2D1380 Artificial Intelligence

PLANNING

Danica Kragic

September 20, 2005

Acting logically

- So far: search-based planning and logical planning
- Problem-solving agents are able to plan before acting
- Knowledge-based agents can select actions based on explicit, logical representation of the current states and effects of actions
- How do we augment these to complex planning problems?

From problem solving to planning

- Difference:

- representation of goals, states and actions

- the construction of action sequences

- A problem-solving agent:
  - Actions generate successor state descriptions
  - State representations for successor generation, heuristic eval function and goal testing
  - Goals used in the goal test and the heuristic function

Problems with a search agent ?

-
-
-

Problems with a search agent

- Too many actions and too many states to consider
- Heuristic functions choose only among states and cannot eliminate actions from considerations: So, which action should be taken?
- Agent is forced to consider actions starting from the initial state

Solutions ....

- Different representation of states, goals, actions:
  - states and goals represented by sets of sentences
  - actions represented by logical descriptions of preconditions and effects
  - **Direct connections between states and actions**
- A planner is free to add actions when needed: Important decisions can be made before less important ones
- For complex problems: different parts of problems are commonly independent of each other and can therefore be solved independently

Classical Planning Environments

- Fully observable
- Deterministic outcome
- Finite steps
- Static (change happens only due to action)
- Discrete in time, action, objects and effects
- Non-classical during the next hour .....

Planning

Let us start by

looking at an everyday problem

that is too difficult

to solve by search or inference techniques:

Grocery Shopping example

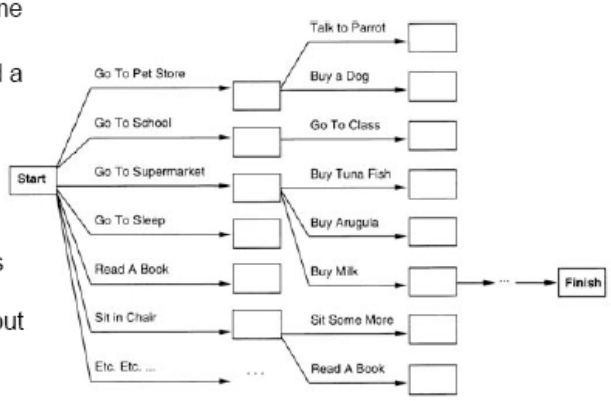
Get milk, bananas, and a cordless drill from a supermarket

Initial state: at home

Goal state: Get milk, bananas and a cordless drill

Depending on the set of actions… branching factor can be huge

Heuristic functions can guide the search of states, but cannot eliminate states



Analyzing Grocery Shopping example

- initial state: agent is at home
- actions: all of the things the agent can do: watch tv, sleep, go jogging,...
- assume that an informed search technique is used – **heuristic function: minimize the number of items we have not yet acquired**
- How do we choose which action is best? branching factor could be enormous
- even if we get to supermarket, still have to consider all the possible choices of which item to buy

Planning

Why is search inadequate here?

Planning

Why is search inadequate here?

- complex state descriptors (a house with all the furniture, objects, etc.)
- large amount of possible actions (human or robot moving)
- usually combined goals (example in R&N: buying 4 books online)

Planning - and back to grocery

Problem is difficult because:

agent must consider action sequences starting from initial state -

Before we can purchase anything, have to get to supermarket!

but the agent (using a search technique) does not know this.

Planning requires explicit knowledge!

Planning =
find a sequence of actions that achieves a goal

Solutions I

- New representation of actions:
  - So far, the only way of knowing if an action is suitable was to execute it and examine the resulting state
  - Now, we want to explicitly represent the preconditions and effects of actions
- Order of planning $\neq$ order of execution
  - We can plan how to *pay(bread)* before planning how to *get(supermarket)*

Solutions II

- Divide-and-conquer:
  An efficient planner should consider **problem decomposition**- it should work on subgoals independently
- No free lunch .....
  How to combine the resulting subplans?

Language of planning problems

- We have: states, actions and goals
- We need a formal representation language
  - expressive enough to describe wide variety of problems
  - restrictive enough to allow for efficient execution
- Basic representation language of classical planners - STRIPS

Before starting with STRIPS .....

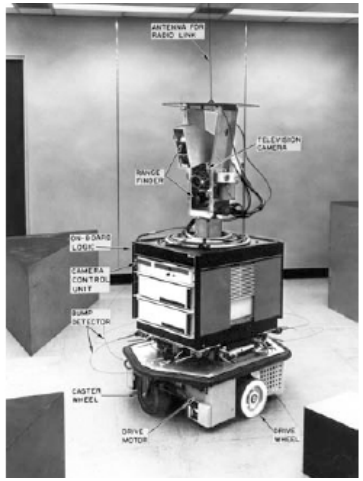
How is this different from problem-solving via search?

- states, goals, and actions use representations in some formal language
- we can look inside actions:
  - with search, actions defined only by state transitions
  - with planning, describe actions by:
    - preconditions: what must be true before an action can be performed
    - effects (postconditions): what must be true (what has changed) after action is performed
  - this allows planning systems to reason about interaction between different actions
- we can consider actions in any order
  - with search, only consider continuous sequences of actions from initial state to goal
  - with planning: we can add steps incrementally, regardless of their final position in the sequence

Back to Grocery Shopping example

- we know that buying the milk is a precondition of having the milk, so agent won't waste any time considering other unnecessary purchases
- since we can add steps in any order, agent can decide that it needs to buy milk before figuring out where to buy it (and how to get there and what to do afterwards)
- this allows agent to make important decisions first, and this helps reduce the branching factor for later choices
- we know that getting to the grocery store is a precondition of buying the milk, so agent will choose that action rather than going for a jog

