Computing the cheapest winning coalition in a multi-agent game

Loïs Vanhée

▶ To cite this version:

Loïs Vanhée. Computing the cheapest winning coalition in a multi-agent game. Computer Science and Game Theory [cs.GT]. 2010. dumas-00530790

HAL Id: dumas-00530790
https://dumas.ccsd.cnrs.fr/dumas-00530790v1

Submitted on 29 Oct 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

IRISA NICTA

Computing the cheapest winning coalition in a multi-agent game

Loïs Vanhée

Supervisors:
Sophie Pinchinat
Scott Sanner
Sylvie Thiebaux

2010

Université de Rennes 1, Research unit: IRISA, Team: S4
Australian National University, Research unit: NICTA, Team: SMLG
IFSIC Research Master's degree in Computer Science
ENS Cachan, antenne de Bretagne

Contents

1 Introduction **4**

2 Turn-based games **5**
2.1 Definition and Notations . . . 5
2.2 Coalitions Plays and Strategies . . . 6

3 Classical algorithms **7**
3.1 $MinMax$ algorithm . . . 7
3.2 $pre^*_C(Goal)$ algorithm . . . 7

4 Theoretical results **8**
4.1 The Problem to Solve . . . 8
4.2 Winning Transitivity . . . 9
4.3 Winning positions properties . . . 10
4.4 Barricades . . . 10
4.5 Overestimate a Biggest Winning Coalition with a Path . . . 11
4.6 Worst Case . . . 11

5 Algorithmic Solutions **12**
5.1 Checkers Optimizations . . . 12
5.2 Coalition-Based Seeker . . . 13
5.3 Position-based Algorithm . . . 17

6 Implementation Aspects **19**
6.1 Frontiers . . . 19
6.2 Symbolic Representations of Structures . . . 20
6.2.1 Minimal Representations and Operations . . . 20
6.2.2 Automaton Objects Representations . . . 21
6.2.3 Sets Representation and Manipulation . . . 22
6.2.4 Closed Sets of Sets Representation . . . 23
6.3 Cache . . . 24
6.4 The Position-Based Algorithm in a Symbolic Structure . . . 24
6.5 Computing a Cheapest Strategy . . . 24

7 Discussion on Games and Costs **26**
7.1 Games . . . 26
7.1.1 Autistic Games . . . 26
7.1.2 STRIPS Games . . . 28
7.1.3 Dynamic Adversarial Planing Games . . . 28
7.2 Cost Functions . . . 30
7.2.1 Definition and Exploitation . . . 30
7.2.2 Cardinal Cost . . . 31
7.2.3 Monotonic Cost . . . 31
7.2.4 Action Cost . . . 31
7.2.5 Combo Cost . . . 31
7.2.6 Unconstrained Cost . . . 32

8 Experiments and Results **32**
8.1 Experimental Settings . . . 32
8.2 Results . . . 33

9 Conclusion **34**

10 Bibliography **35**

11 Annexes **37**
11.1 Plots and Draws . . . 37
11.2 Properties and Proofs . . . 39
11.3 $MinMax$ Checker . . . 41
11.4 Comportments of the Coalition Based Seeker . . . 43
11.5 Heuristics . . . 45
11.6 Notes . . . 48
11.7 Future Developments . . . 49
11.8 Computing efficiently mc . . . 50
11.9 Improving the Maximal Losing Coalition algorithm . . . 50

1 Introduction

To run a company, employers need to hire people to realize a product, a sport coach can motivate some players intend to win a match, a designer have to force synchronization of some parallel processes to avoid deadlocking. In these examples, a device external to the global system tries to force the behavior of some agents, aiming at achieving a global objective. For realism purposes, we assume that forming a coalition has a cost that should be minimal. This cost models the fact that individuals of the coalition may be deviated from their original objective. Game theory is a vast field that addresses this kind of problem.

Depending on the type of games that are considered, the problem is more or less difficult to solve, and can be even non-computable. The inability for the agents to communicate can lead to stupid solutions, where e.g. agents with compatible objectives may reach a deadlock. Therefore, some frameworks allow agents to communicate in order to team up: this is the “coalition formation” [21]. Agents of the coalition share a common strategy that fulfills every individual objective (otherwise a rational agent would not have joined it). The traditional coalition formation problem leads to extended problems of game theory: it focuses on issues such as determining whether some agents are always in a same coalition or not, whether negotiations terminate or not, etc. These kinds of issues tightly depend on the social welfare resolution and the payoff assignation mechanism [12]. In such theory, every agent cares only about her objective, without caring about the global objective. As we cannot know how non-hired agents can react, in particular their objectives may be in conflict with ours, these “opponents” may team up to prevent the outcome from being in our favor. In this case, the setting easily reduces to determine the winner in a 2-player game, namely the coalition against all other players.

In this work we focus on turn-based games, a simple and broadly explored branch of game theory. Turn-based games offer well-known tools, among which the most famous *minmax* algorithm where a constrained optimization problem is addressed [19]. The problem we aim at solving is to find a minimal cost coalition which enforces some position in a given set (the goal) to be eventually reached.

Note that the problem we handle can be represented by a derived game. Let us add a new agent a_m for “agent manager” who starts the game: a_m select a coalition C and moves to a copy of the initial game, where the payoff of every agent in C is 1 if the objective is achieved and 0 otherwise; the reversed payoff is given to the other agents. If the objective is achieved a_m is rewarded $-cost(C)$ otherwise she is rewarded $-\infty$ and never accepts to enter a coalition. The objective of a_m is to maximize her payoff, amounting to select the lower cost winning coalition. Also, every agent in the selected coalition wants to achieve the global objective, to be rewarded 1 instead of 0. Any solution based on this derived game is always less efficient than any dedicated tool that generates on-the-fly the choice of a_m.

Such a translation of a payoff setting problem into a game is one of the first results of the Mechanism Design: this branch of game theory allows to modify a game with the intentions to distinguish a particular objective. In our case, we transform the game by setting the reward of agents, the distinguished objective becomes the global objective. More details about Mechanism Design can be found in [18].

In a complementary way, our problem can be artificially transformed into an adversarial planing problem [10], [14],[13]. Indeed, in the first step, the system agent selects

the coalition she wants. This selection induces a cost that is the one of the coalition. Then, every state of a coalition agent is a system state and the other states belong to the environment. The matter is that planers are not adequate to handle coalition properties. Some heuristics can help the search, but the same heuristics could be used in a dedicated tool. So, planing is not the immediate best solution, but some planing tools may be used since both domains are close.

Unfortunately, none of these settings can help solving our problem in polynomial time, since finding an optimal coalition with e.g. a minimal size (for a reachability objective) is an NP-hard problem [2]. This is our starting motivation for designing efficient algorithmic solution to solve the cheapest winning coalition problem.

Our work consisted in implementing a prototype to compute efficiently the cheapest winning coalition in a turn-based game for a reachability objective. The present report describes our approach and is organized as follows.

The games we have considered are presented in section 2.1. Then classic algorithms to check if a given coalition is winning are recalled in section 2.

Next, our contribution starts by theoretical foundations, section 4.1: we show that the coalition space can be reduced to its minimal winning and maximal losing coalitions, which is a central object in our algorithmic solutions described in section 5. These algorithms work efficiently thanks to adequate data structures (see section 6). Experiments of these algorithms are explained in section 8. Then we extend our framework to specific games and cost functions in section 7, and we finally give our conclusion in section 9.

2 Turn-based games

2.1 Definition and Notations

Informally, in the real word, to play a game, we must gather players (agents), tell them rules (playable actions and their effect on the game). Then, we let them play a while and when some conditions are achieved, the game ends and some winner(s) is declared.

There are different kinds of games. Some of them allow players to act simultaneously (concurrent games), add random features (stochastic games) or having a partial knowledge of the current situation (partial information games). We focused on deterministic turn-based games with perfect information.

Definition 1 *A* turn-based game *is a 7-tuple*

$$(Agt, Pos, owner, Act, Edg, p_0, Goal)$$

where:

- *Agt is a finite set of agents (or players).*
- *Pos is a finite set of positions. It describes the state of the game.*
- *A complete function $owner : Pos \rightarrow Agt$. $owner(p)$ defines who plays in position p*
- *Act is a finite set of actions. In using an action, a player affects the play.*
- *$Edg \subseteq Pos \times Act \rightarrow Pos$ defines the transition function.*

- $p_0 \in Pos$ *is the initial position.*
- $Goal \subseteq Pos$ *the set of goal positions.*

We generalize owner to the domain 2^{Pos} *in a natural way: for* $P \subseteq Pos$, $owner(P) = \bigcup_{p \in P} owner(p)$ *Also, we abuse notation by writing* $(p, a, p') \in Edg$ *to mean* $Edg(p, a) = p'$.

We now explains how games are played. Each position p is controlled by $owner(p)$ who selects an action a she plays and $Edg(p, a)$ is the position resulting of this action.

In the rest of this report, we consider that $G = (Agt, Pos, owner, Act, Edg, p_0, Goal)$ is given.

From this first general definition of the game, extended notations have to be presented. Let us define $next(p)$, the set of successors of the position p. Formally, $next(p) = \{Edg(p, a_i) | \forall a_i \in Act\}$.

Symmetrically, let us define $pred(P)$, the set of predecessors of a set of positions P by $pred(P) = \{p | p \in Pos \wedge (\exists p' \in P \text{ s.t. } p' \in next(p))\}$.

The game G can be represented by an automaton $G = (Pos, Act, Edg, p_0, Goal)$, where: Pos is the set of states, Act is an alphabet, Edg is the transition function, p_0 is the initial state, $Goal$ is the set of accepting states.

2.2 Coalitions Plays and Strategies

Definition 2 *A* coalition *is a set* $C \subseteq Agt$.

A sub-coalition (resp. super-coalition) of a coalition C *is a coalition* C' *s.t.* $C' \subseteq C$ *(resp* $C \subseteq C'$*)*

The opponents or enemies *of a coalition* C *is the coalition* $\bar{C} = Agt \setminus C$.

Let us write the set of coalitions: $Coal = 2^{Agt}$.

When coalitions are formed, the game can start. To monitor how the game evolves, we define the notion of play.

Definition 3 *A* play pl *from a position* p_0 *is an infinite sequence* $(p_i, act_i)_{i \in \mathbb{N}}$ *s.t.* $\forall i > 0, p_i \in Edg(p_{i-1}, \alpha_i)$*, where* $p_i \in Pos$ *and* $\alpha_i \in Agt$*. We note* $pl(i)_p$ *be* p_i *s.t.* (p_i, α_i) *is the* i^{th} *position of* pl*. Symmetrically,* $pl(i)_a$ *is the* i^{th} *action played in* pl

A play is goal-winning *if it contains a position in* $Goal$*. Let* $Play$ *be the set of every play.*

We now turn to strategies the way agents selects the actions to lead the play where they want.

When an agent plays, she can use the whole knowledge she can extract since the beginning of the game, to establish her next action. So, knowing the play, she has to decide what is her next action: this is the *strategy*.

Definition 4 *A* strategy for a player a *is a partial function* $strat : Pos^+ \to Act$*, defined for every chain of positions ending in some positions* p *with* $owner(p) = a$.

A strategy for a coalition C *is a strategy for every agent in this coalition:* $strat_C = \{strat_a\}_{a \in C}$.

A play pl resulting from the application of the strategies S *s.t. for every* $agt \in Agt$*, exists* $s \in S$ *s.t.* s *is a strategy for* a *is the (unique) play generated by these strategies. Let us note this play* $pl = \wedge_{s \in S} s$.

An interpretation of this definition is that, before the game starts, players in the coalition share a strategy. They establish together how each player in this coalition plays in every position. They form a perfect team playing as a single agent.

Let us note that, given a reachability goal we can restrict to memoryless strategies, that is a strategy that can be expressed by a function $strat : Pos \rightarrow Act$. The proof of this assumption is presented in [6] and [17]. Such property allows us to define the strategies for the positions, instead of the chains of positions.

Definition 5 *A* winning strategy s for a coalition C from $p \in Pos$ *is a strategy s.t. whatever agents in* $\bar{C}$ *plays, every play initiated from* p *according to* s *is goal-winning. In this case, we say that* p is a winning position of C *and* C is goal-winning from $p \in Pos$

A winning strategy s for a coalition C *is a winning strategy for* C *from* p_0*. We say then that* coalition C is goal-winning*. Otherwise,* C *is* losing.

Let us introduce notations according to these definitions. Let us define the set of goal-winning coalitions $GW \subseteq Coal$ s.t. $GW = \{C|C \text{ is goal-winning}\}$. Let us define the set WP_C of the winning positions of a coalition C, $WP_C = \{p|p \text{ is a winning position of } C\}$. Finally, let us define for every position p the set $winners_p : Coal$ of goal-winning coalitions from p s.t. $winners_p = \{C|C \in Coal \wedge p \in WP_C\}$.

3 Classical algorithms

3.1 $MinMax$ algorithm

$MinMax$ is a well-known algorithm that computes the outcome of 0-sum 2-player game in every location of its game-tree. One player (Max) tries to maximize the outcome and the other one (Min) tries to minimize it. Every node in the game tree is controlled by one of these players. Every leaf contains a value, representing the payoff of the game.

This algorithm performs a DFS search and evaluate a node n of the tree by:

- its value, if it is a payoff leaf
- the minimum of the successors values if n is a Min node
- the maximum of the successors values if n is a Max node

Some optimizations exists, to avoid the complete exploration of the tree. The most famous one is $\alpha\beta$.

$\alpha\beta$ prunes nodes without a complete exploration of their successor, when the outcome of this successor cannot be better than the current one. Then, in modifying the search, this pruning factor can be maximized. The literature about this domain is huge and cannot be presented here.

A modification of this algorithm is used in section 11.3.

3.2 $pre^*_C(Goal)$ algorithm

Let us define a function $pre_C : 2^{Pos} \rightarrow 2^{Pos}$ defined by: $pre_C = S \cup \{p|p \in pred(S) \wedge owner(p) \in C\} \cup \{p|p \in pred(S) \wedge next(p) \subseteq S\}$

Property 1 *For every $S \subseteq Pos$ and coalition C s.t. $S \subseteq WP_C$ then $pre_C(S) \subseteq WP_C$*

Proof For every $p \in pre_C(S)$, either:

- $p \in S$ then $p \in WP_C$
- $p \notin S$ then either:
  - $owner(p) \in C$. By definition of $pre_C(S)$, there exists $p' \in Pos$ s.t. $p' \in next(p)$. Consequently, there exists $\alpha \in Act$ s.t. $p' = Edg(p, \alpha)$. Then, C have a strategy to lead the game from p into a winning position, then $p \in WP_C$
  - $owner(p) \notin C$. By definition of $pre_C(S)$, $next(p) \subseteq S$, then, whatever the action played p the game is lead to a winning position, so $p \in WP_C$.

□

Then, let us extend pre_C to $pre^i_C : 2^{Pos} \rightarrow 2^{Pos}$ s.t. $pre^0_C(S) = S$ and if $i > 0$ $pre^i_C(S) = pre(pre^{i-1}_C(S))$. Let us define $pre^*_C(S)$ as the fix-point of $pre_C(S)$. Trivially, $pre^*_C(Goal) \subseteq WP_C$

A trace of this algorithm can be found on Figure 1. In 4.4 we prove that $pre^*_C(Goal) = WP_C$.

4 Theoretical results

4.1 The Problem to Solve

Definition

The initial goal of our problem is to design an efficient algorithm that takes in input a game and returns a winning coalition with the smallest size.

We extend this definition for every cost function over the coalitions. As a result, the inputs are the game and a cost function *cost*. Formally, our problem is to find C s.t.

$$C \in GW \wedge \forall C' \in GW \Rightarrow cost(C') \geq cost(C)$$

In the following, the set of cheapest winning coalitions is $\mathbf{C_{opt}}$ and a cheapest winning coalition is $C_{opt} \in \mathbf{C_{opt}}$

Where Does the Time Go By?

The problem to check if a coalition of size k can win a given game is NP-complete (as shown in [2]). Let us define which parts of our problem are difficult to solve polynomially. We manipulate 2 spaces exponential to their representation: the game space and the coalition space.

First, let us focus on the game space. In our case, we restrict to finite games. Nonetheless, even in this case, the size of the automaton representing the game can be exponential over the size of its representation. In k rules (transitions defined for a subset of the positions), up 2^k positions can be expressed.

For instance, let us define a 1-player vector-modification game. Every location is a binary vector of size k and $p_0 = (0, 0, \ldots, 0)$. The ith rule sets the ith bit to 1 and every

rule is triggerable in every position. Trivially, every binary vector of size k can be a position of this game. As there is 2^k vectors, there is 2^k positions.

Such representations implies to use specially designed tools that avoids enumerating every position. The tool we used (BDDs) is presented in section 6.2. Moreover, such representation triggers that algorithms should not enumerate the position-space, restricting the set of algorithms that can be used.

The game must be stored in memory intend to use algorithms on it. This space limit is so a hard constraint.

