An Ant Colony Optimization Approach to the Traveling Tournament Problem

David C. Uthus
University of Auckland
Private Bag 92019
Auckland, New Zealand
dave@cs.auckland.ac.nz

Patricia J. Riddle
University of Auckland
Private Bag 92019
Auckland, New Zealand
pat@cs.auckland.ac.nz

Hans W. Guesgen
Massey University
Private Bag 11222
Palmerston North
New Zealand
h.w.guesgen@massey.ac.nz

ABSTRACT

The traveling tournament problem has proven to be a difficult problem for the ant colony optimization metaheuristic, with past approaches showing poor results. This is due to the unusual problem structure and feasibility constraints. We present a new ant colony optimization approach to this problem, hybridizing it with a forward checking and conflict-directed backjumping algorithm while using pattern matching and other constraint satisfaction strategies. The approach improves on the performance of past ant colony optimization approaches, finding better quality solutions in shorter time, and exhibits results comparable to other state-of-the-art approaches.

Categories and Subject Descriptors

I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search—*heuristic methods, scheduling*

General Terms

Algorithms

Keywords

Ant colony optimization, constraint processing, traveling tournament problem

1. INTRODUCTION

The scheduling of a sports league is a difficult task. They often require complex schedules containing multiple feasibility constraints or objective goals, resulting in a hard combinatorial problem. For example, the traveling tournament problem (TTP)[10], an abstracted representation of the Major League Baseball schedule, has been shown to be challenging to solve. Optimal solutions have only been proven for the smallest few problem instances[17] while the best known solutions for larger problem instances have required large amounts of computation across multiple processors[19].

In this paper, we present a new approach for applying ant colony optimization (ACO)[9] to the TTP. Our approach incorporates Prosser's forward checking and conflict-directed backjumping algorithm (FC-CBJ)[15] while using ideas of pattern matching and other constraint satisfaction strategies to allow the ants to quickly construct solutions. Our approach is able to outperform past ACO approaches[3, 4] to this problem in terms of both solution quality and the time needed to find these solutions. In addition to this, it displays results comparable to other single-processor state-of-the-art approaches[1, 6], a first for any ACO approach.

2. TRAVELING TOURNAMENT PROBLEM

The TTP is a sports scheduling problem consisting of an even n teams with an associated $n \times n$ matrix of distances between teams. The objective of the problem is to create a double round robin tournament which minimizes the total summed travel distance amongst all the teams. Distances are calculated individually for each team's schedule, with each team starting at home prior to the first round and ending at home after the final round. There is no travel distance when a team plays consecutive games at home.

The TTP's double round robin tournament requires each team to play every other team twice, once at home and once away. The tournament consists of r rounds, with $r = 2 \cdot (n - 1)$, and each team must play once every round. Two additional constraints for the TTP are the *no_repeat* and *at_most* constraints. The *no_repeat* constraint restricts a team from playing the same team consecutively. The *at_most* constraint restricts the number of consecutive games that a team can play either at home or away to three.

There have been many different approaches to the TTP since its introduction. Some of the relevant approaches used integer and constraint programming[11], tabu search[6], simulated annealing[1, 13, 19], and ACO[3, 4]. Metaheuristics have found nearly all of the best known solutions. Most TPP problem sets have only been optimally solved for up to eight or ten teams[17], showing the difficulty of the TTP.

3. ANT COLONY OPTIMIZATION

ACO is a metaheuristic that draws its inspiration from real life ants and their ability to find a shortest path from a food source to their nest. ACO algorithms generally consist of a colony of artificial ants constructing solutions in a cyclic manner. During a cycle, each ant constructs a candidate solution. In some applications of ACO, the solutions are then improved using a local search heuristic or metaheuristic. At the end of the cycle, all solutions are evaluated, and either some or all of the ants update the pheromone matrix, which helps direct the ants in constructing future solutions.

So far, ACO has shown lackluster performance for the TTP. The first approach to this problem by Crauwels and Van Oudheusden[4] showed very poor results, even with its excessive running time. This was due to its reliance on only using backtracking search to handle the problem constraints. The backtracking search resulted in a lot of wasted time constructing solutions, allowing for few cycles to be performed. A second approach to this problem by Chen et al.[3] used ACO as a hyper-heuristic. It showed better results than the earlier ACO approach, but still had inferior results when compared to other state-of-the-art approaches.

3.1 Integrating ACO with Forward Checking and Conflicted-Directed Backjumping

One of the difficulties in constructing solutions for the TTP is that it is a constrained optimization problem. In order to apply ACO directly to the TTP, one is required to use constraint processing techniques. There have been different approaches in the past combining ACO and constraint processing techniques. Work by Meyer and Ernst[14] showed good results combining ACO and constraint propagation for job scheduling problems. As mentioned earlier, Crauwels and Van Oudheusden combined ACO with backtracking search for the TTP. But their results showed backtracking was not sufficient for this problem. For the single round robin maximum value problem, Uthus et al.[18] took the integration of ACO and backtracking further by combining ACO with backjumping search, allowing for faster solution construction.

In this paper, we leverage the ideas from these past approaches, looking forward with constraint propagation and moving backwards with backjumping search, by integrating ACO with FC-CBJ. This integration is called "ant colony optimization with forward checking and conflict-directed backjumping," or AFC. We refer to Dechter's pseudocode description[5] when describing AFC.

