Adversarial Constraint Satisfaction by Game-tree Search

Kenneth N. Brown, James Little, Paidi J. Creed and Eugene C. Freuder [1]

Abstract. Many decision problems can be modelled as *adversarial* constraint satisfaction, which allows us to integrate methods from AI game playing. In particular, by using the idea of opponents, we can model both collaborative problem solving, where intelligent participants with different agendas must work together to solve a problem, and multi-criteria optimisation, where one decision maker must balance different objectives. In this paper, we focus on the case where two opponents take turns to instantiate constrained variables, each trying to direct the solution towards their own objective. We represent the process as game-tree search. We develop variable and value ordering heuristics based on game playing strategies. We examine the performance of various algorithms on general-sum graph colouring games, for both multi-participant and multi-criteria optimisation.

1 INTRODUCTION

Constraint satisfaction has been successfully applied to a wide range of practical decision problems, but usually assuming there is a single solver in complete control. Many decision problems can, however, be modelled as an interaction between two or more adversaries, where each one attempts to guide the solution towards their own end. Here, we combine the two approaches in what we call *adversarial constraint satisfaction*. We consider what effect the notion of an adversary has on the constraint solving process; in particular we integrate techniques from AI game playing into constraint solving methods. We show that we can model problem solving in collaborative environments, where intelligent participants with different objectives must work together to produce a solution that satisfies all the problem constraints, and also that we can model multi-criteria optimisation, where a single decision maker must find a solution which obtains a balance between two or more objectives.

There are many protocols which the intelligent solvers could use to collaborate on the problem. In this paper, for a situation with two adversaries, we focus on one approach: the participants take turns to instantiate constrained variables, and thus effectively search over a game-tree. The game, however, must end in a complete satisfying solution, so the players must backtrack out of situations which lead to an inconsistency. As usual in game-tree search, the participants must reason about their own best moves in the context of the likely moves of their opponent. They must also now reason about the consistency of the constraints, and integrate this with the move selection. We draw an analogy between CSP variable and value ordering heuristics and strategies from game-tree search. We examine the performance of various algorithms on general-sum graph colouring games for both multi-participant and multi-criteria optimisation.

As a motivating example, consider planning university committee meetings. Each committee has possible meeting times, and each room on campus has limited availability. Researchers want to cluster meetings together, to leave more time for research. Administrators want to minimise travel time, preferring to locate the meetings close to the administration block. How should the University produce a schedule? The approach considered in this paper would appoint two agents, one for each interest group, and have them take turns choosing rooms and times for individual meetings, in the hope that the interplay between their choices would produce a fair settlement. The agents would clearly bring their own objectives to the problem. If the university prefers a particular balance, it could appoint agents with appropriate negotiating skills.

Our research has two main objectives: (i) to provide assistance for self-motivated decision makers in possibly adversarial situations, and (ii) to provide a convenient framework for modelling and solving multi-criteria constrained optimisation problems. Within the context of one particular game scenario, for (i) we propose configurations of the constraint-based searcher for play against known opponents. For (ii) we show how to configure both players to achieve desired results.

2 BACKGROUND

A binary constraint satisfaction problem consists of a set of variables $\{X_1, \ldots, X_n\}$, a set of domains of values for the variables $\{D_1, \ldots D_n\}$, and a set of constraints restricting the values pairs of variables can take simultaneously $\{C_1, \ldots, C_m\}$, where if C_{ij} is a constraint over (X_i, X_j) then $C_{ij} \subseteq D_i \times D_j$. A solution to a problem is an assignment to each X_i of a value v_i from its domain D_i, such that for each constraint C_{ij}, $(v_i, v_j) \in C_{ij}$. Many algorithms have been proposed for constraint solving, but the default has become *maintaining arc consistency* (MAC) [11], in which backtracking search is extended with constraint propagation. After each choice of value assignment, unassigned variables have their domains made arc consistent by deleting every value which does not have a supporting value in a domain with which it is constrained. Search time can be improved by choosing the order in which variables are considered, and by choosing the order in which values are tried. The standard variable ordering heuristic is to select the variable which has the fewest remaining domain values. Possible value ordering heuristics include selecting the value most likely to lead to a solution (i.e. the least constraining choice), or, for maximisation problems, selecting the value giving the biggest local increase in the objective function.