STRIPS



- STRIPS: Stanford Research Institute Planning System (1970)
- a restricted language (essentially propositional logic) used in most classical planners
- First major planning system - planner for the Shakey robot

STRIPS

- **State** - represented by conjunction of positive literals,

- state description is complete: anything unspecified is assumed false

  Cold ∧ *Dark* - state of a certain country (propositional literals)
  In(Dani, E3) ∧ *In(Student1, E3)* ∧ *In(Student2, E3)* (first-order literals)

- **Goal** - similar to the state, represented by conjunction of positive ground literals
- **Action** - specified by *preconditions* that must hold before it can be executed and *effects* that it produces

- the effect sometimes divided into the **add list** for positive literals and the **delete list** for negative literals

- **Add**: What new facts become true after the action is executed (add to KB)

- **Remove**: What facts become false after the action is executed (remove from KB)

STRIPS - action example

Bringing my laptop from A to B

Action(Bring(p, from, to),
PRECOND: *At(p, from)* ∧ *Laptop(p)* ∧ *Room(from)* ∧ *Room(to)*
EFFECT: ¬ *At(p, from)* ∧ *At(p, to))*

example: go(l1,l2)

* preconditions: at(l1)
* add list: at(l2)
* delete list: at(l1)

30 sec breathing .....
Grocery Shopping Example

- State:
- Goal:
- Action:

30 sec breathing .....
Grocery Shopping Example

- State: *At(Home)* or
- State: *At(Home)* ∧ ¬*Have(Milk)* ∧ ¬ *Have(Bananas)* ∧ ¬*Have(Cordless Drill)* ∧ ...
- Shopping goal: *At(Home)* ∧ *Have(Milk)* ∧ *Have(Bananas)* ∧ *Have(Cordless Drill)*
- Action: *Action(Walk(p, from, to)*,
  PRECOND: *At(p, from)* ∧ *Person(p)* ∧ *Home(from)* ∧ *Store(to)*
  EFFECT: ¬ *At(p, from)* ∧ *At(p, to))*

STRIPS example

- Initial state:
  - [at(robot,living_room), at(beer,kitchen), at(fred,living_room), door_closed(kitchen,living_room)]
- Target/Goal state:
  - [at(beer,living_room), at(fred,living_room), at(robot, living_room), door_closed(kitchen,living_room)]

Operator	Precondition	Postconditions	
		Add	Delete
open(R1,R2)	at(robot,R1) door_open(R1,R2)	door_open(R1,R2)	door_closed(R1,R2)
close(R1,R2)	at(robot,R1) door_closed(R1,R2)	door_closed(R1,R2)	door_open(R1,R2)
move(R1,R2)	at(robot,R1) door_open(R1,R2)	at(robot,R2)	at(robot,R1)
carry(R1,R2,O)	door_open(R1,R2) at(robot,R1) at(O,R1)	at(robot,R2) at(O,R2)	at(robot,R1) at(O,R1)

STRIPS

- An action is applicable in any state that satisfies the precondition; else the action has no effect
- STRIPS assumption: Every literal not mentioned in the effect remains un-changed
- STRIPS is not expressive enough for some real domains
- Different language variants have been developed: ADL (Action Description Language) - Fig.11.1 gives a comparison

Different planners

- Once description available, planning can be done
- Progression vs. regression planners (state-space search)
- Partial order planning (incomplete planning)

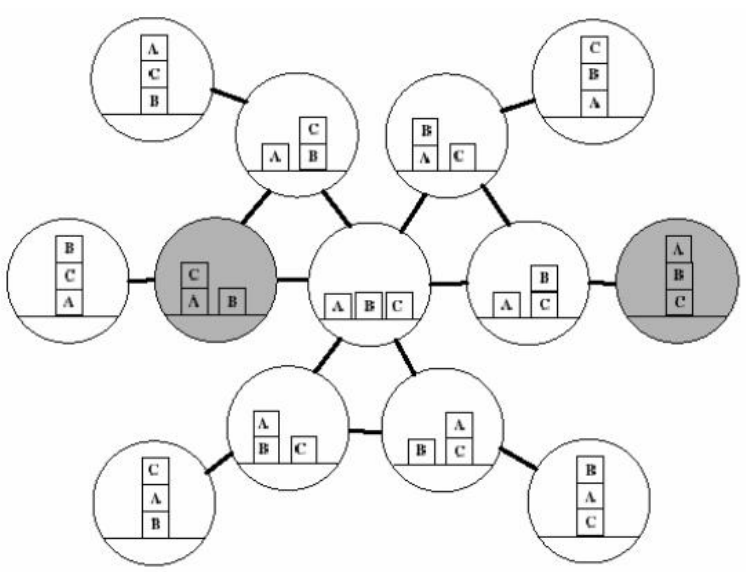
State-space search (Situation Space Planners)

- Searching through a space of situations
- Remember: both preconditions and effects of action given
- Similarly to forward and backward chaining (data driven and goal driven reasoning) procedures mentioned for logical inference, there are forward and backward algorithms for planning
- Usually impossible to solve real ”real-world” problems using the method of state space search.
- SSS requires *complete* description of the states searched and requires the search to be carried out *locally*

SSS

- Completely describing each state that is expanded during a search: computationally expensive when real world problems are involved
- The local property of state space search refers to the fact that search can proceed only by expanding states that are directly reachable from the current state

Blocks World



Two approaches to search

- Forward state-space search - progression planning
- Start with initial state, **choose actions (How?)**, goal test
- Backward state-space search - regression planning
- Start with goal state, apply only **relevant** actions

Forward state space search - Progression Planners

- search forward from initial state
- determine which actions apply using preconditions
- use add/delete lists to compute new state
- main problem: often have huge search space because of large branching factor

