TDT4136 Logic and Reasoning Systems

Chapter 10 & 11 - Planning

Lester Solbakken
`solbakke@idi.ntnu.no`

Norwegian University of Science and Technology

08.11.2011

Outline

1. What is planning?
2. PDDL/STRIPS language
3. State-space planning (progression and regression)
4. Plan-space planning (POP)
5. Planning graphs
6. Planning and acting in the real world

What is planning?

A plan is a collection of actions for performing some task

- E.g., my daughter's birthday celebration
- E.g., a conference participation
- E.g., assembling furniture

There are many programs that help human planners

The goal in AI is to generate plans automatically

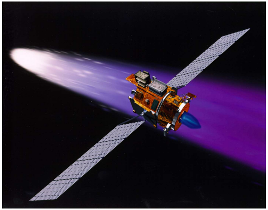
NASA's Deep Space 1

Launched in 1998 to test technologies and perform flybys of asteroid Braille and Comet Borrelly

First spacecraft to be controlled by an AI system without human intervention

Remote Agent (remote intelligent self-repair software) system used plan onboard activities and correctly diagnose and respond to simulated faults

Planning system was later used on the ground-based Mars Exploration Rovers



http://ti.arc.nasa.gov/tech/asr/planning-and-scheduling/remote-agent/

Search vs. planning

A planning problem is described just like a search problem (states, actions/operators, goal), but the problem representation is more structured:

	Search	**Planning**
States	Data structures	Logical sentences
Actions	Code	Preconditions/outcomes
Goal	Goal test	Logical sentence (conjunction)
Plan	Sequence from S_0	Constraints on actions

Two main difficulties arise in more complex search problems:

- Huge branching factor
- Difficulty of finding good heuristic functions

Planning vs. Logic

Planning involves searching over sets of states

We use logic to describe sets of states

Key idea: describe states and actions in propositional logic and use forward/backward chaining to find a plan

STRIPS

Developed at Stanford in early 1970s (Stanford Research Institute Planning System) for the first “intelligent” robot

States are represented as first-order predicates over objects

  Closed-world assumption: everything not stated is false

Actions defined in terms of:

  Preconditions: when can the action be applied?

  Effects: what happens after the actions?

(No explicit description of how the action should be executed)

Goals: conjunctions of literals

STRIPS representations

States are represented as conjunctions:

$In(Robot, room) \land \neg In(Charger, r) \land \ldots$

Goals are represented as conjunctions:

$In(Robot, room) \land In(Charger, r)$ $(\exists r \ \ is\ implicit)$

Actions:

- Name: $Go(here, there)$
- Preconditions: expressed as conjunctions
  $At(Robot, here) \land Path(here, there)$
- Effects (postconditions): expressed as conjunctions
  $At(Robot, there) \land \neg At(Robot, here)$

Variables can only be instantiated with objects of correct type

STRIPS action representation and semantics

Actions have a name, preconditions and effects (or postconditions)

Preconditions are conjunctions of positive literals*

If the preconditon is false in a world state, the action does not change anything (since it cannot be applied)

Effects are represented in terms of:

- Add-list: list of propositions that become true after action
- Delete-list: list of propositions that become false after action

This is a very restricted language, meaning we can do efficient inference!

*PDDL accepts negative literals in preconditions and goals

Example: Buy action

Buying action in STRIPS:

$$\begin{aligned}&\mathit{Action}(\mathit{Buy}(x),\\ &\quad \text{PRECOND} : \mathit{At}(s) \land \mathit{Sells}(s, x, p) \land \mathit{HaveMoney}(p)\\ &\quad \text{DELETE-LIST} : \mathit{HaveMoney}(p)\\ &\quad \text{ADD-LIST} : \mathit{Have}(x))\end{aligned}$$

Buying action in (AIMA version of) PDDL:

$$\begin{aligned}&\mathit{Action}(\mathit{Buy}(x),\\ &\quad \text{PRECOND} : \mathit{At}(s) \land \mathit{Sells}(s, x, p) \land \mathit{HaveMoney}(p)\\ &\quad \text{EFFECT} : \neg\mathit{HaveMoney}(p) \land \mathit{Have}(x))\end{aligned}$$

In general, note that many important details are abstracted away!