Algorithm 1 AFC

```
 1: procedure ConstructSolution($X, D, C$)
 2:     $i \leftarrow 1$
 3:     SelectVariable
 4:     $D'_i \leftarrow D_i$ for $1 \leq i \leq n$ // Initialize domains
 5:     $J_i \leftarrow \emptyset$ for $1 \leq i \leq n$ // Initialize conflict sets
 6:     while $1 \leq i \leq n$ do
 7:         $x_i \leftarrow$ SelectValueACO
 8:         if $x_i$ = null $\wedge$ #backjumps = limit then
 9:             RestartAnt
10:         else if $x_i$ = null then
11:             $i' \leftarrow i$
12:             $i \leftarrow$ latest index in $J_i$
13:             if SafeBackjump then
14:                 $J_i \leftarrow J_i \cup J_{i'} - \{x_i\}$
15:             Undo changes to $D'$ and $J$ from $i'$ to $i$
16:         else
17:             $i \leftarrow i + 1$
18:             SelectVariable
19:     return $X$
```

Algorithm 1 shows the AFC algorithm for constructing solutions. Using the same nomenclature as Dechter, it takes in a set of variables X, their associated domains D, and the set of constraints C. With the exception of unsafe backjumping, AFC follows Dechter for handling conflict sets, domains, and backjumping. In addition, a couple of lines are omitted which are inconsistent with Prosser's original algorithm. These two lines take place after line 18 and would have resulted in domains being inconsistent and conflict sets being erased. For more detailed pseudocode and definitions, we direct the reader to their works[5, 15].

We do take note that with AFC, should there be no feasible solutions, AFC would then infinitely loop through ant restarts. AFC is designed for problems with large, feasible solution spaces. For problems that have a small, or empty, feasible solution space, it would be better to use an ACO algorithm that is designed for these types of problems.

The key changes made when integrating ACO with FC-CBJ is a modification to the SelectValue method along with implementation of unsafe backjumping and ant restarts.

3.1.1 Value Selection

We have made a small modification to the original SelectValue method for FC-CBJ. We do not display the procedure due to space limitations, since only one line has changed. In the normal usage of FC-CBJ, values were chosen either randomly or using some sort of heuristic information[5]. With AFC, values are chosen using the probabilistic action choice rules used for ACO algorithms[9]. Following a candidate value being chosen, the algorithm performs the normal constraint propagation and updating of conflict sets. If no domains become empty, the procedure returns the candidate value. If any domains become empty, the procedure discards this value, undoes changes to domains and conflict sets, and tries choosing another value. Should the current domain become empty, the procedure returns null.

3.1.2 Unsafe Backjumping

An important requirement when using backjumping with complete algorithms is that it needs to make safe backjumps, otherwise some feasible solutions may be missed. Complete algorithms will either find a feasible solution or prove that there are no feasible solutions. With ACO, this is not required since ACO, in itself, is an incomplete algorithm. Because of this, we tried the possibility of using unsafe backjumping to improve the performance of solution construction.

Backjumping algorithms are considered safe in that they do not jump far enough back that they would miss any feasible solutions[5]. Unsafe backjumping, on the other hand, can jump far enough back that it may miss some feasible solutions. This happens when the algorithm backjumps to index i and then has to do a second backjump from index i to index j. The reason to use unsafe backjumps is that it is more apt to jump back further than safe backjumping. This helps the algorithm get out of an infeasible partial-solution faster so that it can continue constructing from a feasible partial-solution. To achieve the unsafe backjumps, line 14 in Algorithm 1 is omitted.

3.1.3 Ant Restarts

A second feature incorporated into AFC is ant restarts. After a certain number of backjumps have been done by an ant while constructing a solution, the ant will reset the solution and start from the beginning. The limit for the number of backjumps is set to $b \cdot n$, with b being a scalar variable and n the number of teams.

The ant restarts are designed to give the algorithm sufficient time to construct a solution, but limit the time in cases where excessive backjumping is taking place. Ant restarts are a general approach that can be used for any problem type, allowing us to avoid having to hardcode for every situation that may arise and cause excessive backjumping.

The ant restarts described here are similar in nature to other works by Meyer and Ernst[14], and Beck[2]. Meyer and Ernst, when combining ACO and constraint propagation, would use a death penalty should a variable's domain become empty. Beck's approach, which used a constructive search algorithm of original design, would restart after a certain number of backtracks. Their work differs in that they used a dynamic restart policy with a short restart schedule.

3.2 Applying AFC to the TTP

We now describe the specifics of applying AFC to the TTP. Our approach uses a combination of $\mathcal{MAX}-\mathcal{MIN}$ Ant System ($\mathcal{MM}$AS)[16] and Ant Colony System (ACS)[7] as the base ACO algorithm. From $\mathcal{MM}$AS, we use its pheromone limits and restarts. From ACS, we use its value selection rule. We do not use ACS's local pheromone updates during solution construction, since this would require additional overhead to undo when backtracking.

3.2.1 Variables and Values

In common with Crauwels and Van Oudheusden's ACO approach to the TTP, solutions are constructed one round at a time from round 1 to r. AFC chooses variables using dynamic variable ordering[5]. AFC picks a team, *team*1, for the current round, r_c, that has the least number of teams left that it can play. In cases where there is a tie, it starts from the team corresponding to an ant's number and chooses the first team that has the least number of teams to play. An ant's number is a number assigned to it at the beginning of each cycle. When the number of teams and ants is equal, this number corresponds to the order that the ants construct a solution. When they are not equal, the ants are given a random number in the range of $[1..n]$ at the beginning of each cycle. This ensures that the solution space is being adequately searched and one team is not being favored over others, which is similar to when ACS is applied to the traveling salesman problem[7]. Once all teams have been paired for the current round, AFC can then begin the next round.

Using the definition of teams and rounds as variables, values are the remaining unplayed teams in *team*1's domain for the current round. These unplayed teams are the teams *team*1 can play at home or away. The team that is chosen to play against *team*1 is labeled *team*2.