Backward state-space search - Regression Planners

- search backwards from goal state to initial state
- note: we must deal with partial descriptions of the state since we do not yet know what actions will get us to goal
- in typical problems, goal state has a small number of conjuncts, each of which only has a few appropriate actions
- note: we have to be careful not to take actions that undo the desired literals
- searching backwards is complicated because we have to achieve a conjunction of goals (one alone is easy)

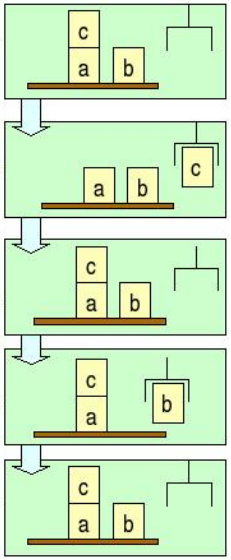
Quick Review of the Blocks World

unstack(x,y)
Pre: on(x,y), clear(x), handempty
Eff: ~on(x,y), ~clear(x), ~handempty, holding(x), clear(y)

stack(x,y)
Pre: holding(x), clear(y)
Eff: ~holding(x), ~clear(y), on(x,y), clear(x), handempty

pickup(x)
Pre: ontable(x), clear(x), handempty
Eff: ~ontable(x), ~clear(x), ~handempty, holding(x)

putdown(x)
Pre: holding(x)
Eff: ~holding(x), ontable(x), clear(?x), handempty

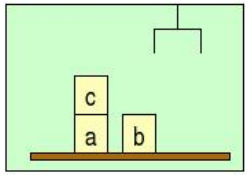


Progression and regression planners

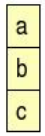
- Both progression and regression planners tend to have difficulty with complex problems. An alternative is to search through a space of plans.
- Work in a linear fashion, adding steps that build on their immediate successors or predecessors.
- A better alternative is the use of partial plans, in which ordering constraints are imposed as needed at execution time.

Sussman anomaly

- Exercise 11.11 in the book
- A non-interleaved planner: given two subgoals G_1 and G_2, it produces either a plan for G_1 concatenated with a plan for G_2 or vice-versa.
- On the below problem, STRIPS based approach cannot solve it. Why?



Initial state



goal

Planning order

- Forward and backward state-space search are particular forms of totally ordered plan search - strictly linear sequences of actions directly connected to the start or goal
- Problem decomposition is not facilitated: We would like a planner that solves important problems first
- Delaying choice during search - *least commitment*: only necessary choices are made

Partial-Order Planner

Any planner that can place two actions in a plan without specifying which comes first.

Partial-Order Planner: Definitions

Each plan has the following components:

- *plan steps* (actions)
- *ordering constraints*: $A \prec B$ (A executed **sometimes** before B)
- *Causal links*: A **achieves** p for B

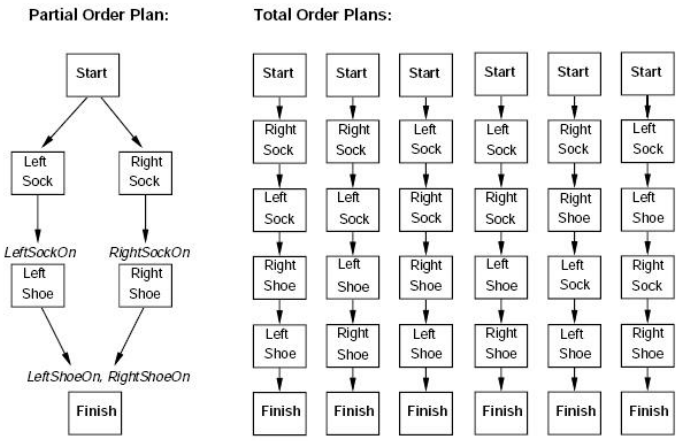
$$A \xrightarrow{p} B$$

$$BuyEggs \xrightarrow{HaveEggs} MakeOmelette$$

means that $HaveEggs$ is an effect of the $BuyEggs$ and a precondition of $MakeOmelette$

- **Conflict** occurs if there is an action C that has an effect $\neg p$ and, according to the *ordering constraints* comes after A and before B
- *open preconditions* - a precondition is open if it is not achieved by some action in the plan

Partial and Total Order Plans for Putting on shoes and socks



Partial-Order Planner: Definitions

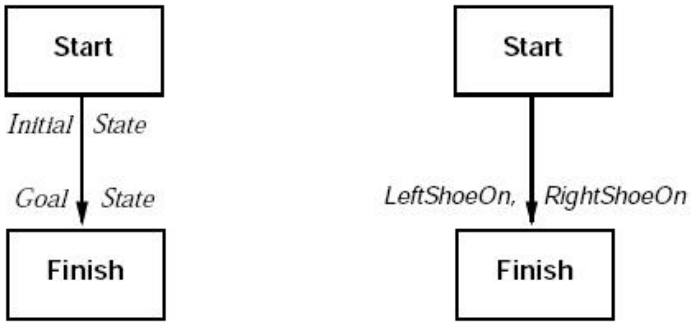
Each plan has the following components:

- A plan is *complete* if every precondition of every step is achieved by some other step
- A plan *achieves* a condition if the condition is one of the effects of the step
- A plan is *consistent* if there are no contradictions in the ordering or binding constraints

Formulation for POP

- The initial plan:
- $Start$ and $Finish$,
- ordering constraint $Start \prec Finish$
- no casual links
- all preconditions in $Finish$ are open preconditions
- Successor function arbitrarily picks one open precondition p on an action B and generates a successor plan for every possible consistent way of choosing an action A that achieves p

