The Pairing Constraint for Round Robin Tournament Scheduling

Martin Henz [a] Tobias Müller [b] Tony Tan [a] Sven Thiel [c]

[a] School of Computing, National University of Singapore, Singapore 117543

[b] Programming Systems Lab, Saarland University, 66041 Saarbrücken, Germany

[c] Max-Planck-Institut für Informatik, Im Stadtwald, 66123 Saarbrücken, Germany

Abstract

At the heart of round robin tournament problems lies a constraint that groups teams into pairs of opponents. Arc-consistency for this pairing constraint can be reduced to finding in a graph all edges that are not covered by any maximum matching. We give an algorithm for this problem by adapting related techniques from graph matching and analyse its complexity. The performance of the resulting technique for round robin scheduling is examined on a variety of benchmarks. For typical tournament problems, it outperforms previous encodings by one order of magnitude.

Key words: tournament scheduling, constraint propagation, graph matching

1 Introduction

In round robin sport competitions, each team plays each other team once (single round robin) or several times (multiple round robin). Standard round robin tournaments are easy to schedule, since there are known algorithms, published tournament schedules and software that supports the tournament planner. However, in the presence of additional constraints, the round robin planning quickly degenerates into a combinatorial search problem. Techniques that have been used to solve these problems include integer programming [17,14], local search [20] and constraint programming [16,10,6]. Constraint programming has been shown recently to outperform operations research techniques on practical problems [7]. Constraint programming allows to systematically exploit the round robin and other constraints, often leading to relatively small search trees for medium-sized tournaments.

To solve large problems, stronger propagation algorithms for pruning the search trees become important. In finite domain programming systems such

Preprint submitted to Elsevier Preprint 5 May 2000

as Ilog Solver [9] and Mozart [12], where constraints are encoded as propagators that operate on a constraint store which stores domains of variables, arc-consistency is the strongest kind of propagation that can be achieved for a given constraint. A practical consideration in modeling of combinatorial search problems in these systems is the trade-off between the strength of propagation at each node of the search tree and its computation time. A good example for this is the all-different constraint, which requires that the values of a given set of finite domain variables are pairwise distinct. A naive propagation algorithm for this constraint is not stronger than an encoding using pairwise disequality constraints. Regin [15] shows how to employ an algorithm for matching in bipartite graphs to achieve arc-consistency with respect to one all-different constraint in $O(d \cdot n \cdot \sqrt{n})$, where n is the number of variables and d is the maximal cardinality of variable domains. The naive algorithm sometimes outperforms the arc-consistent one; in such cases, the decrease in the size of the search trees does not outweigh the increase of time that is spent at each of their nodes. This situation is common in constraint programming and necessitates an experimental evaluation for a given application domain. Regin's work is part of a research direction in constraint programming that identifies complex (so-called global) constraints that are relevant for applications, designs propagation algorithms for them and evaluates their performance on benchmark applications.

This work is part of that research program, where the application area is round robin tournament planning. A central constraint present in all constraint programming solutions confines the teams in a round to form pairs of opponents. So far, this constraint has been encoded using pairwise equality constraints similar to the naive propagation algorithm for all-different. We shall see that this approach does not achieve arc-consistency for pairing. Similar to Regin's approach, we reduce the problem of arc-consistency for pairing to a graph problem. Instead of matching in bipartite graphs, we need to consider matching in general graphs. More specifically, the problem of achieving arc-consistency for pairing is reduced to the problem of finding all edges in an undirected graph that do not belong to any maximum matching. Using ideas from [1], we employ Edmond's blossom-shrinking algorithm [3,4] to achieve a complexity of $O(d \cdot n^2 \cdot \alpha(n, d \cdot n))$ for the arc-consistency algorithm, where α is the inverse Ackermann-function.

After introducing basic notations in constraint satisfaction in Section 2, we show in Section 3, how round robin scheduling problems are represented as constraint satisfaction problems and discuss the strength of propagation of various encodings, motivating arc-consistent pairing. Section 4 presents a propagation algorithm for arc-consistent pairing, which relies on an algorithm for augmenting paths given in Section 5. Section 6 analyses the complexity of the propagation algorithm and Section 7 evaluates the performance of the alternative encodings on a variety of round robin benchmark problems.

2 Notations

A finite constraint satisfaction problem (CSP) $\mathcal{P}$ is a tuple (X, D, C), where X is a finite set of variables, and D assigns to each variable $x \in X$ a finite domain D_x of possible values. Each element c of C expresses a constraint on some variables $x_1, \ldots, x_k$, and thus $c \subseteq D_{x_1} \times \cdots \times D_{x_k}$.

A solution s to the constraint problem $\mathcal{P}$ assigns to each variable $x \in X$ a value $s_x \in D_x$ such that each constraint is satisfied. This means that for every constraint c on variables $x_1, \ldots, x_k$, $(s_{x_1}, \ldots, s_{x_k}) \in c$ holds. The set of all solutions to a constraint problem $\mathcal{P}$ is denoted by $sol(\mathcal{P})$.

The constraint programming approach to solving combinatorial search problems such as round robin scheduling problems is roughly as follows. Encode the problem as a constraint satisfaction problem $\mathcal{P}$, find a new problem $\mathcal{P}'$ that has the same set of solutions by applying so-called consistency techniques. Now augment $\mathcal{P}'$ in two ways, by adding a new constraint c' and its negation, respectively, to $\mathcal{P}'$. To the resulting problems $\mathcal{P}' + c'$ and $\mathcal{P}' + \neg c'$, apply again consistency techniques, find new constraints c'' for each of the problems, and so on. This process leads to a binary search tree at whose leaves are either problems that have no solution, or problems where D contains only singletons, which directly correspond to solutions.