On this exponential-size space, the space of coalitions added on it is exponential too. As every coalition can be the smallest winning one, the search-space, or the coalition space is the space of partitions of Agt. Its size is equals to $2^{|Agt|}$.

Contrary to what we need for the game representation, there is no real need to store the whole search space. To solve our problem, we can easily enumerate every coalition and test if it is winning one by one. Even if being memoryless is time-consuming, in the algorithms we design, keeping in memory a set that can increases up to the whole search space is intractable.

In this section, we present theoretical foundation needed in our algorithms and data structures. Every piece of theory is linked to its applications and to notes if necessary.

4.2 Winning Transitivity

Coalition space is interesting since it presents this properties:

Property 2

- *If a coalition is goal-winning then, every super-coalition is also goal-winning.*
- *If a coalition is losing then, every sub-coalition is also losing.*

Formally, $\forall C, C' \in Coal, C' \supseteq C \wedge C \in GW \Rightarrow C' \in GW$ *and,* $\forall C, C' \in Coal, C' \subseteq C \wedge C \notin GW \Rightarrow C' \notin GW$

Proof The proof of $\forall C, C' \in Coal, C' \supseteq C \wedge C \in GW \Rightarrow C' \in GW$ is trivial. If C have a winning strategy, then C' have for winning strategy: the same winning strategy than the C one for every agent in $C \cap C'$ and a random one for agents in $C' \setminus C$. As the C winning strategy is played then the new strategy is winning for C'.

The proof of $\forall C, C' \in Coal, C' \subseteq C \wedge C \notin GW \Rightarrow C' \notin GW$ is symmetrical : if $\bar{C}$ have a strategy to prevent C to be goal winning, then $\bar{C}' \supseteq \bar{C}$ have the same winning strategy. Consequently $C \notin GW \Rightarrow \forall C' \subseteq C \wedge C' \notin GW$. □

Definition 6 *A set* $\boldsymbol{S}$ *of sets is a* downward-closed set *iff* $S \in \boldsymbol{S} \Rightarrow \forall S' \subseteq S . S' \in \boldsymbol{S}$. *A set* $\boldsymbol{S}$ *of sets is an* upward-closed set *iff* $S \in \boldsymbol{S} \Rightarrow \forall S' \supseteq S . S' \in \boldsymbol{S}$.

The property 2 shows that GW is upward-closed. Oppositely, the set of losing coalitions is downward closed.

Note that the theory of this property is far more general than game theory. Every problem ruled by a search space with a partial order function over its elements can use these properties and applications.

4.3 Winning positions properties

Proposition 3 *If a position is winning for a coalition, then this position is winning for every bigger coalition. Consequently, the set of winning positions is increasing with the coalition increase. Formally:*

(1)$\forall C \subseteq C' \in Coal, \forall p \in Pos, p \in WP_C \Rightarrow p \in WP_{C'}$

(2)$WP_C \subseteq WP_{C'}$

Proof The proof is quite trivial and the argument is the same than the one presented in section 4.2: if C have a winning strategy from p then C' have the same winning strategy. □

4.4 Barricades

Definition 7 *A* barricade B for a coalition C *is a set of positions s.t.* $owner(B) \subseteq \bar{C}$ *and cuts the game (every path from the initial location to a goal contains at least one position in B) in 2 parts:* B_W *and* B_L *(and* $B \subseteq B_L$*).* B_L *contains no goals and every position* $p \in B$ *contains a successor in* B_L*, by an action* $go_back(p)$*.*

Proposition 4 *Every position* $p \in B_L$ *is losing.*

Proof In selecting the action $go_back(p)$ for every position in B, $\bar{C}$ prevents any play from L to reach a goal, so no position in B_L can reach the goal. Consequently, every position $p \in B_L$ is losing. □

Property 5 $B = pred(pre^*_C(Goal)) \setminus pre^*_C(Goal)$ *is a barricade for* C

Proof For every $p \in pred(pre^*_C(Goal)) \setminus pre^*_C(Goal)$, $owner(B) \subseteq \bar{C}$ is true otherwise p would be in $\{p|p \in pred(pre^*_C(Goal)) \wedge owner(p) \in C\}$, so p would we in $pred(pre^*_C(Goal))$.

Trivially, any path from a losing to a winning position goes through a predecessor of a winning position, so B is a cut. B_L does not contain any goal, otherwise a losing position would be a goal. Every position $p \in B$ contains a successor in B_L, otherwise $p \in \{p|p \in pred(pre^*_C(Goal))) \wedge succ(p) \subseteq pre^*_C(Goal))\}$ and then $p \in pre^*_C(Goal)$. □

In the remaining $B = pred(pre^*_C(Goal)) \setminus pre^*_C(Goal)$. Let us illustrate a barricade on the figure 2.

Corollary 6 $B_W = pre^*_C(Goal) = WP_C$

Proof $pre^*_C(Goal) \subseteq WP_C$ has been proved in section 3.2. $pre^*_C(Goal) \supseteq WP_C$: as $B = pred(pre^*_C(Goal)) \setminus pre^*_C(Goal)$ is a barrier, any position $p \notin pre^*_C(Goal)$ that would be in B_L, so p cannot be winning. □

Corollary 7 $WP_C = WP_{C \cup (Agt \setminus owner(B))}$

Proof The argument is the same one presented in Propriety 16, but in an iterative point of view. Let us define by $\{a_1 \dots a_k\}$ the agents in $C \cup Agt \setminus owner(B)$.

In adding a_1 the barrier remains the same. This reasoning can be iterated in setting C to $C \cup a_1$. Then in adding a_2 the barrier still remains the same, and to on.

Consequently, $WP_C = WP_{C \cup (Agt \setminus owner(B))}$. □

Corollary 8 *C is losing $\Rightarrow C \cup (Agt \setminus owner(B))$ is losing.*

Proof $WP_{C \cup (Agt \setminus owner(B))} = WP_C$. If C is losing, then $p_0 \notin WP_C$ then $p_0 \notin WP_{C \cup a}$ so $C \cup (Agt \setminus owner(B))$ is losing. □

Corollary 9 *For any barricade B for a coalition C, $Agt \setminus owner(B)$*

Proof Let us relax the problem in setting $G' = B_W$: if a C have a winning strategy to reach G then C can use the same strategy to reach $G' \supseteq G$.

Given this relaxation, let WP'_C be the set of winning positions in this relaxation. Trivially, B is a barricade for this extended problem (with the same reasons as before).

As B is a barricade, agents of $owner(B)$ have a strategy s to prevent C to be goal-winning. In applying s in the original problem, agents in $owner(B)$ prevents C to be goal-winning.

As a result, the coalition $C \cup (Agt \setminus owner(B))$ is losing. □

4.5 Overestimate a Biggest Winning Coalition with a Path

Property 10 *A winning coalition C_w can be computed from any path 'path' from p_0 to a winning position. Moreover, $C_w = \{a | \exists p \in Pos, p \in path \wedge owner(p) = a\}$*

Proof This coalition is the set of agents owning a position in the path. As a result, the strategy s that consists in following this path is trivially winning. Formally, $\forall 1 \leq i \leq |path|, (p = path(i)) \Rightarrow (\exists a \in Act, s.t.(s(p) = a) \wedge Edg(p, a) = path(i+1))$ is winning. In every position of this path (starting from the p_0), the position is owned by C_w and leads the game to the next position of this path until reaching the goal. □

This coalition is not necessarily the smallest one, maybe a smaller coalition (even the empty coalition) can be the smallest winning coalition. This coalition can be not comparable with the inclusion to C_w. Moreover, instead of reaching a goal, the path can stop on a winning position: from such position there exists a winning strategy leading to the goal.

This method can then be applied to reach a WP_C set, in this case the estimation is $C_w \cup C$.

4.6 Worst Case

Unfortunately,none of these assumptions can not help us to avoid the worst case. This case is the following: every coalition of size $|Agt|/2 + 1$ is winning and every coalition of size $|Agt|/2$ is losing. The *cost* function is the cardinal cost. In this case, we must (at least) enumerate $C_{|Agt|}^{|Agt|/2}$ coalitions (coalitions of size $|Agt|/2$) to test if one of these is winning. Pruning cannot reduce this number of checks, but caching can reduce the check time.

5 Algorithmic Solutions

5.1 Checkers Optimizations

Cache

We showed in section 11.3 and in section 3.2 that checking algorithms uses efficiently sub-estimations of WP_C. For every coalition C_s and C_b s.t. $C_s \subseteq C_b$ then $WP_{C_s} \subseteq WP_{C_b}$. We can then cache the WP_{C_s} set.

For instance, when computing WP_{C_b}, instead of starting from scratch, the WP_{C_s} set can be reused. Instead of computing $pre^*_{C_b}(G)$ we can compute $pre^*_{C_b}(WP_{C_s})$. For the $MinMax$ algorithm applied on C_b, when the search reaches a node $n = (i, p)$ s.t. $p \in WP_{C_s}$ then this node can be immediately replaced by a W leaf.

In section 6.3 more details are provided on the implementations of this cache.

To keep this cache as minimal as possible without loss of information, the set of computed elements evolves dynamically. If every direct super-coalitions of C_s are computed, then C_s can be removed since if any bigger coalition C_B needs an approximation of WP_{C_B} then there exists $C_b \subseteq C_B$ and $WP_{C_s} \subseteq WP_{C_b} \subseteq WP_{C_B}$, so WP_{C_b} is a better approximation of WP_{C_B}.

Of course, if the computer runs out of memory, it is possible to safe space in removing item of this cache.

The Cheapest Strategy in a Game

In section 7.2.4 we are interested by games where every action is combined to a price.

Let us define a function $cost_\alpha : Act \to \mathbf{R}$ be a cost function for every action. Then, let us define the function $cost_p$ by $cost_p : Play \to \mathbf{R}$ s.t. if pl is not goal-winning then $cost_p(pl) = +\infty$ let l be the index of pl s.t. $pl(l) \in Goal$. Then, the cost of this play is $\sum_{i=1..l-1} cost_\alpha(pl(i)_a)$.

Definition 8 *A strategy s_C is the cheapest strategy for a coalition C, if :*

- *for any strategy $s_{\bar{C}}$ for $\bar{C}$ s.t. $s_{\bar{C}} = argmax_{s \text{ strategy of } \bar{C}} cost(s_C \wedge s_{\bar{C}})$,*
- *for any other strategy s'_C for C and strategy $s_{\bar{C}2}$ for $\bar{C}$ s.t. $s_{\bar{C}2} = argmax_{s \text{ strategy of } \bar{C}} cost(s_C \wedge s_{\bar{C}2})$*

then $cost_p(s_C \wedge s_{\bar{C}}) \leq cost_p(s'_C \wedge s_{\bar{C}2})$.

There is different ways to compute the cheapest strategy for a coalition C in a game with a cost on every action. The simplest one is to add payoffs on the $MinMax$ search. Then $\alpha\beta$ pruning optimizations can be used to remove prematurely useless branches.

Here we propose a variant of the $pre^*_C(G)$ algorithm, because this algorithm has proven its performances over $MinMax$ without heuristics. Similar algorithms exists to compute optimal value in stochastic games like in [5], [4] or for Markov Decision Processes [9]. The principle consists in storing the best cost for every position in a map $m : Pos \to \mathbf{R}$ and updating this value until reaching a fix-point thanks to this process: $m^0 := \{p \in Goal \Rightarrow 0, others \Rightarrow +\infty\}$ m is updated at every iteration by this process : the cost c of every position p s.t. $owner(p) \in C$ is: $c = argmin_{\alpha \in Act} cost_\alpha(\alpha) + m(Edg(p, \alpha))$. Reciprocally

```
m ← {g ∈ Goal ⇒ 0, others ⇒ +∞};
repeat
    foreach Position p ∈ Pos do
        if owner(p) ∈ C then
            m(p) ← argmin_{α∈Act} cost_α(α) + m(Edg(p, α));
        else
            m(p) ← argmax_{α∈Act} cost_α(α) + m(Edg(p, α));
        end
    end
until m has not converged ;
```

Algorithm 1: the cost of the cheapest strategy in an game with costful actions

if $p \in \bar{C}$, $c = argmax_{\alpha \in Act} cost_\alpha(\alpha) + m(Edg(p, \alpha))$. This algorithm updates the values until reaching a fix-point. This specification is described by the algorithm 1.

More details about the symbolic implementation of this algorithm are discussed in section 6.5.

5.2 Coalition-Based Seeker

Introduction

In this section, we present an algorithm that looks for the cheapest winning coalition in searching the coalition-space. In using a checker (presented in section 3) this algorithm checks winning and losing coalitions intend to compute the cheapest winning one: C_{opt}

As we manipulate huge games, every checking operation is costful (and empirically, the time of every check for the same game is constant). Our objective is to look for the best coalitions to check, intend to get the cheapest winning coalition with a minimal number of checks.

Let us define the structure we manipulate in this framework:

- $Comp \subseteq Coal$ the set of checked coalitions
- $GW_e \subseteq GW$ the estimation of the winning coalitions
- $Los_e \subseteq \bar{GW}$ where $\bar{GW} = \{C | C \notin GW\}$, the estimation of losing coalitions
- $Unk_e = Coal \setminus Los_e \setminus GW_e$ be the set of unknown coalitions
- Let $C_{opt_e} \in GW_e$ be the estimation of the cheapest winning coalition

A simple way to solve our problem is to check every $C \in Coal$ and then C_{opt_e} is the cheapest winning coalition. To enhance performances, we used some theoretical properties to avoid such useless computations.

There are different ways to explore the coalition space, so we defined simple functions to explore it. In annexes 11.4 we present the tool we designed intend to enhance performances of the search, by allowing more subtle comportments. In our data-sets, such specialized tools were less efficient than the following basic search.

The search ends when $mc(C_{opt_e}) \setminus Los_e = \emptyset$, then $C_{opt_e} = C_{opt}$. mc is defined in section 7.2.1.

Definition 9 *A* check of a coalition C is useless *if C is checked and exists a coalition C' s.t. $C' \subset C$,$C' \in GW$ and C' is checked, or exists coalition $C' \supset C$, $C' \notin GW$ andC' is checked*

This definition is linked to the cache operation, presented later.

Formal Definition of the Coalition-Based Search

Given these definitions, let us define formally the problem. Let $OptCoal \subseteq 2^{Coal}$ be the set of sets of coalitions coalitions s.t. for any $Calc \in OptCoal$, in checking any coalition $C \in Calc$:

- exists an optimal coalition C_{opt} s.t. $C_{opt_e} = C_{opt}$
- $mc(C_{opt_e}) \setminus Los_e = \emptyset$

The problem is to compute $Opt_{Calc} \in OptCoal$ s.t. $|Opt_{Calc}|$ is minimal.

In our case, we restrict to an approximation of this minimum. Let us note that Opt_{Calc} have no useless checks.

Pruning

In section 4.2, we showed that: $\forall C \in Coal, \forall C' \supseteq C, C \in GW \Rightarrow C' \in GW$ and $\forall C \in Coal, \forall C' \subseteq C, C \notin GW \Rightarrow C' \notin GW$

As a result, each time a coalition C is checked, we can

- remove from Unk_e and add to Los_e the set: $\{C' | C' \subseteq C\}$ if C is losing
- remove from Unk_e and add to GW_e the set: $\{C' | C' \supseteq C\}$ if C is goal-winning

This property allows to prune a lot of coalitions in very few checks (the maximal bound is given in section 11.6). The most famous example is in a game that cannot be won. In checking only the coalition Agt (that is losing), we can prune the whole state-space, since every smaller coalition is losing.

Definition 10 *A* check of a coalition C is useless *a priori* (resp. *a posteriori*) *if C is computed in an iteration i there exists $i' < i$ (resp. $i' > i$) s.t. a coalition $C' \subset C$ and $C' \in GW$ is checked in the iteration i' or a coalition $C' \supset C$ and $C' \notin GW$ is computed in the iteration i'.*

Then, a check of a coalition C is useless *a posteriori* C is pruned by C', so in permuting the checking order C would not have been checked (because useless *a priori*).

In the following we extend the checkers s.t. when a coalition C is checked, either:

- $C \in GW_e$ or $C \in Los_e$ then no check is necessary and the correct value can be returned immediately
- $C \in Unk_e$ then a set GW_e or Los_e is updated thanks to this check.

Then, in every cases, when a checking operation is done, it cannot be useless *a priori*.

Barricade

Barricades are defined in section 4.4. Immediately from the corollary 8, given WP_C, if $C \notin GW$, then $C \cup (Agt \setminus owner(pred(WP_C) \setminus WP_C)) \notin GW$. In computing only for the predecessors of WP_C, we can add to GW_e a bigger coalition than C. Let B be this barricade.

Then, thanks to the corollary 9, any other barricade B' allows to prune $C \cup (Agt \setminus owner(B'))$. We used $pre^*_C(Goal)$ to compute WP_C. To detect another barricade, we simplify the problem like in the proof of the corollary, by adding B to the set of winning positions. Then we iterate on : $pre^*_C(pre^*_C(Goal) \cup B)$. Once again, if $p_0 \notin pre^*_C(pre^*_C(Goal) \cup B)$, we can iterate on $pre^*_C(pre^*_C(pre^*_C(Goal) \cup B) \cup B')$ where $B' = pred(pre^*_C(pre^*_C(Goal) \cup B)) \setminus pre^*_C(pre^*_C(Goal) \cup B)$ and so on.