Example: Initial plan for Putting on shoes and socks



Formulation for POP

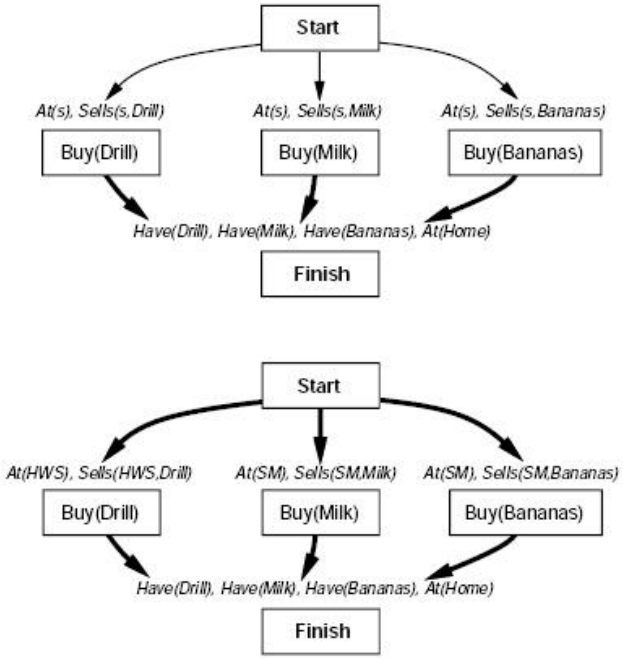
- Consistency enforced by
- Causal link $A \xrightarrow{p} B$ and the ordering constraint $A \prec B$ added to the plan. If A is a new action, add it to the plan and also add $Start \prec A$ and $A \prec Finish$
- Resolve conflict between new causal links and all existing actions (by, for example, making some action C happening outside the protected period defined by $A \xrightarrow{p} B$)
- Checking if the current plan is a solution to the original planning problem: Goal test
- since only consistent plans are generated, check that there are no open preconditions

Partial-Order Planning

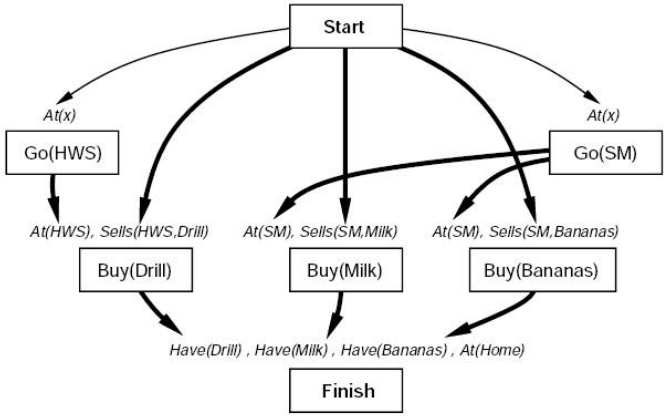
- search through plan space rather than situation space (state of the world)
- start with an initial plan consisting of the start and finish steps
- at each iteration, add one more step
- if this leads to an inconsistent plan, backtrack and try another branch of search space
- only add steps that achieve a precondition that has not yet been achieved

