Massively Parallel Matching of Knowledge Structures

William A. Andersen[*] James A. Hendler[†]
Matthew P. Evett[‡] Brian P. Kettler[§]

Department of Computer Science
University of Maryland
College Park, MD 20742

Abstract

As knowledge bases used for AI systems increase in size, access to relevant information is the dominant factor in the cost of inference. This is especially true for analogical (or case-based) reasoning, in which the ability of the system to perform inference is dependent on efficient and flexible access to a large base of exemplars (cases) judged likely to be relevant to solving a problem at hand.

In this chapter we discuss a novel algorithm for efficient associative matching of relational structures in large semantic networks. The structure matching algorithm uses massively parallel hardware to search memory for knowledge structures matching a given probe structure. The algorithm is built on top of PARKA, a massively parallel knowledge representation system which runs on the Connection Machine. We are currently exploring the utility of this algorithm in CaPER, a case-based planning system.

[*]Email: waander@cs.umd.edu
[†]Email: hendler@cs.umd.edu
[‡]Email: evett@cs.umd.edu
[§]Email: kettler@cs.umd.edu

1 Introduction

In case-based reasoning (CBR), previously encountered problem solving experiences (cases) are used as analogues in solving similar new problems. Most CBR systems retrieve one or more cases judged to be similar to the current problem situation, map these cases to the current situation, and adapt them to the current problem. When the adapted solution is executed (either in the world or by simulation), failures are noted, and the solution is repaired and stored for future use. To minimize the amount of adaptation required, CBR systems must support a large set of diverse cases and must provide an efficient means for accessing this casebase.

In this chapter we discuss a novel algorithm for case retrieval in CBR that uses parallel search for knowledge structures in a large case memory (casebase). Both the retrieval probe (query) and the stored cases are represented as graph structures in a semantic network. The algorithm relies on massively parallel hardware (CM-2) to quickly match knowledge structures in memory against a retrieval probe. Case retrieval techniques based on serial hardware have relied on the technique of "flattening" relational structures into indexed feature vectors for efficient matching. The structure matcher imposes no such restriction on the form of representation allowed. Cases to be efficiently retrieved from a large casebase that is not pre-indexed while maintaining a rich relational representation. This algorithm is currently being evaluated in CaPER [12], a case-based planning system designed to exploit massively parallel case retrieval techniques.

2 Motivation

Most CBR systems pre-index case memory. Pre-indexing is the designation, at case storage time, of features ("indices") through which a case may subsequently be retrieved. In serial case-based systems, it is too expensive to compare the current problem situation with every case stored in the casebase. In order to make case retrieval tractable, pre-indexing techniques identify a privileged set of features to be used to organize the cases in memory (e.g., in a discrimination network). In some systems, derived features (ultimately computed as a function of surface features) may be used as well. Pre-indexing can reduce case retrieval times to logarithmic in the number of cases.

Determining what kinds of indices are useful for case retrieval is known as "the indexing problem", a problem that is the subject of much CBR research (e.g. [14]). Many approaches to indexing have been proposed, ranging from the ad hoc to the use of machine learning techniques such as explanation-based learning/generalization (EBL/EBG) and induction (e.g. [8]).

Planning cases are often indexed by pragmatic features such as the goals of the case. The CHEF system [9] indexes recipes by their main ingredients. Planning cases are also frequently indexed by features of their initial situations including surface features (the actors, objects, etc.) and derived features, such as spatial or other kinds of relations among the actors, objects, etc. CHEF additionaly indexes cases by the failures they avoid, assuming such failures can be diagnosed. Cases that avoid known failures are preferred in future retrievals.

Other types of derived features typically used include causal and structural relations among the surface features and higher-order relations (i.e., relations involving other relations) (e.g. [7]). Story cases are often indexed by derived features such as the moral or theme, in addition to surface features such as the characters, setting, and props.

2.1 Some Shortcomings of Memory Pre-indexing

There are several problems with pre-indexing case memory. First, indexing hinders flexibility in case retrieval. The selection of privileged indices effectively "flattens" the representation of a case into a feature vector. This limits retrieval flexibility in two ways: a) Any features not chosen as indices simply cannot be used during retrieval. Cases that share unindexed features with the target problem will not be retrieved; b) Any relations between features cannot be examined by retrieval mechanisms, since they have been thrown away for the sake of efficiency by the indexing process. It is unlikely that the designer of an indexing scheme could anticipate and provide for all of the ways in which a case might usefully be retrieved in the future.

Second, indexing schemes are often domain- and task-specific and can thus limit the general-purpose utility of a case memory. A general-purpose organization of the knowledge base (KB) is desirable if it is to be used by multiple systems that may use case-based or other reasoning mechanisms. For example, a casebase of computer equipment repair calls would be useful for both a diagnostic CBR system for repairing failures and for a CBR system for product redesign. Memory organizations that rely on indexing lack the flexibility needed to support multi-purpose memory use. For example, a discrimination tree statically orders indexes by salience: indexes for more salient features are at higher levels of the tree.

Third, in many domains of interest for CBR, such as design, spatial reasoning, and perception, the relevance of a case is determined primarily by the structural properties of the case and not by abstract thematic properties. In this sense, a case may simply be a description of a structure, device or process, in which the structural and functional properties of the case are strongly related. It is unclear how an indexing scheme might be constructed for accessing cases in these domains.

Finally, the psychological plausibility of pre-indexing is questionable given evidence from psychological studies of human analogue retrieval (e.g., [7, 19, 21]). Human analogue retrieval is fast and highly associative, with many remindings being generated. Humans use a combination of semantic, structural, and pragmatic features of cases for retrieval, depending on the domain/task, and are capable of finding analogies both within a single domain or between domains ("cross-domain" or metaphorical analogies).