There are many degrees of freedom in this process, including the original encoding of the problem, the consistency techniques to be applied, the choice of new constraints at each step and the order in which the resulting search tree is explored. The success of constraint programming relies on good choices for all these components. However, we are here mainly concerned with consistency techniques. For the other aspects of constraint-based round robin scheduling, see [6,7].

A propagation technique is a function that maps constraint problems $\mathcal{P} = (X, D, C)$ to new constraint problems $\mathcal{P}' = (X, D', C)$, where for every $x \in X$, ${D_x}' \subseteq D_x$, and where $sol(\mathcal{P}) = sol(\mathcal{P}')$.

A CSP is arc-consistent with respect to one of its constraints c on variables $x_1, \ldots, x_k$, if for each index $i \in \{1, \ldots, k\}$, and each value $v \in D_{x_i}$, there exists an element $(v_{x_1}, \ldots, v_{x_{i-1}}, v, v_{x_{i+1}}, \ldots, v_{x_k}) \in c$. A CSP is arc-consistent, if it is arc-consistent with respect to all of its constraints.

Arc-consistent propagation is a propagation technique that turns a given CSP into an arc-consistent CSP. It can be shown that arc-consistent propagation can be achieved by repeatedly achieving arc-consistency with respect to the given constraints.

3 Constraints for Round Robin Scheduling

The round robin scheme determines that every team plays against every other team a fixed number of times g during the competition. For simplicity of the discussion, let us assume $g = 1$, thus we are dealing with single round robin tournaments. A temporally dense single double round robin (DSRR) for n teams distributes the $n(n-1)$ matches over a minimal number of rounds such that every team plays at most one match per round. If n is even, the number of rounds is $2(n-1)$. A DSRR with an odd number of teams consists of $2n$ rounds in each of which $n-1$ teams play and one team does not. This team is said to have a bye. In the following, we are limiting ourselves to an even number of teams, since the problem for an odd number of teams $n-1$ can be reduced to the n case by introducing an additional team that always "plays" against the team with a bye.

The planning of a DSRR consists in assigning for each round an opponent team to each team. Often other decisions have to be taken as well, such as the place in which the matches are carried out. For intermural tournaments, this amounts to the question whether a team plays home or away, and for intramural tournaments, a court needs to be selected. Here, we concentrate on opponent team assignment, since these decisions are common to all round robin tournaments. Other decisions are discussed in Section 7.

The single round robin schedule on the right shows a valid assignment of opponent teams for $n = 6$ and $g = 1$ in each round. The value in row t and column r tells the team against which team t plays in round r.

		rounds				
		1	2	3	4	5
teams	1	2	4	6	3	5
	2	1	3	5	6	4
	3	5	2	4	1	6
	4	6	1	3	5	2
	5	3	6	2	4	1
	6	4	5	1	2	3

The simplest constraint satisfaction problem for opponent team assignment in DSRR represents the target timetable by an $n \times (n-1)$ matrix o of variables, whose variables $o_{t,r}$ tell the opponent team against which team t plays in round r. More formally, we define a DSRR problem as $\mathcal{P}_{DSRR} = (X, D, C)$, where X contains all variables in o and $D_{o_{t,r}} = \{1, \dots, n\}$ for every team t and round r. The set C contains the following constraints.

$$\textit{all-different}(o_{t,1}, \dots, o_{t,n-1}), \text{for every } t \in \{1, \dots, n\}, \text{ and} \tag{1}$$

$$\textit{pairing}(x_{1,r}, \dots, x_{n,r}), \text{for every } r \in \{1, \dots, n-1\}. \tag{2}$$

where the constraints *all-different* and *pairing* are defined as follows:

$$\begin{aligned} \textit{all-different}(x_1, \dots, x_m) = \quad & \{(v_1, \dots, v_m) \in D_{x_1} \times \cdots \times D_{x_m} | \\ & \qquad \forall_{i,j}. v_i \neq v_j\}, \qquad (3) \\ \textit{pairing}(x_1, \dots, x_m) = \quad & \{(v_1, \dots, v_m) \in D_{x_1} \times \cdots \times D_{x_m} | \\ & \qquad \forall_{i,j,i \neq j}. v_i \neq i \wedge v_i = j \leftrightarrow v_j = i\} \qquad (4) \end{aligned}$$

Observe that *pairing* is strictly stronger than *all-different* for $m \geq 2$;

$$\textit{pairing}(x_1, \dots, x_m) \subset \textit{all-different}(x_1, \dots, x_m)$$

It is important to note at this point that the choice of constraints is by far not obvious, even when the set of variables to model the problem is fixed. Although the set of solutions for different encodings must be the same, this choice has a decisive impact on the available propagation algorithms and thus on the constraint-based search process. For example, we may have decided to have one single constraint ($I = \{1\}$), which is the conjunction of all constraints above. This latter model is currently not particularly useful, since there is no known propagation algorithm for the conjunction of all constraints.