POP Example: Initial Plan

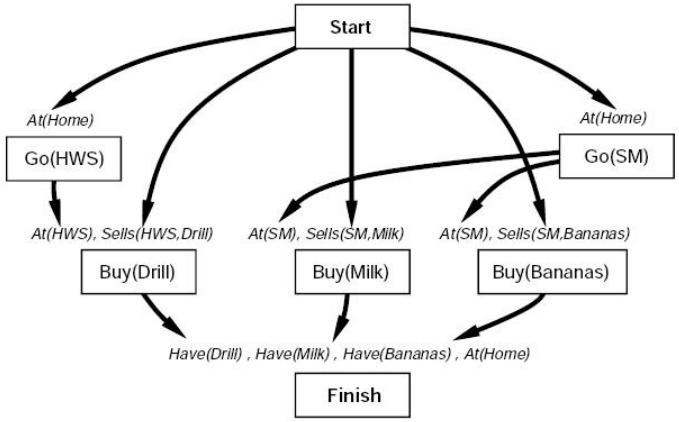
A Partial Plan I



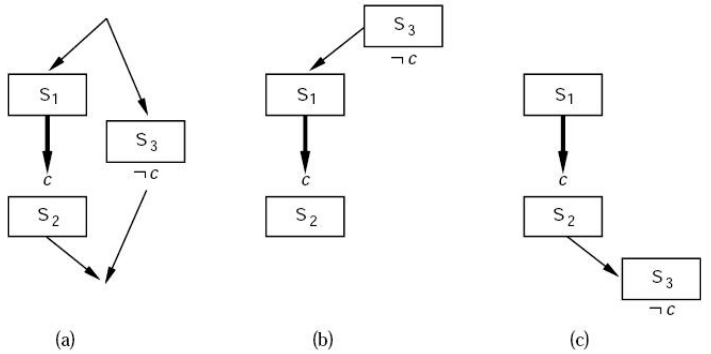
A Partial Plan II



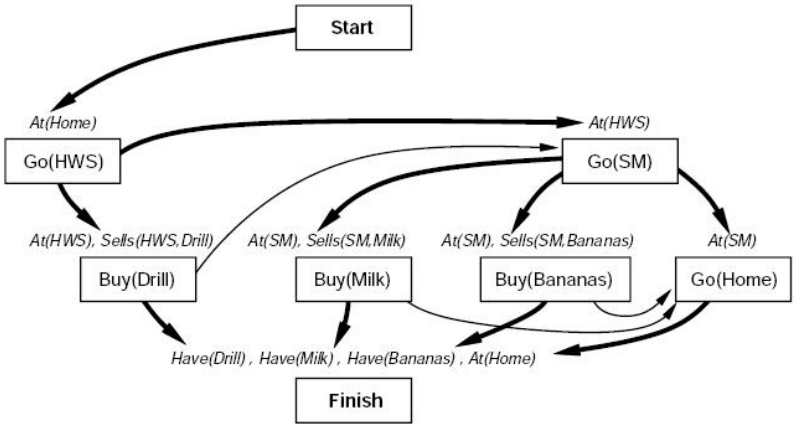
A Partial Plan III



Protecting causal Links



A Partial Plan III



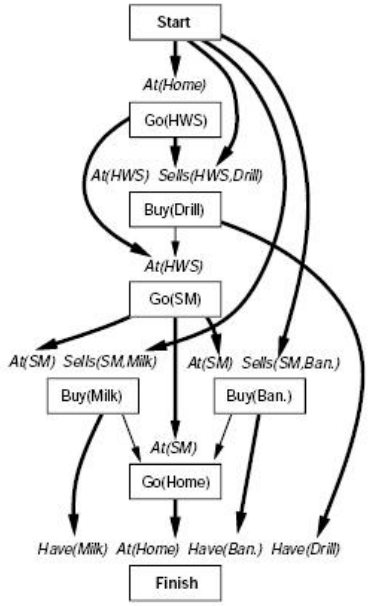
Achieving *At(Home)*

Achieving At(Home)

Candidate link	Threats
At(x) to initial state	Go(HWS), Go(SM)
At(x) to Go(HWS)	Go(SM)
At(x) to Go(SM)	At(SM) preconds of Buy(Milk), Buy(Bananas)

Solution: Link At(x) to Go(SM), but order Go(Home) to come after Buy(Bananas) and Buy(Milk).

A Final Plan



Strength of POP

- Takes a huge state space problem and solves in only a few steps.
- Least commitment strategy means that search only occurs in places where sub-plans interact.
- Causal links allow planner to recognize when to abandon a doomed plan without wasting time exploring irrelevant parts of the plan.

Practical planners

STRIPS approach is insufficient for many practical planning problems.

- Cannot express:
  - resources: Operators should incorporate resource consumption and generation. Planners have to handle constraints on resources efficiently.
  - time: Real-world planners need a better model of time.
  - hierarchical plans: Need the ability to specify plans at varying levels of detail.
- Also need to incorporate heuristics for guiding search.

Planning Graphs

- data structure (graphs) that represent plans and can be efficiently constructed
- planning graphs allow for better heuristic estimates
- **GRAPHPLAN**: algorithm that processes the planning graph, using backwards search, to extract a plan.
- **SATPlan**: algorithm that translates a planning problem into propositional axioms and applies a CSP algorithm to find a valid plan.

POP and GRAPHPLAN

- POP deals with negative interactions between actions in problem decomposition using causal links: however, each conflict must be resolved with a choice (put the conflicting action before or after the link) and choices can multiply exponentially
- GRAPHPLAN: avoids the choices during the graph construction phase, using *mutex links* to record conflicts without making a choice of how to resolve them

Russell and Norvig:Example

$Init(Have(Cake))$
$Goal(Have(Cake) \wedge Eaten(Cake))$
$Action(Eat(Cake)$
PRECOND: $Have(Cake)$
EFFECT: $\neg Have(Cake) \wedge Eaten(Cake))$
$Action(Bake(Cake)$
PRECOND: $\neg Have(Cake)$
EFFECT: $Have(Cake)$

Figure 11.11 The "have cake and eat cake too" problem.

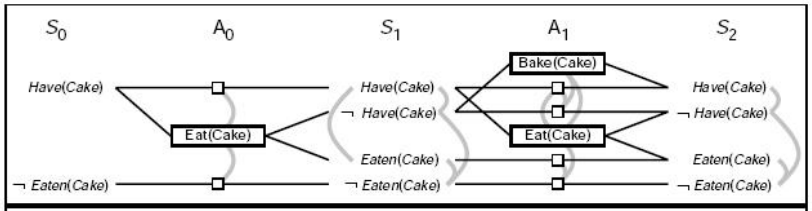


Figure 11.12 The planning graph for the "have cake and eat cake too" problem up to level S_2. Rectangles indicate actions (small squares indicate persistence actions) and straight lines indicate preconditions and effects. Mutex links are shown as curved gray lines.

Planning graph

- **State level** S: initial, goal or current state
- **Action level** A: all the actions for which the preconditions are satisfied in the previous level (connecting again preconditions and effects)
- **Persistence actions**: leave state unaltered
- **Mutex links**: connect mutually exclusive actions

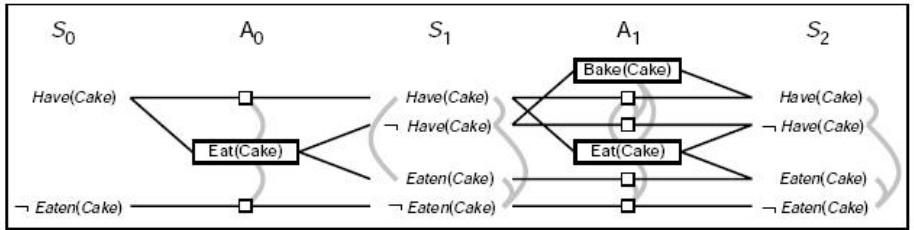


Figure 11.12 The planning graph for the "have cake and eat cake too" problem up to level S_2. Rectangles indicate actions (small squares indicate persistence actions) and straight lines indicate preconditions and effects. Mutex links are shown as curved gray lines.

