Improving search using indexing: a study with temporal CSPs

Nikos Mamoulis and **Dimitris Papadias**
Department of Computer Science
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
http://www.cs.ust.hk/{~mamoulis, ~dimitris}

Abstract

Most studies concerning constraint satisfaction problems (CSPs) involve variables that take values from small domains. This paper deals with an alternative form of temporal CSPs; the number of variables is relatively small and the domains are large collections of intervals. Such situations may arise in temporal databases where several types of queries can be modeled and processed as CSPs. For these problems, systematic CSP algorithms can take advantage of temporal indexing to accelerate search. Directed search versions of chronological backtracking and forward checking are presented and tested. Our results show that indexing can drastically improve search performance.

1 Introduction

Many problems in a variety of application domains can be modeled and solved as constraint satisfaction problems (CSPs). A binary CSP is defined by:

- a set of n variables $v_1, \ldots ,v_n$
- for each variable v_i a domain D_i of m_i values: $\{u_{i1},\ldots,u_{im_i}\}$
- for each pair of variables $\{v_i, v_j\}$, $i\neq j$, a binary constraint C_{ij}, which is a subset of $D_i \times D_j$.

An assignment $\{v_i \leftarrow u_{ia_i}, v_j \leftarrow u_{ja_j}\}$ is consistent if $(u_{ia_i}, u_{ja_j}) \in C_{ij}$. The goal is to find one or all solutions, i.e., n-tuples $(u_{1a_1},\ldots,u_{ia_i},\ldots,u_{ja_j},\ldots,u_{na_n})$ such that for each $\{i,j\}$, $i\neq j$, $\{v_i \leftarrow u_{ia_i}, v_j \leftarrow u_{ja_j}\}$ is consistent.

Several systematic search heuristics that aim to minimize the number of consistency checks have been proposed. Based on the general idea of *backtracking*, these methods try to improve the *backward step*, e.g. *backjumping*, or the *forward step*, e.g. *forward checking* [Haralick and Elliot, 1980]. Hybrid algorithms combine different types of backward and forward steps [Prosser, 1993]. The aforementioned search methods apply exhaustive search in the variable domains while assigning or pruning values. When the number of potential values for a variable is small (i.e. less than 100) linear scan suffices; for large domains, however, the overhead can be significant due to repetition of scan.

In this paper we study how indexing may be utilized by CSP algorithms to direct search and avoid linear scan of domains. As an application, we deal with CSPs, where the values are temporal intervals and the constraints are disjunctions of temporal relations as defined in [Allen, 1983]. Such problems may occur in planning or temporal databases. A previous work that uses indexing in temporal databases is [Dean, 1989]. In contrast to our method, where intervals are indexed according to their position in space, that method directs search using a conceptual hierarchical structure, the *discrimination tree*. Algorithms that combine CSP search and indexing to facilitate retrieval of structural queries in large spatial databases were presented in [Papadias *et al.*, 1998]. Here, we extend this work by investigating the application of directed search in temporal databases and study the performance gain of directed search under various problem conditions using different CSP algorithms.

The rest of the paper is organized as follows. Section 2 introduces data structures and search methods for temporal intervals. Section 3 shows how conventional methods for solving CSPs can be modified to accelerate search using indexing. Section 4 experimentally compares directed search algorithms with methods that do not use indexing. Finally, Section 5 concludes the paper with directions for future work.

2 Temporal CSPs

The class of problems that we deal with in this paper includes CSPs where domain values are well defined intervals in *discrete* time, and each binary constraint C_{ij} is a disjunction of the permissible set of Allen's [1983] temporal relations between v_i and v_j. Notice that the current issue is different from traditional temporal reasoning problems (e.g. [van Beek, 1992]) where the aim is to identify a consistent scenario for a set of continuous variables, given information about their relationships. Here, we *search* into domains of intervals for variable assignments that satisfy the given constraints.