A propagation algorithm for arc-consistency of the all-different constraint has been presented by Regin [15]. This algorithm is highly effective in round robin scheduling. Previous approaches to constraint-based round robin tournament scheduling did not aim for arc-consistent propagation for the *pairing* constraint. Instead, they use a weaker form of propagation, where for each pairing constraint a half-matrix of auxiliary variables $r_{i,j}$ are introduced, where $1 \leq i < j \leq m$, whose domains are $D_{r_{i,j}} = \{0, 1\}$. The constraint *pairing*$(x_1, \dots, x_m)$ is then (following its definition) expressed as

$$\textit{neq}(x_i, i, 1) \text{ for } 1 \leq i \leq m \qquad (5)$$

$$\textit{eq}(x_i, j, r_{i,j}) \text{ for } 1 \leq i < j \leq m \qquad (6)$$

$$\textit{eq}(x_i, j, r_{j,i}) \text{ for } 1 \leq j < i \leq m \qquad (7)$$

The constraints *neq* and *eq* reflect the equality and disequality, respectively, of the first two arguments into the third argument, and thus are called reified constraints. Arc-consistent propagation with respect to these constraints is trivial.

The propagation behavior resulting from this approach, however, is strictly weaker than arc-consistent propagation for the pairing constraint.

Example 1 *For* $m = 6$, *let* $D_{x_1} = \{2, 3, 4, 5, 6\}$, $D_{x_2} = \{1, 2, 3, 6\}$, $D_{x_3} = \{1, 2, 4, 5, 6\}$, $D_{x_4} = \{1, 3, 6\}$, $D_{x_5} = \{1, 2, 3, 5\}$, $D_{x_6} = \{1, 2, 3, 5\}$. *Arc-consistency with respect to the neq constraints removes* 2 *from* D_{x_2} *and* 5 *from D_{x_5}, and arc-consistency with respect to the eq constraints removes* 6 *from D_{x_4},* 2 *from D_{x_5} and* 5 *from D_{x_6}. Arc-consistency with respect to the neq and eq constraints fails to reach arc-consistency with respect to the pairing constraint, which further removes* 2, 3 *and* 6 *from D_{x_1},* 1 *and* 3 *from D_{x_2},* 1, 2 *and* 6 *from D_{x_3}, and* 1 *and* 3 *from D_{x_6}.*

Adding the redundant constraint *all-different*$(x_1, \ldots, x_m)$ with arc-consistent propagation improves the propagation behavior in some cases. In Example 1, arc-consistent propagation with respect to this constraint and the *neq* and *eq* constraints achieves arc-consistency with respect to the *pairing* constraint. The next example shows that this is not always the case.

Example 2 *For $m = 6$, let $D_{x_1} = \{2, 3, 6\}$, $D_{x_2} = \{1, 6\}$, $D_{x_3} = \{1, 4, 5\}$, $D_{x_4} = \{3, 5\}$, $D_{x_5} = \{3, 4\}$, $D_{x_6} = \{1, 2\}$. Here, arc-consistency with respect to the neq and eq constraints and the all-different constraint is not able to remove any values from any domain, whereas arc-consistency with respect to the pairing constraint removes* 2 *and* 6 *from D_{x_1},* 1 *from D_{x_2},* 4 *and* 5 *from D_{x_3},* 3 *from D_{x_4},* 3 *from D_{x_5}, and* 1 *from D_{x_6}, thus fixing the pairing $1 - 3$, $2 - 6$ and $4 - 5$.*

We conclude from these examples that arc-consistent propagation for the pairing constraint deserves consideration and—assuming that there is an efficient algorithm for it—has the potential for improving round robin scheduling beyond the addition of an arc-consistent redundant all-different constraint.

4 Arc-Consistency for Pairing

Consider the constraint *pairing*$(x_1, \ldots, x_m)$. Suppose we impose the constraints 5, 6 and 7 from the previous section on the variables of the pairing constraint and perform arc-consistent propagation. Then the following holds

$$i \notin D_{x_i} \text{ for } 1 \leq i \leq m \quad \text{and} \quad i \in D_{x_j} \leftrightarrow j \in D_{x_i} \text{ for } 1 \leq i < j \leq m \qquad (8)$$

This means that the domains of the variables can be seen as the adjacency lists of an undirected graph $G = (V, E)$ which does not contain any self loops. We call this graph the *variable graph*; its vertex set is $V = \{x_1, \ldots, x_m\}$ and its edge set is $E = \{\{x_i, x_j\} \mid j \in D_{x_i}\}$. (Note that the latter is well-defined since constraint 8 holds.)

We want to give a graph-theoretical characterization of the solutions of our pairing constraint. For this aim we introduce some terminology. A set $M \subset E$ is called a *matching* in G, if no two distinct edges in M share a common endpoint. We call a vertex v *matched* by M if it is incident to some edge in M, and *free* otherwise. Let v be a matched vertex and let $e = \{v, w\} \in M$

denote the edge incident to v in the matching, then w is called the *mate* of v. A matching M is called *perfect* if it covers all vertices of G, i.e. there are no free vertices.

Looking at the definition of the pairing constraint, it is easy to derive a one-to-one correspondence between the solutions of the constraint and the perfect matchings in its variable graph G:

- Let $(v_1, \ldots, v_m) \in \textit{pairing}(x_1, \ldots, x_m)$ denote a solution, then the set $M = \{\{x_i, x_{v_i}\} \mid 1 \leq i \leq m\}$ is well-defined, a subset of E and a perfect matching in G.
- The matching M' corresponds to the solution $(v'_1, \ldots, v'_m)$ of the constraint where v'_i is the index of the mate of the node x_i in the matching M'.

