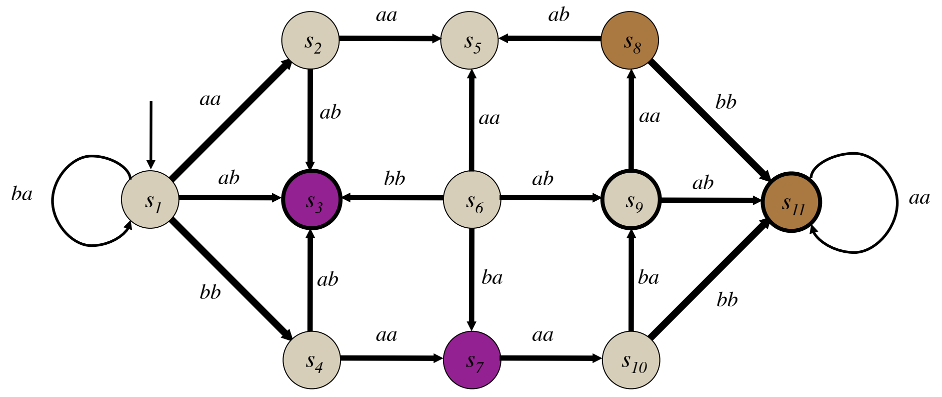
General Game Playing

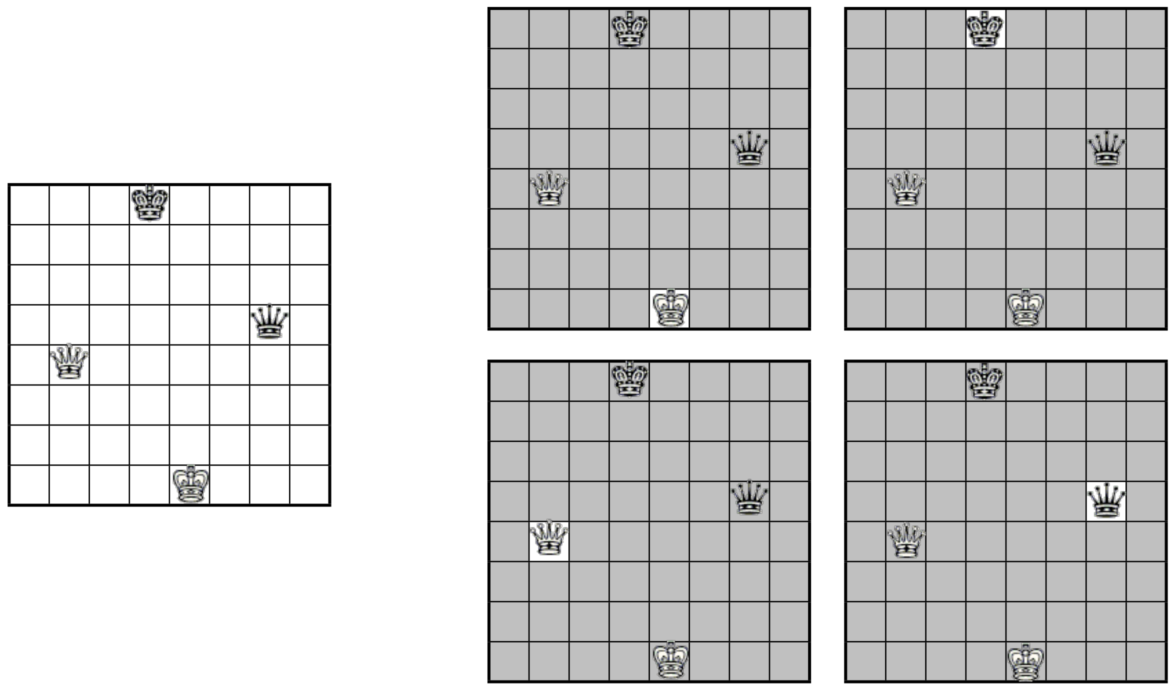
Propositional Nets

Games as State Machines



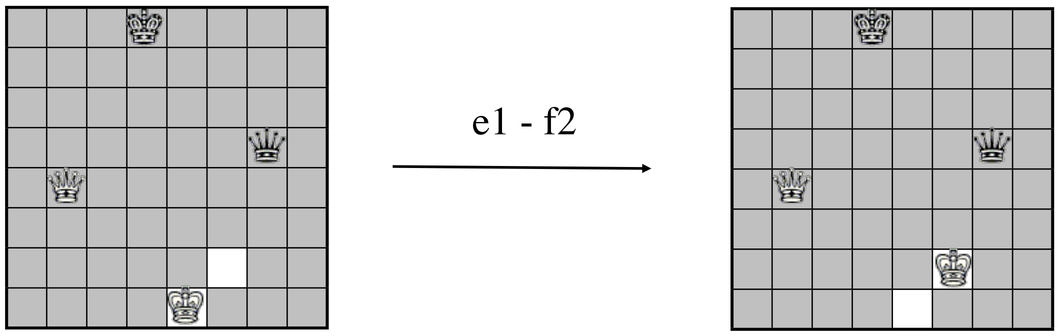
In the introduction, we saw that it is possible to think of the dynamics of a game as a state graph like the one shown here. A game is characterized by a finite number of states, a finite number of players, each with a finite number of actions. At each point in time the game is in one of the possible states; players choose from their possible actions; and, as the players perform their chosen actions, the game changes from one state to another.

States and Propositions

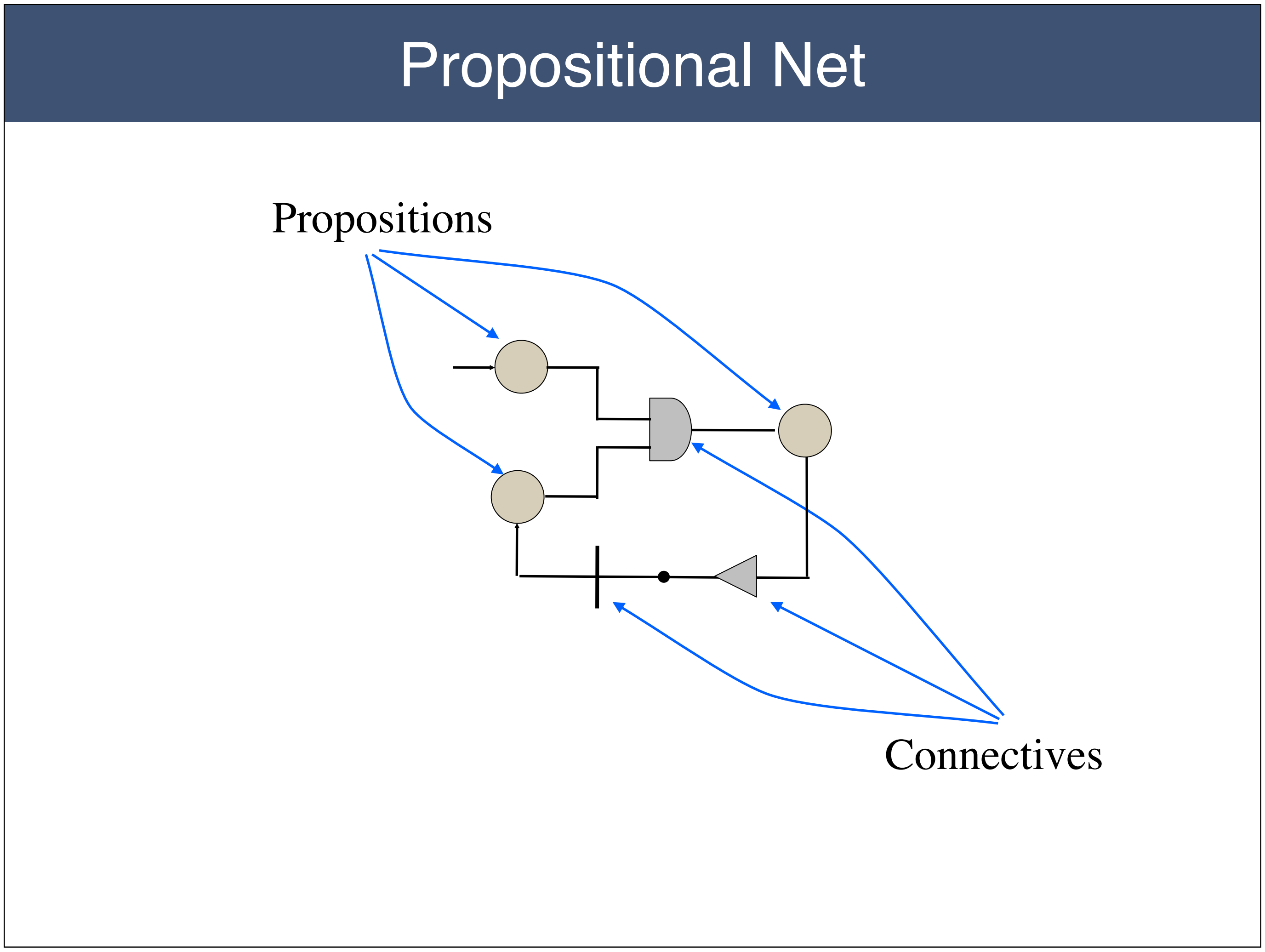


In the vast majority of games, states and actions are not monolithic - they can be defined in terms of more fundamental entities. In Chess, for example, states can be conceptualized in terms of the locations of individual pieces on the Chess board - e.g. the location of the white king, the white queen, teh black king, and the black queen. In Risk, where many pieces are moved at once, overall actions can be defined in terms of the movements of individual pieces.

Limited Influence

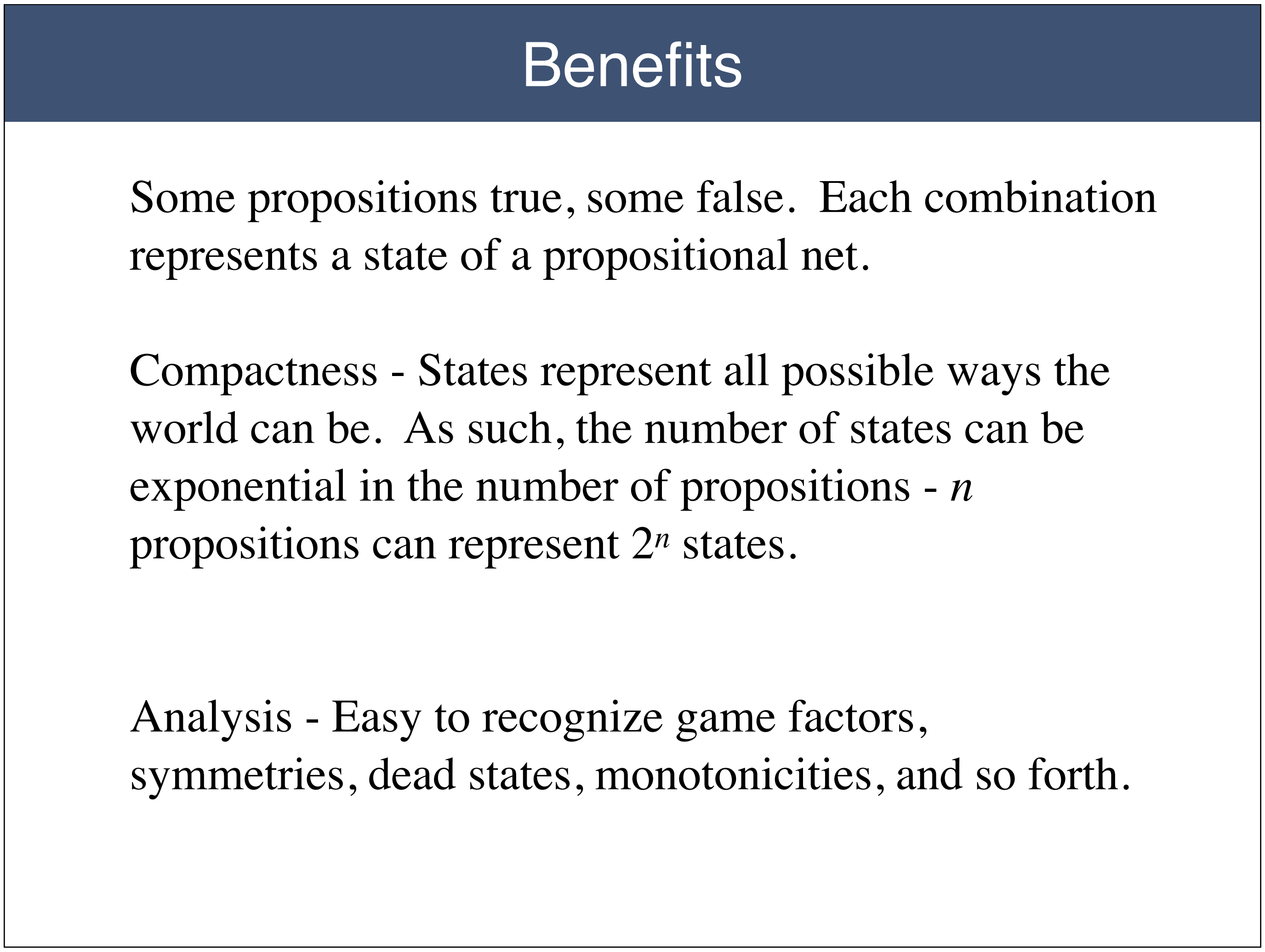


Also, in the vast majority of games, the effects of actions are local. As actions are performed, some propositions become true and others become false. However, the truth values of most propositions are independent.



Propositional Net

The ideas of state decomposition and limited influence suggest a conceptualization of games in terms of the individual propositions rather than states together with a representation of the effects of actions on these propositions rather than entire states. The result is an alternative representation of dynamics called a propositional net. Unlike a state machine, in which the nodes represent states, in a propositions net the nodes are individual propositions and actions together with connectives that represent their behavior (as we shall see).



Benefits

Some propositions true, some false. Each combination represents a state of a propositional net.

Compactness - States represent all possible ways the world can be. As such, the number of states can be exponential in the number of propositions - n propositions can represent 2^n states.

Analysis - Easy to recognize game factors, symmetries, dead states, monotonicities, and so forth.

One of the benefits of formalizing games as propositional nets is compactness. A set of *n* propositions corresponds to a set of 2^*n* states (all different combinations of truth values for the *n* propositions). Thus, it is often possible to characterize the dynamics of games with graphs that are much smaller than the corresponding state machines. For example, a propnet with just three propositions corresponds to a state machine with eight states.

A second benefit is ease of analysis. It is sometimes possible to use a propnet to discover independence / game factors, dead states, and other features that can dramatically reduce the cost of game tree search.

Programme

Formal definitions

Games as propositional nets

Game playing with propositional nets

Game Analysis with propositional nets

In the next segment, we formalize propositional nets. In the section thereafter, we show how to describe games in this way. In the segment after that, we see how to play games using game descriptions encoded as propositional nets rather than GDL. We then see how to use propositional nets in restructuring games and discovering game-specific heuristics.

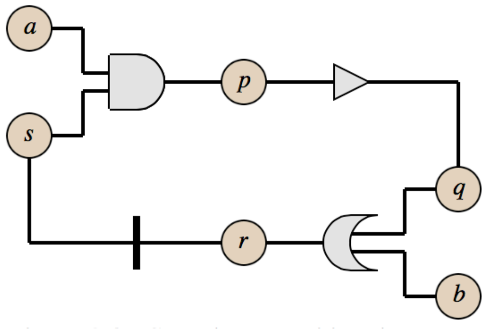
GENERAL
GAME
PLAYING

GENERAL
GAME
PLAYING

Definitions

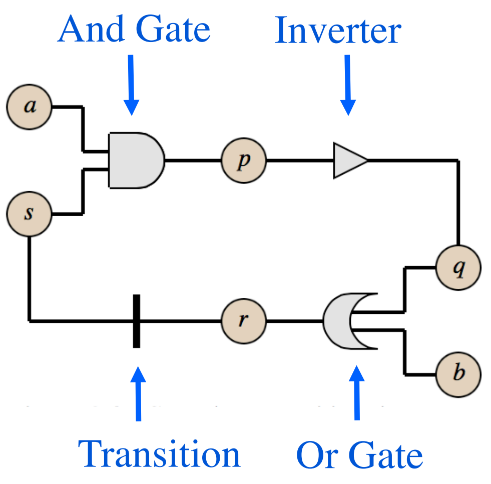
Propositional Net

A *propositional net* (or *propnet*) is a directed, bipartite graph consisting of *propositions* alternating with *connectives*.

