Game Applications of HTN Planning with State Variables

Dana Nau

Dept. of Computer Science, and
Institute for Systems Research

University of Maryland

Invited talk, *ICAPS Workshop on Planning in Games,* 2013

Introduction and Outline

- I've done lots of research in two areas
  - AI planning
  - games and game theory
- But mostly as separate topics
  - Many incompatibilities, difficult to combine

- **But:**
  - Workshop last year on AI and games
  - Most of the participants were doing research on video games
  - A lot of them were using planning algorithms

- **I'll talk about**
  - Incompatibilities
  - Ways to fix some of them

Planning Versus Games

	Typical AI Planning	Typical Games
State	Set of propositions	Data structures
Actions	Add/delete propositions	Modify data structures
Agents	One	Many
World	Static	Dynamic
Time available	Whatever the planner needs	Small
Objective	Find complete solution	Find partial solution
Execution	Starts after planning ends	Simultaneous with planning

- Lots of incompatibilities
  - Some easy to fix, some more difficult

Using Planning in Games

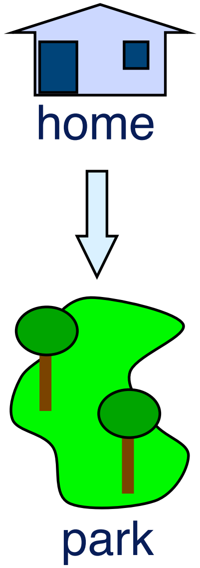
- ***Approximate*** some part of the game as a planning problem
  - » Develop a ***special-purpose*** planner for that problem
  - » Use it as a subroutine

- I'll discuss some examples that involve HTN planning
  - » But first, a description of how HTN planning works

HTN Planning

- Motivation
  - » For some planning problems, we may already have ideas for how to look for solutions
- Example: travel to a destination that's far away:
  - » Brute-force search:
    - Many ways to combine vehicles and routes
  - » Experienced human: small number of "recipes"
    - e.g., flying:
      1. buy ticket from local airport to remote airport
      2. travel to local airport
      3. fly to remote airport
      4. travel to final destination
  - » HTN planners use such recipes to generate the search space
- Ingredients
  - » states, tasks, operators, methods, planning algorithm

States and Tasks



- **State**: description of the current situation
  - I'm at home, I have €20, there's a park 8 km away
- **Task**: description of an activity to perform
  - Travel to the park
- Two kinds of tasks
  - ***Primitive*** task: a task that corresponds to a basic action
  - ***Compound*** task: a task that is composed of other simpler tasks

Operators

- **Operators**: parameterized descriptions of what the basic actions do
  - ***walk*** from location x to location y
    - Precond: agent is at x
    - Effects: agent is at y
  - ***call taxi*** to location x
    - Precond: (none)
    - Effects: taxi is at x
  - ***ride taxi*** from location x to location y
    - Precond: agent and taxi are at x
    - Effects: agent and taxi at y, agent owes 1.50 + ½ distance(x,y)
  - ***pay driver***
    - Precond: agent owes amount of money r, agent has money $m \geq r$
    - Effects: agent owes nothing, agent has money $m - r$
- **Actions**: operators with arguments

