CMSC 671
Fall 2010

Tue 10/26/10

Scheduling and HTN Planning
Multiagent Planning

Chapter 11

Prof. Laura Zavala, laura.zavala@umbc.edu, ITE 373, 410-455-8775

Real-world planning domains

- Real-world domains are complex and don't satisfy the assumptions of STRIPS or partial-order planning methods
- Some of the characteristics we may need to deal with:
  - Modeling and reasoning about resources
  - Representing and reasoning about time
  - Planning at different levels of abstractions
  - Conditional outcomes of actions
  - Uncertain outcomes of actions
  - Exogenous events
  - Incremental plan development
  - Dynamic real-time replanning

} Scheduling

} Planning under uncertainty

} HTN planning

Scheduling

Scheduling

- **Representing and solving planning problems that include *temporal* and *resource* constraints**

- **Scheduling**: Given a set of actions, resources, and constraints, find the assignment of actions to resources (including time assignments) that satisfies or optimizes the set of constraints

Scheduling

- "Plan first, schedule later" approach
  - Common in real-world manufacturing and logistical settings where the planning is often performed by human experts.
  - Automated classic planning methods that produce plans with just the minimal ordering constraints can also be used for the planning phase.

    GRAPHPLAN, SATPLAN

    (search based methods produce totally ordered plans)

Representing temporal and resource constraints
Job-shop scheduling

- Set of **jobs** to be completed (actions + ordering constraints)
- Available Resources
- Actions: duration and resource constraints (usage, consumption, and production)

Jobs({*AddEngine1* ≺ *AddWheels1* ≺ *Inspect1*},
    {*AddEngine2* ≺ *AddWheels2* ≺ *Inspect2*})

Resources(*EngineHoists*(1), *WheelStations*(1), *Inspectors*(2), *LugNuts*(500))

Action(*AddEngine1*, DURATION:30,
    USE:*EngineHoists*(*1*))
Action(*AddEngine2*, DURATION:60,
    USE:*EngineHoists*(*1*))
Action(*AddWheels1*, DURATION:30,
    CONSUME:*LugNuts*(20), USE:*WheelStations*(1))
Action(*AddWheels2*, DURATION:15,
    CONSUME:*LugNuts*(20), USE:*WheelStations*(1))
Action($Inspect_i$, DURATION:10,
    USE:*Inspectors*(1))

Assembling two cars

A<B:
action A must precede action B

Representing temporal and resource constraints
Job-shop scheduling

- A solution specifies the start times for each action and must satisfy all constraints (ordering and resources)
- Cost function is the total duration of the plan (**makespan**)

$Jobs(\{AddEngine1 \prec AddWheels1 \prec Inspect1\},$
$\quad \{AddEngine2 \prec AddWheels2 \prec Inspect2\})$

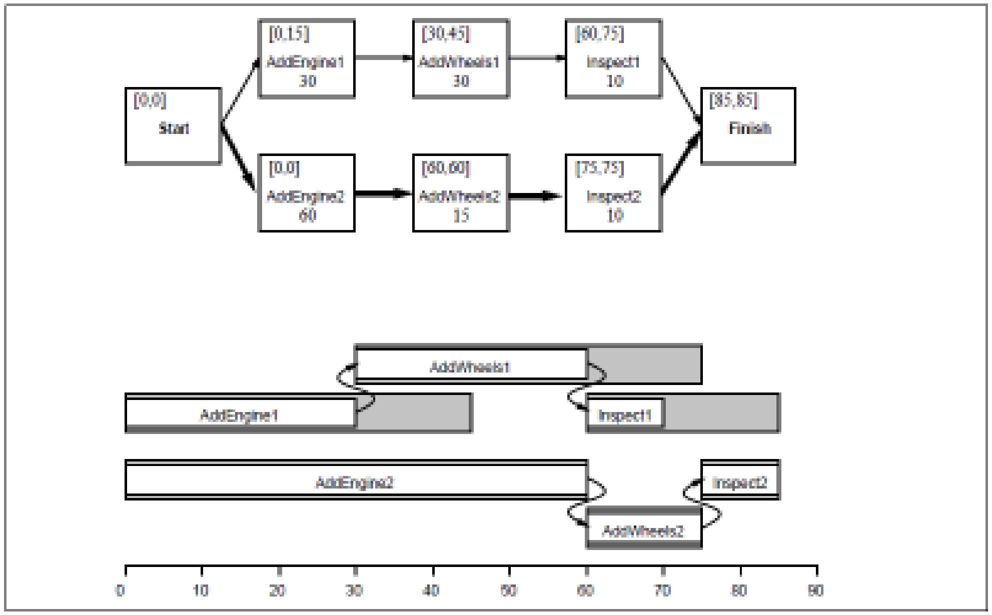
$Resources(EngineHoists(1), WheelStations(1), Inspectors(2), LugNuts(500))$

