Survey of Graph Matching Algorithms

Vincent A. Cicirello

Technical Report
Geometric and Intelligent Computing Laboratory
Drexel University

March 19, 1999

1 Introduction

Graph matching problems of varying types are important in a wide array of application areas. A graph matching problem is a problem involving some form of comparison between graphs. 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. Although it is possible to formulate such a large array of problems as graph matching problems, it is not necessarily a good idea to do so.

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 [10]. Whether or not the *graph isomorphism* problem is in the class of NP-complete problems is an open question [10]. 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 general worst case complexity for the graph isomorphism problem is $2^{\bigcirc(n*\log n)}$ [4, 26]. One class of graphs that poses particular problems for graph isomorphism algorithms is the class of strongly regular graphs. The worst case complexity for strongly regular graph isomorphism is $n^{\bigcirc(n^{\frac{1}{3}}*\log n)}$ [26]. One special case that is solvable in polynomial time is planar graph isomorphism. An $\bigcirc(n*\log n)$ algorithm for planar graph isomorphism may be found in [14] and a linear time solution in [15]. There are other special cases that are solvable in polynomial time that involve a bound on some property of the nodes 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. A few examples of such cases include graphs of bounded valence [17], k-contractible graphs [22], and graphs that are pairwise k-separable [21]

In section 2, I list some necessary definitions and background. Section 3 includes a list of graph invariants and a survey of algorithms for graph and subgraph isomorphism.

2 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 [10]: "is there a one-to-one onto function $f : V_1 \to V_2$ 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 [10] 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 $f : V_2 \to V$ 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 [10].

Digraph D-morphism A problem that is closely related to these problems that is 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 [10] as a function $f : V_1 \to V_2$ 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* [8, 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 $\bigcirc(|V| + |E|)$ storage space.

Adjacency Matrix Another common representation of a graph is the *adjacency matrix* [13, 8, 1, 16]. 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 $\bigcirc(|V|^2)$ 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 [26] 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.

3 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 [27, 12]. 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 [29, 9, 28, 6, 11, 5, 24, 30]. 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 [28].

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 [31], Yang shows how a state machine may be generated from a graph and the resulting state machines then compared for isomorphism. In [25], 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 [19], and the graph decomposition approach described in [18].

3.1 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 [13]. No complete set of invariants for a graph 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.

3.1.1 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 [12]. 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 generalized characteristic polynomial of the adjacency matrix of a graph is equal to that of any graph to which it is isomorphic [27]. The characteristic polynomial of an adjacency matrix A is defined as $det(A - \lambda \cdot I)$. A proof similar to that above may be found in [27].

3.1.2 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. 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 [28]. If the graphs are directed this would include both the in-degree and the out-degree of the nodes [29].

- The number of *nth generation descendents* of the nodes [29]. 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 [29]. 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 [29].

- The number of nodes in the *n-shell of descendents* of the nodes [29, 24]. 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* of the nodes [29, 24]. 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 path shorter than length n. 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 [29].

- A function equal to 1 on nodes included in n-length *unrestricted circuits* and 0 otherwise [29]. Similarly, A function equal to 1 on nodes included in n-length *simple circuits* and 0 otherwise [29]. An unrestricted circuit may include an edge more than once whereas a simple circuit may not include any edge more than once.

3.2 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 [29], 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 set in the node partition a unique set 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 [28].

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 [28]). 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 algo $M^0 = [m^0_{ij}]$ 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 [6] 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 [24], 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 [24] describe a characteristic matrix. The row characteristic matrix XR is an N by $(N-1)$ matrix. xr_{im} is the number of vertices a distance m away from v_i. The column characteristic matrix XC is an N by $(N-1)$ 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.

3.3 Other Approaches

Canonical Adjacency Matrix In [25], 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 relabelled graph is of greatest possible magnitude.

Reduction to Isomorphism of Finite State Machines In [31], Chao-Chih 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 $\parallel A_g - P * A_h * P^T \parallel$. $\parallel A \parallel$ is the sum of all of the elements of the matrix A. This is also equivalent to minimizing $\parallel A_g * P - P * A_h \parallel$. They formulate a linear programming problem, solve the linear programming problem using the simplex method and then use the Hungarian method to 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 in polynomial time. The hungarian method is used for approximate 0-1 integer solutions because 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 [18], 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 [19], the authors attempt to solve the problem of given a database of model graphs known apriori 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 $\bigcirc(M^2)$ 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 $\bigcirc(M^3)$. 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 $\bigcirc(M^3)$ 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 Ullman'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 [20]. 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 [19]. They compute all of the error-correcting isomor-

phisms 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, 23]. 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.

Hill-climbing In [7], an approach to approximate graph and subgraph isomorphism is described using a hill-climbing or gradient descent approach. The application for this algorithm is for comparing the similarity of solid models. The authors develop a graph representation of the dependencies among the design features of the solid models. This representation is termed the *model dependency graph*. The comparison algorithm described first pairs up the nodes of the two graphs randomly. It then uses a gradient descent approach swapping node pairings as to minimize the number of edges that are inconsistently mapped. This algorithm is executed a fixed number of times with different randomly chosen starting points and the best result is chosen. This approach suffers from a lack of completeness. That is, if the graphs are isomorphic there is no guarantee that the isomorphism will be found.

References

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

[7] V. A. Cicirello and W. C. Regli. Resolving non-uniqueness in design feature histories. In *Proceedings of the 1999 ACM Conference on Solid Modeling and Applications*, June 1999.

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

[9] 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.

[10] 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.

[11] 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.

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

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

[14] 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.

[15] 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.

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

[17] 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.

[18] 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.

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

[20] 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.

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

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

[23] 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.

[24] 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.

[25] 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.

[26] 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.

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

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

[29] 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.

[30] 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.

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