Decomposition of Multi-player Games

Dengji Zhao[1], Stephan Schiffel[2], and Michael Thielscher[2]

[1] Intelligent Systems Laboratory
University of Western Sydney, Australia
[2] Department of Computer Science
Dresden University of Technology, Germany

Abstract. Research in General Game Playing aims at building systems that learn to play unknown games without human intervention. We contribute to this endeavour by generalising the established technique of decomposition from AI Planning to multi-player games. To this end, we present a method for the automatic decomposition of previously unknown games into independent subgames, and we show how a general game player can exploit a successful decomposition for game tree search.

1 Introduction

Research in General Game Playing is concerned with the development of systems that understand the rules of previously unknown games and learn to play well without human intervention. Identified as a new Grand AI Challenge, this endeavour requires to combine methods from a variety of a sub-disciplines including Knowledge Representation, Search, Planning, and Learning [1,2,3,4]. An annual AAAI Contest has been established in 2005 to foster research in this area by evaluating general game playing systems in a competitive setting [5].

With this paper we contribute to the science of General Game Playing by tackling an important and open sub-problem: how can game tree search be improved by automatically decomposing a game into independent parts? The general value of decomposition has been widely recognised in AI Planning, where it is used to help solve large, complex problems arising in practical settings using a divide-and-conquer strategy [6,7,8]. In [9] we have shown how this method can be directly adapted to the special case of single-player games. This previous result provides the starting point for our generalisation to multi-player games. Specifically, we address the following two issues in the present paper: Given its mere rules, how can a previously unknown multi-player game be automatically decomposed into independent subgames? And how can a successful decomposition be exploited for a significant improvement of game tree search during play?

We begin (Section 2) with a brief introduction to the formal basis for our analysis, the Game Description Language [5]. In Section 3, we present a general decomposition method for multi-player games. This result is used in Section 4 to obtain a significant improvement of game tree search for decomposable games. In Section 5, we further improve our method in the special case of so-called impartial games. This is accompanied by both a formal complexity analysis and an overview of experimental results. We conclude in Section 6.

A. Nicholson and X. Li (Eds.): AI 2009, LNAI 5866, pp. 475–484, 2009.

2 Preliminaries

The Game Description Language (GDL) [5,10] is the standard language to communicate the rules of an arbitrary game to each player. It is a variant of first-order logic enhanced by distinguished keywords for the conceptualisation of games. GDL is purely axiomatic, that is, no algebra or arithmetics is included in the language; if a game requires this, the relevant portions of arithmetics have to be axiomatized in the game description.

The class of games that can be expressed in GDL can be classified as *n-player* ($n \geq 1$), *deterministic*, *perfect information* games with *simultaneous moves*. "Deterministic" excludes all games that contain any element of chance, while "perfect information" prohibits that any part of the game state is hidden from some players, as is common in most card games. "Simultaneous moves" allows to describe games like Roshambo, where the players move at the same time, while still permitting to describe games with alternating moves, like chess or checkers, by restricting all players except one to a single "noop" move. Also, GDL games are *finite* in several ways: All reachable states are composed of finitely many fluents; there is a finite, fixed number of players; each player has finitely many possible actions in each game state, and the game has to be formulated such that it leads to a terminal state after a finite number of moves. Each terminal state has an associated goal value for each player, not necessarily zero-sum.

A game state is defined by a set of atomic properties, the *fluents*, that are represented as ground terms. The leading function symbol of a fluent will be called a *fluent symbol*. One game state is designated as the initial state. The transitions are determined by the combined actions of all players. The game progresses until a terminal state is reached.

Example 1. Figure 1 shows the GDL rules[1] of "Double-Tictactoe". This game consists of two instances of the well-known Tic Tac Toe played in parallel.

The `role` keyword (lines 1–2) declares the players in the game. The initial state of the game is described by the keyword `init` (lines 3–7). The two Tic Tac Toe boards are described by fluent functions `cell1` and `cell2`, respectively. Constant `b` indicates a blank cell. The fluent function `control` defines whose turn it is.