Because of the add of every position of the barricade in the WP_C set, a position p s.t. $next(p) \subseteq B$, $owner(p) \in \bar{C}$ and $|owner(next(p))| \geq 2$ is in $pre^*_C(pre^*_C(Goal) \cup B)$ but could not be winning in buying only one agent. As a result, the number of barricades found by this algorithm is not an upper bound of the agents needed for victory. Moreover, some agents, not appearing in any barricade can be needed for victory like p for instance.

This optimization is not used in practice, since this check is as costful as a complete search of the *pre* algorithm and the pruning factor is not sufficient in our games to be interesting.

Getting a Maximal Losing/Minimal Winning Coalition

In section 11.6 we evaluate the maximal number of coalitions that are pruned in every check. There, we show that the bigger a losing coalition or the smaller a winning coalition is the more efficient is the pruning. As a result, computing the maximal losing coalitions or minimal winning coalitions prunes heavily the search space.

The algorithm 2 presents how to compute a maximal losing coalition given a losing coalition C. Note that a symmetrical algorithm exists to search of a minimal winning coalition.

```
Input: C: losing coalition
Output: C_m: maximal losing coalition
C_m ← C;
foreach Agent a from Agt do
    if C_m ∪ a is losing then
        C_m ← C_m ∪ a;
    end
end
return C_m
```

Algorithm 2: Maximal Losing Coalition

Property 11 Invariant : *C_m is losing.*

Proof In the first iteration $C_m = C$ and C is losing. For every iteration, a is added to C_m only if $C_m \cup a$ is losing. □

Property 12 *C_m is a maximal losing coalition.*

Proof Let us prove that C_m is maximal. Let us suppose that C_m is not maximal, then a coalition C'_m s.t. $C_m \subset C'_m$. So there exists an agent $a \in C'_m$ and $a \notin C_m$. There is an iteration where a coalition $C_{subm} \subseteq C_m$ checks $C_{subm} \cup a$ and do not add a, so $C_{subm} \cup a$ is winning. Or $C_{subm} \subseteq C_m$, so $C_{subm} \cup a \subseteq C_m \cup a \subseteq C'_m$.

Thanks to the property 2, as C'_m is losing, then every of its sub-coalitions are losing, or $C_{subm} \cup a$ is one of its sub-coalitions and is winning. So, C'_m cannot exist and C_m is a local maximal losing coalition. □

Trivially, the complexity of this algorithm is $|Agt| * \delta_c$ where δ_c is the complexity of the checker used. Let us recall that empirically δ_c is roughly constant. Moreover, the cache (presented in 5.1) is extremely well used in this setting because agents are added 1 by 1.

Let us note that the order agents are selected is not fixed. In selecting different orders, it is possible to get different maximal losing coalitions. The best case is to get losing coalition with the biggest cardinal.

A variation of this algorithm is proposed in annexes 11.9.

The Main Algorithm

In this algorithm we look for the cheapest winning coalition. We showed that if $mc(C_{opt_e}) \setminus Los_e = \emptyset$ then C_{opt_e} is a cheapest winning coalition.

As a result, given an estimation of C_{opt_e}, is computing every $C \in mc(C_{opt_e})$ we can prove either that C_{opt_e} is winning (if every $C \in mc(C_{opt_e})$ is losing), either there exists $C \in GW$ and $C \in mc(C_{opt_e})$. As $cost(C) < cost(C_{opt_e})$ then C_{opt_e} can be updated by C, until reaching the minimal value.

We showed that in checking the maximal losing coalitions, we often prune a far more coalitions in general and in $mc(C_{opt_e})$ than in checking elements than $mc(C_{opt_e})$. Moreover, let max be the set of maximal coalition, practically often $|Max| \leq |mc(C_{opt_e})|$. Let us present the search process in algorithm 3.

Proof

- *Termination:* In every iteration of the while loop, *to_test* checks a coalition $C \in Coal$:
  - either *to_test* is updated, then $C \in GW$, then C_{opt_e} is updated by C and $cost(C_{opt_e})$ decreases. As the cost is affected to a coalition and the set of coalitions is finite, the set of costs is finite. As $cost(C_{opt_e})$ decreases when updated by C, then the number of updates of *to_test* is finite
  - either *to_test* is not updated, then C is a losing coalition and $C \notin Unk_e$. After the iteration at least C is removed to Unk_e. As $Coal$ is finite, the number of checks is finite.
- *Correctness:* Les us show that $C_{opt_e} = C_{opt}$ is trivial from the definition of $mc(C_{opt_e})$. If no coalition from $mc(C_{opt_e})$ is winning then $C_{opt_e} \in \mathbf{C_{opt}}$.

□

The mc operator is capital because it contains unknown losing coalitions like C_u. C_u is not in Los_e then every maximal losing coalition C_m s.t.$C_u \subseteq C_m$ is not computed.

Output: Cheapest winning coalition
if *check(Agt) is losing* **then**
    return null
else
    $C_{opt_e} \leftarrow Agt$
end
$to_test \leftarrow mc(C_{opt_e})$;
while *there exists a coalition* $C \in (to_test \cap (Unk_e \cup GW_e))$ **do**
    if $check(C)$ **then**
        $C_{opt_e} \leftarrow C$;
        $to_test \leftarrow mc(C_{opt_e})$;
    else
        $remove_maximal_losing_coalition_from(C)$;
    end
end
return C_{opt_e}

Algorithm 3: The coalition-based seeker

As a result, any call of $remove_maximal_losing_coalition_from(C)$ removes a new maximum. Moreover, this call prunes (at least one) element of to_remove. A trace of the execution of this algorithm is presented in section 11.1

Complexity

In section 5.2 that, given a losing coalition, a maximum can be found in $|Agt|$ checks. Let δ_c be the check complexity.

Let δ_{mc} be the complexity to get an unknown or winning element from the mc set. Let us note that this complexity is linear on $Coal$ for every of our cost functions.

The complexity of this search algorithm is so :

$$|Max| * |Agt| * (\delta_c + \delta_{mc})$$

Let us note that in the worst case (presented in 4.6), $|Max| = C_n^{n/2}$.

5.3 Position-based Algorithm

This seeker is a kind of dual of the coalition-based resolution process. In the coalition-based method, we compute the WP_C set, in this approach, we compute the $winners_p$ set. As a result, $winners_{p_0} = GW$. We no not actually compute the smallest winning coalition but the GW set. Once done, we can know if C is winning in $O(|Agt|)$ instead of $O(\delta_c)$ (where δ_c is the cost of a check).

Let us present the main idea. In every position p, either:

- we buy the $owner(p)$, so a $p' \in next(p)$ must be winning
- we do not buy $owner(p)$, so for every $p' \in next(p)$, p' must be winning

The algorithm updates the $winners_p$ set for every position p where $winners_p$ represents the set of winning coalitions from p. Let us call $winners_p^0$ be the initialization and $winners_p^i$ obtained after i iterations. Let us present the updating process:

$$\begin{aligned} winners_p^{i+1} = & \\ & winners_p^i \\ & \bigcup \\ & (a)\{C \cup owner(p) | \exists p' \in next(p), C \in winners_{p'}^i\} \\ & \bigcup \\ & (b)\{C | \forall p' \in next(p), C \in winners_{p'}^i\} \end{aligned} \tag{1}$$

A trace of the execution of this algorithm is presented in Figure 1.

Property 13 *For every i, $C \in winners_p^i \Leftrightarrow p \in pre_C^i(Goal)$*

Proof We proceed by induction over i.

$i = 0$: for every coalition $C \in Coal$ and every position $p \in Pos$, if $p \in Goal$ then $p \in pre_C^0(Goal) = Goal$ and $C \in winners_p^0 = Coal$, else $p \notin pre_C^0(Goal)$ and $C \notin winners_p^0 = \emptyset$.

Let us suppose this property true for the iteration i, let us prove that this property is true for the iteration $i+1$.

- $C \in winners_p^{i+1} \Rightarrow p \in pre_C^{i+1}(Goal)$:
  - if $C \in winners_p^i$ then by hypothesis $p \in pre_C^i(Goal)$, as $pre_C^i(Goal)$ is increasing, $p \in pre_C^i(Goal)$
  - $C \notin winners_p^i$ (and C is added into $winners_p^{i+1}$)then:
    - by (a): $C \in \{C' \cup owner(p) | \exists p' \in next(p), C' \in winners_{p'}^i\}$. Let $p' \in next(p)$ s.t. $C' \in winners_{p'}^i$. Trivially $C' \in winners_{p'}$, then $C \supseteq C' \in winners_{p'}$. Then, by hypothesis, as $p' \in pre_C^i(Goal)$ and $p \in pred(p')$, then $p \in \{p | p \in pred(pre_C^i(Goal)) \wedge owner(p) \in C\}$ so $p \in pre_C^{i+1}(Goal)$.
    - by (b): $C \in \{C | \forall p' \in next(p), C \in winners_{p'}^i\}$. Then as C is in every $p' \in next(p)$, $C \in winners_{p'}^i$. By hypothesis, every $p' \in next(p) \in pre_C^i(Goal)$, then $p \in \{p | p \in pred(pre_C^i(Goal)) \wedge next(p) \subseteq pre_C^i(Goal)\}$ so $p \in pre_C^{i+1}(Goal)$.
- $C \notin winners_p^{i+1} \Rightarrow p \notin pre_C^{i+1}(Goal)$.
  - If $owner(p) \in C$ then, for every $p' \in next(p)$, $C \notin winners_{p'}^i$ (otherwise C would have been added to p by the rule (a)). Consequently for every $p' \in next(p)$, $p' \notin pre_C^i(Goal)$. Consequently, no successor of p is in $pre_C^i(Goal)$, then $p \notin pre_C^{i+1}(Goal)$
  - If $owner(p) \notin C$ then exists p' s.t. $C \notin winners_{p'}^i$ (otherwise C would have been added to p by the rule (b)).

    Consequently, $owner(p) \notin C$ and $next(p) \subseteq pre_C^i(Goal)$ is false, so $p \notin pre_C^{i+1}(Goal)$.

□

Corollary 14 *For every position p, $winners_p$ converges to the set of the winning coalitions in every p.*

Proof Immediate from the lemma 13. □

6 Implementation Aspects

6.1 Frontiers

Frontiers are used to represent (downward or upward) closed sets. Formal definition are given in 6. Such sets are frequently used to represent the coalition space (in both the coalition and the position based algorithms), thanks to the properties of 4.2.

For simplifications purposes, we focus on downward-closed frontiers, since upward-closed frontier are symmetrical.

Let us define by $\mathbf{S}$ the set of set to be represented. In this section, sets of sets are represented in bold and sets in capital letters.

Let us call $\mathbf{F} \subseteq \mathbf{S}$ the frontier represented.

Exhaustive Implementation

In this naive implementation, every S is exhaustively conserved in $\mathbf{F}$. Then, each time a set S, every $S' \subseteq S$ is added to $\mathbf{F}$. Of course, this implementation intractable: for the search space frontier $\mathbf{F}$ rise up to $|Coal|$.

Nonetheless, with an efficient hash function, this implementation allows to compute efficiently $S \in \mathbf{F}$

Symbolic Implementation

Instead of manipulating exhaustively every $S \in \mathbf{S}$, we manipulate only a least representative subset $\mathbf{S_m}$. $\mathbf{S_m}$ is the set of maximal sets of $\mathbf{S}$. Formally, $S \in \mathbf{S} \Leftrightarrow \exists S' \in \mathbf{S_m}, S \subseteq S'$.

To add S in $\mathbf{S}$: let us update $\mathbf{S_m}$ s.t. $(\mathbf{S_m} \setminus \{S' | S' \in \mathbf{S_m} \wedge S' \subset S\}) \cup S$. To test if S in $\mathbf{S}$, it is sufficient to test for every $S_m \in \mathbf{S_m}$ if $S \subseteq S_m$.

In this case, these operations have a complexity in $O(|\mathbf{S_m}| * |Agt|)$.

For the search-space, in the worst case (in section 4.6), $|\mathbf{S_m}|$ can increase up $C_{|Agt|}^{|Agt|/2}$ but no more. In this case, every coalition is either a sub-coalition or a super-coalition of a coalition of this border.

We present in section 6.2.4, how to use symbolic structures to represent S_m. In applications, s.t. the position-based algorithm (in section 5.3, this kind of representation dramatically enhance performances. Note that for symbolic sets, such frontier can represent in a single BDD upward and downward closures.

Let us note that with a symbolic representation using a BDD, the complexity to test a coalition is $O(|Agt|)$, and to add a new agent $O(|\mathbf{S_m}| * |Agt|^2)$ (the product of the sizes of the 2 structures).

6.2 Symbolic Representations of Structures

In this section we present incrementally how to design a transition system and to manipulate it with a symbolic representation.

Such symbolic representations can be implemented efficiently thanks to a BDDs. Lots of applications uses this feature, like [3] or [10].

6.2.1 Minimal Representations and Operations

First, let us consider symbolic representation of simple objects.

Let $Vars$ be an ordered set of variables. Let $Form$ be the set of propositional formulas with variables from $Vars$. Let Obj be the set of objects we want to represent by a formula.

Let us introduce a function $\mathit{code} : \mathit{Obj} \times 2^{Vars} \rightarrow Form$. The use of Obj allows to overload this function for the different objects we manipulate.

Let us define a reverse function $\mathit{decode} : Form \times 2^{Vars} \rightarrow \mathit{Obj}$, s.t. for every $O \in \mathit{Obj}$ and every $V \in 2^{Vars}$, $O = \mathit{decode}(\mathit{code}(O, V), V)$ and for every $F \in Form$, $F = \mathit{code}(\mathit{decode}(F, V), V)$.

Let us define $nb_vars : O \rightarrow \mathbf{N}$ be the number of variables needed to code/decode an object O. In the following, we suppose that $|V| = nb_vars(O)$.

We precise that if $O \in \mathit{Obj}$ is not a set, $|V| = nb_vars(O)$ and $F = \mathit{code}(O, V)$ then, in our definitions of code, F have a unique model $M_F \subseteq V$. We extends naturally decode to $\mathit{decode} : 2^V \times Vars \rightarrow \mathit{Obj}$, where $\mathit{decode}(F, V) = \mathit{decode}(M_F, V)$. If F have more models, decode is extended to : $\mathit{decode} : 2^V \times Vars \rightarrow 2^{\mathit{Obj}}$. This function is detailed with the set operations.

Let us define a function $swap_variables : Form \times Vars^* \times Vars^* \rightarrow Form$.

$swap_variables(F, L_V, L'_V)$ is defined if $|L_V| = |L'_V|$, if for every $v \in L_V$ then $v \notin L'_V$ and v have only one occurrence in L_V. Then $F' = swap_variables(F, L_V, L'_V)$ if the formula F where every occurrence of $v_i = L_V(i)$ is replaced by $v'_i = L'_V(i)$.

First, let us present $\mathit{code} : \{0, 1\} \times 2^{Vars} \rightarrow Form$, encoding a bit value b with the variables V. $nb_vars(b) = 1$, let $V = \{v\}$. $\mathit{code}(b, V) = v$ if $b = 1$ else $\mathit{code}(b, V) = \neg v$. Reciprocally, if F have one unique model M_F, $\mathit{decode}(F, V) = 1$ if $M_F \cap V = \{v\}$, $\mathit{decode}(F, V) = 0$ if $M_F \cap V = \emptyset$, otherwise the formula can represent both or none bits value, cannot be the code of a single value but of a set.

Then, let us represent the pairing operation: $\mathit{pair} : \mathit{Obj} \times \mathit{Obj} \rightarrow (\mathit{Obj} \times \mathit{Obj})$ the well-known operation $\mathit{pair}(a, b) = (a, b)$.

Let us define the code of a pair. We suppose that the codes of the 2 objects $O, O' \in \mathit{Obj}$ are disjoint, so their sets of variables $V, V' \subseteq Vars$ is s.t. $V \cap V' = \emptyset$. We extend code to $\mathit{code} : (\mathit{Obj} \times \mathit{Obj}) \times (Vars \times Vars) \rightarrow Form$. $\mathit{code}((O, O'), (V, V')) = \mathit{code}(O, V) \wedge \mathit{code}(O', V')$.

Let us present the reverse decoding operation. In a same way, we extend the decode operation to: $\mathit{decode} : Form \times (Vars \times Vars) \rightarrow (\mathit{Obj} \times \mathit{Obj})$ with the same setting: $\mathit{decode}(F, (V, V')) = (\mathit{decode}(first(F, (V, V'), V)), \mathit{decode}(\mathit{second}(F, (V, V'), V')))$.

Let us define $first : Form \times (Vars \times Vars) \rightarrow Form$ by the natural extension to the $first$ operation to $Form$ domain. $first(F, (V, V')) = \exists_{v \notin V} v.F$. Idem for second: $\mathit{second}(F, (V, V')) = \exists_{v \notin V'} v.F$.