There is no specific rule for picking values designed for $\mathcal{MM}$AS. Two possibilities which can be used are ant system's (AS) random proportional rule[8, 16] and ACS's pseudorandom proportional rule[7]. When applying to the TTP, AS's rule is defined as:

$$p_{ijk} = \begin{cases} \frac{[\eta_{ijk}^{\alpha}\tau_{ijk}^{\beta}]}{\sum_{k \in \text{allowed}_k}[\eta_{ijk}^{\alpha}\tau_{ijk}^{\beta}]} & \text{if } j \in D_{ik} \\ 0 & \text{otherwise.} \end{cases} \qquad (1)$$

where p_{ijk} is the probability of choosing team i to play at home against team j during round k; D_{ik} is the domain of remaining, unplayed teams of team i during round k; τ_{ijk} is the pheromone values; η_{ijk} is the heuristic value; and α and β are scalar variables that determine the strength of the pheromone and heuristic values. The heuristic value, η_{ijk}, for the TTP is defined as $\eta_{ijk} = (d_{ijk})^{-1}$, with d_{ijk} representing the distance added to the schedule when team i plays at home against team j during round k.

ACS's rule is defined as:

$$s = \begin{cases} \text{argmax}_{j \in D_{ik}}\{[\eta_{ijk}^{\alpha}\tau_{ijk}^{\beta}]\}, & \text{if } q \leq q_0 \\ S, & \text{otherwise.} \end{cases} \qquad (2)$$

where s is the resulting choice, S is a random value chosen using the proportional function defined in Equation (1), q is a random value and q_0 is a constant.

In the end, a value was chosen from the variable's domain using ACS's pseudorandom proportional rule without heuristic information. This is accomplished by setting $\alpha \leftarrow 0$ and $\beta \leftarrow 1$. The reason for using ACS's rule without heuristic information is that the approach performed better when utilizing this rule, as seen later in Section 4.1.2.

After having selected a candidate value, AFC then performs constraint propagation. This involves removing *team*1 and *team*2 from the domains of other, unpaired teams for the current round; removing *team*1 and *team*2 from each others' domains in the future with regards to whom played at home and away; and removing the required teams from *team*1's and *team*2's domains in the next round to make sure the *at_most* and *no_repeat* constraints are not violated. Additionally, it applies pattern matching at this point.

3.2.2 Pattern Matching

We invent a technique of pattern matching to enhance constraint propagation for an individual team's schedule. This idea can be used by any heuristic applied to the TTP that constructs solutions using constraint propagation. The purpose of the pattern matching is to find patterns in the number of remaining home and away games, and to utilize these patterns to help reduce the conflicts created from the *at_most* constraint of the TTP. The patterns help make sure that the solution constructed up to round r_c is still feasible for the remaining rounds needing to be constructed. While these patterns will only rarely be applicable, it is generally these rare cases which cause the most thrashing, even when using unsafe backjumping.

There are four symbols used with these patterns. $\mathcal{H}$ and $\mathcal{A}$ represent a team being restricted to either playing at home or away respectively, $\mathcal{B}$ represents a team being able to play at home or away, and $\mathcal{U}$ means that it is not yet known which of these is the case. Letting the variable X represent any of the four symbols, $\mathcal{H} \cup \mathcal{A} = \mathcal{B}$, $\mathcal{U} \cup X = X$, and $\mathcal{B} \cup X = \mathcal{B}$. Additionally, $!\mathcal{B} = \mathcal{B}$, $!\mathcal{U} = \mathcal{U}$, $!\mathcal{H} = \mathcal{A}$, and $!\mathcal{A} = \mathcal{H}$.

Making Patterns.

The patterns are created at the beginning of the algorithm and stored for later use in solution construction. The algorithm creates these patterns itself through exhaustive search, as seen in Algorithm 2. The procedure begins by taking in a 3-dimensional matrix, P, to store the patterns. P's first two dimensions refer to the number of remaining home and away games and its third dimension refers to the rounds of the specific pattern. The notation P_{ha} refers to a pattern with h and a remaining number of home and away games respectively while P_{hai} refers to round i in the pattern at P_{ha}. The procedure also takes in a value κ, which is the maximal number of consecutive games allowed by the *at_most* constraint. The procedure then goes through an iterative process of creating patterns for all possible combinations of home and away games with respect to κ.

Algorithm 2 Pattern making

```
 1: procedure MakePatterns($P, \kappa$)
 2:     for $i \leftarrow 1, n$ do
 3:         for $j \leftarrow i \cdot \kappa + 1, n$ do
 4:             for $k \leftarrow 1, i + j$ do
 5:                 $P_{ijk} \leftarrow \mathcal{U}$
 6:             PatternMaker($P, V, i, j, i, j, 1$)
 7:             for $l \leftarrow 1, i + j$ do
 8:                 $P_{jil} \leftarrow !P_{ijl}$

 9: procedure PatternMaker($P, V, h, a, h_r, a_r, d$)
10:     if $h_r = 0 \wedge a_r = 0$ then
11:         for $i \leftarrow 1, h + a$ do
12:             $P_{hai} \leftarrow P_{hai} \cup V_i$
13:     if $h_r > 0 \wedge$ at_most = feasible then
14:         $V_d \leftarrow \mathcal{H}$
15:         PatternMaker($P, V, h, a, h_r - 1, a_r, d + 1$)
16:     if $a_r > 0 \wedge$ at_most = feasible then
17:         $V_d \leftarrow \mathcal{A}$
18:         PatternMaker($P, V, h, a, h_r, a_r - 1, d + 1$)
```

The actual pattern maker procedure is a recursive procedure, trying every possibility of home and away combinations across the $h + a$ rounds, with h being the number of remaining home games and a the number of remaining away games. It takes in a pattern matrix, P; a vector to hold the current pattern being designed, V; the originally remaining number of home and away games, h and a; the remaining number of those games during the recursive run of the algorithm, h_r and a_r; and the depth it is currently at in the recursive process, d. When creating these patterns, it takes into consideration the *at_most* constraint, which is done on lines 13 and 16. While h_r and a_r are both not equal to zero, the procedure will try both possibilities of adding either a home game or an away game to the current pattern in V. Once $h_r = a_r = 0$, it knows it was able to successfully create a pattern, and combines the pattern in V to the pattern currently stored at P_{ha}.