Additional propositions can be added to show that now the store has the money, the stock has decreased etc.

Pros and cons of STRIPS

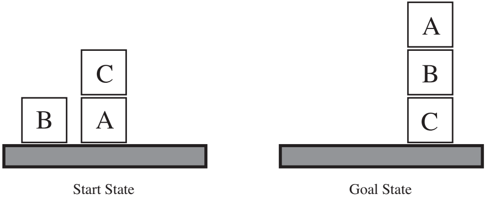
Pros:

- Since it is restricted, inference can be done efficiently
- All actions are additions and deletions of propositions in KB

Cons:

- Assumes only a small number of propositions will change for each action
- Limited language, so not applicable to all domains of interest

Example: Blocks World



Initial state:

$$On(A, Table) \land On(B, Table) \land On(C, A) \land Clear(B) \land Clear(C)$$

Goal state:

$$On(A, B) \land On(B, C)$$

Move action:

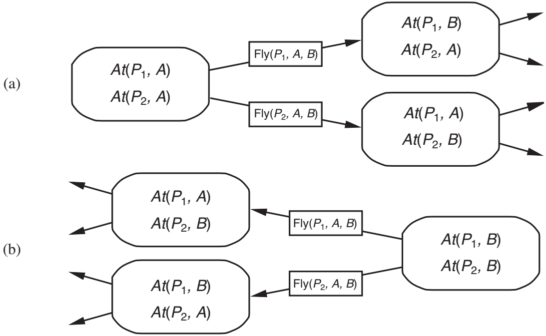
$$\begin{aligned} &Action(Move(b, x, y), \\ &\quad \mathrm{PRECOND} : On(b, x) \land Clear(b) \land Clear(y) \\ &\quad \mathrm{EFFECT} : On(b, y) \land Clear(x) \land \neg On(b, x) \land \neg Clear(y)) \end{aligned}$$

However, there are some problems here. Which?

Two basic approaches to planning

1. State-space planning works at the level of states and actions
   - Finding a plan is formulated as search through state space
   - Most similar to constructive search
2. Plan-space planning works at the level of plans
   - Finding a plan is formulated as search through space of plans
   - Start with partial incorrect plan, then apply changes to correct it
   - Most similar to iterative improvement

State-space planners

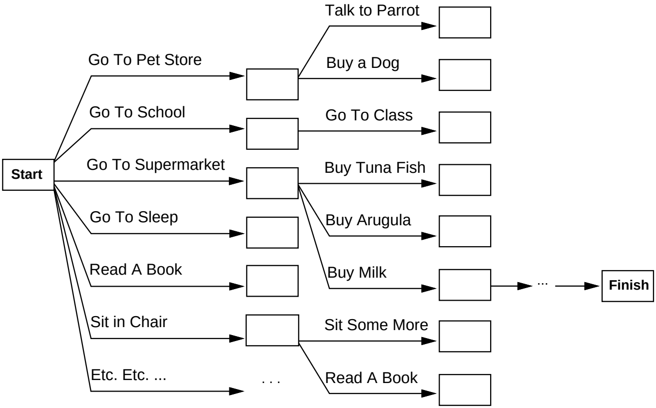


(a) Progression planners reason from the start state, trying to find the actions that can be applied (match preconditions)

(b) Regression planners reason from the goal state, trying to find the actions that will lead to the goal (match effects)

Example: supermarket domain

Consider the task get milk, bananas, and a cordless drill



Progression (forward) planning

Algorithm:

1. Determine all actions that are applicable in the start state
2. Ground the actions, by replacing any variable with constants
3. Choose an action to apply
4. Determine the new content of the knowledge base, based on the action description
5. Repeat until goal state is reached

