Abstract Strategic Reasoning in Tic-Tac-Toe

Filippos Nakas

1 Abstract

This paper makes use of a method presented in (Zhang et al., 2014) to very efficiently prove facts about explicitly described abstract strategies in Tic-Tac-Tao using the theorem proving capacities of Answer Set Programming. To our knowledge, this is the first attempt of proving the properties of explicitly described high-level strategies in tic-tac-toe or a game of similar complexity using this method. Our main interest in the problem is motivated by our undergoing thesis work in the connection between analogy and abstraction. In particular, we see this work as a first proof-of-concept for the idea that two game elements can be considered analogous in a pragmatic sense if they play the exact same role in a valuable abstract strategy.

2 Introduction

In this paper, we present a full answer set programming (ASP) implementation of the method presented in (Zhang et al., 2014) for performing explicit strategic reasoning for finite, complete-information, multiplayer games, applied to abstract strategies in Tic-Tac Toe. Our implementation, which we provide in Appendix B, allows us to prove in fractions of a second that certain abstractly described strategies for each player are guaranteed to never lose (or always win) when played against a given set of strategies by the other player. Most importantly, the complete strategies our program reasons about are specified by composing a series of incomplete and non-deterministic strategies (”tactics”) which are defined on predicates that abstract away from many low-level facts of a game state in a way that intuitively matches the strategic reasoning of human players. For example, one of these tactics is ”forking” (also applicable in a different guise in chess).

Our technical contribution is two-fold. First, we demonstrate that the methodology can be very efficiently applied to games of higher strategic complexity than the running example of the paper. Moreover, we provide our full code which can immediately run as is in the browser-version of the clingo ASP prover (Gebser et al., 2019) to replicate the results. Secondly, given our particular interest in abstract, human-like tactics, we expended a large amount of effort in enhancing and reformulating the ”base” description of tic-tac-toe rules in a manner that makes it easy to define abstract tactics (see section 4). In the future we will attempt generalize and possible automate the reformulation for a broader set of games that involve some notion of space (or the notion enemy, valuable material, etc.). This could set the stage for analogical transfer of abstract tactics learned in one game to other games that share high-level properties relevant to those tactics.

The theoretical value of this work with respect to our research in analogy is to give some formal sense to the intuitive notion that certain game elements are "analogous" when it comes to some abstract strategy but not analogous when it comes to another. For instance, rows, and diagonals are analogous with respect to activating the "block an attack" strategy in the sense that low level facts about each of them can independently activate it (similarly for "fork"). Diagonals, however, are not analogous with rows with respect to the "fill a corner or center square" tactic since all low-level facts about diagonals are relevant to the realization of this action, unlike all low-level facts about columns and rows. This definition is still relatively informal but we believe that we can formalize it in future work by leveraging the information is encoded in the relations between the low-level facts that describe the base game elements and the higher-order predicates that are used in defining the tactic. We can then hopefully apply the definition in a more complicated game like chess to show that knights and bishop are analogous in their ability to fork but dis-analogous in their ability to pin (a knight cannot pin) and generalize to apply across different games.

3 Motivation and Background

Analogical reasoning is usually defined as the ability of a reasoning system to leverage structural correspondences between a newly experienced situation (the target domain) and an older more familiar one (the source domain) to make quick inferences about the former based on preexisting knowledge of the latter. Providing an adequate overview of this vast literature (Bartha, 2013) is far beyond the scope of the paper so we will concentrate on the two of the most influential symbolic approaches. SME (structural mapping engine) (Falkenhainer et al., 1986) receives an input logical descriptions of two situations and attempts to generate a mapping between their constituent objects and relations given certain heuristic criteria. One of its problems, however, is that it usually relies on at least certain relations or objects of the two situations to be identically named in the logical descriptions, which is arguably not essential to the underlying logic of the two domains.

Like SME, Second-Order Anti-unification (Schmidt et al., 2014) takes as input two logical descriptions of the domains. Unlike SME, however, it first produces an intermediary abstract description consisting of relations that results from anti-unifying structurally similar relations (arity, connections to other relations) in the two domains which mediates the mapping. Although this approach produces a common abstraction, it still computes it based on similarities in the logical encoding of two situations that may not be essential to the domains themselves. But most importantly, it does not, to our knowledge, relativise the concept of two relations or objects being analogous to the purpose for which the analogy will be made (see Introduction for chess and tic-tac-toe examples). Our thesis work is aimed at finding a logical approach that addresses these two limitations. Here, we concentrate on the latter and choose the field of game-playing because it often offers itself to complete logical descriptions and is very rich in very intuitive analogies. We chose high-level strategies as candidates for a common abstractions because 1) they are inherently goal-specific and 2) their effectiveness in a game is a property of the game itself and not its description.