When the procedure is finished, it will have created a single pattern. For example, if $a = 2$ and $h = 1$, then this will result in the pattern $\{\mathcal{B}, \mathcal{B}, \mathcal{B}\}$, since it can create schedules of $\{\mathcal{H}, \mathcal{A}, \mathcal{A}\}$, $\{\mathcal{A}, \mathcal{H}, \mathcal{A}\}$, and $\{\mathcal{A}, \mathcal{A}, \mathcal{H}\}$. On the other hand, if $a = 5$ and $h = 1$, then this will result in the pattern $\{\mathcal{A}, \mathcal{A}, \mathcal{B}, \mathcal{B}, \mathcal{A}, \mathcal{A}\}$, since it can only create feasible schedules of $\{\mathcal{A}, \mathcal{A}, \mathcal{H}, \mathcal{A}, \mathcal{A}, \mathcal{A}\}$ and $\{\mathcal{A}, \mathcal{A}, \mathcal{A}, \mathcal{H}, \mathcal{A}, \mathcal{A}\}$.

Patterns can be mirrored for mirror values of home and away, as done on line 8. The algorithm creates the patterns for the cases when there are more away games than home games, and reflects the resulting pattern for when there is a larger number of home games. For example, with the pattern $\{\mathcal{A}, \mathcal{A}, \mathcal{B}, \mathcal{B}, \mathcal{A}, \mathcal{A}\}$ when $a = 5$ and $h = 1$, this turns into the pattern $\{\mathcal{H}, \mathcal{H}, \mathcal{B}, \mathcal{B}, \mathcal{H}, \mathcal{H}\}$ for $a = 1$ and $h = 5$.

As mentioned earlier, the patterns are made with respect to κ. They are only created when either the remaining number of home or away games is more than κ times larger than the other. This limits the patterns to situations where it will be able to propagate constraints, and helps speed up the creation of the set of patterns. This is possible due to the following theorem. Without loss of generality, we restrict ourselves to $h \leq a$ and leave the symmetric case of $a \leq h$ for the reader.

Lemma 1. *Let h and a be the number of remaining home and away games such that $h = 1$ and $h \leq a \leq \kappa$. All patterns that meet these characteristics will be composed of all $\mathcal{B}$s.*

Proof. Suppose to the contrary there exists a pattern with $h = 1$ and $h \leq a \leq \kappa$ that is not composed of all $\mathcal{B}$s. Then there is a value for a for which this is true. For every value of a, create all possible schedules by initially creating a schedule of the one $\mathcal{H}$ followed by a $\mathcal{A}$'s. Rotate the $\mathcal{H}$ right from one slot to the next until it is in the last slot:

$a = 1 : \mathcal{H}_1\mathcal{A}_1|\mathcal{A}_1\mathcal{H}_1$

$a = 2 : \mathcal{H}_1\mathcal{A}_1\mathcal{A}_2|\mathcal{A}_1\mathcal{H}_1\mathcal{A}_2|\mathcal{A}_1\mathcal{A}_2\mathcal{H}_1$

...

$a = \kappa : \mathcal{H}_1\mathcal{A}_1...\mathcal{A}_\kappa|\mathcal{A}_1\mathcal{H}_1...\mathcal{A}_\kappa|...|\mathcal{A}_1...\mathcal{H}_1\mathcal{A}_\kappa|\mathcal{A}_1...\mathcal{A}_\kappa\mathcal{H}_1$

For all values of a, the composition of the set of schedules will result in a team being able to play either at home or away for each round, thus resulting in a pattern of all $\mathcal{B}$s. This contradicts the assumption that there is a pattern with $h = 1$ and $h \leq a \leq \kappa$ that is not composed of all $\mathcal{B}$s. □

Lemma 2. *Let h and a be the number of remaining home and away games such that $h > 1$ and $h \leq a \leq h \cdot \kappa$. All patterns that meet these characteristics will be composed of all $\mathcal{B}$s.*

Proof. Suppose to the contrary there exists a pattern with $h > 1$ and $h \leq a \leq h \cdot \kappa$ that is not composed of all $\mathcal{B}$s. Let $c = \lceil \frac{a}{h} \rceil$ and $d = a - c \cdot (h - 1)$. If $d = 0$, create an initial schedule such that there are h sets composed of one $\mathcal{H}$ followed by c $\mathcal{A}$s. If $d \neq 0$, create an initial schedule such that there are $h - 1$ sets composed of one $\mathcal{H}$ followed by c $\mathcal{A}$s with a final set of the last $\mathcal{H}$ followed by the final d $\mathcal{A}$s.