3 Structure matching for case retrieval

As an alternative to the use of indexed case memory, we propose the use of a structure matching algorithm for the problem of rapid retrieval of knowledge structures from a large case memory. While such an algorithm would not preclude the use of derived features for retrieval, we believe that equal emphasis for case retrieval in CBR should be placed on matching structural properties of stored cases to a retrieval probe.

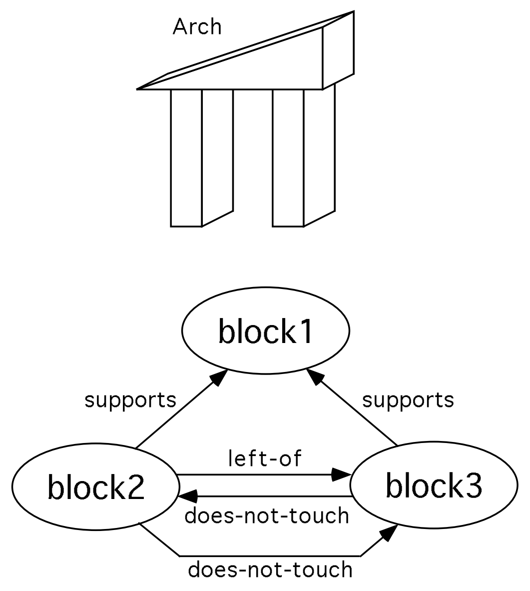

Figure 1: Description of an arch

Consider the task of classifying a target structure as an arch or non-arch (e.g., [23]) using its similarity to exemplars of arches (and/or dissimilarity to exemplars of "non-arches" stored in its casebase). With a pre-indexing approach, these stored exemplars would need indices of some sort through which they can efficiently be retrieved. If these exemplars were stored under high-level indices such as "arch" and "non-arch", then it would be necessary to know whether or not the target structure is an arch or not in order to retrieve similar exemplars. The archness of target structure, however, is what we are trying to determine;

we want to retrieve exemplars that have similar physical structures. Thus it is desirable to use the structural relations of the target as the retrieval probe to match against the structural relations of the stored exemplars, rather than matching some computed high-level index stored with each of them. Figure 1 shows an example arch and a structure in a semantic network representing the arch.

3.1 Representing structures for matching

Our description of the problem of structure matching follows that given in [22]. A knowledge base defines a set, P, of unary and binary predicates. Unary predicates have the form $P_i(x)$ and binary predicates have the form $P_j(x_1, x_2)$, where each x_i is a variable on the set of frames in the KB. An existential conjunctive expression on these predicates is a formula of the form $\exists x_1, \ldots, x_m : P_1 \wedge P_2 \wedge, \ldots, \wedge P_n$, where $n \geq 1$. Our task is to retrieve all structures from memory which match a given conjunctive expression. Therefore, we would like to find all such satisfying assignments for the x_i.

We can view the problem of matching knowledge structures in two ways. The first is as a subgraph isomorphism problem. We view variables as nodes and binary predicates as edges in a graph. We want to find structures in memory which "line up" with the graph structure of the query expression. The other way to view the matching problem is as a problem of unification or constraint satisfaction. If we can find a structure in memory which provides a consistent assignment to the variables x_i (i.e., unification), then that structure matches the conjunctive expression. For the remainder of this paper we will use the subgraph matching view with occasional reference to the constraint satisfaction view as appropriate.

In Figure 1, an arch is represented by a set of blocks and a set of relations which hold between the blocks. Specifically, the conjunction of following set of relations describes the arch:

supports(block2,block1)
supports(block3,block1)
left-of(block2,block1)
does-not-touch(block2,block3)
does-not-touch(block3,block2)

Seen this way, case memory can be 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 cases similar to a retrieval probe is reduced to a problem of structural matching, or of identifying subgraphs in the semantic network that are isomorphic to the probe.

4 Implementation of structure matching using PARKA

The structure matching algorithm is implemented on top of the PARKA knowledge representation (KR) system [3]. PARKA is a general-purpose knowledge representation system, intended for use by other AI systems. It uses a frame-based representation scheme, optimized to take advantage of the massive parallelism of the Connection Machine. PARKA can perform fast top-down inheritance inferences (in the tenths of seconds) on extremely large KBs (10^5 to 10^7 frames).

The structure matching algorithm exploits PARKA's ability to perform extremely fast top-down inheritance. Top-down inheritance inferences are of the form "What is the color of every truck?" or "Which trucks are yellow and blue?". PARKA computes these kinds of inferences in time proportional to the depth of the semantic network and independent of its size in $O(d)$ time. Via the use of pipelining techniques, PARKA can perform multiple top-down inheritance queries in $O(k + d)$ time, for a set of k top-down queries[1]. In [4], we present timing results of pipelined inheritance queries on a PARKA encoding[2] of a subset of the Cyc commonsense knowledge base [15]. For this encoding of the Cyc KB, consisting of over $30,000$ concepts and $120,000$ property assertions, PARKA was able to process queries with 22 conjuncts in approximately 900 milliseconds.

We built PARKA with run-time performance rather than expressiveness as the primary goal of the system, including as much representational capability as would still allow computationally effective inferencing. Restricting expressiveness to achieve tractability is an undesirable but necessary practice that has become fairly common in the KR community. Although PARKA's semantics are roughly based upon those of NETL [5] and KL-ONE [2], we avoided semantic constructs lacking a computationally effective implementation. PARKA's run-time performance on even very large KBs more than compensates for its restricted expressiveness when compared to that of other, serial, KR systems.