$Action(AddEngine1, \text{DURATION}{:}30,$
$\quad \text{USE}{:}EngineHoists(1))$
$Action(AddEngine2, \text{DURATION}{:}60,$
$\quad \text{USE}{:}EngineHoists(1))$
$Action(AddWheels1, \text{DURATION}{:}30,$
$\quad \text{CONSUME}{:}LugNuts(20), \text{USE}{:}WheelStations(1))$
$Action(AddWheels2, \text{DURATION}{:}15,$
$\quad \text{CONSUME}{:}LugNuts(20), \text{USE}{:}WheelStations(1))$
$Action(Inspect_i, \text{DURATION}{:}10,$
$\quad \text{USE}{:}Inspectors(1))$

Assembling two cars

A<B:
action A must precede action B

Job-shop scheduling
Assembling two cars example



Directed graph and timeline representations of the ordering constraints

Jobs({*AddEngine1* $\prec$ *AddWheels1* $\prec$ *Inspect1*},
{*AddEngine2* $\prec$ *AddWheels2* $\prec$ *Inspect2*})

Resources(*EngineHoists*(1), *WheelStations*(1), *Inspectors*(2), *LugNuts*(500))

Action(*AddEngine1*, DURATION:30,
USE: *EngineHoists*(1))
Action(*AddEngine2*, DURATION:60,
USE: *EngineHoists*(1))
Action(*AddWheels1*, DURATION:30,
CONSUME: *LugNuts*(20), USE: *WheelStations*(1))
Action(*AddWheels2*, DURATION:15,
CONSUME: *LugNuts*(20), USE: *WheelStations*(1))
Action($Inspect_i$, DURATION:10,
USE: *Inspectors*(1))

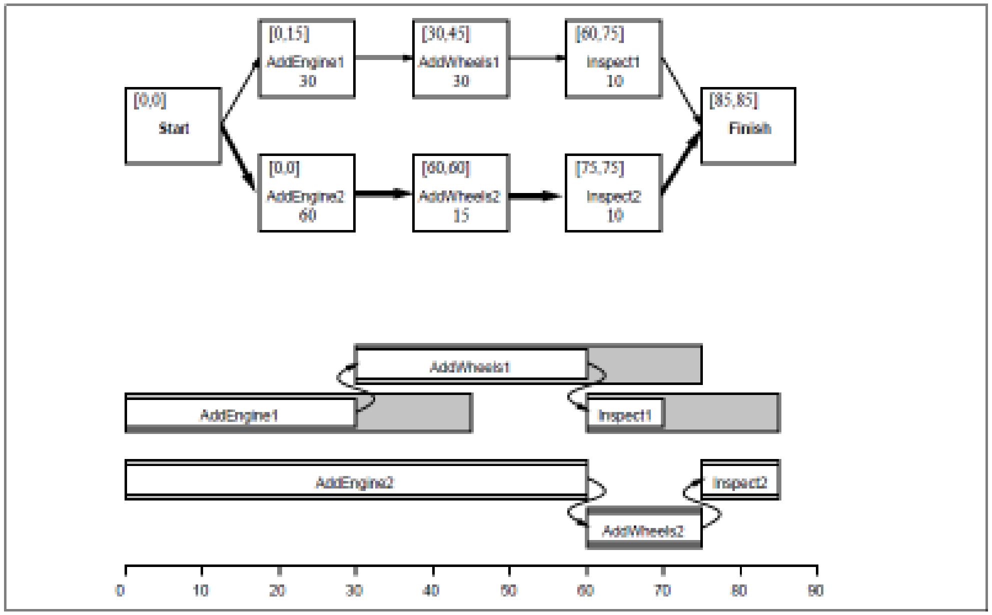
Temporal scheduling

- Minimize **makespan** (total duration of all actions)
  - Actions have earliest and latest possible start times: [ES, LS]
  - LS-ES = slack time of an action
- Treat as graph-theoretic problem of finding shortest path from earliest start time to latest end time of any action
  - Path = linearization of plan
  - Shortest path = path with shortest overall duration
- **Critical path method**: dynamic programming approach for finding the shortest path