Beginning with the first set, enumerate from there a set of feasible schedules by rotating the $\mathcal{H}$ through the set of $\mathcal{A}$s one slot at a time as done in Lemma 1. Once the first set is done, continue on with the next set and do so until all sets have had their $\mathcal{H}$ rotated through their $\mathcal{A}$s. Due to the rotating of the $\mathcal{H}$s and $\mathcal{A}$s into every slot at some point in time, the composition of all the schedules will result in the possibility of a team playing either at home or away for every slot. This is a result of Lemma 1. Each set created can be treated as a case of Lemma 1, and the rotating will lead to that set being a set of all $\mathcal{B}$s. This will then result in a full pattern of all $\mathcal{B}$s. This contradicts the assumption that there is a pattern with $h > 1$ and $h \leq a \leq h \cdot \kappa$ that is not composed of all $\mathcal{B}$s. □

Theorem 1. *Let h and a be the number of remaining home and away games. Then the smallest ratio of $h : a$ games in which a pattern will be created that is not all $\mathcal{B}$s is greater than $1 : \kappa$.*

Proof. Suppose to the contrary there exists a pattern with a ratio smaller than or equal to $1 : \kappa$ that is not composed of all $\mathcal{B}$s. Then there is a value for h and a for which this is true. If $h = 1$, then $h \leq a \leq \kappa$ and this will create a pattern of all $\mathcal{B}$s due to Lemma 1. If $h > 1$, then $h \leq a \leq h \cdot \kappa$ and this will also create a pattern of all $\mathcal{B}$s due to Lemma 2. Since in both cases every pattern will be composed of all $\mathcal{B}$s, this contradicts the assumption that there is a pattern with the ratio smaller than or equal to $1 : \kappa$ that is not composed of all $\mathcal{B}$s. □

If the pattern making procedure is unable to make a pattern with h and a having values that are impossible due to the *at_most* constraint, then the pattern in P_{ha} would be composed of all $\mathcal{U}$s. This is not a concern since this pattern will never be applied as long as the patterns are properly applied during constraint propagation. The patterns ensure that a partially-constructed schedule will never have a set of values of h and a for a team that would make it impossible for the team to construct an individual, feasible schedule.

Using Patterns.

Algorithm 3 shows how the patterns are applied during constraint propagation. The procedure takes in P, the team t, the number of remaining home games h, the remaining away games a, and the current round r_c. It first checks if there is a pattern with the corresponding values of h and a. If so, it will then go through the domains of t for all rounds following r_c. For each round, it checks if the symbol for the pattern is $\mathcal{B}$ or not. If it is $\mathcal{B}$, it does nothing for that round. If it is a $\mathcal{H}$ or $\mathcal{A}$, it will then remove all remaining away or home games from the domain for that round, and remove t from the opponents who are removed from its domain. It is important to note that patterns cannot undo a pattern applied during a previous round; they only further reduce the domains of future rounds.

Algorithm 3 Applying patterns for constraint propagation

```
 1: procedure ApplyPatterns($P, t, h, a, r_c$)
 2:     if $P_{ha}$ exists then
 3:         for $i \leftarrow 1, h + a$ do
 4:             $r \leftarrow r_c + i$
 5:             if $P_{hai} = \mathcal{H}$ then
 6:                 for $j \leftarrow 1, n$ do
 7:                     if $@j \in D_{tr}$ then
 8:                         $D_{tr} \leftarrow D_{tr} \setminus @j$
 9:                         $D_{jr} \leftarrow D_{jr} \setminus t$
10:             else if $P_{hai} = \mathcal{A}$ then
11:                 for $j \leftarrow 1, n$ do
12:                     if $j \in D_{tr}$ then
13:                         $D_{tr} \leftarrow D_{tr} \setminus j$
14:                         $D_{jr} \leftarrow D_{jr} \setminus @t$
```

An example of using a pattern is as follows. Assume for a team, the domains of the last five rounds are composed of:

$$\{\{+1, -2, -4\}\{-2\}\{+1, -5\}\{-2, -3\}\{+1\}\}$$

Positive values are teams it plays at home, negative values are teams it plays away. Since it has four away games and one home game left to play, this results in the pattern $\{\mathcal{A}, \mathcal{B}, \mathcal{B}, \mathcal{B}, \mathcal{A}\}$. Applying this reduces the domains to:

$$\{\{-2, -4\}\{-2\}\{+1, -5\}\{-2, -3\}\{\}\}$$

As seen, the pattern has caused the last domain to become empty, thus the algorithm knows that the *at_most* constraint will be violated with the current partial solution.

3.2.3 Pheromone

One of the central ideas of ACO is the pheromone matrix, which influences the ants' decisions when constructing solutions. For the TTP, pheromone is defined as τ_{ijk}, which is the desirability of team i playing at home against team j during round k. We follow $\mathcal{MM}$AS's standard usage of the pheromone limits of τ_{max} and τ_{min} along with pheromone reinitialization. The pheromone is reinitialized to τ_{max} once c cycles have passed since both the best known solution was last improved and the previous pheromone reinitialization.

When updating the pheromone matrix, the algorithm uses different ants for different cycles. Prior to pheromone reinitialization, it alternates between the iterative best ant and global best ant. The iterative best ant is the best ant for the current cycle while the global best ant is the best ant since the beginning. After pheromone reinitialization, the algorithm alternates between the iterative best ant and restart best ant. The restart best ant is the best ant since the previous pheromone reinitialization. This differs from many pheromone update schedules of other ACO approaches to other problems, which use the global best ant throughout the running of the algorithm. As seen later in Section 4.1.3, the pheromone update schedule described was found to be the best schedule for this approach.

3.2.4 Local Search

The algorithm uses a general tabu search (TS)[12] approach when applying local search. It uses the neighborhood definitions described by one of the simulated annealing approaches, traveling tournament simulated annealing (TTSA)[1], taking into account the overlap of neighborhoods as described by the TS approach, composite-neighborhood tabu search (CNTS)[6]. TTSA's definitions are used since they are easier to check for feasibility and require less overhead. Secondly, this allows the tabu search approach to be as general as possible since CNTS uses specialized neighborhood definitions.

Unlike these other approaches to the TTP, the algorithm only searches in the neighborhood of feasible solutions. Exploring infeasible solutions is important for other approaches because it is currently unknown if the solution space of feasible solutions is fully connected[6]. This is not a concern here since TS is only being used for short runs from different starting solutions.

