Planning for Interactions among Autonomous Agents

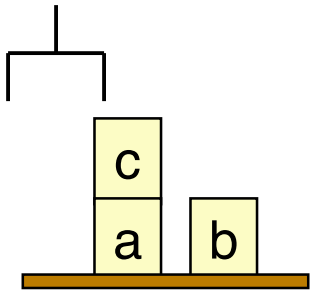
Dana Nau
University of Maryland
College Park, MD

Laboratory for **Computational** Cultural Dynamics

- Highly cross-disciplinary partnership at the University of Maryland
  - **Computer Science**: Dorr, Getoor, Nau, Subrahmanian, Varshney
  - **Political science**: Telhami, Wilkenfeld
  - **Psychology**: Kruglanski
  - **Criminology**: LaFree
  - **Linguistics**: Resnik
  - **Public Policy**: Steinbruner
  - **Business**: Fu, Raschid
  - **Systems Engineering**: Nau, Fu, Silverman (UPenn)
- Objective:
  - Make contributions to the behavioral and social sciences analogous to those made by CS to the biological sciences during the past decade

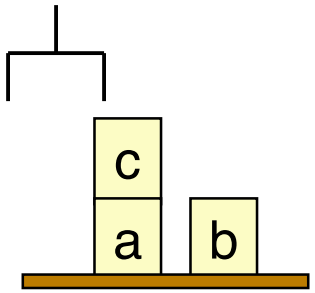
Planning How to Interact with Other Agents

- AI planning research has usually assumed there's just one agent in the world
  - » The plan executor



Planning How to Interact with Other Agents

- AI planning research has usually assumed there's just one agent in the world
  - » The plan executor



- But the universe contains other agents
  - » We may need to interact with them
- Usually they are not under our control
  - » We can't be sure what they're going to do
- To carry out our plans successfully, we need ways to deal with the things they *might* do

Planning How to Interact with Other Agents

I'll discuss two cases:

(1) Generate plans that will achieve our goals regardless of what the other agents do

- » Combine two techniques

  BDDs, to reason about abstract sets of states

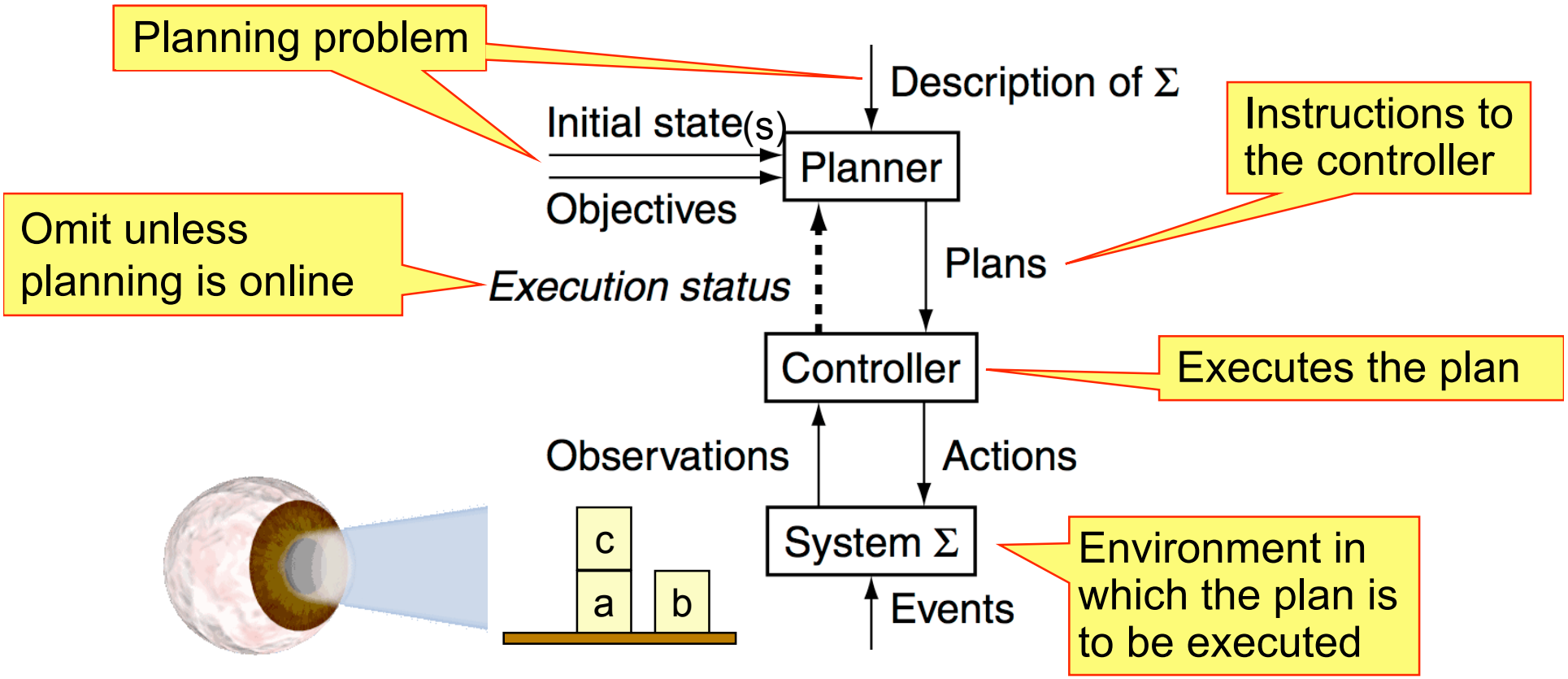
  HTNs, to keep focused on what's relevant

- » Experimental results: Hunter-Prey problems

(2) Generate plans that are likely to work well if the other agents behave the way we expect

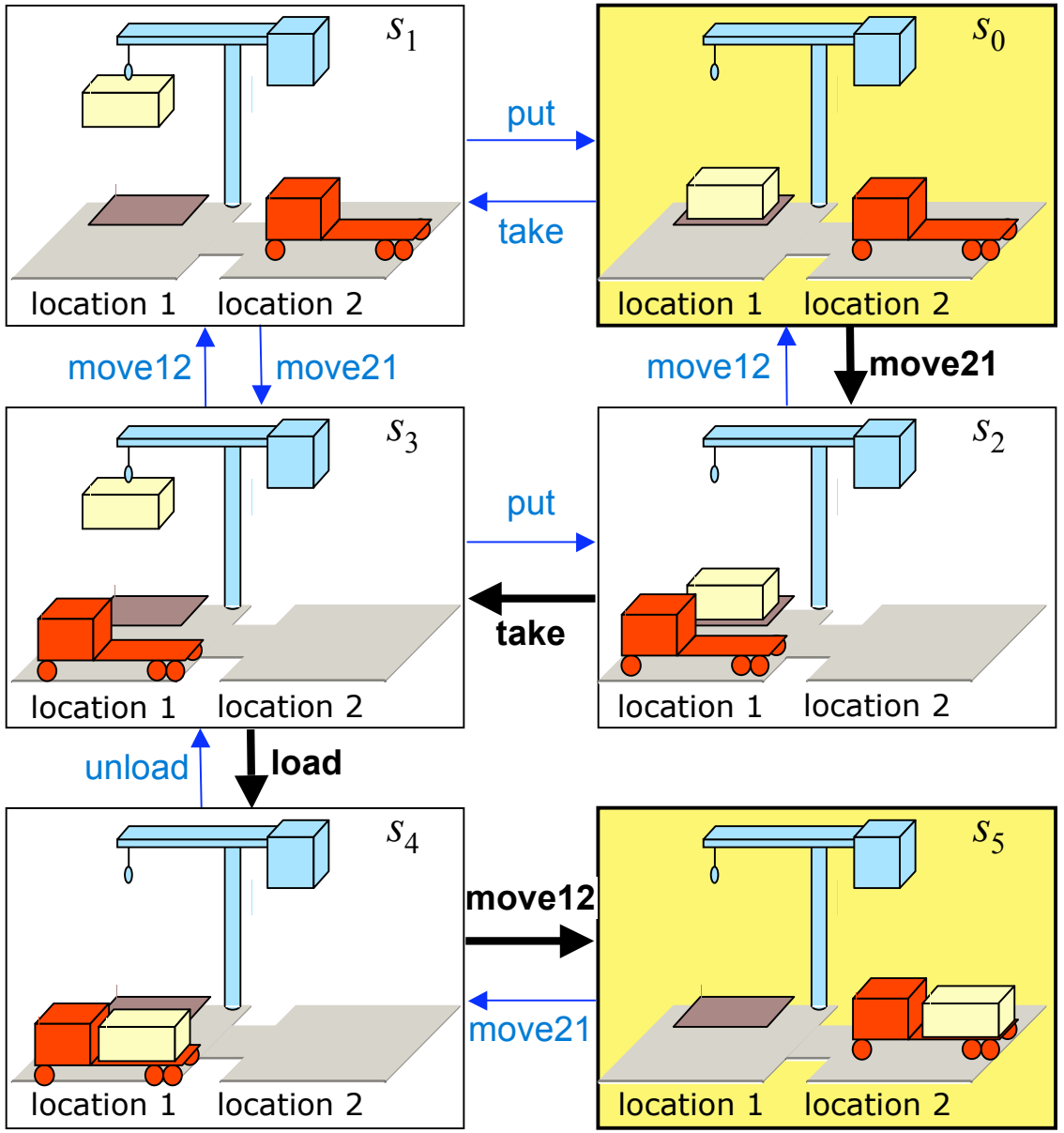
- » Build predictive models from observations of other agents' behavior
- » Use these models to plan our actions and to filter noise
- » Results from the 20th Anniversary Iterated Prisoner's Dilemma Competition

Extremely Quick Review of AI Planning



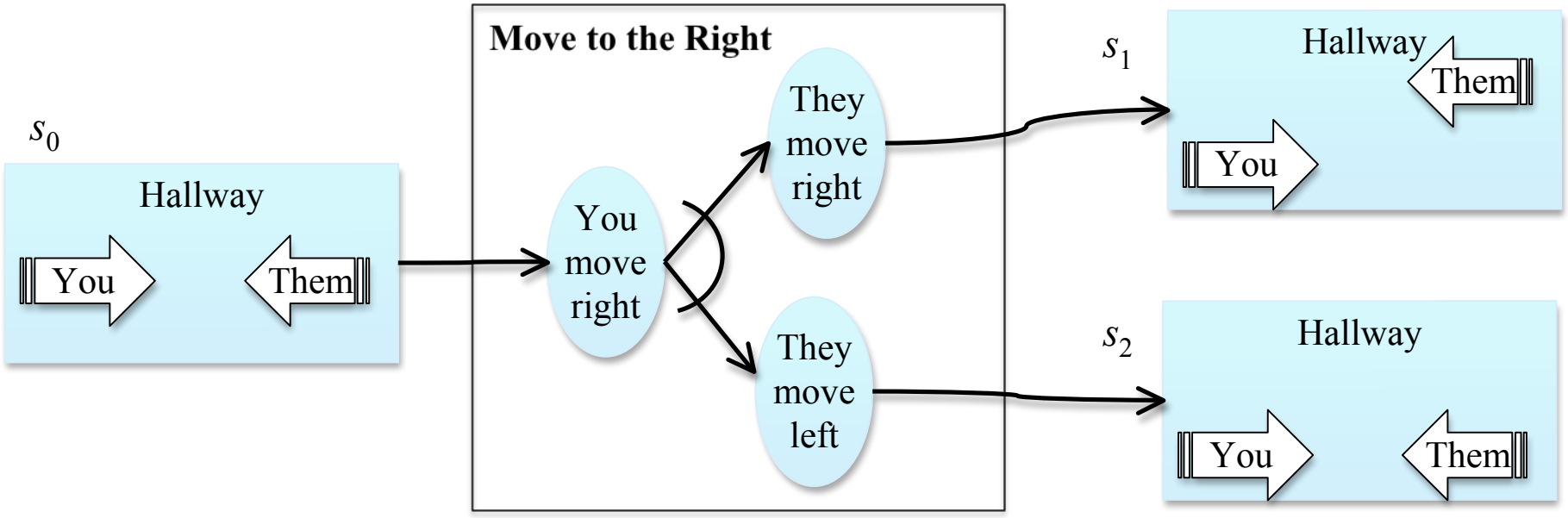
Plans

- **Classical action**:
  a partial function $a: S \rightarrow S$
  where S = {all states}
- **Classical plan**:
  a sequence of actions
  ⟨move21, take, load, move12⟩
- This requires the world to be
  - » *Static*
    - No exogenous events
  - » *Deterministic*
    - Each action has only one outcome
  - » *Fully observable*
    - We always completely know the current state
- Inadequate if there's more than one agent



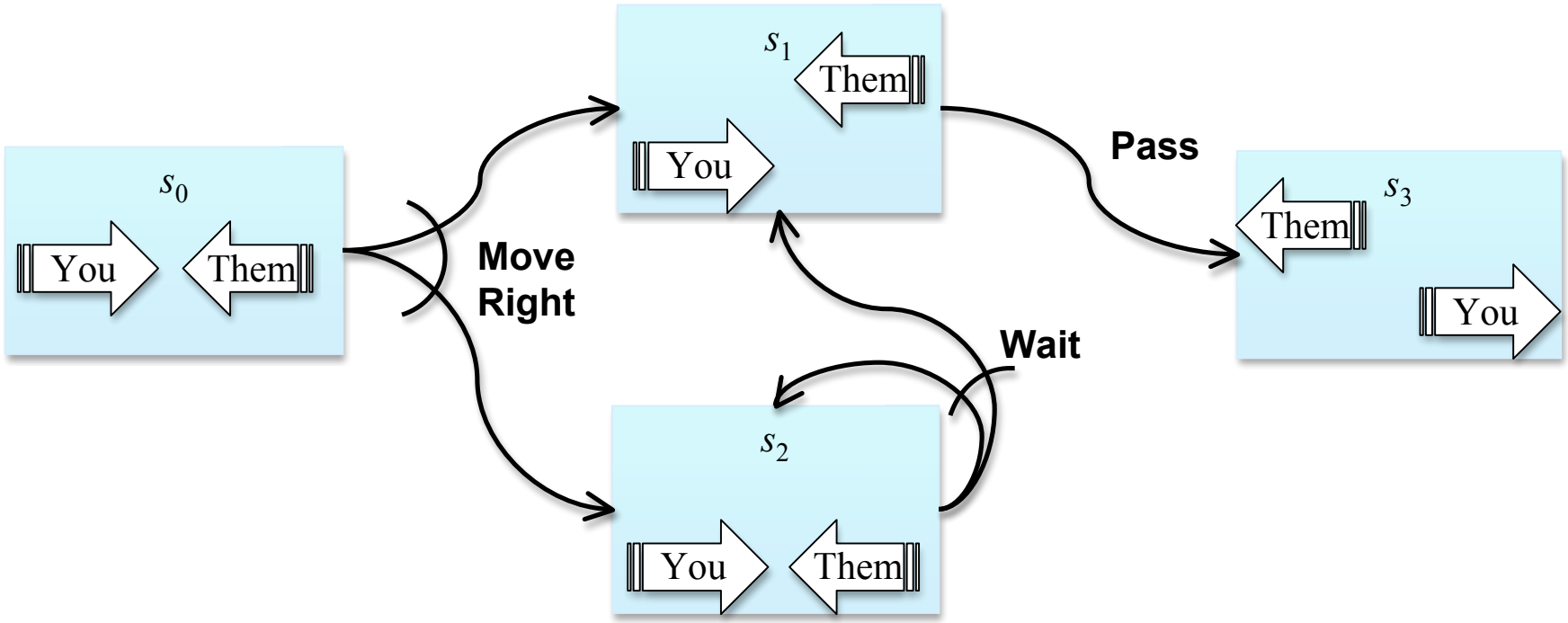
Nondeterministic Actions

- Each action is a function $a: S \rightarrow 2^S$
  - » *i.e.,* multiple possible outcomes
- Somewhat like a Markov Decision Process, but without probabilities
- To model other agents:
  - » Include the results of their actions as possible outcomes of our actions