This temporal CSP can be solved by employing traditional search techniques. Each time a variable is visited,

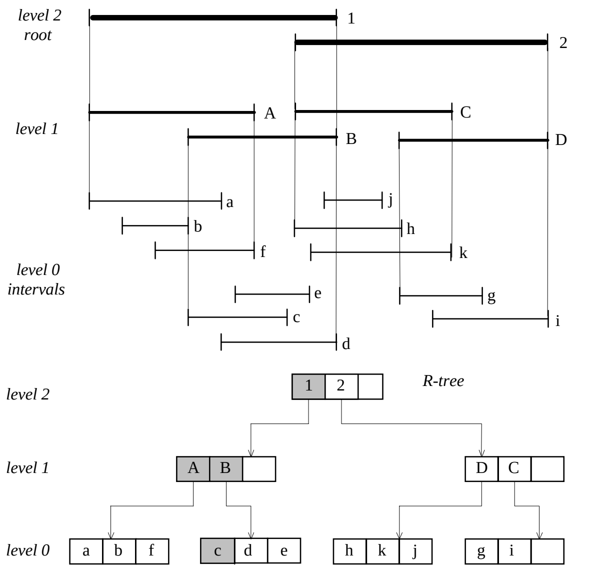


Figure 1: Organizing a set of intervals into a 1-d R-tree

its whole domain has to be scanned in order to identify consistent assignments. When the variable domains are small, classic CSP algorithms behave well, but as the domain size increases, linear scan for consistent values is expensive.

CSP algorithms can avoid linear scans by employing interval-based data structures that index the variable domains. The search for solutions can be *directed* upon *satisfaction* of the temporal constraints, using the data structure to minimize the number of consistency checks. Since search through the domains is repeated a large number of times, directed search has a significant effect on performance.

For the indexing of variable domains we use R-trees [Guttman, 1984], a data structure aimed at indexing multidimensional rectangles in spatio-temporal databases. The data rectangles are stored into leaf nodes; each intermediate node contains a pointer to a lower level node and the *minimum bounding rectangle* of the rectangles in this node. We chose R-trees because they are becoming a standard in both research and industry (e.g., Illustra, Informix) due to their efficiency, dynamic nature and relatively simple implementation. Alternatively, any interval-based data structure, such as interval trees and segment trees [Preparata and Shamos, 1985], can be employed.

Figure 1 illustrates how a 1-dimensional R-tree can be used for interval storage and retrieval, assuming maximum tree node capacity equal to 3. The data intervals (*a* to *k*) are stored in the leaf nodes. Artificial intervals in the higher tree nodes are formulated by grouping intervals from the level below (e.g., *a*,*b* and *f* are grouped in *A*). The end points of an intermediate node interval are the leftmost and rightmost end points of the intervals it indexes. The R-tree for a static set of intervals can be built in a bottom-up fashion after sorting them with respect to one end-point, e.g. in this example the left one

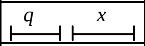
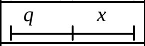
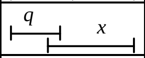
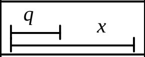
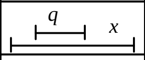
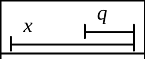
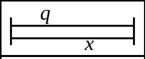
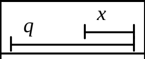
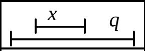
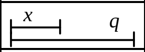
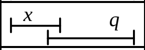
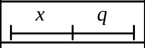
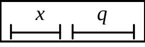
Relation	*Illustration*	*I.left*	*I.right*
b: q before x		*unbounded*	> *q.right*
m: q meets x		≤ *q.right*	> *q.right*
o: q overlaps x		< *q.right*	> *q.right*
s: q starts x		≤ *q.left*	> *q.right*
d: q during x		< *q.left*	> *q.right*
f: q finishes x		< *q.left*	≥ *q.right*
e: q equal x		≤ *q.left*	≥ *q.right*
fi: q finished by x		< *q.right*	≥ *q.right*
di: q contains x		< *q.right*	> *q.left*
si: q started by x		≤ *q.left*	> *q.left*
oi: q overlapped by x		< *q.left*	> *q.left*
mi: q met by x		< *q.left*	≥ *q.left*
bi: q after x		< *q.left*	*unbounded*

Table 1: Bounding conditions for intermediate node entries

(*R-tree packing*). Thus, the R-tree construction for *n* intervals costs *O(nlogn)*.