This observations motivates the following definitions: We call an edge e of G *matchable* if there is a perfect matching M in G containing e and *unmatchable* otherwise.

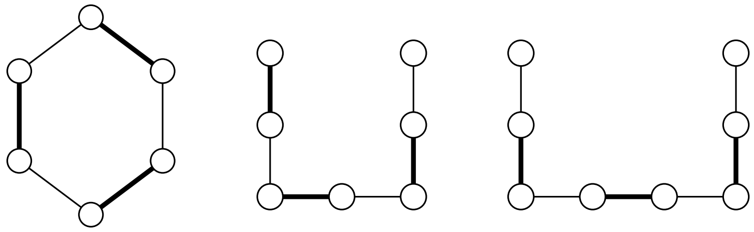
Lemma 1 *The following algorithm is a propagation technique for arc-consistent propagation with respect to the constraint pairing*$(x_1, \ldots, x_m)$*. Iterate over the domains* $D_{x_1}, \ldots, D_{x_m}$ *to enforce constraint 8 and construct the variable graph G for the constraint. For every pair of values i and j, such that* $\{x_i, x_j\}$ *is not a matchable edge in G, remove i from* D_{x_j} *and remove j from* D_{x_i}*.*

Proof: The algorithm is a propagation technique, i.e. it does not change the set of solutions. This holds because every solution corresponds to a perfect matching and no values corresponding to matchable edges are removed from the domains of the variables.
Now, we show arc-consistency: Let j be a value in the domain of x_i after termination of the algorithm. Then the edge $e = \{x_i, x_j\}$ is a matchable edge in the variable graph, and hence there is a perfect matching M containing e. The solution for pairing which corresponds to M has in its i-th component the value j. $\square$

Before we continue, we want to reconsider the approach to constraint-based round robin tournament scheduling, which used only the constraints 5, 6 and 7 and explain why this approach leads to weak propagation. As long as all domains of the variables contain more than one value after the propagation, even arc-consistent propagation only ensures that the domains of the variables encode an undirected loop-free graph G. But it does not remove any unmatchable edge from G.

It is easy to enforce constraint 8 and construct the variable graph $G = (V, E)$.

Fig. 1 Examples of alternating paths: Matching edges are drawn thicker than non-matching edges. Only the rightmost alternating path is augmenting.



Thus the remaining task is to find all unmatchable edges in G. This problem was presented in [1, Excercise 12.42]. In order to describe the algorithm we need some more terminology from matching theory. Consider a (not necessarily perfect) matching M in a graph G. A simple path $p = [e_0, \ldots, e_{k-1}]$ from a vertex v to a vertex w in G is called an *alternating* path with respect to M if:

- the edges in p are alternately in M and in $E \setminus M$
- if p is a cycle, i.e. $v = w$, then exactly one of e_0 and e_{k-1} is in M
- either e_0 is in M or v is free and either e_{k-1} is in M or w is free.

An alternating path is called *augmenting* if both endpoints are free. Figure 1 shows examples of alternating and augmenting paths.

The following lemma states some basic facts about alternating paths and matchings, it is a summary of the Lemmas 14 and 15 in [11, Section 7.6.2]:

Lemma 2 *Let p be an alternating path with respect to a matching M in a graph G and let M' denote an arbitrary matching in G. Then the following holds:*

- $M \oplus p = (M \setminus p) \cup (p \setminus M)$ *is also a matching.*
- $M \oplus M'$ *consists of alternating paths and alternating cycles.*
- *If $|M| < |M'|$ then there is at least one augmenting path in G with respect to M.*

The lemma implies that an augmenting path p with respect to M can be used to extend M, for the cardinality of the matching $M \oplus p$ is one larger than that of M, because $M \oplus p$ covers the same vertices as M plus the two free endpoints of p.

Now, we are ready to tackle our problem. Assume that we have an initial perfect matching M in $G = (V, E)$ and fix an edge $e = \{v, u\}$ in M. Let $G_e = (V, E \setminus \{e\})$ and $M_e = M \setminus \{e\}$. We have that M_e is a matching in G_e and that the only free vertices are v and u. Now consider an edge e' incident to v or u. If e' can belong to some perfect matching M' in G, then M' is

Fig. 2 Arc-consistent propagation algorithm for Pairing Constraint

```
M = some perfect matching of G
if 2 * card(M) ≠ card(V)
then
   fail // no perfect matching in G
end
forall edges e' ∈ E \ M {
  let v be a vertex incident to e' and u be its mate in M
  remove the edge e = {v, u} from M and G
  if there is no augmenting path w.r.t. M starting or ending with e'
  then
     remove e' from G
  end
  insert e in M and G again
}
```

also a perfect matching in G_e. By Lemma 2, the set $M \oplus M'$ contains an augmenting path with respect to M connecting v and w. Thus our algorithm can be outlined as shown in Figure 2.

The problem of finding an initial perfect matching can be solved with the blossom-shrink algorithm by Edmonds [3,4] in time $O(nm \cdot \alpha(m,n))$ with Gabow's implementation [5]. Here α denotes the so-called inverse Ackermann-function, which grows very slowly and can be considered constant in our applications. We give the details of finding augmenting paths in the next section, because they are relevant for the complexity analysis in Section 6.