Our idea of using high-level strategies as abstractions is loosely inspired from the notion of abstract plans used in hierarchical planning and state-abstraction methods in the planning domain (Banihashemi et al., 2017; Bäckström et al., 2022). Prior work (Hinrichs et al., 2013) in analogical transfer between games has used hierarchical task networks in combination with an adapted version of SME to transfer high-level strategies between similar single-player games (which are basically planning problems). Our approach differs in two main ways: 1) we address a single two-player game which requires reasoning about strategies instead of plans (harder) and consider analogies within the same game (easier), and 2) our focus is theoretical: we focus on describing the common abstraction structure that supports the analogies and, at this stage, ignore the question how it would be found. Strategical reasoning in multiplayer games is significantly more complicated than classical planning since it involves taking account of the strategies of other players. For this reason the options of logical languages that performs such reasoning are much sparser. All alternatives we know of other than the method we chose are based on non-trivial extensions of modal, dynamic, and temporal logics (Chatterjee et al., 2007; Benthem, 2013, Zhang et al., 2015) that result in rich multi-modal languages, with many complicated axioms relating their modalities. We initially attempted using LEO-III (Steen et al., 2018) to encode and reason in one of these logics but discovered the task was too complicated and risky for the time we had. We aim to try this in the future.

4 Overview of Proposed Approach: Reasoning about Strategies with ASP

The most influential language for fully describing and running the rules of any finite, multi-player, complete-information games is the Game Description Language (GDL) (Genesereth et al., 2005), which is a variant of Datalog, which is, however, not expressive enough for strategic reasoning. Zhang and Thiesler provide a way of translating GDL to an extended version of the situation calculus that supports strategic reasoning, a substantial fragment of which can in turn be translated to an ASP specification (Lifschitz et al., 2019) that supports efficient theorem-proving. ASP solvers first ground their input programs to a propositional version (which guarantees decidability) and then use a version of SLD resolution adapted to answer-set semantics while delegating computations to SAT-solvers, whenever possible. Such solvers are especially efficient for programs that obey rule stratification of the predicates, which we tried to ensure.

The abstract description of performing the two translations is rather involved and the paper's running example is simpler than Tic-Tac-Toe. In addition, the final translation in ASP for the example is incomplete and only involves relatively easily definable strategies that do not require many auxiliary high-level predicates like our own. For this reason we had to almost significantly reformulate the final ASP specification to define the abstract strategies we were interested in. We focus here on describing at a high level the main parts of the final specification (the full runable specification can be found in Appendix B).

Facts like: $player(0;1), index(1..3), square(cell(X,Y)) : -index(X), index(Y)$, $time(1..9), direction(left; right)$ encode permanent facts about every state of the game. We enhance them with a partial mereology of relating board concepts with each other defined with rules such as:

```
part_of(cell(X,4-X), diag(right)) :- index(X).
```

Fluents are facts represented as functions that have changing truth values as the game progresses through time such as the marking of individual cells. They are used in statements like holds(marked(0, cell(2,2), 1), which states that player 0 (that is, player X) has marked the middle cell at the first time-step. Rules like the above allow us to relate how the effects of actions on ground fluents affect fluents about more abstract board entities like lines and columns. For instance:

```
holds(marked(P,Y,N),T) :- {holds(marked(P,X),T) : square(X),
part_of(X,Y)} == N, N = 0..3, player(P), time(T), board_entity(Y).
```

The first rule uses the *part_of* relation and the base fluents to define the number of total marks of a particular type (X or O) on a higher level board entity (row, column,...). This is used in another rule to define when a player threatens with obtaining three in a row. This relation, in turn, will later be used later in the program to define the "fork" tactic. Finally, there exist rules that enforce the rules and dynamics of the game: initial conditions, action legality and effects, terminal and winning conditions etc. Strategies are specified by characterizing conditions under which an action is considered "strategic" for the given strategy. For instance:

```
fork(a(P,C),T) :- action(a(P,C)), time(T),holds(threatens(P,L1,1),T),
holds(threatens(P,L2,1),T), L1 != L2, part_of(C, L1), part_of(C,L2).
```