Temporal scheduling

- **Critical path method**
  - Determine the possible start and end times of each action
- **Critical path**
  - Path whose total duration is the longest
  - Delaying the start of an action slows down the whole plan
  - LS-ES = slack time of an action
    - Each action on the critical path has no slack

Job-shop scheduling
Assembling two cars example



Directed graph and timeline representations of the ordering constraints

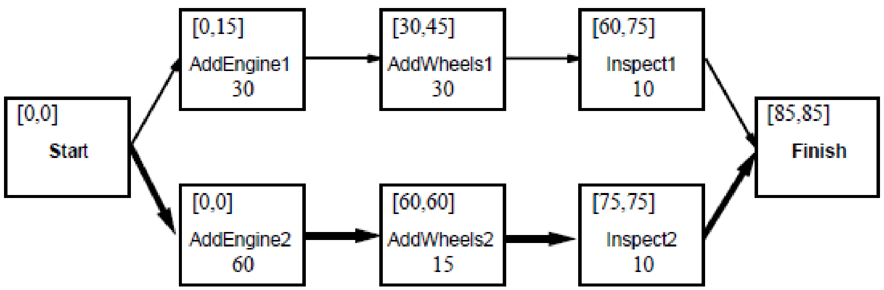
Jobs({*AddEngine1* ≺ *AddWheels1* ≺ *Inspect1*},
{*AddEngine2* ≺ *AddWheels2* ≺ *Inspect2*})

Resources(*EngineHoists*(1), *WheelStations*(1), *Inspectors*(2), *LugNuts*(500))

Action(*AddEngine1*, DURATION:30,
USE: *EngineHoists*(*1*))
Action(*AddEngine2*, DURATION:60,
USE: *EngineHoists*(*1*))
Action(*AddWheels1*, DURATION:30,
CONSUME: *LugNuts*(20), USE: *WheelStations*(1))
Action(*AddWheels2*, DURATION:15,
CONSUME: *LugNuts*(20), USE: *WheelStations*(1))
Action($Inspect_i$, DURATION:10,
USE: *Inspectors*(1))

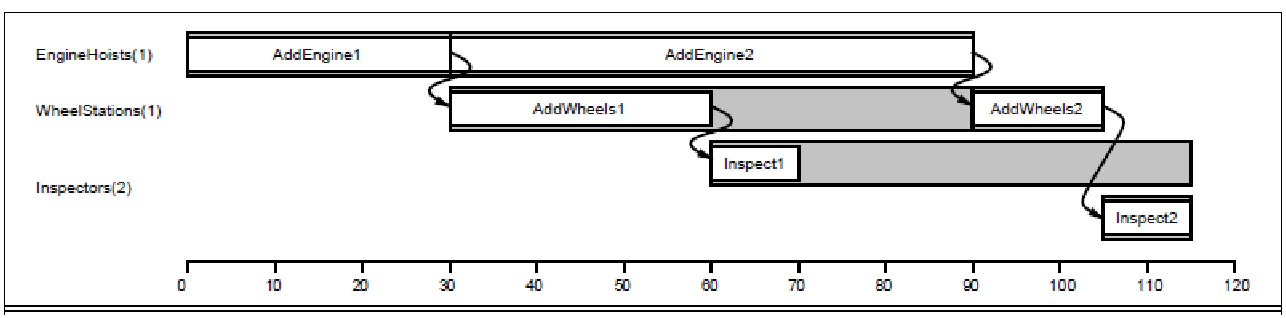
Temporal scheduling

- **Critical path problems are easy to solve**
  - Linear in number of actions and branching factor
  - Conjunction of linear equalities on the start and end times



