Distributed Search in Railway Scheduling Problems

Montserrat Abril, Miguel A. Salido, Federico Barber
Dpto. Sistemas Informáticos y Computación, Universidad Politécnica de Valencia
Camino de Vera s/n, 46022, Valencia, Spain
{mabril, msalido, fbarber}@dsic.upv.es

Abstract

Many problems of theoretical and practical interest can be formulated as Constraint Satisfaction Problems (CSPs). Solving a general CSP is known to be NP-complete; however, distributed models may take advantage of dividing the problem into a set of simpler interconnected sub-problems which can be more easily solved. The purpose of this paper is three-fold: first, we present a technique to distribute the constraint network by means of selection of tree structures. Thus, the CSP is represented as a *meta-tree CSP structure* that is used as a hierarchy of communication by our distributed algorithm. Then, a distributed and asynchronous search algorithm (DTS) is presented. DTS is committed to solving the *meta-tree CSP structure* in a *Depth-First Search Tree*. Finally, an intra-agent search algorithm is presented. This algorithm takes into account the *Nogood_message* to prune the search space. We have focused our research on the railway scheduling problem which can be distributed by tree structures. We show that our distributed algorithm outperforms well-known centralized algorithms.

keywords: Distributed Constraint Satisfaction Problems, Tree Partition, Train Scheduling.

Introduction

Many real problems in Artificial Intelligence (AI) as well as in other areas of computer science and engineering can be efficiently modelled as Constraint Satisfaction Problems (CSPs), which can be solved using constraint programming techniques. Some examples of such problems include: spatial and temporal planning, qualitative and symbolic reasoning, diagnosis, decision support, scheduling, hardware design and verification, real-time systems and robot planning. Most of the work is focused on general methods for solving CSPs.

However, many of the problems solved by using centralized algorithms are inherently distributed. Some works are currently based on distributed CSPs (see special issue of Artificial Intelligence, Volume 161, 2005). Furthermore, many researchers are working on graph partitioning (Schloegel, Karypis, & Kumar 2003). The main objective of graph partitioning is to divide the graph into a set of regions so that each region has roughly the same number of nodes and the sum of all edges connecting different regions is minimized.

Preprint submitted to Elsevier

Researchers are almost always interested in the size of the problems (nodes or edges), although a few studies have been made on the graph structure of the sub-problems induced by the partition (Miller 1986). For instance, one study seeks a node-separator whose induced graph is Hamiltonian.

Graph partitioning can also be applied to constraint satisfaction problems. Thus, we can use graph partitioning when dealing with large-scale CSPs to distribute the problem into a set of sub-CSPs. For instance, we can divide a CSP into several subCSPs so that constraints among variables of each sub-CSP are minimized (Salido & Barber 2006).

Otherwise, a domain-dependent partition can be used. This requires a deeper analysis of the problem to be solved. In this paper, we also show that a domain-dependent partition obtains a more adequate distribution so that greater efficiency is obtained.

In this paper, we present two techniques for structuring and solving CSPs. To this end, the CSP is previously organized in a *meta-tree CSP structure* so that the original constraint graph is partitioned into trees that represent sub-problems. Thus, the search algorithm carries out the search in each node in linear time (Freuder 1982), (Dechter & Pearl 1987).

Our research also focuses on the railway scheduling problem. Railway traffic has increased considerably, which has created the need to optimize the use of railway infrastructures. This is, however, a hard and difficult task. Our aim is to model the railway scheduling problem as a Constraint Satisfaction Problem (CSP) and solve it using a distributed CSP solver. Due to the topological properties of the railway scheduling problem, the resultant CSP can be distributed in semi-independent sub-problems so that the solution can be solved easier.

In the following section, we summarize some definitions about CSPs. A tree partition method is presented in section 3. Our distributed algorithm for solving *meta-tree CSP structures* is presented in section 4. We present our intra-agent search method in section 5. In section 6, we describe the railway scheduling problem. An evaluation of our methods over real railway networks is presented in section 7. Finally, we summarizes our conclusions in section 8.

Centralized, Distributed and Partitionable CSPs

In this section, we present some basic definitions related to CSPs, which will be convenient for our purposes and will unify works from the constraint satisfaction community. Then, we present three ways of solving a CSP: as a centralized problem, as a partitionable problem and as a distributed problem.

A ***CSP*** consists of: a set of variables $X = \{x_1, x_2, ..., x_n\}$; each variable $x_i \in X$ has a set D_i of possible values (its domain); a finite collection of constraints $C = \{c_1, c_2, ..., c_p\}$ that restricts the values that the variables can simultaneously take.

A **solution** to a CSP is an assignment of values to all the variables so that all constraints are satisfied; a problem with a solution is termed *satisfiable* or *consistent*.

A ***binary constraint network*** is a network in which every constraint subset involves at most two variables. In this case, the network can be associated with a constraint graph, where each node represents a variable and the arcs connect nodes whose variables are explicitly constrained (Dechter 1992).

A ***meta-tree CSP structure*** is a tree whose nodes are composed by trees. Thus, we will refer to the main tree as *meta-tree CSP structure* and to each individual tree as *single-tree*. We will define each node of the *meta-tree CSP structure* as *meta-node* and each individual and atomic node of the trees as *single-node*. It can be deduced that each *meta-node* corresponds to a *single-tree*. Each constraint between two *single-nodes* of different *meta-nodes* is called ***inter-constraint***. Each constraint between two *single-nodes* of the same *meta-node* is called ***intra-constraint***.

Partition : A partition of a set C is a set of disjoint subsets of C whose union is C. The subsets are called the blocks of the partition.

Distributed CSP: A distributed CSP (DCSP) is a CSP in which the variables and constraints are distributed among automated agents (Yokoo & Hirayama 2000).

Each agent has some variables and attempts to determine their values. However, there are *inter-constraints* and the value assignment must also satisfy them. In our model, there is a set of agents and each agent knows the set of constraints and the domains of variables involved in these constraints.

Partition of CSPs

There are many ways to solve a CSP. However, these problems can be classified into three categories: centralized problems, distributed problems, and partitionable problems.

- A CSP is a *centralized CSP* when there are no privacy/security rules between parts of the problem, and all knowledge about the problem can be gathered into one process. It is commonly recognized that centralized CSPs must be solved by centralized CSP solvers. Many problems are represented as typical examples to be modelled as a centralized CSP and solved using constraint programming techniques. Some examples are: sudoku, n-queens, map coloring, etc.
- A CSP is a *distributed CSP* when the variables, domains and constraints of the underlying network are distributed among agents. This distribution is carried out due to many factors: constraints may be strategic information that should not be revealed to competitors, or even to a central authority; a failure of one agent can be less critical and other agents might be able to find a solution without the failed agent. Examples of such systems are sensor networks, meeting scheduling, web-based applications, etc.
- A CSP is a *partitionable CSP* when the global problem can be divided into smaller problems (sub-problems) which must be coordinated to find the solution to the global problem. For example, the search space of a CSP can be divided into several regions, and a solution is found by using parallel computing.