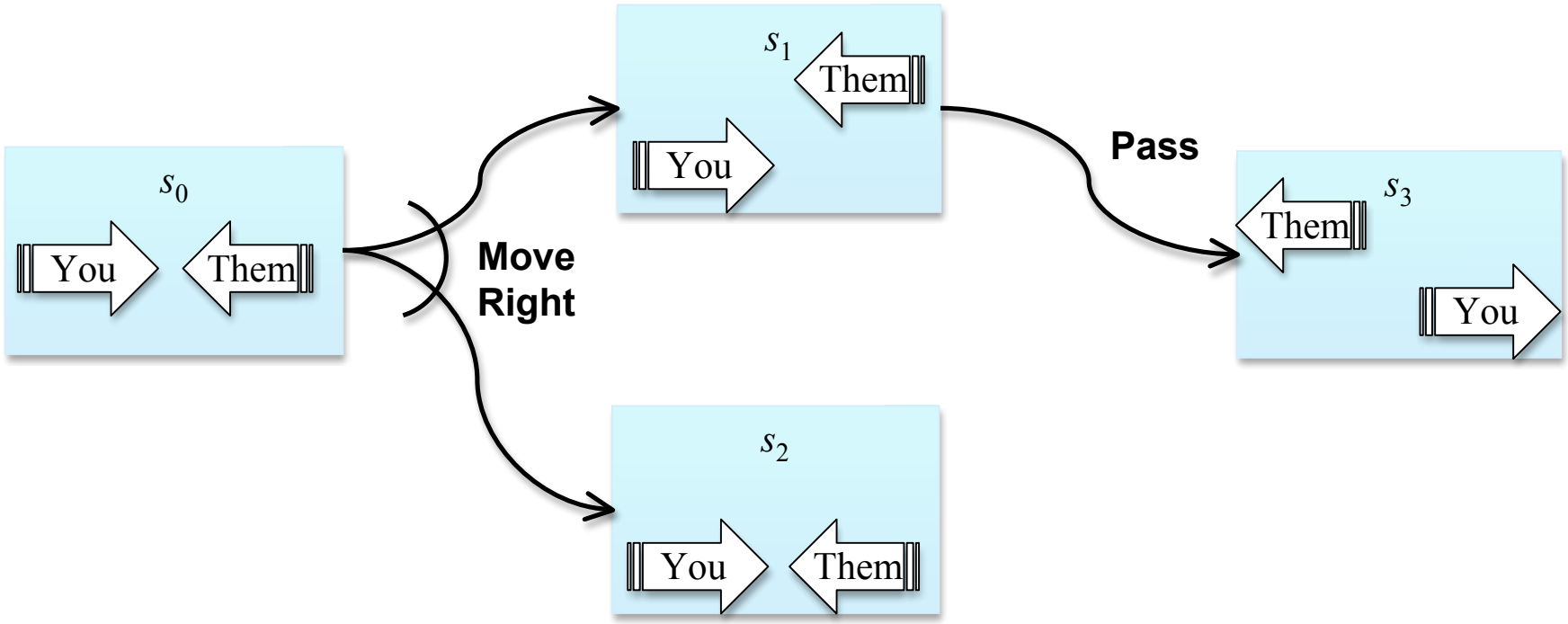
Policies and Execution Structures

- A linear plan won't work as a solution
  - » It can't encode contingencies
- A solution is a *policy*:
  a partial function $\pi: S \rightarrow A$, where A = {all actions}
  - » e.g., π = {(s_0, MoveRight), (s_1, Pass), (s_2, Wait)}
- *Execution structure*: graphical representation of a policy and its outcomes



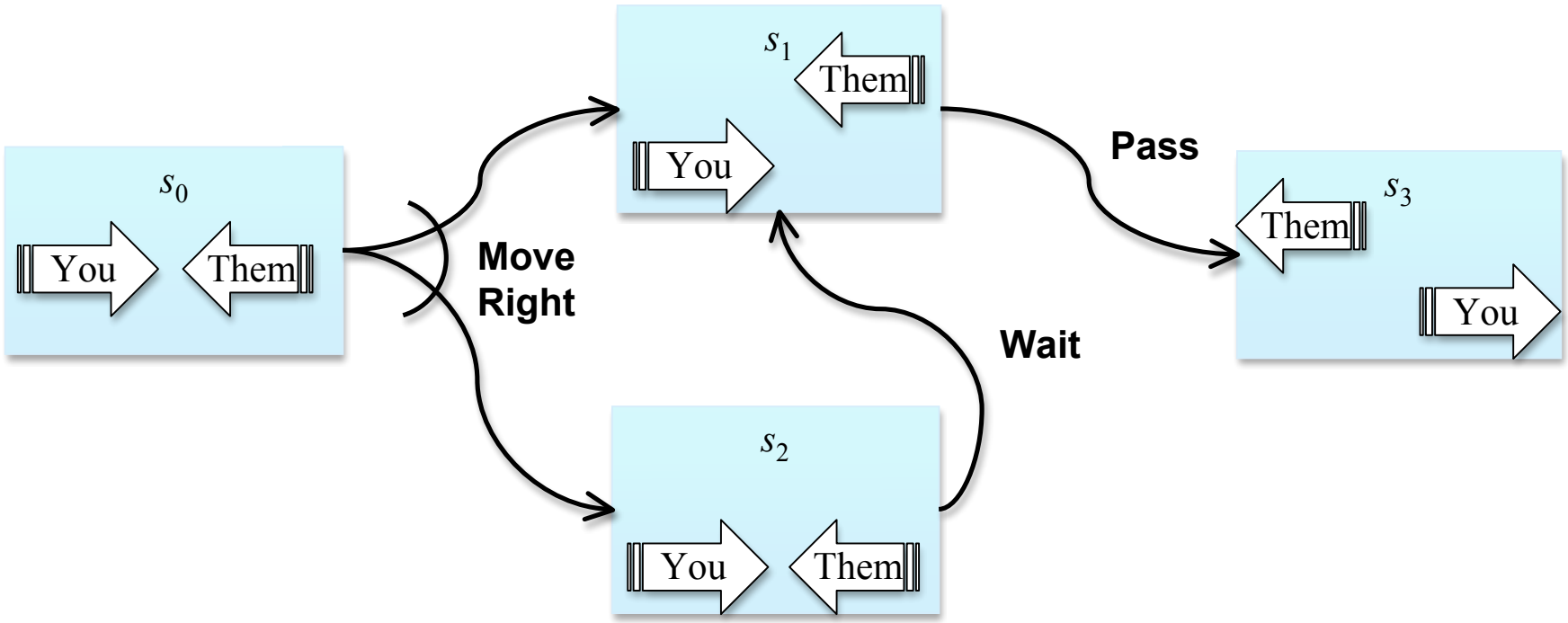
Types of Solutions

- Weak solution:
  - » At least one of the possible executions will reach a goal
- Example:
  - » $\pi = \{(s_0, \text{MoveRight}), (s_1, \text{Pass})\}$



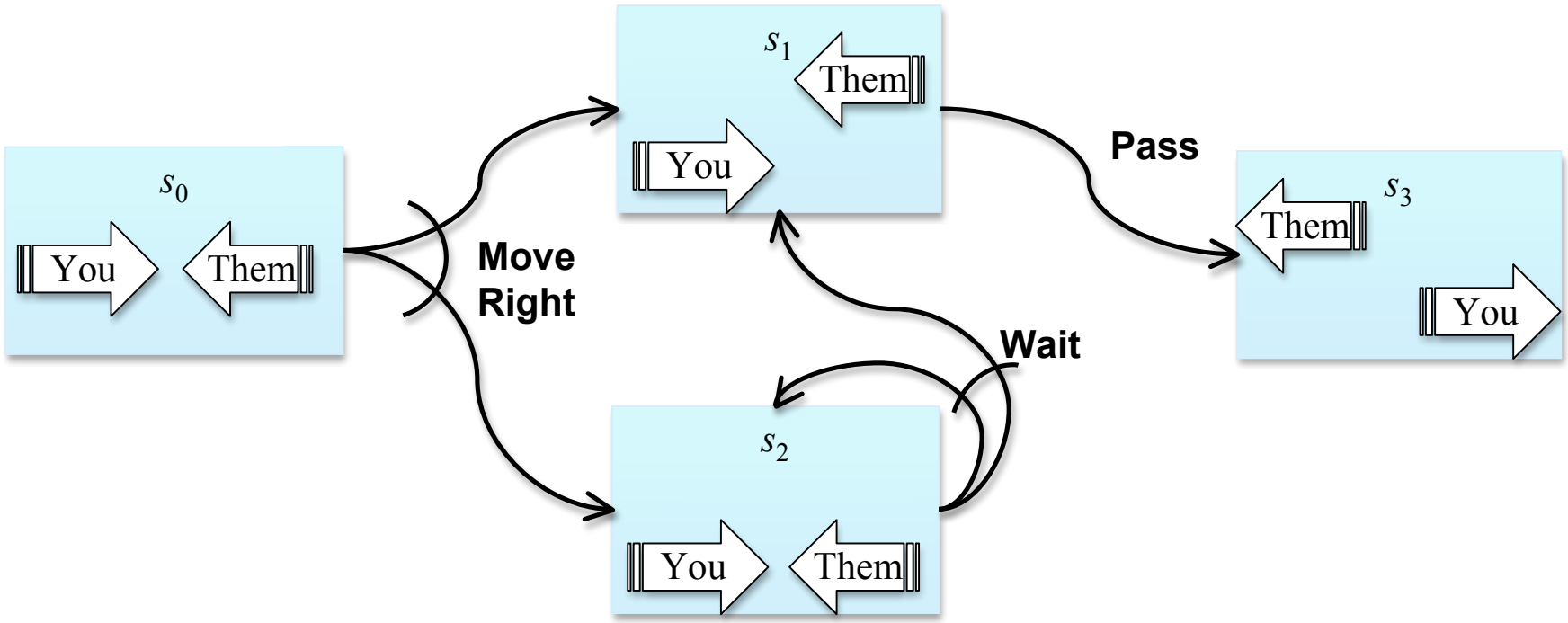
Types of Solutions

- Strong solution:
  - » Every possible execution will reach a goal
- Example:
  - » Suppose we know that if we wait, they'll *always* move right
  - » $\pi = \{(s_0, \text{MoveRight}), (s_1, \text{Pass}), (s_2, \text{Wait})\}$

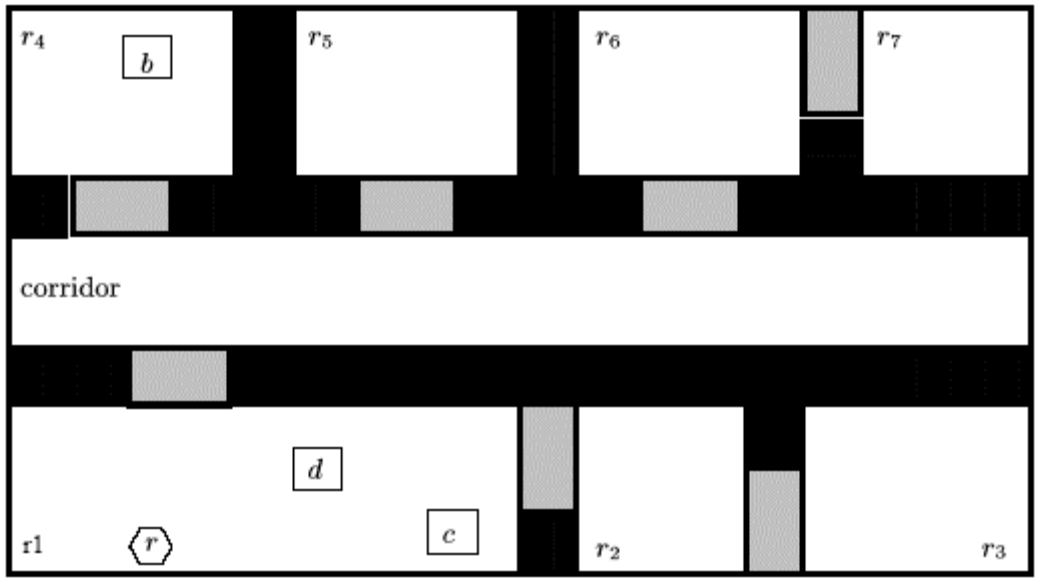


Types of Solutions

- Strong-cyclic solution:
  - » Every *fair* execution will reach a goal
  - » Works if each possible outcome of an action has nonzero probability
- Example: suppose we know that if we wait, they'll *eventually* move right
  - » $\pi = \{(s_0, \text{MoveRight}), (s_1, \text{Pass}), (s_2, \text{Wait})\}$