R-trees are very efficient in answering *range intersection* queries, i.e., find all intervals that intersect (share a common point with) a *query interval*. In addition, the hierarchical nature of R-tree facilitates general selection queries, i.e., retrieval of intervals that satisfy any temporal condition with respect to a query interval. As an example, consider the query: “find all intervals that start *immediately* after interval *b*” applied to the R-tree of Figure 1. Retrieval starts from the root of the tree; due to the relative positions of *b* and *2*, there can be no interval indexed by the root entry *2* that starts immediately after *b*. Thus the sub-tree under *2* is pruned. On the other hand, entry *1* can contain qualifying intervals and is recursively searched. Both entries *A* and *B* may point to a query answer and they are followed; the only solution is interval *c*. Typically, the retrieval cost is proportional to the number of solutions; when this is small, worst case performance is logarithmic to the number of stored intervals.

Let *q*=[*q.left*,*q.right*] be a query interval; Table 1 shows, for each temporal relation, the *bounding conditions* that an *intermediate-level* interval *I* should satisfy in order to potentially point to a query answer *x*. This is the criterion of following an R-tree node during retrieval. In the previous example (*b meets x*) the intermediate nodes to be searched should satisfy *I.left* ≤ *b.right* and *I.right* > *b.right*. Intermediate interval *2* is pruned because it violates *2.left* ≤ *b.right*. When the query is a disjunction of temporal relations, the disjunction of the bounding conditions is applied during search. For instance, if the search predicate is *starts* ∨ *during* ∨ *finishes*, the bounding conditions for an intermediate level entry I are $I.left \leq q.left$ and $I.right \geq q.right$. Although for the examples throughout the paper we use the tree of Figure 1, for our implementation we assume that each variable domain is different and indexed by a separate R-tree. The next section describes algorithms that use these indexes to prune the search space (the methods can be easily applied when some domains/trees are common).

3. Directed Search for Temporal CSPs

Classic local consistency methods, like *arc* and *path consistency*, perform well for small domains, because in this case there is a high probability that a value will be pruned. However, for large variable domains and dense constraint graphs usually they do not pay-off. Therefore, we do not consider conventional local consistency methods for pre-processing, but apply the temporal network maintenance algorithm from [Allen, 1983] prior to search in order to infer undefined constraints and refine the existing ones. This is equivalent to a path consistency algorithm that does not check value consistency, but *constraint graph consistency*, and its complexity is, therefore, independent from the domain sizes. For instance, consider a constraint graph of three variables, where C_{12} = *meets*, C_{23} = *during*, and C_{13} is undefined. The *implied* constraint C_{13} is then *overlap* $\vee$ *starts* $\vee$ *during*. If C_{13} was defined, then the *updated* constraint is $C_{13} \wedge$ {*overlap* $\vee$ *starts* $\vee$ *during*}. Notice that this reasoning method detects inconsistency prior to search, e.g. when $C_{13} \wedge$ {*overlap* $\vee$ *starts* $\vee$ *during*}$= \emptyset$.