These rules defines the conditions under which an action is a fork. Other strategies we define include "defend threat", "occupy non-edge piece", "threaten non-center cell". We call these mini-strategies tactics because they are incomplete, i.e., they cannot be applied in all cases by a player. However, we can use two operations defined in the original paper to compose them into complete and more specialized strategies. Prioritised disjunction and prioritised conjunction take two strategies/tactics A and B and create strategies $A \triangledown B$:"try followiing A; if you can't, follow B" and $A \triangle B$:"try an action that follows both A and B; if you can't just follow B". These operations can be recursively applied to create arbitrarily complicated strategy specifications (see code for how they are implemented). We use our set of abstract tactics and the operators to define a total of 4 complete abstract strategies:

strat_mark_any: mark any cell (this strategy encompasses all possible strategies that a player can play)

strat_edge_mistake_O: mark any cell but if you are the second to play mark any edge on your first move

$strat\_good\_for\_X$: defeat ▽ defend threat ▽ fork ▽ (mark a center or corner cell △ mark a cell that does not threatens the center) ▽ mark any cell

$strat\_good\_for\_O$: defeat ▽ defend threat ▽ fork ▽ (mark a center or corner cell △ mark the center) ▽ mark any cell

We can then use rules that enforce that we only compute answer sets (possible games) in which the moves of each player always abide to their assigned strategy. We can determine general properties about what happens in such games by imposing restrictions such as: $: -notwinsO.$, which means that player O should win (not not win). If player X plays $strat\_good\_for\_X$, player O plays *strat_mark_any* and the solver finds no answer set, i.e., no possible game, that satisfies these restrictions (as is the case), it means that player 1 always wins or forces a tie no matter what player O does (since "mark any" subsumes all possible strategies of player O). We can similarly prove whether in a particular strategic confrontation a player always wins or always forces a tie.

5 Evaluation Methodology and Results

We evaluate our implementation in terms of its ability to efficiently prove useful results about the effectiveness of abstract strategies. Proving the effectiveness of an abstract strategy which treats game elements as equivalent in forcing a positive game outcome is necessary to establish that the elements are analogous in a pragmatically strong way. We list the established results along with their computation time using clingo.

X: $strat\_good\_for\_X$, O: *strat_mark_any*, X never loses, 0.383s

X: $strat\_good\_for\_X$, O: *strat_make_edge_mistake_O*, X always wins, 0.201s

X: *strat_mark_any*, O: $strat\_good\_for\_O$, O never loses, 0.270s

X: $strat\_good\_for\_X$, O: $strat\_good\_for\_O$, always tie, 0.203s

6 Conclusion

We have shown that our implementation of the method allows us to define strategies for tic-tac-toe that abstract from low-level elements and facts about board positions that give meaning to the fact that elements and facts that play the same role in the strategy are analogous to each other with respect to that strategy. Furthermore we have proven that these strategies can force positive outcomes for players following them which means that the game elements that have the same role in them are analogous in a pragmatically useful way.

7 References

Zhang, Dongmo and Michael Thielscher. "Representing and Reasoning about Game Strategies." Journal of Philosophical Logic 44 (2014): 203-236.

Hinrichs, T., Forbus, K. D. (2011). Transfer Learning through Analogy in Games. AI Magazine, 32(1), 70. https://doi.org/10.1609/aimag.v32i1.2332

Genesereth, M., Love, N., Pell, B. (2005). General Game Playing: Overview of the AAAI Competition. AI Magazine, 26(2), 62. https://doi.org/10.1609/aimag.v26i2.1813

Falkenhainer, B., Forbus, K.D., Gentner, D. (1986). The Structure-Mapping Engine. AAAI.

Bäckström, Christer, and Peter Jonsson. "A framework for analysing state-abstraction methods." Artificial Intelligence 302 (2022): 103608.

Schmidt, Martin et al. "Heuristic-Driven Theory Projection: An Overview." Computational Approaches to Analogical Reasoning (2014).

Bartha, Paul. "Analogy and analogical reasoning." Stanford Encyclopedia of Philosophy (2013).

Banihashemi, Bita, Giuseppe De Giacomo, and Yves Lespérance. "Abstraction in situation calculus action theories." Thirty-First AAAI Conference on Artificial Intelligence. 2017.

Chatterjee, K., Henzinger, T.A., Piterman, N. (2007). Strategy Logic. In: Caires, L., Vasconcelos, V.T. (eds) CONCUR 2007 – Concurrency Theory. CONCUR 2007. Lecture Notes in Computer Science, vol 4703. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-74407-85

Zhang, D., Thielscher, M. (2015). A Logic for Reasoning About Game Strategies. Proceedings of the AAAI Conference on Artificial Intelligence, 29(1). https://doi.org/10.1609/aaai.v29i1.9416

GEBSER, M., KAMINSKI, R., KAUFMANN, B., SCHAUB, T. (2019). Multi-shot ASP solving with clingo. Theory and Practice of Logic Programming, 19(1), 27-82. doi:10.1017/S1471068418000054

Lifschitz, Vladimir. Answer set programming. Heidelberg: Springer, 2019.