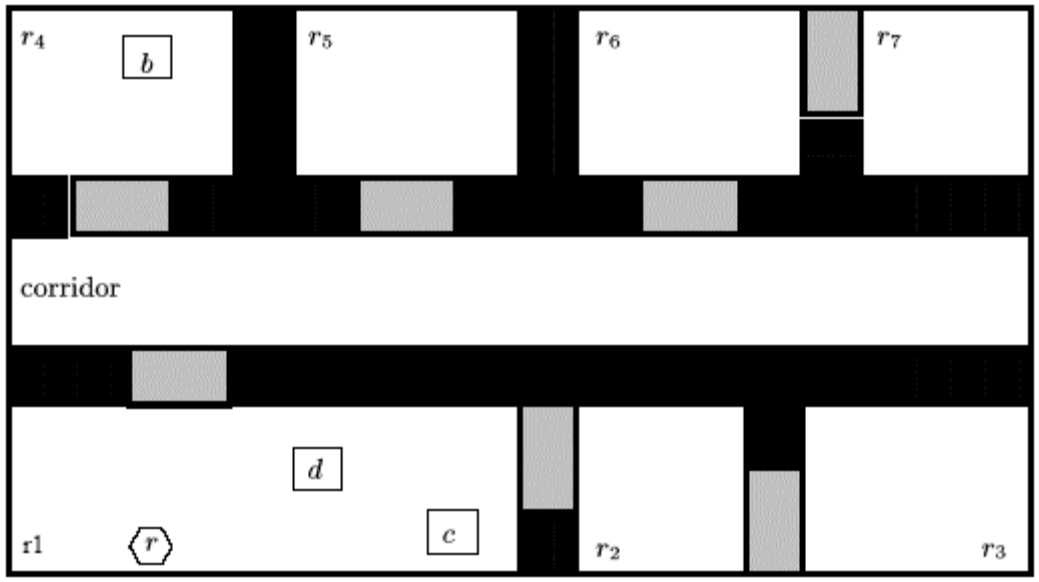


The Robot Navigation Domain



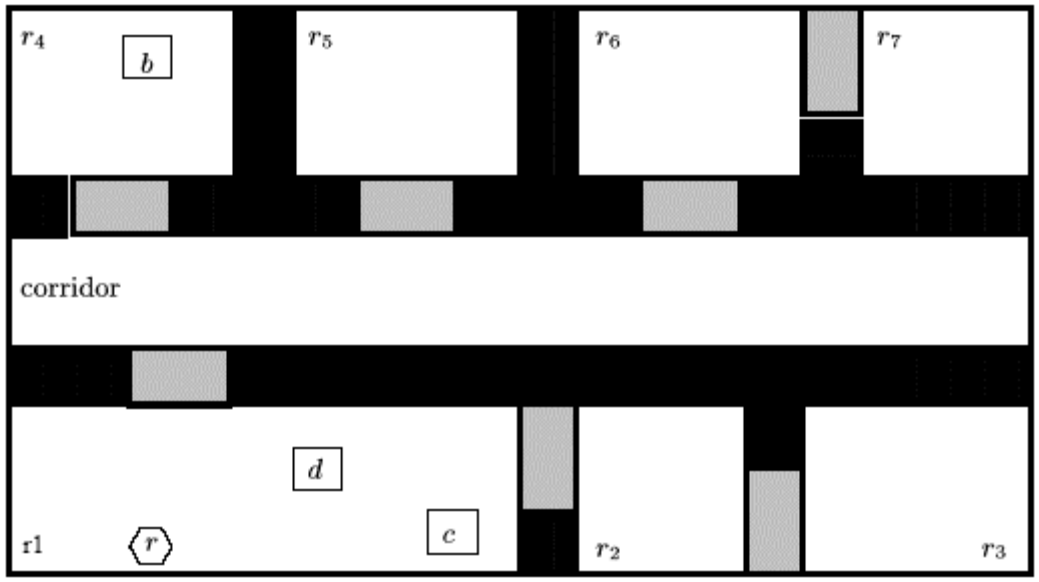
- A robot is supposed to go around a building, picking up packages and delivering them to their destinations
  - » Actions: pick up or put down a package, open or close a door, move through an open doorway
- A kid is running around, opening and shutting doors at random
  - » The kid moves much faster than the robot
  - » After each of the robot's actions, each one of the doors may be either open or closed
  - » n doors => 2^n combinations => each action has exponentially many outcomes

The Robot Navigation Domain



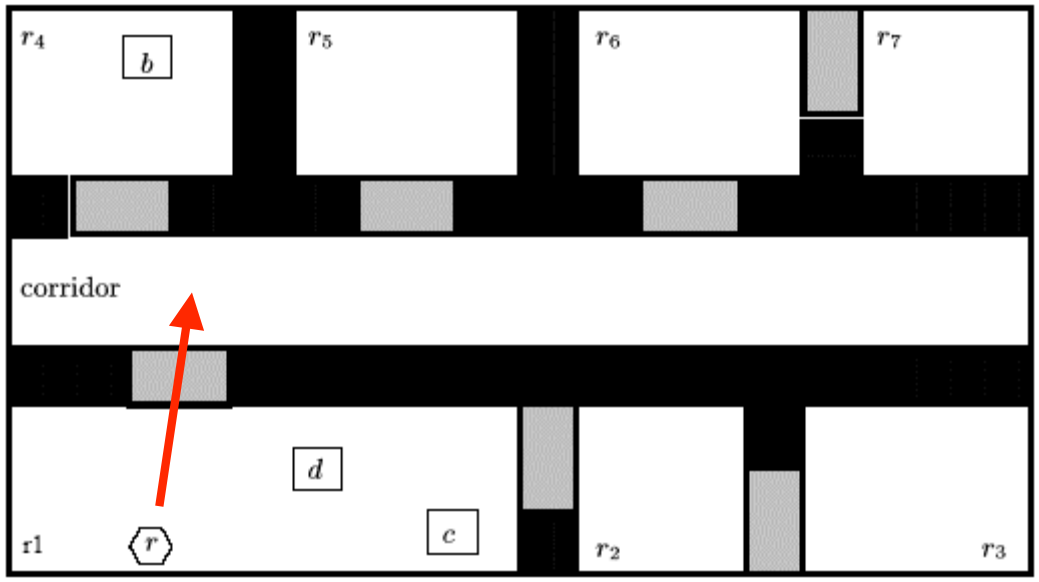
- If we represent all 2^n outcomes explicitly, then
  - The action has exponential size
  - We need to plan what to do next in 2^n different cases
- Our solutions will have doubly exponential size, and will take doubly exponential time to generate

The Robot Navigation Domain



- If we represent all 2^n outcomes explicitly, then
  - » The action has exponential size
  - » We need to plan what to do next in 2^n different cases
- Our solutions will have doubly exponential size, and will take doubly exponential time to generate
- This is not very good …

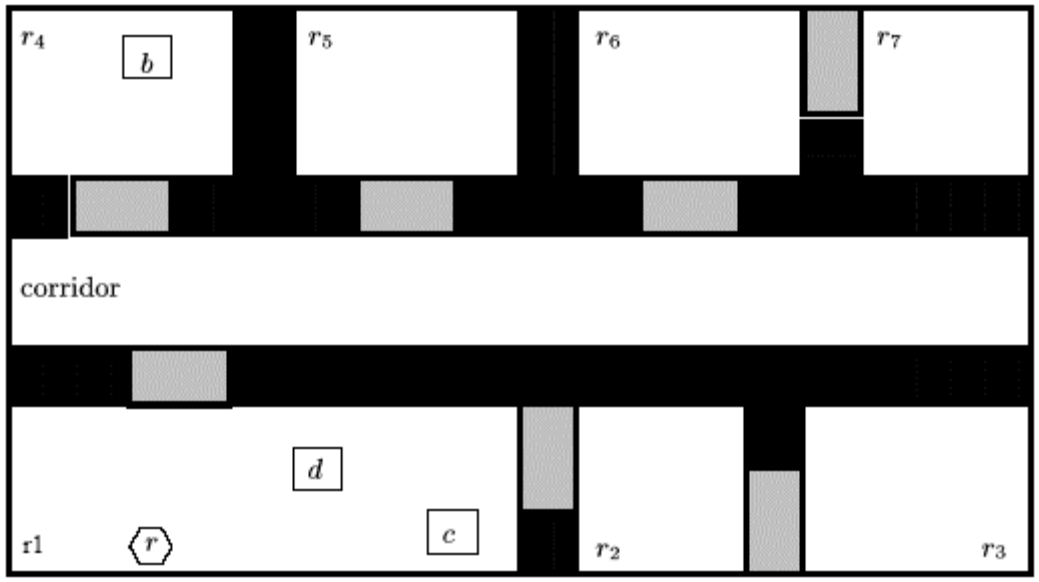
The Robot Navigation Domain



- If we want the robot to go through door 1
  - » We care whether door 1 is open,
    but we don't care whether the other n–1 doors are open
  - » Those doors are irrelevant to almost all of our applicable actions
- We'd like to plan for two sets of states:
  - » The states in which door 1 is open,
    and the states in which it's closed

- Need a way to represent sets of states

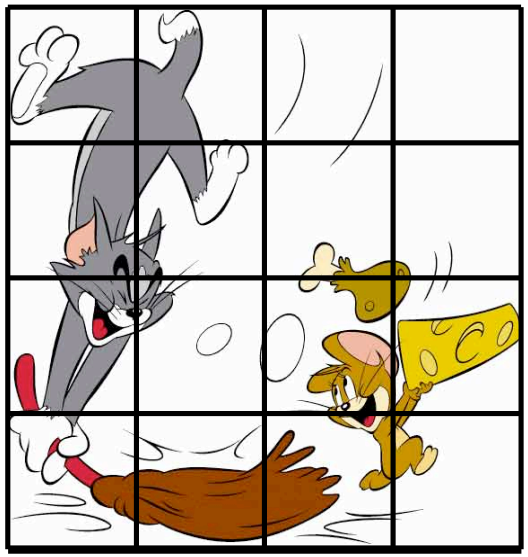
Representing Sets of States



- Binary Decision Diagrams (BDDs) (Bryant, 92)
  - » Propositional formulas to encode sets of states
  - » Techniques to transform one formula into another to model state transitions
- The MBP planner (Traverso *et al.*, 2001)
  - » Backward search from the goal
  - » Each node in the search space is a BDD for a set of states
- This avoids the problems with combinatorial explosion in the Robot Navigation Domain
  - » MBP can easily solve Robot Navigation problems

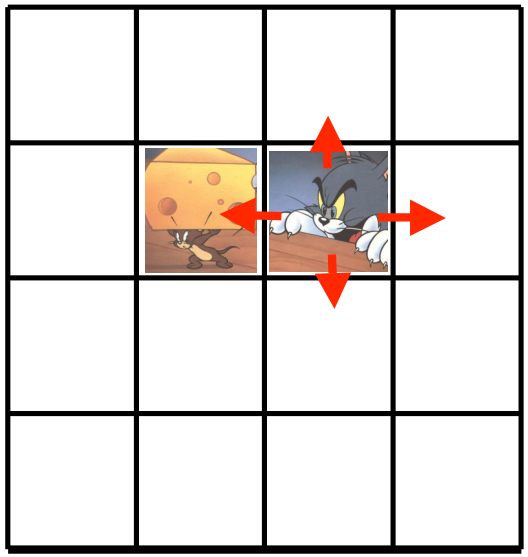
An Example

- Hunter-Prey Problems
  - A hunter and k prey, on an n x n grid
- Hunter's possible actions: *N, S, E, W, Catch*
  - *Catch* is applicable only when the hunter and a prey are at the same location
- Each prey's possible actions: *N, S, E, W, Wait*
  - As before, represent these as nondeterministic outcomes of the hunter's actions
  - As before, there is a combinatorial blowup
    - About $(n^2+1)^{k+1}$ possible states
    - An action by the hunter may have up to 5^k possible outcomes



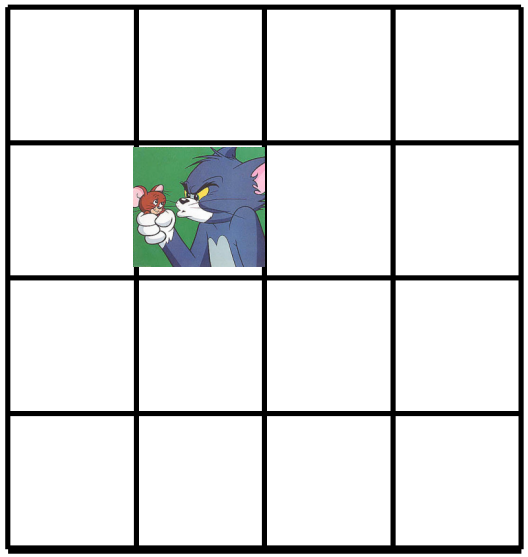
An Example

- Hunter-Prey Problems
  - » A hunter and k prey, on an n x n grid
- Hunter's possible actions: *N, S, E, W, Catch*
  - » *Catch* is applicable only when the hunter and a prey are at the same location
- Each prey's possible actions: *N, S, E, W, Wait*
  - » As before, represent these as nondeterministic outcomes of the hunter's actions
  - » As before, there is a combinatorial blowup
    - About $(n^2+1)^{k+1}$ possible states
    - An action by the hunter may have up to 5^k possible outcomes



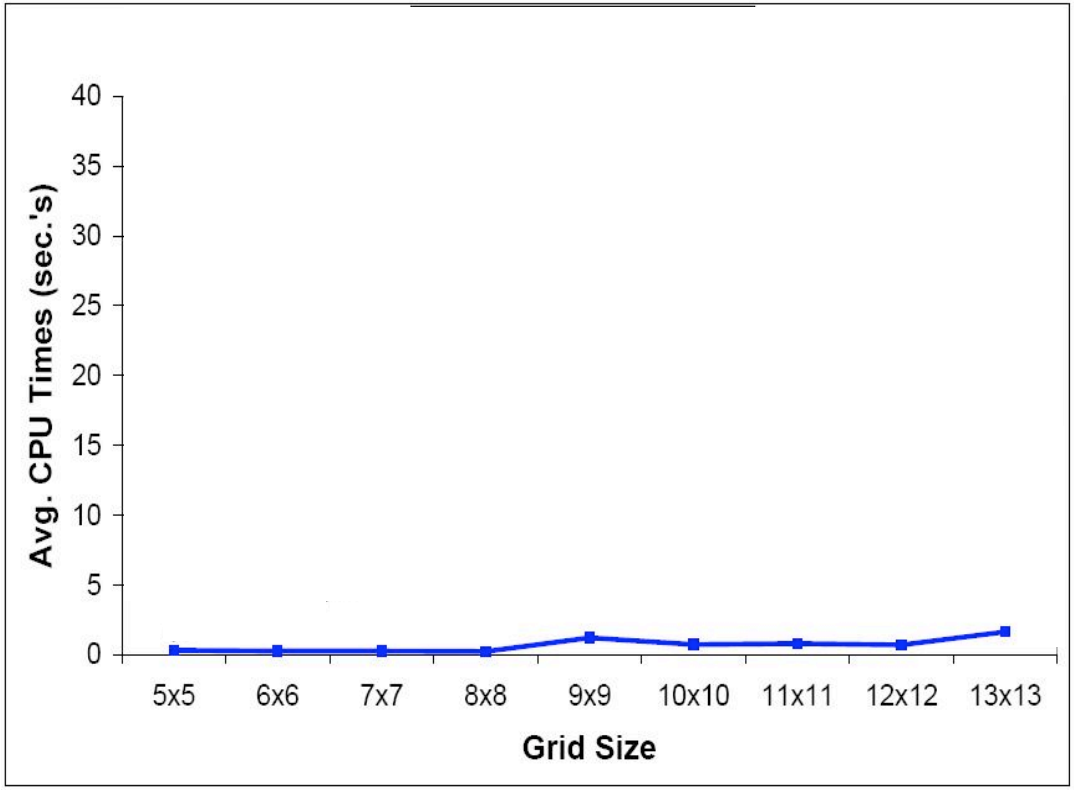
An Example

- Hunter-Prey Problems
  - » A hunter and k prey, on an n x n grid
- Hunter's possible actions: *N, S, E, W, Catch*
  - » *Catch* is applicable only when the hunter and a prey are at the same location
- Each prey's possible actions: *N, S, E, W, Wait*
  - » As before, represent these as nondeterministic outcomes of the hunter's actions
  - » As before, there is a combinatorial blowup
    - About $(n^2+1)^{k+1}$ possible states
    - An action by the hunter may have up to 5^k possible outcomes