Both PARKA and the structure matching algorithm were implemented in *Lisp (pronounced "star lisp") on a Thinking Machines Connection Machine (CM-2) [11]. The CM-2 is a massively parallel supercomputer with up to 65,536 1-bit processing elements (PEs). The PEs are connected by a general interconnection network (twisted hypercube) and a "grid" communication network, providing communication among immediately neighboring processors. Each PE may have up to 1M bits of bit-addressable memory. The CM-2 has a SIMD architecture that supports a data-parallel computation model, in which a serial

[1] A naive implentation would require $O(kd)$ time. The use of pipelining allows processing of queries in various stages of completion to be overlapped, increasing processor utilization.

[2] For these tests, we extracted the inheritance hierarchy (*isa* relations) from the Cyc KB. In addition, we included only those property assertions which were directly asserted or present as the result of forward propagation.

front-end processor broadcasts a stream of instructions, each of which is executed simultaneously by all PEs. Control flow is achieved by selecting and deselecting PEs. Selected PEs perform broadcast instructions, while deselected PEs sit idle. *Lisp is an implementation of Common Lisp for the CM-2 augmented to provide support for data-parallel programming. It provides support for interprocessor communication operations (broadcast, permutation and parallel-prefix), as well as parallel intraprocessor operations.

5 The structure matching algorithm

The structure matching algorithm operates by comparing a retrieval probe, P, against a knowledge base (KB) to find all structures in the KB which are consistent with P. This match process occurs in parallel across the entire knowledge base. A PARKA KB consists of a set of frames and a set of relations on those frames. Because PARKA supports property inheritance most relations are only implicitly specified. By computing inherited values for a relation, all pairs defining the relation are made explicit. We currently allow only unary and binary relations[3].

A retrieval probe is specified as a graph consisting of set of variables $V(P)$ and a set of predicates (or constraints) $C(P)$ that must simultaneously hold on frames bound to those variables. The result of the algorithm is a set of k-tuples, $Result$, where each k-tuple encodes a unique $1-1$ mapping of frames to variables in $V(P)$, that unifies the description of the structure in memory with $C(P)$. Figure 2 shows a simple example of the structure matching algorithm at work; given a knowledge base represented as a semantic network and a probe graph, the algorithm finds all matching subgraphs in the network which are isomorphic to the probe.

The set of concepts that can bind to each variable is initially restricted by a set of constraints indicated by unary predicates. Each unary constraint may only constrain the values of one variable. Examples of these constraints are "X is a dog" or "the color of X is yellow". We allow set theoretic combinations of the unary constraints, for example "X is a dog and the color of X is yellow", or "X is a dog but X is not yellow".

Constraints between frames bound to variables are specified by a set of binary constraints. For example, we can say "the color of X must be Y", or "X is a part of Y", for some X and Y in $V(P)$. Binary constraints are processed by "expanding" the binary relation given in the constraint. By expansion we mean that all pairs participating in a relation R in the KB are made explicit by computing the inherited property values for each frame for the relation R. The pairs allowed to participate in the expanded relation are restricted to those in the domains of the variables related by R. For example, a binary constraint

[3]PARKA currently supports only unary and binary relations ("slots"). Future versions of PARKA will support higher-order relations.

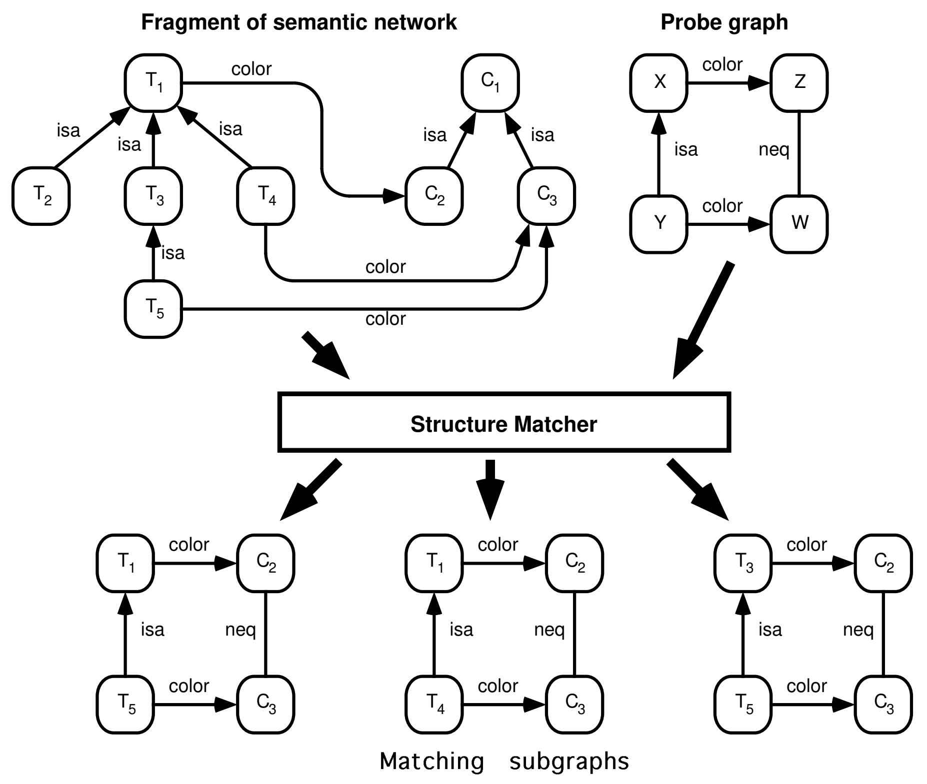

Figure 2: A simple example of structure matching.

may be expressed as $color(X, Y)$. In this case the values for each concept in the domain of X are computed for the *color* relation. Pairs that have values outside the domain of Y are excluded.