As V and V' do not shares any variable and the code operation apply the $\wedge$ operator between the 2 formulas, then $first(\mathit{code}((O, O'), (V, V')), (V, V')) = first(\mathit{code}(O, V) \wedge$

$code(O', V'), (V, V')) = \exists_{v \notin V} v.(code(O, V) \wedge code(O', V')) = code(O, V)$. This property is symmetrical for *second*.

Note that, given a the bit representation and the pairing operation, we can represent the infinite tape of a Turing machine. So, every data structure can be represented thanks to these minimal formulas. Let us add more power and expressiveness to this representation.

Let us note that the pairing operation extends easily to the tupling operation. With similar reasoning we can get the ith projection of a tuple.

Let us extend this representation to integers from a finite subset $I \subseteq \mathbf{N}$ and exists k s.t. for every $0 \leq i \leq 2^k$, $i \in I$. Let B_n be the binary representation of $n \in I$. Let us represent B_n by a finite set on integers s.t. $\sum_{i \in B_n} 2^i = n$. Trivially,to represent $i \in I$, $nb_vars(i) = k$, let $V = \{v_1 \dots v_k\}$. Then, $code(n, V) = (\wedge_{i \in B_n} v_i) \wedge (\wedge_{i \notin B_n} \neg v_i)$. Let M_F be the model of F on V. $decode(F, V) = \sum v_i \in M_F 2^i$.

Note that the size of the representation of the elements of I is logarithmic on $|I|$.

6.2.2 Automaton Objects Representations

These transformations allows us to represent positions of a game. Independently of the games, our positions are tuples: $(Int \times Prop \times \cdots \times Prop)$, where Int is an integer value from a finite set and $Prop$ is a bit value representing a proposition of the game. More details about positions are described in section 7.1. Some positions are a tuple of positions, so still can be represented thanks to our formalism.

Actions and agents are coded by integers (from a finite set). A transition t is a triple (p, act, p') where p is the starting position, act the action played and p' the resulting location.

We manipulate during this report a symbolic representation of the automaton thanks to transitions. In order to simplify representations, we allocate to different objects of the automaton different variables. Let us define these sets now, given a transition (p, act, p')

- $V_{agt} \subseteq Vars$ for the agent playing in the starting position p. Symmetrically for the ending position p': $V'_{agt} \subseteq Vars$.
- $V_{prop} \subseteq Vars$ for the status of propositions in the starting position. Symmetrically V'_{prop} for the ending position.
- $V_p = V_{prop} \cup V_{agt}$ for the starting position p. Symmetrically, $V'_p = V'_{prop} \cup V'_{agt}$ for p'.
- V_{act} for the action act, in the transition.
- $V_{trans} = V_{prop} \cup V'_{prop} \cup V_{agt}$ for the whole transition function.
- V_{Coal} for the sets of coalitions of the frontiers (see section 6.1 for more details).

Given the definition, let us present how to generate a game through an example for the STRIPS games. In section 7.1.2, we present an extension of STRIPS for games. We show here how to generate a symbolic representation from such representation. The algorithm 4 present the method to generate such game. To avoid confusion, a STRIPS action is called a rule. Intend to simplify the construction, let the set of propositions *pre* of STRIPS be the set V_{prop} be equals and *add*, *rem* be the same propositions than the set V'_{prop}.

Output: T:Transition Function
$T \leftarrow \emptyset$;
foreach *Rule* $r = (pre, add, rem, owner, next)$ **do**
$T \leftarrow T \vee ((\wedge_{v \in pre} v) \wedge (\wedge_{v \in add} v) \wedge (\wedge_{v \in rem} \neg v) \wedge (\wedge_{v'_i \notin (add \cup rem)} (v_i \Leftrightarrow v'_i)))$;
end

Algorithm 4: Generator of a transition function of a multi-agent STRIPS

In this symbolic representation, if the preconditions are met in the starting location, the resulting location have its *add* propositions true and *rem* propositions false. Other propositions are unchanged.

Given such game, let us define symbolically how to get easily simple symbolic representation of interesting sets. For instance, get the predecessors of a set of locations is a useful problem (as presented in sections 3.2, 11.5 or 5.2).

Given a set of positions S_P, let $F_{S_P} = code(S_P, V'_p)$, $F_T = code(T, V_{trans})$, the symbolic representation of the set of predecessors of S_P is : $F_{pred} = \exists_{v \in (V_{act} \cup V'_p)} v.F_T \wedge F_{S_P}$

Given F_{pred}, the representations of biggest subset of *pred* s.t. $owner(pred) \subseteq C$ is obtained by : $F_{pred} \wedge code(C, V_{Coal})$.

In section 11.5 we are interested by the smallest path from p_0 to a set of position $P \subseteq Pos$ We proceed in computing a level graph like in an optimization of the well-known Ford-Fulkerson algorithm.

This graphs split the game into n disjoints sets $S_1 \dots S_n$. For every position $p \in S_i$ the shortest path from p_0 is a path of size i. Then, the distance $d = min_{i \in \mathbf{N}} S_i \cup P \neq \emptyset$ is the minimal distance between p_0 and P. Then, a shortest path is a path s.t. $(p_0, \dots, p_d)$ s.t. $p_{i+1} \in next(p_i)$, $p_i \in S_i$ and $p_d \in P$. Thanks to the definition of the level graphs, such path exists.

In this section, we showed how to represent symbolically objects of a game. Then, we presented how to generate a game from a symbolic description and some basic operations over this game.

6.2.3 Sets Representation and Manipulation

Symbolic representations are extremely efficient to represent compactly sets with similarities between its elements. Let us present the basic set operations : generating the empty-set, generating a singleton, union, intersection and complement).

Let S, S' be 2 sets whose objects can be represented thanks to the same variables V. Let us define the basic set operations in a symbolic setting:

- $code(\emptyset, V) = \bot$
- $code(\{O\}, V) = code(O, V)$
- $code(S \cup S', V) = code(S, V) \vee code(S', V)$
- $code(S \cap S', V) = code(S, V) \wedge code(S', V)$
- $code(S \setminus S', V) = code(S, V) \wedge \neg code(S', V)$
- $decode(F, V) = \{decode(M_F, V) | M_F \text{ model of } F\}$.

- The test $O \in S$ is equivalent to test $code(\{O\} \cap S, V) = code(\emptyset, V)$

Let us note that $\top$ have every model in V, then $\top$ is the set of every elements O that can be expressed on V. This note let us extend the representation to symbolic sets.

Let us consider the following example, manipulating a set of transitions T, we want to add the constraint "player 2 plays always and only after player 1".

Let us code the player 1 as the starting position player: $F_{a_1} = code(a_1, V_{agt})$. Idem for player 2 as ending position player : $F_{a_2'} = code(a_2, V'_{agt})$.

Then: $T \wedge (F_{a_1} \Leftrightarrow F_{a_2})$ codes for this new transition system. Every transition t s.t. agent 2 is not the successor of the agent 1 does not satisfy this new formula that is T. t have no model, so $t \notin decode(T, V_{trans})$.

6.2.4 Closed Sets of Sets Representation

Closed sets of sets are used in section 6, section 5.1 and in section 5.3. The symbolic representation of such sets highly improved the performances of our tool. Let S be the set of objects we manipulate and $\mathbf{S} = 2^S$ be the set of set. To represent such sets of sets, we represent $S_s \in 2^S$ by a vector of the elements of S. So, for instance, let $S = \{a, b, c\}$ and $S_s \in 2^S$ s.t. $S_s = \{a, c\}$ then we represent the set S_s in the variables $\{v_a, v_b, v_c\}$ by the vector $\{v_a = 1, v_b = 0, v_c = 1\}$.

Manipulation of symbolic sets of sets is presented in [11]. In our case we are interested by closed sets, this problem have been discussed for data mining purposes in [16].

Let us define the variables $V_S \in Vars$ s.t. $|V_S| = |S|$. For every $s \in S$ there exists a unique v_s. Let $\mathbf{A} \subseteq \mathbf{S}$, let us represent $code(A, V_S)$.

- We redefine $code : \mathbf{S} \times 2^{Vars} \rightarrow Form$ by $code(S_s, V_S) = (\wedge_{s \in S_s} v_s) \wedge (\wedge_{s \notin S_s} \neg(v_s))$
- $code(\emptyset, V_S) = \bot$
- The test $S_s \in \mathbf{A}$ is equivalent to test $code(S_s, V_S) \wedge code(\mathbf{A}, V_S) = \bot$
- Add an upward closed set S_u: $A_u = code(\mathbf{A}, V_S) \vee (\wedge_{s \in S_u} v_s)$. Any evaluation of set $S_s \supseteq S_u$ have valuation $M_{S_s} \supseteq \{v_s | s \in S_u\}$, so $S_s \in A_u$
- Add a downward closed set S_d: $code(\mathbf{A}, V_S) \vee (\wedge_{s \notin S_d} \neg(v_s))$
- Union and intersection set operations have the same *code* function than the sets. The complementary operation is trickier and not proposed.

It is important to note that a set $S_s \subseteq S$ have 2 possibles codes: one to represent a traditional sub-set of S and one to represent a closed element of $\mathbf{S}$ and cannot be mixed.

This closed set of set representation improves time and space mean-time during the execution. Because of a bug in the Java BDD-interface we get, we could not test the efficiency on ZBDDs like authors in [16] do.

6.3 Cache

The cache *Cache*, presented in section 5.1 stores a subset of WP_C during the checking a operations.

A first implementation of this cache is done with a map operation. The problem is that, given a coalition C, we would like to get $\bigcup_{C' \subseteq C} WP_{C'}$. Unfortunately, getting coalitions for every subsets is intractable. Some solutions, more efficient, consisted in testing only the direct sub-coalitions of C, adapting the search using this fact (in adding agents one by one). Moreover, as the set of computed coalitions remains empirically small, it is possible to test for every coalition in *Comp* if this coalition is a subset of C or not.

Nevertheless, in storing symbolically the cache, with same representation than in the position-based algorithm (presented in section 6.4), we can more get more efficiently the cache. In this representation, *Cache* is defined by: $Cache \subseteq 2^{Coal \times 2^{Pos}}$. Then $Cache_C = \exists_{v_i \in V_{Coal}}.(Cache \wedge code(C, V_{coal}))$.

Let us note that for the BDD representation, variable ordering is extremely important for performances purposes (as presented in [20]). Our BDD interface allowed us to select efficiently a good variable ordering.. We do not detail here the use of BDD variable ordering, but oppositely to the position-based algorithm, $V_{coalition}$ must have a smaller value than V_{pos}, intend to keep correct performances.

6.4 The Position-Based Algorithm in a Symbolic Structure

In this implementation, we update the map $winners : position \rightarrow Coal$, given the proposition presented in section 5.3. In symbolic setting, this map can be confused with a set.

In every iteration, Then, we combine these successors with T to get the set $M : starting \times ending \times coalitions$. Depending on the rule applied, application, we can get C or $\bar{C}$ predecessors with an and operation on M with the BDD representing C or $\bar{C}$. Other operations are less relevant to our application.

The position-based algorithm is presented in algorithm 5.

6.5 Computing a Cheapest Strategy

In section 5.1 we presented an algorithm that computes the cheapest strategy s for a coalition C in a game. The problem is that the cost of every position must be exhaustively enumerated, which is intractable. Fortunately, some algebraic data structures are designed to handle such graph problems with efficiency (for instance the algorithm used in [9] can handle million of states).

These structures are similar to symbolic propositional formulas, but for every evaluation of this formula is not $\top$ of $\bot$ but a value from a set of constants Q. The exact definition given in [1], let us simplify it by: a formula f is an algebraic formula from a set of variables V if f is a function $f : \{0,1\}^{|V|} \rightarrow Q$.

An ADD is a data structure that represents efficiently such formulas where $Q = \mathbf{R}$, peculiarly for graph or matrix operations. In few words, ADDs are a data structure very similar than BDDs, but, instead of associating leafs by $\bot$ and $\top$, leafs are associated to real values.

ADDs operations are boolean, arithmetic and abstractions:

Input: G:game
Output: Set of winning coalitions
$winners \leftarrow \{(goal \Rightarrow \top), (\text{others} \Rightarrow \bot)\}$;
repeat
  $winners' \leftarrow swap_variables(winners, V_p, V_p')$;
  $winning_coalition_and_predecessors \leftarrow T \wedge winners'$;
  $buy_agent \leftarrow winning_coalition_and_predecessors$;
  $buy_agent \leftarrow \exists_{v \in (V_p' \cup V_{act})} v.buy_agent$;
  %buy_agent, is a map $p \rightarrow \{winners_{p'} | p' \in next(p)\}$
  foreach *Agent agt* **do**
    $buy_agent \leftarrow buy_agent \wedge (code(agt, V_{agt}) \Rightarrow (code(agt, V_{coal}) \wedge buy_agent))$;
  end
  %buy_agent is a map $p \rightarrow \{winners_{p'} \cup owner(p) | p' \in next(p)\}$
  $winning_coalitions_by_action \leftarrow$
  $\exists_{v \in V_p'} winning_coalition_and_predecessors$;
  %$winning_coalitions_by_action$, is a map
  %$p \times Act \rightarrow \{winners_{p'} | p' \in Edg(p, a)\}$
  $buy_successors \leftarrow \top$;
  foreach *Action act* **do**
    $buy_successor_by_a \leftarrow$
    $\exists_{v \in V_{act}} v.(winning_coalitions_by_action \wedge code(act, V_{act}))$;
    %$buy_successor_by_a$ is the map
    %$p \rightarrow winners_{Edg(p,a)}$
    $buy_successors \leftarrow buy_successors \wedge buy_successor_by_a$;
  end
  %$buy_successors$ is the map
  %$p \rightarrow \bigcap_{a \in Act} winners_{next(p)}$
  $winners \leftarrow winners \vee buy_successors \vee buy_agent$;
until *the fix-point for winners is reached* ;
return $winners_{p_0}$

Algorithm 5: The position-based algorithm

- The boolean operation is the traditional *ite* applied on ADDs, where $\top$ is replaced by 1 and $\bot$ by 0. Other traditional propositional logic operators can then be defined.
- Arithmetic operations: $(+, -, min, max \dots)$. $Apply(f, g, op)$ is equivalent to associate the algebraic formula h, s.t. for every valuation of $V = \{(0, \dots, 0), (0, \dots, 1), \dots, (1, \dots, 1)\}$ the algebraic formula $h = \{(0, \dots, 0) \rightarrow op(f(0, \dots, 0), g(0, \dots, 0)), (0, \dots, 1) \rightarrow op(f(0, \dots, 1), g(0, \dots, 1)), \dots, (1, \dots, 1) \rightarrow op(f(1, \dots, 1), g(1, \dots, 1))\}$
- Variable abstraction: such abstraction remove a variable by abstracting it (like the $\exists$ and $\forall$ operators do in formulas. Let us restrict to closed and commutative operators applied on variables 1 by 1 (some specifications allows to apply operators on sets of variables, causing problems of associativity). Let us call abs_v^{op} such operator of abstraction. Let us define the substitution operation of a formula F: $F_{v \leftarrow F'}$ is F where every occurrence of v is replaced by F'. Then, $abs_v^{op}(F) = op(F_{v \leftarrow \top}, F_{v \leftarrow \bot})$.

The implementation we propose is a variant to the one proposed on [9], which is efficient even on gigantic state-space. The generic algorithm is presented in algorithm 1 and in algorithm 6 we propose the variant for symbolic structures.

Let T be the transition function of the game, represented with to a BDD. Let $ADD_{gen} : BDD \times \mathbf{R} \times \mathbf{R} \rightarrow ADD$ be an function s.t. $ADD_{gen}(b, v_1, v_2)$ is b where v_1 replace the $\top$ node and v_2 replace $\bot$ node.

The function *swap_variables* is extended for ADDs with the same definition than the function for BDDs presented in section 6.2.1

This algorithm can be extended to the position-based way: it is possible to compute, for every position p, the cost of every coalition. This idea could not implemented since our ADD library was a too much time-consuming and to our knowledge, no ADD library are implemented nor interfaced for Java.

Let us note that such technique allows the use of a cache. For every $C' \supseteq C$, let m_C (resp $m_{C'}$) be m for C (resp. C'), then for every position p, $m_C(p) \geq m_{C'}(p)$. So instead of starting computing m_C from scratch, m_C can be initialized for every p by $m_C(p) = min(m_{C_1}(p), \dots m_{C_k}(p))$ for every $C_i \subseteq C$

7 Discussion on Games and Costs

7.1 Games

7.1.1 Autistic Games

These games have been designed to test our software. Every agent plays "autistically" front of her own solo-game without interacting with someone else. Agents plays circular order.

These games are convenient to set the winning coalition in by setting for goal "every agent reaches its solo-game goal state". As agents do not interact, every agent that can choose to win or lose in its solo-game must be added to the winning coalition. Oppositely, if a agent plays a game automatically wins, no matter what she plays, then she is not needed in the minimal coalition. By her actions, she cannot prevent the game to win, even if she is an enemy.