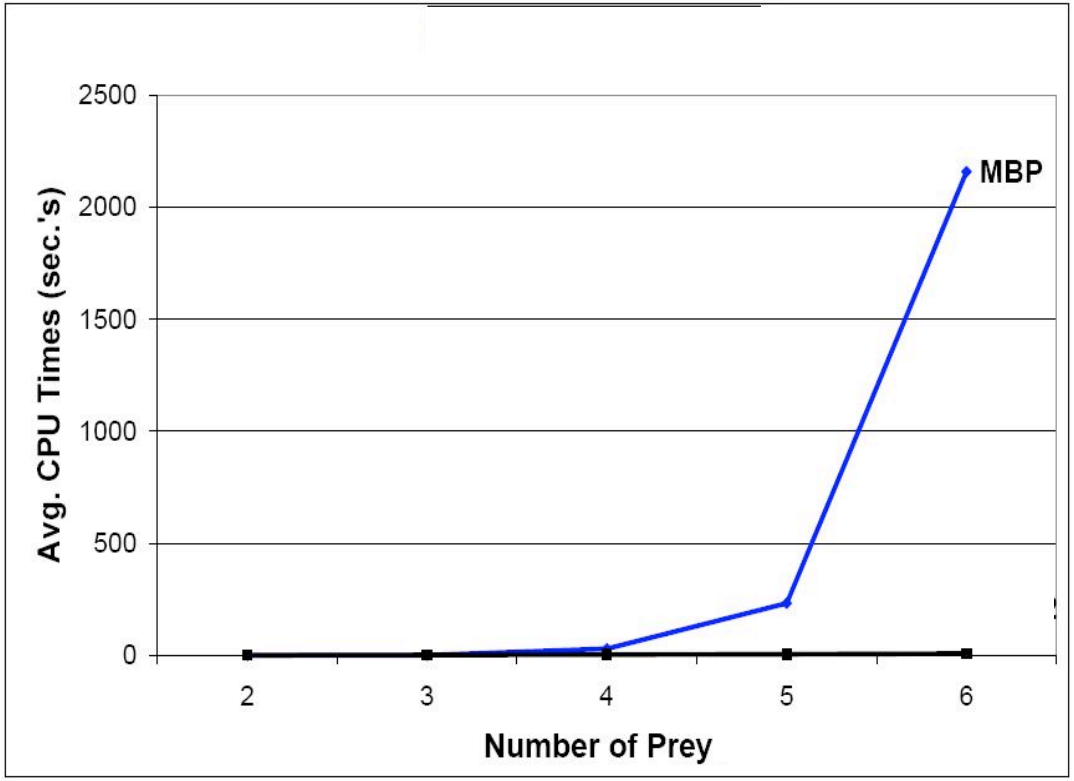


MBP Can "Abstract Away" Information that it *Knows* is Irrelevant

- MBP does very well with one prey, regardless of grid size
  - » It partitions the states based on which actions are relevant
- Example
  - » One prey, 2x2 grid
  - » {states where the hunter is in the upper right corner and the prey isn't}
  - » hunter-at-upper-right & ¬ prey-at-upper-right
- In all of these states, there are two possible actions:
  - » move-down and move-left



MBP Has Attention Deficit Disorder



- MBP does badly with multiple prey, regardless of grid size
- k prey => hunter's actions may each have up to 5^k possible outcomes
- At each point in its search, MBP tries to plan for all 5^k of them
  - » Each state has a different set of applicable actions
  - » Each applicable action is relevant for catching one of the prey
  - » It can't ignore any of them, because it wants to catch all the prey
- Runtime exponential in the number of prey
  - » e.g., more than 35 CPU minutes for 6 prey on a 4x4 grid

Domain Knowledge

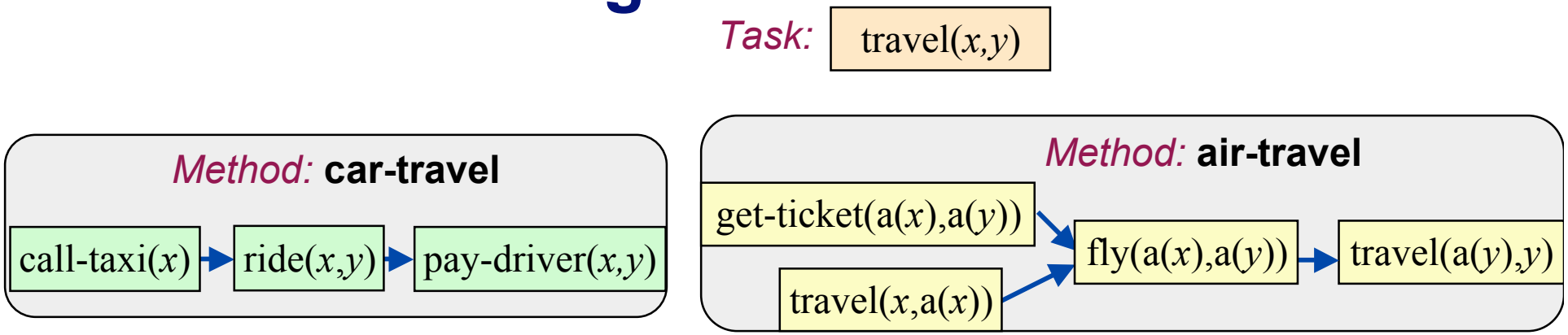
- To generate a policy for the Hunter-Prey domain, it's best to focus on catching one prey at a time
  - First chase prey #1, then prey #2, then prey #3, …
  - This is an example of *domain knowledge*
    - Problem-solving information that's specific to a particular domain

- AI Planning researchers don't like to use domain knowledge
  - They prefer to work on planning algorithms that are "domain-independent"
  - They feel that using domain knowledge is "cheating"

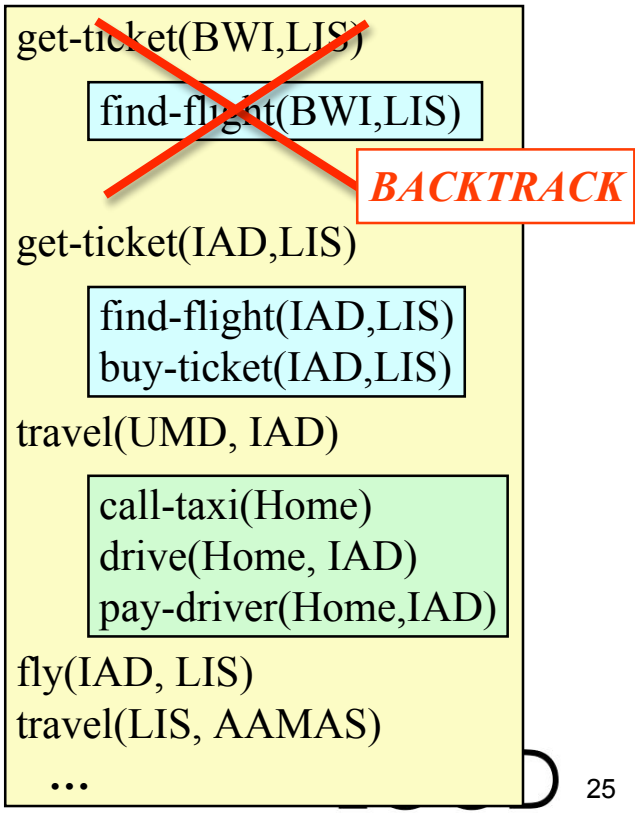
Domain Knowledge

- Researchers in other fields have trouble comprehending why we wouldn't want to use domain knowledge
  - » Operations research, control theory, engineering, …
  - » Why would anyone *not* want to use the knowledge they have about a problem they're trying to solve?
- Furthermore, the very notion of domain independence is illusory
  - » Classical AI planning entails a very restrictive set of restrictive assumptions
  - » Very few real-world planning problems can satisfy these assumptions
- If you want to write planners that scale up to real-world problem solving, domain knowledge is essential
  - » Need a way to express it, and planning engines that can use it

HTN Planning



- Problem reduction
  - » *Tasks* (activities) rather than goals
  - » *Methods* with *constraints* and *subtasks*
    - Method *m* is applicable to task *t* in state *s* if *t* and *s* satisfy *m*'s constraints
    - Decompose *t* into the *subtasks*
  - » Keep applying methods until you reach *primitive* tasks that you can accomplish directly
- Noah, Nonlin, O-Plan, SIPE, SIPE-2,SHOP, SHOP2
- Lots of real-world applications



SHOP2

- SHOP2
  - » My group's HTN planning system
  - » Won award at the 2002 International Planning Competition
  - » Freeware, open source
    - `http://www.cs.umd.edu/projects/shop`
  - » Nearly 10,000 downloads
  - » Used in hundreds of projects worldwide

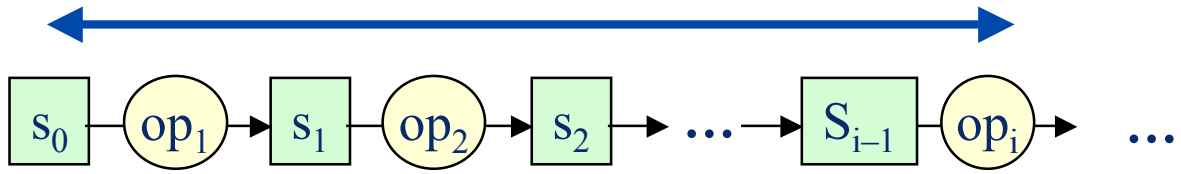
http://www.cs.umd.edu/~nau/publications

Nau, Au, Ilghami, Kuter, Murdock, Wu, & Yaman.
SHOP2: an HTN Planning System.
JAIR, Dec 2003.

Nau, Au, Ilghami, Kuter, Muñoz-Avila, Murdock, Wu, & Yaman.
Applications of SHOP and SHOP2.
IEEE Intelligent Systems, 2005.

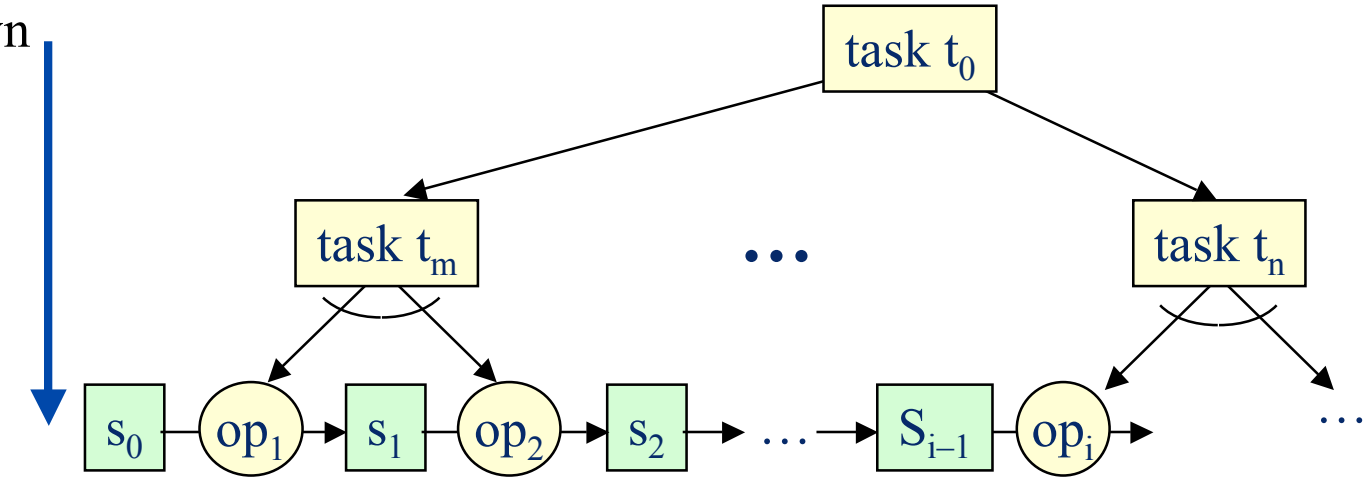
Comparison to Forward and Backward Search

- In state-space planning, must choose whether to search forward or backward



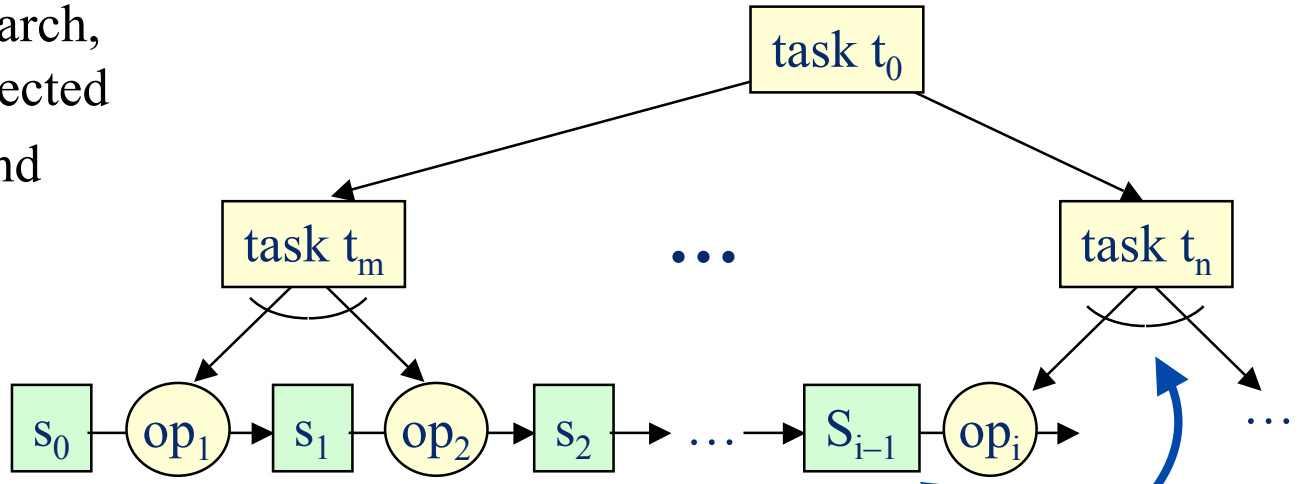
- In HTN planning, there are *two* choices to make about direction:
  - forward or backward
  - up or down

- SHOP2 goes *down* and *forward*



Comparison to Forward and Backward Search

- Like a backward search, SHOP2 is goal-directed
  - Goals correspond to tasks



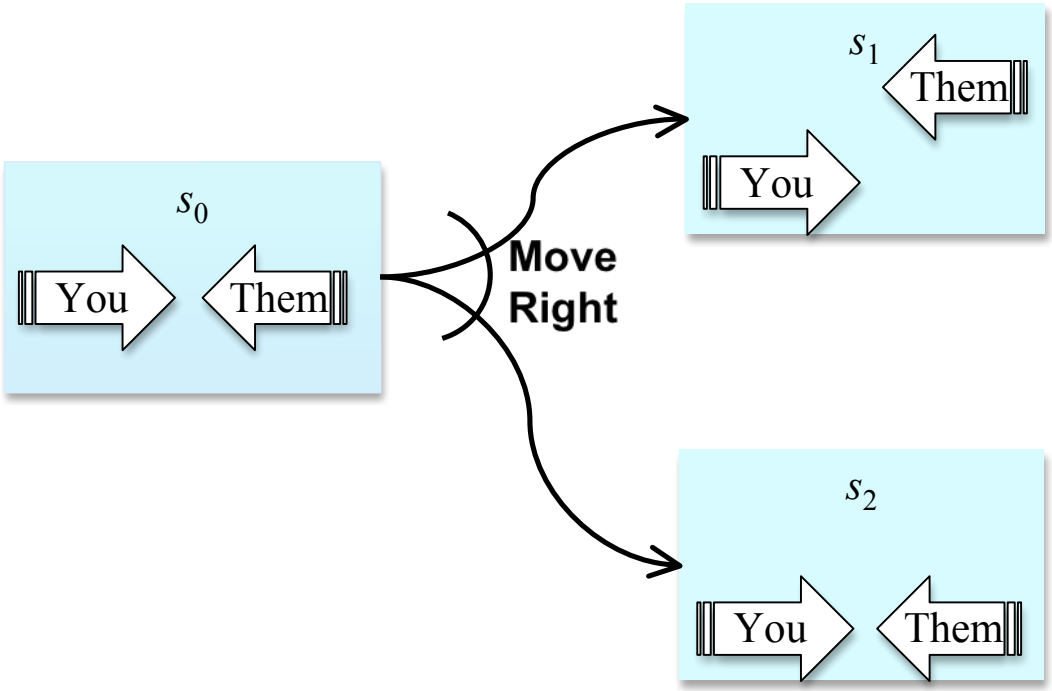
- Like a forward search, SHOP2 generates actions in the same order in which they'll be executed
- Whenever we want to plan the next task
  - We've already planned everything that comes before it
  - Thus we know the current state of the world
- This makes it much easier to infer whether complicated constraints are satisfied