An additional binary constraint, *neq*, is provided to enforce an inequality constraint on frames bound to two variables appearing in $V(P)$. This constraint acts as a filter, eliminating any tuples from *Result* for which the constrained variables are bound to the same frame[4].

[4] Other binary predicates could be supported as well, such as inequality constraints (e.g. X is greater than Y). In the current implementation, however, we do not provide support for these constraints.

5.1 Initialization

The structure matching algorithm is shown in Figure 3. In step 1 the algorithm has an initially empty set of result tuples (i.e. $Result = \emptyset$). For example, for a probe with variables $\{X, Y, Z, W\}$, the result relation will be a set of 4-tuples, with positions in each 4-tuple indexed by a variable in $V(P)$. We will denote each position in the result as $Result(v)$, where v is a variable in $V(P)$. Each variable in $V(P)$ has an associated domain – the set of legal values to which the variable may be bound. These domain sets are denoted by $Domain$, an array of sets where for each variable v, $Domain(v)$ represents the set of legal domain values. In step 2, $Domain(v)$ is initialized to the empty set for each variable, v.

In Figure 3, $Var(C)$ denotes the set of variables appearing in a constraint C, $UnaryConstraints(P)$ the set of unary constraints in P, and $BinaryConstraints(P)$ the set of binary constraints in P.

1. $Result \leftarrow \emptyset$
2. $\forall_{v \in V(P)} : Domain(v) \leftarrow \emptyset$

3. do for c in $UnaryConstraints(P)$
4. $\quad Domain(Var(c)) \leftarrow Domain(Var(c)) \cup ComputeUnaryConstraint(c)$

    end do

5. $B \leftarrow BinaryConstraints(P)$
6. do while $B \neq \emptyset$
7. $\quad C \leftarrow ChooseBinaryConstraint(B)$
8. $\quad R \leftarrow ComputeBinaryConstraint(C, Domain)$
9. $\quad$ if $Result = \emptyset$ then
10. $\qquad Result \leftarrow R$

    $\quad$ else

11. $\qquad Result \leftarrow Result \bowtie_{Var(Result)=Var(R)} R$
12. $\quad B \leftarrow B - \{C\}$
13. $\quad \forall_{v \in V(P)} : Domain(v) \leftarrow \bigcup_{\tau \in R} \{v(\tau)\}$

    end while

Figure 3: The structure matching algorithm

5.2 Processing unary constraints

The algorithm processes all unary constraints (steps 3 and 4) before processing binary constraints. Each unary constraint is computed using the property inheritance algorithm of PARKA. The result of processing each constraint is placed into a set of legal domain values for the constrained variable. If a constraint C constrains variable X, then $Domain(X)$ will contain the result of processing C.

The algorithm computes two different kinds of unary constraints. The first involves computing concepts reachable via some transitive relation. For example, we might constrain X to be in the set of dogs by saying $isa(X, Dog)$. These types of constraints are computed in time proportional to the topological depth of the relation mentioned in the constraint. The second type of unary constraint involves the computation of top-down inheritance (e.g., $color(X, Yellow)$). As mentioned above, unary constraints can be combined by set-theoretic operations. If a variable v is mentioned in two or more separate unary constraints, then $Domain(v)$ is taken to be the union of the results of processing the individual constraints.

5.3 Processing binary constraints

Most of the work of the structure matching algorithm is done in processing binary constraints. We must first choose a constraint from the remaining constraints, expand it using PARKA's inheritance algorithm, merge it into the accumulated result, and then propagate constraints to the domains for the probe variables.

5.3.1 Choosing a constraint to process

In step 10, we use a relational join operation to merge binary constraints as they are processed. Thus, the order in which the constraints are processed is important for minimizing the size of intermediate join results. If two relations, R and S, are to be joined, the size of $R \bowtie S$ is $O(|R||S|)$. Several heuristic techniques are possible to minimize the size of intermediate join results as each binary constraint is processed. We would like a heuristic which minimizes $|R \bowtie S|$ for each join performed. Currently we employ a greedy algorithm which selects from the remaining binary constraints the one with the corresponding relation likely to be of least cardinality. The constraints are chosen in decreasing order: inequality constraints, functional relations, non-functional relations, and transitive relations.

This choice is indicated by the call to $ChooseBinaryConstraint$ in step 7. Note that we must choose a constraint from B with a relation which may be joined with $Result$ (i.e., one which shares at least one variable with the variables already bound in $Result$). Inequality constraints are most strongly preferred. Because they require only filtering of $Result$ and no expansion of a relation, they can be performed in $O(|Result|/p)$ time. Constraints with functional relations are the next best, since such relations may only have a cardinality equal to $|Domain(V)|$, where V is the variable on which the relation computed for the new constraint will be joined with $Result$, and thus cannot increase the cardinality of $Result$ when joined. Non-functional relations are considered next. These relations (e.g. *isa*) have a cardinality of $O(|C(KB)|^2)$ (i.e. each concept may be related to any other), but typically are quite a bit smaller since each relation is only relevant to a small subset of the concepts in the KB (see 6). Finally, transitive relations are considered. For example, these typically constrain X to be an *isa* ancestor of Y, which can only be determined by transitive closure on the *isa* relation. Since every concept in the KB participates in the *isa* relation, it is important that this relation be processed as late as possible, so that the domains of the constrained variables may be made as small as possible.

5.3.2 Expanding a binary relation

As mentioned above, since PARKA supports property inheritance, most binary relations are only implicitly defined. They must be expanded before they can be merged with $Result$. This is done in step 8 of Figure 3 by the function $ComputeBinaryConstraint$.