Input: G : Game, C : $Coal$
Output: Cost of a cheapest strategy of C
$next_cost \leftarrow (g \in Goal \Rightarrow 0, others \Rightarrow +\infty)$;
foreach *Action act* **do**
    $trans \leftarrow \exists_{v \in V_{act} \cup V_{Agt}} v.T \wedge code(act, V_{act})$;
    % $trans$ is the transition function when act is selected
    $Q_{act} \leftarrow ADD_{gen}(trans, cost(act), 0)$;
end
%Q_{act} is the cost to add to $m(p)$ when act is played in p
repeat
    $m \leftarrow next_cost$;
    $only_C \leftarrow ADD_{gen}(code(C, V_{agt}), 0, +\infty)$;
    $only_{\bar{C}} \leftarrow ADD_{gen}(code(\bar{C}, V_{agt}), 0, -\infty)$;
    foreach *Action act* **do**
        $cost_{act} \leftarrow Q_{act} + next_cost$;
        %$cost_{act} : p \rightarrow \mathbf{R}$ is cost from p if a is played
        $cost_{act,C} = cost_{act} + only_C$;
        $cost_{act,\bar{C}} = cost_{act} + only_{\bar{C}}$;
        % if $owner(p) \in \bar{C}$ then $cost_{act,C} = +\infty$
    end
    $next_cost \leftarrow cost_{act}$;
    %let $next_cost$ be a selection of an action
    foreach *Action act* **do**
        $next_cost \leftarrow max(next_cost, cost_{act,\bar{C}})$;
        $next_cost \leftarrow min(next_cost, cost_{act,C})$;
        %as $cost_{act,\bar{C}}(p) = -\infty$ or $next_cost, cost_{act,C} = -\infty$ then in one min/max
        %operation, $next_cost(p)$ is selected, the ordering of these operations is not
        %important
    end
until *m has not converged* ;
return m

Algorithm 6: Compute the cheapest strategy for C

These games are interesting to present some synchronization problems. For instance, if we have a set of independent processes that terminate only if they all simultaneously enters in a configuration.

Our framework allows to answer problems like: "which systems do we need to control in order to terminate, or how can we shorten the execution by adding more agents or which is the best compromise?"

Nonetheless, this games may look too specific to generate general heuristics. For the objective presented higher, in defining a goal where everyone must win hers local game, they is only one minimal winning coalition. We showed in proposition 17 that such game can be solved in $|Agt|$ checks. The resolution of this problem lacks of difficulty.

7.1.2 STRIPS Games

STRIPS is a well-known language of specification of planing domains (as presented in [7]). This language represents the world (or planing domain) thanks to propositions and allows to interact on this world in using actions that modify some of these propositions.

Let us recall quickly a simple propositional STRIPS formalism. The world is described by a set of propositions P. Every state of the world is generated from P: a state s is a subset of P : $s \subseteq P$. The state-space SS is then the set of states.

We define a subset of propositions $P_w \subseteq P$ *we want to achieve*. The goal state set $G \subseteq SS$ is the set of states s.t. if s is a state of G then $P_w \subseteq s$.

Let us define how to act on the world. Every action act is a tuple (pre, add, rem) : $(2^P \times 2^P \times 2^P)$. From a given state s, this action is triggerable if the preconditions are met: $pre \subseteq s$. If act is triggered, then some propositions in world are modified by the effects of act. Formally, when act is triggered, the world in a state s evolves to a new state s' where $s' = s \cup add \setminus rem$.

This formalism allows to represent compactly and intuitively the planing domain. The transformation of a planing domain into a 1-agent game with a reachability objective is trivial. Let us now extend this formalism to a multi-agent setting.

First, the states of the multi-agent world are extended to contain the current agent number. Formally, the multi-agent state-space is $SS \times Agt$. A state s_m is denoted by a couple $s_m = (s, agt)$ where $s \in SS$ and $agt \in Agt$. Then, we extend the formalism to affect a set of actions for every agent and determine the next agent. Formally, the extended actions are defined by the tuple: $(pre, add, rem, owner, next) : (2^P \times 2^P \times 2^P \times Agt \times Agt)$. The meaning of pre, add and rem remains the same but, an action is now triggerable from $s_m = (s, agt)$ if $pre \subseteq s$ and $owner = agt$ and the resulting multi-agent state is $s'_m = (s \cup add \setminus rem, next)$.

This formalism allows us to easily generate games.

7.1.3 Dynamic Adversarial Planing Games

Adversarial planing is well introduced in [10], they present it by a 2-agent game with a reachability objective (for a concurrent-game setting). In these games, the 2 agents are a_s for agent system and a_e for agent environment. a_s has a reachability objective and a_e plays arbitrarily, but, they suppose that a_e may wants to prevent a_s to succeed.

[13] and [14], extends this adversarial planing framework to a large multi-agent setting: the action of the environment represents the action of every other agents. This operation

allows to split the large multi-agent game to a set of 2-agent games, potentially easier to solve. If an agent have a winning strategy s in this sub-game, then s is a winning strategy in the real game.

We propose to extend turn-based adversarial planing framework thanks to our tool. Informally, we propose to a_s to pay to forbid a_e to trigger a set of transitions. Our tool computes the smallest cost for these changes.

Such operations can model a cooperation between 2 players in the initial problem: if a player have a strategy to reach the goal, she can "sell" to other players actions not related to her strategy. Our framework offers direct solutions, because we can increase the number of agents with winning strategies in the sub-games thanks to this cooperation. Moreover, our framework is linked to coalition formation problems, including social warfare (e.g. increasing prices of highly demanded transitions, offer the possibility to cancel the prevention of transitions during the play, selling different transitions during the play...).

Let us define formally the problem. Given a 2-agent game :

$G_2 = (\{a_e, a_s\}, Pos_{G_2}, owner(G_2), Act_{G_2}, Edg_{G_2}, p_{0G_2}, Goal_{G_2})$, we define a set $removable$: $2^{2^{2^{Pos_{G_2} \times Act_{G_2} \times Pos_{G_2}}}}$ of sets of sets of transitions s.t. for every $\mathbf{R}, \mathbf{R'} \in removable$, if $T \in \mathbf{R}$ and $T' \in \mathbf{R'}$, either $T \cap T' = \emptyset$ or $T = T'$. We add next a function val : $2^{2^{2^{Pos_{G_2} \times Act_{G_2} \times Pos_{G_2}}}} \to \mathbf{R}$. The problem is: get the minimal value v s.t. exists $\mathbf{R} \in removable$ s.t. $G'_2 = (\{a_e, a_s\}, Pos_{G_2}, owner_{G_2}, Act_{G_2}, Edg_{G_2} \setminus r, p_{0G_2}, Goal_{G_2})$ and the player a_s is goal-winning in G'_2 and $val(\mathbf{R}) = v$.

Changing such game into a turn-based game G'' is very simple. Let us present it formally :

- Let us define $Agt_{G''} = \{a_e, a_s\} \cup Agt_d$ where $Agt_d = \{a_T | \exists \mathbf{R} \in removable, T \in \mathbf{R}\}$. We add one fictive player for every set of transition of the elements of $removable$

- $Pos_{G''} = Pos_{G_2} \cup P_f \cup \top$ where $P_f = \{p_t | \exists \mathbf{R} \in removable, \exists T \in \mathbf{R}, t \in T\}$ stands for fictive positions. For every transition in the sets of transitions in the set of set of transitions of removable, we generate a fictive position.

- for every $p \in Pos_{G_2}$, $owner(p)_{G''} = owner(p)_{G_2}$. For every $p_t \in P_f$, as every set of transitions from the elements of $removable$ are disjoint, exists a unique set of transitions T from the elements of $removable$ s.t. $t \in T$. Then $owner(p_t)_{G''} = a_T$

  We set the same owner to every transition is a set of transitions of a set of set of transitions of $removable$.

- $Act_{G''} = Act_{G_2} \cup \{W, B\}$. W stands for win and B for back.

- $Edg_{G''} = (Edg_{G_2} \setminus T_r) \cup T_f \cup T_b$ where $T_r = \bigcup_{\mathbf{R} \in removable, T \in \mathbf{R}} T$, $T_f = \{(p, act, p_t) | \exists p_t \in P_f, t = (p, act, p_t)\}$ $T_b = \{(p_t, B, edg(p, act)) | \exists p_t \in P_f, t = (p, act, p_t)\} \cup \{(p_t, W, \top) | \exists p_t \in P_f, t = (p, act, p_t)\}$

- $p_{0G''} = p_{0G_2}$

- $Goal_{G''} = Goal_{G_2} \cup \top$

We replace the successor p' of every transition $t = (p, act, p')$ that can be removed in the original problem to the dummy position p_t. p_t is linked to $\top$ and to p'. Let T be the set of transitions s.t. $t \in T$, then $owner(p_t)''_G = a_T$. If a_T is an enemy agent then a_T plays B in p_T otherwise a_T plays W and leads to direct victory. As a result, is a_T is bought, then the original transition cannot be triggered.

Then, for every $\mathbf{R} \in removable$, for every T in $\mathbf{R}$ is linked a unique agent a_T. Then removing every transitions T of $\mathbf{R}$ is equivalent to buy the coalition $C = \{a_T | T \in \mathbf{R}\}$, so $cost(C) = val(R)$. If there do not exists any $\mathbf{R} \in T$ s.t. $C = \{a_T | T \in \mathbf{R}\}$, then $cost(C) = +\infty$.

Using this transformation, it is possible to transform the initial problem in our problem. The complexity to change this algorithm is trivially $|Edg|$.

Let us note that the number of agents increases up to $|\{T | \exists \mathbf{R} \in removable, T \in \mathbf{R}\}|$. As the problem of finding the minimal coalition is NP, the number of removable transitions should not explode up to its maximal possible value : $|Edg|$.

Note that symmetrically, it is possible to use a similar trick to pay to add some transitions to the game.

7.2 Cost Functions

7.2.1 Definition and Exploitation

Let us define the cost function $cost : Coal \rightarrow \mathbf{R}$. To certify that the optimal coalition C_{opt} is minimal, a way to proceed is to apply $cost$ on every coalition in the search space. This is equivalent to look for the minimum of an unsorted set, whose size is $2^{|Agt|}$.

To avoid this useless enumeration when possible, we link to every $cost$ function, the minimal check function $mc : Coal \rightarrow 2^{Coal}$. $mc(C) = \{C' | cost(C') < cost(C) \land \forall C'' \supset C', cost(C'') \geq cost(C)\}$

Informally, $mc(C)$ denotes the set of maximal coalitions cheaper than C.

Property 15 *If $C \in GW$ and if for every $C' \in Coal$, $C' \in mc(C)$ and $C' \notin GW$ then $C = C_{opt}$*

Proof For every cheaper coalition C'', either every of its super-coalitions are more expensive than C, then $C'' \in mc(C)$ and is losing, either there exists set of coalition S s.t. for every $C_S \in S, cost(C_S) < cost(C)$ and $C'' \subset C_S$. Let us consider the coalition $C_S = \mathrm{argmax}_{C_S \in S}\, cost(C_S)$. Then every of the C_S super-coalitions are more expensive than C, so $C_S \in mc(C)$ and $C_S \notin GW$. Thanks to the pruning property (property 2) as this C_S is losing, every of its sub-coalitions are losing, so C'' is losing.

We showed that, if for every $C' \in Coal$ s.t. $C' \in mc(C)$ and $C' \notin GW$ then every coalition cheaper than C is losing. As a result, C is the cheapest winning coalition.

□

Using this property, given an optimal coalition $C_{opt} \in GW$, instead of computing the cost and the winning property of every coalition C in the search space, it is sufficient to do this test only every $C \in mc(C_{opt})$.

7.2.2 Cardinal Cost

This function is the simplest and the most specific one. This can be applied in domains where the cost of every agent is constant, for instance the cost of items to garbage after the play.

Trivially, $cost(C) = |C|$ and $mc(C) = \{C' || C'| = |C| - 1\}$.

7.2.3 Monotonic Cost

The monotonic costs are useful to represent the cost of agents with different salaries. Then, the addition of costs for every agent forms a monotonic function. Moreover, super-additive and sub-additives functions can be used as a monotonic cost, representing the increasing/decreasing costs of synchronization for instance. Of course, the cardinal cost is a peculiar case of the monotonic cost.

A function $cost_m : Coal \rightarrow \mathbf{R}$ is monotonic if for every coalition $C \in Coal$ and $C' \subsetneq C \Rightarrow cost(C') < cost(C)$.

Note that the cost is not necessarily proportional to the number of agents. For instance a 3-agent coalition C can be cheaper than a 2-agent one, but no sub-coalition of C can be cheaper one than C.

For instance, let us define 2 groups of agents in different geographic places. If we form a mono-group coalition, adding one agent in this group is less expensive than adding one agent from the other group, but the cost still increase when an agent is added.

The $mc(C)$ function can be computed in computing the maximal cheaper coalitions maximums of C. We have the same properties as presented in section 4.2 : every super-coalitions of a more expensive coalition than C is more expensive and every sub-coalition of a cheaper coalition than C is also cheaper. As a result, the coalition-seeker techniques can be used, like the one presented in section 11.9.

7.2.4 Action Cost

The cost function "cost by action" associates a cost to every play, depending on the actions used during this play. Let us define this function $cost_{act} : Coal \rightarrow \mathbf{R}$ by $cost_{act}(C)$ is the cost of the cheapest strategy for C.

Trivially, adding an agent in the coalition reduces the total cost. Consequently, $cost_{act}$ is monotonic decreasing and the cheapest coalition given $cost_{act}$ is Agt.

The computation of this cost function is a very simple game problem, but is harder to solve in a symbolic representation. Such algorithm is presented in section 5.1.

7.2.5 Combo Cost

This cost function associates a cost to every action in the game, plus the cost of to hire agents in the coalition. This cost can represent the cost of salaries of agents and the resources they consume during their work.

Formally: $cost(C) = cost_{agt}(C) + cost_{act}(C)$, where $cost_{agt}$ is a monotonic increasing cost function and $cost_{act}$ is a monotonic decreasing cost function. This cost function is interesting since it opens 2 problems: we want to get the cheapest winning coalition without computing neither costs and checks, and this function is not monotonic, even if

it allows some pruning over the coalition space. Let us present some properties of these functions.

Let us define a coalition C s.t. $cost_{agt}(C) \geq cost(C_{opt})$. Then, since the cost function $cost_{agt}$ increases, every coalition $C' \supset C$ have a cost $cost_{agt}(C') \geq cost(C_{opt})$. Reciprocally with $cost_{act}(C) \geq cost(C_{opt})$, then every sub-coalition of C is more expensive than C_{opt}.

These assumptions (similar to property 2) allows to prune over the search-space of the cost.

Moreover, these assumptions allows to underestimate the price of a coalition : $cost(C) \geq (argmax_{C' \subset C} cost_{agt}(C')) + (argmax_{C \subset C'} cost_{act}(C'))$. Then if this underestimation is greater than C_{opt}, the cost of this function does not need to be computed.

Of course, sometimes, computing a cost for a coalition C allows to prune bigger coalitions and refines the approximation, so in avoiding to compute $cost(C)$ we can have to compute costs of more coalitions.

Let us note that an closed set ADD can store efficiently the previous computations. Details about closed sets is presented in section 6.2.4, ADDs are presented in [1].

7.2.6 Unconstrained Cost

In the most general case, some cost functions can be expressed without monotonicity. For instance, a agent can be an intermediate between two agents: adding this agent reduces communication cost, reducing the total cost.

This cost function is the most general one, but getting its minimum is as hard as searching the minimum of an unsorted set of size 2^{Agt}. Nonetheless, this function can be reduced to a monotonic cost function.

We can redefine the search space inclusion relation by: $C \sqsubset C' \Leftrightarrow C \subset C' \wedge cost(C) < cost(C')$. Doing so, coalitions C that are more expensive than one of their super-coalitions C' are removed from the cost search space.

This function does not changes the outcome of the search, since if C is winning then C' is winning (thanks to the property 2), so as $cost(C) \geq cost(C')$ then C cannot be the cheapest winning coalition. Consequently, the cheapest winning coalition for $\subset$ is the cheapest winning coalition for $\sqsubset$.

Of course, using this operation imply to explore the whole search-space at least once, which is intractable in our case.

8 Experiments and Results

8.1 Experimental Settings

The goal of this training period is to build a tool that computes efficiently the cheapest coalition, we present here experiments we ran on our framework.

We use the STRIPS games (presented in 7.1.2) to generate our tests. Such formalism is interesting since it allows to easily generate games. Of course, as they are randomly generated, they do not capture any explicit and meaningful interaction between agents. Moreover, it is well known that real life cases may be quite different from laboratory case studies. Nevertheless, to our knowledge, there are no available multi-agent games data bases that would have given us a chance to test our tool. If the reader is aware of any data base, we would be pleased to challenge our application on such games.