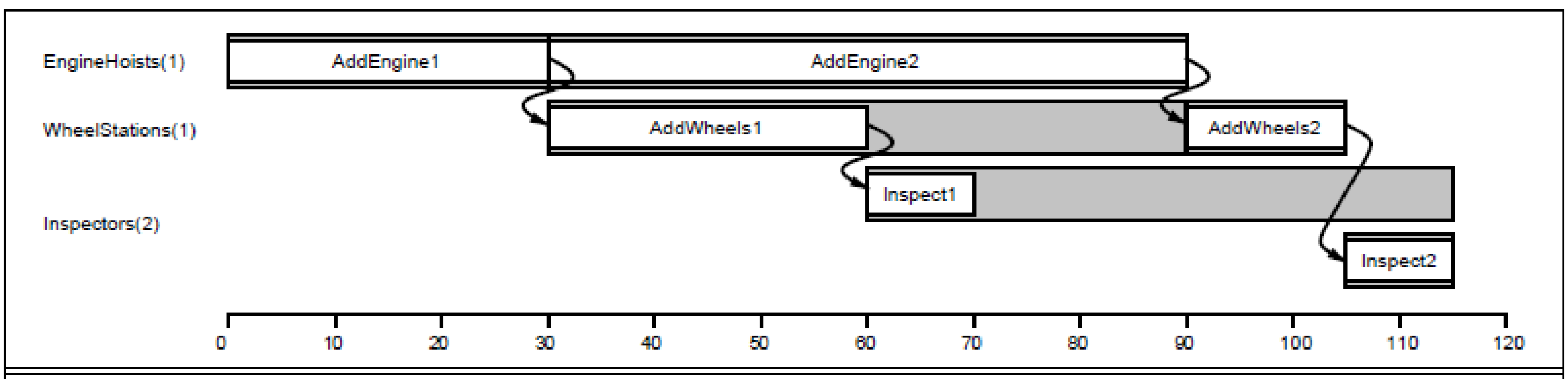
Adding resource constraints

- Constraints may now be disjunctive:
  - Two actions, A and B, sharing a resource can’t overlap
  - A could end before B starts, or start after B ends
  - Finding the optimal ordering is now NP-hard!
- Heuristics for finding a good ordering:
  - Minimum slack: Greedy algorithm that chooses the unscheduled action with the least slack (essentially a most-constrained heuristic)



Adding resource constraints

- Constraints may now be disjunctive:
  - Two actions, A and B, sharing a resource can't overlap
  - A could end before B starts, or start after B ends
  - Finding the optimal ordering is now NP-hard!



The AddEngine actions require the same EngineHoist
Shortest duration solution (115 minutes)

Adding resource constraints

- Adding disjunctions makes scheduling with resource constraints NP-hard.
- Heuristics for finding a good ordering:
  - Minimum slack: Greedy algorithm that chooses the unscheduled action with the least slack (essentially a most-constrained heuristic)

HTN Planning

HTN Planning

- We may already have an idea how to go about solving problems in a planning domain
- Exponential number of actions for real-world plans
  - E.g. Travel to a far away destination
    - Really difficult to make it as sequences of *right, left, up, down* moves only
    - Domain-independent planner:
      - many combinations of vehicles and routes
- Solution - To do what humans appear to do:

  Plan at higher levels of abstraction

HTN Planning

- 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

- How to enable planning systems to make use of such recipes?

Hierarchical decomposition

- Hierarchical decomposition, or hierarchical task network (**HTN**) planning, uses **abstract operators** to **incrementally** decompose a planning problem from a **high-level goal** statement to a **primitive plan network**
- **Primitive operators** represent actions that are **executable**, and can appear in the final plan
- **Non-primitive operators** represent **tasks** (equivalently, **abstract actions**) that require further decomposition (or *operationalization*) to be executed
- Tasks decompose into **subtasks**

