.

[30] B.T. Messmer and H. Bunke. Subgraph isomorphism in polynomial time. Technischer Bericht IAM 95-003, Institut für Informatik, Universität Bern, Schweiz, 1995.

[31] B.T. Messmer and H. Bunke. Fast error-correcting graph isomorphism based on model precompilation. Technischer Bericht IAM-96-012, Institut für Informatik, Universität Bern, Schweiz, 1996.

[32] G. L. Miller. Isomorphism of graphs which are pairwise k-separable. *Information and Control*, 56:21–33, 1983.

[33] G. L. Miller. Isomorphism of k-contractible graphs. a generalization of bounded valence and bounded genus. *Information and Control*, 56:1–20, 1983.

[34] W. C. Regli, S. K. Gupta, and D. S. Nau. Extracting alternative machining features: An algorithmic approach. *Research in Engineering Design*, 7(3):173–192, 1995.

[35] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, Upper Saddle River, New Jersey 07458, 1995.

[36] O. W. Salomons, F. J. A. M. van Houten, and H. J. J. Kals. Review of research in feature-based design. *Journal of Manufacturing Systems*, 12(2):113–132, 1993.

[37] K. E. Sanders, B. P. Kettler, and J. A. Hendler. The case for graph-structured representations. In *Proceedings of the Second International Conference on Case-based Reasoning (ICCBR)*, Berlin-Heidelberg-New York, 1997. Springer-Verlag.

[38] D. C. Schmidt and L. E. Druffel. A fast backtracking algorithm to test directed graphs for isomorphism using distance matrices. *Journal of the Association of Computing Machinery*, 23(3):433–445, July 1976.

[39] Y. J. Shah, G. I. Davida, and M. K. McCarthy. Optimum features and graph isomorphism. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-4(3):313–319, May 1974.

[40] V. Shapiro and D. L. Vossler. Construction and optimization of CSG representations. *International Journal of Computer Aided Design*, 23(1):1–20, January/February 1991.

[41] V. Shapiro and D. L. Vossler. Separation for boundary to CSG conversion. *ACM Transactions on Graphics*, 12(1):35–55, January 1993.

[42] D. Spielman. Faster isomorphism testing of strongly regular graphs. In *Proceedings of the 28th Annual ACM Symposium on Theory of Computing*, pages 576–584, 1996.

[43] C. Thornton and B. du Boulay. *Artificial Intelligence: Strategies, Applications, and Models Through Search*. Amacom, New York, New York, 1998.

[44] J. Turner. Generalized matrix functions and the graph isomorphism problem. *SIAM Journal of Applied Mathematics*, 16(3):520–526, May 1968.

[45] J. R. Ullmann. An algorithm for subgraph isomorphism. *Journal of the Association of Computing Machinery*, 23(1):31–42, Jan 1976.

[46] S. H. Unger. GIT - a heuristic program for testing pairs of directed line graphs for isomorphism. *Communications of the ACM*, 7(1):26–34, Jan 1964.

[47] J. H. Vandenbrande and A. A. G. Requicha. Spatial reasoning for the automatic recognition of machinable features in solid models. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(12):1269–1285, December 1993.

[48] A. K. C. Wong, M. You, and S. C. Chan. An algorithm for graph optimal monomorphism. *IEEE Transactions on Systems, Man, and Cybernetics*, 20(3):628–636, 1990.

[49] C. Yang. Structural preserving morphisms of finite automata and an application to graph isomorphism. *IEEE Transactions on Computers*, 24(11):1133–1139, Nov 1975.