Defining mutex links for actions

- *Inconsistent effects*: one action negates an effect of the other ($Eat(Cake)$ and $Have(Cake)$)
- *Interference*: one of the effects of one action is the negation of a precondition of the other ($Eat(Cake)$ interferes with the persistence of $Have(Cake)$ by negating its precondition)
- *Competing needs*: one of the preconditions of one action is mutually exclusive with a precondition of the other ($Bake(Cake)$ and $Eat(Cake)$ are mutex since they compute on the value of precondition $Have(Cake)$)

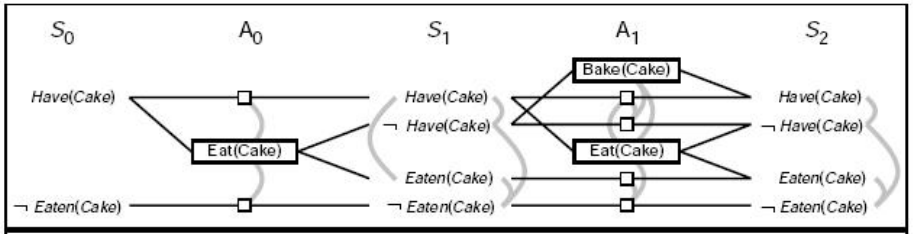


Figure 11.12 The planning graph for the "have cake and eat cake too" problem up to level S_2. Rectangles indicate actions (small squares indicate persistence actions) and straight lines indicate preconditions and effects. Mutex links are shown as curved gray lines.

Defining mutex links for literals

- *Inconsistent support*: if each possible pair of actions that could achieve the two literals is mutually exclusive
- A mutex between two literals at the same level holds if one is the negation of the other

- Note! $Have(Cake)$ and $Eaten(Cake)$ in S_1 and S_2

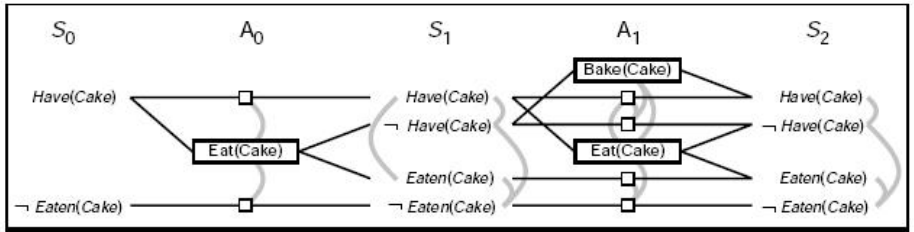


Figure 11.12 The planning graph for the "have cake and eat cake too" problem up to level S_2. Rectangles indicate actions (small squares indicate persistence actions) and straight lines indicate preconditions and effects. Mutex links are shown as curved gray lines.

GRAPHPLAN

- Extracts plan from a planning graph
- It alternates between two phases: *graph expansion* and *solution extraction*
- *solution extraction*: Are all the goal literals present in the current level with no mutex links between any pair of them?
- *graph expansion*: adding the actions for the current level and the state literals for the next level
- This continue until solution found or no solution exists

2D1380 Artificial Intelligence

PLANNING AND ACTING IN THE REAL WORLD

Danica Kragic

September 20, 2005

Planning and Scheduling

- 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**
- (the above also known as **scheduling**)
- **Conditional** outcomes of actions
- **Uncertain** outcomes of actions
- **Incremental** plan development
- **Dynamic** real-time replanning

Time, schedules, resources

- Time essential for **job shop scheduling** applications
- Complete a set of jobs that consist of an action sequence
- Find a schedule that minimizes the total execution time

Scheduling vs planning

- STRIPS reps allow only to decide **relative ordering** among planning actions
- In scheduling, we also decide the **absolute**:
  - Time when an event/action will occur
  - Its duration

Job shop scheduling problem

Given: K jobs to do

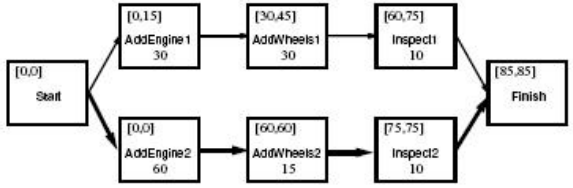
- job=fixed sequence of actions
- have quantitative time durations
- use resources (which can be shared among jobs)

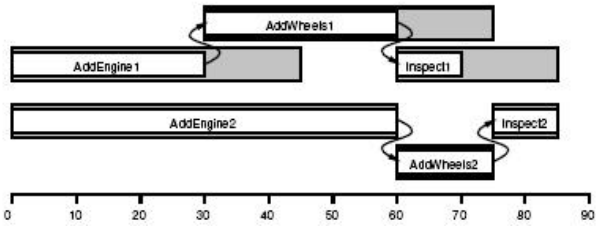
Find: A schedule that

- Determines a start time of each action
- Obeys constraints such as resource sharing
- Minimizes the **total time** to perform all jobs

Example: Car Assembly Scheduling

- 2 jobs, job=3 actions, resource sharing





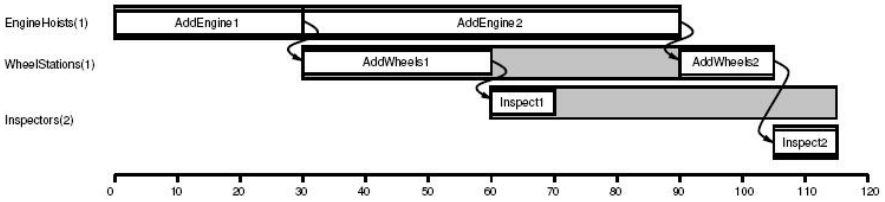
Example: Car Assembly Scheduling

1 Create a partially-ordered plan

2 Use critical path method to determine the schedule

- **Critical path**: longest sequence of actions, each of which has no slack (i.e. if not started on time, delays whole plan)