Experimental settings for STRIPS are various by slightly modifying the generated game (e.g. the number of possible actions for every agent). Nevertheless, changing such settings does not make differences in the global shape of the results, so we discard the analysis of such parameters.

The most important parameter is the number of agents; the size of the game is exponential in the number of agents. Adding an agent doubles either the size of the game and the search space. Modifying independently the coalition-space to the game would have been more suitable for tests. This problem is quite difficult to handle, since games capture interactions between agents. In these experiments, we try to split the checking time (dependent only on the game) to the seeking time (dependent on the coalition space).

Every value in these graphs is the mean value of values computed from 100 randomly generated STRIPS data sets. We ran them experiments on a 2-intel core 6700 2.66GhZ CPU with 2Go of RAM, on a Fedora 12 distribution, with Java JRE 1.6. The BDD library used, JavaBDD is a java interface of BDD C++ libraries. In our case we use the interface of BuDDy, already used in research, like in [8]. These results are described in page 42. We managed to generate games out of 21 agents, so with an exhaustive size of $O(2^{21})$ positions, but we did not extend our search so far. The reason is that on some games (even with only 19 agents), we encounter very bad cases that blocked our tests for days, before getting stopped before terminating.

First, in experiment 1, we present the evolution of the number of checks of the base algorithm depending on the number of agents. We depict also number of checks needed to detect C_{opt}, for both the basic search and an heuristic-driven search. The heuristics used is the one that computes the distance from the shortest path, that is in this setting the most efficient.

Then, in experiment 2, we present the ratio of the mean checking time during a run, for the basic search and the heuristic search.

Finally, in experiment 3, we present the change in the mean checking time with and without caching, correlated with the total time of the search and the computational time of the position-based algorithm.

8.2 Results

Experiment 1 shows that, we are far from the exponential worst-case limit (presented first in [2] and in section 4.6), since the number of checks increases reasonably on the number of agents. Let us note that the number of checks between the detection of C_{opt} and the proof that C_{opt} is indeed optimal remains roughly constant (here nearly 5 checks). We do not find any reason that explains such behavior. Moreover, even with out general heuristic, the number of checks to detect C_{opt} is smaller than the base algorithm and this gap tends to increase. Nonetheless (as showed in experiment 2), computing the heuristic is costful.

We investigated the approach combining a search starting with a heuristic search and switching to the basic search when a minimal winning coalition is detected. Empirically, this approach is less efficient (in time and in number of checks) than the basic search. The reason is that the heuristic search does not prune efficiently the coalition space and the basic search detects easily the solution (at least in detecting one of its super coalition). Nevertheless, results indicate that for a search that approximates the solution, heuristic search can be better.

The experiment 2 shows how the time is spent during the search. First, for heuristic search, the computation of the heuristic costs half of the search time. Actually, for every coalition checked, the heuristic computes the shortest path, that is as costful as a check. Consequently, half of the time is spent in computing the heuristic.

The result given by the basic search is more informative on the time costs of the search. We showed previously that the search is efficient in number of checks, here we discuss its complexity. The experiment 2 shows that the ratio of the checking time on the total time is increasing with the number of agents. Remaining time is the time to build the game and seeking time. If this value was constant, then the seeking time would be proportional to the checking time. As this ratio increases, the seeking time is less and less important front of the checking time.

The experiment 3 shows that the time costs increases exponentially with the number of agents. First, let us note that the position-based algorithm and the coalition-based algorithm have similar performances (modulo a constant) on such generated games. Maybe there exists classes of games s.t. one approach is better than the other. Secondly, the cache shows its efficiency by reducing on 5 to 10 times checking and seeking operations. Finally, let us note that total seeking time operation is proportional mainly to the checking time operation. Even if the coalition-space increases exponentially, we manage to keep a number of checks low without heavy structural costs.

In combining every of these result, we show that:

- The size of the generated games is proportional on the size of the coalition space
- The complexity of searching the coalition space is strictly less important than the complexity of check (proportional on the number of agents). Indeed, as presented in section 5.2, the exponential part of the seek consists in testing every coalition in $mc(C_{opt})$. Let us note that, even if this part is exponential, this search represent only a small part of the total runtime.
- The number of checks remains logarithmic on the search space, showing the efficiency of the search.

9 Conclusion

The objective of this training period was to produce an efficient implementation to compute the cheapest winning coalition in a reachability turn-based game. As this problem is NP-complete [2], we were interested in the mean-case efficiency and in finding better complexity bounds.

During this training period, we developed a tool that solves this problem. We generated games using symbolic data structures (Binary Decision Diagrams), which efficiency is acknowledged: in particular, they allow to efficiently represent and manipulate state/transition objects. Then, we used general algorithms to check if a given coalition is winning to design two main algorithms: the *coalition-based seeker* and the *position-based algorithm*. The former explores the coalition space using the checking algorithms intending to find the cheapest winning coalition and checks that no cheaper coalition is winning. Thanks to our theoretical results, we managed to exploit the game and the coalition-space structure to significantly improve the search. We managed to bound the

number of checks to $|Agt|*|Max|$, where $|Max|$ is the number of maximal losing coalitions, even if the search remains exponential in the worst case. The latter algorithm computes, the set of winning coalitions for every position, but does not seek the cheapest winning one. Of course, this second algorithm can give its results to the former that searches the cheapest winning coalition without using a costful checking algorithm.

Next, we tested our software in order to validate its performances. On our randomly generated inputs, we showed that the two algorithms have a similar run-time complexity: The coalition-based seeker finds the cheapest winning coalition in $|Agt|$ checks in the mean case (so logarithmically in the search space size), while spending most of its run-time in checking the coalitions but not in seeking. Nevertheless, the search time increases exponentially in the (not that rare) worst case, because testing that every smaller coalitions are losing cannot be avoided. We presented a solution of this problem that we could not implement because of the Java interface.

Our major concern is to challenge our tool on large multi-agent games from real applications. We unsuccessfully wasted a lot of time in searching such data-bases. Consequently, the presented results may highly depend on the particular games we have generated. Further testing need being considered to assert the efficiency of our solutions in the general mean case. During my training period I had the opportunity[1] to visit the NICTA Institute of Canberra, Australia. I met experts in the fields of automated planing, diagnosis, machine learning, and mechanism design, where the addressed problems are tightly connected to ours. During my stay, we could extend our framework to handle various cost functions and games relevant in these domains (for instance, the cost of actions in traditional planning problem). I am thankful especially to Scott Sanner that supervised me there, gave me wise advices, precious libraries and the openness to research fields such as Markov Decision Processes or Reinforcement Learning. I would like to thank also Sylvie Thiébaux for initiating this training period.

To conclude, during this training period we brought a solution to the initial problem of computing the cheapest winning coalition in turn-based reachability game, exhibiting important interconnected parameters in the theoretical worst case complexity.

Then, we intensively tested our algorithms and managed to compute solutions by invoking the checker a number of time that is linear in the number of agents, that logarithmic in the size of the search space. Moreover, the seeking time is low compared to the checking time.

Notice that our solution has a large impact since, as we demonstrated, our framework allows to efficiently handle problems from connected domains, using linear time reductions of the latter to the former.

10 Bibliography

References

[1] R. Iris Bahar, Erica A. Frohm, Charles M. Gaona, Gary D. Hachtel, Enrico Macii, Abelardo Pardo, and Fabio Somenzi. Algebraic decision diagrams and their applications. In *ICCAD '93: Proceedings of the 1993 IEEE/ACM international conference*

[1] thanks to the "Ecole doctorale MATISSE"

on Computer-aided design, pages 188–191, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[2] Thomas Brihaye, Mohamed Ghannem, Nicolas Markey, and Lionel Rieg. Good friends are hard to find! In *TIME '08: Proceedings of the 2008 15th International Symposium on Temporal Representation and Reasoning*, pages 32–40, Washington, DC, USA, 2008. IEEE Computer Society.

[3] J. R. Burch, E. M. Clarke, K. L. McMillan, and David L. Dill. Sequential circuit verification using symbolic model checking. In *DAC '90: Proceedings of the 27th ACM/IEEE Design Automation Conference*, pages 46–51, New York, NY, USA, 1990. ACM.

[4] Georgios Chalkiadakis and Georgios Chalkiadakis. Abstract a bayesian approach to multiagent reinforcement learning and coalition formation under uncertainty, 2007.

[5] Anne Condon. On algorithms for simple stochastic games. In *Advances in Computational Complexity Theory, volume 13 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 51–73. American Mathematical Society, 1993.

[6] E. A. Emerson and C. S. Jutla. Tree automata, mu-calculus and determinacy. In *Proceedings 32nd Annual IEEE Symp. on Foundations of Computer Science, FOCS'91, San Jose, Puerto Rico, 1–4 Oct 1991*, pages 368–377. IEEE Computer Society Press, Los Alamitos, California, 1991.

[7] R. E. Fikes and N. J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. In J. Allen, J. Hendler, and A. Tate, editors, *Readings in Planning*, pages 88–97. Kaufmann, San Mateo, CA, 1990.

[8] Mike Gordon and Ken Friis Larsen. Combining the hol98 proof assistant with the buddy bdd package. Technical Report UCAM-CL-TR-481, University of Cambridge, Computer Laboratory, 15 JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom, phone +44 1223 763500, December 1999.

[9] Jesse Hoey, Robert St-aubin, Alan Hu, and Craig Boutilier. Spudd: Stochastic planning using decision diagrams. In *In Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence*, pages 279–288, 1999.

[10] Rune M. Jensen, Manuela M. Veloso, and Michael H. Bowling. Obdd-based optimistic and strong cyclic adversarial planning. In *In Proceedings of the Sixth European Conference on Planning*, pages 265–276, 2001.

[11] Timothy Kam. *Synthesis of Finite State Machines: Functional Optimization*. Kluwer Academic Publishers, Norwell, MA, USA, 1996.

[12] Sarit Kraus, Jonathan Wilkenfeld, and Gilad Zlotkin. Multiagent negotiation under time constraints. *Artif. Intell.*, 75(2):297–345, 1995.

[13] Viliam Lisý, Branislav Bošanský, Michal Jakob, and Michal Pěchouček. Adversarial search with procedural knowledge heuristic. In *AAMAS '09: Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems*, pages 899–906, Richland, SC, 2009. International Foundation for Autonomous Agents and Multiagent Systems.

[14] Viliam Lisý. Adversarial planning for large multi-agent simulations. In *AAMAS 2010*, 2010.

[15] Shin-ichi Minato. Zero-suppressed bdds for set manipulation in combinatorial problems. In *DAC '93: Proceedings of the 30th international Design Automation Conference*, pages 272–277, New York, NY, USA, 1993. ACM.

[16] Shin-ichi Minato and Hiroki Arimura. Frequent pattern mining and knowledge indexing based on zero-suppressed bdds. In *KDID'06: Proceedings of the 5th international conference on Knowledge discovery in inductive databases*, pages 152–169, Berlin, Heidelberg, 2007. Springer-Verlag.

[17] A. W. Mostowski. Games with forbidden positions. Technical Report 78, Univ. of Gdansk, 1991.

[18] Y. Narahari, Dinesh Garg, Ramasuri Narayanam, and Hastagiri Prakash. *Game Theoretic Problems in Network Economics and Mechanism Design Solutions*. Springer Publishing Company, Incorporated, 2009.

[19] Nils J. Nilsson. *Principles of Artificial Intelligence*. Springer, 1982.

[20] Richard Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *ICCAD '93: Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design*, pages 42–47, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[21] Tuomas W. Sandholm and Victor R. Lesser. Coalitions among computationally bounded agents. *Artificial Intelligence*, 94:99–137.

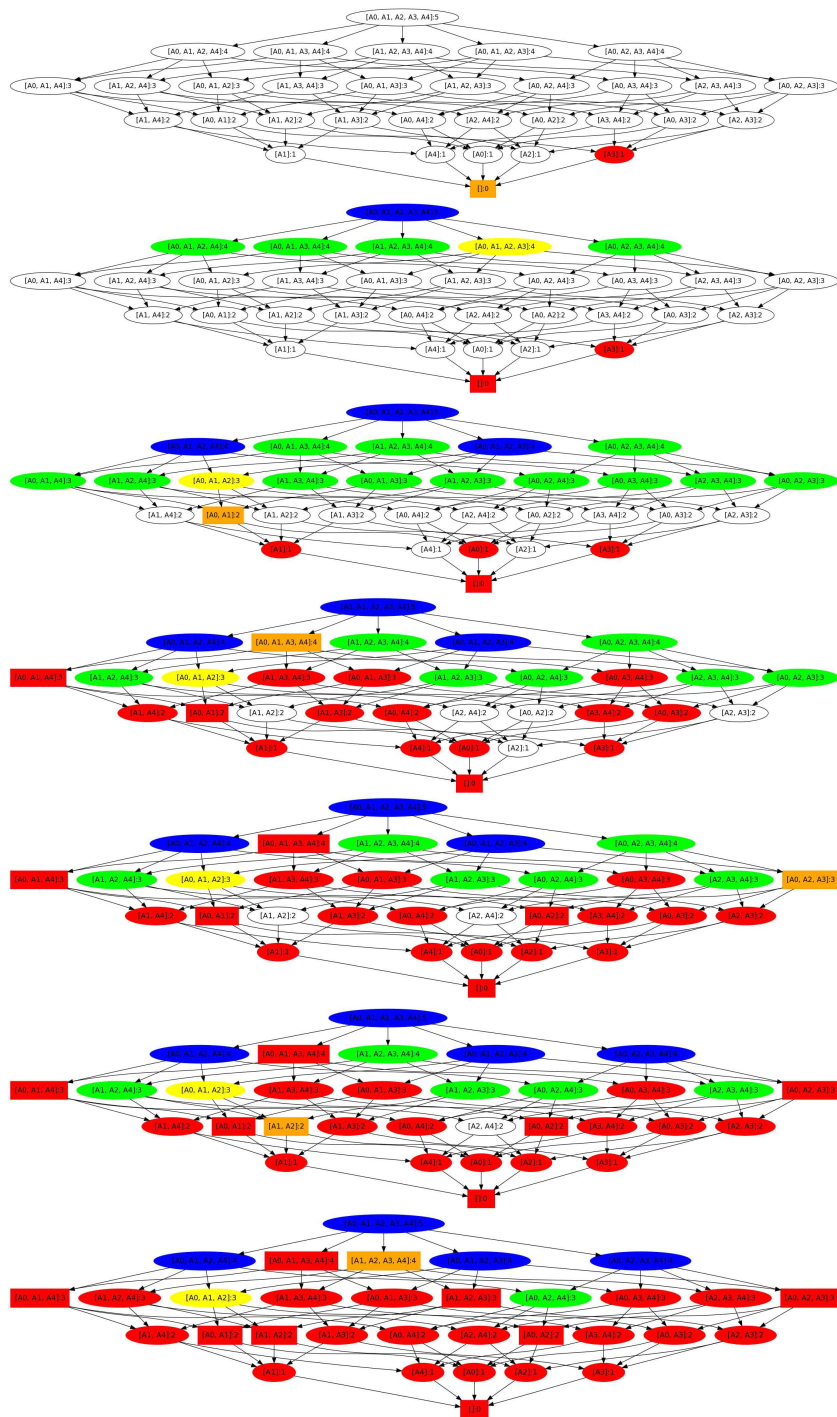
[22] Anthony L. Stornetta. Implementation of an efficient parallel bdd package, 1995.

11 Annexes

11.1 Plots and Draws

In table 1, we present a trace of execution of the coalition-based algorithm. We each iteration represent the knowledge of the coalition-space. A coalition is linked to its direct sub/super-coalitions. Let us present shapes and colors.