Constraint satisfaction methods traditionally assume one solver with a well-defined objective operating on a static problem. Some frameworks have been proposed where the solver is not in complete control of the environment, and must cater for uncertainty caused by probabilistic or random events - e.g. *Mixed* [1] and *Stochastic CSP* [13] both assume a subset of the variables are uncontrolled. Both of these frameworks can be considered to be sub-types of adversarial constraint satisfaction, in which the external world acts as an ad-

[1] Cork Constraint Computation Centre, Dept. of Computer Science, University College Cork, Ireland. *http://4c.ucc.ie/*

versary making probabilistic assignments. The CSP has also been extended to distributed models, where agents have control over subsets of the variables [5]. Agents are assumed to be cooperative, and share a goal of reaching a final solution. For cases where all variables can be instantiated by any participant, [3] configures teams of self-motivated agents with individual interests, but focuses on compromise strategies rather than reasoning about opponents' moves.

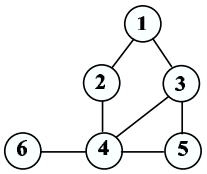
A variety of ways of handling multi-criteria optimisation within constraint problems have been proposed. [4] finds the complete non-dominated Pareto frontier using Point Quad trees. [8] use a single strategy to find one solution, and then switch strategies to find subsequent solutions. [2] introduce bounding constraints for each criterion, and use an iterative approach to solve job shop scheduling problems. For each of these approaches, to distinguish between the different solutions, the user has to provide a ranking of the criteria.

Game playing is a significant application area for AI, although focused mainly on two-player zero-sum games. The basic algorithm is *minimax*, in which each player chooses the move which minimises the maximum score the opponent can achieve. For small games, it is possible to evaluate all possible move sequences and determine the optimal move. The analysis of such games is at the heart of *Game Theory*. For larger games, the evaluation must be cut off after some limit. The states at the limit are then evaluated by estimating the score liable to be achieved by the relevant player, and these scores are propagated back up the search tree to provide an estimate of the value of each move. The cut-off and the evaluation need to take into account *quiescence* (is the state locally stable, or will the score fluctuate wildly on subsequent moves?) and the *horizon effect* (are there unavoidable moves which will affect the score but are not being factored in because they have been pushed beyond the cut-off horizon?). The best known algorithms use variants of *alpha-beta pruning* [6], in which moves which cannot improve on already discovered scores are pruned from the tree.

There has been some limited research on applications of game-tree search and game theory in constraint programming. [10] models each variable in a CSP as an agent, and attempts to maximise the degree of satisfaction of the CSP; solutions correspond to Nash equilibrium points. [7] shows how underlying consistency techniques used in CSPs correspond to certain game mechanisms.

3 THE GAME SETUP

The problem we consider in this paper is graph colouring with four colours, with two adversaries, although the approach naturally extends to arbitrary constraint optimisation problems with many adversaries. We will refer to the adversaries as agents or players. We represent the problem as a binary CSP, where each variable is a node with domain $\{r, b, g, y\}$, and a set of constraints requiring connected nodes to take different colours. Each agent has access to the full problem. The agents will take turns to select a variable and instantiate it with a value. On discovering an inconsistency, both agents must backtrack to the last decision point where a consistent choice was available, and continue from that point. We assume: (a) agents are correct and trustworthy - no agent will report an inconsistency if it doesn't exist; (b) both are committed to playing by the rules of the game - each agent will play in turn by instantiating a single uninstantiated variable with a value from its domain, and no agent will choose a value which it knows will lead to an inconsistency; (c) the agents are systematic complete searchers - a solution will be found if one exists. The agents may be self-motivated or antagonistic - they may make decisions which ignore or degrade the other agent's objective. The individual agents have different objective functions. We consider two main types: (1) maximise the number of nodes coloured with a specific colour; and (2) maximise the sum of weights, where each node has a unary function mapping colours to integer weights. We assume that each player knows the other's objective.