Generalizing SHOP2 to handle Nondeterminism

- SHOP2 generates plans
- First, modify it to generate policies
  - » SHOP2's forward search becomes a search from the leaf nodes of the current policy's execution structure

http://www.cs.umd.edu/~nau/publications

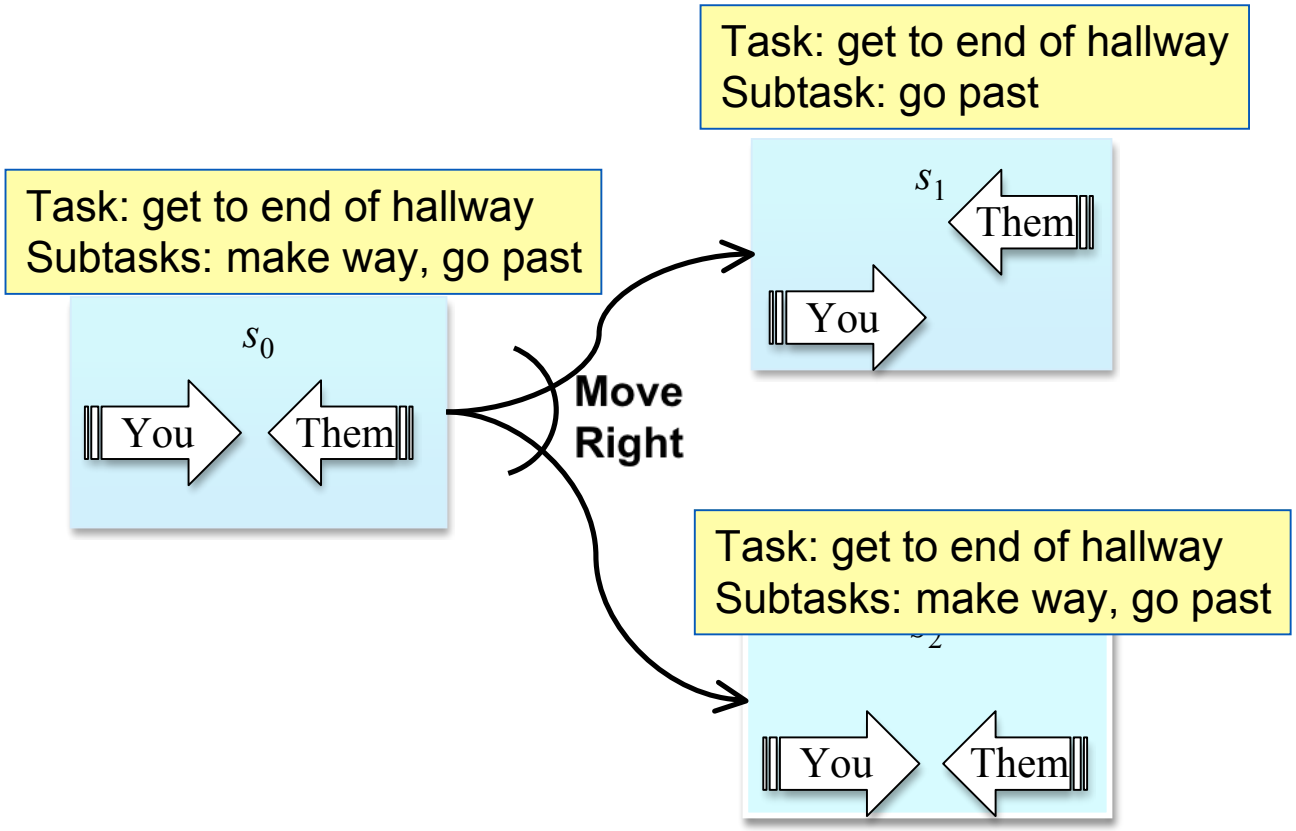
U. Kuter and D. Nau. Forward-chaining planning in nondeterministic domains. *AAAI*, 2004



Generalizing SHOP2 to handle Nondeterminism

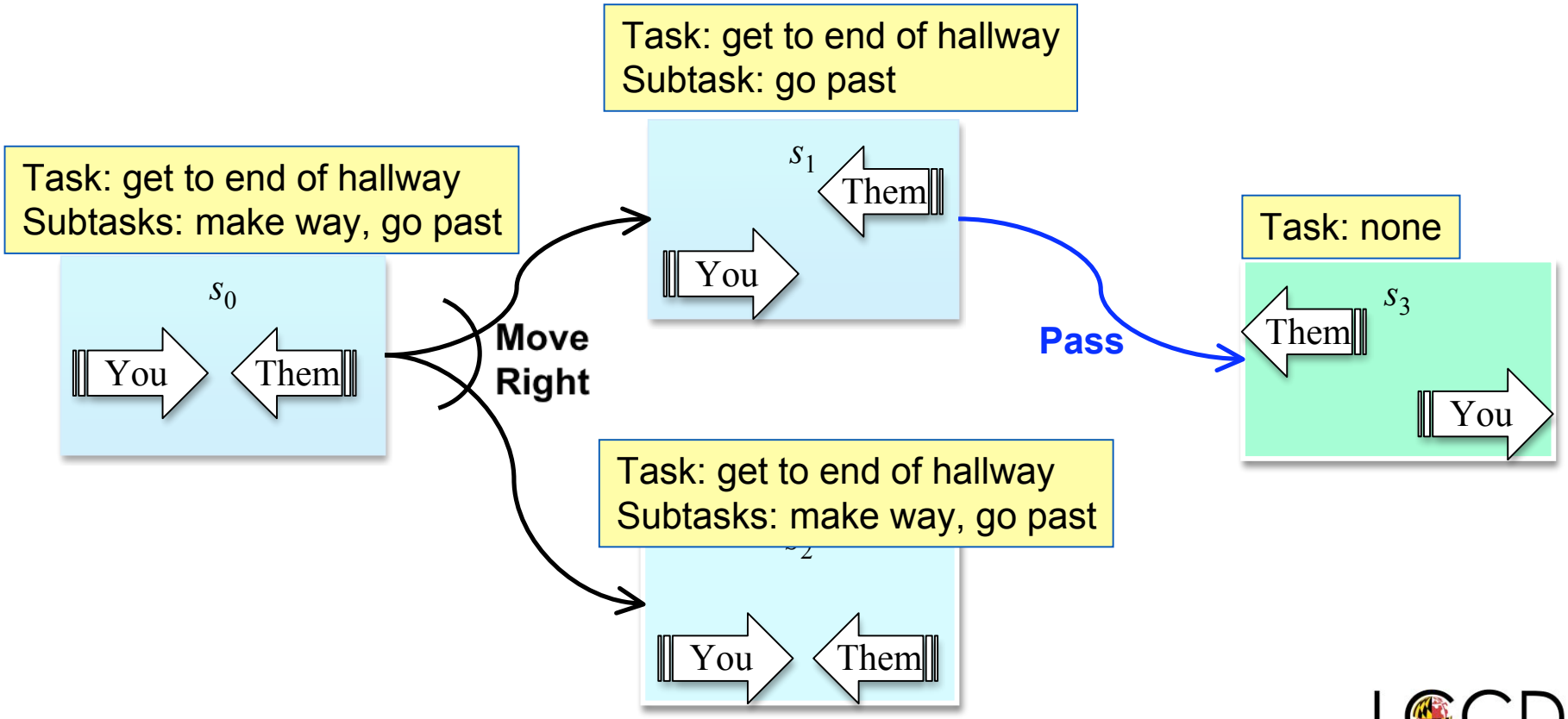
- An HTN is associated with each node of the execution structure
  - » It tells what task(s) need to be accomplished at that node

…



Generalizing SHOP2 to handle Nondeterminism

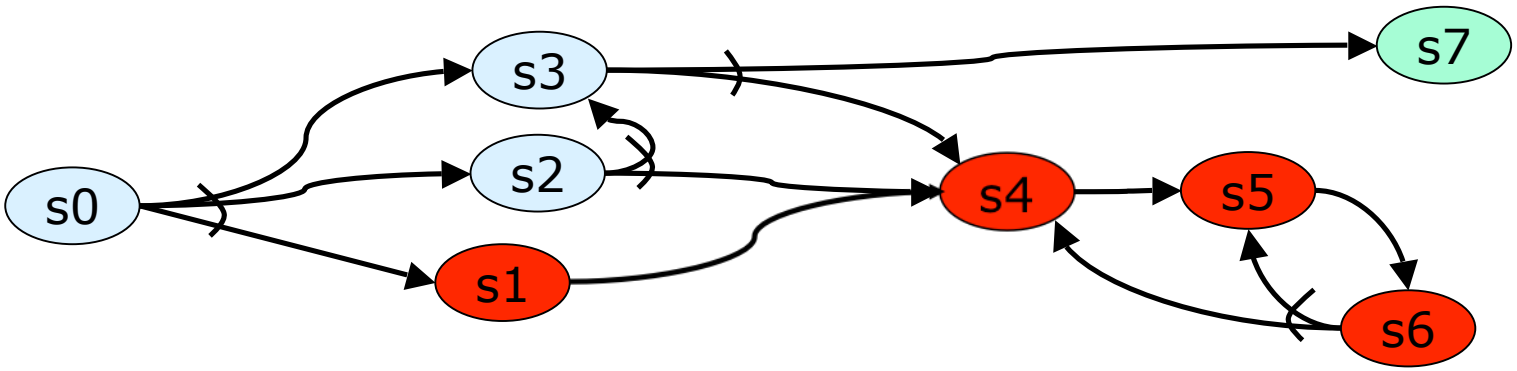
- An HTN is associated with each node of the execution structure
  - » It tells what task(s) need to be accomplished at that node
- Until all leaf nodes are goal nodes, do:
  - select a leaf node, use HTN decomposition to generate an action for it



Checking Cyclic Policies

- Unacceptable Cyclic Executions
  - » Executions with no possibility of achieving goals
- *π-descendancy* check for detecting unacceptable cycles
  - » For each node, check whether there's a possible transition to a node that either is solved or has unexplored descendants

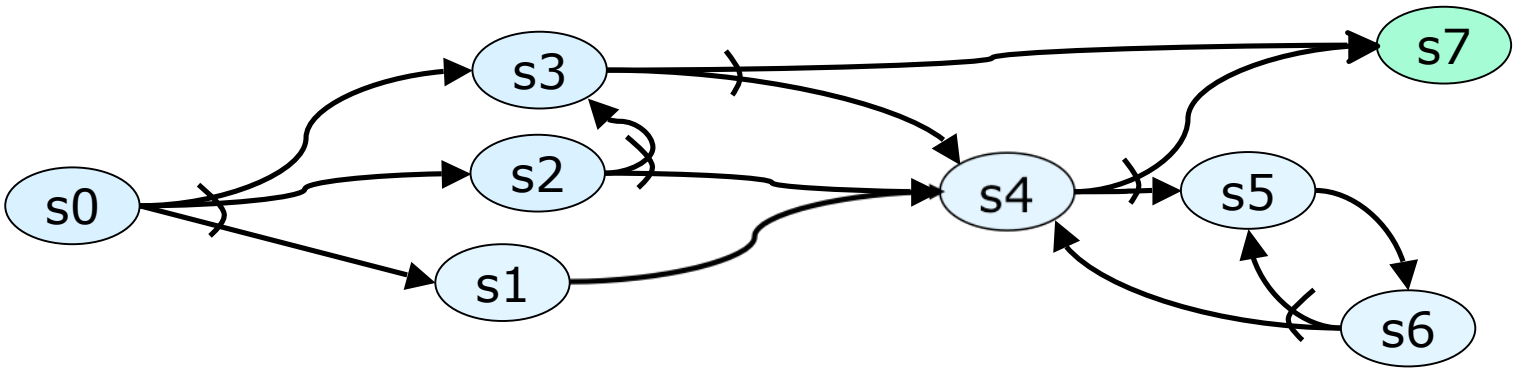
- Example of a policy that **fails** the π-descendancy check:



Checking Cyclic Policies

- Unacceptable Cyclic Executions
  - » Executions with no possibility of achieving goals
- *π-descendancy* check for detecting unacceptable cycles
  - » For each node, check whether there's a possible transition to a node that either is solved or has unexplored descendants

- Example of a policy that **passes** the *π*-descendancy check:



Pseudocode

- ND-SHOP2
  - » Incorporate nondeterministic outcomes and π-descendancy checks
- The same can be done to any other forward-search planner

Handling Open States

loop
  if $S = \emptyset$ then return(π)
  select a state $s \in S$ and remove it from S
  if s satisfies g then insert s into $solved$
  else if $s \notin S_\pi$ then

SHOP2 ***or any other*** forward-search planner

    $actions \leftarrow \{a \mid a$ is applicable to s
                and acceptable(s, a, x, D) holds$\}$
    if $actions = \emptyset$ then return$(failure)$
    nondeterministically choose $a \in actions$
    $s' \leftarrow result(s, a)$
    $\pi' \leftarrow append(\pi, a)$
    $x' \leftarrow$ progress(s, a, x, D)