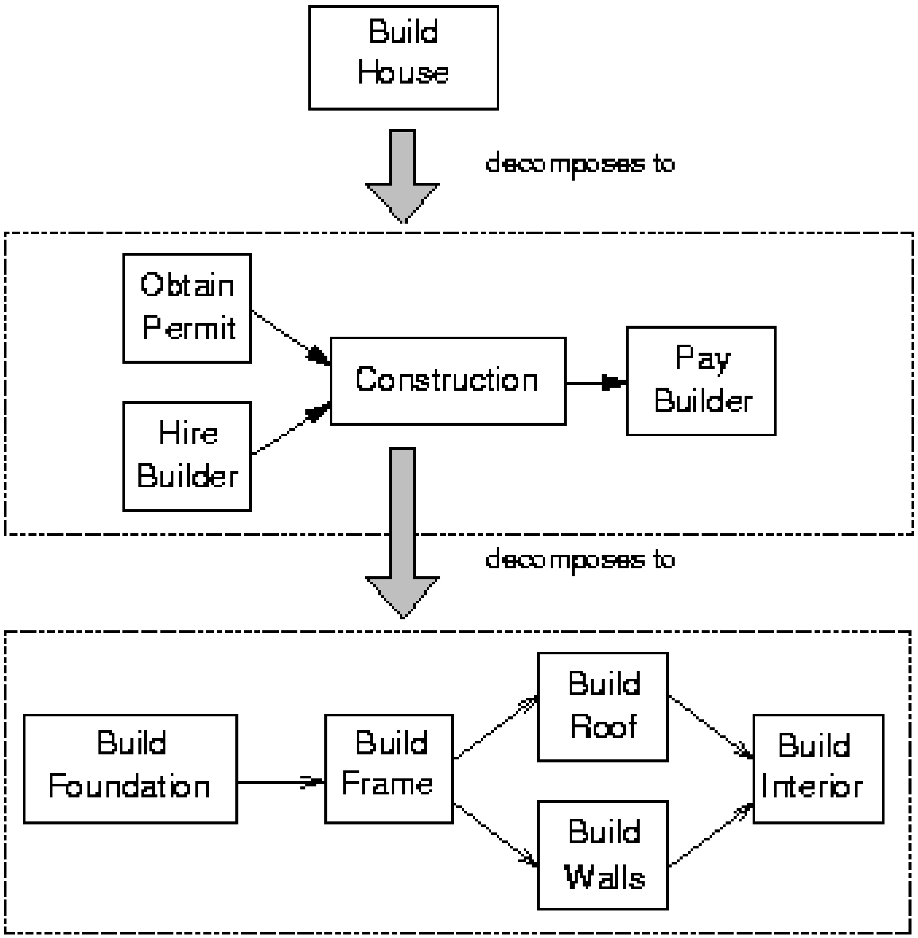
  Constraints

  Backtrack if necessary
- There is no "right" set of primitive actions: One agent's goals are another agent's actions!

HTN operator: Example

```
OPERATOR decompose
PURPOSE: Construction
CONSTRAINTS:
    Length (Frame) <= Length (Foundation),
    Strength (Foundation) > Wt(Frame) + Wt(Roof)
        + Wt(Walls) + Wt(Interior) + Wt(Contents)
PLOT: Build (Foundation)
      Build (Frame)
      PARALLEL
            Build (Roof)
            Build (Walls)
      END PARALLEL
      Build (Interior)
```

HTN planning: example



Assumptions

- Full observability
- Determinism
- Availability of a set of actions
  - Primitive Actions
- High level actions
  - One or more possible refinements into a sequence of actions (HLA or primitive)
- HLA library

Refinements

- Embody knowledge about *how to do things*
- Go to San Francisco airport
  - Drive or take a taxi
  - Buting milk, sitting down, etc., are not considered

$Refinement(Go(Home, SFO),$
  STEPS: $[Drive(Home, SFOLongTermParking),$
    $Shuttle(SFOLongTermParking, SFO)]\ )$
$Refinement(Go(Home, SFO),$
  STEPS: $[Taxi(Home, SFO)]\ )$

Refinements example 2

- Navigating in the vacuum world
  - To get to a destination, take a step, and then go to the destination
  - Recursive nature of refinements
  - Use of preconditions

$Refinement(Navigate([a, b], [x, y]),$
  PRECOND: $a = x \ \wedge \ b = y$
  STEPS: [ ])
$Refinement(Navigate([a, b], [x, y]),$
  PRECOND: $Connected([a, b], [a-1, b])$
  STEPS: $[Left, Navigate([a-1, b], [x, y])]$)
$Refinement(Navigate([a, b], [x, y]),$
  PRECOND: $Connected([a, b], [a+1, b])$
  STEPS: $[Right, Navigate([a+1, b], [x, y])]$)
...