One of the problems of using TS for local search is that it is very slow in exploring the whole neighborhood, especially in the TTP. To mitigate this, TS uses an elite candidate list[12]. It creates a master list of length $4 \cdot n$ of the best ranked moves. With each iteration, the moves in the master list are re-evaluated to make sure they are still feasible, and to re-rank them. This list is remade every $\frac{n}{2}$ iterations, or when there is no feasible, un-tabood moves left in the master list that improves the solution. We note these parameters for the master list were chosen arbitrarily.

4. TESTS AND RESULTS

We ran two sets of experiments. The first was to look at the various aspects of our approach: how the different ideas improve the performance for AFC; the best ways of choosing values; and the best way of scheduling pheromone updates. The second set consisted of timed tests in order to compare our approach with other approaches to the TTP.

Unless otherwise specified, we ran our experiments with 5 ants, $20 \cdot n$ cycles until pheromone reinitialization after the last improvement to the best known solution, $q_0 = .9$, and the standard definitions of τ_{max} and τ_{min}[16]. We apply the local search on all the ants for $5 \cdot n \cdot (n-1)$ iterations each, with $n \cdot (n-1)$ representing the number of pairings in a schedule. We use a pheromone decay rate of 0.8. All

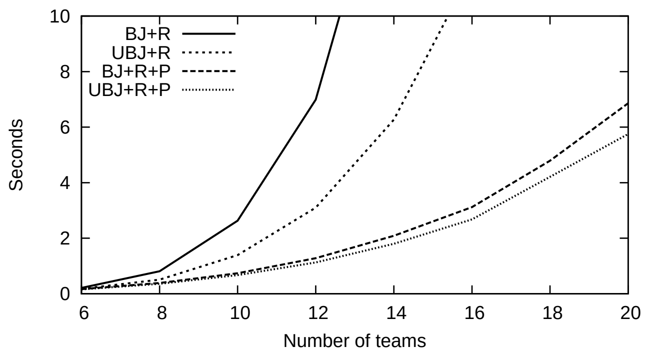

Figure 1: Comparison of using backjumping(BJ) and unsafe backjumping(UBJ) with and without ant restarts(+R) and pattern matching(+P). Timings represent the number of seconds for one ant to construct 1000 solutions averaged over 100 trials.

tests in this paper were run using single cores of Intel Dual Core Processors running at 2.13Ghz. We list all timings in seconds. All NL and CIRC instances used are from the Challenge Traveling Tournament Instances webpage[17].

4.1 Configuration

We tested three interesting aspects. The first section describes our testing of the different components of AFC, including the usage of pattern matching. The next two sections deal with testing ACO itself. The first describes how to choose a value: whether to use AS's or ACS's rules and whether or not to use heuristic information. The second describes the pheromone update schedule and which ant to use for the updates.

4.1.1 AFC

Three sets of tests were run to look at how unsafe backjumping, ant restarts, and pattern matching impact AFC's performance when it is applied to the TTP. The first set of tests looked at how the components influence the time needed for constructing a solution. These tests were done on team set sizes of 6 to 20. Each test was run by having one ant construct 1000 solutions, with timings averaged over 100 runs. Pheromone, heuristic information, and local search were not used, allowing the tests to focus on the components of AFC. When ant restarts were used, b was set to 100.

Figure 1 shows the results of these tests. Using backjumping and unsafe backjumping by themselves are not shown since they could not finish the tests for 10 teams or more within a few days of running time. In the tests shown, using unsafe backjumping resulted in shorter times than using normal backjumping. Unsafe backjumping is able to get out of an infeasible partial-solution faster, which also reduces the need of doing ant restarts. Pattern matching had a significant impact, reducing the time needed for both forms of backjumping. This is because it is able to greatly reduce the number of conflicts caused by the *at_most* constraint.

The next set of tests dealt with ant restarts and the parameter b. In addition to 100, other values tested for b were 50, 200, and 500. These were tested on team sets between 6 and 16 while using unsafe backjumping and pattern matching. They used the same testing procedure as the first test. There was very little variation with the different values, with b being set to 100 resulting in the minimal time needed for all cases. For the largest set of 16 teams, the range of timings was small, between 2.68 and 2.91 seconds. This shows the robustness of using ant restarts and not needing to spend much time optimizing b. Additionally, a final test was run with b set to 1, giving a configuration similar to past works described earlier[2, 14]. This resulted in much poorer performance, with 16 teams taking an average of 4.11 seconds. This shows that it is beneficial to give the ants time to construct solutions instead of drastically limiting the tolerance for backtracks.

Table 1: Comparison of solution quality using safe backjumping against unsafe backjumping, both using ant restarts and pattern matching.

Set	Time	BJ+R+P	UBJ+R+P
NL6	20	24433.7 (282.3)	24493.7 (213.6)
NL8	40	42365.4 (637.0)	42309.7 (547.7)
NL10	80	70281.9 (810.8)	70006.0 (908.8)
NL12	160	135669.3 (1106.5)	135836.9 (1044.1)
NL14	320	245393.6 (2157.2)	245181.2 (2408.9)
NL16	640	343678.5 (2740.7)	343110.7 (2588.7)

The final tests done with AFC were to make sure that using unsafe backjumping does not degrade the solution quality. Unlike the earlier tests, these tests used pheromone and heuristic information. The tests were done with the NL set, 30 trials each. Backjumping and unsafe backjumping were run against each other, both using ant restarts and pattern matching. As shown in Table 1, using unsafe backjumping improved the solution quality more often than degrading it, with only NL6 and NL12 showing worse results. This is due to the ants being able to construct solutions faster, allowing for more cycles as the problem size grows.

4.1.2 ACS and AS

AS's and ACS's rules for choosing values were tested with and without heuristic information to see which one allowed better solution quality. Experiments showed that the algorithm performed best when using ACS's rule without heuristic information. Due to space limitations, this experiment's results are not shown.