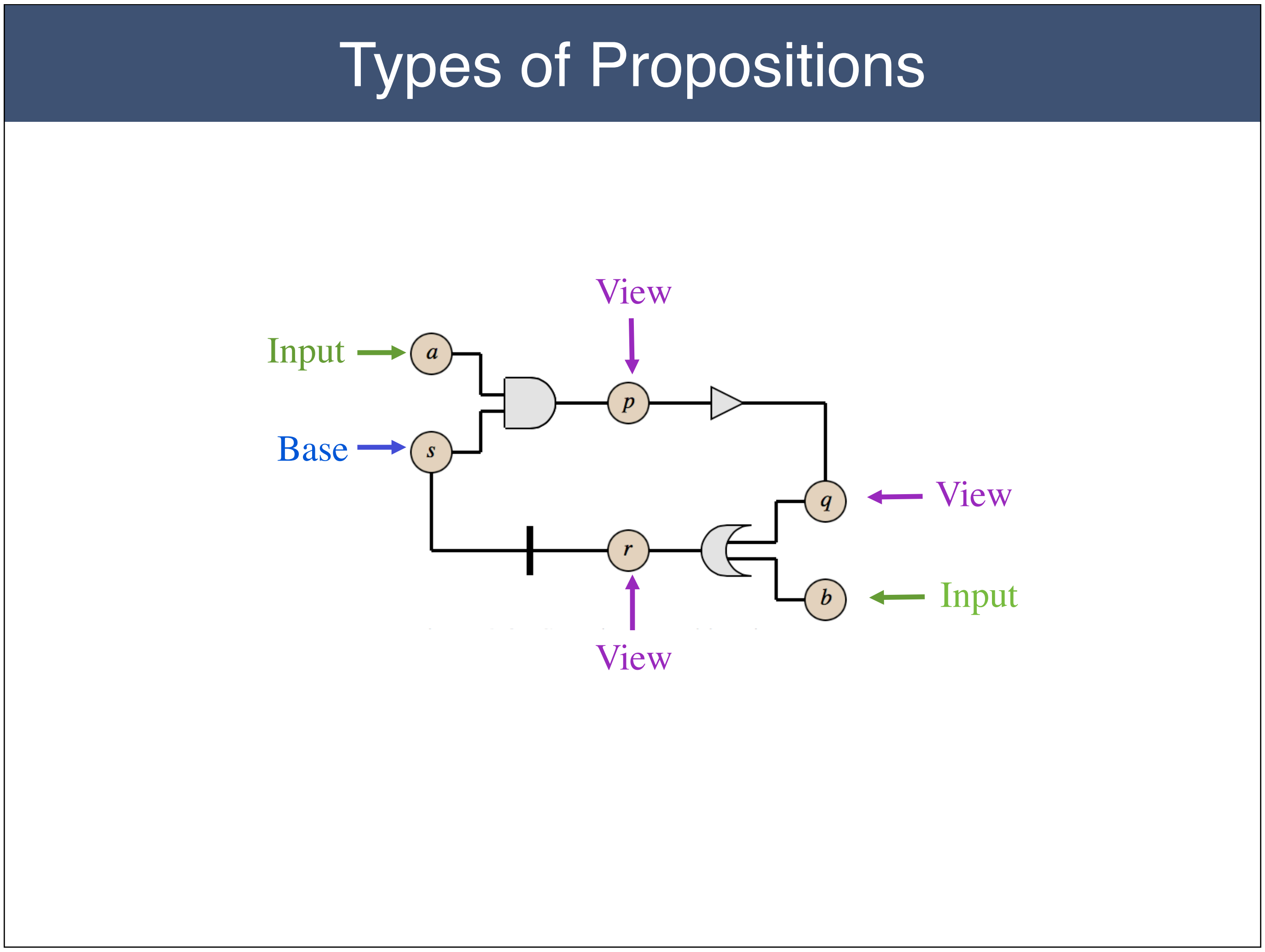


Formally, we define a *propositional net* (or *propnet*) as a directed, bipartite graph consisting of *propositions* alternating with *connectives*. In this case, there are six propositions (the nodes labelled *a*, *b*, *p*, *q*, *r*, and *s*); and there are four connectives (the grey and black nodes in the graph).

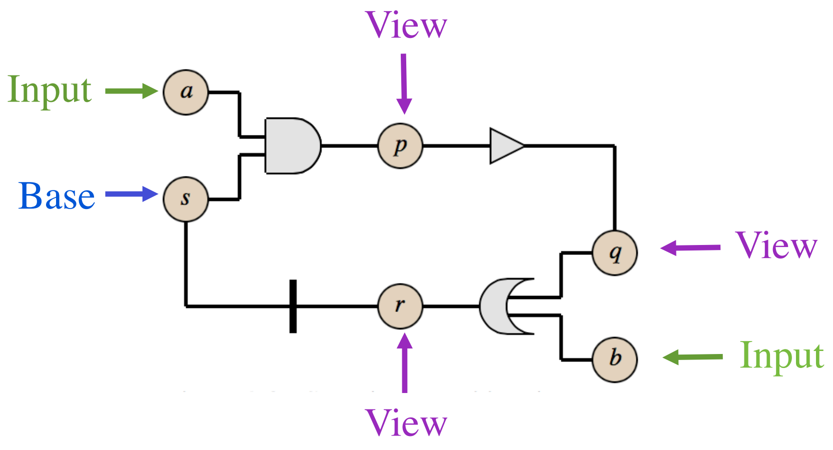
Propositional Net



In basic propnets, there are four types of connectives, and they are all present here. There is an and-gate on the upper left, an inverter on the upper right, an or-gate on the lower right, and a transition on the lower left.



Types of Propositions



Propositions are typically partitioned into three classes: *input propositions* (those with no inputs); *base propositions* (those with incoming arcs from transitions); and *view propositions* (those with incoming arcs from connectives other than transitions). In our example, nodes *a* and *b* are input propositions; node *s* is a base proposition; and nodes *p*, *q*, and *r* are view propositions.

Markings

An *input marking* maps input propositions to booleans.

$$m\colon \mathrm{I} \rightarrow \{1,0\}$$

A *base marking* maps base propositions to booleans.

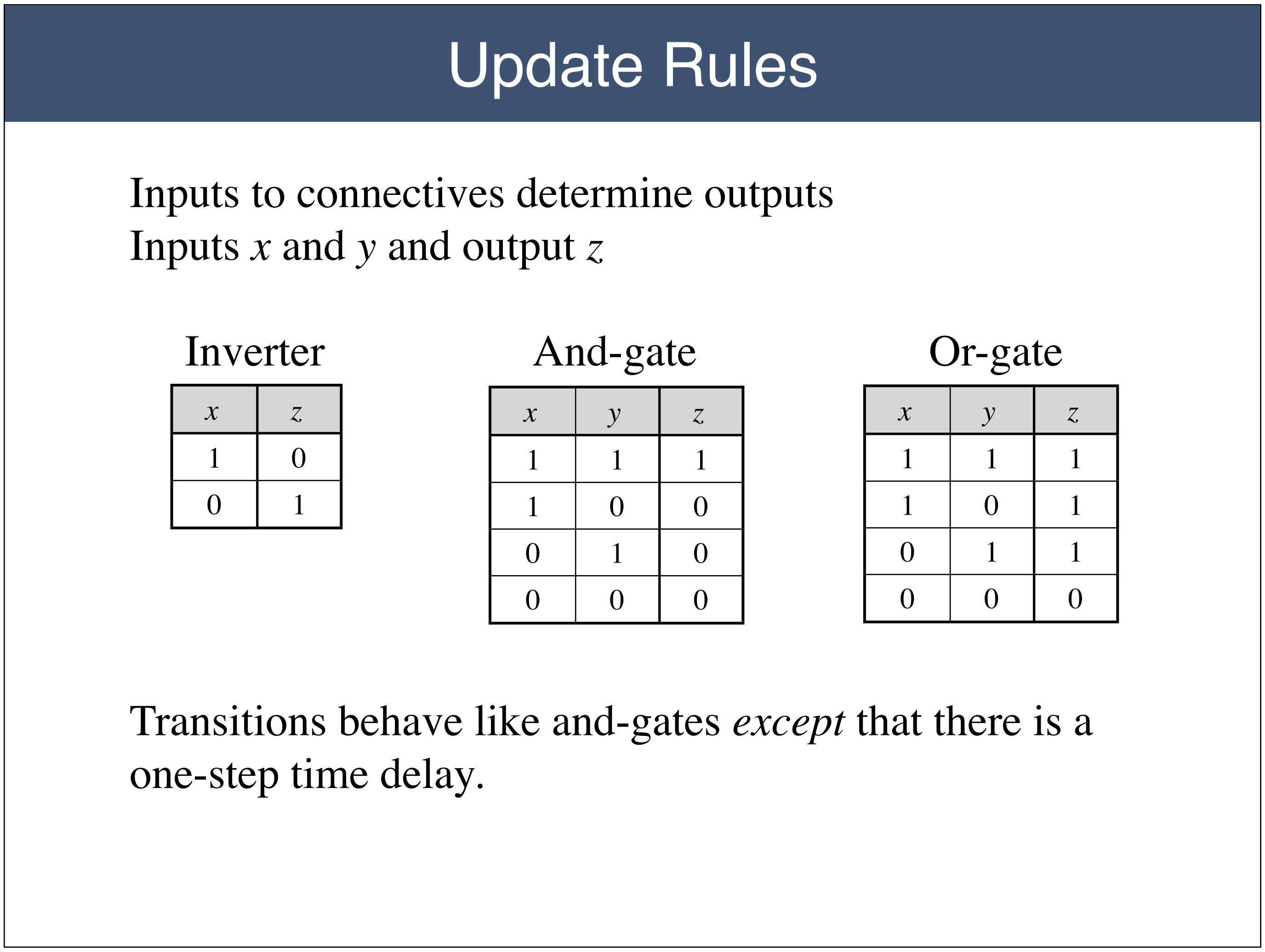
$$m\colon \mathrm{B} \rightarrow \{1,0\}$$

A *view marking* maps view propositions to booleans.

$$m\colon \mathrm{V} \rightarrow \{1,0\}$$

A *marking* is a combination of input, base, view marking.

An *input marking* is a function from the input propositions of a propositional net to boolean values. A *base marking* is a function from the base propositions of a propositional net to boolean values. A *view marking* is a function from the view propositions of a propositional net to boolean values.



Update Rules

Inputs to connectives determine outputs
Inputs *x* and *y* and output *z*

Inverter

x	*z*
1	0
0	1

And-gate

x	*y*	*z*
1	1	1
1	0	0
0	1	0
0	0	0

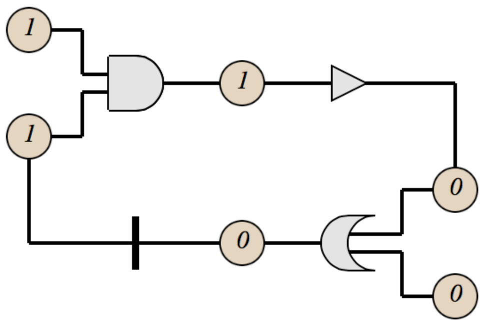
Or-gate

x	*y*	*z*
1	1	1
1	0	1
0	1	1
0	0	0

Transitions behave like and-gates *except* that there is a one-step time delay.

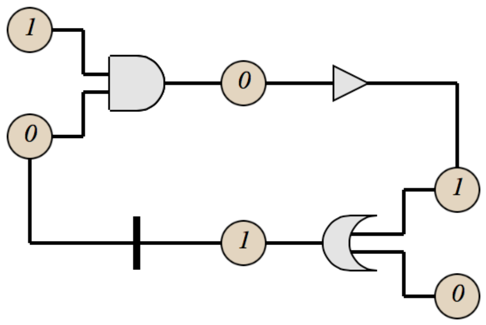
Given a propnet, an input marking and a base marking determine a unique view marking for that propnet. This is based on the types of gates leading into the view propositions. The output of an inverter is true if and only if its input is false. The output of an and-gate is true if and only all of its inputs are true. The output of an or-gate is true if an only if at least one of its inputs is true. Transitions behave like and-gates except that there is a one step time delay.

Example



Here is an example. Suppose we had an input marking that assigned *a* the value 1 and *b* the value 0, and suppose we had a base marking that assigned *s* the value 1. Then, the output of teh and-gate would be 1; the output of the inverter would be 0; and the output of the or-gate would be 0. At this point, we have values for all of the view propositions in the propnet.

Example

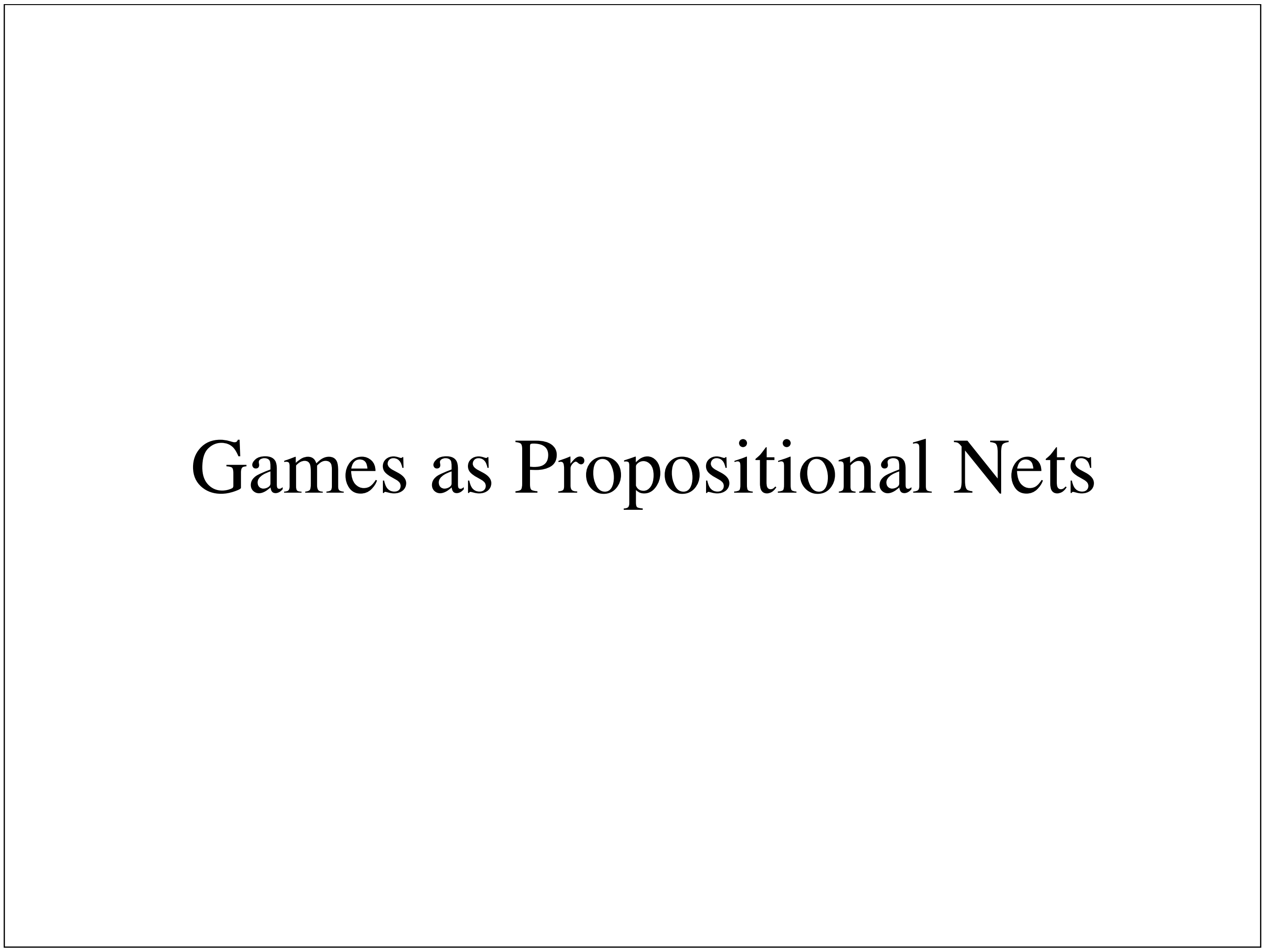


Now, let's move on to the next step. Suppose the input marking for the second step is the same as the first, i.e. *a* is 1 and *b* is 0. What is the value of our base proposition on this step? Since it is the output of a transition, its value on this step is the same as the value of that transition's input on the preceding step. In this case, the transition's input was 0 on the preceding step, and so the value is 0 on this new step. As before, we can compute the view marking corresponding to the new input marking and this new base marking. In this case, since the second input to the and-gate is 0, the ouptut is 0, the output of the inverter is 1; and the output of the or-gate is 1 as well.

If we leave the input values the same, the propnet will go on alternating in this way. If input a ever becomes false, it will stop alternating. However, the alternation will begin again as soon as it is set to true again.

GENERAL
GAME
PLAYING

GENERAL
GAME
PLAYING



Games as Propositional Nets

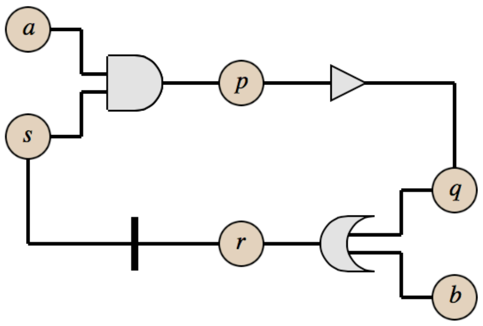
Propnets are an alternative to GDL for expressing the dynamics of games. With a few additional provisions, it is possible to convert any GDL game description into a propositional net with the same dynamics.