Efficiency of CSP search depends on the order by which variables are instantiated [Dechter and Meiri, 1994]. The usual rule for *static variable ordering* (SVO) schemes is to “place the most constrained variable first”. A simple way to apply this rule is to sort the variables in decreasing order of their degree in the constraint graph. Since in our problem the constraint edges do not have the same tightness, we follow another method: instead of “adding 1” for each edge that goes out from a variable, we add a *weight* that represents the tightness of the constraint edge. For a temporal relation r, weight(r) is the inverse of the probability[1] $P(r)$ that r will be satisfied between two arbitrary intervals. In typical cases, the relation with the largest weight is *equal*, and the relations with the smallest, *before* and *after*. If a constraint is a disjunction of primitive relations, the weight is the inverse sum of the relation weights that participate in the disjunction, e.g.,

$$weight(meets \vee starts) = \frac{1}{P(meets) + P(starts)}$$

The weight of a variable is then the sum of the weights of all constraint edges adjacent to it. SVO will instantiate the “heaviest” variable first and the one with the smallest weight last. Whenever dynamic variable ordering (DVO) is employed, the above method is used to find the variable with the largest weight and place it first. The order of subsequent variables is dynamically changed according to their domain sizes during search.

[1] The probabilities of temporal relations can be estimated either by sampling the input data, or by using probabilistic estimation formulae given the distribution of interval positions and sizes.

3.1 Directed Search Backtracking

Integration of directed search into backtracking (BT) is rather simple. Whenever a variable v_j is visited, its consistent values are retrieved from the corresponding R-tree R_j by applying the instantiations of the previous variables v_i, $i<j$, as query intervals, and the constraints C_{ij} as retrieval conditions. During retrieval, the conjunction of bounding conditions, defined by each v_i, direct the search at intermediate R-tree levels. At the leaf level, the entries that satisfy the constraint C_{ij} with each variable v_i, are retrieved as consistent values for v_j.

To clarify the retrieval procedure, consider the following example. Let the query *overlap*(v_1,v_3) $\wedge$ *overlapped by* (v_2,v_3) and the instantiations $v_1 \leftarrow f$, $v_2 \leftarrow h$ in Figure 1 (i.e., v_3 should overlap f to the right and h to the left). In order to retrieve all consistent values for v_3, first the bounding conditions (BC) according to v_1 and v_2 are calculated using Table 1; BC_{13} = {$I.left < f.right$ and $I.right > f.right$} and BC_{23} = {$I.left < h.left$ and $I.right > h.left$}. The conjunction of the above constraints results in the bounding conditions {$I.left < f.right$ and $I.right > h.left$}, used to guide R-tree search. Retrieval starts from the root, where entry *2* is pruned as it does not satisfy $2.left < f.right$. Similarly, entry A violates $A.right > h.left$ and the sub-tree below is pruned. Only intervals under entry B can satisfy both C_{13} and C_{23}. Finally, d and e constitute the consistent assignments for v_3. Notice that the whole process costs 4 consistency checks at the intermediate levels and 6 at the leaf level (each interval under B is tested against both v_1 and v_2), whereas a linear scan would cost 14 consistency checks.

The above method can be applied with alternative forms of backtracking (e.g., backjumping, dynamic backtracking) using information about *nogoods*. In the next subsection we illustrate how forward checking can employ the index to prune the domains of future variables.

3.2 Directed Search Forward Checking

After a variable instantiation, forward checking (FC) *marks* in the domains of all future variables the values that are consistent with the current variable (*check-forward*). When a subsequent variable is to be instantiated only the marked values will be considered. This marking mechanism can be thought of as a *linear index* in the variable domain that points to the consistent values. Assume that an intermediate variable v_i is being instantiated. The domain of each future variable v_j ($i<j$) has already been pruned by the instantiation of variables before v_i. As the linear indexes of some variables may still be large, check-forward during instantiation of v_i can be rather costly. For these variables we apply directed

search using the R-tree and filter the results using the linear index. Filtering with respect to the linear index is needed, because directed search is performed on the whole domain (i.e., an interval which satisfies the current constraint with v_i may have been pruned out by a previous instantiation). The issue to be investigated is when to apply directed search instead of linear scan.