- White coalitions are unknown coalitions
- The orange (resp. light blue) coalition is the last checked and losing (resp. winning) coalition



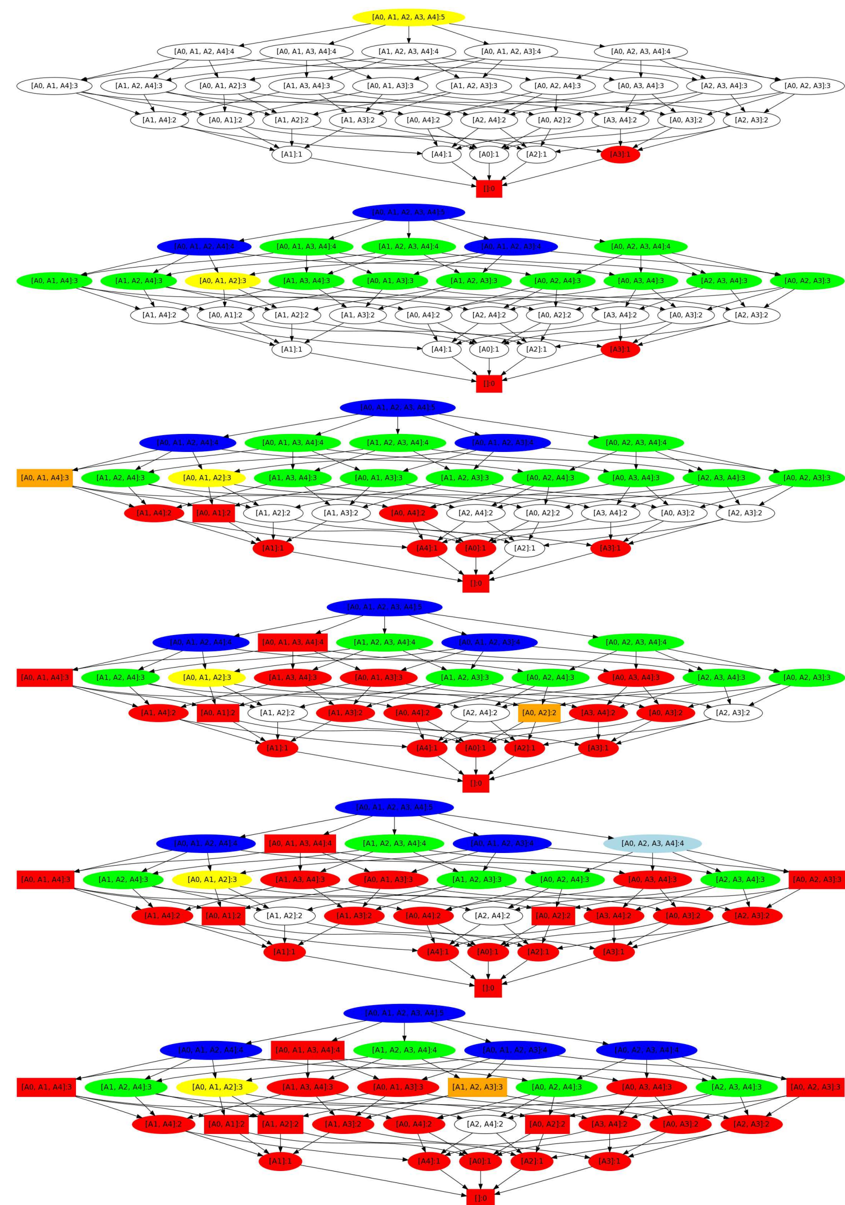


Table 1: An execution trace of the coalition-based algorithm

- The yellow coalition is the optimal coalition found
- Green coalitions are unknown but not needed to check
- Blue (resp. red) coalitions are winning (resp. losing) coalitions
- A rectangle coalition represents a coalition which value is stored in the cache.cached

Let us note some special events during the search:

- In the first iteration, the red (removed) coalition is due to a barricade from the empty coalition (as presented in section 5.2).
- Iteration 4 shows an example of pruning: coalition [A0,A1,A2,A4] is pruned thank to the check of [A0,A1,A2].
- In iterations 2, 3, 4, 8 and 11 an unknown coalition is taken from $mc(C_{opt})$. In other iterations, the losing coalition found is increased until reaching a maximal losing coalition.
- Let us note that in the last iteration, the value of the green coalition is not computed: we have proven that C_{opt} is the cheapest winning coalition.

11.2 Properties and Proofs

Barricades

Corollary 16 $\forall a \in Agt \setminus owner(B), WP_{C\cup a} = WP_C$.

Proof $WP_{C\cup a} \supseteq WP_C$ has been proven in section 4.3.

Let us show that $WP_{C\cup a} \subseteq WP_C$. As $a \notin owner(B)$, the same minimal barrier blocks C and $C \cup a$, so $WP_C = B_{WC} = B_{W(C\cup a)} = WP_{C\cup a}$.

□

Single Minimum

Proposition 17 *If C_{min} is the unique smallest winning coalition (in the sense of the inclusion), then this minimum is: $C_{min} = \{a|\text{the coalition } Agt \setminus a \text{ is losing}\}$*

Proof
- Let us prove that $C_{min} \subseteq \{a|\text{the coalition } Agt \setminus a \text{ is losing}\}$. Suppose that there exists an agent $a \in C_{min}$ and not in $\{a|\text{the coalition } Agt \setminus a \text{ is losing}\}$, so $Agt \setminus a$ is winning. $Agt \setminus a$ is not a superset of C_{min}. In this case, either :
  - $Agt \setminus a$ is a superset of a smallest winning coalition, contradicting the unicity of C_{min}
  - $Agt\setminus a$ have no smaller winning coalition, then it is a smallest winning coalition, contradicting the unicity of C_{min}.

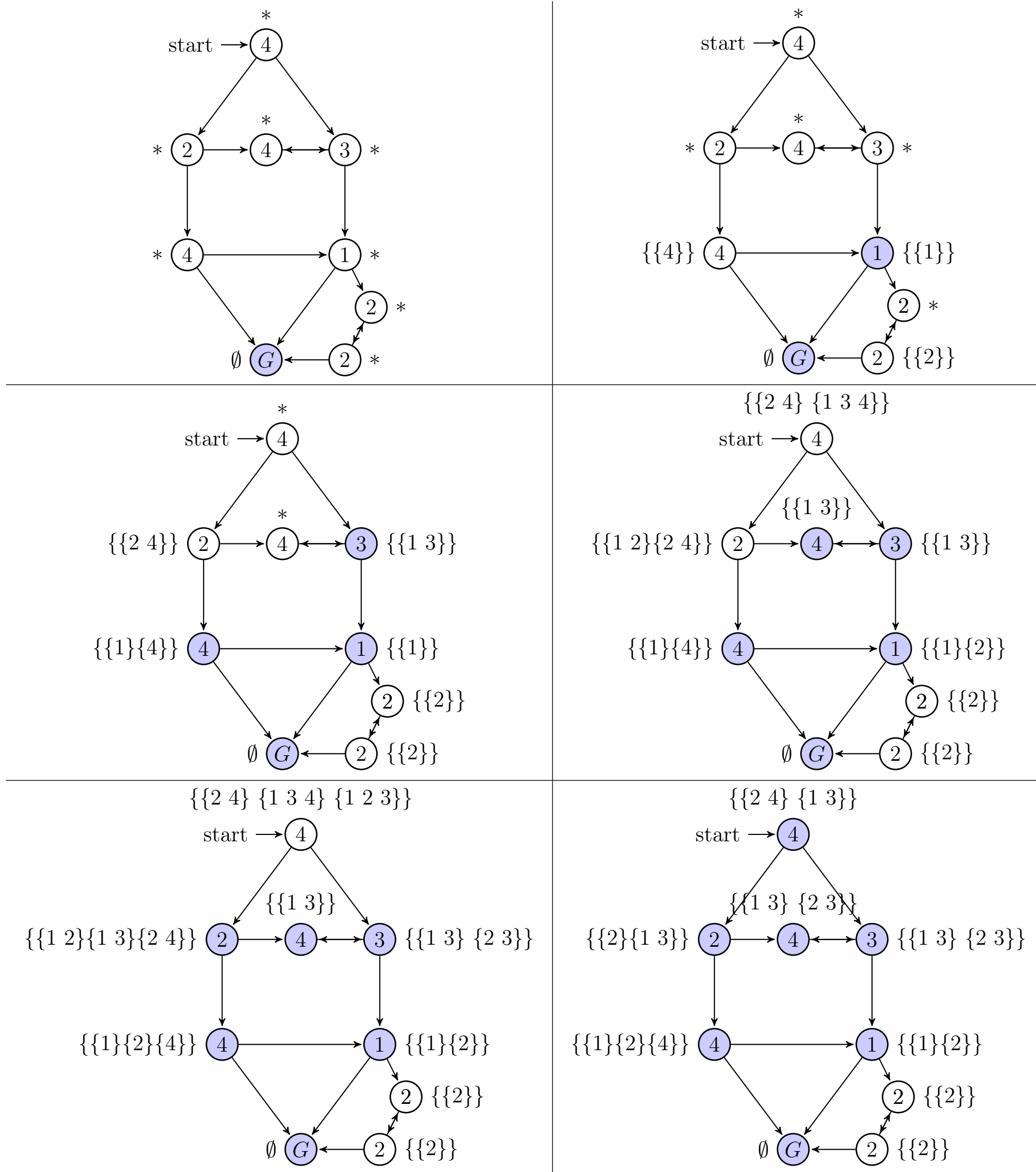


Figure 1: a trace of the executions of the $pre^*_{\{1,3\}}(G)$ (in blue) and the position-based algorithm (the sets). Note that the positions where the coalition {1,3} is computed as winning are the same on both algorithms. Coalitions printed are only the smallest ones, every bigger coalitions are winning

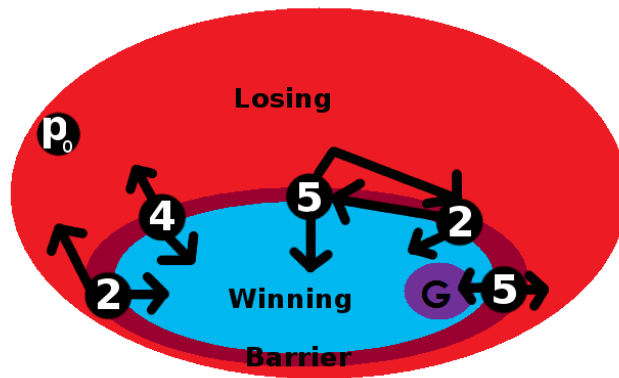


Figure 2: No agent out of {2,4,5} can go through this barricade}

- Let us prove that $C_{min} \supseteq \{a|\text{the coalition } Agt \setminus a \text{ is losing}\}$. Let us suppose that there exists an agent a in $\{a|\text{the coalition } Agt \setminus a \text{ is losing}\}$ not in C_{min}. This is impossible: $Agt \setminus a$ is a super-coalition of C_{min}, so must be winning. As C_{min} is winning then $Agt \setminus a$ must be winning.

As a result, if C_{min} is the unique smallest winning coalition (in the sense of the inclusion), then this minimum is: $C_{min} = \{a|Agt \setminus a \text{ is losing}\}$ □

For domain-specific games with a single minimum, the minimal coalition can be found in exactly $|Agt|$ tests, that is linear over the coalition-space, instead of an exponential search.

Corollary 18 *C_{min} is winning and $C_{min} = \{a|Agt \setminus a$ is losing $\} \Leftrightarrow C_{min}$ is the unique smallest winning coalition in the sense of the inclusion.*

The proof is presented in section 11.2

Proof $\Leftarrow$ direction of the proof is proved in section 17.

Let us show C_{min} is winning and $C_{min} = \{a|Agt \setminus a$ is losing $\} \Rightarrow C_{min}$ is the unique smallest winning coalition in the sense of the inclusion. Suppose there is a minimal winning coalition C' different from C_{min}. As C' is a minimal winning coalition, $C_{min} \nsubseteq C'$. So, exists $a \in C_{min}$ and $a \notin C'$. So, $Agt \setminus a$ a super-coalition of C' that is winning, contradicting the hypothesis.

□

11.3 *MinMax* Checker

We designed an algorithm very close to $MinMax$ (presented in section 3.1) to solve GW.

Let us represent the game tree of a reachability game by a tuple $T = (N, r, t, L, owner)$ where $N = 2^{\mathbf{N} \times Pos}$ is the set of nodes, $r = (0, p_0) \in N$ is the root of the tree, $L = \{W, L\}$ is the set of leafs, $t : 2^{Node \times (Node \cup L)}$ are the transition function, and $owner : N \to \{Min, Max\}$ the owner of a node. As values of leafs are important, let $W > L$. Let $next : N \to 2^N$ be the function $next(n) = \{n'|(n, n') \in t\}$

Let us consider the following construction of T :

- if $p_0 \in G$ then $r = W$ else $r = (0, p_0)$

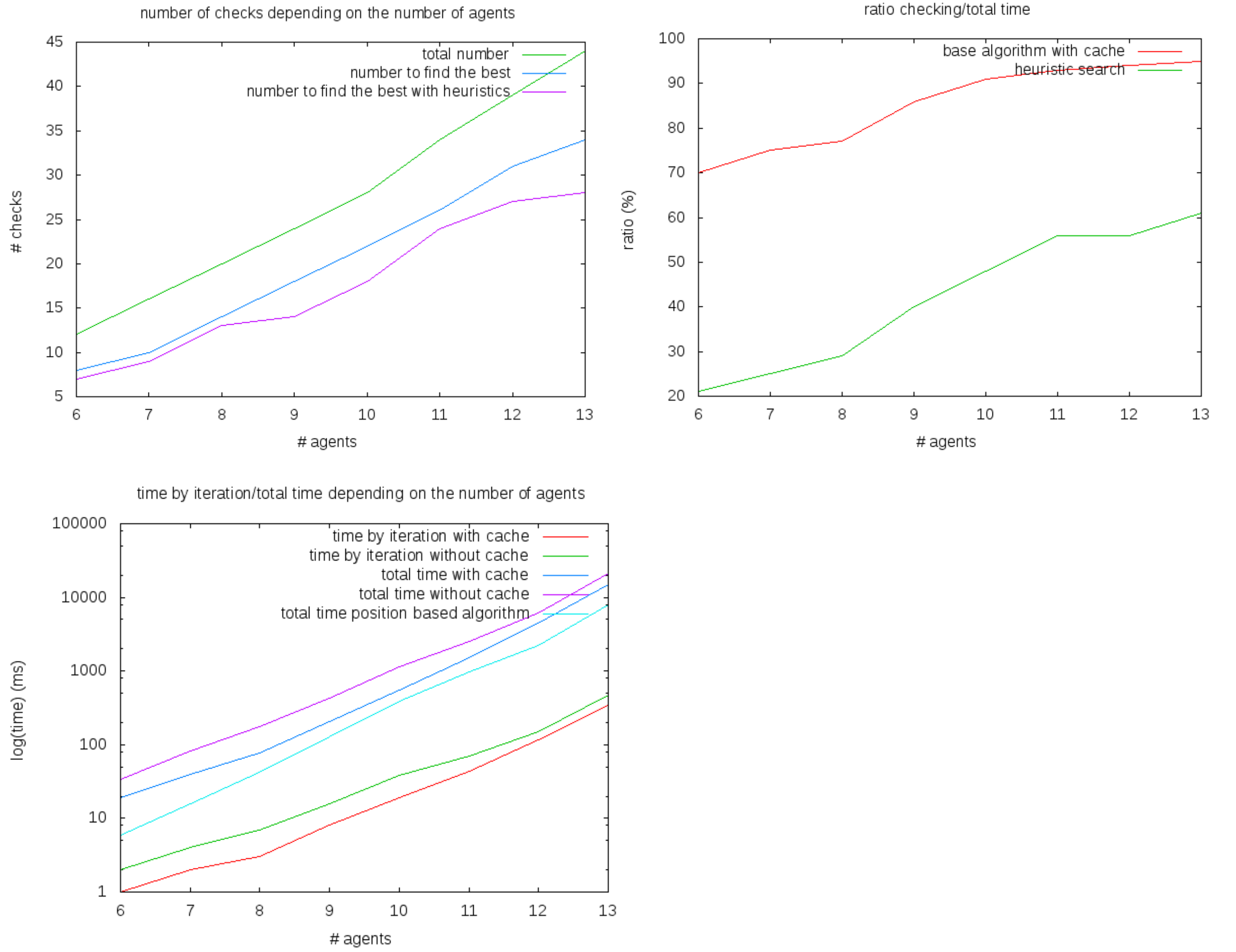


Figure 3: In experiment **1** (top left) we compare the numbers of checks done to find (with or without heuristics) the cheapest winning colition C_{opt} and to valid it. In experiment **2** (top right) we present the ratio between the total runtime spent in the search and the runtime spent in the checker. Then in experiment **3**, we present the runtime for one iteration with and without the cache, the total runtime for the search with and without the cache and the runtime of the position-based seeker

- for every node $n = (i, p)$, for every $p' \in next(p)$ if p' is in the branch containing n, then $(n, L) \in t$ else if p' is a winning position then $(n, W) \in t$ else $(n, (i', p')) \in t$ where i' is an integer not affected to any other node.

- let $n = (i, p)$, $owner(n) = max$ if $owner(p) \in C$ else n is a min node.

Definition 11 *Let s_G be a memoryless strategy for C in G and s_T be a strategy of Max in T s.t. for every position $p \in Pos$ $s_G(p) = p'$ and every node $n = (i, p) \in N$, $s_T(n) = (i', p')$ where $p' \in next(p)$ and $(i', p') \in t(n)$. By definition of T, (i', p') exists. Then we say that $s_G \equiv s_T$*

Property 19 $C \in GW \Leftrightarrow MinMax(T) = Max$

Proof Let us suppose s_G without loop and $s_T \equiv s_G$

Trivially, s_G is goal winning $\Leftrightarrow$ every prefix of play from p_0 reaches a goal without any loop $\Leftrightarrow$ s_T is winning for Min

Idem C is losing $\Leftrightarrow$ s_G can prevent any play to reach a goal $\Leftrightarrow$ s_T enters into a loop (thanks to the pumping lemma).

□

As there is only 2 possible outcomes, some pruning is possible: if a max node n have a winning successor then n is winning (no need to explore other branches, max have an action to ensure its victory). Idem reciprocally for losing nodes.