Example

```
role(white)
base(s)
input(white,a)
input(white,b)

legal(white,a)
legal(white,b)

p :- does(white,a) & true(s)
q :- ~p
r :- true(q)
r :- does(white,b)

next(s) :- r

goal(white,100) :- true(s)
goal(white,0) :- ~true(s)

terminal :- true(q)
```

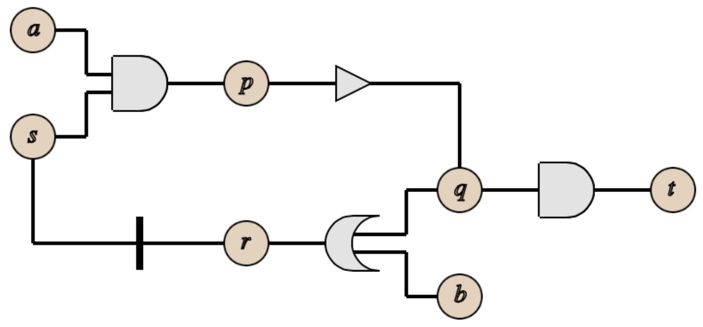
As an example of this, consider the simple game described here. There is just one role. There is just one base proposition, and there are two actions. The two actions are always legal. The player gets 100 points if s is true; otherwise, the player gets 0 points. The game ends if q ever becomes true.

Physics



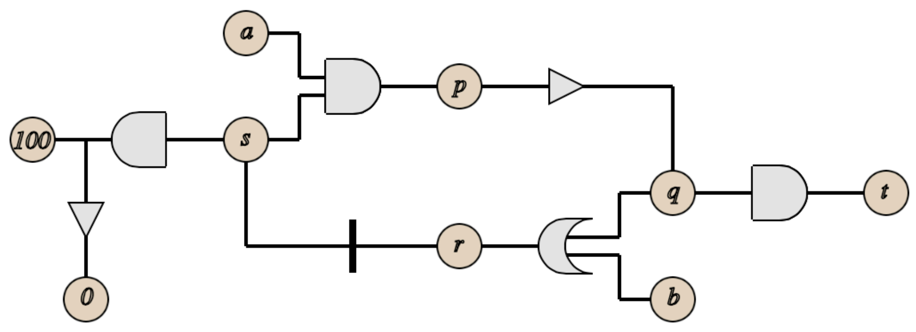
Now, let's build a propnet for this game. We start with the game's physics. The base propositions in the propnet consist of the propositions defined by the base relation in the game description (viz. s). The input propositions correspond to the actions defined by the input relation in the game description (viz. a and b). We use the next relation to capture the dynamics of the game. Starting with the base and input propositions, we add links for each rule (using inverters for negations, and-gates for multiple conditions, and or-gates for multiple rules). In so doing, we augment the propnet with additional view propositions as necessary. The result is the propnet introduced in the last segment.

Termination



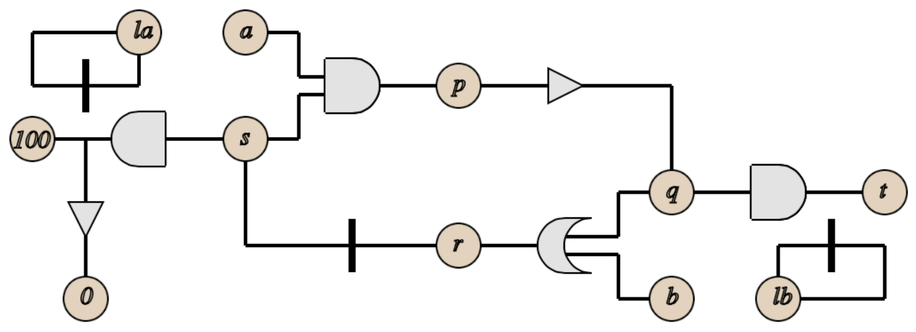
We model terminal in the propnet by adding a special node for termination. In this case, terminal corresponds exactly to q, so we could just use q as our terminal node. However, for the sake of clarity, we can add a new node t and insert a connection from q to t. Note that we use a one-input and-gate here. We could equally well use a two-input and-gate with both inputs supplied by q, but this is simpler. The behavior is the same.

Rewards



Rewards are handled analogously. We create a new node for each reward value, and we use the definitions for these values to extend the propnet further. In this case, goal(white,100) corresponds exactly to s, so we do not need to add a new node for goal(white,100). However, for clarity, in our example, we have added one node for each of the two goal values.

Legality



Legality is the trickiest part. There are various ways of doing this. The simplest method conceptually is to add one legality propositions for each possible action and extend the propnet to say when these nodes are true. In this case, we would add two new propositions la and lb, corresponding to actions a and b. In this case they are always true. To model this, we add a self-loop for each of these legality propositions, and we initialize the proposition to true. Because of the dynamics of transitions, the propositions will remain true indefinitely.

That's it. Once this is done, we have a propnet that reflects the game described in the given GDL. In the next chapter, we discuss how to use this propnet to play games.

GENERAL
GAME
PLAYING

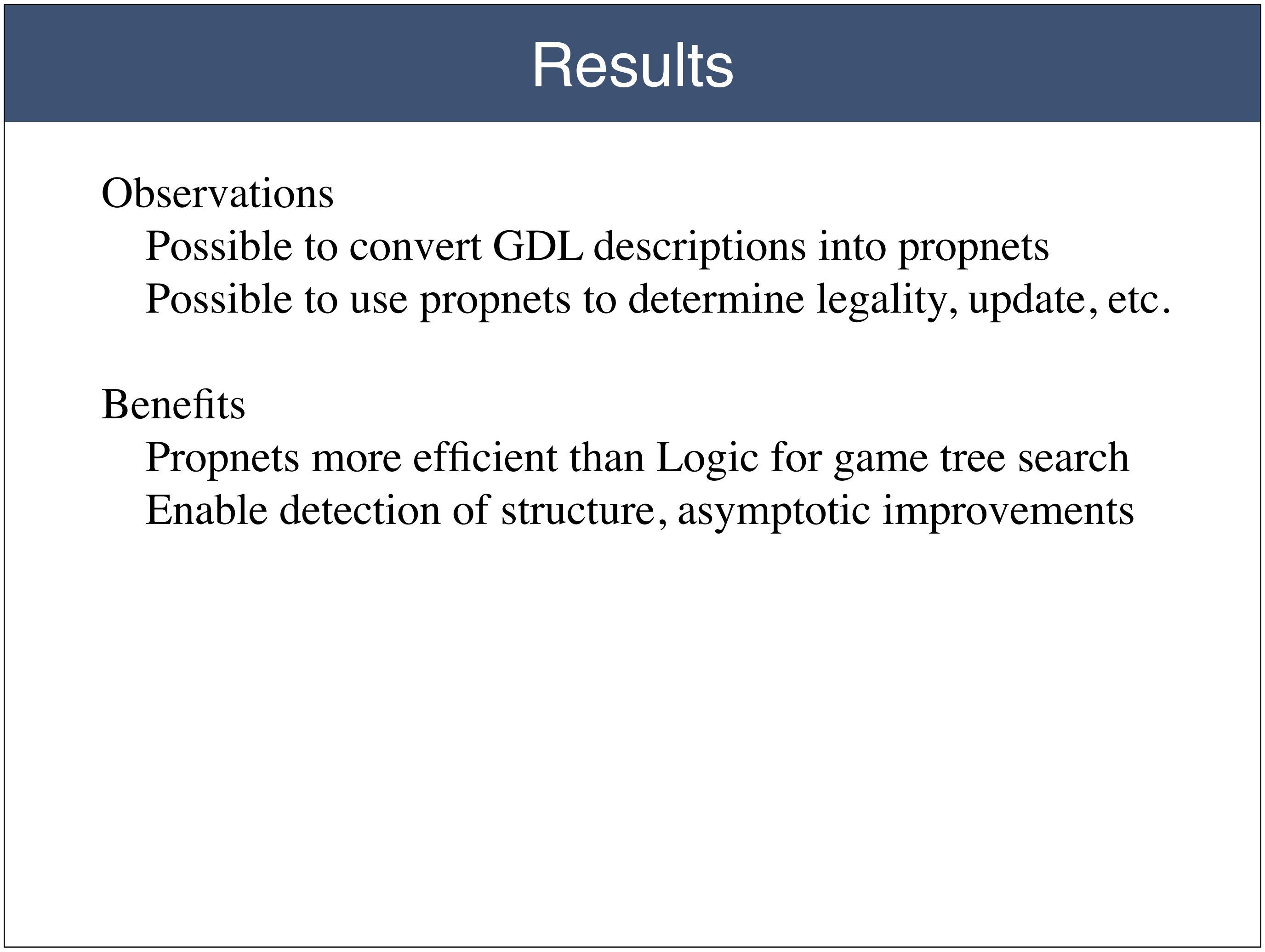
GENERAL
GAME
PLAYING

Game Playing with Propnets

Results

Observations
- Possible to convert GDL descriptions into propnets
- Possible to use propnets to determine legality, update, etc.

In our work thus far, we have been concentrating on game players that process GDL directly during game play. This works reasonably well, as we have seen; but we can do better. As it turns out, it is possible to convert an arbitrary GDL game description into an equivalent propositional net; and it is possible to use propositional nets to determine legality, update, termination, and so forth.



Results

Observations

- Possible to convert GDL descriptions into propnets
- Possible to use propnets to determine legality, update, etc.

Benefits

- Propnets more efficient than Logic for game tree search
- Enable detection of structure, asymptotic improvements

Doing things this way is frequently (though not always) more efficient than interpreting the GDL. Moreover, as we shall see, it facilitates the discovery of game structure that can dramatically alter the complexity of playing many games.

Results

Observations
- Possible to convert GDL descriptions into propnets
- Possible to use propnets to determine legality, update, etc.

Benefits
- Propnets more efficient than Logic for game tree search
- Enable detection of structure, asymptotic improvements

Programme
- See notes for details on use of propnets
- This segment talks about issues and illustrates benefits
- See next segment for use in restructuring games

The details of using propnets to play games are tedious, and we will not try to go through them here. If you want to learn more about this, see the notes. In this segment, we will simply summarize some of the main issues and the benefits of using propnets during game play. And in the next we will explore how propnets can be used offline to restructure games in dramatic ways.

Experiment

Winner detection - variant on maxscore
- takes game description as argument
- explores entire game tree
- returns true if there is a forced win for the first player

Two Alternatives
- genwinnerp - uses GDL game description as before
- propwinnerp - uses propnet version of the GDL

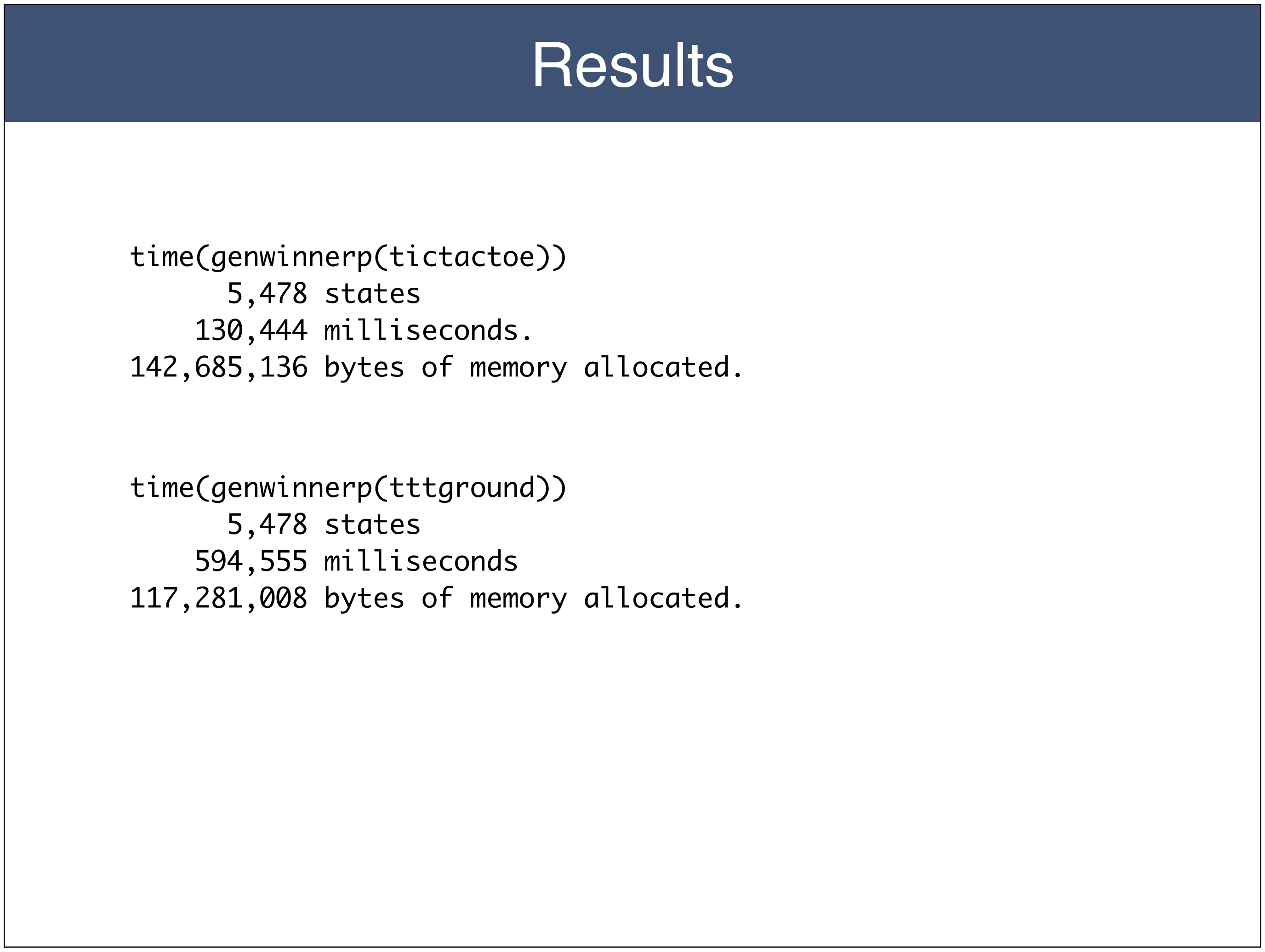
Two Descriptions
- tictactoe - our usual GDL encoding of Tic Tac Toe
- tttground - GDL version with all variables instantiated

Consider a simple variation on the maxscore subroutine we saw earlier. The subroutine takes a game description as argument, explores the entire game tree, and returns true if and only if the first player has a forced win in the specified game. Let's consider two implementations. The first, called genwinnerp, uses a GDL description of a game; and the second, called propwinnerp, uses a propnet. In order to compare the two, let's consider two versions of Tic Tac Toe. The first is just our usual encoding in GDL. The second, called tttground, is also a GDL encoding but with all variables replaced by ground terms. I am including this case to see whether the performance using propnets is due to the elimination of variables or other factors. Now, let's look at the results.