Benthem, Johan van. "Reasoning about strategies." Computation, Logic, Games, and Quantum Foundations. The Many Facets of Samson Abramsky. Springer, Berlin, Heidelberg, 2013. 336-347.

Steen, Alexander, and Christoph Benzmüller. "The higher-order prover Leo-III." International Joint Conference on Automated Reasoning. Springer, Cham, 2018.

8 Appendix A: GDL specification of Tic-Tac-Toe, taken from: http://gamemaster.stanford.edu/library/tict

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% tictactoe
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%%% metadata
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

 role(x)
 role(o)

 base(cell(M,N,x)) :- index(M) & index(N)
 base(cell(M,N,o)) :- index(M) & index(N)
 base(cell(M,N,b)) :- index(M) & index(N)
 base(control(R))  :- role(R)

 action(mark(M,N)) :- index(M) & index(N)

 index(1)
 index(2)
 index(3)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% init
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

 init(cell(1,1,x))
 init(cell(1,2,x))
 init(cell(1,3,o))
 init(cell(2,1,b))
 init(cell(2,2,o))
 init(cell(2,3,b))
 init(cell(3,1,b))
 init(cell(3,2,o))
 init(cell(3,3,x))
 init(control(x))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% legal
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

 legal(mark(X,Y)) :- cell(X,Y,b)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% operations
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

 mark(M,N) :: control(R)  ==> cell(M,N,R) & ~cell(M,N,b)
 mark(M,N) :: control(x) ==> ~control(x) & control(o)
 mark(M,N) :: control(o) ==> ~control(o) & control(x)
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% goal
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

 goal(x,100) :- line(x) & ~line(o)
 goal(x,50) :- line(x) & line(o)
 goal(x,50) :- ~line(x) & ~line(o)
 goal(x,0) :- ~line(x) & line(o)
 goal(o,100) :- ~line(x) & line(o)
 goal(o,50) :- line(x) & line(o)
 goal(o,50) :- ~line(x) & ~line(o)
 goal(o,0) :- line(x) & ~line(o)

 row(M,X) :- cell(M,1,X) & cell(M,2,X) & cell(M,3,X)
 col(N,X) :- cell(1,N,X) & cell(2,N,X) & cell(3,N,X)
 diag(X) :- cell(1,1,X) & cell(2,2,X) & cell(3,3,X)
 diag(X) :- cell(1,3,X) & cell(2,2,X) & cell(3,1,X)

 line(X) :- row(M,X)
 line(X) :- col(N,X)
 line(X) :- diag(X)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% terminal
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