	r	g	b	y
1	2	3	1	3
2	3	1	0	2
3	1	4	2	3
4	2	1	3	0
5	3	2	5	2
6	2	1	1	6

Figure 1. Simple graph colouring problem

Consider the simple graph in Fig. 1. Player A wants to maximise the number of nodes coloured red, while player B wants to maximise the sum of the weights. The maximum score achievable by A is 3, while the maximum achievable by B is 22 - the constraints eliminate the naive estimates of 6 and 24 respectively. The minimum scores are 0 for A and 6 for B. In one possible sequence of moves, player A starts by selecting node 6 and colouring it red (a least constraining choice compatible with its objective). Player B selects node 5 and colours it blue (maximising weight). Player A colours node 2 red (again a least constraining choice, but this time also appearing to favour B's objective). Player B colours node 3 green (maximising the remaining weights). Player A now has no more nodes it can colour red, so it selects node 1 and colours it yellow. Finally, B must select node 4, and must colour it yellow. A has achieved an objective value of 2 (compared to a maximum of 3), while B has obtained 17 (of a possible 22), so both players have achieved a reasonable score. However, better solutions for both players exist: for example $\{(1, y), (2, r), (3, g), (4, b), (5, r), (6, r)\}$ gives a score of 3 for A, and 18 for B. Better reasoning by B early in the game might have spotted that A would choose red for node 5 if given the chance, and so B should have chosen to instantiate node 4 with blue instead. In the next section, we examine algorithms, propagation methods, strategies and heuristics aimed at emulating that type of reasoning, combining techniques from constraint satisfaction with game-tree search.

4 PLAYING THE GAME

4.1 The Game Exectuion Algorithm

```
play(player i,pos p): pos
1 if (finished(p)) return p
2 soln = null
3 ng = {}
4 move = select(i, p, ng, 0)
5 while (soln == null && move != (0,0))
6    p = apply(move, p)
7    soln = play(next(i),p)
8    if (soln == null)
9       move = select(i,p,ng+{move},0)
10 return soln
```

We assume the game is managed by a game controller. The basic algorithm is shown above. We start by passing the initial position to the first player, and asking for a move (line 4). While no complete solution has been found, and a valid move is returned, it applies the move (6), and then switches to the next player (7). If focus switches to a player on a final position, the position is returned (1). If no solution is possible below this point, it returns to the previous player and asks it to select a different move (9). For convenience, we record

here the nogoods for any particular position (lines 3 and 9). If no more moves are possible, it returns to the previous position (10). This algorithm is essentially the standard algorithm for executing games, but with the addition of backtracking if a dead-end is reached. Note that we do not specify here what constitutes a 'position' - this could be simply the current variable assignments, or it could also include the current state of the domains of uninstantiated variables. Similarly, in line 6, the game controller may simply make the assignment and check it for consistency, or it could apply constraint propagation to reduce the options needed to be considered by the participants.

4.2 The Player's algorithm

For a constraint solving agent, the game consists of periodic requests to select and instantiate a single variable in a dynamic CSP. How should the player make those decisions? A complete search of all possible decisions and their possible continuations would provide the most information. In large games, this is likely to require excessive time. Instead, as is common in game-tree search methods, we assume each player has a limited depth to which it can look ahead when selecting a move. The algorithm is shown below. The player considers each of its own possible decisions in turn (lines 3-5, making the CSP explicit), and evaluates the resulting positions (7) by considering each of the opponents possible decisions from that position (12). Each of those moves are evaluated in the same way until the depth limit is reached (11), where the nodes are evaluated by a static evaluation function. A value of 0 is returned for an inconsistent position (1). This is essentially a standard limited-depth game-tree search, but with legal positions defined by a CSP, and with backtracking on inconsistency. We can therefore use game-tree search techniques to improve the solver's constraint reasoning. Each individual player must decide how deep to look ahead, how to order branches for searching, how to evaluate partial solutions, how to account for the decisions of the other players, and how to decide upon the best variable and value. We consider each of these choices below.

```
select(player i, pos p, NG ng, depth d): move
1 eval = 0
2 best = (0,0)
3 for each var X in p
4    for each v in X's domain not in ng
5       if (X,v) is consistent with p
6          np = apply((X,v),p)
7          if (better(eval(i,np,d),eval)
8             best = (X,v)
9          eval = evaluate(i,np,d)
10 return best

eval(player i, pos p, depth d): integer
11 if (depth == limit(i))
12     return evaluatePos(i, d, p)
13 else
14     move = select(next(i),pos,{},d+1)
15     return eval(next(i), move,d+1))
```