Results

```
time(genwinnerp(tictactoe))
      5,478 states
    130,444 milliseconds.
142,685,136 bytes of memory allocated.
```

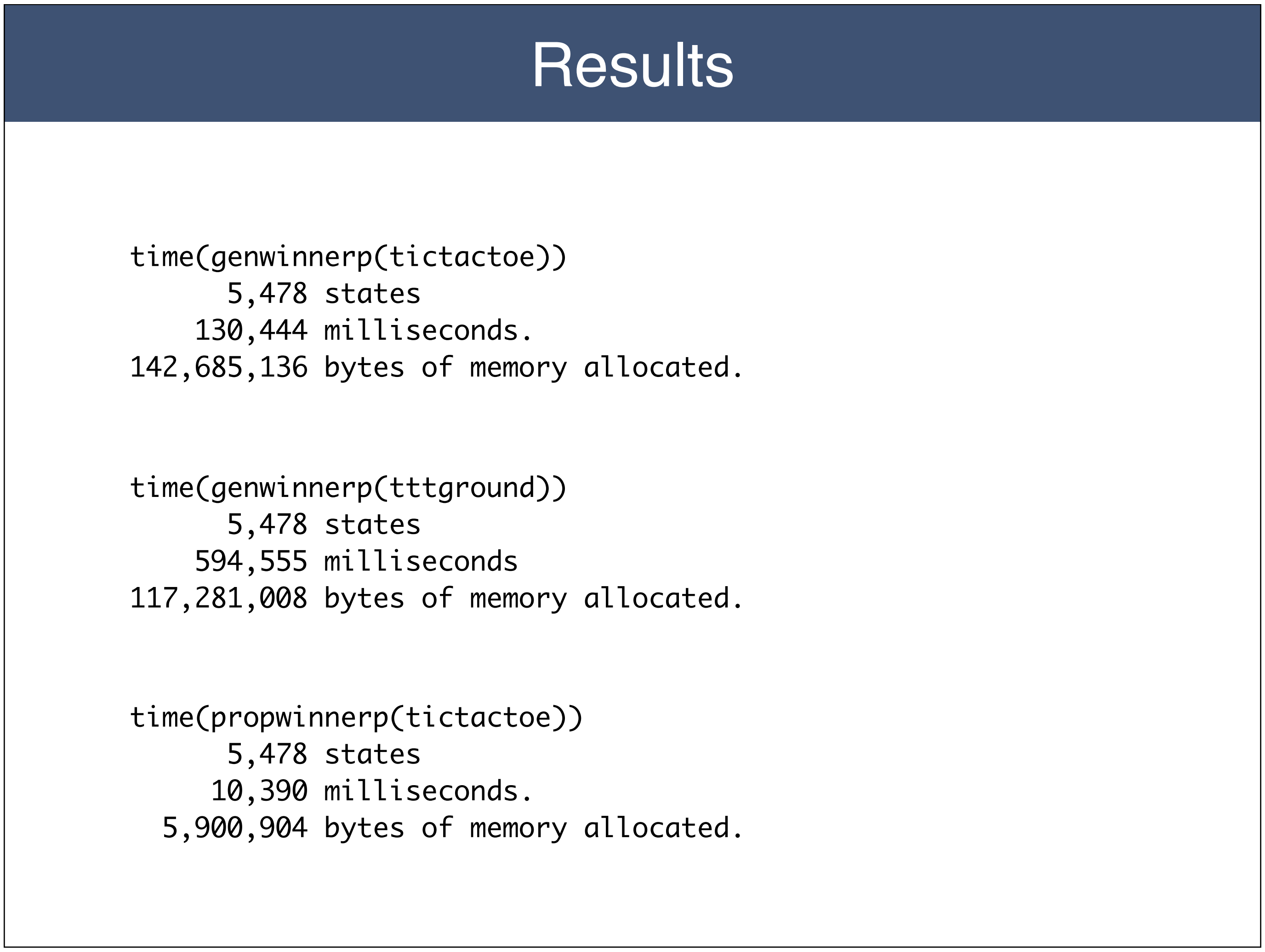
In our experiment, we first applied genwinnerp to tictactoe to get a baseline. Sure enough, it explored all 5478 states in the game tree. It took approximately 130 seconds and used 142 Megabytes of memory. That is a little slow, but this was run some time ago on a relatively slow computer.



Results

```
time(genwinnerp(tictactoe))
      5,478 states
    130,444 milliseconds.
142,685,136 bytes of memory allocated.


time(genwinnerp(tttground))
      5,478 states
    594,555 milliseconds
117,281,008 bytes of memory allocated.
```

Next, we applied genwinnerp to tttground to see whether the elimination of variables would help. In fact, things got much worse. Though the program used less memory, the runtime increased to almost 600 seconds. Actually, this was not all that surprising. By eliminating variables, grounding things out, we increase the number of rules that must be checked and this increased the runtime.



Results

```
time(genwinnerp(tictactoe))
      5,478 states
    130,444 milliseconds.
142,685,136 bytes of memory allocated.


time(genwinnerp(tttground))
      5,478 states
    594,555 milliseconds
117,281,008 bytes of memory allocated.


time(propwinnerp(tictactoe))
      5,478 states
     10,390 milliseconds.
  5,900,904 bytes of memory allocated.
```

Finally, we used the propositional net program propwinnerp on the propositional net description tttpropnet. It explored the same 5478 states. But this time, the runtime decreased to just over 10 seconds, and the memory usage dropped to just under 6 Megabytes. A significant saving.

Compilation

Specialized Data Structures for state
- e.g. state as a vector of values
- e.g. state as a bit vector

Specialized programs
- game operations defined in terms of bit operations
- programs can be written automatically from GDL

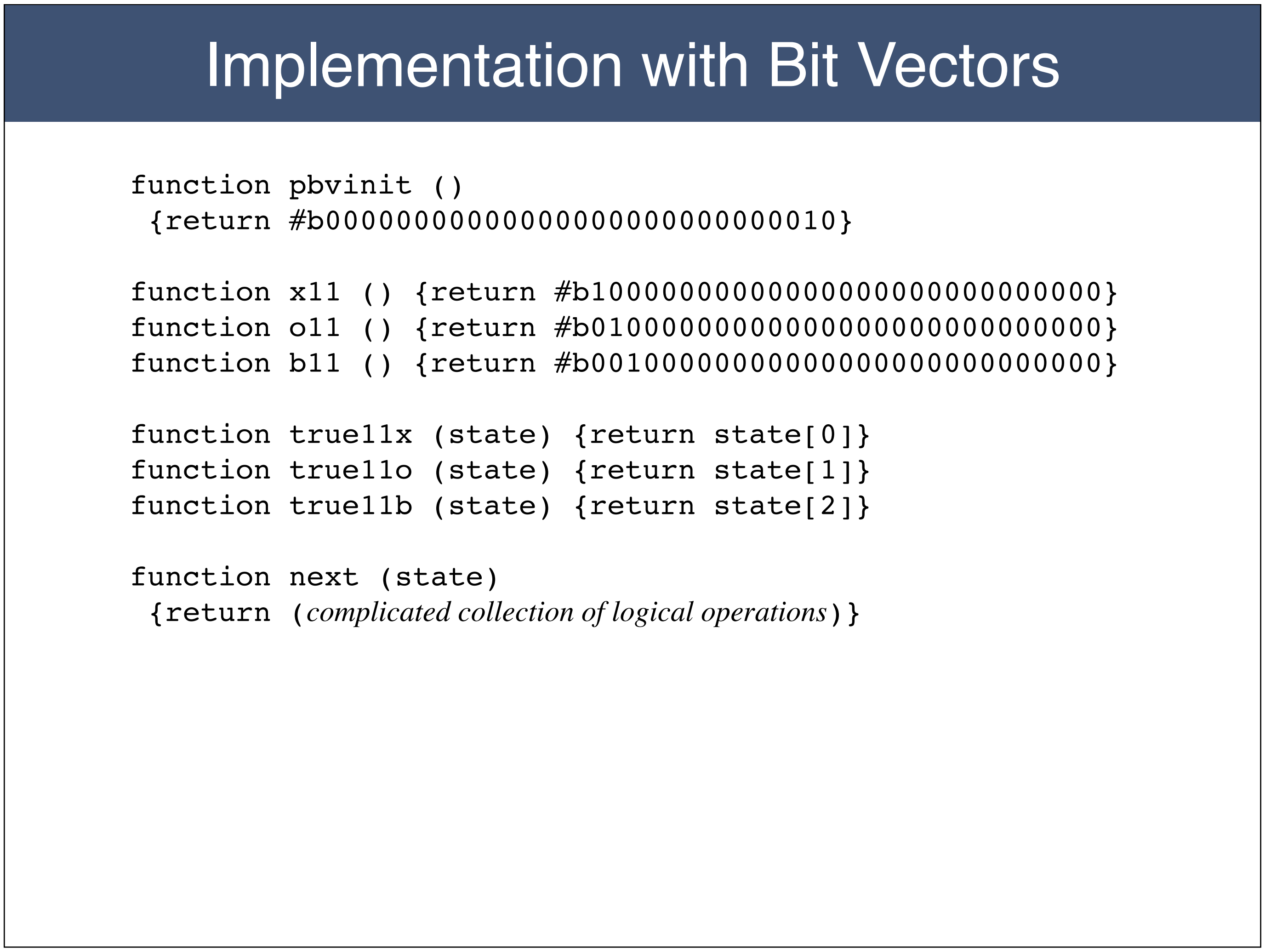
Compile programs into machine code

Can we do better? The answer is yes. Propwinnerp still processes propnets interpretively. But it does not have to be done that way. We can represent the state of the propnet as a list of values or as a bit vector, and we can convert the propnet and its interpreter to special purpose code to process these representations of state in performing the usual game analysis operations. This translation can be done entirely automatically. Moreover, we can then compile the resulting programs.

Implementation with Value Lists

```
function tttinit ()
 {return [0,0,1,0,0,1,0,0,1,
          0,0,1,0,0,1,0,0,1,
          0,0,1,0,0,1,0,0,1,1,0)}

function true11x (state) {return state[0]}
function true11o (state) {return state[1]}
function true11b (state) {return state[2]}

function next11x (input,state)
 {return (does('white','mark11',input) || true11x(state))}

function next11o (input,state)
 {return (does('black','mark11',input) || true11o(state))}

function next11b (input,state)
 {return (true11b(state) &&
         !does(‘white’,‘mark11’,input)
         !does('black','mark11 ‘,input))}
```

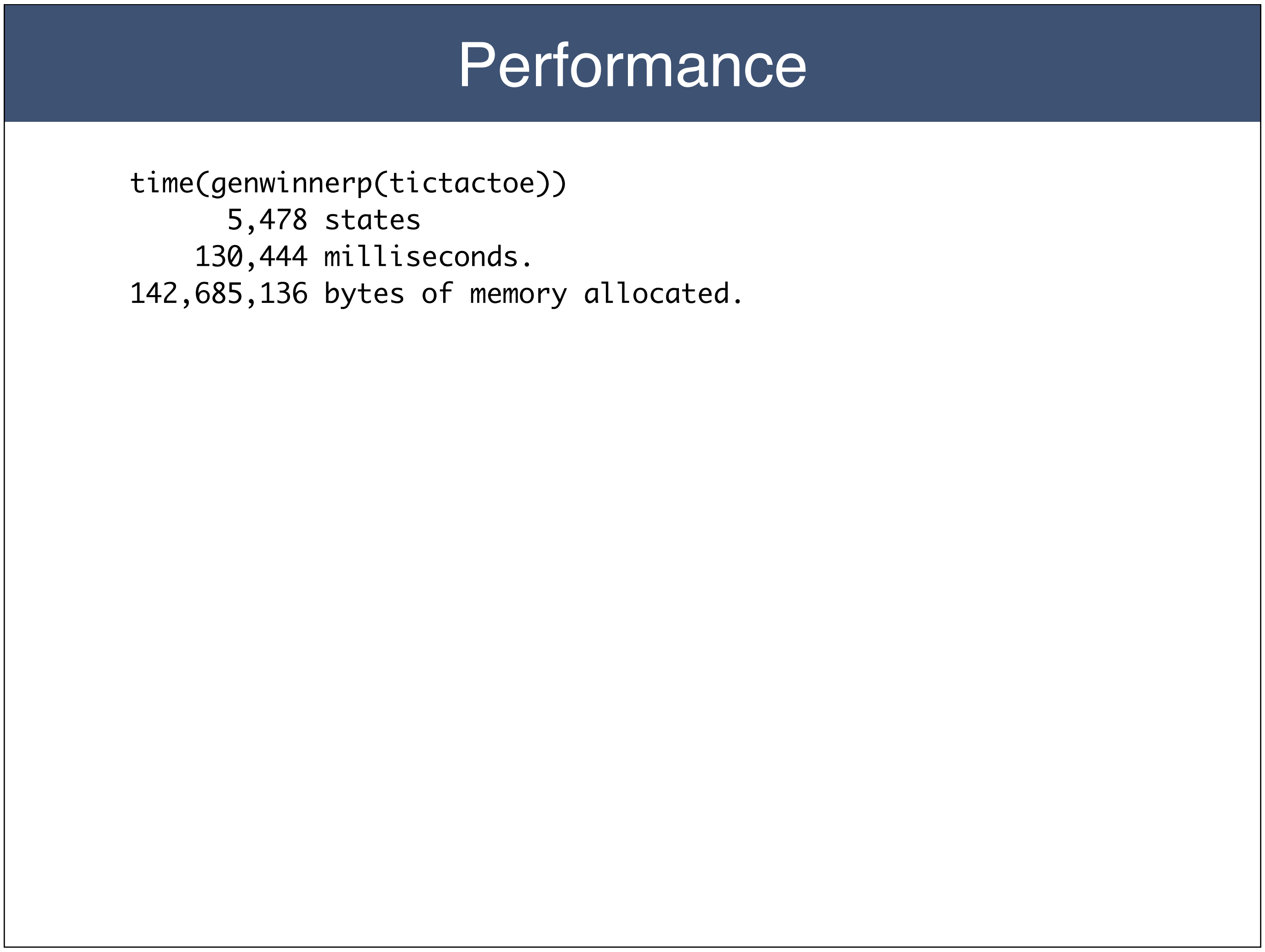
Here, for example, is an implementation of Tic Tac Toe in which we represent the state of the game a list of 29 Boolean values. That’s one bit for X, one bit for O, and one bit for blank in each of the 9 cells together one bit for control by white and one bit for control by black. (Obviously, we can do even better by exploiting mutual exclusions. For example, it is not possible for both white and black to have control at the same time, so we really need just one bit for control rather than two. But the translation from GDL in this way is easy; and, as we shall see, we still get plenty of benefit.) Okay, given this representation of state, we can define operations for testing states and updating states, and so forth. For example, we can determine whether there is an X in cell 1,1 by taking the zeroth component of our list of values. We can determine whether there is an O in cell 1,1 by taking the second component. We can compute update in similar fashion. If white does mark11 or there is already an X in cell 1,1, then there will be an X there in the next state. And so forth.



Implementation with Bit Vectors

```
function pbvinit ()
 {return #b00000000000000000000000000010}

function x11 () {return #b10000000000000000000000000000}
function o11 () {return #b01000000000000000000000000000}
function b11 () {return #b00100000000000000000000000000}

function true11x (state) {return state[0]}
function true11o (state) {return state[1]}
function true11b (state) {return state[2]}

function next (state)
 {return (complicated collection of logical operations)}
```

By using propositional bit vectors in place of lists of booleans, we can do even better. Here, we have defined our initial bit vector and bit vectors for each of the base propositions. We can then implement our subroutines by performing bit operations on these vectors. Doing things this way allows us to compute an entire state update in a single operation rather than operations for each proposition and thereby achieve greater efficiency.



Performance

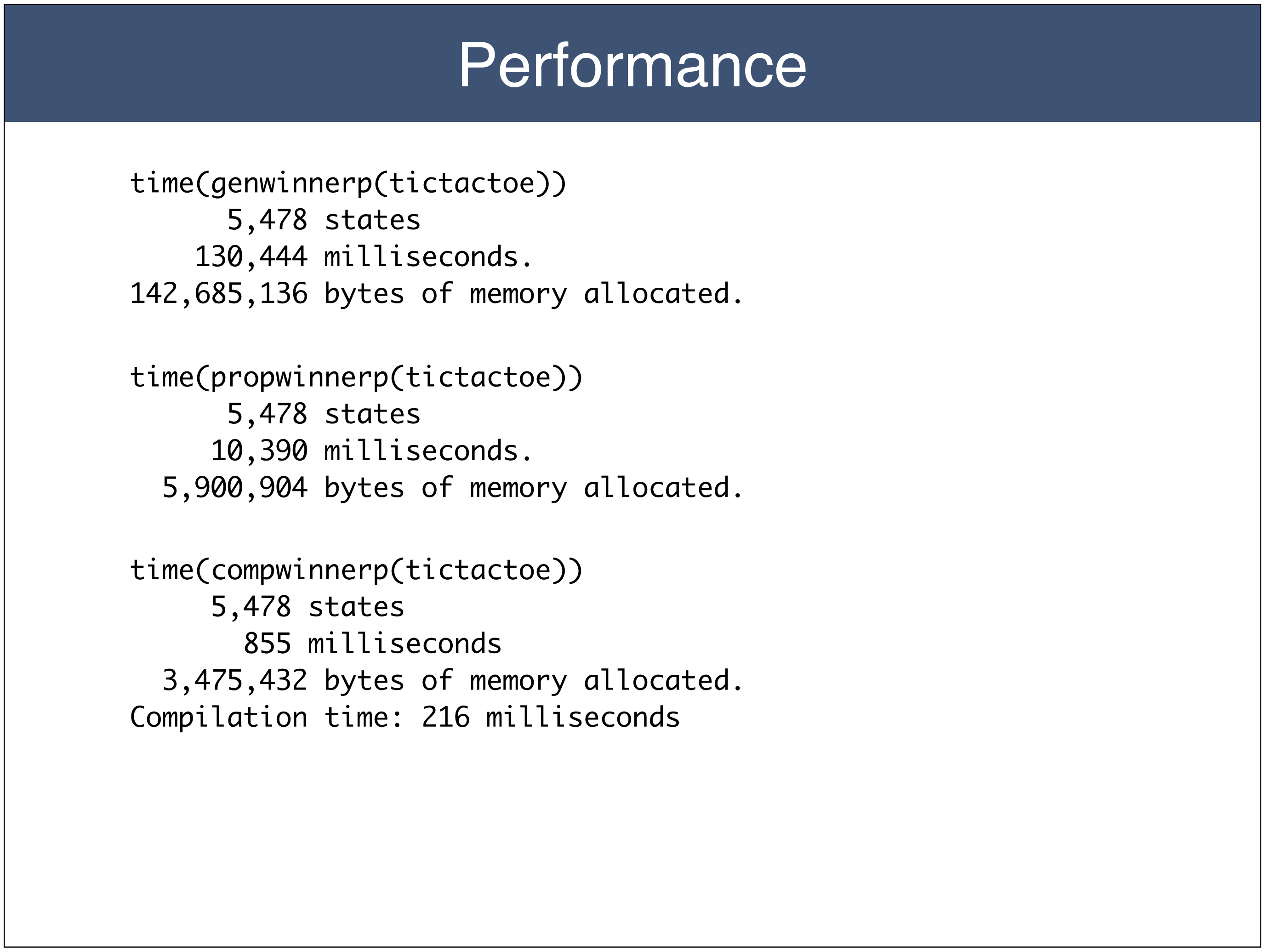
```
time(genwinnerp(tictactoe))
      5,478 states
    130,444 milliseconds.
142,685,136 bytes of memory allocated.
```

Does all of this automatic coding and compilation help us? Yes, quite a bit as it turns out. Here are the results from our previous experiment. 130 seconds for genwinnerp on the GDL description.