5 Finding Augmenting Paths

Assume that we have a graph G and a matching M in G such that only two vertices u and v of G are free and all others are matched. We want to find augmenting paths between u and v. We modify the blossom-shrink algorithm such that it takes advantage of the fact that we already have a "nearly" perfect matching, i.e. only two free nodes. Our description is based on [11, Section 7.7].

We fix one of the two vertices as start vertex, say v, and grow a tree of alternating paths with root v. Every vertex of G can have one of the labels EVEN or ODD or be unlabeled. The algorithm maintains the following invariants:

- All vertices belonging to the tree rooted at v are labeled EVEN or ODD. A vertex w is labeled EVEN if the path from w to v in the tree contains an even number of matching edges, and it is labeled ODD otherwise.
- All other vertices are unlabeled. If a vertex is unlabeled then its mate is

also unlabeled, except for u which has no mate in the matching M.

To facilitate the description we call an EVEN/ODD labeled vertex *even/odd*. The root v of the tree is even and all the leaves of the tree are also even. The tree can only branch at even vertices, since the child of an odd vertex is the mate in the matching. The tree is extended by exploring an edge $e = \{x, y\}$ incident to an even vertex x. We distinguish the following cases:

(1) $y = u$:
We have found an augmenting path between v and u, namely the path consisting of the edge e and the path from x to v in the tree.

(2) y is unlabeled and $y \neq u$:
We extend the tree by making y the child of x and the mate of y the child of y. Thus y becomes odd and its mate becomes even.

(3) y is an odd node:
We have discovered another odd-length alternating path to y and do nothing.

(4) y is an even node:
We have discovered a so-called blossom (see Figure 3). Let b be the lowest common ancestor of x and y in the tree. Since only even nodes can have more than one child, b is even. The *blossom* is the subgraph which consists of the edge e and the tree paths from x to b and y to b. We call the edge e the *bridge* of the blossom and the vertex b the *base* of the blossom. The action we take is to *shrink* the blossom. This means we collapse the whole blossom into a single vertex which we identify with the base b. Or in other words, we obtain the shrunken graph G' from G by removing all nodes which belong to the blossom except for the base and for every edge $\{s, t\}$ which connects a vertex s in the blossom with a vertex t outside we insert the edge $\{b, t\}$. We shrink the tree of alternating paths in the same way.

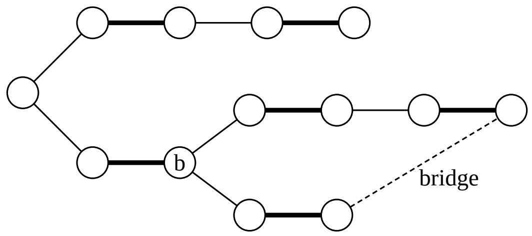
In case 1 we find an augmenting path and can terminate. In the other cases the invariants are preserved. For case 4 this follows immediately from the fact that all even nodes in the blossom are identified with an even node and that all tree edges incident to odd nodes of the blossom are matching edges and hence do not leave the blossom.

Let us look at the shrink operation more closely. The following lemma is stated as Lemma 22 in [11, Section 7.7.2]. We repeat the proof here, since it shows how augmenting paths in the shrunken graph translate to the original graph.

Lemma 3 *Let G' be graph obtained by shrinking a blossom in G. If G' contains an augmenting path then G also contains an augmenting path.*

Proof: Suppose G' contains an augmenting path p. If p avoids b there is

Fig. 3 Discovery of a blossom: The base b is marked and the bridge is drawn in dashed style.



nothing to show. Otherwise we break p at b into three pieces. Let $e' = \{b, y\}$ be the non-matching incident to b on p. We may assume w.l.o.g. that p has the form $p_1 \cdot [e'] \cdot p_2$ and that p_1 is either empty or ends with the matching edge incident to b. The edge e' is induced by some edge $e = \{x, y\}$ of the original graph G, where x is a node of the blossom. The crucial observation is that there is a (possibly empty) alternating path p_b from b to x in the blossom of even length. For the even nodes this is obvious and for the odd nodes it can be obtained by crossing the bridge of the blossom. The path $p_1 \cdot p_b \cdot [e] \cdot p_2$ is an augmenting path in G. $\square$

The algorithm terminates either when it finds an augmenting path or when the tree of alternating paths becomes stable, i.e. it cannot be grown anymore and there are no more blossoms to shrink. In [11, Section 7.7.2] it is shown that in the latter case the matching M in the original graph G is perfect, which implies that there is no augmenting path in G with respect to M.

6 Complexity Analysis

We now give some implementation details of the blossom-shrink algorithm and analyze its complexity. Then we analyze the complexity of the propagation algorithm for the pairing constraint (see Figure 2 at the end of Section 4) which removes all unmatchable edges from an undirected graph.

We want to be able to shrink blossoms without actually modifying the original graph. This can be achieved by maintaining an equivalence relation on the vertices of the original graph. Two vertices are equivalent, if and only if they belong to the same blossom. At the beginning every vertex forms an equivalence class of its own. When a blossom is discovered, we unite the classes of the vertices in the blossom to a new class which replaces the old classes. Every equivalence class has a canonical representative, we denote the canonical

representative of the class of a node x by $Base(x)$. As the name suggests we choose for every blossom its base as canonical representative (and for singleton classes the node itself). Thus an edge $\{x, y\}$ in the original graph corresponds to the edge $\{Base(x), Base(y)\}$ in the shrunken graph if $Base(x) \neq Base(y)$; in case $Base(x) = Base(y)$ the edge runs within the blossom.
In order to represent the equivalence relation we use a union-find data structure [19]. Let n denote the number of vertices and l denote the number of find (the base) operations. Then the total time required to maintain the relation and process the operations is $O((n + l) \cdot \alpha(l, n))$. The space requirement is $O(n)$.