HLA implementation

- An HLA refinement that contains only primitive actions

Navigate ([1,3], [3,2])

[Right, Right, Down]

[Down, Down, Right]

Achieving the goal

- A high level plan achieves the goal from a given state if at least one of its implementations achieves the goal from that state
  - Note: not all implementations need to achieve the goal
- Finding a solution plan
  - Search among the implementations for one that works
  - Reason directly about the HLAs

Search among the implementations for one that works

- Repeatedly choose an HLA in the current plan and replace it with one of its refinements

```
function HIERARCHICAL-SEARCH(problem, hierarchy) returns a solution, or failure
  frontier ← a FIFO queue with [Act] as the only element
  loop do
     if EMPTY?(frontier) then return failure
     plan ← POP(frontier)   /* chooses the shallowest plan in frontier */
     hla ← the first HLA in plan, or null if none
     prefix,suffix ← the action subsequences before and after hla in plan
     outcome ← RESULT(problem.INITIAL-STATE, prefix)
     if hla is null then   /* so plan is primitive and outcome is its result */
        if outcome satisfies problem.GOAL then return plan
     else for each sequence in REFINEMENTS(hla, outcome, hierarchy) do
        frontier ← INSERT(APPEND(prefix, sequence, suffix), frontier)
```

Figure 11.5 A breadth-first implementation of hierarchical forward planning search. The initial plan supplied to the algorithm is [*Act*]. The REFINEMENTS function returns a set of action sequences, one for each refinement of the HLA whose preconditions are satisfied by the specified state, *outcome*.

Plans are considered in order of depth of nesting of the refinements rather than the number of primitive steps

Search among the implementations for one that works

- Explores the space of sequences of actions, restricted ot guided by the knowledge in the HLA library
- Very computationally efficient
- Even more efficient if the HLAs in the library have a small number of refinements each yielding a long action sequence
  - A case not very commonly found in practice: long action sequences usable across a wide range of problems
- Generalize and Learn!

Searching for abstract solutions

- High level planning

  [Drive(Home, SFOLongermParking), Shuttle (SFOLongTermParking, SFO)]

- No need to know the details
  - (route, parking spot, etc)
- Preconditions and effects for the HLAs
- Provably correct plans are derived without consideration of low level implementations
  - We can always work out the details of each step
  - Exponential reduction

Searching for abstract solutions

```
function ANGELIC-SEARCH(problem, hierarchy, initialPlan) returns solution or fail
  frontier ← a FIFO queue with initialPlan as the only element
  loop do
     if EMPTY?(frontier) then return fail
     plan ← POP(frontier)   /* chooses the shallowest node in frontier */
     if REACH+(problem.INITIAL-STATE, plan) intersects problem.GOAL then
        if plan is primitive then return plan   /* REACH+ is exact for primitive plans */
        guaranteed ← REACH−(problem.INITIAL-STATE, plan) ∩ problem.GOAL
        if guaranteed≠{ } and MAKING-PROGRESS(plan, initialPlan) then
           finalState ← any element of guaranteed
           return DECOMPOSE(hierarchy, problem.INITIAL-STATE, plan, finalState)
        hla ← some HLA in plan
        prefix,suffix ← the action subsequences before and after hla in plan
        for each sequence in REFINEMENTS(hla, outcome, hierarchy) do
           frontier ← INSERT(APPEND(prefix, sequence, suffix), frontier)
```

```
function DECOMPOSE(hierarchy, s0, plan, sf) returns a solution
  solution ← an empty plan
  while plan is not empty do
    action ← REMOVE-LAST(plan)
    si ← a state in REACH−(s0, plan) such that sf ∈REACH−(si, action)
    problem ← a problem with INITIAL-STATE = si and GOAL = sf
    solution ← APPEND(ANGELIC-SEARCH(problem, hierarchy, action), solution)
    sf ← si
  return solution
```