There are two different kinds of binary constraints. The first kind is computed using the property inheritance facility of PARKA. An example of such a constraint would be $color(X, Y)$. Here we constrain any bindings for X and Y to be bound to concepts which are related by the relation *color* in the KB and which are elements of $Domain(X)$ and $Domain(Y)$, respectively. Here the domain membership constraint may be applied before or after the property inheritance calculation. There is no advantage to doing things either way.

The other type of binary constraint involves constraints on transitive relations, such as *isa* or $partof$, these are computed using a special algorithm which returns the transitive closure of the relation mentioned in the constraint. Since the size of these types relations is likely to be large, it is important that the domain constraints on the variables be applied before performing the closure operation. $ComputeBinaryConstraint$ simply examines the type of constraint and calls the proper PARKA function to expand the associated relation.

5.3.3 Merging an expanded relation

Initially, $Result = \emptyset$, so the algorithm simply places the first binary constraint processed into $Result$ (step 10 of Figure 3). For example, if $R(X, Y)$ is the first relation processed, the pairs in $R(X, Y)$ are placed into $Result(X)$ and $Result(Y)$.

As each subsequent binary relation R is processed in step 11, its values are merged into $Result$ by one of two parallel relational join algorithms. For example, $Result$ has bindings for variables X and Y, and a new binary relation $R(Y, Z)$, constraining variables Y and Z is to be merged. After the join $Result$ will contain tuples with bindings for variables X, Y, and Z, produced by joining $R(Y, Z)$ with $Result$ on the keys indexed by variable Y. We use two different join algorithms for efficiency's sake. When the size of the cartesian product of the relations to be joined is small, we use a brute force algorithm which enumerates the cartesian product and applies a selection operation. This algorithm runs in time $\Theta(|R||S|/p)$ where R and S are the two relations being joined, and p is the number of physical processors. For relations with larger cross products, we use a parallel merge join algorithm. The run-time for this algorithm is $O(|R||S|)$ in the worst case, but in practice it is much better since only keys appearing in both relations are considered.

5.3.4 Propagation of constraints

After each binary relation is processed and joined with $Result$, a constraint propagation step (step 13) is performed to update the domains of the variables appearing in $Result$ so far. For example, suppose variable X is initially restricted to the set of all dogs, and Y the set of all colors. After $color(X, Y)$ is processed, $Domain(X)$ should be updated to include only those dogs which have some color, and $Domain(Y)$ to only those colors which are had by some dog. We perform the propagation step after joining each relation with $Result$. As mentioned earlier, we try to defer the processing of non-functional and transitive relations as long as possible. By restricting the domains of the variables in $Result$ as processing proceeds, the cardinalities of any non-functional or transitive relations encountered later in processing are minimized. This step also eliminates needless work in the join processing by eliminating domain values which cannot possibly participate in the final result (because they violate one of the binary constraints).

5.4 The result

After all binary constraints in $C(P)$ have been processed, $Result$ contains a (possibly empty) set of k-tuples. Each k-tuple encodes variable bindings for concepts which simultaneously satisfy all unary and binary constraints. $Result$ is maintained in a parallel data structure (a *Lisp *pvar*, or parallel variable) which can either be used for further parallel computation or sent to the serial front end (see Section 4) for subsequent processing.

6 Complexity of the algorithm

The time required by the structure matching algorithm is the sum of the time to process the unary constraints and the time to process the binary constraints.

As discussed in [4], the computation of individual unary constraints can be performed by PARKA in $O(d)$ time. Via pipelining techniques used in PARKA, multiple unary constraints can be processed in $O(k + d)$ time, for a set of k unary constraints. Thus, the computation of the unary constraints requires $O(|UnaryConstraints(P)| + d)$ time.

The complexity of computing the binary constraints is more difficult to characterize. In the worst case, the work performed by this operation is exponential in the number of binary constraints since each individual join requires $O(|R||S|)$ key comparisons for joining relations R and S and we perform $|BinaryConstraints(P)| - 1$ such joins. However, in actuality, the amount of work done at each step is far less than $|R||S|$. The reason for this is twofold.

First, the size of the result of an individual join (for join on variable V) is equal to

$$\sum_{k_i \in V(R)} |r_j \in R : V(r_j) = k_i||s_k \in S : V(s_k) = k_i|$$

Thus, the size of the join result is a function of the "overlap" between the two relations on the join keys (i.e. the intersection of the sets of join keys). Since many relations in actual KBs are functional, we can expect the size of individual join results to be no larger than the size of the larger of the relations being joined, and for the most part much less[5].

Second, the size of the accumulated partial solution, $|Result|$, after processing each binary relation is a function of the domain constraints which have been propagated and reflected in $Result$ at each step. Thus, the amount of work done at each step (by joining relation S with $Result$ on variable V) is given by the recurrence relation:

$$|Result_i| = \sum_{k_i \in V(S)} |s_j \in S : V(s_j) = k_i||r_k \in Result_{i-1} : V(r_k) = k_i|$$

Assume we are joining a new relation S with $Result$ on variable X. Denote the probability that an element of $Result(X)$ will appear in $S(X)$ as $P_{overlap(S,Result)}$, and the average branching factor for frames bound to X on relation S as B_S. Then the recurrence has the form:

$$|Result_i| = P_{overlap(S,Result)} B_S |S||Result_{i-1}|$$

Here, we can see that the size of intermediate results is sensitive to size of the set of keys which "overlap" in $Result_{i-1}$ and S, as well as the branching factor (B_S) of S. In actual KBs, we believe that the amount of overlap will be small. For example, the relations *hair-color* and *profession* may be large, but the number of frames (presumably representing people) that participate in both relations is likely to be very small.