A good heuristic is to employ directed search at a future variable only when the number of remaining consistent values is large, and the search constraint is tight, so the search would benefit from the R-tree. For example, let C_{13} = *before* and C_{23} = *equal*. After v_1 is instantiated, directed search is applied to prune the domains of v_2 and v_3. When v_2 is given a value it is worth applying directed search again while checking forward for consistent values of v_3, because the domain of v_3 is still large and C_{23} is a tight constraint. The results of directed search that do not satisfy *before* with respect to the value of v_1 are filtered out. Now let C_{13} = *equal* and C_{23} = *before*. After v_1 is given a value, the domain of v_3 is expected to become very small; thus, during instantiation of v_2, linear scan of D_3 is cheaper than using the tree.

This mechanism is called a *double index*, in the sense that we keep both the whole domain R-tree and the linear index of consistent values and use either both, or only the linear index. Directed search is applied when the *expected* number of values that satisfy the search conditions is smaller than the number of values in the linear index. The number of retrieved values is estimated using the temporal relation probabilities. For instance, if C_{ij} = *overlap*, *P(overlap)* = 0.02 and $|D_j|$= 1000, then 20 intervals in the domain of v_j are expected to be consistent with the value of a previous variable v_i. If while instantiating v_i, the remaining consistent values of v_j from former checks is smaller than 20, linear scan at *check_forward*(v_i,v_j) will be preferred to directed search.

Alternatively, versions of the variable R-trees could be maintained at each instantiation level, where only consistent values would remain in the data structure, and search could be performed at each check-forward. We do not use this method because dynamic operations on data structures (i.e. insertion, deletion, construction) are usually expensive.

4 Experiments

In this section we compare the performance of *directed BT* (dirBT) and *directed FC* (dirFC) with plain versions of the algorithms based on linear scan. FC and dirFC use the *fail first* (FF) DVO heuristic [Haralick and Elliot, 1980]. BT and dirBT apply the SVO heuristic described in Section 3, because, as suggested in [Bacchus and van Run, 1995], BT with DVO does at least as much work as FC with DVO, thus it is just a FC-DVO algorithm with redundant checks.

The problems were randomly generated by modifying the parameters $<n,m,p_1,p_2>$ (see [Dechter and Meiri, 1994]), where n is the number of variables, m the cardinality of the domains, p_1 the probability that a random pair of variables is constrained (*constraint network density*), and p_2 the probability that a random assignment for a constrained pair is inconsistent (*constraint tightness*). The centers of the intervals in the variable domains are uniformly distributed and their sizes take values with an average 1.5% of the workspace. The intervals in each domain are indexed by R-trees with node capacity equal to 20.

In order to identify the hard region of the problems we use the *constrainedness* measure [Gent *et al.*, 1996]:

$$\kappa = 1 - \frac{\log_2(Sol)}{\log_2 \prod_{i=1}^{n} m_i} = 1 - \frac{\log_2(Sol)}{n \log_2 m} \qquad (1)$$

The hard region for an ensemble of problems is when $\kappa \approx 1$, whereas problems with $\kappa << 1$ and $\kappa >> 1$ are *easy and soluble* and *easy and insoluble*, respectively. The denominator of eq. (1) denotes the *size* of the problem and *Sol* denotes the number of solutions in a random problem of the class. If the binary constraints are independent, *Sol* can be estimated by:

$$Sol = \prod_{v=1}^{n} m_v \cdot \prod_{1 \le i, j \le n, i \ne j} P(C_{ij}) \qquad (2)$$

where $P(C_{ij})$, as described in the previous section, is the probability that two intervals from D_i and D_j satisfy C_{ij}. The first product in (2) corresponds to the total number of n-tuples of intervals, while the second one to the probability that a tuple constitutes a solution. For acyclic networks the constraints are independent and (2) will give a good estimation of the problem solutions. When cycles exist the constraints are no longer independent, but applying (2) for the minimum spanning tree of the graph (w.r.t. the constraint tightness) will give an overestimation of the expected solutions.