The keyword `legal` (lines 8–13) defines what actions (i.e., moves) are possible for each player depending on the properties of the current state, which in turn are encoded using the keyword `true`. The game designer has to ensure that each player always has at least one legal action in every game state. In turn-taking games, players typically have "noop" as their only legal move if it is not their turn. In Double-Tictactoe, the player whose turn it is has to choose one of the two boards and a cell on this board to mark.

The keyword `next` (lines 14–23) defines the effects of the players' actions. For example, lines 14–16 declare that cell (`M`, `N`) on the first board is marked with constant `x` if `xplayer` executes action `mark1(M,N)`. The reserved keyword `does` refers to the actions executed by the players. GDL also requires the game

[1] We use Prolog notation with variables denoted by uppercase letters.

```
1  role(xplayer).
2  role(oplayer).
3  init(cell1(1,1,b)).
   ...
4  init(cell1(3,3,b)).
5  init(cell2(1,1,b)).
   ...
6  init(cell2(3,3,b)).
7  init(control(xplayer)).
8  legal(W,mark1(X,Y)) :-
9    true(cell1(X,Y,b)),
10   true(control(W)),
11   not terminal1.
   ...
12 legal(xplayer,noop) :-
13   true(control(oplayer)).
14 next(cell1(M,N,x)) :-
15   does(xplayer,mark1(M,N)),
16   true(cell1(M,N,b)).
   ...
17 next(cell1(M,N,X)) :-
18   does(P,mark2(X2,Y2)),
19   true(cell1(M,N,X)).
20 next(control(xplayer)) :-
21   true(control(oplayer)).
22 next(control(oplayer)) :-
23   true(control(xplayer)).
24 open1 :- true(cell1(M,N,b)).
25 goal(xplayer,100) :-
26   line1(x), line2(x).
27 goal(oplayer,75) :-
28   line1(o), not line2(x),
29             not line2(o).
   ...
30 terminal :-
31   terminal1, terminal2.
32 terminal1 :- line1(x).
33 terminal1 :- line1(o).
34 terminal1 :- not open1.
   ...
```

Fig. 1. Some GDL rules of the game Double-Tictactoe

designer to specify the non-effects of actions by *frame axioms*; e.g., lines 17–19 say that marking a cell on the second board does not affect the first board.

The `goal` predicate (lines 25–29) assigns a number between 0 (loss) and 100 (win) to each role in a terminal state. It is defined with the help of the *auxiliary* predicates `line1(W)` and `line2(W)`. Auxiliary predicates are not part of the predefined language, but are defined in the game description itself. The game is over when a state is reached that implies `terminal` (lines 30–34).[2]

3 Subgame Detection

In our previous work [9], we have developed an algorithm to detect independent subgames and applied this algorithm to single-player games. The basic idea is to build a *dependency graph* for a given GDL description of a game, consisting of the actions and fluents as vertices and edges between them if a fluent is a precondition or an effect of an action. The *connected components* of this graph then correspond to independent subgames of that particular game.

While in principle this idea can be applied to multi-player games, some improvements are necessary in order to extend the range of decomposable games. One problem arises from the fact that in [9] the dependency graph is composed of the mere fluent and action symbols of a game. This does not allow to decompose a game based on different *instances* of these fluents and actions.

Example 2. Consider the following rules of the well-known game Nim with four heaps (`a,b,c,d`), where the size of the heaps is represented by the fluent `heap`.

[2] For a complete definition of syntax and semantics of GDL we refer to [11].

```
1  init(heap(a,1)).
2  init(heap(b,2)).
3  init(heap(c,3)).
4  init(heap(d,5)).  ...
5  legal(W,reduce(X,N)) :-
6    true(control(W)),
7    true(heap(X,M)),
8    smaller(N,M).
9  ...
10 next(heap(X,N)) :-
11   does(W,reduce(X,N)).
12 next(heap(X,N)) :-
13   true(heap(X,N)),
14   does(W,reduce(Y,M)),
15   distinct(X,Y).
16 ...
```