While pathological cases could easily be contrived which exhibit worst-case complexity, we believe that on KBs of realistic size and topology the product

[5]Empirical data collected from a PARKA encoding of the Cyc commonsense knowledge base [15] indicates that approximately 1/3 of relations in Cyc are functional, and that a concept is equally likely to participate in both functional ($P = 0.045$) and non-functional ($P = 0.049$) relations. In addition, the majority of the non-functional relations in the PARKA encoding of the Cyc KB had small cardinality.

$P_{overlap(S,Result)}B_S|S|$ will be less than 1 in the limit as more constraints are processed, thus making the algorithm linear in the number of binary constraints. This belief seems to be supported by the empirical data collected both from our PARKA encoding of the Cyc KB and from experimental runs of the structure matcher on test KBs (see Figure 6.) We plan to do further empirical analysis both using the Cyc KB and with the CaPER casebases to determine if actual performance meets this theoretical prediction.

7 Use of structure matching in CaPER

The CaPER case-based planner uses the structure matching algorithm to retrieve cases from a large unindexed memory. We believe the avoidance of memory pre-indexing gives CaPER advantages in flexibility over traditional case-based systems, in which it must be determined presciently how a case is to be retrieved at storage time, rather than at retrieval time.

In CaPER, no special indexing structures are used to organize memory. All cases are matched against the probe simultaneously through their surface and structural features using the structure matching algorithm. The ability to issue highly-specific queries multiple times at low cost allows CaPER to retrieve pieces of many plans. Figure 4 shows an example plan structure taken from an automobile construction planning domain being used to test the CaPER prototype.

7.1 Example query from CaPER

Below we provide an example query from a transportation logistics domain being used to test CaPER which shows CaPER's ability to retrieve plans from memory on the basis of structural similarity. In this domain, the initial configuration of the packages with respect to each other is useful in retrieving cases from the casebase that are applicable to the current problem. In the following examples, constraints are specified as lists of the form `(predicate arg_1 arg_2)`. Predicates (or relations) are indicated as symbols. Arguments may be either variables (indicated by plain symbols) or concepts (indicated by a symbol preceded by the #! syntax).

In this query, CaPER is probing memory for plans in which two packages were delivered to the same location. The first seven constraints, of the form `(isa X C)`, where `X` is a variable, and `C` a concept, are unary constraints. They constrain the domains over which the associated variables may range. For example, the constraint `(isa pkg1 #!package)` constrains any concept binding to variable `pkg1` to be a member of the set of packages in memory. The constraint `(object goal1 pkg1)` constrains any concept bound to the variable `goal1` to have an object which is a concept bound to `pkg1`.

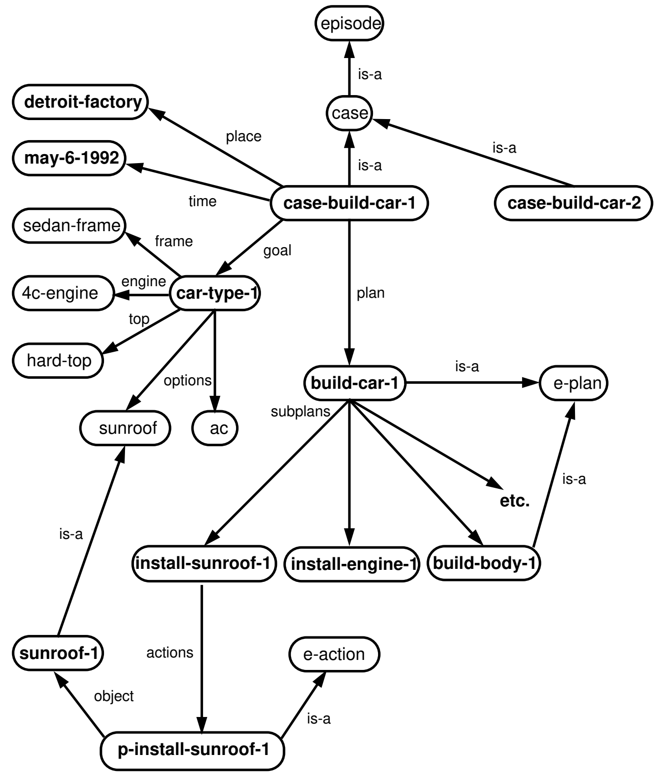

Figure 4: A case from CaPER's car construction domain

8 Empirical results

To test the efficiency and scalability of using structure matching for case retrieval in CaPER, several sample queries in transport logistics domain[6] were processed using the structure matcher. The purpose of these experiments was to determine the absolute time for retrieval probes and to determine the scalability of parallel case retrieval methods to large casebases.

The sample queries, chosen to be representative of those issued by CaPER for case retrieval, were (in their English equivalents along with the number of conjuncts required to encode them):

QX: Q1: find all top-level plans involving 2 or more packages (14 conjuncts)Q2: find all plans using a particular airport (14 conjuncts)Q3: find all top-level plans using 2 or more trucks (14 conjuncts)Q4: find all plans for getting an object between locations in the same city (13 conjuncts)Q5: find all plans for getting an object between locations in different cities (15 conjuncts)Q6: find all top-level plans where a truck and an object are in the same

For example, one query (Q2) was to find all plans using a particular airport.