We first tested the performance improvement achieved by the path consistency (PC) method of Section 3 for *acyclic* networks (p_1 = (n-1)/(n(n-1)/2) = 2/n), by running experiments for n=10 and m=1000, and several values of p_2 around the hard region. For each value of p_2 we solved an ensemble of 100 problems and measured the average cost for finding one solution. Usually, random CSPs are generated with every constraint having exactly the same tightness. Because this cannot be done for the current problem, we chose disjunctions of temporal relations which may be different for each constraint, but have average tightness within $\pm 10^{-4}$ of the target value p_2. The hard region (κ = 1) is when $p_2 \approx 0.99953$. This large value of p_2 is due to the large domain size and the sparseness of the graph. In general, problems with sparse graphs are easy [Dechter and Pearl, 1987] and the few

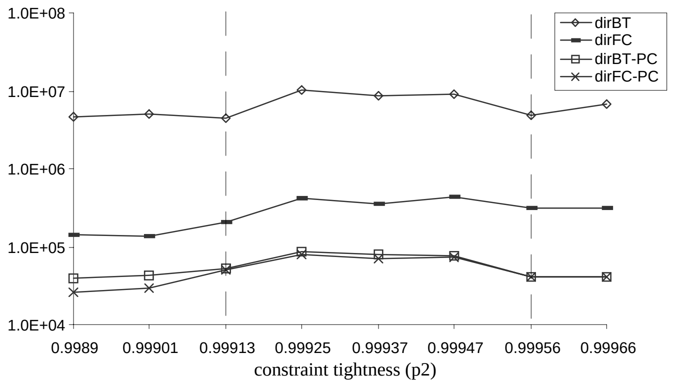


Figure 2: cost of directed search with and without PC

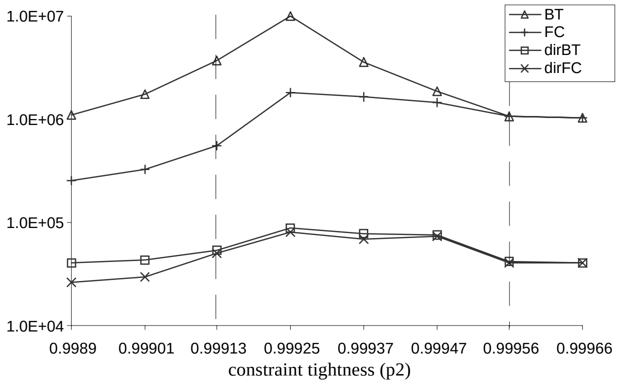


Figure 3: comparison of directed and plain search for trees

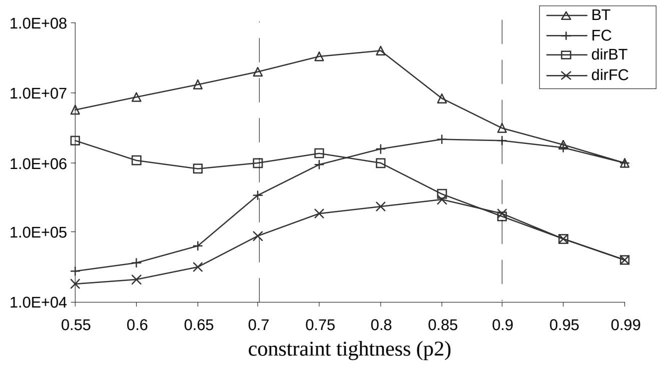


Figure 4: comparison of directed and plain search for cliques

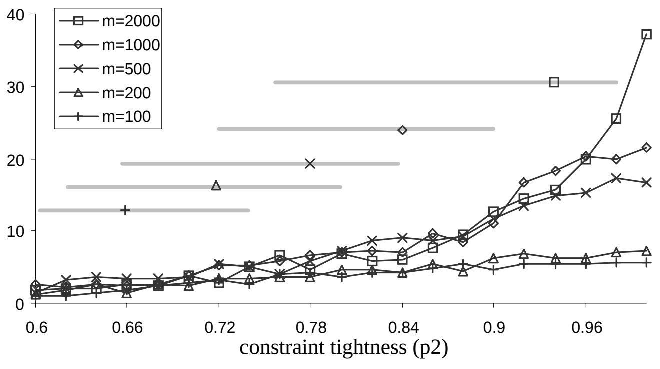


Figure 5: Performance gain of dirFC for various domain sizes

existing constraints must be very tight in order to generate a small number of solutions.

Figure 2 shows the cost (in terms of consistency checks[2]) of directed search BT and FC, with and without PC. The dotted vertical lines include the phase transition, i.e. the first ensemble that included an insoluble problem (left line) and the last ensemble that included a soluble one (right line). As expected, the performance gain of PC is significant since the inferred constraints drastically prune the search space. In the sequel we use PC for both directed search and plain versions of the algorithms. The nearest point to 50% solubility (crossover point) was at p_2 = 0.99925 (κ = 0.93748), where 54% of the problems were soluble.

The second set of experiments compares the cost of directed versus undirected search for tree (p_1 = 2/n) and clique (p_1 = 1) graphs. The experimental settings are the same as in the previous experiments (n=10 and m=1000; each ensemble contains 100 problems for each value of p_2 around the hard region). Figure 3 illustrates the cost for tree graphs. The directed search versions outperform the original algorithms by more than one order of magnitude. Tree constraint graphs can alternatively be solved by the application of *arc consistency* prior to search, which would lead to backtrack-free search [Dechter and Pearl, 1987]. This method is also expected to profit from indexing.