Methods

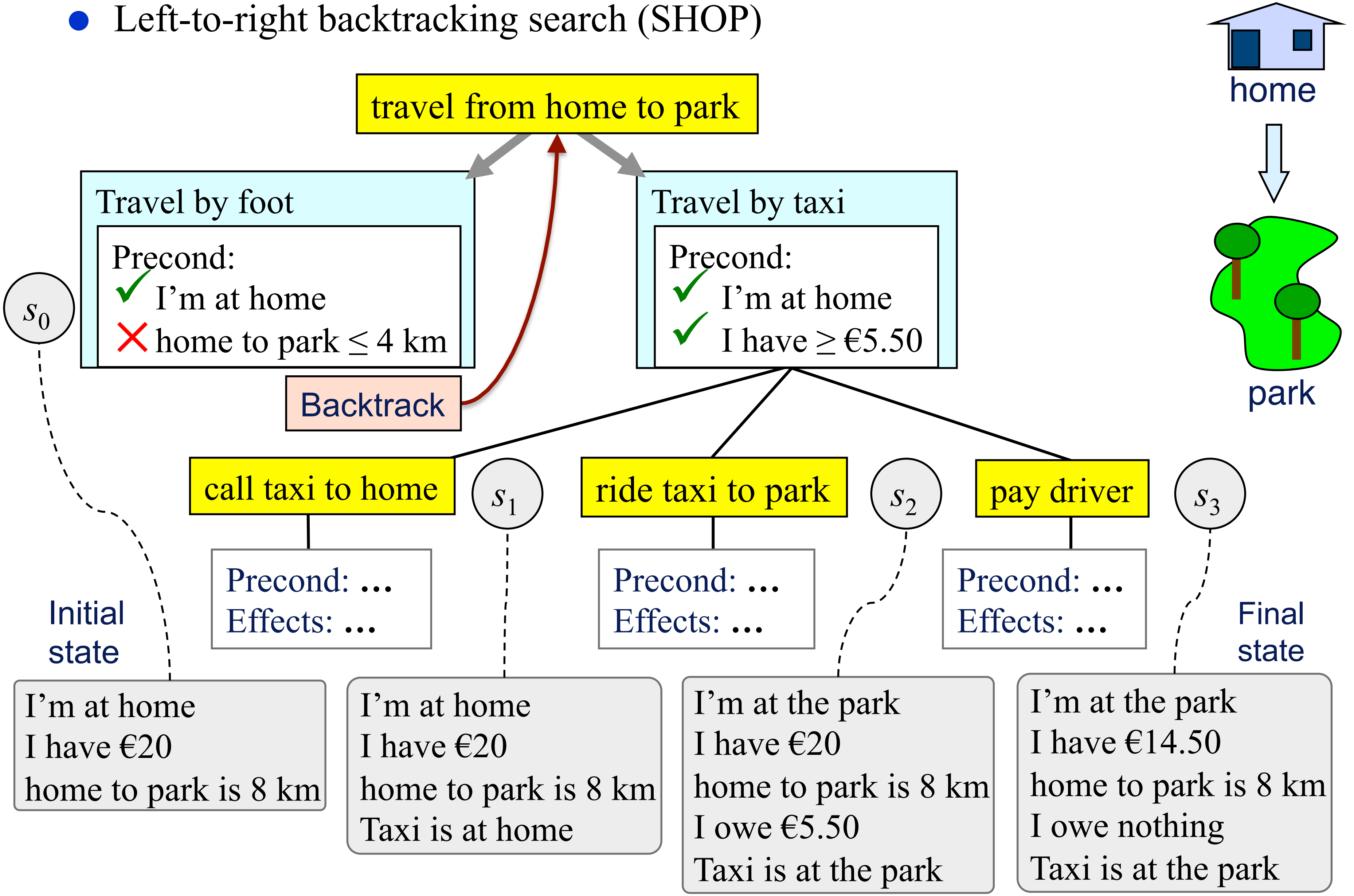
- Method: parameterized description of a possible way to perform a compound task by performing a collection of subtasks
- There may be more than one method for the same task

  » ***travel by foot*** from x to y
    - Task: travel from x to y
    - Precond: agent is at x, distance to y is ≤ 4 km
    - Subtasks: walk from x to y

  » ***travel by taxi*** from x to y
    - Task: travel from x to y
    - Precond: agent is at x, agent has money $\geq 1.5 + \frac{1}{2}$ distance(x,y)
    - Subtasks: call taxi to x,
ride taxi from x to y,
pay driver

Simple Travel-Planning Problem

- Left-to-right backtracking search (SHOP)