Given these three categories, we can conclude that a distributed CSP cannot be solved by centralized techniques. However, can a centralized CSP be solved by distributed techniques? The answer is 'yes' if the CSP is partitionable and the size of the problem is high enough to decompose into a set of subproblems.

Real problems usually imply models with a great number of variables and constraints, causing dense networks of inter-relations. Problems of this kind can be handled as a whole only at overwhelming computational cost. Thus, it could be an advantage to decompose problems of this kind into several simpler interconnected sub-problems which can be more easily solved.

For instance, the map coloring problem is a typically centralized problem. The goal of a map coloring problem is to color a map so that regions sharing a common border have different colors. Let's suppose that each country of Europe must be colored. Figure 1 (1) shows a colored portion of Europe. This problem can be solved by a centralized CSP solver. However, if the problem is to color each region of each country of Europe (Spain, Figure 1(3); France, Figure 1(4)), it is easy to see that the problem can be partitioned into a set of sub-problems that are grouped by clusters. This problem can be solved as a distributed problem, even when the problem is not inherently distributed.

A map coloring problem can be solved by first converting the map into a graph where each region is a vertex and an edge connects two vertices if and only if the corresponding regions share a border. In our problem of coloring the regions of each country of Europe, it can be observed that the corresponding graph maintains clusters representing each country (Spain, Figure 1(3); France, Figure 1(4)). Thus, the problem can be solved in a distributed way.

Why Tree Partition?

As Rina Dechter states in (Dechter 1992), a problem is considered easy when it admits a solution in polynomial time. In the context of constraint networks, a problem is easy if an algorithm like backtracking can solve it in a backtrack-free manner, i.e., without dead-ends, thus producing a solution in linear time with regard to the number of variables and constraints. Theoretical research has identified topological features that determine this level of consistency and has yielded tractable algorithms for transforming some networks into backtrack-free representations.

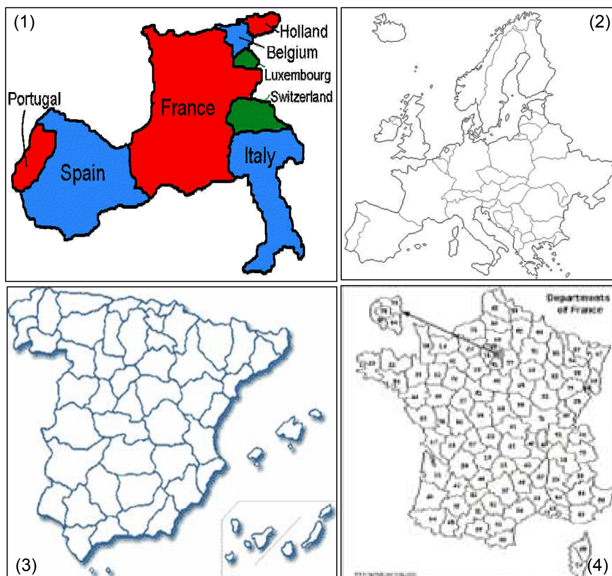

Figure 1: map coloring of Europe.

A main topological feature is centered on a graphical parameter called *width*, and the definitions are relative to the primal constraint graph. An ordered (primal) constraint graph is defined as one in which the nodes are linearly ordered to reflect the sequence of variable assignments executed by the backtracking algorithm. The *width of a node* is the number of arcs that connect that node to previous ones, the *width of an ordering* is the maximum width of all nodes, and the *width of a graph* is the minimum width of all orderings of that graph. It is known that only trees are width-one graphs (Freuder 1982). An ordered constraint graph is backtrack-free if the level of directional strong consistency along this order is greater than the width of the ordered graph. Thus, if the graph has width-one (i.e., it is a tree), a directional two-consistency is sufficient (Dechter 1992) to solve the problem in linear time.

How to Convert a binary CSP into a Meta-Tree CSP Structure

Any binary CSP can be translated into a *meta-tree CSP structure*. However, there are many ways to divide a graph into trees. Depending on user requirements, it may be desirable to obtain balanced *single-trees*, that is, each *single-tree* maintains roughly the same number of *single-nodes*; or it may be desirable to obtain *single-trees* in such a way that the number of edges connecting two *single-trees* are minimized.

Due to the complexity of finding the best partition, our technique finds an unbalanced tree partition in polynomial time (n^2 in the worst case, where n is the number of variables). It divides the problem into an undefined number of trees.

The *TreePartition* algorithm (Algorithm 1) divides the network graph G into k sub-graphs which are trees. The nodes and edges of graph G are the variables and constraints of the CSP, respectively. *TreePartition* randomly selects a root node that does not belong to another sub-graph, then the *SearchTree* function constructs a tree. This function recursively carries out a Depth First Search in graph G. The *SearchTree* function selects a new node i which is connected with *VarNode* and whose inclusion in the present *Tree* does not introduce a cycle, that is, node i is not connected with another node of the present *Tree*. Node i is marked as *visited*; it indicates that this node already belongs to one tree. Tree construction finishes when either there are no *unvisited* nodes or the remainder nodes introduce cycles in the present tree. The *TreePartition* algorithm finishes when all nodes of G belong to any sub-graph.

```
Algorithm TreePartition(G)
Input: Graph G, originally all nodes are unvisited.
       Start meta-node v of G
Output: Tree_partition
  Tree_partition=∅;
  while G≠ ∅ do
      Tree=∅;
      RootNode = selectNode(G);
      insert RootNode into Tree;
      mark RootNode as visited;
      SearchTree(G,RootNode,Tree);
      insert Tree into Tree_partition;
  end
end
function SearchTree(G, VarNode, Tree)
  forall i adjacent(1) to VarNode ∧ i unvisited do
      if NoCycle(i,Tree) then
          insert i into Tree;
          mark i as visited;
          SearchTree(G,i,Tree);
      end
  end
  /* (1) two single nodes are
     adjacent if at least one
     constraint exists between them.
     */
```

Algorithm 1: *TreePartition* Algorithm.

The next step is to build the *meta-tree CSP structure* with k *meta-nodes* that will be studied by *agents*. This *meta-tree CSP structure* is used as a hierarchy to communicate messages between *meta-nodes*. The *meta-tree CSP structure* is built using Algorithm 2. The nodes and edges of graph G are, respectively, the *meta-nodes* and *inter-constraints* obtained after the CSP partition. The root *meta-node* is obtained by selecting the most constrained *meta-node*. The *MetaTreeCSPStructure* algorithm then simply puts *meta-node* v into the *meta-tree CSP structure* (*process(v)*); it initializes a set of markers to indicate which vertices have been visited; it chooses a new *meta-node* i and then calls *MetaTreeCSPStructure*(G,i) recursively. If a *meta-node* has several adjacent *meta-nodes*, it would be equally correct to choose them in any order, but it is very important to delay the test for whether a *meta-node* is visited until the recursive

calls for previous *meta-nodes* are finished.

Algorithm MetaTreeCSPStructure(G,v)

Input: Graph G, originally all nodes are unvisited.
Start *meta-node* v of G
Output: *meta-tree CSP structure*

process(v);
mark v as visited;
forall meta-node i *adjacent*$^{(1)}$ *to v unvisited* **do**
    MetaTreeCSPStructure(G,i);
end
/* (1) *meta-node i* is adjacent to *meta-node v* if at least one *inter-constraint* exists between *i* and *v*. */

Algorithm 2: *MetaTreeCSPStructure* Algorithm.