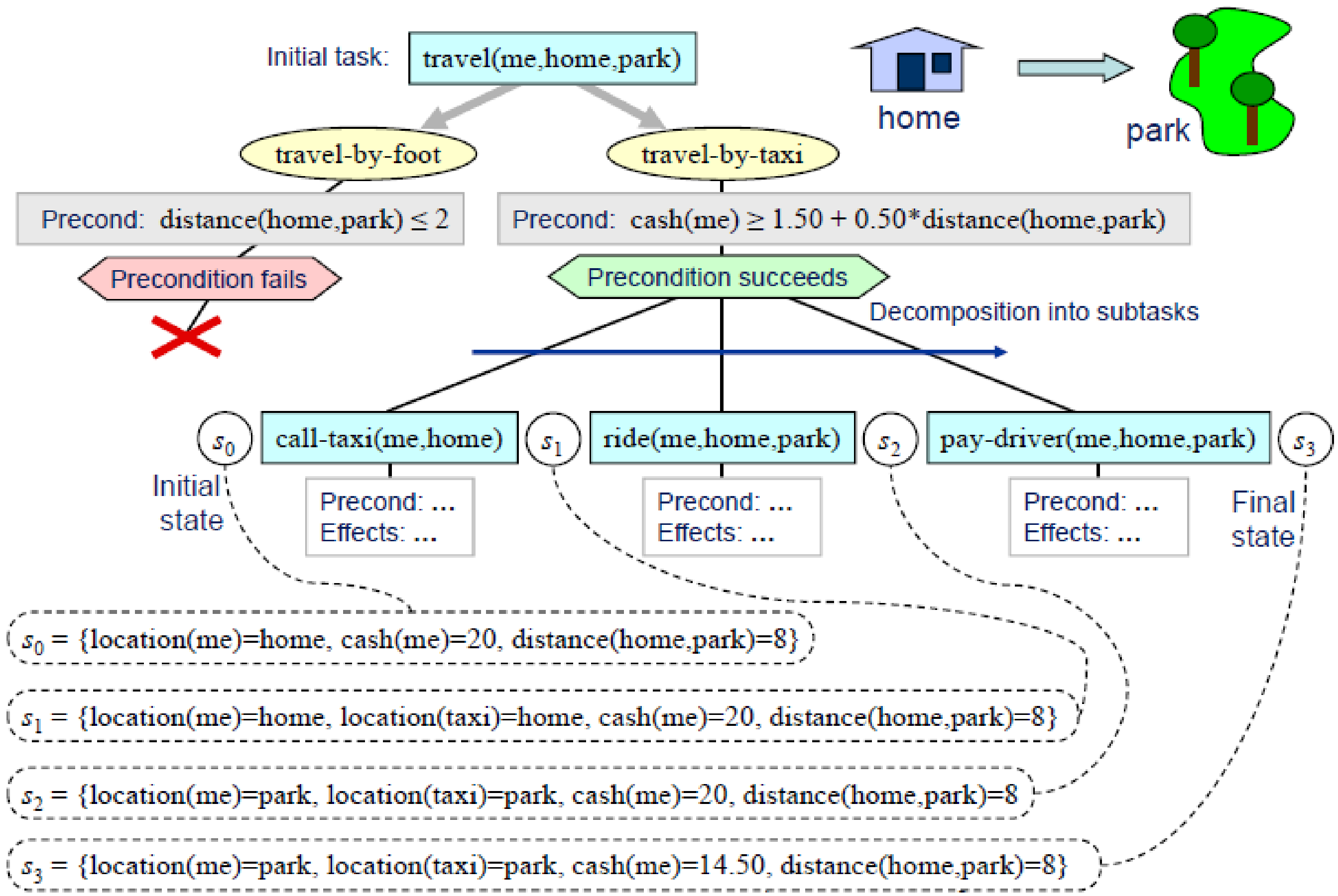
Identify and commit to high-level plans that work while avoiding high-level plans that don't.

MAKING-PROGRESS checks to make sure that we aren't stuck in an infinite regression of refinements

Example

Planning Problem:

I am at home, I have $20,
I want to go to a park 8 miles away



Reasoning about resources

- Introduce numeric variables that can be used as *measures*
- These variables represent resource quantities, and change over the course of the plan
- Certain actions may produce (increase the quantity of) resources
- Other actions may consume (decrease the quantity of) resources
- More generally, may want different types of resources
  - Continuous vs. discrete
  - Sharable vs. nonsharable
  - Reusable vs. consumable vs. self-replenishing

Other real-world planning issues

- Conditional planning
- Partial observability
- Information gathering actions
- Execution monitoring and replanning
- Continuous planning
- Multi-agent (cooperative or adversarial) planning

Planning summary

- **Planning representations**
  - Situation calculus
  - STRIPS and PDDL representation: Preconditions and effects
- **Planning approaches**
  - State-space search (STRIPS, forward chaining, backward chaining)
  - Plan-space search (partial-order planning, HTN)
  - Constraint-based search (GraphPlan, SATplan)