[6]based on the domain used in [20]

```
(structure-retrieve
  (isa    case  #!case)
  (isa    plan  #!e-plan)
  (isa    goals #!goal-set)
  (isa    goal1 #!goal-located-at)
  (isa    goal2 #!goal-located-at)
  (isa    pkg1  #!package)
  (isa    pkg2  #!package)
  (isa    plan  case)
  (goals  plan  goals)
  (isa    goal1 goals)
  (isa    goal2 goals)
  (object goal1 pkg1)
  (object goal2 pkg2)
  (neq    pkg1  pkg2))
```

Figure 5: A structure matcher query in CaPER's transportation logistics domain

This query might be useful in avoiding plans that used an airport that is closed in the current situation. These queries were issued as probe graphs for the PARKA Structure Matcher.

Unindexed casebases of the various sizes were used. The number of cases, *e-plans*, and frames for each casebase is shown in Table 1. *E-plans* roughly correspond to subplans in traditional generative planning systems. This is important to note because each *e-plan* can be retrieved independently and is the functional equivalent of a case in a CBP system that stores cases monolithically. Cases consist of plans generated by the Nonlin ([18]) planner on randomly generated problems and encoded into PARKA's frame-based representation.

Cases	*E-plans*	*Frames*
20	296	3357
40	523	4660
60	740	5899
80	982	7280
100	1213	8616

Table 1: Casebases of Transport Logistics Plans Used

Queries were run using PARKA running on a Connection Machine CM-2 with $32,768$ virtual processors ($16,384$ physical processors). The results are shown in Figure 6.

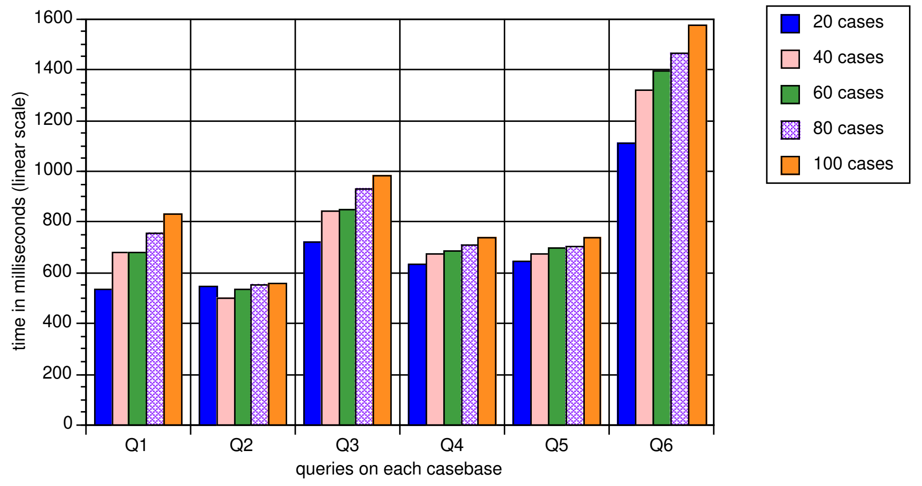

Figure 6: Query times (in milliseconds, linear scale) for the structure matcher on casebases of varying size.

These results show absolute parallel times ranging from 500 milliseconds on the smaller casebase to 1600 milliseconds on the larger casebase. As a point of comparison, an optimized serial version of this algorithm was implemented. The parallel algorithm running on the CM-2 achieved speedups ranging from 100 to 10,000 over our serial version on representative CaPER queries[7]. Also, in [12], we show that the serial implementation of the structure matcher scales supralinearly in the size of the KB.

We expect the parallel structure matcher's performance to scale (for reasons given in Section 6) to larger knowledge bases provided the size of the machine is not less than the size of the KB. More detailed empirical work will have to be done to fully characterize the time complexity of the algorithm.

9 Related work

Special purpose parallel systems have been constructed which are capable of performing matching of relational structures in semantic networks. SNAP [16] (Semantic Network Array Processor) is designed for efficient marker-passing

[7] The serial version was implemented on a Macintosh IIcx using Macintosh Common Lisp. While it is not useful to draw a direct comparison between the absolute performance of a Connection Machine and a Macintosh, we do show empirically in [12] that the structure matcher's run time scales extremely well on massively parallel hardware as compared with serial hardware.

and associative memory operations. In SNAP the nodes and relations of a semantic network are distributed across the array processors. Inferences are drawn by propagating markers in parallel from node to node. SNAP includes a rich language for writing marker-passing algorithms, including operations for controlling propagation, logical, and set-theoretic operations. Higuchi's IXM2 Parallel Associative Processor ([10]) is a machine which has been used as a special purpose associative memory for AI work, including machine translation applications involving matching of syntactic structures. The IXM2 has 64 PEs each with a 4KW of associative memory. This allows for the development of very fast special purpose processing of certain AI operations. (In some tests, the IXM outperformed the CM2 by a considerable margin (about 30 times), but it is a far less general machine.) The IXM2 has been used for the work in machine translation and also in the area of Genetic Algorithms. While both of these systems perform quite well on the classes of problems the were designed to address, we believe that the structure matching algorithm is a much more general solution.

PARADYME [13], like PARKA, was a frame system that used the massive parallelism of the CM. One application of PARADYME was an unindexed memory of cases, potentially of use to a CBR system but never actually used by one. PARADYME provided different procedures for case retrieval than those used by CaPER. PARKA's query algorithms are more complete and significantly faster than those used by PARADYME [3].

There are other massively parallel knowledge representation systems. Aronis [1] has developed a frame-based system for studying nonmonotonic inheritance mechanisms. The inheritance mechanisms of that system are much more powerful than PARKA's, but are nearly two orders of magnitude slower. Geller [6] has developed a frame-based system that offers essentially constant-time inheritance inferences, but only for tree-structured knowledge networks.