Our aim is to solve CSPs by dividing the constraint graph by means of trees. Figure 3-1 shows a simple example of CSP. In Figure 3-2, this CSP has been divided into several trees and has been translated into a *meta-tree CSP structure*.

DFSTreeSearch Algorithm (DTS)

In the specialized literature, there are many works about distributed CSPs. In (Yokoo & Hirayama 2000), Yokoo et al. present a formalization and algorithms for solving distributed CSPs. These algorithms can be classified as centralized methods, synchronous backtracking or asynchronous backtracking (Yokoo & Hirayama 2000).

Our algorithm, called DFSTreeSearch (DTS), can be considered as a distributed and asynchronous algorithm. DTS is committed to solving the *meta-tree CSP structure* in a *Depth-First Search Tree* (*DFS Tree*) where the root *meta-node* is composed of the most constrained *single-tree* (in the sense that this *single-tree* maintains a higher number of *single-nodes*). *DFS trees* have already been investigated as a means to boost search (Decher 2003). Due to the relative independence of nodes lying in different branches of the *DFS tree*, it is possible to perform search in parallel on these independent branches.

Once the variables are divided and arranged, the problem can be considered as a distributed CSP, where a group of *agents* manages each *single-tree* with its variables (*single-nodes*) and its constraints (*intra-constraints*). Each *agent* is in charge of solving its own sub-problem by means of any search method. Each sub-problem is composed of its CSP, which is subject to the variable assignment generated by the ancestor *agents* in the *meta-tree CSP structure*.

Thus, the root *agent* works on its sub-problem (root *meta-node*). If the root *agent* finds a solution, then it sends the consistent partial state to its children *agents* in the *meta-tree CSP structure*, and all children work concurrently to solve their specific sub-problems knowing the consistent partial states assigned by the root *agent*. When a child *agent* finds a consistent partial state, it again sends this partial state to its children and so on. Finally, leaf *agents* try to find a solution to their own sub-problems. If each leaf *agent* finds a consistent partial state, it sends an OK message to its parent

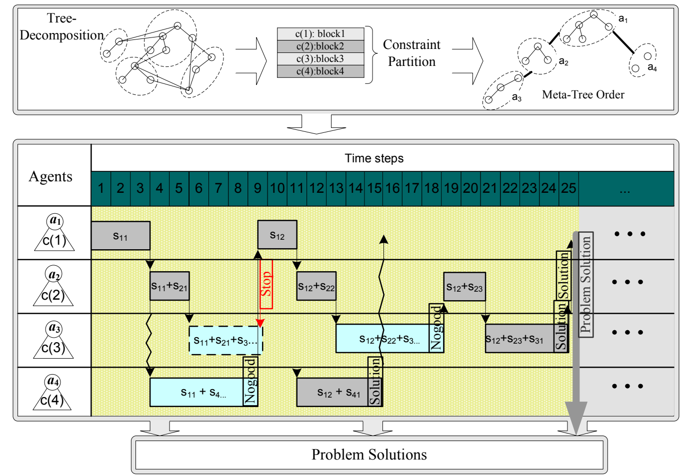

Figure 2: Execution of DFSTreeSearch Algorithm.

agent. When all leaf *agents* answer their parents with OK messages, a solution to the entire problem is found. When a child *agent* does not find a solution, it sends a Nogood message to the parent *agent*. The Nogood message contains the variables that empty the variable domains of the child *agent*. When the parent *agent* receives a Nogood message, it stops the search carried out by the children and tries to find a new solution taking into account the Nogood information and so on. If a parent *agent* finds a new solution, it will start the same process again sending this new solution to its children. Each *agent* works in the same way with its children in the *meta-tree CSP structure*. However, if the root *agent* does not find a solution, then DTS returns *no solution found*.

The top of Figure 2 shows our technique for partitioning a CSP into a *meta-tree CSP structure*. Then, DTS is carried out. Once the CSP is partitioned, the root *agent* (a_1) starts the search process finding a partial solution. It sends this partial solution to its children. *Agents* that are brothers are committed to concurrently finding the partial solutions of their sub-problems. Each *agent* sends the partial problem solutions to its children *agents*. A problem solution is found when all leaf *agents* find their partial solution. For example, (state $s_{12}+s_{41}$) + (state $s_{12}+s_{23}+s_{31}$) is a problem solution. The concurrence can be seen in Figure 2 in *Time step* 4 in which *agents* a_2 and a_4 are concurrently working. *agent* a_4 sends a Nogood message to its parent (*agent* a_1) in step 9 because it does not find a partial solution. Then, *agent* a_1 stops the search process of all its children and it finds a new partial solution which is sent to its children. Now, *agent* a_4 finds its partial solution, and *agent* a_2 works with its child, *agent* a_3, to find their partial problem solution. When *agent* a_3 finds its own partial solution, the global problem will be found. This happens in *Time step* 25.

Example

Figure 3 shows an example to analyze the behavior of DTS.

First, the constraint network of Figure 3(1) is partitioned into three trees and the *DFS tree* is built (Figure 3(2)). Agent

a finds its first partial solution ($X_1 = 1, X_2 = 1$) and sends it to its children: agents b and c (see Figure 3(3)). This is a good partial solution for agent c (Figure 3(4)); however this partial solution empties the X_3 variable domain. Thus, agent b sends a Nogood message to its father (Nogood ($X_1 = 1$)) (Figure 3(5)). Then, agent a processes the Nogood message, prunes its search space, finds a new partial solution ($X_1 = 2, X_2 = 2$) and sends it to its children (Figure 3(6)). At this point in the process, agent c sends a Nogood message to its father (Nogood ($X_1 = 2$)) because X_5 variable domain is empty (Figure 3(7)). Agent a stops the search of agent b (Figure 3(8)) and then processes the Nogood message, prunes its search space, finds a new partial solution ($X_1 = 3, X_2 = 3$) and sends it to its children (Figure 3(9)). Since this last partial solution is good for both children, they respond with an OK message and the search ends (Figure 3(10)), and returns the solution presented in Figure 3(11).

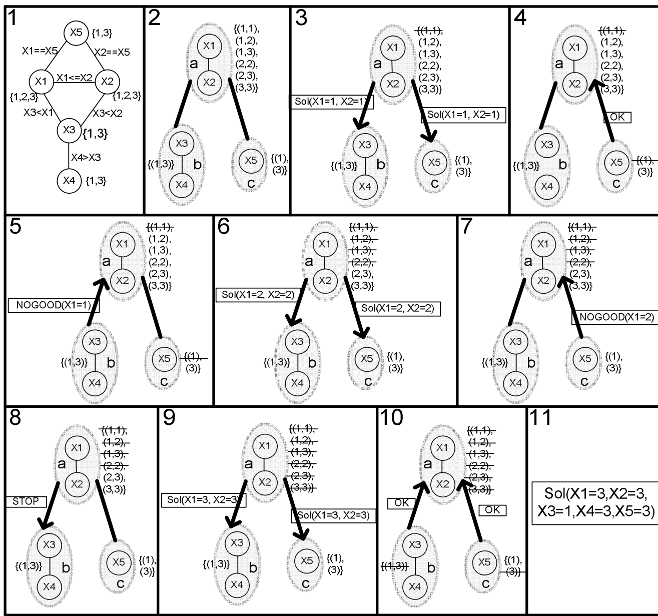

Figure 3: Example of DTS execution.