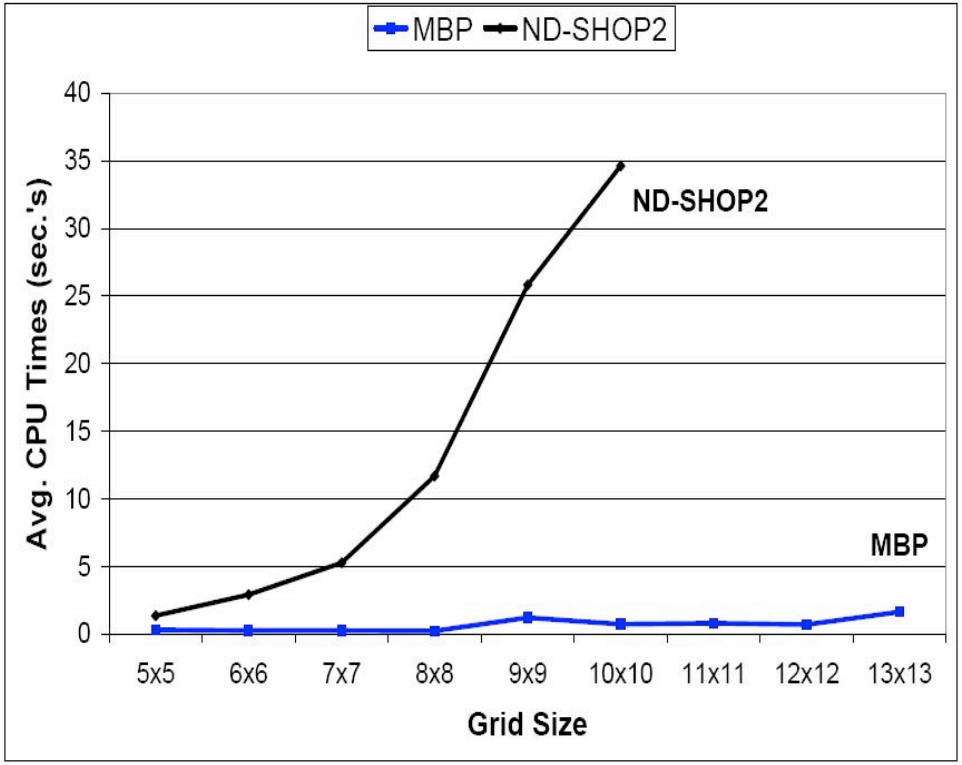
Cycle Checks

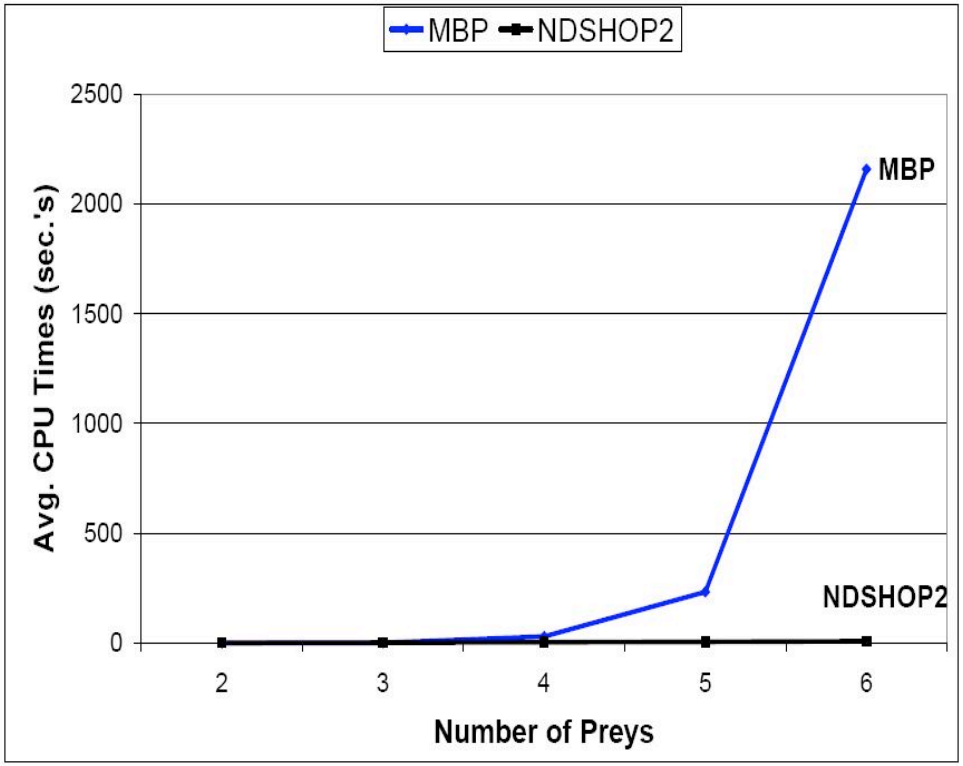
  else if s has no π-descendants in $(S \cup solved) \setminus S_\pi$
    then return$(failure)$

HTNs for the Hunter-Prey Domain

- For the Hunter-Prey domain, the HTN domain knowledge that we gave to ND-SHOP2 was quite simple:
  - To catch all uncaught prey
    - If there's at least one uncaught prey, then
      - Select a prey *p*
      - Chase *p* until you've caught it
      - Catch all uncaught prey
  - To catch *p*:
    - If we're at *p*'s location, then grab *p*
    - Otherwise
      - Move toward *p*
      - Chase *p* until you've caught it

MBP versus ND-SHOP2





One prey on a large grid

- MBP does much better than ND-SHOP2
- MBP can reason about large sets of states, ND-SHOP2 can't

Many prey on a small grid

- ND-SHOP2 does much better than MBP
- ND-SHOP2 can focus on one prey at a time, MBP can't

Can we combine the advantages of both?

Yoyo: HTN Decomposition on BDDs

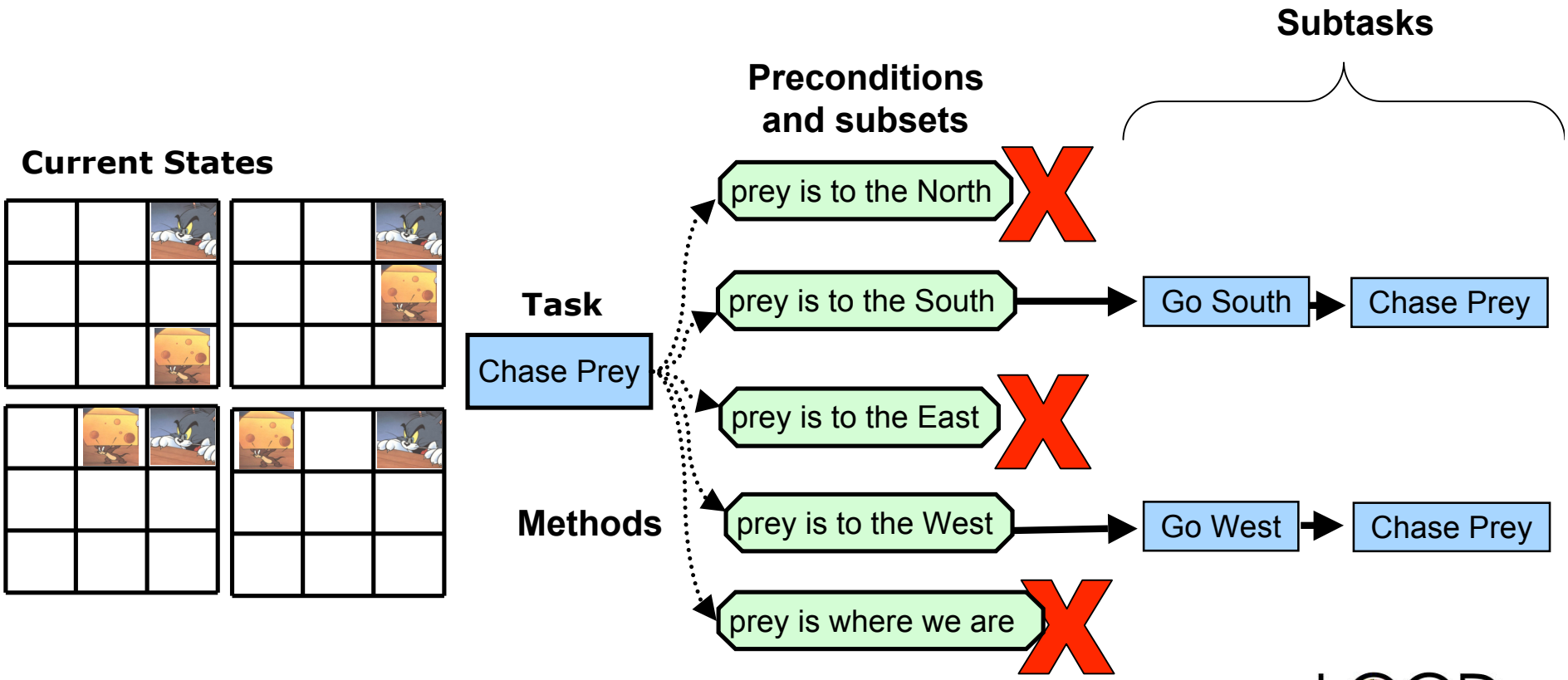
- Yoyo combines HTN-based search control with BDD-based state representations
  - » Like MBP, it generates policies over abstract classes of states
  - » Like ND-SHOP2, it generates these policies using HTN decomposition

http://www.cs.umd.edu/~nau/publications

Kuter, Nau, Pistore, and Traverso.
A hierarchical task-network planner based on symbolic model checking.
ICAPS, June 2005.

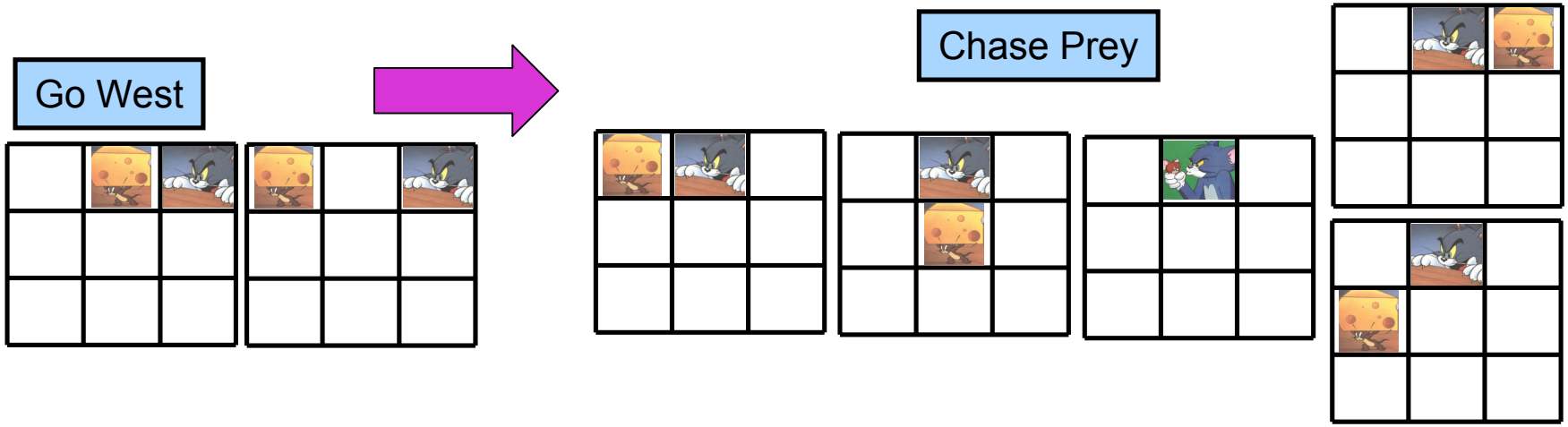
Generating Actions in Yoyo

- Given a task T, and a BDD B that represents a set S of states in which T needs to be accomplished
  - » Find methods $m_1, \ldots, m_k$ that are applicable to subsets of S
  - » Partition S into a new set of BDDs $B_1, B_2, \ldots, B_k$ such that each B_i satisfies m_i's preconditions



Generating successor states

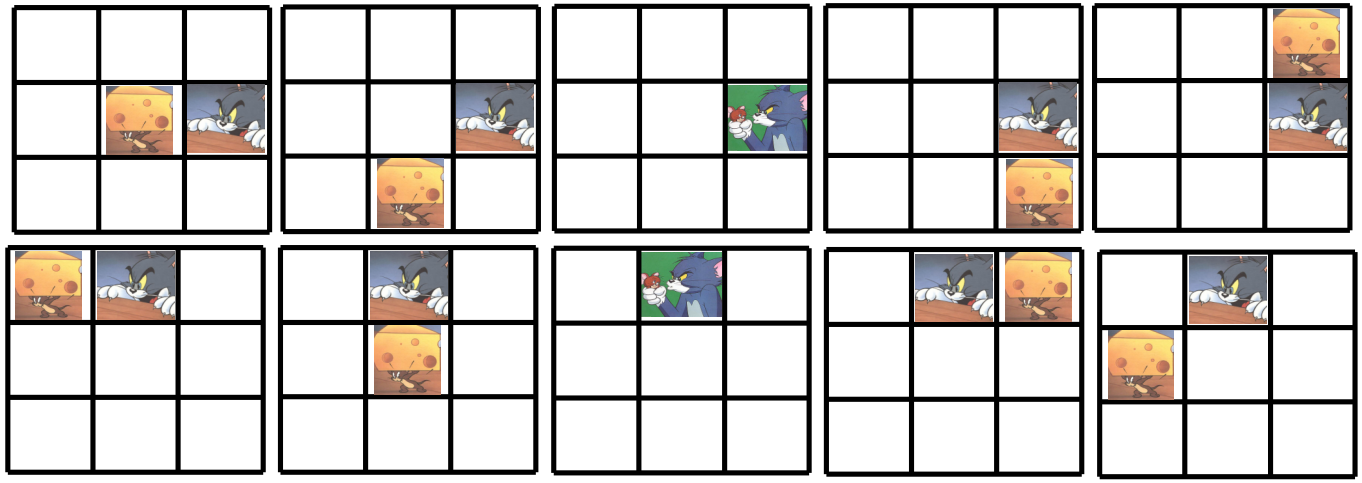
- Apply the actions to get sets of successor states represented as BDDs $B'_1, B'_2, \ldots, B'_k$



Compose Successor States into Larger Sets

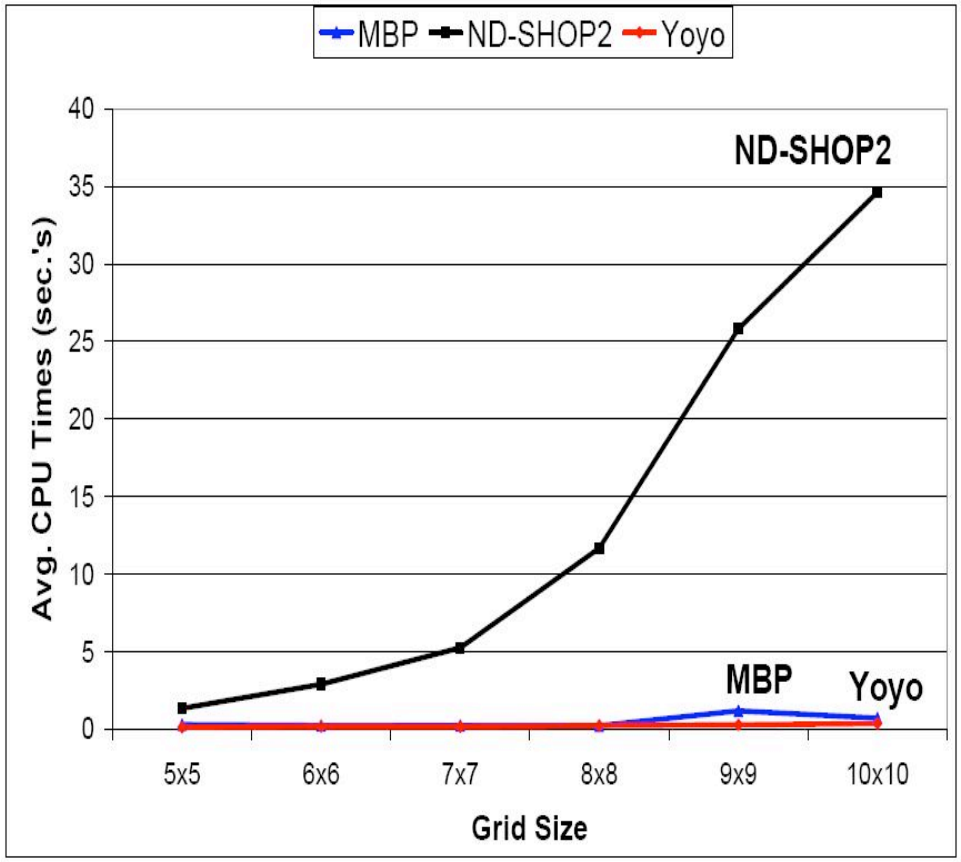
- For each set of BDDs $B'_1, B'_2, \ldots, B'_k$ that have the same task network, replace them by the BDD $B'_1 \vee B'_2 \vee \ldots \vee B'_k$
  - » In the example, this produces a single set of states