 terminal :- line(x)
 terminal :- line(o)
 terminal :- ~open

 open :- cell(M,N,b)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

9 Appendix B: Our ASP specification. Copy paste on the browser version of clingo to run: https://potassco.org/clingo/run/

```
player(0;1).
index(1..3).
square(cell(X,Y)) :- index(X), index(Y).
time(1..9).
direction(left;right).

part_of(cell(X,Y), row(X)) :- index(X), index(Y).
part_of(cell(X,Y), col(Y)) :- index(X), index(Y).
part_of(cell(X,X), diag(left)) :- index(X).
part_of(cell(X,4-X), diag(right)) :- index(X).

part_of(diag(D), diag_space) :- direction(D).
part_of(cell(X,Y), edge_space) :- square(cell(X,Y)), not part_of(cell(X,Y), d
part_of(cell(X,Y), board) :- square(cell(X,Y)).

part_of(X,Y) :- part_of(X,Z), part_of(Z,Y).

board_entity(Y) :- part_of(X,Y), square(X).

line(row(X)) :- index(X).
line(col(Y)) :- index(Y).
line(diag(D)) :- direction(D).

holds(marked(P,Y,N),T) :- {holds(marked(P,X),T) : square(X), part_of(X,Y)} ==
holds(threatens(P,L,N),T) :- player(P), line(L), holds(marked(P,L,N),T), hold
holds(gets_three(P,L),T) :- player(P), line(L), holds(marked(P,L,3),T).

wins(P,T) :- holds(gets_three(P,L),T).
open_board(T) :- square(C), time(T), not holds(marked(0,C),T), not holds(mark
terminal(T) :- wins(P,T).
terminal(T) :- time(T), not open_board(T).
tie(T) :- not wins(0,T), not wins(1,T), terminal(T).

action(a(P,C)) :- player(P), square(C).

holds (turn (0) ,1).
holds (turn (P),T+1) :- holds (turn (1-P),T), time(T).

legal (a(P,C),T) :- holds(turn(P),T), action(a(P,C)), not holds(marked(P,C),T
holds (marked(P,C), T+1) :- holds(marked(P,C),T), player(P), square(C), time(
holds (marked(P,C), T+1) :- action(a(P,C)), does(a(P,C),T), time(T).
```

```
1 { does (A,T) : action (A) } 1 :- time(T), not terminal(T).
:- does(A,T), not legal (A,T).

:- non_strategic(T).
non_strategic(T) :- does(A,T), not strat (A,T).

defend_threat(a(P,C),T) :- action(a(P,C)), time(T), holds(threatens(1-P,L,2),
defeat(a(P,C),T) :- action(a(P,C)), time(T),holds(threatens(P,L,2),T), part_o
fork(a(P,C),T) :- action(a(P,C)), time(T),holds(threatens(P,L1,1),T), holds(t
mark_diag_space(a(P,C),T) :- action(a(P,C)), time(T),part_of(C, diag_space).
mark_edge_space(a(P,C),T) :- action(a(P,C)), time(T),part_of(C, edge_space).
threaten_non_center(a(P,C),T) :- action(a(P,C)), time(T),holds(threatens(P,L,
mark_any(a(P,C),T) :- action(a(P,C)), time(T), square(C).
capture_center(a(P,C),T) :- action(a(P,C)), time(T), C == cell(2,2).


exists_defeat(P,T) :- defeat(a(P,C),T).
exists_defend_threat(P,T) :- defend_threat(a(P,C),T).
exists_fork(P,T) :- fork(a(P,C),T).
exists_threaten_non_center(P,T) :- threaten_non_center(a(P,C),T).
exists_mark_diag_space(P,T) :- mark_diag_space(a(P,C),T).
exists_capture_center(P,T) :- capture_center(a(P,C),T).

strat_good_for_X(a(P,C),T) :- defeat(a(P,C),T).
strat_good_for_X(a(P,C),T) :- defend_threat(a(P,C),T), not exists_defeat(P,T)
strat_good_for_X(a(P,C),T) :- fork(a(P,C),T), not exists_defeat(P,T), not exi
strat_good_for_X(a(P,C),T) :- mark_diag_space(a(P,C),T), threaten_non_center(
strat_good_for_X(a(P,C),T) :- mark_diag_space(a(P,C),T), not exists_threaten_
strat_good_for_X(a(P,C),T) :- mark_any(a(P,C),T), not mark_diag_space(a(P,C),

strat_good_for_O(a(P,C),T) :- defeat(a(P,C),T).
strat_good_for_O(a(P,C),T) :- defend_threat(a(P,C),T), not exists_defeat(P,T)
strat_good_for_O(a(P,C),T) :- fork(a(P,C),T), not exists_defeat(P,T), not exi
strat_good_for_O(a(P,C),T) :- mark_diag_space(a(P,C),T), capture_center(a(P,C
strat_good_for_O(a(P,C),T) :- mark_diag_space(a(P,C),T), not exists_capture_c
strat_good_for_O(a(P,C),T) :- mark_any(a(P,C),T), not exists_capture_center(P

strat_any(a(P,C),T) :- mark_any(a(P,C),T).

strat_edge_mistake_O(a(1,C),2) :- mark_edge_space(a(1,C),2).
strat_edge_mistake_O(a(1,C),T) :- time(T), T!= 2, square(C).

%decide the strategy for each player using one of the strategies above
%playerX has index 0 and playerO has index 1
strat(a(1,C),T) :- strat_good_for_O(a(1,C),T).
strat(a(0,C),T) :- strat_good_for_X(a(0,C),T).
```

```
winsO :- wins(1,T), terminal(T).
winsX :- wins(0,T), terminal(T).
tie :- tie(T).

%decide on a constrain given what you want to prove. For example,
%if there are no answer sets for
%:- not tie, we infer that the game will always be a tie.
%:- winsX, we infer that player X always wins.
%:- not winsX, we infer that player O never loses.
%:- winsO, we infer that player O always wins.
%:- not winsO, we infer that player X never loses.

:- not tie.

#show does/2.
#show terminal/1.
#show wins/2.
#show tie/1.
```