Intra-agent search

In section 4, we showed the distributed algorithm DTS for solving distributed CSPs, and we pointed out that each *agent* is in charge of solving its own sub-CSP by means of a search method. *Agents* are free to select their own search methods in order to find *partial_solutions*.

In this section, we propose a new algorithm to solve sub-CSPs with a tree structure: Tree Search Algorithm (*TSA*). This algorithm is based on the theorem presented in (Decher 2003): "Let *d* be a width-1 ordering of a constraint tree *T*. If *T* is arc-consistent relative to *d*, then the network is backtracking-free along *d*". Therefore, TSA finds a *partial_solution* in polynomial time. Furthermore, our algorithm is based on *Nogood_message*, which allows us to prune the search space.

TSA(G, *Nogood_message*)

Input: Constraint Graph G=$\{X, E\}$ where E is a set of constraints and X=$\{x_1, ..., x_n\}$ is a set of variables ordered according to d=$(x_1, ..., x_n)$; *Nogood_message*: set of inconsistent variables.
Output: *partial_solution*

if Nogood_message==∅ **then**
  DAC(G); `/* apply directional arc-consistency along d. */`
  if $\forall X_i$: $D_i \neq \emptyset$ **then**
    solution ← assignValue(1);
  else
    return *NO_SOLUTION*;
  end
else
  markVarNogood(X, *Nogood_message*);
  if *((*$\exists\, x_j$,$x_s \in X$: x_j, $x_s \in Nogood_message$ *and* $s > j$*)* **and** *(*$\nexists\, x_t \in X$:*(*$x_t \in Nogood_message \wedge j < t < s$*))* **then**
    set value(x_j) restricted by x_s;
    reviewDAC(x_j,x_s);
  else
    reviewDAC(x_j,x_1);
  end
  solution← NO_SOLUTION;
  while *solution == NO_SOLUTION* **do**
    k←j;
    while *(*$k \geq$ *1* ***and*** D_k*==∅)* **do**
      $k \leftarrow previousVarNogood$;
    end
    if $k < 1$ **then**
      return *NO_SOLUTION*;
    else
      foreach $x_i \in X$ / $j \leq i < n$ **do**
        reset effects of $x_i \leftarrow value(x_i)$;
      end
      solution← assignValue(k);
      if $\nexists\, x_i \in Nogood_message$: x_i *changed* $value(x_i)$ **then**
        solution← NO_SOLUTION;
        markVarNogood(X,*Nogood_message*);
        set value(x_j) restricted by x_s;
        reviewDAC(x_j,x_s);
      end
    end
  end
end
return *solution*;

Algorithm 3: Tree Search Algorithm.

reviewDAC(*startVar*, *endVar*)

if *(parent(*$startVar$*)* $<$ $endVar$ ***and*** *review(parent(*$startVar$*),*$startVar$*,*$endVar$*))* **then**
  reviewDAC(parent($startVar$),$endVar$);
end

Algorithm 4: ReviewDAC Algorithm.

```
review(i, j, end)

    removed ← false;
    foreach v ∈ D_i do
        if (∄ y ∈ D_i / c_ij is true then
            set v restricted by x_fin;
            removed ← true;
        end
    end
    return removed;
```

Algorithm 5: Review Algorithm.

```
assignValue(i)

    x_i ← a_s: a_s ∈ D_i;
    foreach x_j / x_j is child of x_i do
        prune D_j according to (x_i ← a_s and c_ij)
    end
    if i < n then
        unmark x_{i+1} varNogood;
        assignValue(i + 1);
    end
```

Algorithm 6: AssignValue Algorithm.

```
markVarNogood(Var, Nogood_message)

    x_i ← a_s: a_s ∈ D_i;
    foreach x_i / (x_i ∈ Var and x_i ∈ Nogood_message)
    do
        mark x_i as VarNogood;
        x_j ← parent(x_i);
        while x_j ≠ Ø do
            mark x_j as VarNogood;
            x_j ← parent(x_j);
        end
    end
```

Algorithm 7: MarkVarNogood Algorithm.

The algorithms 3, 4, 5, 6 and 7 show the pseudo-code of the Tree Search Algorithm (TSA). Algorithm 3 shows the main function of TSA and the others show the auxiliary functions. The main algorithm starts with a tree network structure and a width-1 ordering d. TSA has a different behavior for finding the first solution of the subproblem than for finding the rest of the solutions of the subproblem. To find the first solution, TSA carries out directional arc-consistency along d. If the network is directional arc-consistent, then TSA uses the algorithm $assignValue(1)$ following the established order d to ensure that no backtracking will be necessary. If the network is not directional arc-consistent, then the problem is not consistent and returns $NO-SOLUTION$.

If TSA receives a $Nogood$ message and X_j is the highest variable included in the $Nogood$ message(according to ordering d), TSA bounds the search space by means of avoiding the domains of higher variables to X_j. Thus, any valid partial solution can be lost due to the fact that the removed search space is not consistent with the entire problem. We must take into account that no variable included in the $Nogood$ has been assigned to any other value of its domain.

Once TSA receives a $Nogood$ message, it labels all the variables involved in the $Nogood$ message and the preceding variables, (according to d), as $VarNogood$. Then TSA carries out the algorithm $reviewDAC$ because a value corresponding to the variables involved in the $Nogood$ message has been removed or bounded. To carry our this task, we must take into account two different cases:

- If the $Nogood$ message maintains only one variable (X_j) owned by the agent, then the assigned value of this variable must be removed and the algorithm reviewDAC must check the domains of the variables ranging between X_1 and X_j to ensure that these variables guarantee directional arc-consistency. It is not necessary to check directional arc-consistency for the rest of the variables due to the fact that the value removed from variable X_j does not affect to the rest of the variables.
- If the $Nogood$ message maintains more than two variables, the current value of variable X_j (the variable with the highest order, according to d, included in the $Nogood$ message) will not be removed. However it will be labelled as bounded by the value of variable X_s (the second variable with the highest order, according to d, involved in the $Nogood$ message). In this case, the algorithm reviewDAC must check the domains of variables ranging between X_j and X_s since when to as X_s changes its current value, the value of X_j that is bounded by X_s will be included in the domain again.

The algorithm reviewDAC only checks directional arc-consistency in the branch of the tree that is involved in the removal of a value of a variable, since this does not affect any other branch of the tree.

The following step in the search process is to backtrack to the highest variable involved in the $Nogood$ message X_j. To this end, TSA searches the first variable with no empty domain from X_j, in decreasing order. If no variable is found, then TSA returns $NO-SOLUTION$. Otherwise, if a variable X_k has no empty domain, then TSA unassigns the value of all the variables that are higher than X_k according to d (X_k included). Then, TSA calls algorithm $assignValue$ to assign values to variables from X_k to X_n. To prevent the new solution generating the same $Nogood$ message, TSA searches for a solution with different value assignments in, at least, one of the variables involved in the previous $Nogood$ message.