We show how to implement the algorithm such that it explores every edge at most once. We use a breadth first search approach in order to grow the tree. We maintain a queue Q which contains all even vertices whose edges have not been explored yet. At the beginning we initialize Q with the root v of the tree of alternating paths. As long as Q is not empty, we pop the first vertex x from Q and examine all non-matching edges incident to x. If we grow the tree (cf. case 2), we append the newly discovered even vertex to Q. When we discover a blossom (cf. case 4), we update our equivalence relation and we add the odd vertices in the blossom to Q, because their incident non-matching edges have not been explored yet and in the shrunken graph they become incident to the base of the blossom. The edges of an even vertex in the blossom have already been explored or the vertex is still in the queue.

There is one more detail we have to discuss before we can talk about the complexity of the propagation algorithm of Figure 2. When we shrink a blossom, we have to find its base. Assume we have discovered the bridge $\{x, y\}$ and we want to find the base of the blossom, i.e. the lowest common ancestor of x and y in the tree. We can do this by walking up the paths from x and y to root of the tree in lock step, we mark each even node which we visit. We stop when we first hit a marked node, this is the base of the blossom. We unmark all marked nodes and collapse the blossom. Thus shrinking a blossom is linear in the size (number of edges) of the blossom. Since an edge in a blossom disappears after the blossom has been shrunken, the total time for shrinking all blossoms in a run of the algorithm can be bounded by $O(m \cdot \alpha(m, n))$, where m denotes the number of edges of the original graph. The same bound holds for the time needed to explore the edges and to grow the tree so that the time complexity of the algorithm is $O(m \cdot \alpha(m, n))$.

In the algorithm we presented at the end of Section 4, there is a subroutine which determines for an edge $e' = \{v, w\}$ if there is an augmenting path starting or ending with it. Clearly, if there is such a path then e' is the only edge incident to v on that path. So the straightforward way to answer the question is to remove all edges from G incident to v except for e' and then run the blossom-shrink algorithm. If it finds an augmenting path, then e' is matchable, otherwise it is unmatchable. (Of course, we have to restore the deleted edges after the blossom-shrink algorithm has terminated.) Since we have to call the blossom-shrink algorithm for every non-matching edge e, the complexity of the propagation algorithm in Figure 2 using this technique is $O(m^2 \cdot \alpha(m,n))$, and in terms of the maximal cardinality d of variable domains $O(n^2 \cdot d^2 \cdot \alpha(n \cdot d, n))$; its space requirement is $O(n+m)$, and $O(n \cdot d)$. This algorithm is called *pairing 1* in Section 7.

A hint in the exercise in [1] states that we can check all edges incident to v in a single run of the blossom-shrink algorithm. Now, let us look at another also straightforward way. We do not start our search for an augmenting path at the free node v but at the other free node u. Then v can never become labeled and hence can never belong to a blossom. Now, suppose that the algorithm finds an edge from an even vertex x to v, i.e. case 1 applies, then by Lemma 3 this edge stands for an edge $e = \{y, v\}$ in the original graph G and there is an augmenting path in G which ends with e. We remove e from the graph, store e in a list L and rerun the blossom-shrink algorithm. If we find another augmenting path, we do the same as before. We repeat this until we find no more augmenting path, then all edges which are still incident to v are unmatchable. We delete those edges and restore all edges in L.
The crucial observation is that we do not have to start from scratch after we have put an edge e into L, we can simply remove it from the graph and go on, pretending it has never been there. As the blossom-shrink algorithm has not seen e before, it would make exactly the same computations if we started it from scratch on the modified graph. So we can check the edges incident to a single vertex v with one call to the blossom-shrink algorithm. Thus the time complexity of the propagation algorithm now becomes the same as the complexity of finding its initial perfect matching, $O(n \cdot m \cdot \alpha(m,n))$, and $O(n^2 \cdot d \cdot \alpha(n \cdot d, n))$; its space requirement is $O(n+m)$, and $O(n \cdot d)$. This algorithm is called *pairing 2* in Section 7.

7 Experimental Evaluation

For the experimental evaluation, we use the programming system Mozart 1.1.0 [12] for the concurrent constraint language Oz [18], which provides extensive support for finite domain constraint programming. We implemented the propagation algorithms *pairing 1*, *pairing 2*, and Regin's algorithm for arc-consistent all-different using the LEDA library [11] and made them available to the Mozart system through Mozart's Constraint Propagator Interface [13]. The run times given in this section are always the average user time of five runs on a 256 MB 400MHz Pentium II PC running Linux. The coefficient of deviation (standard deviation / arithmetic mean) was always below 3 %.