Property 20 *If exists $n = (i, p)$ s.t. n is winning, then $p \in WP_C$.*

Proof The proof is simple: if n is winning then there exists a winning-strategy s_T for Max from n. Then C has the winning strategy $s_G \equiv s_T$ from p. □

Oppositely, if $n = (i, p)$ is losing, every node $n' = (i', p)$ is also losing in the same way is false: let p we a winning position, if a branch of the tree contains any $p' \in next(p)$, then (n, p) is losing since every action is looping.

Property 21 *If $p \in WP_C$ then for every $n = (i, p)$ replacing n by W does not change the outcome of the $MinMax(T)$*

Proof Let us define T' equivalent to T where every node $n = (i, p)$ is replaced by W.

If $MinMax(T)$ is winning for Max, trivially $MinMax(T')$ is winning for Max.

If $MinMax(T)$ is winning for Min, then $\bar{C}$ have a strategy s_G s.t. no plays can reach a winning position. Then, the strategy $s_T \equiv s_G$ for Min prevents any play can reach $n = (i, p)$. So changing n to W does not change the outcome of $MinMax(T)$. □

This property allows us to replace directly any node $n = (i, p)$ by W when p is winning, permitting pruning in the search.

The cache (presented in 5.1) can store WP_C, allowing to dynamically reuse information for bigger coalitions. We updated the $MinMax$ algorithm to explore this tree directly from the original game. The current branch is given in parameters and the WP_C set is static. It simulates a $MinMax$ exploration of the tree. This is presented in algorithm 7.

This algorithm can be driven by an heuristic. As we focus on general games, we could not such heuristics but for some domain-specific applications, such tools highly increase performances (for instance in [14]). In our games, $MinMax$ is less performant than $pre^*_C(Goal)$. We think the reason is the exhaustive enumeration of the positions in $MinMax$.

11.4 Comportments of the Coalition Based Seeker

Intend to enhance performances of the search we produced a framework that combine a set of behaviors. The initial search pattern was first to look for a winning coalition, then to compute for one of its local minimum and finally prune unknown coalitions.

Intend to do this, we implemented simple behaviors we combined to improve results or reduce the number of checks. The Occam's razor is here right: the simplest solution (presented in section 5.2) is (far) the best. In fact, what happens is that the pruning factor of this solution outperforms the tricky search we tried to find. In spending lot of efforts in trying to find a minimal coalition, first the numbers of checks increases quadratically

```
Input: n = (i, p) : N
Output: n is a winning node for Max
if n ∈ L then
    return n = W;
end
if p ∈ WP_C then
    return true
end
foreach n' ∈ next(n) do
    if n is a Max node and MinMax(n') then
        add p to WP_C ;
        return true;
    else
        if n is a Min node and not MinMax(n') then
            return false;
        end
    end
end
if n is a max node then
    return false;
else
    add p to WP_C;
    return true;
end
```

Algorithm 7: *MinMax*

on the number of agent but also the seek time was highly exponential on the coalition space.

For space considerations we cannot present this work here, even if we spent lot of time to try to improve our methods. In a nutshell, we developed behaviors to use extensively the cache (in adding agent one by one), to proceed by dichotomy on the search space in splitting in 2 the distance between a checked coalition and a target, to look for local minimums when a winning coalition is found etc...

Empirically, the current algorithm still be better because small winning coalitions are statistically quickly detected.

11.5 Heuristics

An heuristic is a function $h : Coal \rightarrow \mathbf{R}$ that estimates the cost a coalition. If $C \notin GW$ then $cost(C) = +\infty$. If, for every coalition $C \in Coal$, $h(C) \leq cost(C)$ this heuristic is said *admissible* and can be used for an A^* algorithm. We suppose that heuristic search is a basic definition and A^* is known. In a few words, A^* stores a set $S = (s, h(s))$, where s is an solution of the problem and $h(s)$ an admissible estimation of its cost, ordered by increasing value of $h(s)$. Let $cost(s^*)$ be the cost of the best solution found, in computing every s s.t. $h(s) < cost(s^*)$. If there exists $s^{*'}$ s.t. $cost(s^{*'}) < cost(s^*)$ then this process is iterated on $s^{*'}$. Otherwise s^* is the optimal solution (since for every other solution s', $cost(s^*) \leq h(s') \leq cost(s')$.

In the implementation, since exploring the whole search-space is intractable, we limit the heuristic search to $H : 2^{Coal}$ s.t. $H = C \cup a$ where $C \in Comp_e \cap Los_e$

Such heuristics are designed to reduce the number of checks to find the minimal coalition, but also to find big losing coalitions. Such techniques are not relevant to our application, first because the number of checks remains low, secondly because we do not do any assumptions on games and search-space. For instance, if some agents have complementary competences could be specified to enhance heuristic search. Maybe for more specific games, such efficient heuristics could be designed.

A path to Winning Positions

We showed in section 4.5 that a winning coalition can be computed for every path from p_0 to a winning position. Here we get to compute the shortest one, that *a priori* contains the least agents. So, for a shortest path $path_C$ from the p_0 to WP_C, $h(C) = |C \cup \{agt | agt \in Agt \wedge \exists pos \in path_C \wedge owner(pos) = agt\}|$.

In symbolic representations, computing the shortest path (as presented in section 6.2.2) is as costful as a check. Given the performances in the current setting, this approach is not interesting.

Instead, we propose a new heuristic $h'_C : Coal \rightarrow \mathbf{N}$ defined for every coalitions $C, C' \in Coal$ s.t. $C \cup agt = C'$ where $agt \in Agt$. Let $occ : path \times Agt \rightarrow \mathbf{N}$ s.t. $occ(path, agt) = |\{p | p \in path \wedge owner(p) = agt\}|$. occ is the number of occurrences of agt is $path$. Then let us define h'_C by :

$$h'_C(C') = |C| + |path_C| - \frac{occ(path_C, agt)}{|path|}$$

With this definition of h_C, we get an order on the coalitions by: $|C| + |path_C|$, giving an over-estimation on the total cost, but *a priori* the bigger $|path_C|$ is the farther C' is. Then we order over frequencies of an agent in this path.

In practical cases, this approach is much less time-consuming than the first one (even if it is less accurate). Of course, this algorithm is useless if the game is played in a circular order (i.e. the agent $(i+1) mod |Agt|$ plays after agent i).

This heuristic is not admissible and so cannot be used to prune with A^*.

Relaxing the Problem: Forbid Actions Enemy Agents

This heuristic is a relaxation of the initial problem. Let us limit the actions of agents in $\bar{C}$ to an action act. This relaxation reduces the game to a simple planing problem: if there is a plan (a set of actions) for this "game", then there exists a strategy (using the same actions) for C. This heuristic is admissible and can be used to prune with A^*.

Trivially, if there is no plan, then the strategy $s(p) = a$ for every $p \in Pos$ s.t. $owner(p) \in \bar{C}$ prevents C to reach the goal, so C is losing. Otherwise, if enemy players have more actions then C could be losing also. Moreover, for any coalitions $C, C' \in Coal$, s.t. $C \subseteq C'$, $h(C) = 0$ implies $h(C') = 0$ and $h(C') = 1$ implies $h(C) = 0$. In this case, instead of a checker, such test can be used to remove trivially losing coalitions.

This heuristic have 2 problems:

- it provides only results 0 or 1 (depending if there is a plan or not). Such heuristic does not drive efficiently the search.
- the use of the planner is theoretically as costful as a check algorithm, even if in real case, heuristics can be designed intend to enhance planning performances for specific games. Let us note that the number of iterations in a variant of the *pre* algorithm for planing is the size of the shortest path in the game. Instead the number of iterations of the *pre* algorithm can go up to the size of the longest path in the game.

Nonetheless, shortest generated plan (which is a path) can be used by the path heuristic, as a serious approximation over the "difficulty" to reach the goal.

A variant has been tested in reducing the actions of only one agent agt to act. The drawbacks remains the same, but this time a real check operation is done. Lack of results (because of the games) prevented us to explore farther in this direction.

Winners First

This heuristic considers the agents who, in adding them, lead the game to the victory. The idea is to detect "powerful" agents that are crucial to the game.

We use the value $occ : Agt \rightarrow \mathbf{N}$ of occurrences of an agent in every minimal winning coalition.

Then $h(C) = \sum_{agt_i \in C} occ(agt)/|C|$

This heuristic is not admissible.

Coupling Heuristic

This heuristic extends the previous one in testing couples of agents that are winning together. We update a matrix containing L which $a_{i,j}$ the link value between agent i and

agent j. When a winning coalition is found, we add a "link value" l between every pair of agents. l is decreasing on the size of the winning coalition.

Then $h(C) = \sum_{a_i, a_j \in C} L(i,j)/|C|^2$.

This heuristic is not admissible. A problem is that in our games that are randomized, no interaction is represented in the games to link the agents. As a result, this heuristic is not very good efficient here. Nonetheless, for real problems needing an "union of abilities", this tool should become more efficient.

Single Minimum

Property 22 *C_{min} is winning and $C_{min} = \{a | Agt \setminus a$ is losing $\} \Leftrightarrow C_{min}$ is the unique smallest winning coalition in the sense of the inclusion.*

The proof is presented in section 11.2 This property allows to detect in $|Agt|$ checks that a coalition is the only minimal one. Note that these assumptions does not rely on the structure of games. They rely only the structure of the search space. As a result, these proofs are far more general than game theory and our framework.

Extended Reachability Objectives

To achieve reachability objectives, the C must exhibit a winning strategy s s.t. leading every time in $Goal$. This kind of objective is quite simple but can be extended to more interesting ones in modifying the game.

Objectives can be represented by a TL formula. We can then express :

- an objective O must be achieved before the second O' : the transitions of positions O leads to a copy of the game where O' is the objective. Every p in O have a unique transition to a p', copy of p in the image of the game.
- two objectives can be reached : idem but p leads to the copy of p_0
- a set of objectives must be reached without order : every time an objective is reached, the game is transfered into a copy (like when O must be achieved before O', where O is no more an objective, but other objectives must be reached.
- the set of positions P must be avoided : replace transitions to a position P by a transition leading to a failure position

As these objectives can be combined, some interesting objectives can be presented, like: if O is reached, then O' must be reached else if O'' is reached then O''' must be reached.

For symbolic representations, this operation is similar to add propositions to the transition function. When an objective O is reached, a variable v_O initially set to $\perp$ is set to $\top$ in the next position. Goals are then defined with these propositional values: a formula from v_O must be true.

11.6 Notes

Framework Extensible to Other Types of Games

The coalition-based algorithm thanks to its checker independence (presented in Section 5.3) is a very generic tool. For instance, if an appropriate checker is given, this tool can handle concurrent games and stochastic games.

In fact, the search space, the cost function, the cache and the seeking process remains the same. Only heuristics and some optimizations may be lost. For stochastic games, for instance, the cache can be reused to restart computations with a good approximation (as presented in the end of Section 5.1).

The barricade optimization can be generalized for concurrent games. An agent is rejected if, whatever she plays, she never changes the outcome of the action : being in or out of WP_C, whatever the possible resulting positions. Note that agents can be pruned one by one, but the removing of 2 agents implies that no combination of action can lead to a winning position (etc for more agents). For stochastic games, this property is not usable. The counterexample is a bit tricky and out of this work, to be presented here.

Many heuristics are no more usable. But, more specific game type algorithms can be designed.

Counting the Number of Subcoalitions of a Coalition

Let us count how many sub-coalitions have a coalition C. Reciprocally, the number of super-coalitions of C is the number of sub-coalitions of $Agt \setminus C$.

There is $C(1,k)$ coalitions in removing one agent to C, there is $C(2,k)$ coalitions in removing two agent from the k agents contained C, and so on. As a result, the number of sub-coalitions of C is $\Sigma_{i:1..|C|} C_k^i$ coalitions.

C and $\bar{C}$ can Draw

If a coalition C has a reachability objective, $\bar{C}$ aims at preventing the game to reach the goal. Nonetheless, $(\neg C \in GW) \nRightarrow \bar{C} \in GW$.

For instance, consider the following 2-player game. 2 agents play twice and the goal is reached if they repeat the same action. A coalition C containing only one agent cannot be goal-winning since the agent of $\bar{C}$ can play 2 different actions. The opposite coalition, containing only one agent too, cannot be goal-winning either.

No Loops Strategies

Proposition 23 *If $p \in WP_C$, there exists a goal-winning strategy from p for C without loops.*

Proof As $p \in WP_C$, then exists a minimal i s.t. $p \in pre_C^i(G)$. As i is minimal, then either $owner(p) \in C$ then exists $p' \in pre_C^{i-1}(G)$ s.t. $p' \in next(p)$ either for every $p' \in next(p)$, $p' \in pre_C^{i-1}(G)$. As a result, in selecting p' as a successor for p, any play goes to a successor found in an iteration. From p', detected in iteration i' in applying the same choice for the action p cannot be reached (since $i > i'$). By induction, we easily show that with this strategy, no play contains a loop. □

11.7 Future Developments

Combining Both Algorithms

Both algorithms can be used synchronously to solve the problem. For instance, in using the heuristics to compute a good small winning coalition, this information being added to the position-based search, allowing it to terminate earlier.

Partial Solutions

Many intractable problems tries to avoid wasting time in computing exact solutions in computing approximations. In our case, we can highly reduce the search time in the worst case with approximations.

Lower bounds can be proven : instead of checking for every cheaper sub-coalition, we can check coalition C' s.t. $cost(C') < cost(C_{opt}) - 2$. Such approximation may highly reduce the search space in the worst case and does not affect the mean case (since the search still looks for extremums).

For the cardinal cost, for instance, instead of testing that every coalition of size $|C_{opt}|$ are winning (so up to $C_{|Agt|}^{|Agt|/2}$), we can test this only for $argmin_{k<approx} C_{|Agt|}^{|Agt|+k/2}$ cutting down the number of iterations.

Paralleliationm

Our framework can highly be parallelized. BDDs have proven their scalability in [22], so the position-based algorithm can be extended to harder problems, as for checks in bigger games.

Moreover, parallel checks can be done synchronously. The search space may be harder to split without losses, but as long as it is sequential, the coalition-based algorithm can task checks.

So, our framework is highly scalable.

Randomization

When selecting the next coalition to check, using a naive strategy consisting in adding always smaller agents first is not optimal. The problem comes from the pruning, in adding agents with small identifier first in coalitions, such agents are pruned lot of times, but not bigger one. So, coalitions containing agents with big identifiers are detected only later on the search.

We think that randomizing the blind add of agents will improve performances of our framework, once again, further tests must be investigated.

Hard constraints

We would like to handle the fact that 2 agents cannot be teamed together, so no coalition containing such couple can be winning. In the current framework, it is possible to add this improvement in adding a constraint on the symbolic representing the winning/losing coalitions.

Mechanism Design, Pay for Private Information and Bounded Rationality

In our application, the private information of enemy player is their goal. We suppose the worst case : we cannot know this goal and they maybe want to prevent us to reach the goal. Mechanism design is first discussed in the introduction.

An interesting extension, allowing lot of applications, would by to buy to know the goal of agents. Knowing which agents have an objective common with us may be interesting. Such cost can represent an "extended talking time" with a manager during her work application.

Adaptation of the work presented in [21] can offer an interesting framework thanks to bounded rationality. For instance, the more time is spent with an agent, the more precise is our knowledge of her goals. Moreover, finding the best for computational time split between the search of an optimal coalition and increasing informations is an interesting point. Then, minimizing the regret between the solution found and the optimal one is a new framework.

11.8 Computing efficiently mc

We showed in the section 8.2 that the bottleneck of the search consists in computing is $mc(C_{opt})$ contains a winning or an unknown coalition and, if it is the case, computing it.

In the current setting, we enumerate exhaustively every coalition that is in $mc(C_{opt})$. Of course this approach is highly exponential: even if every coalition is losing, we enumerate them.

For the cardinal cost, this problem is nicely handled by ZDDs (as presented in [15]). A nice extension of our problem would add this tool to generate mc. The problem is that our BDD interface does not implement ZDDs (even if CUDD or BuDDy, interfaced thanks to this library does). Using ZDDs could solve this problem.

For other costs functions this problem is more tricky and needs more investigations.

With such symbolic tool, we can symbolically represent the set of unknown mc by $mc(C_{opt}) \cap Unke)$. Then given any affectation in this set, we can get an unknown coalition, avoiding enumerating exhaustively search space.

11.9 Improving the Maximal Losing Coalition algorithm

A more subtle approach to the initial one consists in keeping a set of maximal *computed* losing coalitions M. When checking C_{cheap}, we look for these maximums. If one of them is a local losing maximum then we remove it (it is useless). Else, if exists $m \in M$ s.t. $C \cup m \in Unk_e$, we start the search from M. If no coalition in M can be added to C_{cheap}, then there exists a maximal losing coalition C_m s.t. $C_{cheap} \subseteq C_m$. Then we declare that C_m is a local maximum. This algorithm avoids the exact seeking of every losing maximum of the search space. In every time, we do the best effort to remove a coalition and we avoid the computation of useless maximal losing coalition that does not prunes the mc space.