Propagation The default case would be to do no propagation, but to rely on backward checking of the constraints. In order to evaluate the partial solutions, we will examine the domains of the uninstantiated variables and reason about the remaining values. The default case does not change the domains of uninstantiated variables, so we don't consider it any further. Instead, our simplest propagation case is forward checking. In line 5, each time we consider a pair (X,v), we would examine each uninstantiated variable Y constrained by X, and remove all values in Y's domain inconsistent with (X,v). If we assume each player maintains the domains between moves, then we know that (X,v) is never inconsistent with previously assigned variables. Therefore the move (X,v) is inconsistent only if it empties the domain of an uninstantiated variable. MAC is a higher level of propagation than FC, extending it by establishing arc consistency after forward checking - if any domains were reduced in size, all other constrained (uninstantiated) domains are checked for inconsistent values, and the process repeats until all domains are stable, or a domain empties. This constraint propagation directly addresses the problem of the horizon effect. Reasoning about the consequences of decisions identifies some of the future moves that are impossible, by removing the values from the domains. Similarly, if a domain is reduced to a single value, then the move is inevitable, and can be accounted for even if it occurs beyond the horizon. In addition, since players will be forced to backtrack if a deadend is reached, propagation helps to identify inevitable dead-ends earlier in the search.

Evaluation Functions The method of evaluating a partial solution depends on the particular objective function maintained by the relevant player. In each case, we examine the domains of uninstantiated variables, and estimate the chances of receiving a particular score from that domain. For maximising the number of nodes with a given colour, we score 1 for each domain reduced to the desired colour, and 0.5 for each domain containing that colour among others. For maximising weights, we score the given function for each domain reduced to a single value, plus the mean of the maximum weights for the other domains.

Game strategies Given the evaluation functions above, each player still needs a strategy for selecting moves - first to predict which move an opponent will make in any given state, and then to decide on an appropriate move for itself. We have investigated four variations of the basic minimax strategy from game-tree search, designed for general sum games. *Minimax* is a literal translation from zero-sum games, in which a player selects the move which minimises the maximum score its opponent may get. We do not expect this to perform well on our general sum games. *Maximin* guards against an opponent minimising the current player's score, and so selects the move which maximises the minimum score it may achieve. This strategy is similar to the 'paranoid' one in [12]. *Maximax* assumes each player will attempt to maximise its own objective; thus level 2 returns the maximising move for player 2, and level 1 returns the move which would lead to the maximum for player 1. This strategy is similar Max^n for multi-player games [9]. Finally, *MaxWS* is designed to handle inaccuracies in the evaluation function. The evaluations can only estimate the score from each position, since future moves and constraint propagation may reduce the score obtainable. Also, since on subsequent moves the opponent will be looking ahead to a deeper level, it will have more information on which to base the decision. Thus for moves under the opponent's control, we return the evaluations weighted by a probability that the opponent will make that move, while for the current player's moves, we choose the move that maximises a weighted sum of the evaluation. The probability of an opponent's move is calculated by dividing the payoff for that opponent by the sum of the payoffs over all moves. For each of the above strategies, we break ties lexicographically. In Fig. 2 we show the choices made for a lookahead of 2 in a two player game under different strategies. The circled choice indicates the move (A or B) chosen by player 1.

Depth of search The players may choose to look ahead as deep into the tree as they are able. For the initial experiments, we limit our

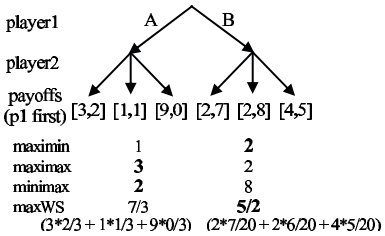


Figure 2. Game Playing Strategies

search to a depth of 1 or 2 (for complexity reasons, discussed below). That is, a player considers only its own moves, or considers both its own moves and the moves of its opponent.