Table 1 Benchmarks on unconstrained DSRR; n: number of teams; F: number of failures in the search tree; UT: user time; the ? symbols indicate that 30 minutes of runtime were exceeded

	neq / eq		*all-diff*		*pairing 1*		*pairing 2*	
n	F	UT	F	UT	F	UT	F	UT
6	0	0.030	0	0.048	0	0.020	0	0.020
10	3	0.428	1	0.558	0	0.236	0	0.220
14	62	3.18	0	4.35	0	1.29	0	1.13
18	20	5.82	0	8.86	0	5.70	0	4.37
22	675	81.3	2	25.0	0	17.3	0	12.7
26	?	?	2	57.1	0	44.0	0	32.0
30	?	?	0	122.	0	96.6	0	71.2
34	?	?	17	241.	0	221.	0	145.
38	?	?	5	429.	0	419.	0	272.
42	?	?	4	742.	0	755.	0	484.

The C++ and Oz source code for generating and running these benchmarks is available at [8]. The goals of the experimental evaluation are as follows:

- evaluate the usefulness of the arc-consistent pairing constraint for round robin applications by comparing the sizes of search trees and run times resulting from searches that use the new constraint with searches that use arc-consistent all-different or simply the encoding using *neq* and *eq*,
- investigate the range of round robin problems to which arc-consistent pairing provides advantages over other techniques, and
- evaluate the efficiency of the two implementations of arc-consistent pairing for practical applications.

Unconstrained Single Round Robin

Here, we compare the performance of arc-consistent pairing versus the encoding using *neq* and *eq* and the redundant all-different constraint. In this and all following benchmarks, we use constraint-based tree search by using constraints of the form $o_{t_1,r} = t_2$ as branching constraints c (see Section 2), which means that we enumerate the opponent variables. We order the variables round-wise. At each node, we enumerate the first variable that has a non-singleton domain. The value t_2 is always the smallest element of this domain. More sophisticated enumeration techniques such as first-fail do not improve the search. Table 1 compares the encodings using *neq / eq* (Section 3, formula 4), the addition of a redundant arc-consistent all-different constraint as discussed in Example 1, the simpler implementation of arc-consistent pairing *pairing 1*, and the more sophisticated algorithm *pairing 2* both discussed in discussed in Section 6.

Table 2 Benchmarks on tightly constrained DSRR; n: number of teams; S: number of solutions; F: number of failures in the search tree; UT: user time

			neq / eq		*all-diff*		*pairing 2*	
file	n	S	F	UT	F	UT	F	UT
`s_6_yes`	6	4	7	0.088	5	0.124	4	0.060
`s_8_yes`	8	5	47	0.540	24	0.606	10	0.200
`s_10_yes`	10	1	37	0.704	17	0.686	1	0.106
`s_12_yes`	12	1	3216	77.9	1452	64.1	179	6.26
`s_14_yes`	14	1	8407	242.	1328	75.3	527	20.4
`s_6_no`	6	0	4	0.048	4	0.066	4	0.022
`s_8_no`	8	0	13	0.172	12	0.246	6	0.086
`s_10_no`	10	0	20	0.530	13	0.646	6	0.168
`s_12_no`	12	0	241	6.64	111	5.37	25	0.794
`s_14_no`	14	0	537	16.7	182	10.9	69	2.54
`s_16_no`	16	0	1467	64.5	213	18.0	86	5.37
`s_18_no`	18	0	593	38.6	95	9.57	30	2.29
`s_20_no`	20	0	?	?	2755	314.	254	23.0
`d_6_yes`	6	4	21	0.066	4	0.084	2	0.020
`d_8_yes`	8	32	5776	19.2	2736	28.1	226	0.93
`d_10_yes`	10	2	76646	409.	38251	677.	5956	35.6

We observe that for these kinds of benchmarks, arc-consistent pairing achieves optimal propagation in a sense that there are no failures in the search trees, whereas the encoding using *neq / eq* requires search. From 26 teams onwards, this method fails to produce solutions within reasonable time. Arc-consistent all-different occasionally requires a bit of search, but the performance difference to arc-consistent pairing is not dramatic here. In all benchmarks, the method *pairing 2* is faster than *pairing 1* and therefore we concentrate on *pairing 2* in the remainder of this section.

Tightly Constrained Round Robin Tournaments

The next set of benchmarks looks at tightly constrained problems. We constrain the problems by randomly forbidding opponent teams, until there are very few or no solutions to the problem. Table 2 compares the three remaining propagation algorithms for finding all solutions, or proving unsatisfiability. The problems `s_*` are single round robin problems as described throughout the paper, whereas the problems `d_*` are double round robin problems. For the latter, we have instead of an all-different constraint per team (requiring that each team plays each other team once) constraints that force the number of occurrences of all other teams in each row of the o matrix to be 2. There is no restriction on the distance between first leg and return match.

Table 3 Benchmarks on intermural tournaments; n: number of teams; S: number of solutions; F: number of failures in the search tree; UT: user time

			neq / eq		*all-diff*		*pairing 2*	
file	n	S	F	UT	F	UT	F	UT
`i_8_yes`	8	7	0	0.138	0	0.182	0	0.142
`i_12_yes`	12	3	0	0.440	0	0.510	0	0.436
`i_16_yes`	16	4	6	1.30	4	1.51	4	1.22
`i_20_yes`	20	10	35	4.23	22	4.63	22	3.61
`acc97/98`	9	179	273	19.5	273	28.4	268	22.1

Although the results vary considerably, we note that in these tightly constrained problems, arc-consistent pairing results in a reduction of the size of search trees by a factor of up to 10, and of the runtime by even more. The difference between the two techniques increases with the problem size. Note that for the double round robin problems, the reduction of the search tree afforded by the redundant all-different constraint does not justify its computational effort.