Performance

```
time(genwinnerp(tictactoe))
      5,478 states
    130,444 milliseconds.
142,685,136 bytes of memory allocated.

time(propwinnerp(tictactoe))
      5,478 states
     10,390 milliseconds.
  5,900,904 bytes of memory allocated.
```

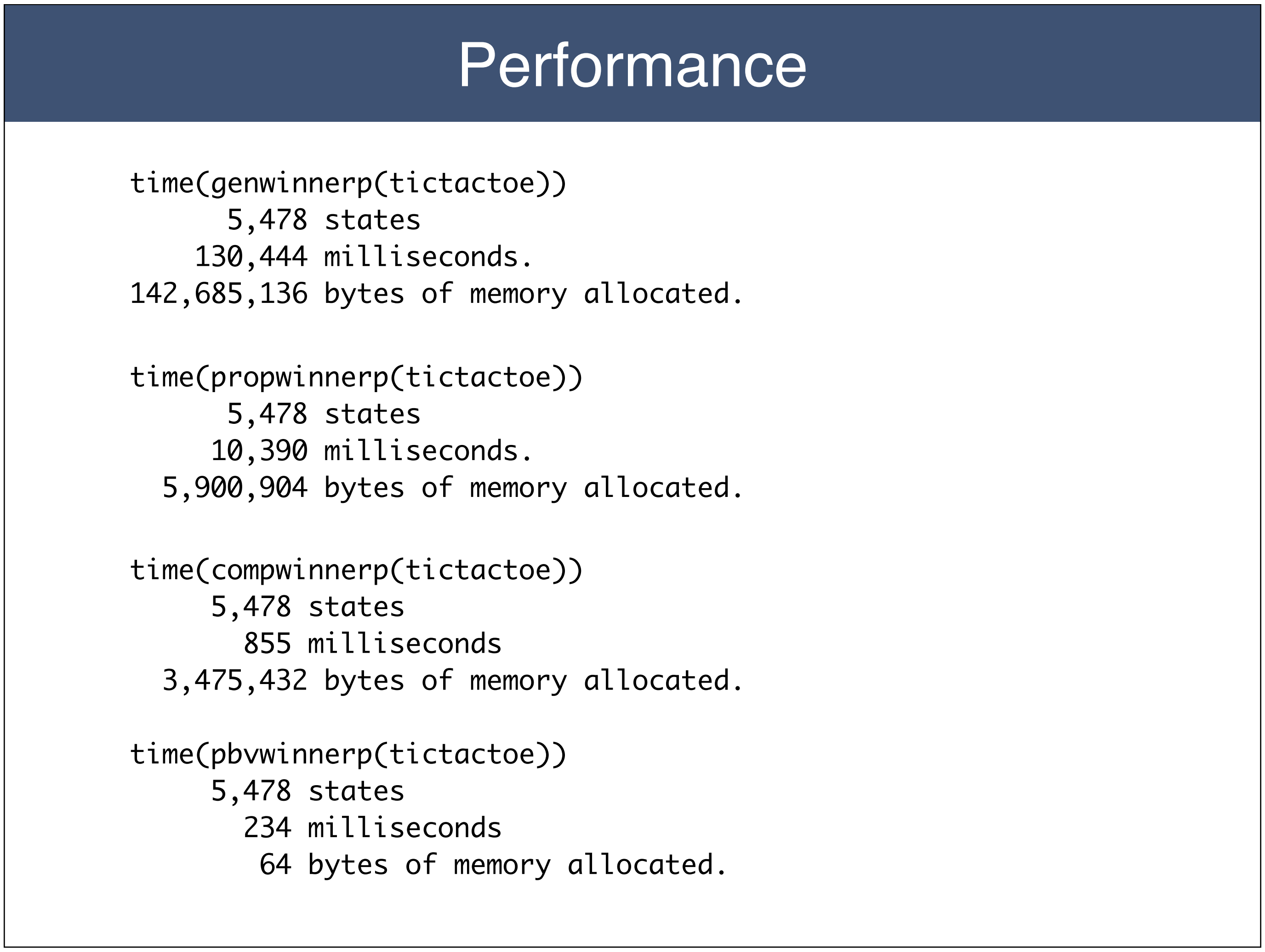
Just over 10 seconds for propwinnerp on the propositional net.



Performance

```
time(genwinnerp(tictactoe))
      5,478 states
    130,444 milliseconds.
142,685,136 bytes of memory allocated.

time(propwinnerp(tictactoe))
      5,478 states
     10,390 milliseconds.
  5,900,904 bytes of memory allocated.

time(compwinnerp(tictactoe))
     5,478 states
       855 milliseconds
  3,475,432 bytes of memory allocated.
Compilation time: 216 milliseconds
```

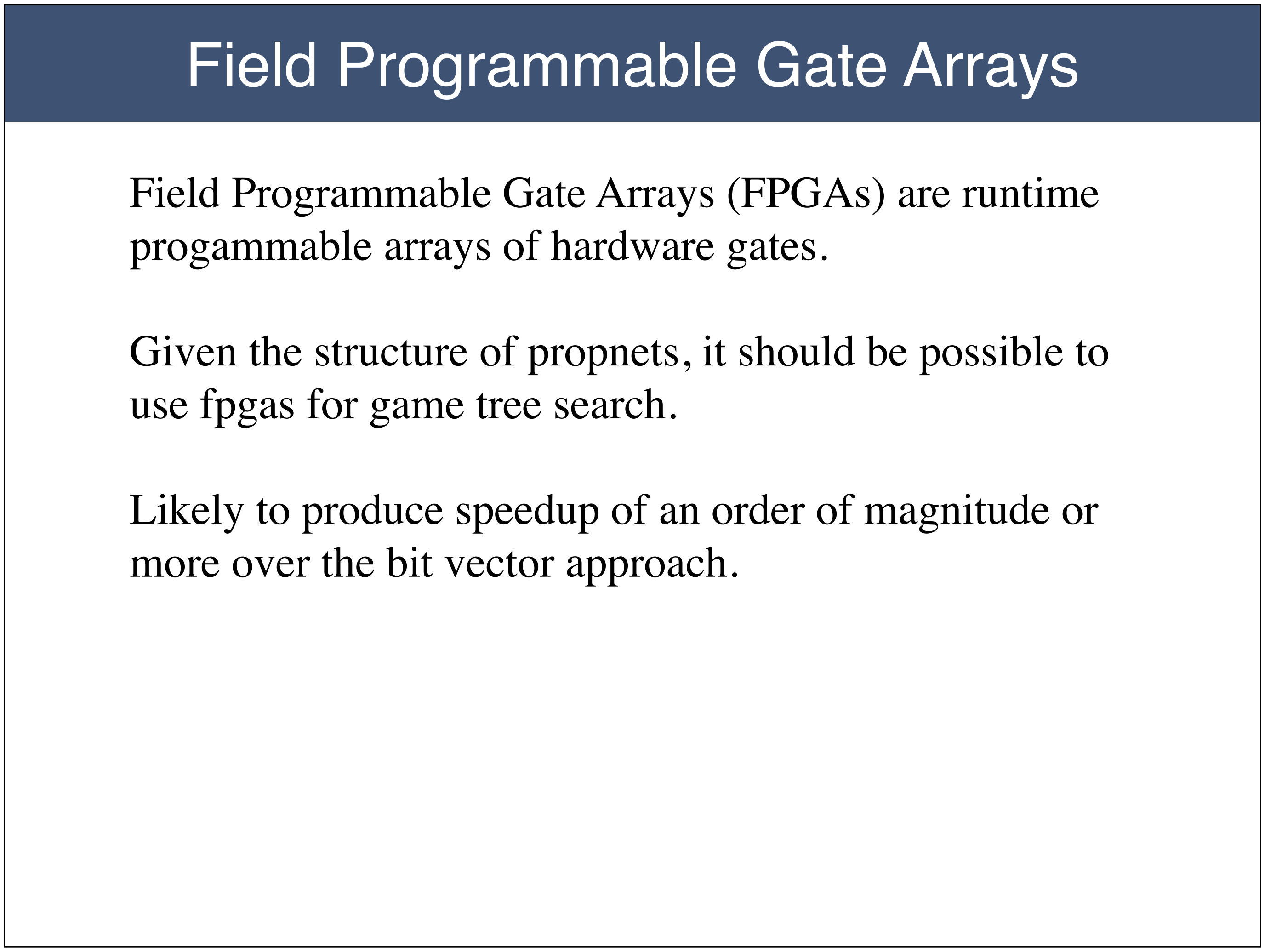
Using the list of values approach and compiling the resulting code, we see that the time drops to to under a second. And the memory usage drops to 3.5 Megabytes. (And in fact this memory usage can be improved significantly.)



Performance

```
time(genwinnerp(tictactoe))
      5,478 states
    130,444 milliseconds.
142,685,136 bytes of memory allocated.

time(propwinnerp(tictactoe))
      5,478 states
     10,390 milliseconds.
  5,900,904 bytes of memory allocated.

time(compwinnerp(tictactoe))
     5,478 states
       855 milliseconds
  3,475,432 bytes of memory allocated.

time(pbvwinnerp(tictactoe))
     5,478 states
       234 milliseconds
        64 bytes of memory allocated.
```

Moreover, moving to propositional bit vectors saves us even more. We are down to just 234 milliseconds and only 64 bytes of memory. This is a lot better than interpreted GDL.



Field Programmable Gate Arrays

Field Programmable Gate Arrays (FPGAs) are runtime progammable arrays of hardware gates.

Given the structure of propnets, it should be possible to use fpgas for game tree search.

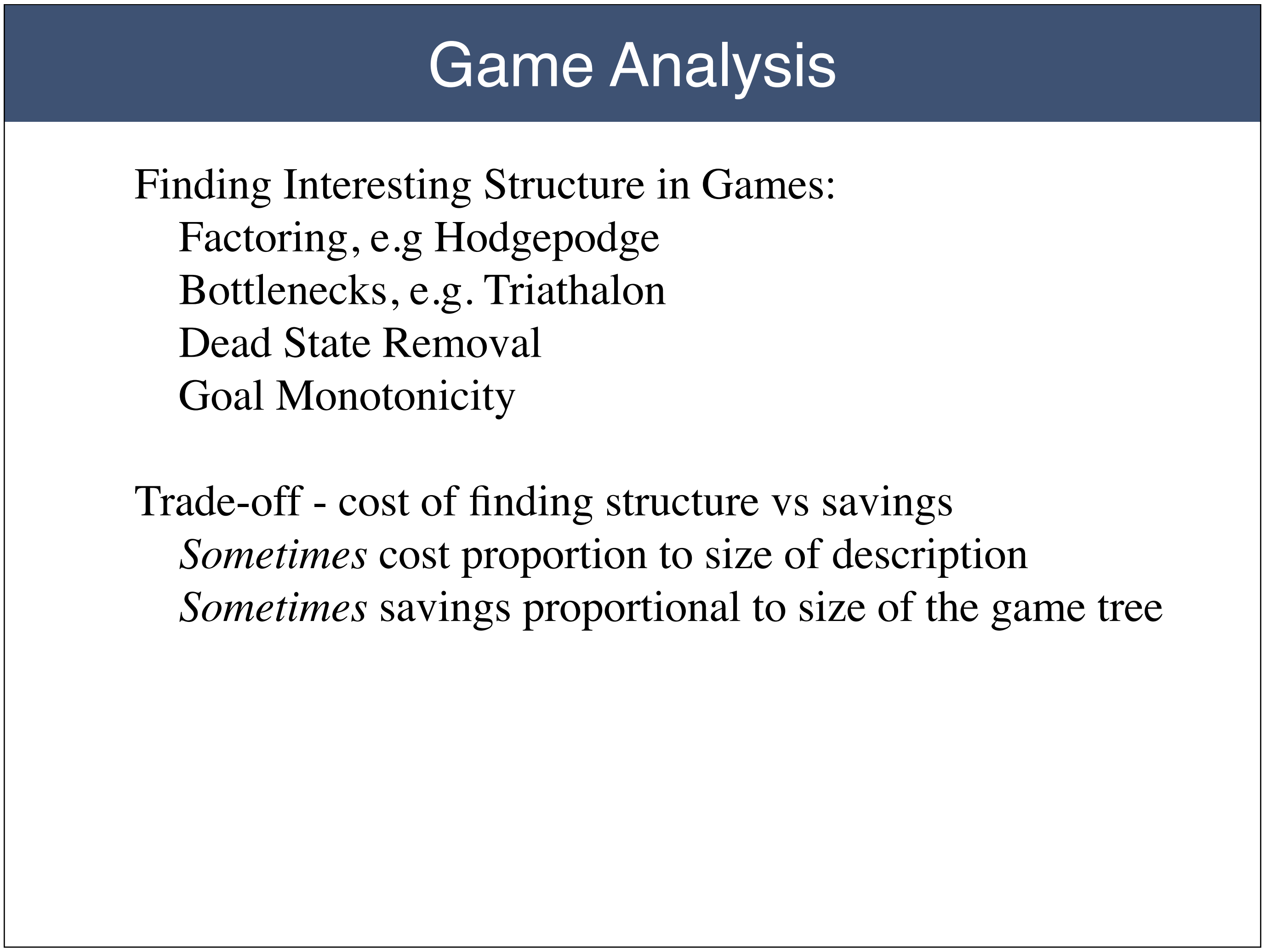
Likely to produce speedup of an order of magnitude or more over the bit vector approach.

That’s pretty impressive. However, it is in principle possible to do even better. One idea here is to use Field Programmable Gate Arrays (FPGAs). These are runtime-programmable arrays of hardware gates. Given the structure of propnets, it should be possible to use fpgas for game tree search. Although nobody has yet done this experiment, it seems likely that so doing could lead to further speedups of an order of magnitude or more.

GENERAL
GAME
PLAYING

GENERAL
GAME
PLAYING

Game Analysis with Propnets



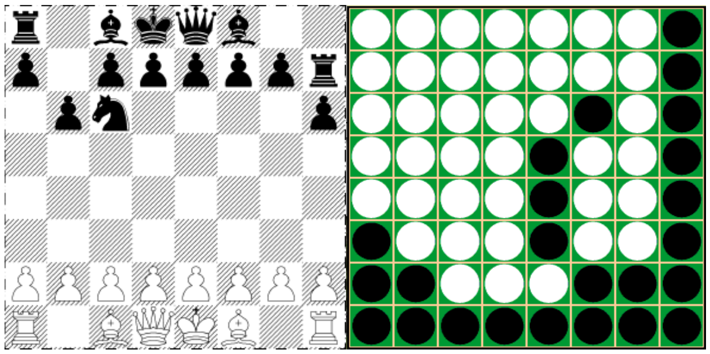
Game Analysis

Finding Interesting Structure in Games:
- Factoring, e.g Hodgepodge
- Bottlenecks, e.g. Triathalon
- Dead State Removal
- Goal Monotonicity

Trade-off - cost of finding structure vs savings
- *Sometimes* cost proportion to size of description
- *Sometimes* savings proportional to size of the game tree

We have seen that propositional nets can be used to advantage in searching game trees. The other main use of propnets is in analyzing games. For example, by analyzing the propnet for a game, it is sometimes possible for a player to detect structure that allows it to actually decrease the size of the game tree in significant ways. Game decomposition, or factoring, is a good example of this.

Compound Games



A *compound game* is a single game consisting of two or more individual games. The state of a compound game is a composition of the states of the individual games. On each step of a compound game, the players perform actions in each of the individual games. A compound game is over when either one or all of the individual games are over (depending on the type of compound game).