Identifying each heap as an independent game is not possible with a dependency graph that does not allow to distinguish different (partial) instances of `heap`.[3]

To overcome this restriction, we base subgame detection for multi-player games on partially instantiated fluent and action terms instead of the mere fluent and action symbols. Considering fully instantiated (i.e., ground) fluents and actions would yield the best results for subgame detection, but this is practically infeasible except for very simple games. Therefore, we instantiate fluents and actions according to the following heuristics:

- The i-th argument of a fluent $\mathtt{f}$ is instantiated with all possible values iff for every rule that matches $\mathtt{next}(\mathtt{f}(\ldots,\mathtt{X_i},\ldots)) :- \mathtt{B}$ the *call graph* (see below) of $\mathtt{B}$ contains $\mathtt{true}(\mathtt{f}(\ldots,\mathtt{X_i},\ldots))$ and does not contain $\mathtt{true}(\mathtt{f}(\ldots,\mathtt{X'_i},\ldots))$ with $\mathtt{X'_i} \neq \mathtt{X_i}$.
- The j-th argument of a move $\mathtt{m}$ is instantiated with all possible values iff the i-th argument of $\mathtt{f}$ is instantiated and there is a rule $\mathtt{next}(\mathtt{f}(\ldots,\mathtt{X_i},\ldots)) :- \mathtt{B}$ where the call graph of $\mathtt{B}$ contains $\mathtt{does}(\mathtt{r},\mathtt{m}(\ldots,\mathtt{Y_j},\ldots))$ with $\mathtt{Y_j} = \mathtt{X_i}$.

A *call graph* [9] of a formula is the least set of atoms containing all atoms in the formula as well as all atoms that occur in a rule whose head matches an atom in the call graph. For computing the call graph, we replace every $\mathtt{does}(\mathtt{R},\mathtt{M})$ in the rules by $\mathtt{legal}(\mathtt{R},\mathtt{M})$, in order to reflect the fact that every executed move must be a legal one.

The idea behind the heuristics is that an argument of a fluent is instantiated if its value does not change from one state to the next and if instances of the fluent that differ in that argument do not interact. If the different instances do not interact they are likely to belong to separate subgames. Arguments of moves are instantiated if they refer to an instantiated argument of a fluent. In example 2, for instance, the first argument of `heap` is instantiated because the rules for `next` always refers directly (line 13) or indirectly (line 11 along with lines 5, 7) to the first argument of `heap` in the current state and there is no other heap referred to. The first argument of the move `reduce` is instantiated because in the first `next`-rule the first argument of `reduce` is identical to that of `heap`.

Another problem we face in multi-player games is to determine which individual action of a joint move (by all players) is responsible for a positive or negative effect. To this end, we extended the definition of potential effects from [9] by the following notion of *a role affecting some fluent*.

[3] See `www.general-game-playing.de` for the complete Nim rules.

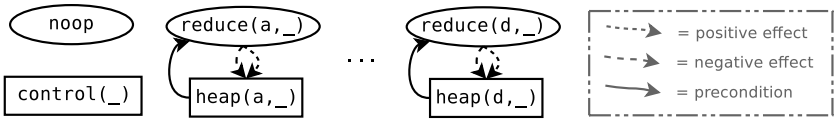

Fig. 2. Dependency graph for the game Nim

Definition 1. *A role r* ***affects*** *a fluent f iff there is a game rule unifiable with* $\texttt{next}(f) :- B$*, where the call graph of B contains* $\texttt{does}(r', m)$ *and r and r' are unifiable.*

A move m is called a ***noop move*** *if it is the only legal move of a player when not in control and if m does not occur in the call graph of any* `next` *rule.*

Fluent f is a ***potential positive effect*** *of move m if there is a game rule unifiable with* $\texttt{next}(f) :- B$ *such that m is not a noop move, B does not imply* $\texttt{true}(f)$*, and B is compatible (see below) with* $\exists r, \vec{y}.\, \texttt{does}(r, m)$ *where r affects f and $\vec{y}$ are the free variables in m.*