For cliques, the hard region cannot be estimated using (1), due to the dependency of the constraints. In order to identify it, we experimented with various values of p_2. The diagram of Figure 4 illustrates the behavior of algorithms for a wide scope of tightness values. Notice that the hard region corresponds to a larger range of p_2 than in the case of tree graphs. While the cost difference between directed and regular versions of the algorithms is again about an order of magnitude for large p_2, their performance converges as the constraints become loose. When p_2 < 0.7 most of the constraints contain relations *before* or *after*. On the average, each such constraint prunes out around 50% of the values in a domain, and directed search is only about twice as fast as linear scan.

As a general conclusion, the performance gain of directed search in comparison to linear scan grows with the constraint tightness p_2. dirFC is, as expected, more efficient than dirBT due to its relevance with FC which in most cases outperforms BT. In the sequel we focus on the performance gain of dirFC with respect to FC for various problem settings.

The next experiment compares FC and dirFC for n=10, clique topology, and various values of m and p_2. Figure 5 shows how many times dirFC is faster than FC (average

[2] We consider as consistency checks the comparisons that take place at all levels of the R-trees.

of 50 instances per experimental setting). The gray horizontal lines present the phase transition for each m, and the symbol on the line indicates the position of the crossover point. With the exception of m=100-200 and very small tightness values (p_2<0.65), dirFC outperforms FC several times. For small domains the constraints should be very tight in order for the intermediate R-tree node comparisons to pay-off. However, directed search is intended for large domains (e.g., in spatio-temporal databases the cardinality often exceeds 10^5), and in such cases the performance gain is significant. For this experiment, we limited m to 2000, so that FC could terminate in reasonable time.

#vars	p_2	%soluble	FC	dirFC	gain
5	0.9996	58%	562466	21974	25.6
10	0.75	42%	2118699	299047	7.1
15	0.68	43%	2715124	848629	3.2
20	0.61	46%	4660763	2922984	1.6
25	0.58	42%	8570553	6619067	1.3

Table 2: Performance gain of dirFC compared to FC for various number of variables

In order to test the implication of the number of variables, we fixed m=1000, p_2 to be around the crossover point, the graph topology to clique, and generated 100 random problems for several values of n. Table 2 presents, for each ensemble, the value of p_2, the percentage of soluble problems, and the mean consistency checks of FC and dirFC. The last row of the table shows how many times dirFC was faster than FC. As the number of variables increases, the value of p_2 at the crossover point decreases. As a result, the costs of dirFC and FC converge due to the relaxation of constraints which deteriorates R-tree search, making it comparable to linear scan.