- with and without resources



Planning and Scheduling

- **Planning**: given one or more goals, generate a sequence of actions to achieve the goal(s)
- **Scheduling**: given a set of actions and constraints, allocate resources and assign times to the actions so that no constraints are violated
- Traditionally, planning is done with specialized logical reasoning methods
- Traditionally, scheduling is done with constraint satisfaction, linear programming
- However, planning and scheduling are closely interrelated and can't always be separated

Planning and Scheduling

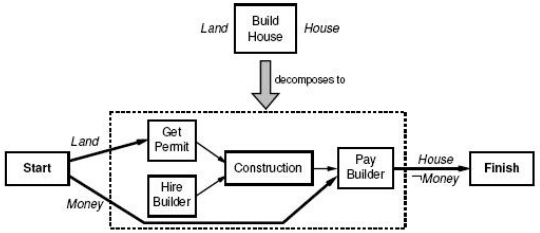
1 **Serial**: first plan, then schedule (POP for planning, the use the plan to determine starting time)

2 **Interleaved**: Mix planning and scheduling - include resource constraints during partial planning

- Note! The STRIPS representation of action has to be extended in order to include the resources - action(name, precond, effect, resource)

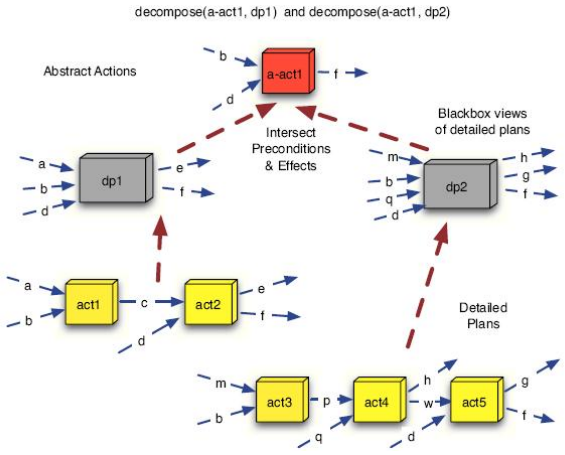
Hierarchical planning

- **Hierarchical decomposition**: express action at one level as a small set of actions at the next level
- **Hierarchical Task Network (HTN) Planning**: start with an abstract plan and continue expanding each activity until only **primitive action** exist

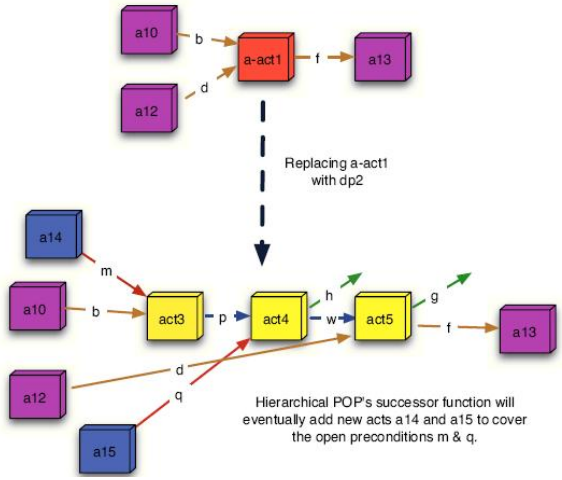


Plan libraries and Hierarchical POP

- Pick an action to fulfill and open precondition
- Expand and abstract action with one of the decompositions from the plan library



Expanding an abstract action



Planning and acting in nondeterministic domains

- Classic env: fully observable, deterministic, static, discrete
- Real world and our knowledge about it:
  - incorrect: some things we know are wrong
  - incomplete: we do not know everything
- Having correct and complete knowledge depends on the level of **indeterminacy** of the world
  - **Bounded indeterminacy**: unexpected effects can be enumerated, conditional POP can handle it
  - **Unbounded indeterminacy**
    - In complicated cases, no complete enumeration is possible
    - Plan for some contingencies
    - Replan for the rest

Planning and acting in nondeterministic domains

- Sensorless planning (Conformant planning): relies on **coercion** - world can be forced into a given state
- Conditional planning (Contingency planning):
  - first plan than act
  - include sensing actions in the plan
- Execution monitoring
  - monitoring what is happening while it executes the plan
  - telling when things go wrong and replan
  - replanning: finding a way to achieve its goals from the new situation (something went wrong according to old plan)
- Continuous planning
  - persist over lifetime (e.g., Mars rovers)

Painting chair and table

- Init: a chair, a table, cans of paints with unknown color

  Goal: the chair and table have the same color
- Different types of planning
- Classical planning: should be fully observable, cannot solve the problem (initial state not fully specified)
- Sensorless planning: coercing
- Conditional planning with sensing: (1) already the same, (2) one painted with the available color, (3) paint both
- Replanning: paint, check the effect, replan for missing spot
- Continuous planning: paint, can stop for unexpected events, continue

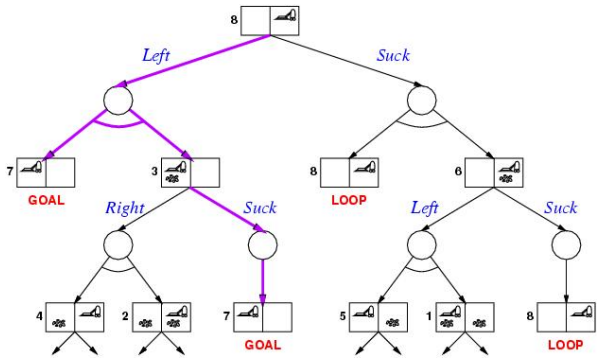
Conditional planning I

CP in fully observable environments (FOE)

- way to deal with uncertainty by checking what is happening in the environment at predetermined points in the plan
- full observability: the agent knows the current state
- nondeterministic env: the outcome of actions cannot be predicted