10 Conclusions

We believe that associative memory plays a crucial role in many artificial intelligence tasks. In this chapter we have presented a algorithm implemented on massively parallel hardware, which promises to provide the ability to perform associative retrieval on relational knowledge structures. This capability provides CBR systems with an alternative to the reliance on pre-indexing for efficient case retrieval. The structure matching algorithm has wide applicability not only to CBR, but to associative memory and classification tasks in general. The matching algorithm can retrieve structures very rapidly from large semantic networks. Memory can be probed flexibly, since no restrictions are placed on the ways in which knowledge may be accessed.

A more detailed description of how the structure matching algorithm is actually used for case retrieval can be found in [12]. The central goal of CaPER is to demonstrate the utility of fast structural retrieval for CBR. In support of the work on CaPER, we are currently investigating extension of the algorithm to incorporate weighted similarity metrics in the retrieval probe including automatic relaxation of the retrieval probe to allow inexact matching.

Acknowledgements

This research was supported in part by grants from NSF(IRI-8907890), ONR (N00014-J-91-1451), AFOSR (F49620-93-1-0065), the ARPA/Rome Laboratory Planning Initiative (F30602-93-C-0039) and by ARI (MDA-903-92-R-0035, subcontract through Microelectronics and Design, Inc.). Dr. Hendler is also affiliated with the UM Institute for Systems Research (NSF Grant NSFD CDR-88003012).

References

[1] Aronis, J., "Implementing Inheritance with Roles on the Connection Machine," Ph.D. dissertation, University of Pittsburgh, Pittsburgh, PA, 1993.

[2] Brachman, R.J. and Schmolze, J.G., "An Overview of the KL-ONE Knowledge Representation System", Cognitive Science, Vol. 9 (1985).

[3] Evett, M.P., Hendler, J.A., and Spector, L., "Parallel Knowledge Representation on the Connection Machine," *Journal of Parallel and Distributed Computing*, 1993 (forthcoming).

[4] Evett, M.P., Hendler, J.A., and Andersen, W.A., "Massively Parallel Support for Computationally Effective Recognition Queries", AAAI Eleventh National Conference on Artificial Intelligence, 1993.

[5] Fahlman, S.E., "NETL: A System for Representing and Using Real World Knowledge", MIT Press, Cambridge, MA, 1979.

[6] Geller, J., and Du, C.Y., "Parallel Implementation of a Class Reasoner," *Journal for Experimental and Theoretical Artificial Intelligence,* **3**, 1991, pp. 109-127.

[7] Gentner, D., "The mechanisms of analogical learning," In *Similarity and Analogical Reasoning*, Eds. S. Vosniadou and A. Ortony, Cambridge: Cambridge University Press, 1989, pp. 199-241.

[8] Goodman, M., "Projective Visualization: Learning to Act from Experience", AAAI Eleventh National Conference on Artificial Intelligence, 1993.

[9] Hammond, K.J. (1990a), "Case-based planning: A framework for planning from experience," *Cognitive Science*, Vol. 14 (1990), pp. 384-443.

[10] Higuchi, T., Kitano, H., Furuya, T., Kusumoto, H., Handa, K., and Kokubu, A., "IXM2: A Parallel Associative Processor for Knowledge Processing," AAAI Ninth National Conference on Artificial Intelligence, 1991.

[11] Hillis, W.D., "The Connection Machine", Cambridge, Massachusetts: The MIT Press, 1985.

[12] Kettler, B.P., Hendler, J.A., Andersen, W.A., Evett, M.P., "Massively Parallel Support for Case-based Planning", *IEEE Expert*, 1993 (forthcoming).

[13] Kolodner, J.L., and Thau, R., *Design and Implementation of a Case Memory*, Technical Report RL88-1, Thinking Machines Corporation, Cambridge, Massachusetts, 1988.

[14] Kolodner, J. L. and Jona, M. Y. "Case-Based Reasoning: an Overview", Technical Report #15, The Institute for the Learning Sciences, Northwestern University, 1991.

[15] Lenat, D.B. and Guha, R.V., "Building Large Knowledge-Based Systems", Addison Wesley, Reading, Mass., 1990.

[16] Moldovan, D.I., "Parallel Processing: From Applications to Systems", San Mateo, California: Morgan Kaufmann, 1993, pp. 484-504.

[17] Ross, B.H., "Some Psychological Results on Case-Based Reasoning," Proceedings: *Case-Based Reasoning Workshop* (DARPA), San Mateo, California: Morgan Kaufmann, 1989, pp. 144-147.

[18] Tate, A., *Project Planning Using a Hierarchic Non-linear Planner*, Research Report No. 25, Department of Artificial Intelligence, University of Edinburgh, 1976.

[19] Thagard, P.R. and Holyoak, K.J., "Why Indexing is the Wrong Way to Think About Analog Retrieval," In *Proceedings: Case-Based Reasoning Workshop* (DARPA), San Mateo, California: Morgan Kaufmann, 1989, pp. 36-40.

[20] Veloso, M.M., *Learning by Analogical Reasoning in General Problem Solving*, Technical Report CMU-CS-92-174, Department of Computer Science, Carnegie Mellon University, 1992.

[21] Waltz, D., "Is Indexing Used for Retrieval?," In *Proceedings: Case-Based Reasoning Workshop (DARPA)*, San Mateo, California: Morgan Kaufmann, 1989, pp. 41-45.

[22] Watanabe, L., and Rendell, L., "Effective Generalization of Relational Descriptions", AAAI Eighth National Conference on Artificial Intelligence, 1990.

[23] Winston, P.H., "Artificial Intelligence", 3rd ed., Addison–Wesley, Redding, Mass., 1992.