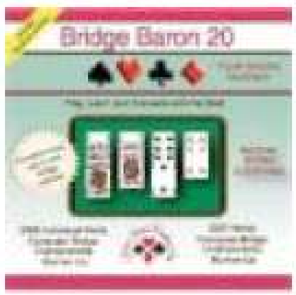
Summary

- Problem solving
  - Atomic representations of states
- Planning combines search and logic
  - Problem solving algorithms that operate on explicit propositional or relational representations of states and actions.
- PDDL describes the initial and goal states as conjunctions of literals, and actions in terms of their preconditions and effects.
- State space planning performs forward or backward search on the state space
  - Progression planners choose **applicable** actions
  - Regression planners choose **relevant** actions
- A planning graph encodes constraints on possible plans which can be used to constrain the search for a valid plan

Summary

- Scheduling
  - Representing and solving planning problems that include temporal and resource constraints
  - Temporal scheduling with critical path method is an easy problem
  - Resource constraints
    - Adding disjunctions makes scheduling with resource constraints NP-hard
- HTN Plannning
  - Plan space planning
  - Library of HLAs
  - Finding a solution plan
    - Search among the implementations for one that works
    - Reason directly about the HLAs
      - Preconditions and effects

Applications



Games

Military Logistics

Manufacturing

Robots

Autonomous Spacecraft

Multiagent Planning

- Planning with multiple agents
  - Each agent makes its plan
  - Joint actions
    - <a_1, …, a_n) where a_i is the action taken by the *ith* actor
    - Transition model and joint planning problem
      - Complexity of the problem grows exponentially
  - Loosely coupled agents
  - Goals and knowledge base might or might not be shared
    - Can each agent just compute the joint solution and execute its own part?
      - There is no right single joint solution

The doubles tennis problem

- Class

$Actors(A, B)$
$Init(At(A, LeftBaseline) \wedge At(B, RightNet) \wedge$
$\quad Approaching(Ball, RightBaseline)) \wedge Partner(A, B) \wedge Partner(B, A)$
$Goal(Returned(Ball) \wedge (At(a, RightNet) \vee At(a, LeftNet))$
$Action(Hit(actor, Ball),$
$\quad \text{PRECOND:} Approaching(Ball, loc) \wedge At(actor, loc)$
$\quad \text{EFFECT:} Returned(Ball))$
$Action(Go(actor, to),$
$\quad \text{PRECOND:} At(actor, loc) \wedge to \neq loc,$
$\quad \text{EFFECT:} At(actor, to) \wedge \neg At(actor, loc))$

Figure 11.10 The doubles tennis problem. Two actors A and B are playing together and can be in one of four locations: $LeftBaseline$, $RightBaseline$, $LeftNet$, and $RightNet$. The ball can be returned only if a player is in the right place. Note that each action must include the actor as an argument.

A: [Go(A, RightBaseline), Hit(A, Ball)]
B: [NoOp(B), NoOp(B)]

A: [Go(A, LeftNet), NoOp(A)]
B: [Go(B, RightBaseline), Hit(B, Ball)]

Multiagent Planning

- Each agent makes its own plan
- Agents are loosely coupled
- Agents need
  - Synchronization
  - Cooperation
    - Conventions
    - Social Laws
  - Coordination
    - Communication (implicit or explicit)
  - Negotiation

Multiagent Planning in practice

Team Work

- Cooperative maneuvering

- Collaborative maneuvering

Air Traffic Management: A Collaborative Multi-Agent System

http://www.natca.org/flight-explorer/united-states.aspx

Multiagent planning

Some practical problems studied in research

- **Target tracking**

  This problem was inspired by a DARPA-sponsored project we, along with MAS researchers from other universities, worked on. The problem involves a set of radars laid out in a field and their need to coordinate in order to track some moving targets.
- **The Mailmen Problem**

  The mailman problem is an instance of the task allocation problem which is discussed in "Introduction to Multiagent Systems" by Mike Wooldridge, Chapter 7.3.1.
  - n deliverators (mailmen), k letters to deliver to m locations
- **Incentive Compatible Package Delivery**

  The basic problem is that we have a number of agents (known as "deliverators"), each one is assigned a number of deliveries. The deliverators try to get some other unsuspecting deliverator to do the next task that they have to deliver for them.