Complexity How expensive is the computation required to make a decision at a node? We assume a player is using MAC in the lookahead phase. Suppose there are t variables remaining, and b values per variable. Suppose there are e constraints (worst case $O(t^2)$). The best AC worst-case complexity is eb^2. For a depth 1 lookahead, to evaluate a single (variable,value) pair requires $O(eb^2)$. There are tb possible decisions to be considered, giving $O(teb^3)$. For a depth 2 lookahead, each single (variable,value) pair has another $(t-1)b$ pairs to consider at depth 2, and each of these takes eb^2. That means we require $O(teb^3)$ to evaluate a single decision at the top level, and there are tb possible decisions, giving $O(t^2eb^4)$. For low levels of lookahead, increasing the lookahead depth by 1 raises the worst-case time required by a factorof tb.

Branch ordering During the lookahead procedure for an individual decision, the ordering on the branches corresponds to an ordering on the variables followed by an ordering on the values. For this paper, we simply select variables and values lexicographically.

CSP ordering heuristics In the overall search process, any combination of the above options can be thought of as a combination of variable and value ordering heuristics for standard constraint search. At a decision point, the player evaluates all options, and selects its best variable-value combination. We can thus consider the adversarial game-tree search as a standard search algorithm, in which we alternate between two (complex) heuristics for making decisions.

5 EXPERIMENTS

We have implemented the above search, propagation and strategies using the Koalog Java constraint library. We have generated 50 random graph colouring problems, each with 4 colours and 15 nodes, at a density of 50% (to ensure that each problem has a number of solutions). We assume that the game controller establishes AC at each step, passing the reduced domains to the next player. For each graph, for each pair of objectives, we generate the Pareto frontier - the set of solutions which are better than all others in at least one objective.

For the multi-participant games, assuming we know the opponent's configuration, we ask four questions: what strategy should I play if (a) I want to beat my opponent's score in individual games; (b) I want to achieve, on average, a higher score than my opponent; (c) I want to get as close to my optimal score as possible; and (d) I want to minimise the 'Pareto regret' i.e. I want to maximise my own score given that my opponent has achieved a particular score. For the multi-criteria problems, we ask how should we configure the two players (e) to achieve a result close to being Pareto optimal, or (f) to bias or balance the performance?

Game 1: Number of reds vs number of blues In Fig. 3, we show the result of one game between two agents on one random graph. We plot the first player's achieved value on the x-axis, and the opponent's