Next, we show an example of TSA execution. It is based on the DCSP shown in Figure 4. This figure shows a CSP divided into two sub-problems with tree structure.

According to the DTS algorithm and TSA, the process starts with the search for the first *partial_solution* in sub-problem 1. First, the TSA carries out a directional arc-consistency according to ordering d={X_1,X_2,X_3,X_4,X_5,X_6}. This action prunes the domains of the variables X_1 and X_4 as shown in Figure 4. Then, sub-problem 1 has the *partial_solutions* shown in Figure 5.

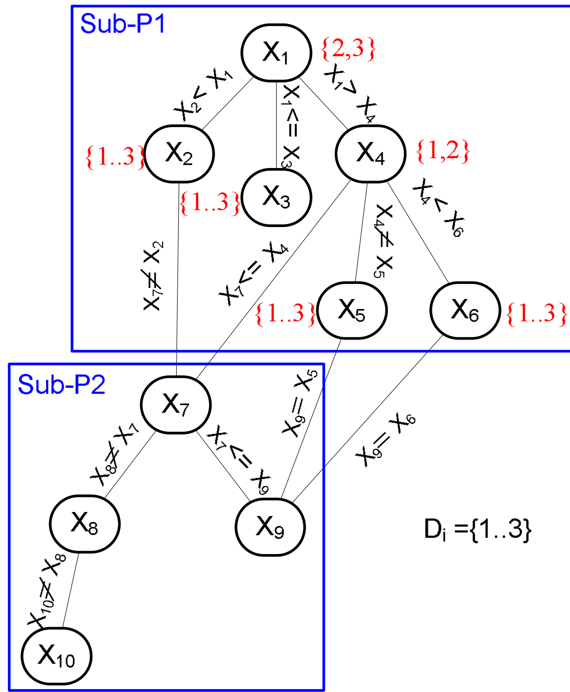

Figure 4: Example of DCSP with tree structure.

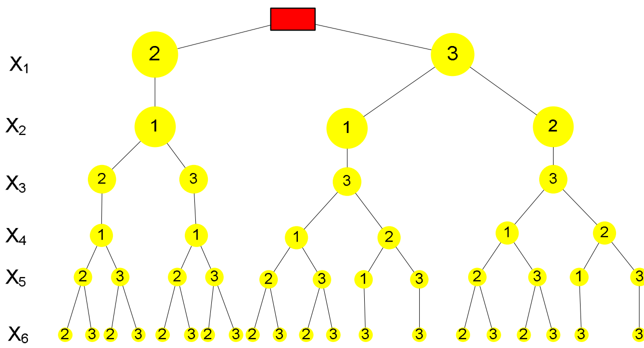

Figure 5: Set of valid *partial_solutions* of Sub-P1 (Figure 4) after directional arc-consistency.

Next, TSA assigns values to its variables. The first *partial_solution* included in the set of valid *partial_solutions* shown in Figure 5 is (X_1=2, X_2=1,X_3=2,X_4=1,X_5=2,X_6=2). This first *partial_solution* is inconsistent with sub-problem 2 since assignments X_2=1 and X_4=1 empty the domain of X_7. Therefore, the agents that owns sub-problem 2 sends the *Nogood* message: (X_2=1, X_4=1 to the other agent.).

When the agent that owns sub-problem 1 gets the *Nogood* message (X_2=1, X_4=1), it marks the value of X_4 restricted by X_2. It causes the pruning of seven *partial_solutions* that are valid for sub-problem 1 but which are inconsistent with the global problem (see Figure 6). Next, this agent marks the variables X_4, X_2 and X_1 as *varNogood* and it searches for a *varNogood* variable that has an available value beginning with X_4. The found variable is X_1. Then this agent assigns new values beginning with X_1 and it finds a new *partial_solution*: X_1=3, X_2=1,X_3=3,X_4=1,X_5=2,X_6=2 (Figure 7). This *partial_solution* does not change any value of the variables in *Nogood* message. Therefore, TSA repeats the same process again: it marks the value of X_4 restricted by X_2. Now, this causes the pruning of three *partial_solutions* (see Figure 7). TSA again marks the variables X_4, X_2 and X_1 as *varNogood*. In this case, variable X_4 is the first *varNogood* variable with available values. Thus, TSA assigns new values beginning with X_4 and it obtains the *partial_solution*: X_1=3, X_2=1,X_3=3,X_4=2,X_5=1,X_6=3.

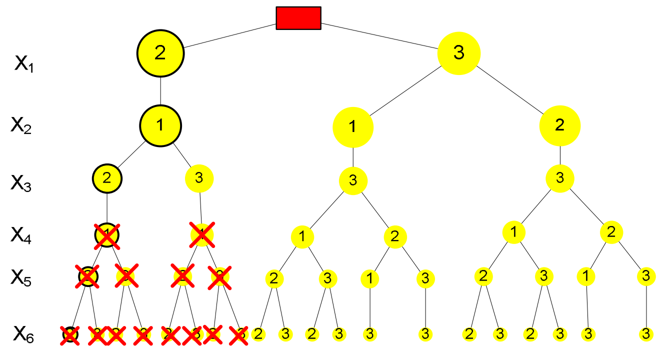

Figure 6: TSA pruning with *Nogood*-message (X_2=1,X_4=1).

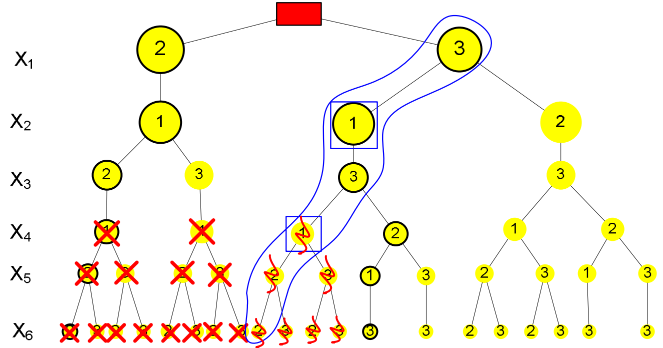

Figure 7: Second TSA pruning with *Nogood*-message (X_2=1, X_4=1).

Again, the agent that owns sub-problem 2 sends a *Nogood* message (X_5=1, X_6=3) since the last *partial_solution* of sub-problem 1 empties the domain of X_9. When the agent that owns sub-problem 1 gets this *Nogood* message, it marks the value of X_6 restricted by X_5, and it marks the variables X_5, X_6 and their ancestors as *varNogood*. Beginning with X_6, the first variable with available values is X_5. The value reassignment, beginning with X_5, obtains the *partial_solution*: X_1=3, X_2=1,X_3=3,X_4=2,X_5=3,X_6=3 (see Figure 8).

Finally, the last *partial_solution* of sub-problem 1 is consistent with sub-problem 2, for example with the assign-

ment: (X_7=2, X_8=1, X_9=3, X_{10}=2). Then, we have a global solution and the DTS algorithm finishes.