Fluent f is a ***potential negative effect*** *of move m if m is not a noop move and there is* no *game rule unifiable with* $\texttt{next}(f) :- B$ *such that for all r that affect f we have* $\forall \vec{y}.\, (\texttt{true}(f) \wedge \texttt{does}(r, m) \Rightarrow B)$ *where $\vec{y}$ are the free variables in m.*

Fluent f is a ***potential precondition*** *of move m if f occurs in the call graph of the body of a game rule with head* $\texttt{legal}(p, m)$*, or head* $\texttt{next}(f')$ *where f' is a potential positive or negative effect of m.*

Compatibility means logical consistency under the constraint that each player can do only one action at a time. Thus a fluent is a potential positive effect if there is a non-frame axiom compatible with the action in question, and it is a potential negative effect if there is no frame axiom for this fluent that applies whenever the action is executed. The potential preconditions of a move include all fluents occurring in a `legal` rule for that move and also the fluents that are preconditions of its (conditional) effects.

The `control`-fluent that is used to encode turn-taking in multi-player games typically occurs in the `legal` rules of all actions. We identify and subsequently ignore the `control`-fluent as precondition during the subgame detection in turn-taking games. Otherwise, it would connect all actions in the dependency graph, effectively rendering subgame detection for turn-taking games impossible.

Applying the above definitions to the Nim game in example 2, we obtain the dependency graph in figure 2 with six subgames: one for each heap consisting of the respective `heap`-fluent and `reduce`-action, one consisting of the `control`-fluent, and one for the `noop`-action.

4 Solving Decomposable Games

Once a multi-player game has been successfully decomposed, it needs to be solved by what we call *decomposition search* (DS). DS is composed of *subgame search* (SGS) and *global game search* (GGS). SGS searches each subgame independently

and returns a set of paths of the subgame tree, which is then used by GGS to compute optimal strategies. As the DS for alternating move games is very similar to the one for simultaneous move games, we will only describe the algorithm for alternating move games here.

Subgame Search (SGS). In each state of an alternating move game, we know whose turn it is next. However, the turn in each state of the subgames is unknown, because in each turn, a player can only choose one subgame to play. Thus SGS needs to consider all players' legal moves during each subgame state expansion and to return a set of paths of the subgame tree, called *turn-move sequences* (TMSeqs).

Definition 2. *A* ***turn-move sequence*** *is a tuple* (Ts, Ms, Es) *where*

- Ts *is a list of roles (****turn sequence****), indicated by* $T_1 \circ T_2 \circ ... \circ T_n$,
- Ms *is a list of moves (****move sequence****), indicated by* $M_1 \circ M_2 \circ ... \circ M_n$,
- Es *is a set of evaluations of* ***local concepts*** *(see below),*

where $n \geq 0$ *is the length of the sequence. If* $n = 0$*, we call it* ***empty turn-move sequence****.*

We extend our notion of *local concepts* from single-player games [9] by recording a sign (positive or negative) for each concept: A *local concept* is a ground literal that occurs in the call graph of a `goal` or `terminal` rule, and the local concept's call graph is only related to the fluents of one subgame. For the `goal` rule in lines 27–29 of figure 1, for example, we get three local concepts: `line1(o)`, `not line2(x)`, and `not line2(o)`. The sign for each concept is determined by whether an even or odd number of negations occurs in the path from the root of the `goal` or `terminal` rule's call graph to the concept. In this way we know that a rule is satisfied if all its local concepts (with signs) are true.

With the help of the extended notion of local concepts, we can not only check the equality of two TMSeqs but also find if one is better than another one by using the following definition.