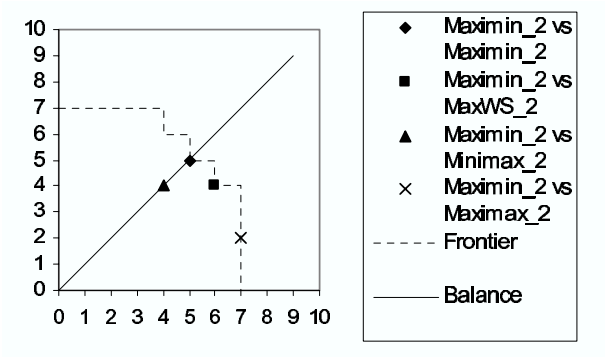


Figure 3. Result for one game, Red v Blue

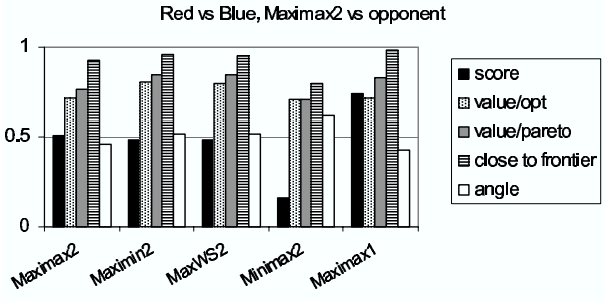


Figure 4. Average results Red v Blues against Maximax 2

achieved value on the y-axis. We also plot the Pareto frontier, showing the best results achievable, and the balance line. In Fig. 4, we show the performance of five different player configurations against an opponent playing Maximax with a lookahed depth of 2, where the opponent plays first. The results are averaged over the 50 graphs. For each possible counter-strategy, we show (i) 'score' - the fraction of games won by the opponent; (ii) 'value/opt' - the ratio of the player's achieved value to its maximum possible value; (iii) 'value/pareto' - the ratio of the player's achieved value to the corresponding Pareto optimal value, or 1 - 'Pareto regret'; (iv) (closeness to Pareto) a measure of how close the combined result was to the Pareto frontier, with 1 being on the frontier; and (v) (angle) a measure of the average winning margin, where 0 is the optimal result for the player, and 1 is the optimum for the opponent. By a win for a player, we mean the player achieved a higher fraction of its optimal value than the opponent. For the multi-participant games, we look at (i), (ii), (iii) and (v), and for the multi-criteria problems, (iv) and (v).

For the multi-participant games, we can see that the best configuration for a player playing second against Maximax 2 varies depending on which question we ask. For (a) and (b), Minimax 2 is clearly best. However, for (c) or (d), the Maximin or MaxWS strategies are best. For the multi-criteria optimisation (e), all strategies except Minimax produce results close to the Pareto frontier. Similarly, (f) most strategies except Minimax are close to the balance line. For both types of problem, similar results are obtained when the opponent plays second, except that measures (i) and (v) are more heavily weighted to the player than the opponent; there is an advantage to playing first. For lack of space, we cannot show the full set of graphs comparing all player configurations; instead, we summarise the results in Fig. 5.

Game 2: Preferences vs number of reds Fig. 6 shows the same set of players, but now playing an asymmetric game, where the opponent, playing first, is maximising preferences. The main thing to note is that the opponent always 'wins'. Again, all strategies except Minimax produce results close to the Pareto frontier. A summary is

	Game: Reds v Blues	Game: Preferences v Reds
Question	Recommended Configuration	Recommended Configuration
a	Minimax	Minimax
b	Minimax	Minimax
c	Maximin or maxWS	Maximin or MaxWS
d	Maximin or maxWS	Maximin or MaxWS
e	Maximax(2ply) vs Maximax(1ply) Maximax(1ply) vs Maximax(2ply) MaxWS(2ply) vs Maximax(1ply) Maximax(1ply) vs MaxWS(2ply) Maximax vs Maximin	All Maximin (except with Minimax), all Maximax (except with itself and Minimax), all MaxWS (except with Minimax)
f balance	Maximax vs Maximax maxWS vs Minimax	Maximax(1ply) vs Maximax(2ply)
bias red	Maximin vs Minimax	Maximax(1ply) vs Maximax(2ply)*
bias Pref		Minimax vs (Maximax or Maximin)

* best possible for reds, although Preferences wins
Assume 2ply unless specified

Figure 5. Summary of two Game Types

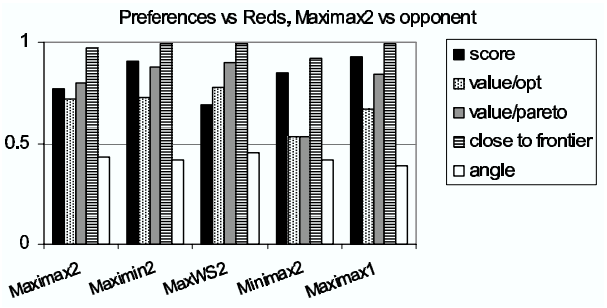


Figure 6. Average results Preferences v Red against Maximax 2

also shown in Fig. 5.

6 DISCUSSION

Some strategies have clear performance implications, depending on the goals of the adversarial solvers. Beating your opponent is different from improving your own objective, and requires a different configuration. Minimax pulls the balance towards itself, at the expense of its own value and of overall quality. It is a destructive strategy, that ignores it own score. MaxWS and Maximin attempt to protect their own objectives, making few assumptions about the opponent; while they lose as many games as they win, they perform well in achieving high individual scores and balanced scores. Maximax performs well in the multi-criteria optimisation, but not so well from the multi-participant view. It takes an optimistic view, but appears to be confused by the evaluation functions. The type of game is important: in the asymmetric game, the greater granularity of the preferences objective seems to allow the players to get closer to the optimum.

The depth of lookahead is important - the deeper the lookahead, the better the individual result. On the other hand, combining players with different lookaheads works well in multi-criteria problems, when one player sacrifices some of its score for a better overall result.