The reason why ACS's rule preformed better is because AS's rule was not aggressive enough, especially when taking into consideration the problem structure and constraints. AS's rule could not build strong, candidate solutions from the pheromone matrix. With ACS's rule, since it takes the best available choice a large percentage of the time, it could build candidate solutions that shared much in common with the best solutions, but varied enough to allow exploration. Essentially, it allowed for a better balance of exploration and exploitation while AS's rule allowed for too much exploration. This concept can be seen in Figure 2. When looking at the average similarity between solutions constructed by the ants and the best ant seen so far, we can see that using ACS's rule allows the ant to construct solutions that share a greater similarity of pairings.

With the heuristic information, it was surprisingly found to be beneficial only when using AS's rule and not with ACS's rule. A possible reason for this is that the heuristic information could be seen as being too short sighted, sacrificing long term gain for short term gain. While this short

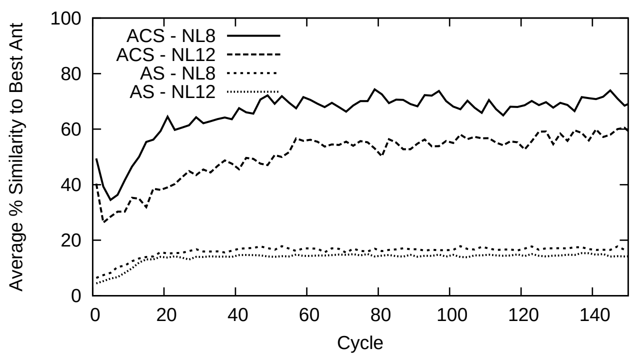

Figure 2: Comparison of using AS's and ACS's rules.

Table 2: Comparison of pheromone update schedules. The times used were 600 seconds for NL8, 1800 for NL10, 3600 for NL12, and 5400 for NL14.

Set	GB+IB	RB+IB	RB/GB+IB
NL8	39772.9(129.7)	39722.8(9.9)	39724.7(13.7)
NL10	61004.7(633.4)	60687.6(367.8)	60607.9(483.0)
NL12	116158.5(1218)	116075.1(642)	116414.7(684)
NL14	204765(3049)	202814(1919)	203541(1760)

term gain may have been beneficial in improving the solution quality when using AS's rule, especially as it was not able to build strong candidate solutions from past solutions, the heuristic information may have had a detrimental effect when using ACS's rule due to its aggressiveness.

4.1.3 Pheromone Update Schedule

Multiple pheromone update schedules using different combinations of iterative best, global best, and restart best ants were tested to see which resulted in the best average solution quality. The combinations describe the usage of ants for pheromone updates after a pheromone reinitialization took place. Prior to this, they all use the same alternation between global best and iterative best ant.

Table 2 lists results from the three best approaches averaged over 30 trials. The first set of results is from alternating between the global best and iterative best ant; second set is from alternating between the restart best and iterative best ant; and the third set is alternating between the restart best and iterative best ant for the first half of the reset length, and then alternating between the global best and iterative best ant until the next pheromone reinitialization. As seen, the best approach was when global best ants were not used after pheromone reinitialization. We believe the reason for this is because the solution space is large for the TTP. Using global best ants after pheromone restarts can anchor us to local minimums in the solution space, even with the pheromone reinitializations. Not using global best ants allows better exploration of the large solution space.

4.2 Comparison With ACO Approaches

We first compared our approach, AFC-TTP, with the two previous ACO approaches. As stated earlier, Crauwels and Van Oudheusden[4] used a direct application of ACO to the problem, while Chen et al.[3] used ACO as a hyper-heuristic.

One of the problems when trying to compare with their approaches is that their papers only stated the best results and not their averages. They also do not clearly state exactly how long their approaches took for each problem set. To make the comparison as fair as possible, we ran our approach for 20 runs each using very short running times for the same problem sets that they worked with. All the times are shorter than any listed times in the two papers.

Table 3: Comparison of AFC-TTP with previous ACO approaches.

Set	Crauwels	Chen	AFC-TTP		
	Best	Best	Time	Avg	Best
NL4	8276	8276	10	8276	8276
NL6	23916	23916	20	23916	23916
NL8	40797	40361	40	39785.5	39721
NL10	67871	65168	80	61951.3	60399
NL12	128909	123752	160	118609.1	115871
NL14	240445	225169	320	208146.7	203205
NL16	346530	321037	640	296220.75	292214

As we can see in Table 3, AFC-TTP outperformed both past ACO approaches even while using much smaller running times. For all instances, the average value found was better than or equal to their best values. This shows how we were able to greatly improve the performance of ACO when applied to the TTP.

4.3 Comparison With Other Approaches

Our second set of comparisons was with the more state-of-the-art single processor approaches: CNTS[6] and TTSA[1]. We ran two separate experiments, running each set of trials for the same average length of time as they reported, taking into account differences in processor speed. When comparing with CNTS, we used the same experimental setup of 10 trials per problem instance. When comparing with TTSA, we were only able to run 30 trials per problem instance instead of the 50 trials TTSA used. This was due to time constraints and the lengthy running times used by TTSA.

Table 4 shows AFC-TTP's results when compared with CNTS and TTSA. The average solutions found by AFC-TTP was always within a few percentage points of the two approaches, and in some cases was even better than CNTS for the smaller instances. These results are very promising, as they show that ACO can perform at a similar level to other metaheuristic approaches on this problem.

5. CONCLUSIONS

We have shown how ACO can be integrated with FC-CBJ to create a new approach, AFC, for tackling hard, constrained problems like the TTP. Using ideas of ant restarts, unsafe backjumping, and pattern matching, AFC is able to construct solutions in a minimal amount of time, allowing for more cycles to be performed on the TTP. In the end, our approach showed strong performance in solution quality, greatly outperforming past ACO approaches and performing comparably with other state-of-the-art approaches.