Definition 3. *A turn move sequence* $s_1 = (Ts_1, Ms_1, Es_1)$ *is* ***evaluation dominated*** *by* $s_2 = (Ts_2, Ms_2, Es_2)$ *(written* $s_1 \preceq_{Cs} s_2$*)* ***under*** *local concepts* Cs *iff* $Ts_1 = Ts_2$ *and* $\forall_{C \in Cs}(Es_1 \models C \Rightarrow Es_2 \models C)$*, where* $Es \models C$ *means* C *is satisfied after playing the moves in the TMSeq. If* $\exists_{C \in Cs}(Es_1 \not\models C \wedge Es_2 \models C)$*, we call* s_1 ***strongly evaluation dominated*** *(*$s_1 \prec_{Cs} s_2$*) by* s_2 ***under*** *local concepts* Cs*. This extends to sets of turn move sequences in the following way:* $T_1 \preceq_{Cs} T_2 \equiv \forall_{s_1 \in T_1} \exists_{s_2 \in T_2} s_1 \preceq_{Cs} s_2$ *and* $T_1 \prec_{Cs} T_2 \equiv (T_1 \preceq_{Cs} T_2 \wedge \exists_{s_1 \in T_1, s_2 \in T_2} s_1 \prec_{Cs} s_2)$.

The TMSeqs are constructed backwards from the leaf nodes to the initial state of a subgame tree. An empty TMSeq, where Es are evaluations of all local concepts, has to be added for every terminal state of the subgame. Because in general the `terminal` rules cannot be evaluated in a subgame state, an empty TMSeq is added for a state that has no legal move for any player or in which

at least one local concept of the `terminal` rules is satisfied. In both cases the subgame state could belong to a terminal state of the game. In each subgame state s a set of TMSeqs is computed for every player in the following way: $TMSeqs(p, s) = \{(p \circ Ts, m \circ Ms, Es) | (Ts, Ms, Es) \in TMSeqs(p', do(p, m, s))\}$. This means that the TMSeqs of a player p in a state s are exactly the TMSeqs of all successor states of s (denoted by $do(p, m, s)$) with the turn sequence Ts augmented by p and the move sequence Ms augmented by the move m that leads to the successor state.

In each state, a set of turn move sequences T_1 obtained from a move of player p is removed if there is a set T_2 from other moves of p such that $\exists_v(T_1 \prec_{Cs} T_2 \wedge \forall_{v'>v} T_1 \preceq_{Cs'} T_2)$ or $\forall_v T_1 \preceq_{Cs} T_2$ where: v and v' are goal values defined for player p in the game rules, Cs are local concepts of `goal(p, v)` and `terminal`, and Cs' are local concepts of `goal(p, v')` and `terminal`. For example, the two paths with dashed lines in figure 3 have the same evaluation under local concepts of `goal(x, 100)` (`line1(x)`) and `terminal` (`terminal1`) as the other two paths, but under local concepts of `goal(x, 75)` (`line1(x)`, `not line1(x)`, `not line1(o)`, and `not open1`) and `terminal`, the dashed path with turn sequence (xo) is strongly dominated by the corresponding solid path because it does not entail `not line1(o)`. Thus the two paths with dashed lines can be removed.

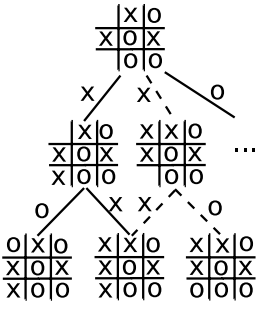

Fig. 3. Subtree of one subgame of Double-Tictactoe

Global Game Search (GGS). GGS is based on standard search techniques (e.g., Minimax, MaxN) but uses TMSeqs returned from SGS instead of the game's `legal` rules to determine the moves in each state. Because of the removal of dominated TMSeqs in SGS the number of moves from the TMSeqs is typically smaller than the number of legal moves. This results in a much smaller game tree compared to full search. Algorithm 4.1 shows the basic idea of DS. We applied iterative-deepening depth-first search (IDDFS) in DS, which finds the shortest solution first and prevents SGS from spending too much time on big subgames.