Intermural Tournaments

The intermural benchmarks in this section show that the pruning obtained from arc-consistent pairing not always outweighs its computational effort. The best strategy for intermural tournament problems is to first find so-called pattern sets [2]. Table 3 shows the run times for finding all solutions of intermural tournament problems, all of which except the last one are randomly constrained as in the previous section. The last problem is the ACC 1997/98 problem [14,7], which is tightly constrained by a variety of conditions. Since we are not concerned with the computation of the pattern sets, we fix a particular pattern set for the benchmarks, except for the ACC 1997/98 problem. The numbers for this last problem include the effort for generating pattern sets, since the pairing constraint already achieves some additional pruning during pattern set generation.

We observe that the effort for neither arc-consistent pairing nor arc-consistent all-different is is justified for these intermural tournaments. The reason is that pattern sets already enforce pairing to such an extent that arc-consistent pairing achieves almost no additional pruning of the search trees.

Summary of Experimental Evaluation

As typical for such global constraints, it depends on the application, whether the saving from pruning the search tree with arc-consistent pairing outweighs its computational effort. This is clearly the case for unconstrained and tightly constrained single and multiple round robin tournaments. For the larger examples, we observe a typical reduction of the search tree and runtime by one order of magnitude. Intermural tournaments do not benefit much from the arc-consistent pairing constraint. The more sophisticated algorithm *pairing 2* is faster than *pairing 1* for all problems considered.

Acknowledgements

The authors would like to thank Ernst Althaus and Kurt Mehlhorn for many helpful discussions.

References

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network flows : theory, algorithms and applications.* Prentice Hall, Englewood Cliffs, NJ, 1993.

[2] W. O. Cain, Jr. The computer-assisted heuristic approach used to schedule the major league baseball clubs. In S. P. Ladany and R. E. Machol, editors, *Optimal Strategies in Sports*, number 5 in Studies in Management Science and Systems, pages 32–41. North-Holland Publishing Co., Amsterdam, New York, Oxford, 1977.

[3] J. Edmonds. Maximum matching and a ployhedron with 0,1-vertices. *Journal of Research of the National Bureau of Standards*, 69B:125–130, 1965.

[4] J. Edmonds. Paths, trees, and flowers. *Canadian Journal on Mathematics*, pages 449–467, 1965.

[5] H. N. Gabow. An Efficient Implementation of Edmonds' Algorithm for Maximum Matching on Graphs. *Journal of the ACM*, 23:221–234, 1976.

[6] M. Henz. Constraint-based round robin tournament planning. In D. De Schreye, editor, *Proceedings of the International Conference on Logic Programming*, pages 545–557, Las Cruces, New Mexico, 1999. The MIT Press, Cambridge, MA.

[7] M. Henz. Scheduling a major college basketball conference—revisited. *Operations Research*, 2000. to appear.

[8] M. Henz, T. Müller, and S. Thiel. Software and benchmarks for the pairing constraint. `http://www.comp.nus.edu.sg/~henz/projects/pairing`, 2000.

[9] ILOG Inc., Mountain View, CA 94043, USA, `http://www.ilog.com`. *ILOG Solver 4.0, Reference Manual*, 1997.

[10] K. McAloon, C. Tretkoff, and G. Wetzel. Sports league scheduling. In *Proceedings of the 1997 ILOG Optimization Suite International Users' Conference*, Paris, July 1997.

[11] K. Mehlhorn and S. Näher. *LEDA: a platform for combinatorial and geometric computing.* Cambridge University Press, Cambridge, November 1999.

[12] Mozart Consortium. The Mozart Programming System. Documentation and system available from `http://www.mozart-oz.org`, Programming Systems Lab, Saarbrücken, Swedish Institute of Computer Science, Stockholm, and Université catholique de Louvain, 1999.

[13] T. Müller. Constraint extensions tutorial. available from `http://www.mozart-oz.org`, Programming Systems Lab, Saarbrücken, Swedish Institute of Computer Science, Stockholm, and Université catholique de Louvain, 1999.

[14] G. L. Nemhauser and M. A. Trick. Scheduling a major college basketball conference. *Operations Research*, 46(1):1–8, 1998.

[15] J.-C. Régin. A filtering algorithm for constraints of difference in CSPs. In *Proceedings of the AAAI 12th National Conference on Artificial Intelligence*, pages 362–367. AAAI Press, 1994.

[16] A. Schaerf. Scheduling sport tournaments using constraint logic programming. *Constraints*, 4(1):43–65, 1999.

[17] J. A. M. Schreuder. Combinatorial aspects of construction of competition dutch professional football leagues. *Discrete Applied Mathematics*, 35:301–312, 1992.

[18] G. Smolka. The Oz programming model. In J. van Leeuwen, editor, *Computer Science Today*, Lecture Notes in Computer Science 1000, pages 324–343. Springer-Verlag, Berlin, 1995.

[19] R. Tarjan. Effeciency of a good but not linear set union algorithm. *Journal of the ACM*, 1:215–225, 1975.

[20] J. P. Walser. *Domain-Independent Local Search for Linear Integer Optimization.* PhD thesis, Universität des Saarlandes, D 66041 Saarbrücken, Germany, Aug. 1998.