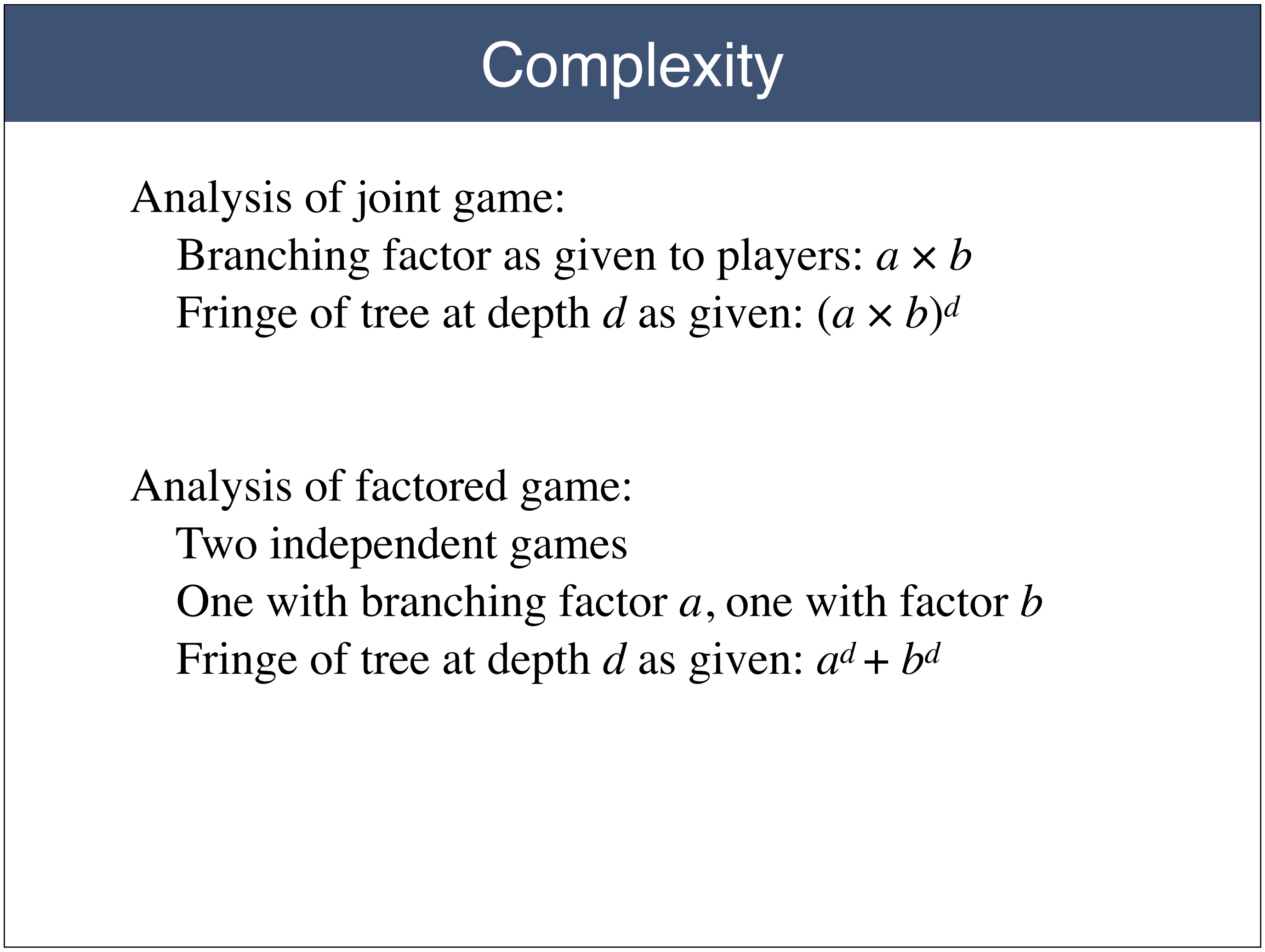
Complexity

Analysis of joint game:

Branching factor as given to players: $a \times b$

Fringe of tree at depth d as given: $(a \times b)^d$

Using the techniques we have seen thus far, compound games can be quite expensive to play. Unless a player recognizes that there are independent subgames, it is likely to search a game tree that is far larger than it needs to be. If one subgame has branching factor *a* and a second has branching factor *b*, then the branching factor of the joint game is $a{\times}b$, and the fringe of the game tree at depth *d* is likely to be something like $(a{\times}b)d$.



Complexity

Analysis of joint game:

Branching factor as given to players: $a \times b$

Fringe of tree at depth d as given: $(a \times b)^d$

Analysis of factored game:

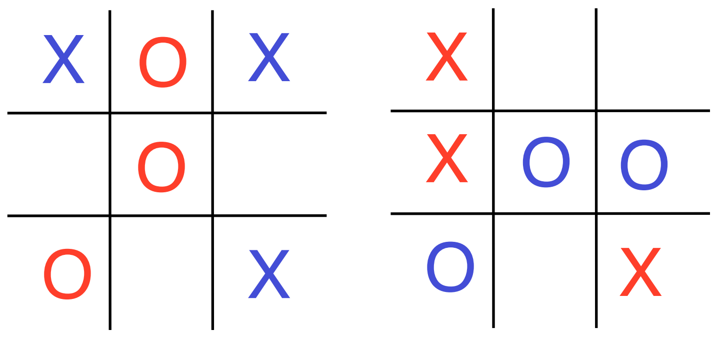
Two independent games

One with branching factor a, one with factor b

Fringe of tree at depth d as given: $a^d + b^d$

This is wasteful if the two subgames are independent. In that case, there are two trees - one with branching factor *a* and one with branching factor *b*, and the total size of the fringe of these trees at depth *d* should be only *ad*+*bd*.

Double Tic Tac Toe



Analysis of joint game:

Branching factor: 81, 64, 49, 36, 25, 16, 9, 4, 1

Branching factor: 9, 8, 7, 6, 5, 4, 3, 2, 1

Double Tic Tac Toe is another example. On each move players play on one of the two boards. First to win on either board wins the overall game. If the game is left unfactored, the branching factors are large - 81 possibilities on the first move, 64 on the second, and so forth. However, if the games are factored, the branching factors are more modest.

Game Factoring

Game factoring (aka game decomposition) is the process of discovering independent subgames inside larger games.

Game Playing with Factoring:
- Compute factors
- Use factors to generate submoves
- Assemble overall move from submoves

Frequently easier to discover factors with propnets than with other representations of games, such as state machines or even GDL.

Game factoring is the process of discovering independent games inside of larger games. Once discovered, game players can use these factors to play the individual games independently of each other and thus cut down on the combinatoric cost of playing such games. It turns out that it is often easier to discover independent subgames using the propnet representation of games rather than other representations.

Programme

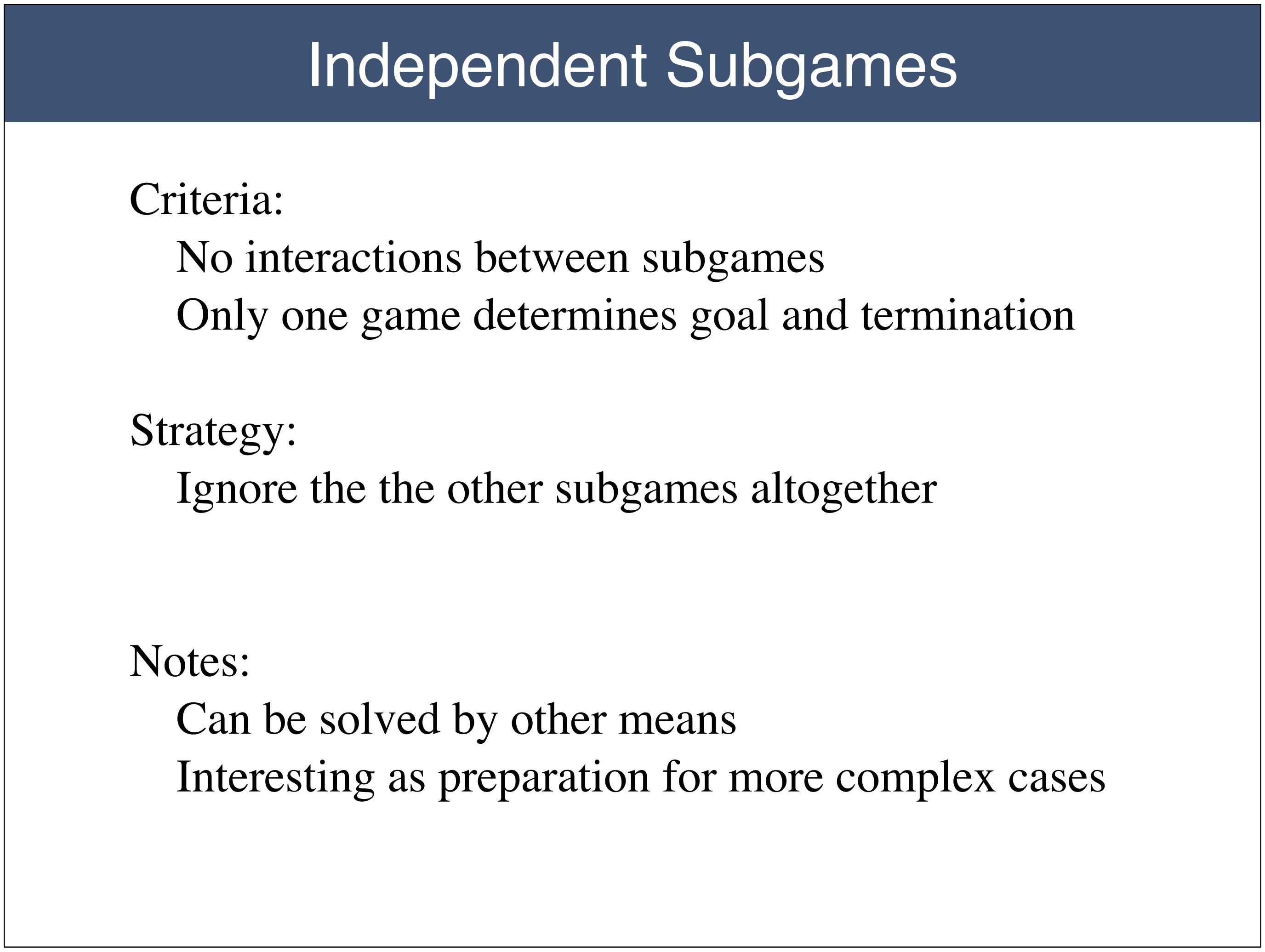
Independent subgames

Independent moves, interdependent termination

Interdependent actions, interdependent termination

Conditional factors

In this segment, we look at some elementary techniques for factoring games using propnets. We first talk about discovering factors with completely independent subgames; we then talk about factoring with interdependent termination and rewards; and after that we talk about factoring with interdependent actions.Finally, we discuss conditional factors, i.e. factors that appear only in certain states of games.



Independent Subgames

Criteria:

- No interactions between subgames
- Only one game determines goal and termination

Strategy:

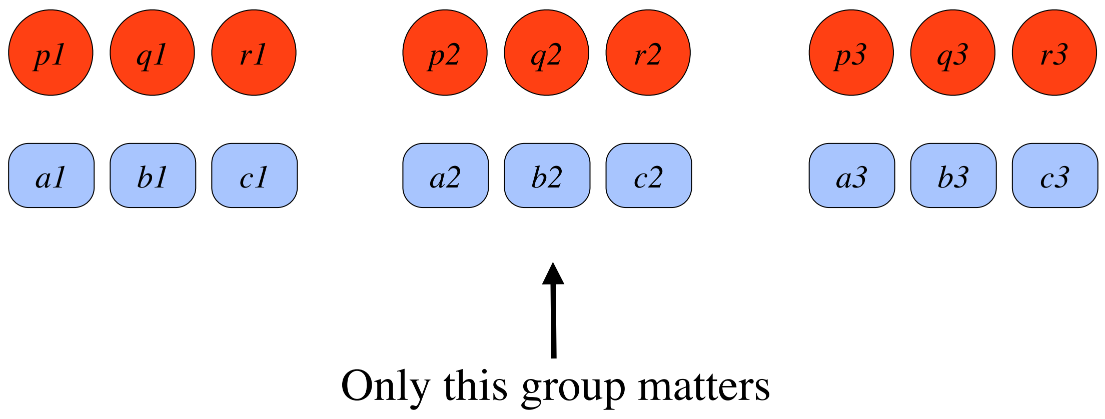
- Ignore the the other subgames altogether

Notes:

- Can be solved by other means
- Interesting as preparation for more complex cases

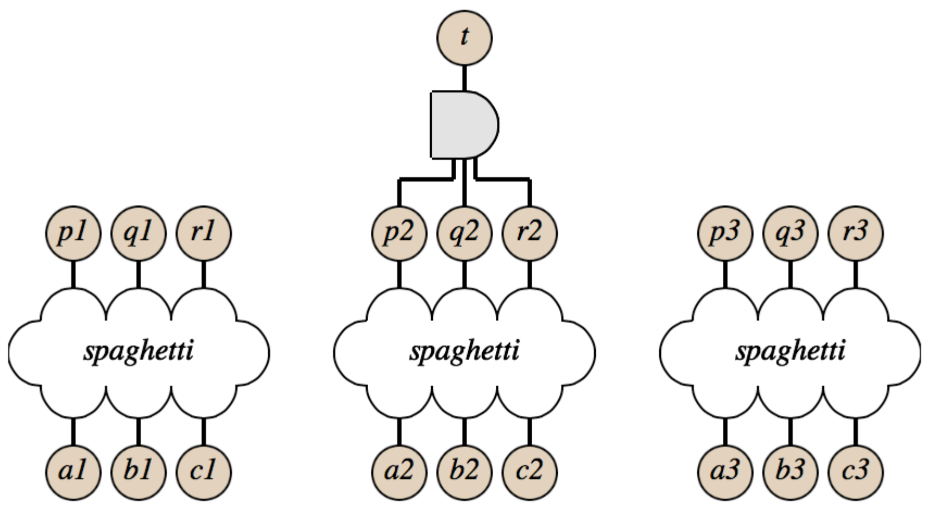
Let's begin our discussion of factoring with the simple case of compound games consisting of multiple completely independent subgames only one of which is relevant to the overall game. Without factoring, a player is likely to consider actions in all subgames when only one of the subgames matters. This is admittedly a very special case. It does not arise often, and it can be solved by means other than factoring (though not by the methods we have seen thus far). Nevertheless, it is worth considering because it prepares us for factoring more complicated games.

Multiple Buttons and Lights



Multiple Buttons and Lights is an example of a game of this sort. The overall game consists of multiple copies of Buttons and Lights. In each copy of Buttons and Lights, there are three base propositions (the lights) and three actions (the buttons). Pushing the first button in each group toggles the first light; pushing the second button in each group interchanges the first and second lights; and pushing the third button in each group interchanges the second and third lights. Initially, the lights are all off. The goal is to turn on all of the lights in the middle group. (The settings of the other lights are irrelevant.)

Propnet for Multiple Buttons and Lights



Here is a propnet for this game. There are three disjoint parts of the propnet - one portion for the first group of buttons and lights, a second portion for the second group, and a third portion for the third group. Note that the goal and termination conditions are based entirely on the lights in the second group.

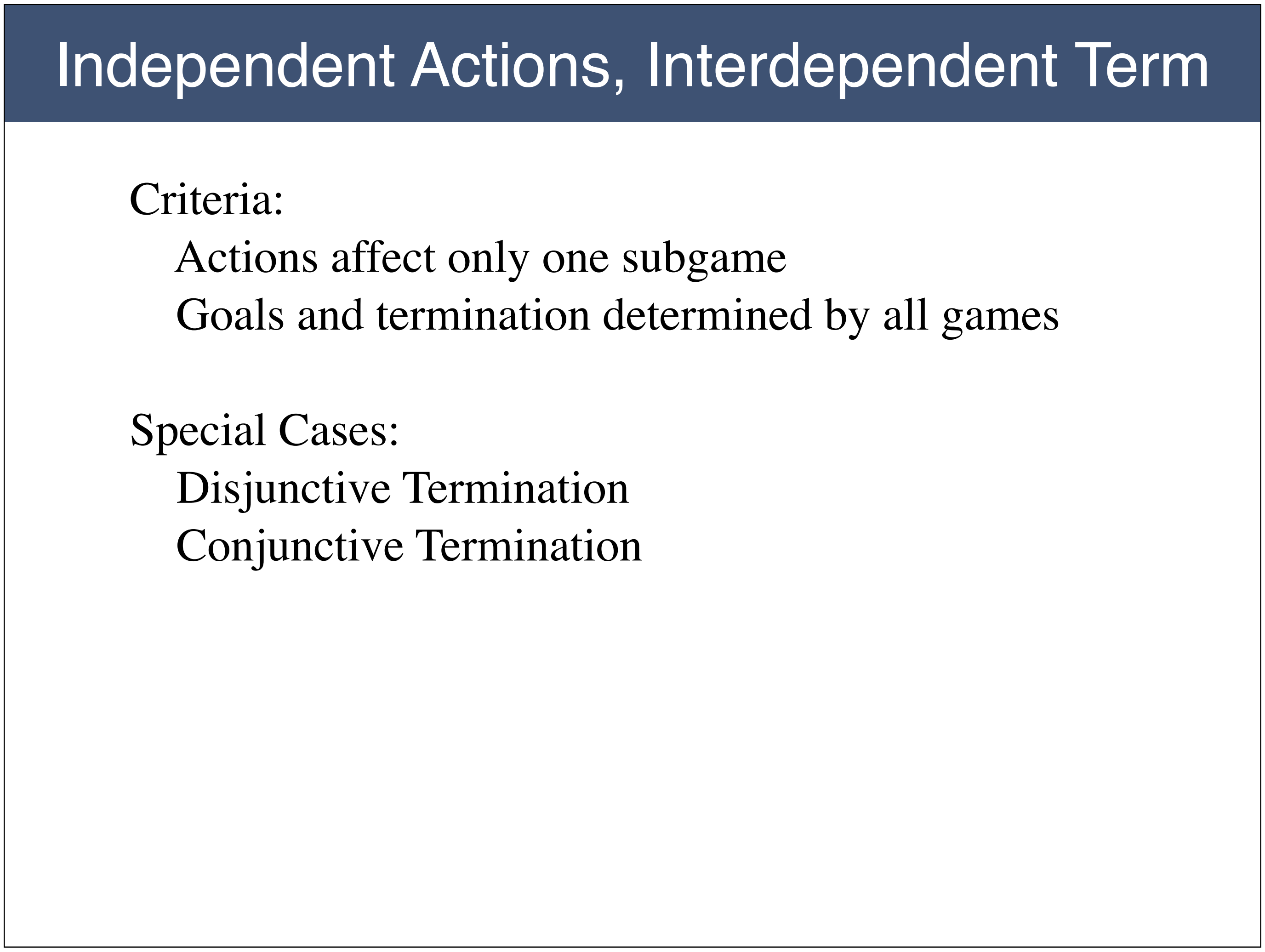
Independent Subgames

Solution
- Retain essential component and discard others
- Play as usual

Multiple Player versions
- Technique similar

Looking at the propnet for this game, it is easy to see that the game has a very special structure. The propnet consists of three completely disconnected subnets, one for each subgame. Finding factors in situations like this is easy - it can be computed in time that is polynomial in the size of the propnet. Having found these factors, the solution in this case is simple. The player just keeps the essential component and discards the others.