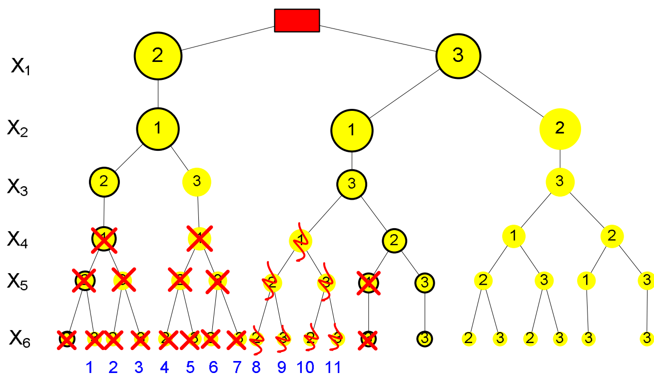

Figure 8: TSA pruning with *Nogood*-message (X_5=1,X_6=3).

Figure 8 shows that thanks to the *Nogood* messages, TSA has pruned eleven *partial_solutions* of sub-problem 1 that are inconsistent with sub-problem 2.

Railway Scheduling Problem

Train timetabling is a difficult and time-consuming task, particularly in the case of real networks where the number of constraints and the complexity of constraints grow drastically. A feasible train timetable should specify the departure and arrival time of each train to each location of its journey, in such a way that the line capacity and other operational constraints are taken into account. Traditionally, train timetables are generated manually by drawing trains on a time-distance graph called a running-map. The train schedule is generated from a given starting time and is manually adjusted so that all constraints are met. High priority trains are usually placed first followed by lower priority trains. It can take many days to develop train timetables for a line, and the process usually stops once a feasible timetable has been found. The resulting plan of this procedure may be far from optimal.

A sample of a running map is shown in Figure 9, where several train crossings can be observed. A running map contains information regarding railway topology (stations, tracks, distances between stations, traffic control features, etc.) and the schedules of the trains that use this topology (arrival and departure times of trains at each station, frequency, stops, crossings, etc,). The names of the stations are presented on the left side of Figure 9, and the vertical line represents the number of tracks between stations (one-way or two-way). The horizontal line represents the time.

The literature of the 1960s, 1970s, and 1980s related to rail optimization was relatively limited. Compared to the airline and bus industries, optimization was generally overlooked in favor of simulation or heuristic-based methods. However, (Cordeau, Toth, & Vigo 1998) point out greater competition, privatization, deregulation, and increasing computer speed as reasons for the more prevalent use of optimization techniques in the railway industry. Our review

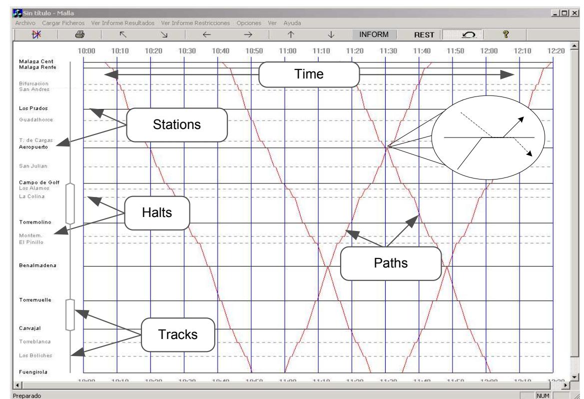

Figure 9: Example of a running-map

of the methods and models that have been published indicates that the majority of authors use models that are based on the Periodic Event Scheduling Problem (PESP) introduced by (Serafini & Ukovich 1989). The PESP considers the problem of scheduling as a set of periodically recurring events under periodic time-window constraints. The model generates disjunctive constraints that may cause the exponential growth of the computational complexity of the problem depending on its size. (Schrijver & Steenbeek 1994) have developed CADANS, a constraint programming- based algorithm to find a feasible timetable for a set of PESP constraints. The scenario considered by this tool is different from the scenario that we have used in this work; therefore, the results are not easily comparable. The train scheduling problem can also be modeled as a special case of the job-shop scheduling problem ((Silva de Oliveira 2001), (Walker & Ryan 2005)), where train trips are considered jobs that are scheduled on tracks that are regarded as resources.

Our goal is to model the railway scheduling problem as a Constraint Satisfaction Problem (CSP) and to solve it using constraint programming techniques. However, due to the huge number of variables and constraints that this problem generates, a distributed model is developed to distribute the resultant CSP into semi-independent sub-problems so that the solution can be found efficiently.

Variables and Constraints in the Railway Scheduling Problem

The variables of the railway scheduling problem are the arrival and departure times of trains at stations. The domain of the variables is the time with a granularity of minutes. There are three groups of scheduling rules in our railway scheduling problem: traffic rules, user requirements rules and topological rules. A valid running map must satisfy the above rules. These scheduling rules can be modelled using the following constraints, where variable $TA_{i,k}$ represents that train i arrives at station k and the variable $TD_{i,k}$ means that train i departs from station k:

1. **Traffic rules** guarantee crossing and overtaking operations. We assume two trains (i and j) going in opposite

directions between stations k and $k+1$. The main constraints to take into account are:

- ***Reception time constraint***. There exists a given time to detour a train back from the main track so that crossing or overtaking can be performed (RecT).

$$(TA_{i,k} + RecT_i < TA_{j,k}) \vee (TA_{j,k} + RecT_j < TA_{i,k})$$

- ***Expedition time constraint***. There exists a given time to put a train back on the main track so that crossing or overtaking can be performed (ExpT).

$$(TD_{i,k} + ExpT_i < TD_{j,k}) \vee (TD_{j,k} + expT_j < TD_{i,k})$$

- ***Crossing constraint***: Any two trains going in opposite directions must not simultaneously use the same one-way track.

$$(TD_{i,k} + T_{i,(k_k+1)} < TD_{j,k+1}) \wedge (TD_{i,k} < TD_{j,k+1} + T_{j,(k+1_k)})$$
$$\vee$$
$$(TD_{i,k} + T_{i,(k_k+1)} > TD_{j,k+1}) \wedge (TD_{i,k} > (TD_{j,k+1} + T_{j,(k+1_k)})$$

- ***Overtaking constraint***: Two trains (i and s) going at different speeds in the same direction can only overtake each other at stations.

$$(TD_{i,k} < TD_{s,k}) \wedge (TD_{i,k} + T_{i,(k_k+1)} < TD_{s,k} + T_{s,(k_k+1)})$$
$$\vee$$
$$(TD_{i,k} > TD_{j,k}) \wedge (TD_{i,k} + T_{i,(k_k+1)} > (TD_{s,k} + T_{s,(k_k+1)})$$

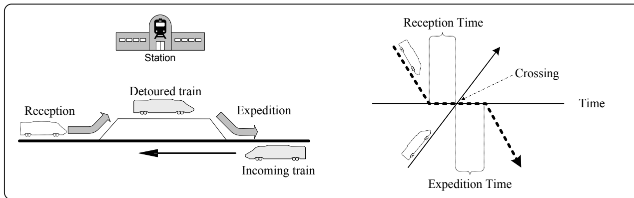

Figure 10: Constraints related to crossing and overtaking in stations

2. **User Requirements**: The main constraints due to user requirements are:

   - ***Type and Number of trains*** going in each direction to be scheduled.
   - ***Path of trains***: Locations used and ***Stop time*** for commercial purposes in each direction.

   $$TD_{i,k} = TA_{i,k} + StopTime_{i,k}$$

   - ***Scheduling frequency***. Train departure must satisfy frequency requirements in both directions. It could be a fixed time (1) or a time interval ($Freq \pm \delta$) (2). Frequency is a very tight constraint and is only sometimes required.