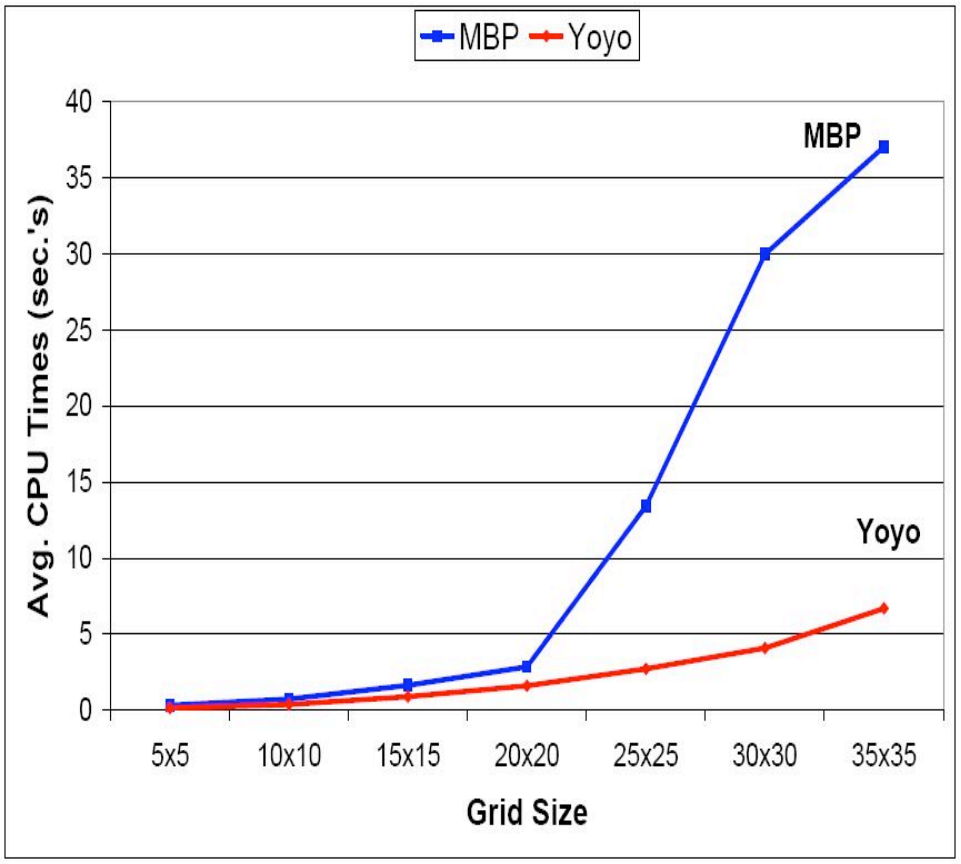
Next States:



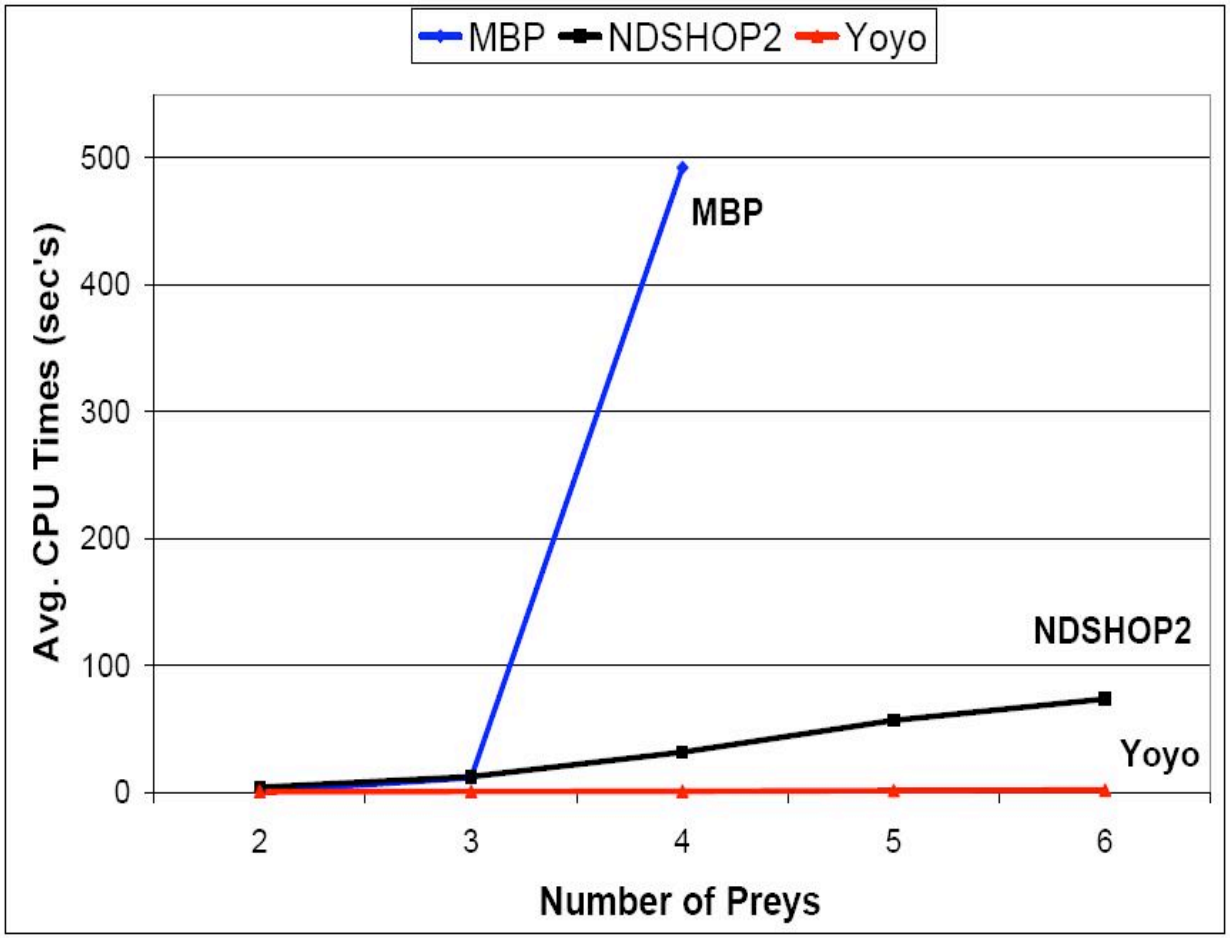
Next Task: Chase Prey

Varying Grid Size, with one prey





Multiple Prey on a Fixed Grid



Utilities in Multi-Agent Environments

- So far, I've talked about finding policies having the following properties:
  - » They reach a goal state
    - All the packages are at their destinations
    - There are no uncaught prey
  - » They're guaranteed to work regardless of how the other agents behave

- In some cases, we may instead want to find plans that optimize a utility function
    - We're doing this in a multi-agent environment
    - The other agents may also have their own utility functions
  - » This is a non-zero-sum game

Games ⟷ Multi-agent planning problems with utilities

Strategies ⟷ Policies

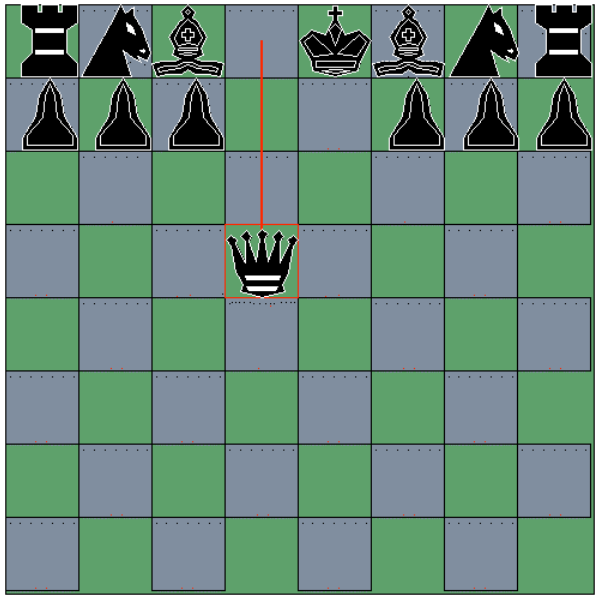
Opponent Modeling

- Mathematical game theory normally assumes the other agents will maximize their expected utilities
- *Nash equilibrium* strategy: mathematical notion of an optimal strategy
  - » **Limitation:** a Nash-equilibrium strategy is optimal only against an opponent who plays *their* Nash-equilibrium strategy
- If the other agents have limited reasoning power, they probably aren't using the Nash equilibrium strategy
  - » Huge literature on *behavioral economics*
    - Originated by [Thaler, 1987], continued by many others
    - Many cases in which humans (or aggregations of humans) tend to make different decisions than the game-theoretically optimal ones
- If the opponent isn't playing the Nash equilibrium strategy, you can do better if you don't use it either

Playing Non-Equilibrium Strategies

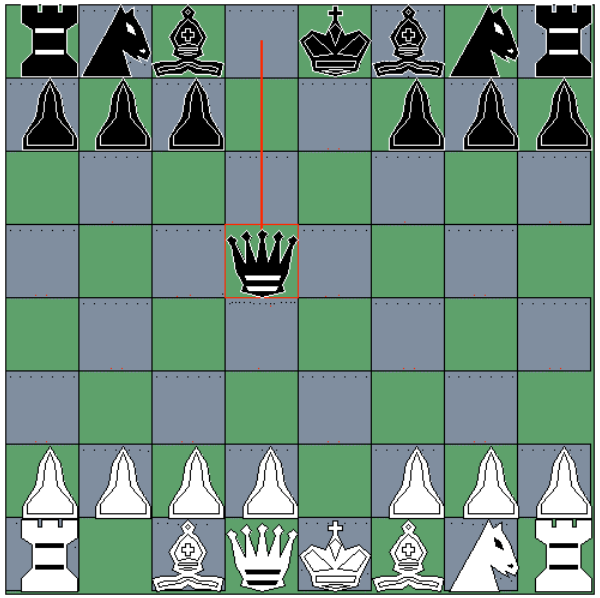
- **Example 1:** Roshambo (rock-paper-scissors)
  - » According to classical game theory, it's trivial
  - » Nash equilibrium strategy:
    - random play, expected utility = 0
- But it's not trivial in practice
  - » If you can learn how the opponent behaves, you can do much better than the Nash equilibrium
  - » International tournaments: 1999, 2000, 2003
  - » Several hundred computer programs submitted
  - » Some of them consistently did *much* better than the equilibrium strategy
    - Construct an opponent model based on their behavior
    - Use this model to predict their likely moves

Example 2: Kriegspiel Chess



- **Kriegspiel**: an imperfect-information variant of chess
  - » Developed by a Prussian military officer in 1824
  - » Became popular as a military training exercise
  - » Progenitor of modern military wargaming
- Like chess, but
  - » You don't know where your opponent's pieces are, because you can't observe most of their moves
- You get info if:
  - » You take a piece, they take a piece, they put your king in check, you make an illegal move
- Size of a *belief state* (set of all states the game *might* be in, given current info)
  - » Texas hold'em: 1,000 = 10^3
  - » bridge: 10,000,000 = 10^7
  - » kriegspiel: 10,000,000,000,000 = 10^{13}

Monte-Carlo Information-Set Search



- Recursive formulas for computing expected utilities of belief states
  - » Includes a model of the opponent's behavior
  - » Infeasible computation, due to belief-space size
- Monte Carlo approximations of belief states
  - » Reduces the computation to sort-of feasible
- Results
  - » World's 2nd-best kriegspiel program (best is at University of Bologna)
  - » The minimax opponent model (equilibrium strategy for zero-sum games) is *not* the best opponent model for kriegspiel
  - » A better model is an "overconfident" one that assumes the opponent won't play very well

http://www.cs.umd.edu/~nau/publications

A. Parker, D. Nau, and V. Subrahmanian. Overconfidence or paranoia? Search in imperfect-information games. *AAAI*, July 2006.

Iterated Prisoner's Dilemma (IPD)

- Axelrod (1984), *The Evolution of Cooperation*
- Two players, finite number of iterations of the Prisoner's Dilemma
- Widely used to study emergence of cooperative behavior among agents
  - » No optimal strategy
  - » Performance depends on the strategies of all of the players
- The best strategy in Axelrod's tournaments:
  - » ***Tit-for-Tat (TFT)***
    - On 1st move, cooperate. On *n*th move, repeat the other player's (*n*–1)-th move
  - » Could establish and maintain advantageous cooperations with many other players
  - » Could prevent malicious players from taking advantage of it

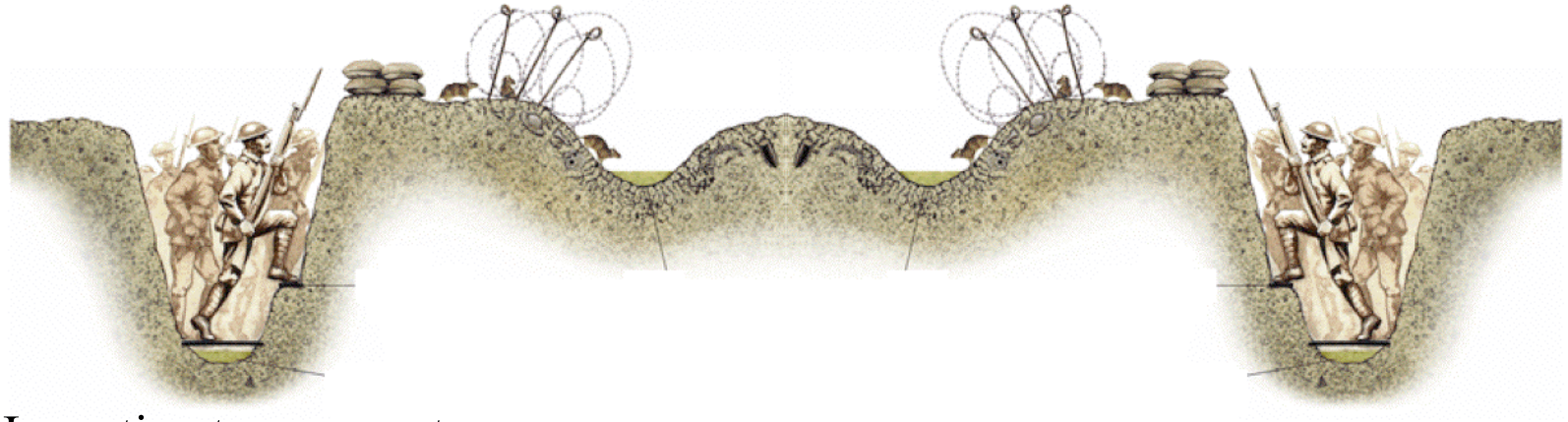
Prisoner's Dilemma:

$Player_1$ \ $Player_2$	Cooperate	Defect
Cooperate	3, 3	0, 5
Defect	5, 0	1, 1

If I defect now, he might punish me by defecting next time

Example:

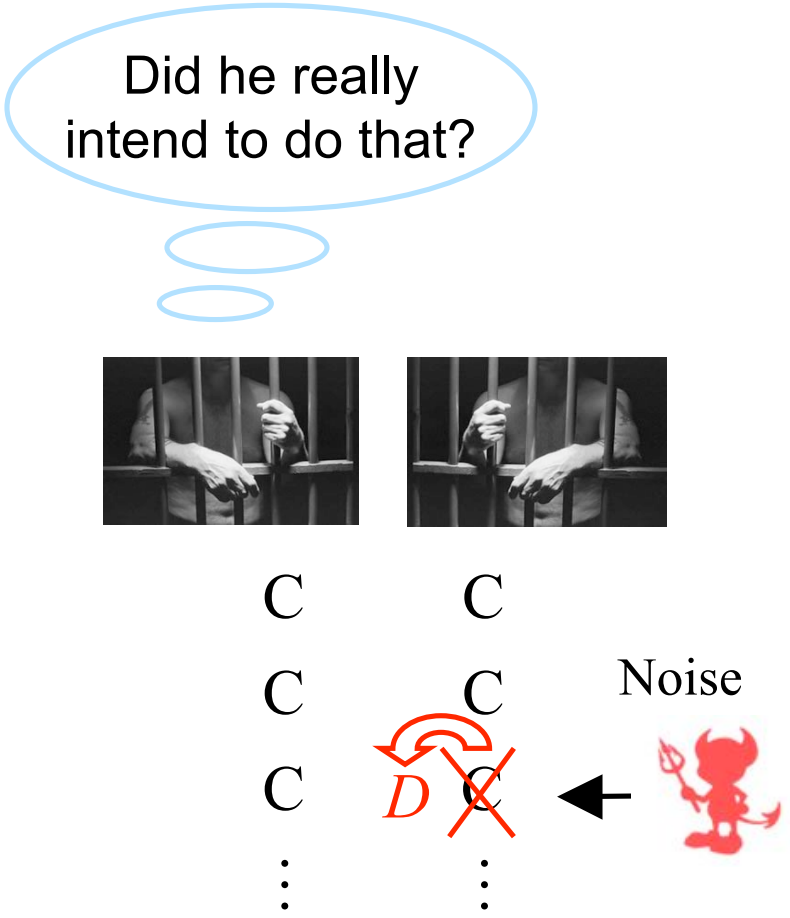
- A real-world example of the IPD, described in Axelrod's book:
  - World War I trench warfare



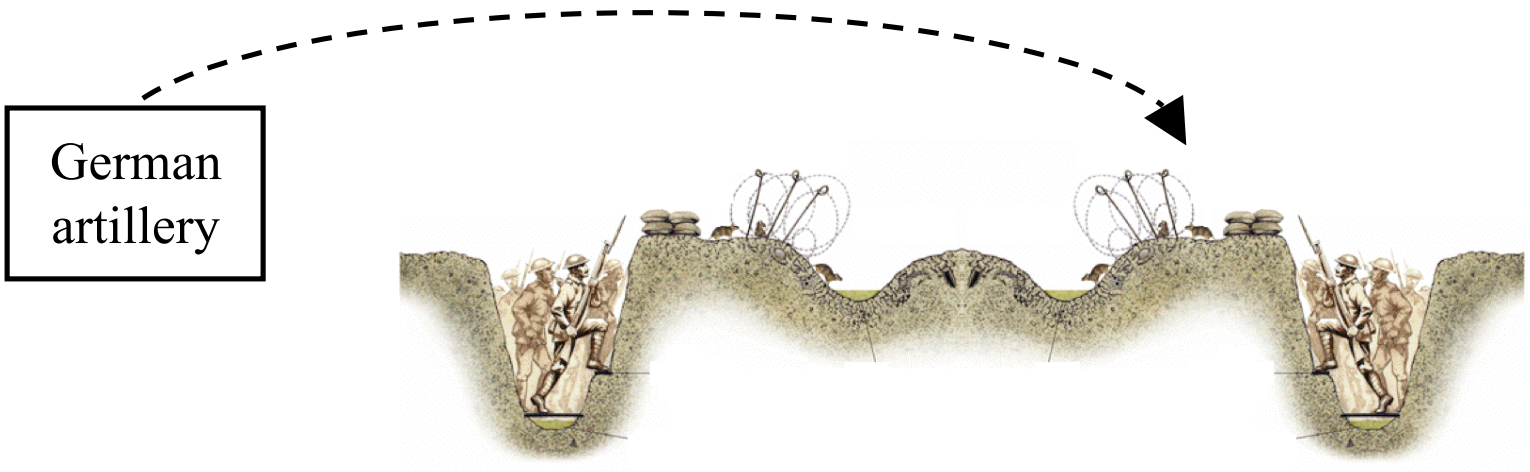
- Incentive to cooperate:
  - If I attack the other side, then they'll retaliate and I'll get hurt
  - If I don't attack, maybe they won't either
- Result: evolution of cooperation
  - Even though the two infantries were *supposed* to be enemies, they avoided attacking each other
- This was one of the reasons why World War I lasted so long

IPD with Noise

- In noisy environments,
  - » There's a nonzero probability (e.g., 10%) that a "noise gremlin" will change some of the actions
    - *Cooperate* (C) becomes *Defect* (D), and vice versa
- Can use this to model accidents
  - » Compute the score using the changed action
- Can also model misinterpretations
  - » Compute the score using the original action



Example of Noise



- Story from a British army officer in World War I:
  - » I was having tea with A Company when we heard a lot of shouting and went out to investigate. We found our men and the Germans standing on their respective parapets. Suddenly a salvo arrived but did no damage. Naturally both sides got down and our men started swearing at the Germans, when all at once a brave German got onto his parapet and shouted out: ``We are very sorry about that; we hope no one was hurt. It is not our fault. It is that damned Prussian artillery."
- The salvo wasn't the German infantry's intention
  - » They didn't expect it nor desire it

Noise Causes Problems Maintaining Cooperation

- Example:
  - Two players who both use TFT
  - One noise event can cause a long string of retaliations


	C	C	
	C	C	
	C	C	Noise
	C	D C	
Retaliation	D	C	
	C	D	Retaliation
Retaliation	D	C	
	C	D	Retaliation
	⋮	⋮	

Some Strategies for the Noisy IPD

Principle: be more forgiving in the face of defections

- Tit-For-Two-Tats (TFTT)
  - » Retaliate only if the other player defects twice in a row
    - Can tolerate isolated instances of defections, but susceptible to exploitation of its generosity
    - Beaten by the TESTER strategy I described earlier
- Generous Tit-For-Tat (GTFT)
  - » Forgive randomly: small probability of cooperation if the other player defects
  - » Better than TFTT at avoiding exploitation, but worse at maintaining cooperation
- Pavlov
  - » Win-Stay, Lose-Shift
    - Repeat previous move if I get 3 or 5 points in the previous iteration
    - Reverse previous move if I get 0 or 1 points in the previous iteration
  - » If opponent always defects, Pavlov will alternatively cooperate & defect

Discussion

- The British army officer's story:
  - … a brave German got onto his parapet and shouted out: ``We are very sorry about that; we hope no one was hurt. It is not our fault. It is that damned Prussian artillery."
- The apology avoided a conflict
  - It was convincing because it was consistent with the German infantry's past behavior
  - The British had ample evidence that the German infantry wanted to keep the peace

- **Principle**: if you can tell which actions are *affected* by noise, you can avoid *reacting* to the noise

Noise Detection by Opponent Modeling

- IPD agents often behave deterministically
  - » For others to cooperate with you, it helps if you're predictable
- The ***DBS*** program:
  - » Observe the other player's behavior
  - » Build a model π of their behavior
    - Basically a probabilistic policy
      - › **If** *current pair of moves* **then** *opponent's next move* [*p*]
    - Partly probabilistic ($0 < p < 1$)
    - Partly deterministic (p=1 or p=0)
  - » **Symbolic Noise Filtering**
    - If other player's actions disagree with π's deterministic part
      - › *Defer judgment*: assume it's noise
    - If the disagreement continues
      - › Assume their strategy has changed
      - › Recompute π based on their recent behavior

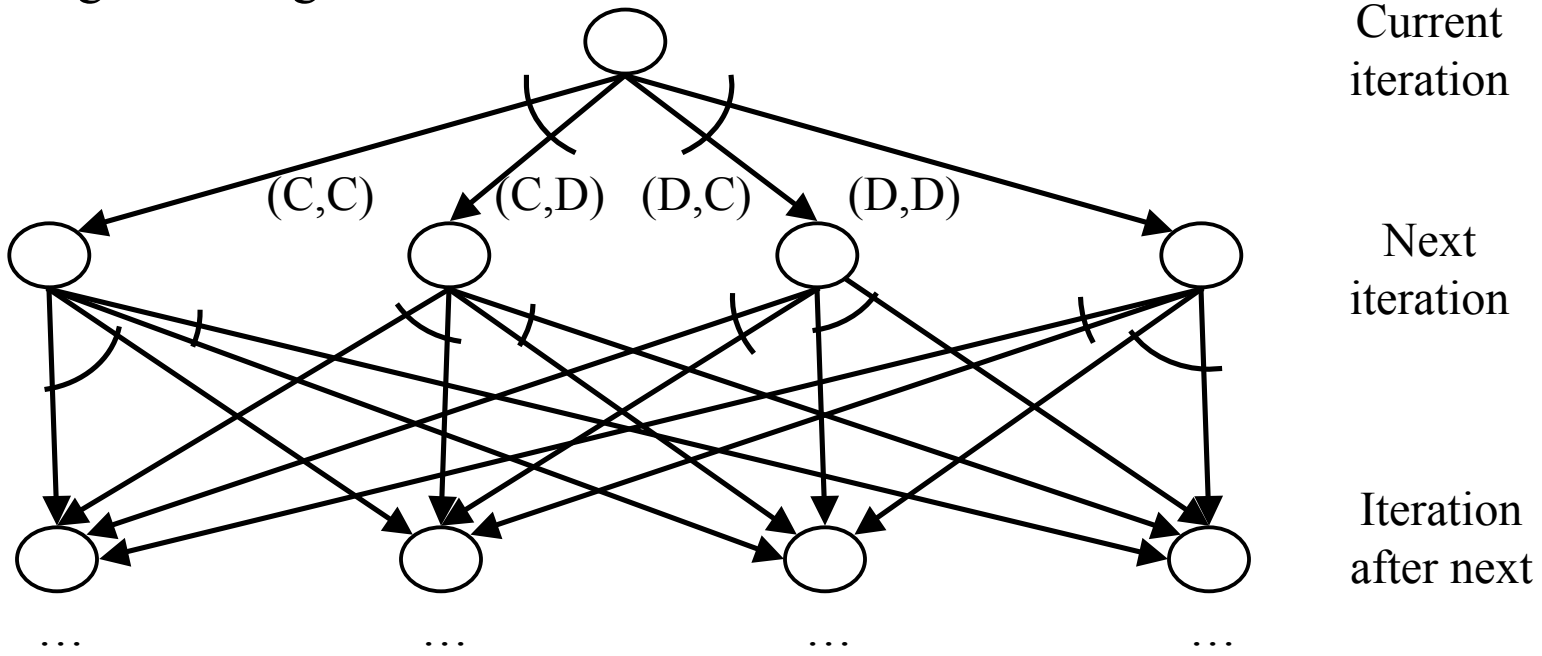
http://www.cs.umd.edu/~nau/publications

T.-C. Au and D. Nau. Accident or intention: That is the question (in the iterated prisoner's dilemma). *AAMAS*, 2006.

T.-C. Au and D. Nau.
Is it accidental or intentional? A symbolic approach to the noisy iterated prisoner's dilemma.
In G. Kendall (ed.), *The Iterated Prisoners Dilemma: 20 Years On*. World Scientific, 2007.

Planning DBS's Actions

- DBS does a game tree search against its model π of the other player
- Problem: game trees grow exponentially with search depth
- Key assumption: π accurately models the other player's future behavior
- Then we can use dynamic programming
  - » Makes the search polynomial in the search depth
  - » Can easily search to depth 60
  - » This generates good moves



The 20th-Anniversary Iterated Prisoner's Dilemma Competition

http://www.prisoners-dilemma.com

- Category 2: IPD with noise
  - » 165 programs participated
- DBS dominated the top 10 places
- But these two programs beat DBS
  - » Both used a very interesting strategy called the *master and slaves* strategy

Rank	Program	Avg. score
1	BWIN	433.8
2	IMM01	414.1
3	DBSz	408.0
4	DBSy	408.0
5	DBSpl	407.5
6	DBSx	406.6
7	DBSf	402.0
8	DBStft	401.8
9	DBSd	400.9
10	lowESTFT_classic	397.2
11	TFTIm	397.0
12	Mod	396.9
13	TFTIz	395.5
14	TFTIc	393.7
15	DBSe	393.7
16	TTFT	393.4
17	TFTIa	393.3
18	TFTIb	393.1
19	TFTIx	393.0
20	mediumESTFT_classic	392.9

Master and Slaves Strategy

- Each participant could submit up to 20 programs
- Some submitted programs that could recognize each other by exchanging sequences of Cs and Ds as identification
- Once they recognized each other, they worked as teams
  - » 1 master, 19 slaves
  - » When a slave plays with its master
    - Slave cooperates, master defects
    - Master gets 5 points, slave gets nothing
  - » When slaves play with agents not in their team
    - they defect, to minimize the other agent's payoff

- Analysis
  - » Average score of each master-slaves team was much lower than DBS's
  - » If BWIN and IMM01 each had ≤ 10 slaves, DBS would have placed 1st
  - » If BWIN and IMM01 had no slaves, they would have done badly

DBS cooperates, not coerces

- Unlike BWIN and IMM01, DBS had *no* slaves
  - » None of the DBS programs even knew the others were there
- DBS worked by establishing cooperation with *many* other agents
- DBS could do this *despite* the noise, because it could filter out the noise

Summary and Future Work

- Achieving a goal regardless of what the other agents are doing
  - » Planning with nondeterminism
  - » Reason about sets of states
    - BDD representation
  - » Maintain focus on current subgoal
    - Domain knowledge using HTNs
  - » Combining the two
  - » Experimental results on Hunter-Prey problems
- Optimizing a utility function in the presence of other agents
  - » Non-zero-sum games
  - » Opponent modeling
  - » Dealing with noise
  - » 20th Anniversary IPD competition

Summary and Future Work

- Achieving a goal regardless of what the other agents are doing
  - » Planning with nondeterminism
  - » Reason about sets of states
    - BDD representation
  - » Maintain focus on current subgoal
    - Domain knowledge using HTNs
  - » Combining the two
  - » Experimental results on Hunter-Prey problems
- Optimizing a utility function in the presence of other agents
  - » Non-zero-sum games
  - » Opponent modeling
  - » Dealing with noise
  - » 20th Anniversary IPD competition

Can generalize many classical planning algorithms to do this
[Paper under review]

Can incorporate limited forms of these into many planning algorithms
- modify the planning operators, not the algorithms themselves

[Paper under review]

Creating new strategies by combining observed behaviors of many other agents:
[Au, Nau, and Kraus, AAMAS 2008]

More sophisticated models of behavioral interactions
- Much more complex games

Any questions?