Note that this technique can be applied equally well to multi-player games. Consider, for example, Multiple Tic Tac Toe, i.e. three games of Tic Tac Toe glued together in which only the middle game matters. The propnet for Multiple TicTacToe is similar to the propnet for Multiple Buttons and Lights, and it is possible to find the structure for Multiple TicTacToe just as easily as for Multiple Buttons and Lights.



Independent Actions, Interdependent Term

Criteria:

- Actions affect only one subgame
- Goals and termination determined by all games

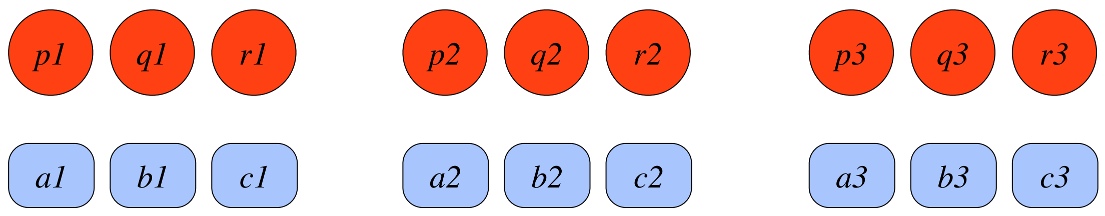
Special Cases:

- Disjunctive Termination
- Conjunctive Termination

Now, let’s consider a slightly more complicated case, viz. *compound games with interdependent termination*. (We can assume interdependent goals as well, but we will ignore goals for now and just focus on termination.) As with the case of independent subgames, actions are partitioned over distinct subgames and there are no incoming connections between those subgames. The main difference is that the termination condition for the overall game can depend on more than one and perhaps all of the subgames.

In games of this sort, the termination of the overall game can be defined as any boolean combination of conditions in the individual subgames. In the case where the combination is a disjunction, the game is said to have *disjunctive termination*. In the case where the combination is a conjunction, the game is said to have *conjunctive termination*.

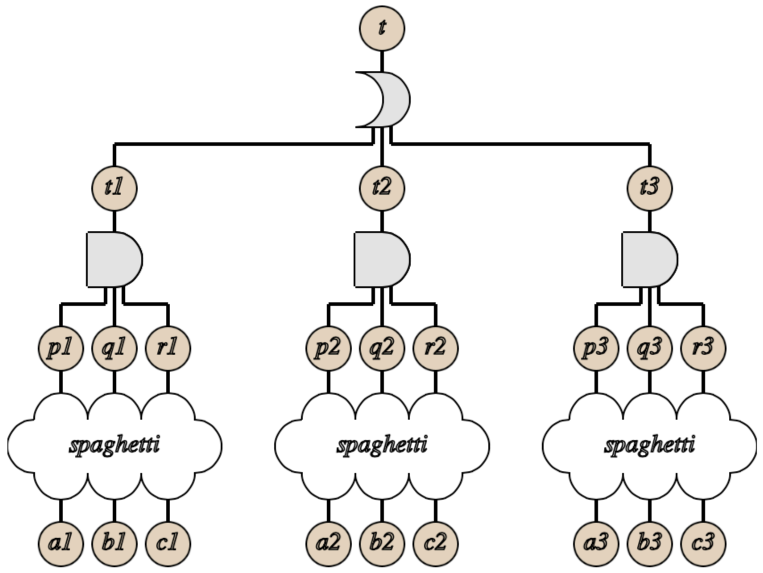
Best Buttons and Lights



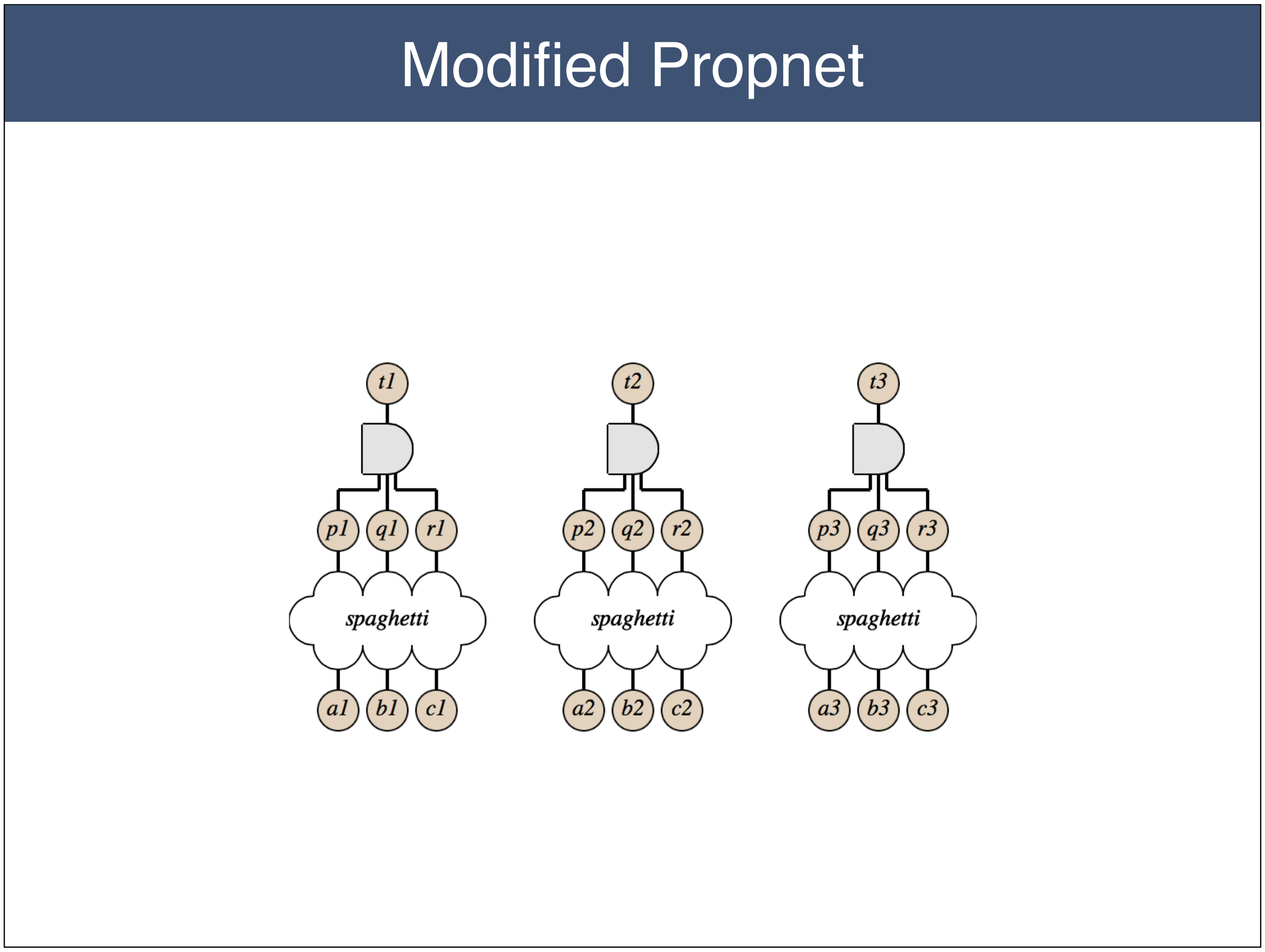
Terminates whenever *any* group matters

As a simple example of a factorable game with disjunctive goals, consider Best Buttons and Lights. In Multiple Buttons and Lights, only one group of buttons and lights matters. In Best Buttons and Lights, the compound game terminates whenever *any* group terminates. This gives the player the freedom to play whichever subgame it likes and rest assured that, if it succeeds on that subgame, it succeeds on the overall game with the same score.

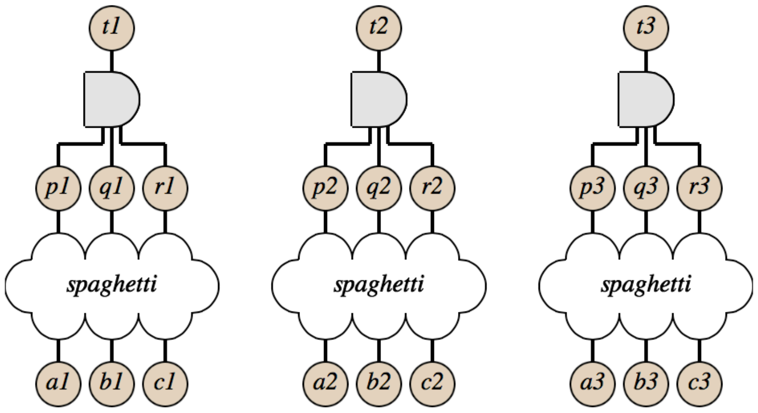
Propnet for Best Buttons and Lights



Here is a propnet for Best Buttons and Lights. It is similar to the propnet for Multiple Buttons and Lights except that the goals and termination are based disjunctively on all three subgames.

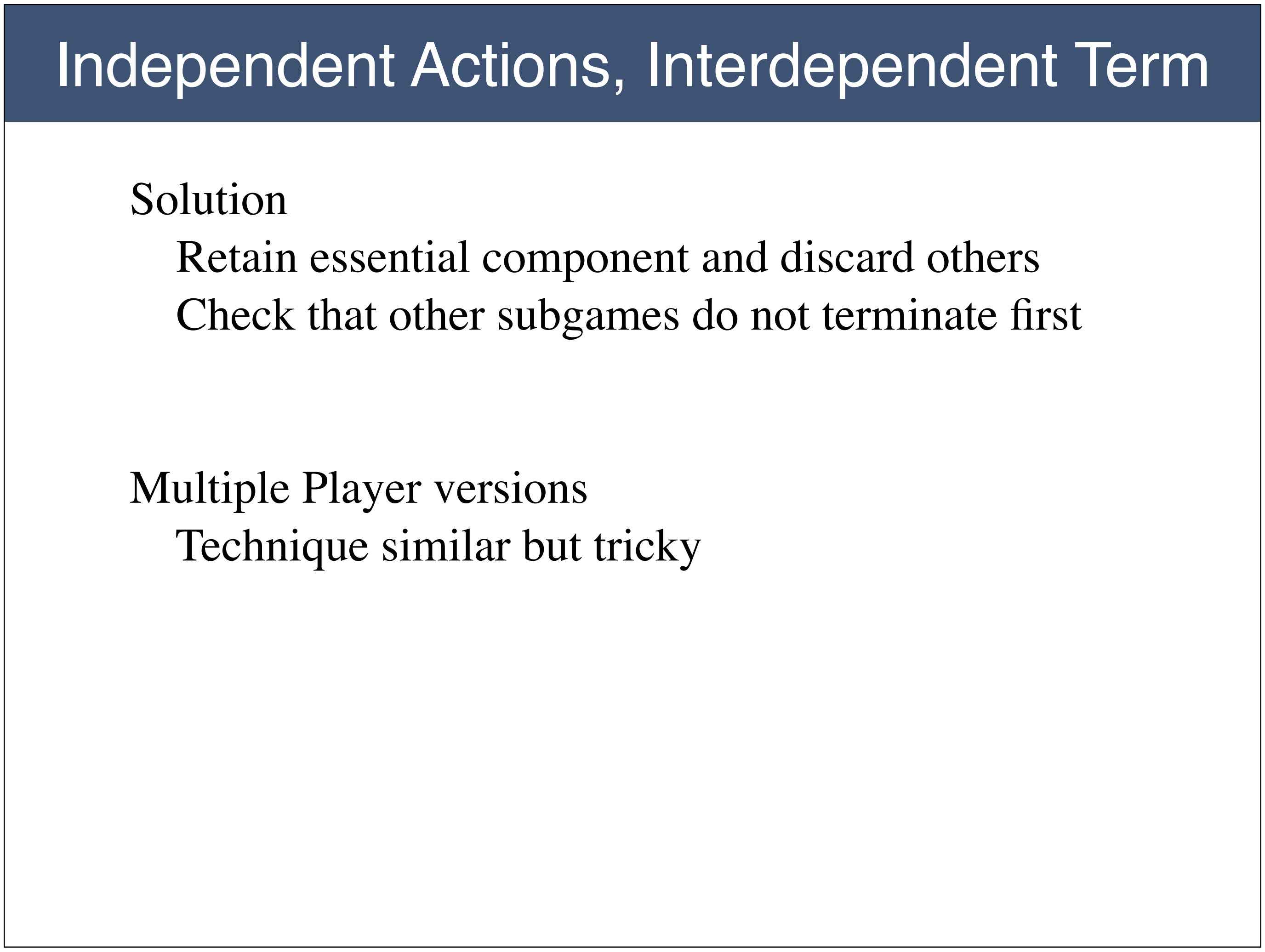


Modified Propnet



The good news is that we can extend the techniques discussed in the preceding section to this case. Let's consider the disjunctive case. If the connective leading to a termination is a disjunction with its inputs in turn supplied by nodes in different subgames, then we simply cut off that node and inputs to the or-gate termination nodes for the overall game. We repeat this process so long as we encounter only disjunctions. If the game is truly disjunctive, this will lead to a separable propnet.

If this process succeeds in factoring the propnet, then the player simply picks one of the subgames and proceeds as with the case of independent subgames. Alternatively and better, the player tries playing all of the subgames and picks the one with the best score. Or at least that is the basic idea.