     (1) $TD_{i+1,k} = TD_{i,k} + Freq$

     (2) $(TD_{i,k} + Freq - \delta) \leq TD_{i+1,k} \leq (TD_{i,k} + Freq + \delta)$
   - ***Departure interval*** for the departure of the first trains going in both the up and down directions.

   $$StartTime_i < TD_{i,1} < EndTime_i$$

3. **Railway infrastructure topology and type of trains** to be scheduled give rise to other constraints to be taken into account. Some of them are:

   - ***Number of tracks in stations*** (to perform technical and/or commercial operations) and the ***number of tracks between two locations*** (one-way or two-way).
   - ***Time constraints***, between each two contiguous stations ($T_{i,(k_k+1)}$).

   $$TA_{i,k+1} - TD_{i,k} = T_{i,(k_k+1)}$$

Figure 11 shows the set of variables of two trains going in opposite directions between stations A and E. After studying the problem, we have detected an advantageous tree partition of the railway scheduling problem. In Figure 11, the edges between two variables represent time constraints ($TD_{i,k}$- $TA_{i,k+1}$) and stop time constraints ($TA_{i,k}$- $TD_{i,k}$), respectively; these constraints make up train paths and they could be private information of railway operators. Figure 12 shows a clear tree partition of the railway scheduling problem where each sub-CSP has all variables of only one train and the respective *intra-constraints* are time constraints and time stop constraints that are usually fixed by railway operators. The *inter-constraints* are the traffic rules that are usually controlled by infrastructure managers.

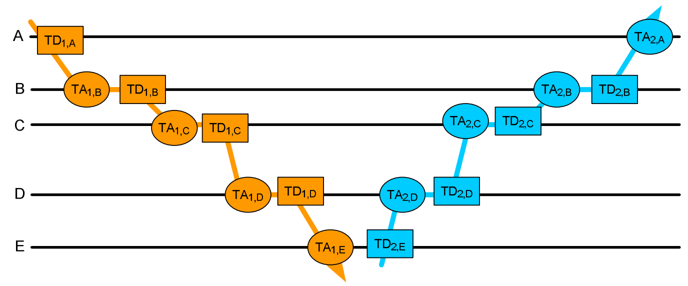

Figure 11: Variables of two opposite trains.

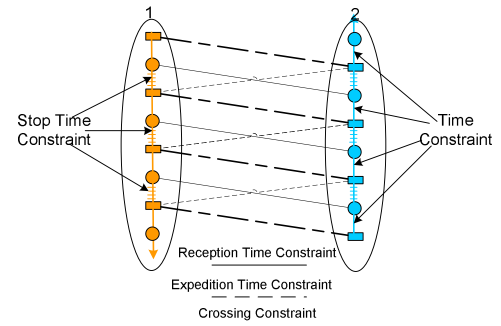

Figure 12: Train-based tree partition.

Evaluation

In this section, we carry out an evaluation between DTS and a centralized CSP solver. To this end, we have used a well-known centralized CSP solver called *Forward Checking* (FC). The classical binary version of FC ((Haralick & G. 1980)) has a prohibitive computational cost for the type of problems evaluated in this section (a simple railway scheduling problem with 4 trains could not be solved after 1 day of

execution); that is why we use the *full path consistency Forward Checking* algorithm (FCPath)[1]. This algorithm performs full path consistency on future variables whenever the current variable is instantiated.

This empirical evaluation of the railway scheduling problem was carried out over a real railway infrastructure that joins two Spanish cities (La Coruna and Vigo). The journey between these two cities is currently divided by 40 stations. In our empirical evaluation, each set of random instances was defined by the 3-tuple $< n; s; f >$, where n was the number of periodic trains in each direction, s the number of stations, and f the frequency (in minutes). The problems were randomly generated by modifying these parameters. Usually, increasing the number of trains involves a CSP with a greater number of variables; increasing the number of stations involves a CSP with a greater number of variables and a greater domain size; and decreasing the frequency involves increasing the problem tightness because the number of conflicts between trains is greater.

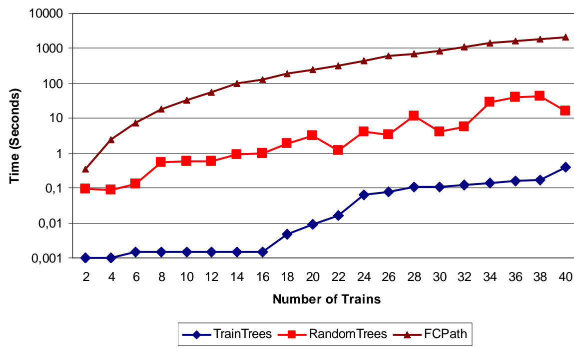

Figure 13: Running Times in problems $< n, 5, 60 >$.

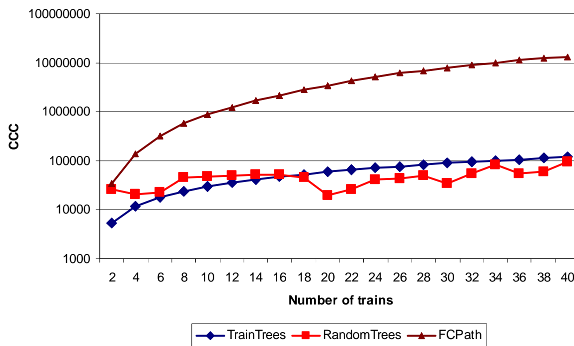

Figure 14: Concurrent constraint checks in problems $< n, 5, 60 >$.

In this evaluation, we compare the running time and *concurrent constraint checks* (CCC) ((Meisels *et al.* 2002)) of DTS with a well-known centralized algorithm: FCPath. DTS is executed over two different tree partitions: *random partition* is obtained using the *TreePartition* algorithm (Algorithm 1), which is a general tree partition method; *train partition* is a domain-dependent partition, which has been illustrated in Figure 12. It must be taken into account that both types of partitions roughly generate the same number of sub-problems and *inter-constraints*.

[1]FCPath was obtained from Van Beek page. It can be found in: http://ai.uwaterloo.ca/ vanbeek/software/software.html

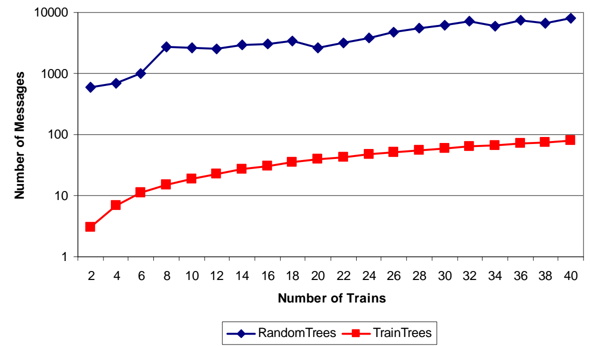

Figure 15: Message exchanges in problems $< n, 5, 60 >$.