Conditional planning I

- Murphy's vacuum world with actions Left, Right, and Suck
- Plan needs to take *some* actions at every state and must handle *every* outcome for the action it takes
- State nodes (squares) and chance nodes (circles)
- chance nodes: every outcome must be handled
- state nodes: some action must be chosen



Conditional planning I

- Disjunctive effects (allow nondeterminism): if Left sometime fails, then
  $Action(Left, Precond: AtR, Effect: AtL \vee AtR)$
- Conditional effects (effect depends on the state action was executed in):
  $Action(Suck, Precond:, Effect: (when\ \ AtL: CleanL) \wedge (when\ \ AtR: CleanR))$

  $Action(Left, Precond: AtR, Effect: AtL \vee (AtL \wedge when\ \ CleanL: \neg CleanL))$
- Conditional steps for creating conditional plans:
  $if\ \ test\ \ then\ \ planA\ \ else\ \ planB$
  e.g., if $AtL \wedge CleanL\ \ then\ \ Right\ \ else\ \ Suck$

Conditional planning I

- Conditional plans have to work regardless of what the outcome of an action is
- A modified **minmax algorithm** with AND-OR used to generate complex plans

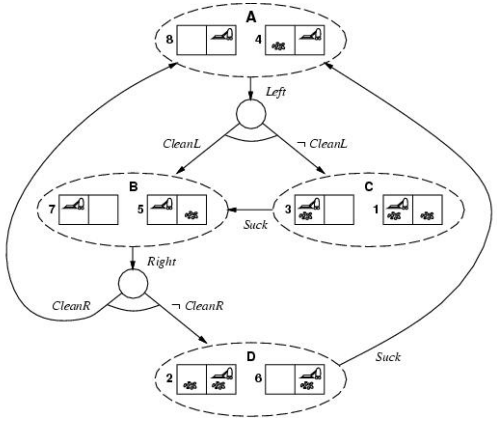
Conditional planning II

CP in partially observable environments (POE)

- Uncertain actions and imperfect sensors
- Initial state is a state set - a belief state
- Sensing is important!
  - Automatic sensing
    - At every step, the agent gets all the available percepts
  - Active sensing
    - Percepts are obtained only by executing specific sensory actions

Conditional planning II

- The vacuum agent is AtR and knows about R, how about L?
- Dirt can sometimes be left behind when the agent leaves a clean square



Replanning via monitoring

- In reality, something can go wrong. How can a replanning agent know that?

1. annotate a plan at each step with preconditions required for successful completion of the remaining steps
2. detect a potential failure by comparing the current preconditions with the state description from percepts

- Sensing and monitoring

- Execution monitoring - see what happens when executing a plan:

  Action monitoring

  Plan monitoring

Replanning - Action monitoring

- Before carrying out the *next action* of a plan
- check the *preconditions of each action* as it is executed rather than checking the preconditions of the entire remaining plan
- work well with realistic systems (action failures)
- Return to the chair-table painting problem
  Plan: [Start; Open(BC); Paint(Table,Blue); Finish]
  - What if it missed a spot of green on the table?
- Loop is created by plan-execute-replan, or no explicit loop
- Failure is only detected after an action is performed

Replanning - Plan monitoring

- Detect failure by checking the preconditions for success of the entire remaining plan
- Useful when a goal is serendipitously (*accidently*) achieved
  - While you're painting the chair, someone comes painting the table with the same color
- Cut off execution of a doomed plan and don't continue until the failure actually occurs
  - While you're painting the chair, someone comes painting the table with a different color

If one insists on checking every precondition, it might never get around to actually doing anything. **Why?**

Difference between CP and RP

- Unpainted area will make the agent to repaint until the chair is fully painted.
- Is it different from the loop of repainting in conditional planning?
- The difference lies in the time at which the computation is done and the information is available to the computation process
  - CP - anticipates uneven paint
  - RP - monitors during execution

Conditional Planning and Replanning

- Conditional planning
  - The number of possible conditions vs. the number of steps in the plan
  - Only one set of conditions will occur
- Replanning
  - Fix problems as they arise during execution
  - Fragile plans due to replanning
- Intermediate planning between CP and RP
  - The most likely ones done by CP
  - The rest done by RP

Combining planning and execution

Continuous planning agent

- execute some steps ready to be executed
- refine the plan to resolve standard deficiencies
- refine the plan with additional information
- fix the plan according to unexpected changes
  - recover from execution errors
  - remove steps that have been made redundant

Goal → Partial Plan → Some actions → Monitoring the world → New Goal

Continuous planning: Basics

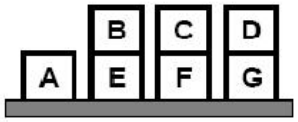
- Always has a plan
  - Plan has a *Start* action representing the current state
  - Plan has a *Finish* action representing top level goals
  - Plan may have flaws that haven't been repaired yet
- Repeat indefinitely:

1. Observe world and upstate the *Start* action
2. Modify the plan to remove a flaw
3. Do an action from the plan or No-op

- New top-level goals added to *Finish* action
  - By user
  - By agent itself, from outside its planner/executor

Continuous Planning - Revisit the blocks world

- Initial state:



- Goal: On(D,B) & On(C, D)
- Action schema:

  Action(Move(x, to),
  PRECOND: Clear(x) & Clear(to) & On(x, from)
  EFFECT: On(x, to) & Clear(from) &
  ~On(x,from) & ~Clear(to)

- Assume
  - partial order planning
  - full observability
  - Unbounded indeterministic actions

Multi-agent planning

- One agent generates a plan for multiple agents
- Cooperation: Each agent generates its own multi-agent plan and they coordinate
- Competition: Each agent generates its own multi-agent plan intended to foil the other(s)