Independent Actions, Interdependent Term

Solution
- Retain essential component and discard others
- Check that other subgames do not terminate first

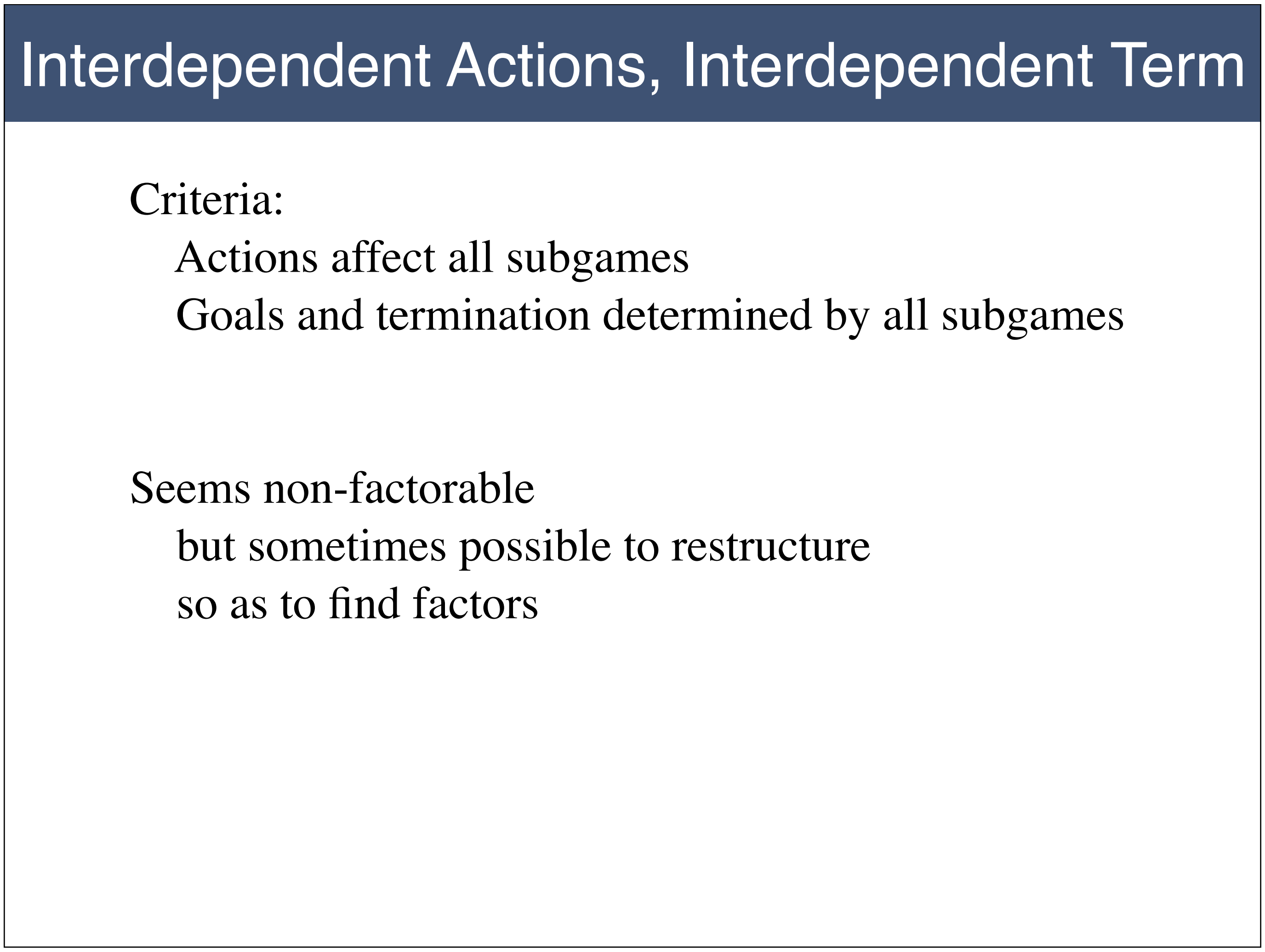
Multiple Player versions
- Technique similar but tricky

Unfortunately it is not quite that simple. There is a problem that does not arise in the case of completely independent subgames. The player may choose a subgame, find a winning strategy, and begin executing that strategy. Unfortunately, in the course of execution on the chosen subgame, one of the other subgames may terminate, terminating the game as a whole before the strategy in the chosen subgame is complete. This can lead to a lower score than the player might have expected.

The solution to this problem is to check each of the subgames for termination when no actions are played. The player takes the shortest time period and plays each of the other subgames with that as step limit and takes the one that provides the best result. For the subgame with the shortest termination, if there is no other game with that step limit, the player tries the next shortest step limit.

Although in this approach, the player searches all of the subgames more than once, this is usually a lot less expensive that searching the game tree for the unfactored game because it is not cross multiplying the branching factors of the independent games as it would if it did not use the game's factors.

Once again, it is possible to extend this technique to multiple player games, though this must be done with some care.



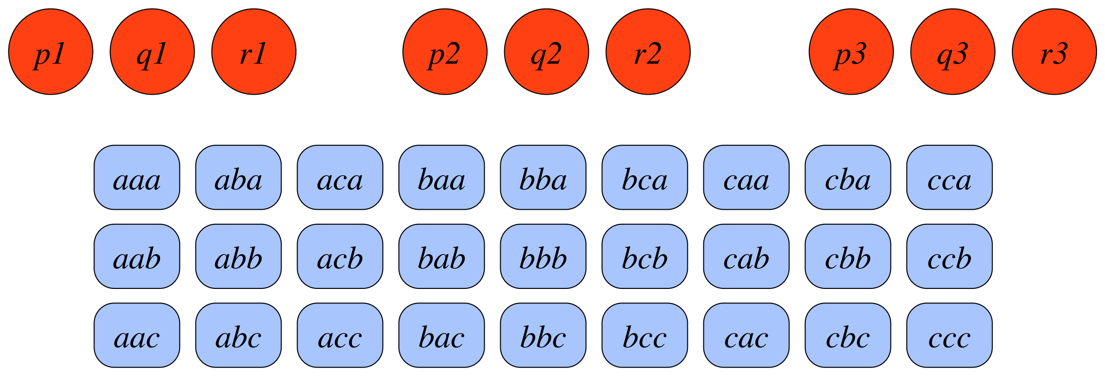
Interdependent Actions, Interdependent Term

Criteria:
- Actions affect all subgames
- Goals and termination determined by all subgames

Seems non-factorable
- but sometimes possible to restructure
- so as to find factors

Finally, let’s look at compound games with interdependent actions. By interdependence of actions, we mean that those actions have effects on more than one of the subgames of the compound game. On first blush, it might seem that games of this sort are not factorable. However, this is not necessarily true. Under certain circumstances, even with interdependent actions, it is possible to identify factors and use those factors to search the game tree more efficiently than without considering these factors.

Joint Buttons and Lights



As an example, consider the game of Joint Buttons and Lights. As with the other Buttons and Lights variants that we have seen, the lights in this game are organized into groups of three. However, unlike the previous variants, buttons are not associated with specific groups. Instead, each button has effects on all groups. Button *aaa* toggles the first light in the first group *and* the first light in the second group *and* the first light in the third group. Button *aab* toggles the first light in the first group and the first light in the second group and interchanges the value of the first light and the second light in the third group. And so forth.

Solution

Actions must be decomposed into actions for each factor and recomposed into joint actions.

More complex than previous cases.

Although all of the buttons in this game affect all of the subgames, the game is still factorable with just three actions per factor. The reason is that there is one button in the compound game for each combination of actions in the other two subgames. Thus the game trees for each subgame can be searched independently of each other and the actions chosen can be reassembled into overall actions of the compound game. For example, if action *a* is chosen in the first and second subgames and action *c* is chosen in the third subgame, then action *aac* can be performed in the compound game.

Solution

Actions must be decomposed into actions for each factor and recomposed into joint actions.

Doable but more difficult than with other cases
- Group actions into equivalence classes
- Check that equivalence classes allow *lossless joins*

Recognizing when this can be done is not easy, but it is doable. There are two steps. In the first step, the player groups actions into equivalence classes of actions for each subgame. In the second step, the player determines whether these classes satisfy the lossless join property.

Finding equivalence classes is done by looking at the propnet for each subgame. Two actions are equivalent if they are used in the same way in the propnet for that subgame. For example, if each action is input to an and-gate with the same other input and if the outputs are connected by an or-gate, then the effects of one action cannot be distinguished form the effects of the other action; hence they are equivalent.

The process of finding equivalence classes is repeated for each potential subgame. In general, the equivalence classes for each subgame will be different. In fact, as we shall see, in order for the game to be factorable, they must be different.

Once a player has equivalence classes for each potential subgame, it then checks whether those equivalence classes are independent of each other. The criterion is simple. Each equivalence class in one potential subgame must have a non-empty intersection with each equivalence class of every other potential subgame. If this is true, then the partitions pass the lossless join criterion.

Solution

Actions must be decomposed into actions for each factor and recomposed into joint actions.

Doable but more difficult than with other cases
- Group actions into equivalence classes
- Check that equivalence classes allow *lossless joins*

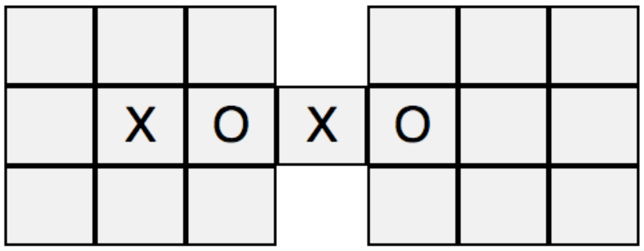
If successful
- modifies propnets with equivalence classes as action
- proceeds as before to find solutions
- chooses action from equivalence class

See notes for more detail.

If a player finds equivalence classes for various two potential subgames and they pass this lossless join test, then the player can factor the game into subgames. In order to benefit from the factoring, the player must modify each propnet so that the individual actions are replaced with the equivalence classes of which they are members. This cuts down on the number of possible actions to consider. Otherwise, the branching factor of the game trees for the subgames would be just as large as the branching factor for the overall game.

Once this is done, the player can then search the game trees for the different subgames to select actions to perform in each game. It can then find an action in the compound game for the particular combination of subgame actions. This is always doable provided that the partitioning of actions satisfies the lossless join property, which has already been confirmed. The player then performs any action in the intersection of the equivalence classes chosen in this process.

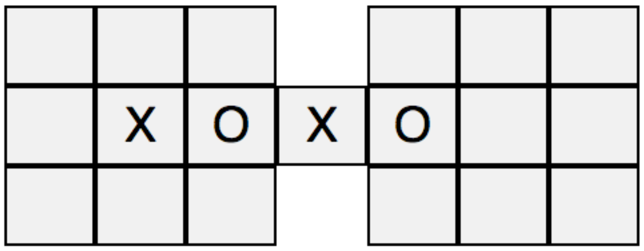
Conditional Factoring



Finally, let's consider conditional factoring, i.e. factoring in cases where a game is not factorable to start but, over time, become factorable. Consider the following game, called Linked Tic Tac Toe - two games of Tic Tac Toe connected by a single square that connects the two. The goal of the game for a player is to get two lines, a row, column or diagonal of that player's mark, with at least two of the marks residing in a specific tic tac toe domain. Diagonals through the middle square do not count. On each turn, the player in control can place two marks, either one in each distinct Tic Tac Toe domain or one mark in a Tic Tac Toe domain and one in the center square. Suppose that the state of the game is as shown here.

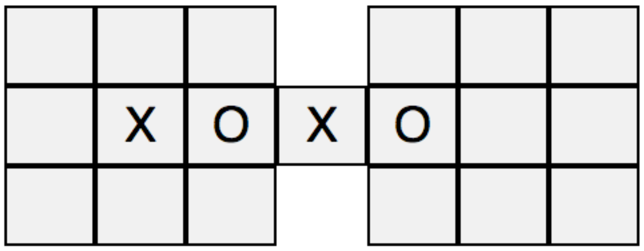
Once it is not possible for either player to achieve a row utilizing the center square, the only possible solutions lie in the domains of the Tic Tac Toe games that are joined by the center square. The states of the two Tic Tac Toe games can be considered independently to find the remaining optimal moves for the duration of the game. Only the game trees for each Tic Tac Toe game, modulo the center square, need to be searched to determine the remaining optimal moves. Given the current state, the game can be factored into two independent sub-games. However, from the initial state,this game cannot be factored into independent games, because the shared middle square intertwines the two domains as it is relevant to the satisfaction of goals in both sub-games.

Conditional Factoring



While this game is not factorable in general, it is factorable contingent upon entering a state in which no row through the middle is possible for either player. We define the game as being contingently factorable, since it reduces to independent simultaneous sub-games, given that it enters a certain state. The raw computational benefit acquired from recognizing contingent factors is less than that of recognizing initial factors, because it requires that the game be in a specific state. However, the relative computational savings are still the same, because the number of accessible fringe nodes reduces to the sum of the remaining accessible fringe nodes of the individual games rather than the product.

Conditional Factoring



At this point in time, there are no established techniques for discovering and exploiting cases of conditional independence. However, it seems likely that some of the techniques just discussed can be adapted to this case as well.

Other Types of Game Analysis

Bottlenecks
- Series of games
- each of which must terminate before next begins

Dead State Elimination
- Find states that cannot lead to acceptable outcomes
- Prune whole subtrees
- See Notes.

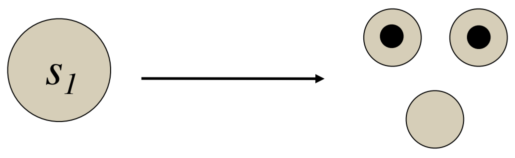
Goal Monotonicity
- Detect monotonicity in states
- e.g. higher goal value in non-terminal states
  correlated with progress toward goal

Game factoring is complicated but worthwhile. It can significantly decrease the cost of game playing. However, it is not the only sort of propnet-based game analysis that is worthwhile. For example, propnets can be used to find bottlenecks, i.e. games structured in series, where the player must win the first game before playing subsequent games. Pronets can be used to eliminate dead states, i.e. states that cannot lead to an acceptable solution and which can, therefore, be pruned from further consideration. And propnets can be use to detect certain types of monotonicities, i.e. cases where higher goal values on non-terminal states correlate with progress toward terminal states with acceptable goal values. And there is much more research to be done in this area.

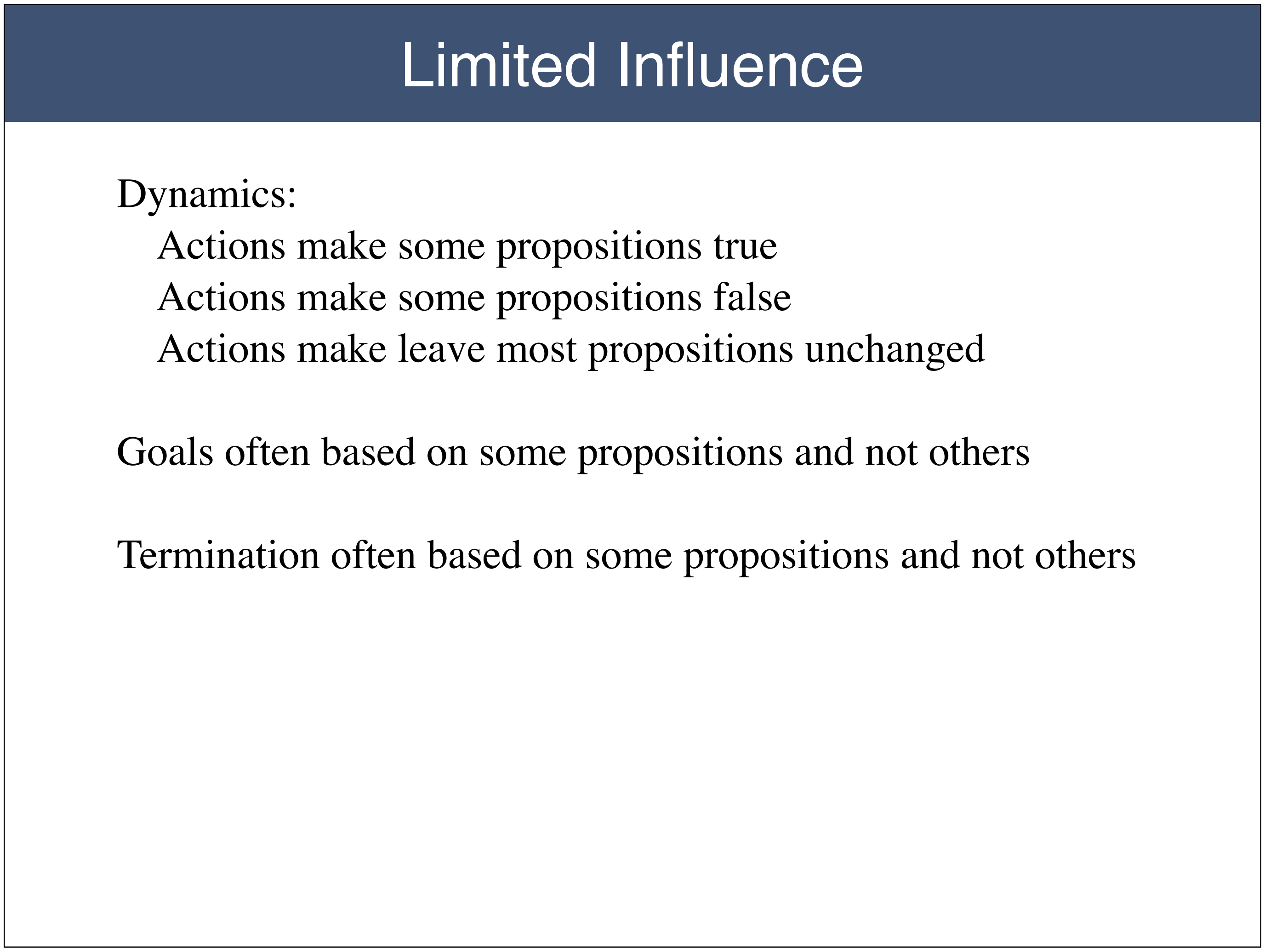
GENERAL
GAME
PLAYING

GENERAL
GAME
PLAYING

States and Propositions



In the vast majority of games, states and actions are not monolithic - they can be defined in terms of more fundamental entities. In Chess, for example, states can be conceptualized in terms of the locations of individual pieces one the Chess board. In Risk, where many pieces are moved at once, overall actions can be defined in terms of the movements of individual pieces.



Limited Influence

Dynamics:
- Actions make some propositions true
- Actions make some propositions false
- Actions make leave most propositions unchanged

Goals often based on some propositions and not others

Termination often based on some propositions and not others

Also, in the vast majority of games, the effects of actions are local. As actions are performed, some propositions become true and others become false. However, the truth values of most propositions are independent.