Algorithm 4.1. Decomposition Search

```
Input: State: global game state, Player: for which the best move is searched
Output: BestMove
TMSeqs ← ∅;
foreach Subgame∈Subgames do
    SubState ← subgame state of Subgame in State;
    SubgameTMSeqs ← SGS(SubState);
    TMSeqs ← TMSeqs ∪ SubgameTMSeqs;
end
BestMove ← GGS(TMSeqs,Player);
```

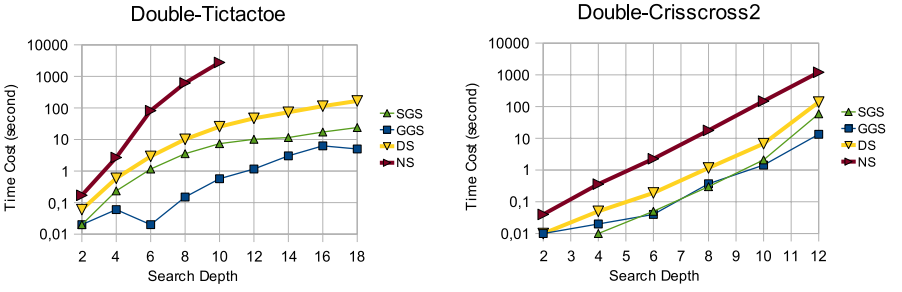

Fig. 4. DS and normal search (NS) testing results

Complexity Comparison and Experiments. The complexity of SGS is the complexity of depth-first search (DFS) plus the complexity of TMSeq simplification. The complexity of GGS depends on the number of TMSeqs returned from SGS. The more TMSeqs can be removed in SGS the less time GGS will use. Assuming a game has n subgames and the average number of states for each subgame is SV, the time complexity of normal search with DFS or IDDFS is $O(|SV|^n + |E_N|)$ while the time complexity of DS is $O(n * |SV| + |E_D| + C)$ where E_N and E_D are edges of the game tree in normal and decomposition search, respectively; $E_N \geq E_D$; and C is the time complexity of GGS. Moreover, the strategies found by DS are just as good as the ones found by normal search.

We have implemented and integrated DS for alternating move games in *Fluxplayer* [3]. Figure 4 shows the time costs of DS and normal search with different search depths (the depth is related to the global game tree, which is the sum of all SGS depths) for two alternating move games. TMSeq simplification works very well for those games; e.g., only 0.0016% (912 of 58242432) and 0.0661% (10448 of 15864465) of the TMSeqs are returned by subgame search for Double-Tictactoe to depth 9 and for Double-Crisscross2 to depth 6, respectively.

5 Impartial Games

Definition 4. *A game G is **impartial** if G is an alternating move game, in each state of G the player whose turn it is has the same legal moves, and the effects of each move are independent on who is making the move.*

Impartial games allow for a special DS that is more efficient than the general method. Before discussing DS for impartial games, let us describe how a general game player can check whether a game is impartial.

Checking Impartiality. According to the definition of impartial games, we would need to check every state of the game to know if the game is impartial. Since this is not feasible in general, we only do a syntactic analysis of the game rules. This yields a sound but incomplete method. The main idea is to verify that the `legal` and `next` rules for all players are equivalent by checking that for

each `legal` and `next` rule that is defined for one player there is a correspondent rule for every other player.

Definition 5. *Given two rules R_1 and R_2 of a multi-player game, R_1 and R_2 are* ***correspondent*** *for players P_1 and P_2 iff simultaneously substituting P_1 for P_2 and P_2 for P_1 in* `control` *fluents and* `does` *predicates in R_1 yields a variant ($R_1[P_1/P_2, P_2/P_1]$) of R_2, that is, $R_1[P_1/P_2, P_2/P_1]$ and R_2 are equal up to renaming of variables and reordering of literals in the bodies of the rules.*

As an example, the `legal` and `next` rules of Nim (example 2) are correspondent to themselves for all players.

Decomposition Search (DS). It is easy to prove that a game G is impartial iff its independent subgames are impartial. Another important theorem used in this section is the Sprague-Grundy theorem, which says that each impartial game with normal play convention is equivalent to some Nim heap. Nim is a typical impartial game, which has been mathematically solved. Thus each subgame is actually a Nim heap. If we have the size (called *nimber*) of each subgame, we can use *Nim-sum* to obtain the size of the global game and solve the impartial game by using the strategies used for Nim. More information about Nim and nimber can be found in [12]; explaining the full theory behind impartial games goes beyond the scope of our paper.