For practical applications, a number of improvements need to be made. We have largely ignored the time taken to make a decision, arbitrarily cutting off look ahead at a depth of 2. The time taken for two players using MAC to a depth of 2 to solve a graph to completion is approximately 5 seconds, but most of that time is taken up on the first 2 or 3 moves. If more time is available, then either higher levels of consistency could be used, or a greater depth of lookahead. If less time is available, we could limit the lookahead to depth 1, or reduce the propagation. Ultimately, we could introduce solvers with no propagation, but using heuristics to select moves. For the multi-criteria case, this would be a variation on standard constraint optimisation search, switching the variable and value ordering heuristic at each depth of the tree. Alternatively, we could change the structure of the game, allowing multiple moves in succession for some players.

Further improvements need to be made to the evaluation functions. Our estimate of the value of an intermediate state is fairly crude, and more accurate evaluations may lead to better results. One possibility would be to develop constraint-based variations of alpha-beta pruning, reasoning about bounds on the values obtainable for each player. Alternatively, soft consistency algorithms, which maintain counts of minimum achievable total scores for each individual value may provide better estimates.

7 CONCLUSION

We have developed the notion of adversarial constraint satisfaction, in which a solver has to cope with decisions made by an adversary during the solution process. This notion encompasses a number of constraint frameworks that have been proposed to handle uncertainty. We have looked in more detail at a narrow version, where adversaries take turns to make assignments. This gives a formulation similar to adversarial search in general AI, but with backtracking on reaching a dead-end. We have developed strategies for general-sum games, allowing the solver to reason about the adversary's likely assignments. By propagating constraints at each move, we can reason about the consequences of individual assignments made by the solver or adversary. We have shown that the framework supports multi-participant constraint games and multi-criteria constraint optimisation problems. For multi-participant games, different solver configurations are advised, depending on the performance measure and the adversary's configuration, but greater lookahead produces better results. For multi-criteria problems, we have shown that good quality solutions can be obtained using appropriately configured players. Furthermore these solutions can also be balanced in their objectives. Giving bias to the solution is also possible, given the available types of game and strategies.

ACKNOWLEDGEMENTS

This work has received support from Science Foundation Ireland under Grant No. 00/PI.1/C075.

REFERENCES

[1] H. Fargier, J. Lang, and T. Schiex, 'Mixed constraint satisfaction: a framework for decision problems under incomplete knowledge', in *AAAI'96*, (1996).
[2] F. Focacci and D. Godard, 'A practical approach to multi-criteria optimization problems in constraint programming', in *CPAIOR'02*, (2002).
[3] E. Freuder and P. Eaton, 'Compromise strategies for constraint agents', in *AAAI'97 Workshop on Constraints and Agents*, (1997).
[4] M. Gavanelli, 'An algorithm for multi-criteria optimisation in csps', in *ECAI'02*, pp. 136–140, (2002).
[5] Y. Hamadi, C. Bessière, and J. Quinqueton, 'Backtracking in distributed constraint networks', in *ECAI-98*, pp. 219–223, (1998).
[6] D. Knuth and R. Moore, 'An analysis of alpha-beta pruning', *Artificial Intelligence*, **6**, 293–326, (1975).
[7] P. Kolaitis and M. Vardi, 'A game-theoretic approach to constraint satisfaction', in *AAAI*, (2000).
[8] F. Le Huede, M. Grabisch, C. Labreuche, and P. Saveant, 'Multi-criteria search in constraint programming', in *CPAIOR'03*, (2003).
[9] C. Luckhardt and K. Irani, 'An algorithmic solution of n-person games', in *AAAI*, pp. 158–172, (1986).
[10] F. Ricci, 'Equilibrium theory and constraint networks', in *Intl Conf on Game Theory*, (1991).
[11] D. Sabin and E. Freuder, 'Contradicting conventional wisdom in constraint satisfaction', in *PPCP'94*, (1994).
[12] N. Sturtevant and R. Korf, 'On pruning techniques for multi-player games', in *AAAI*, pp. 201–207, (2000).
[13] T. Walsh, 'Stochastic constraint programming', in *ECAI*, (2002).