Even with this strong performance, there are still areas on which our approach can possibly be improved. One of these areas is looking at heuristic values. Finding a better way to calculate heuristic values for this problem could lead to better solution quality. As we have shown, the straightforward approach caused our approach to perform worse.

Another area is improving the exploitation and explo-

Table 4: Comparison of AFC-TTP with CNTS and TTSA.

Set	CNTS		AFC-TTP			
	Avg (StdDev)	Best	Time	Avg (StdDev)	Best	Avg % Dif
NL10	60424.2 (823.9)	59876	4969.51	59928.3 (155.47)	59634	-1.0
NL12	114880.6 (948.2)	113729	7660.07	114437.4 (895.7)	112521	-0.4
NL14	197284.2 (2698.5)	194807	20870.07	198950.5 (1294.43)	196849	0.8
NL16	279465.8 (3242.4)	275296	35931.27	285529.6 (3398.57)	278456	2.2
CIRC10	259.2 (2.3)	256	3131.47	254 (3.13)	248	-2.0
CIRC12	440.4 (1.7)	438	8167.18	436 (4)	430	-1.0
CIRC14	694.4 (6.4)	686	10183.59	692.8 (9.25)	674	-0.2
CIRC16	1030.0 (8.7)	1016	18896.9	1039.6 (5.56)	1034	0.9
CIRC18	1440.8 (10.5)	1426	39010.77	1494.8 (7.61)	1486	3.7
CIRC20	1998.4 (17.7)	1968	47316.76	2061.4 (8)	2046	3.1

Set	TTSA		AFC-TTP			
	Avg (StdDev)	Best	Time	Avg (StdDev)	Best	Avg % Dif
NL8	39721 (0)	39721	1188.32	39721 (0)	39721	0.0
NL10	59605.96 (53.36)	59583	29190.02	59773.5 (131.34)	59583	0.3
NL12	113853 (467.91)	112800	49658.27	114427.4 (465.81)	113523	0.5
NL14	192931.86 (1188.08)	190368	169316.89	197656.6 (1079.01)	195627	2.4
NL16	275015.88 (2488.02)	267194	139240.2	283637.4 (1865.52)	280211	3.1

ration of the solution space. While using ACS's rule for choosing values allowed us to better exploit past solutions and lead to better results, further work could help to better balance this issue as we believe the current version is too focused on exploitation.

6. ACKNOWLEDGMENTS

The authors thank Michael Trick, Christine Solnon and Thomas Stützle for their advice.

7. REFERENCES

[1] A. Anagnostopoulos, L. Michel, P. Van Hentenryck, and Y. Vergados. A simulated annealing approach to the traveling tournament problem. *Journal of Scheduling*, 9(2):177–193, April 2006.

[2] J. C. Beck. Solution-guided multi-point constructive search for job shop scheduling. *Journal of Artificial Intelligence Research*, 29:49–77, 2007.

[3] P.-C. Chen, G. Kendall, and G. V. Berghe. An ant based hyper-heuristic for the travelling tournament problem. In *IEEE Symposium on Computational Intelligence in Scheduling*, pages 19–26, 2007.

[4] H. Crauwels and D. Van Oudheusden. Ant colony optimization and local improvement. In *Workshop of Real-Life Applications of Metaheuristics, Antwerp*, 2003.

[5] R. Dechter. *Constraint Processing*. Morgan Kaufmann Publishers, San Francisco, California, 2003.

[6] L. Di Gaspero and A. Schaerf. A composite-neighborhood tabu search approach to the traveling tournament problem. *Journal of Heuristics*, 13(2):189–207, April 2007.

[7] M. Dorigo and L. M. Gambardella. Ant colony system: A cooperative learning approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computation*, 1(1):53–66, April 1997.

[8] M. Dorigo, V. Maniezzo, and A. Colorni. The ant system: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics Part B: Cybernetics*, 26(1):29–41, 1996.

[9] M. Dorigo and T. Stützle. *Ant Colony Optimization*. MIT Press, Cambridge, Massachusetts, 2004.

[10] K. Easton, G. Nemhauser, and M. Trick. The traveling tournament problem description and benchmarks. In *Principles and Practices of Constraint Programming, LNCS*, volume 2239, pages 580–585. Springer, 2001.

[11] K. Easton, G. Nemhauser, and M. Trick. Solving the travelling tournament problem: A combined integer programming and constraint programming approach. In *Practice and Theory of Automated Timetabling IV, LNCS*, volume 2740, pages 101–109. Springer, 2003.

[12] F. Glover and F. Laguna. *Tabu Search*. Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[13] A. Lim, B. Rodrigues, and X. Zhang. A simulated annealing and hill-climbing algorithm for the traveling tournament problem. *European Journal of Operational Research*, 174(3):1459–1478, November 2006.

[14] B. Meyer and A. Ernst. Integrating ACO and constraint propagation. In *ANTS 2004, LNCS*, volume 3172, pages 166–177. Springer, 2004.

[15] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3):268–299, 1993.

[16] T. Stützle and H. H. Hoos. $\mathcal{MAX}-\mathcal{MIN}$ ant system. *Journal of Future Generation Computer Systems*, 16:889–914, 2000.

[17] M. Trick. Challenge Traveling Tournament Instances. http://mat.gsia.cmu.edu/TOURN/.

[18] D. C. Uthus, P. J. Riddle, and H. W. Guesgen. Ant colony optimization and the single round robin maximum value problem. In *ANTS 2008, LNCS*, volume 5217, pages 243–250. Springer, 2008.

[19] P. Van Hentenryck and Y. Vergados. Population-based simulated annealing for traveling tournaments. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence*, pages 267–272, 2007.