Figures 13 and 14 show the behaviors of DTS and FCPath in several instances of n according to the tuple $< n, 5, 60 >$. The number of trains (n) was increased from 1 to 20 trains in each direction. It must be taken into account that both graphs maintain a $log10$ scale. Figures 13 and 14 show that DTS outperforms the FCPath algorithm, both *random partition* and *train partition*, in all instances. Figure 13 shows that DTS with *train partition* always has smaller running times than DTS with *random partition*. However, Figure 14 shows that DTS with *random partition* sometimes has fewer CCC than DTS with *train partition*. The last two assertions seem to be contradictory, but the explanation is in Figure 15: DTS with *train partition* exchanges fewer messages than DTS with *random partition*; thus, *train partition* saves a lot of running time. This is due to the fact that *train partition* involves better agent coordination.

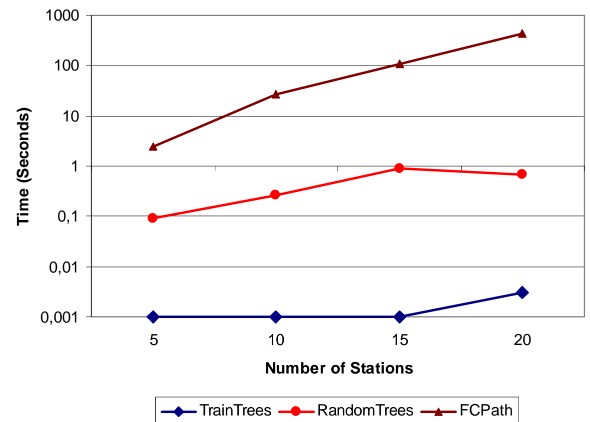

Figure 16: Running Times in problems $< 4, s, 60 >$.

Figure 16 shows the behaviors of DTS and FCPath in several instances of s according to the tuple $< 4, s, 60 >$, where the number of stations (s) was increased from 5 to 20. It can

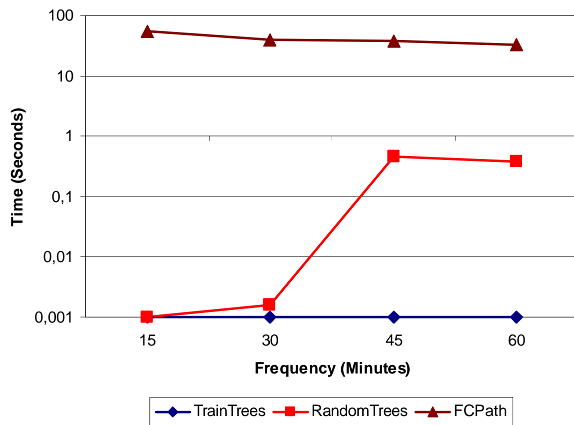

Figure 17: Running Times in problems $< 4, 10, f >$.

be observed that DTS with *train partition* has a better behavior than DTS with *random partition* and FCPath. In Figure 17, we evaluate the behavior of the algorithms with different frequencies according to the tuple $< 4, 10, f >$, where frequency was increased from 15 to 60 minutes. It can be observed that sice the problem instances have the same number of variables and domain size, both DTS with *train partition* and FCPath have homogeneous behaviors. In general, both graphs corroborate the good behavior of the DTS algorithm, particularly with *train partition*.

Conclusions

We have presented three techniques for structuring and solving binary CSPs. The first one translates, in polynomial time, the original binary graph into a *meta-tree CSP structure*, where each node in the *meta-tree CSP structure* is a tree. The second technique is a distributed algorithm (DTS) for solving the resultant *meta-tree CSP structure*. DTS exploits the linear complexity to solve each tree and minimizes the storage of Nogoods. The third technique is an intra-agent search algorithm. This algorithm takes into account the *Nogood_message* to prune the search space. These techniques have been applied to the railway scheduling problem. The evaluation shows that general distributed models have a better behavior than the centralized model and that domain-dependent distributed models are more efficient than general ones. Thus, this technique may be appropriate for solving centralized problems that can be divided into smaller subproblems.

Acknowledgements

This work has been partially supported by the research projects TIN2007-29666-E and TIN2007-67943-C02-01 (Min. de Educacion y Ciencia, Spain-FEDER), FOM-70022/T05 (Min. de Fomento, Spain), GV/2007/274 (Generalidad Valenciana) and by the Future and Emerging Technologies Unit of EC (IST priority - 6th FP), under contract no. FP6-021235-2 (project ARRIVAL).

References

Cordeau, J.; Toth, P.; and Vigo, D. 1998. A survey of optimization models for train routing and scheduling. *Transportation Science* 32:380–446.

Decher, R. 2003. *Constraint Processing*. Morgan Kaufmann.

Dechter, R., and Pearl, J. 1987. Network-based heuristics for constraint satisfaction problems. *Artificial Intelligence* 34:1–38.

Dechter, R. 1992. Constraint networks (survey). *Encyclopedia Artificial Intelligence* 276–285.

Freuder, E. 1982. A sufficient condition for backtrack-free search. *Journal of the ACM* 29:24–32.

Haralick, R., and G., E. 1980. Increasing tree efficiency for constraint satisfaction problems. *Artificial Intelligence* 14:263–314.

Meisels, A.; Kaplansky, E.; Razgon, I.; and Zivan, R. 2002. Comparing performance of distributed constraint processing algorithms. *In Proc. 4th Workshop on Distributed Constraint Reasoning*.

Miller, G. 1986. Finding small simple cycle separators for 2-connected planar graphs. *Journal of Computer and System Sciences* 32:265–279.

Salido, M., and Barber, F. 2006. Distributed CSPs by graph partitioning. *Applied Mathematics and Computation* 183:491–498.

Schloegel, K.; Karypis, G.; and Kumar, V. 2003. Graph partitioning for high-performance scientific simulations. *Sourcebook of parallel computing* 491–541.

Schrijver, A., and Steenbeek, A. 1994. Timetable construction for railned. *Technical Report, CWI, Amsterdam, The Netherlands*.

Serafini, P., and Ukovich, W. 1989. A mathematical model for periodic scheduling problems. *SIAM Journal on Discrete Mathematics* 550–581.

Silva de Oliveira, E. 2001. Solving single-track railway scheduling problem using constraint programming. *Phd Thesis. Univ. of Leeds, School of Computing*.

Walker, C., S. J., and Ryan, D. 2005. Simultaneous disruption recovery of a train timetable and crew roster in real time. *Comput. Oper. Res* 2077–2094.

Yokoo, M., and Hirayama, K. 2000. Algorithms for distributed constraint satisfaction: A review. *Autonomous Agents and Multi-Agent Systems* 3:185–207.