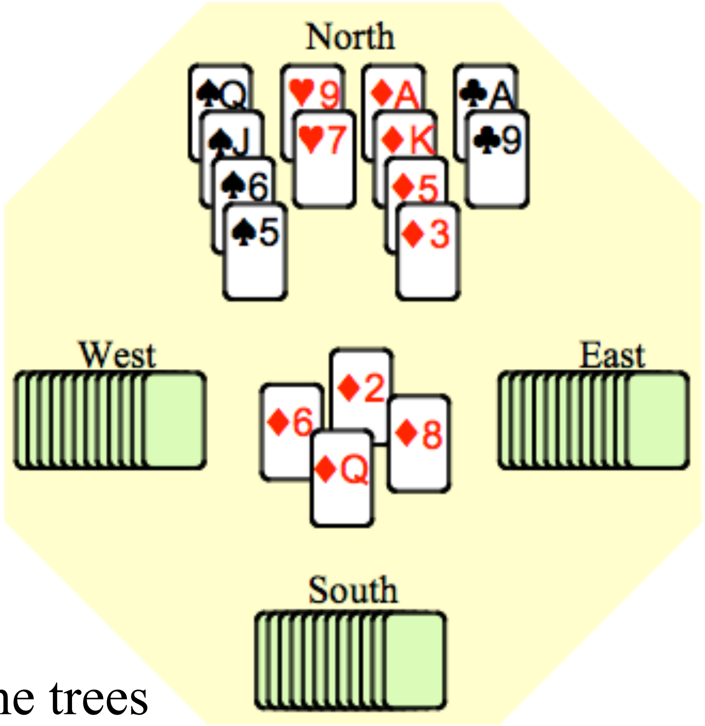


SHOP and SHOP2

- SHOP and SHOP2:
  - http://www.cs.umd.edu/projects/shop
  - HTN planning systems
  - SHOP2 an award in the AIPS-2002 Planning Competition
- Freeware, open source
  - Downloaded more than 20,000 times
  - Used in many hundreds of projects worldwide
    - Government labs, industry, academia

Bridge

- Ideal: game-tree search (all lines of play) to compute expected utilities
- Don't know what cards other players have
  - » Many moves they *might* be able to make
    - worst case about 6x10^{44} leaf nodes
    - average case about 10^{24} leaf nodes
- About 1½ minutes available



Not enough time – need smaller tree

- ***Bridge Baron***
  - » 1997 world champion of computer bridge
- Special-purpose HTN planner that generates game trees
  - » Branches ⇔ standard bridge card plays (finesse, ruff, cash out, …)
  - » Much smaller game tree: can search it and compute expected utilities
- **Why it worked**:
  - » Special-purpose planner to generate trees rather than linear plans
  - » Lots of work to make the HTN methods as complete as possible

KILLZONE 2

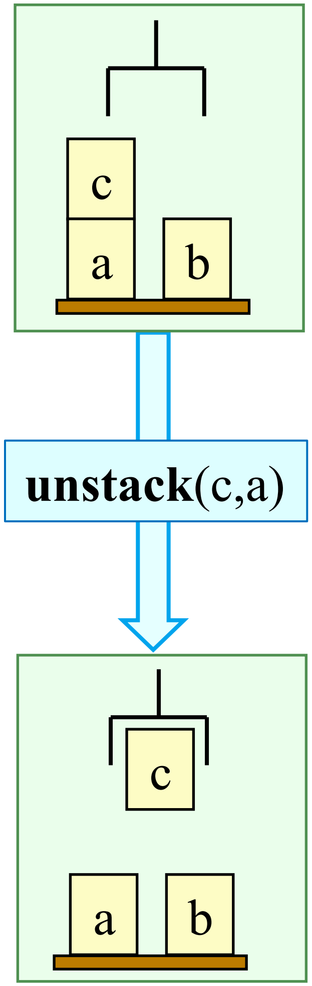
- Special-purpose HTN planner for planning at the squad level
  - Method and operator syntax similar to SHOP's and SHOP2's
  - Quickly generates a linear plan that would work if nothing interferes
  - Replan several times per second as the world changes
- **Why it worked:**
  - Very different objective from a bridge tournament
  - Don't ***want*** to look for the best possible play
  - Need actions that appear believable and consistent to human users
  - Need them very quickly