Subgame Search (SGS). SGS uses depth-first search to search each subgame with only one player to compute the nimber of the subgame. As all players have the same legal moves in each state, it is sufficient to consider one player's legal moves instead of all players. The nimber of terminal states is 0. For intermediate state, the nimber is the minimal excludent of the nimbers of its successors. For example, if the successors of a state have nimbers 0, 1 and 3, the nimber for the state will be 2.

Global Game Search (GGS). The nimber of the global game can be easily computed as Nim-sum of the nimbers of all subgames, and then the winning strategies for Nim are used to do the rest.

Complexity Comparison and Experiments. For a subgame of size n, the worst-case time complexity of SGS is $O(2^n)$. For an impartial game with m heaps (subgames) of sizes n_1, n_2, ..., n_m, the time complexity of DS is $O(\sum 2^{n_i})$, whereas the worst-case time complexity of standard search is $O(2^{\sum n_i})$. In practical play, however, the time complexities are much lower if transposition tables are used in the search; e.g., for Nim this reduces the complexity of SGS to $O(n)$.

The following table shows the time cost of DS and normal search for game Nim with 4 heaps (4 subgames) in *Fluxplayer*:

Time Cost(s)	Normal Play			Misère
	Heaps Size			
	1,5,4,2	2,2,10,10	11,12,15,25	12,12,20,20
Normal Search	0.4	3.5	6607	10797
Decomposition Search	0.01	0.01	0.07	0.06

From the results it is easy to see that the time cost for DS is linear in terms of the biggest heap size, while the one for normal search is exponentially growing in terms of the sum of all heap sizes.

6 Conclusion

We have developed a method by which general game playing systems can search for a decomposition of a multi-player game into independent sub-games in order to significantly improve game tree search. Our result generalises an established method from AI Planning [6,7,8] to General Game Playing. A different, preliminary approach to the decomposition of multi-player games has been independently developed by [13], but there the authors did not address the issue of how a general game player can actually exploit such a reduction during play.

References

1. Kuhlmann, G., Dresner, K., Stone, P.: Automatic heuristic construction in a complete general game player. In: AAAI, pp. 1457–1462 (2006)
2. Clune, J.: Heuristic evaluation functions for general game playing. In: AAAI, pp. 1134–1139 (2007)
3. Schiffel, S., Thielscher, M.: Fluxplayer: A successful general game player. In: AAAI, pp. 1191–1196 (2007)
4. Finnsson, H., Björnsson, Y.: Simulation-based approach to general game playing. In: AAAI, pp. 259–264 (2008)
5. Genesereth, M., Love, N., Pell, B.: General game playing: Overview of the AAAI competition. AI Magazine 26(2), 62–72 (2005)
6. Amir, E., Engelhardt, B.: Factored planning. In: IJCAI, pp. 929–935 (2003)
7. Brafman, R., Domshlak, C.: Factored planning: How, when and when not. In: AAAI, pp. 809–814 (2006)
8. Kelareva, E., Buffet, O., Huang, J., Thiébaux, S.: Factored planning using decomposition trees. In: IJCAI, pp. 1942–1947 (2007)
9. Günther, M., Schiffel, S., Thielscher, M.: Factoring general games. In: Proceedings of the IJCAI Workshop on General Intelligence in Game-Playing Agents (GIGA), pp. 27–34 (2009)
10. Love, N., Hinrichs, T., Haley, D., Schkufza, E., Genesereth, M.: General game playing: Game description language specification. Technical report (2008)
11. Schiffel, S., Thielscher, M.: A multiagent semantics for the Game Description Language. In: ICAART. Springer, Heidelberg (2009)
12. Conway, J.H.: On Numbers and Games. London Mathematical Society Monographs, vol. 6. Academic Press, London (1976)
13. Cox, E., Schkufza, E., Madsen, R., Genesereth, M.: Factoring general games using propositional automata. In: Proceedings of the IJCAI Workshop on General Intelligence in Game-Playing Agents (GIGA), pp. 13–20 (2009)