5 Discussion

This paper studies a specific CSP problem, where variable domains are large collections of well defined intervals and constraints are temporal relations. We show how systematic CSP algorithms can take advantage of indexing to accelerate search. Although we experimented with two representative algorithms, chronological backtracking and forward checking, directed search can be applied with a variety of algorithms and heuristics (e.g. arc consistency). In typical database applications, where m is in the order of 10^5 or above and n<10, the performance gain of directed search is large.

Application of data structures is not limited to the temporal CSP discussed here. Any problem, where variables have large domains and the nature of constraints facilitates directed search, can benefit from it. This particularly applies for spatial and multimedia databases where several types of content-based queries can be modeled as CSPs (e.g., find all triplets (v_1,v_2,v_3) of objects such that v_2 is *inside* v_1, and v_1 is *northeast* of v_3). An alternative approach that solves the above problem by hierarchically applying CSP algorithms at each R-tree level is described in [Papadias *et al.*, 1999].

Acknowledgement

This work was supported by grant HKUST 6151/98E from Hong Kong RGC and grant DAG97/98.EG02.

References

[Allen, 1983] James F. Allen. *Maintaining Knowledge about Temporal Intervals.* Communications of the ACM 26(12): 832-843, 1983.

[Bacchus and van Run, 1995] Fahiem Bacchus, Paul van Run. *Dynamic variable reordering in CSPs.* In Proceedings of CP-95, 1995.

[Dean, 1989] Thomas Dean. *Using Temporal Hierarchies to Efficiently Maintain Large Temporal Databases*. Journal of ACM 36(4): 687-718, 1989.

[Dechter and Meiri, 1994] Rina Dechter, Itay Meiri. *Experimental Evaluation of Preprocessing Algorithms for Constraint Satisfaction Problems*. Artificial Intelligence 68(2): 211-241, 1994.

[Dechter and Pearl, 1987] Rina Dechter, Judea Pearl. *Network-Based Heuristics for Constraint-Satisfaction Problems*. Artificial Intelligence 34(1): 1-38, 1987.

[Gent *et al.*, 1996] Ian P. Gent, Ewan MacIntyre, Patrick Prosser, Toby Walsh. *The Constrainedness of Search*. In Proceedings of AAAI-96, 1996.

[Guttman, 1984] Antonin Guttman. *R-trees: A Dynamic Index Structure for Spatial Searching*. In Proceedings of ACM SIGMOD, 1984.

[Haralick and Elliot, 1980] Robert M. Haralick, Gordon L. Elliott. *Increasing tree search efficiency for constraint satisfaction problems.* Artificial Intelligence 14(3): 263-313, 1980.

[Papadias *et al.*, 1998] D. Papadias, N. Mamoulis and V. Delis. *Querying by Spatial Structure*. In Proceedings of VLDB, 1998.

[Papadias *et al.*, 1999] D. Papadias, P. Kalnis, N. Mamoulis. *Hierarchical Constraint Satisfaction in Spatial Databases*. In Proceedings of AAAI-99, 1999.

[Preparata and Shamos, 1985] F. Preparata, M. Shamos. *Computational Geometry*. Springer, 1985.

[Prosser, 1993] Patrick Prosser. *Hybrid Algorithms for the Constraint Satisfaction Problem*. Computational Intelligence, 9(3): 268-299, 1993.

[van Beek, 1992] Peter van Beek. *Reasoning about qualitative temporal information*. Artificial Intelligence 58: 297-324, 1992.