Planning Versus Games

- These incompatibilities are easy to fix
  - Instead of logical propositions, use state variables

	Typical AI Planning	**Typical Games**
State	Set of propositions	Data structures
Actions	Add/delete propositions	Modify data structures
Agents	One	Many
World	Static	Dynamic
Time available	Whatever the planner needs	Small
Objective	Find complete solution	Find partial solution
Execution	Starts after planning ends	Simultaneous with planning

Propositions Versus State Variables



{ontable(a), on(c,a), clear(c), ontable(b), clear(b), handempty}

{loc(a)=table, clear(a)=0, loc(c)=a, clear(c)=1, loc(b)=table, clear(b)=1, holding=nothing}

unstack(x,y)
Precond: on(x,y), clear(x), handempty
Effects: ¬on(x,y), ¬clear(x), clear(y), holding(x), ¬handempty

unstack(x,y)
Precond: loc(x) = y, $y \neq$ table, clear(x) = 1, holding = nothing
Effects: loc(x) = hand, clear(x) = 0, clear(y) = 1, holding = x

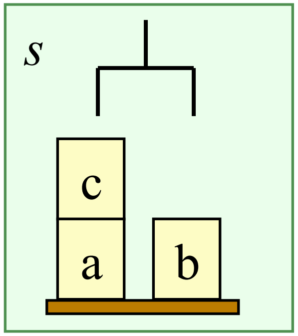
- Classical representation:
  - State: set of propositions
  - Actions add/delete them
- PDDL is based on this
- Reason is largely historical
  - AI planning evolved out of AI theorem proving

- State-variable representation:
  - State: variable bindings
  - Actions change the values
- Same expressive power
- More compatible with conventional computer programming

Pyhop

- A simple HTN planner written in Python
  - » Works in both Python 2.7 and 3.2

- Planning algorithm is like the one in SHOP
- Main differences:
  - » HTN operators and methods are ordinary Python functions
  - » The current state is a Python object that contains variable bindings
    - Operators and methods refer to states explicitly
    - To say `c` is on `a`, write `s.loc['c'] = 'a'` where `s` is the current state



- Easy to implement and understand
  - » Less than 150 lines of code

- Open-source software, Apache license
  - » http://bitbucket.org/dananau/pyhop

Travel-Planning Methods

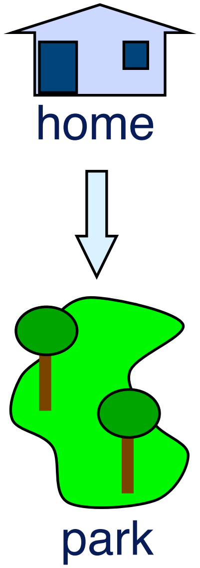
travel by foot from *x* to *y*

Task: travel from *x* to *y*

Precond: agent is at *x*, distance to *y* is ≤ 4 km

Subtasks: walk from *x* to *y*

```
def travel_by_foot(state,a,x,y):
    if state.dist[x][y] <= 4:
        return [('walk',a,x,y)]
    return False
```



travel by taxi from *x* to *y*

Task: travel from *x* to *y*

Precond: agent is at *x*, agent has money ≥ 1.5 + ½ distance(*x,y*)

Subtasks: call taxi to *x*, ride taxi from *x* to *y*, pay driver

```
def travel_by_taxi(state,a,x,y):
    if state.cash[a] >= 1.5 + 0.5 * state.dist[x][y]:
        return [('call_taxi',a,x),
                ('ride_taxi',a,x,y),
                ('pay_driver',a,x,y)]
    return False

declare_methods('travel',travel_by_foot,travel_by_taxi)
```

Travel-Planning Operators (1)

walk from *x* to *y*

Precond: agent is at location *x*

Effects: agent is at location *y*

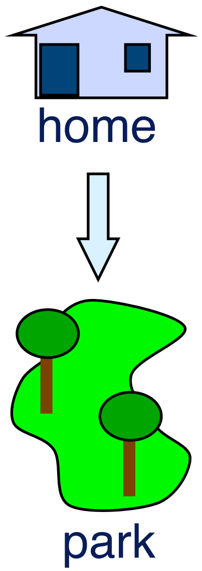
```
def walk(state,a,x,y):
    if state.loc[a] == x:
        state.loc[a] = y
        return state
    else: return False
```

call taxi to location *x*

Precond: (none)

Effects: taxi is at location *x*

```
def call_taxi(state,a,x):
    state.loc['taxi'] = x
    return state
```



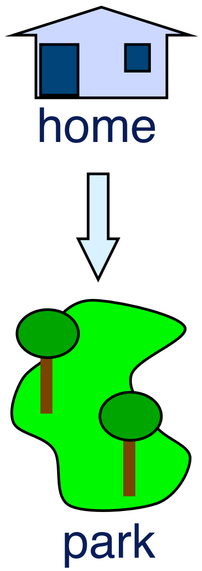
Travel-Planning Operators (2)

ride taxi from *x* to *y*

Precond: agent and taxi are at *x*

Effects: agent and taxi are at *y*, agent owes 1.5 + ½ distance(*x*,*y*)

```
def ride_taxi(state,a,x,y):
  if state.loc['taxi']==x and state.loc[a]==x:
    state.loc['taxi'] = y
    state.loc[a] = y
    state.owe[a] = 1.5 + 0.5*state.dist[x][y]
    return state
  else: return False
```



pay driver

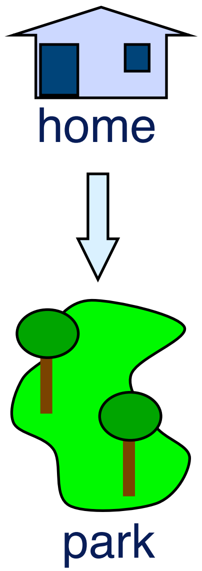
Precond: agent owes money, and has at least as much as what's owed

Effects: agent owes nothing, agent's money reduced by what was owed

```
def pay_driver(state,a):
  if state.cash[a] >= state.owe[a]:
    state.cash[a] = state.cash[a] - state.owe[a]
    state.owe[a] = 0
    return state
  else: return False

declare_operators(walk, call_taxi, ride_taxi, pay_driver)
```

Travel Planning Problem



Initial state: I’m at home, I have €20, there’s a park 8 km away

```
state1 = State('state1’)
state1.loc = {'me':'home’}
state1.cash = {'me':20}
state1.owe = {'me’:0}
state1.dist = {'home':{'park':8}, ’park':{’home':8}}
```

Task: travel to the park

```
# Invoke the planner
pyhop(state1,[('travel','me','home','park')])
```

Solution plan: call taxi, ride taxi from home to park, pay driver

```
[('call_taxi', 'me', 'home'),
 ('ride_taxi', 'me', 'home', 'park'),
 ('pay_driver', 'me')]
```

Planning Versus Games

- Pyhop resolves these incompatibilities

	Typical AI Planning	Typical Games
State	Set of propositions	Data structures
Actions	Add/delete propositions	Modify data structures
Agents	One	Many
World	Static	Dynamic
Time available	Whatever the planner needs	Small
Objective	Find complete solution	Find partial solution
Execution	Starts after planning ends	Simultaneous with planning

- Are there general solutions for these?
  - Or do they need to be game-specific?

Summary

- State-variable representation makes it easier to integrate planning into ordinary programming
- Pyhop is an HTN planner that does this
  - Written in Python
  - Simple algorithm, easy to understand
  - Open source (Apache license)
  - Downloadable at http://bitbucket.org/dananau/pyhop
- I hope some of you will find it useful
  - If you use it, please let me know
  - I hope some of you will post enhancements
- Resolves some of the incompatibilities between AI planning and games
  - But not all of them
  - How best to resolve the others?