Example: supermarket domain

Initial state:

$At(Home) \wedge Sells(HardwareStore, Drill)$

Goal state:

$At(Home) \wedge Have(Drill)$

Go action:

$$\begin{aligned}&Action(Go(x, y),\\ &\quad \text{PRECOND}: At(x)\\ &\quad \text{EFFECT}: At(y) \wedge \neg At(x))\end{aligned}$$

Buy action:

$$\begin{aligned}&Action(Buy(x, y),\\ &\quad \text{PRECOND}: At(x) \wedge Sells(x, y)\\ &\quad \text{EFFECT}: Have(y))\end{aligned}$$

In the start state we have $At(Home)$ which enables Go action

The action can be instantiated as $Go(Home, GroceryStore)$, $Go(Home, HardwareStore)$, $Go(Home, School)$...

The new propositions enable new actions, e.g. Buy

Note that in this case there are a lot of possible actions to perform!

Regression (backward) planning

Algorithm:

1. Pick an action that satisfies (some of) the goal propositions.
2. Make a new goal by:
    - Removing the goal conditions satisfied by the action
    - Adding the preconditions of this action
    - Keeping any unsolved goal propositions
3. Repeat until the goal set is satisfied by the start state

Example: supermarket domain

Initial state:

$At(Home) \wedge Sells(HardwareStore, Drill) \wedge Sells(GroceryStore, Milk)$

Goal state:

$At(Home) \wedge Have(Drill) \wedge Have(Milk)$

Go action:

$$
\begin{aligned}
&Action(Go(x, y),\\
&\quad \text{PRECOND} : At(x)\\
&\quad \text{EFFECT} : At(y) \wedge \neg At(x))
\end{aligned}
$$

Buy action:

$$
\begin{aligned}
&Action(Buy(x, y),\\
&\quad \text{PRECOND} : At(x) \wedge Sells(x, y)\\
&\quad \text{EFFECT} : Have(y))
\end{aligned}
$$

In the goal state we have $At(Home) \wedge Have(Drill) \wedge Have(Milk)$

The action $Buy(HardwareStore, Drill)$ enables us to achieve $Have(Drill)$, so we create a new goal by removing this effect and adding the preconditions

Note that in this case the order in which we try to achieve these propositions matters!

Efficiency of state-space search

Backward search has lower branching factor for most domains
- However, difficult to find good heuristics for backward search

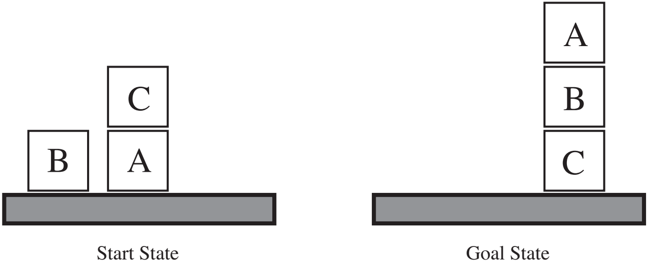
As for CSPs, planning uses factored representations
- Enables good domain-independent heuristics

Heuristics can be derived automatically from the action schema

Sliding-block puzzle example:

$$\begin{aligned}&\mathit{Action}(\mathit{Slide}(t, s_1, s_2),\\ &\quad \text{PRECOND} : \mathit{On}(t, s_1) \land \mathit{Tile}(t) \land \mathit{Blank}(s_2) \land \mathit{Adjacent}(s_1, s_2)\\ &\quad \text{EFFECT} : \mathit{On}(t, s_2) \land \mathit{Blank}(s_1) \land \neg \mathit{On}(t, s_1) \land \neg \mathit{Blank}(s_2))\end{aligned}$$

Sussman Anomaly



Linear planners typically separate the goal (stack A atop B atop C) into subgoals, such as:

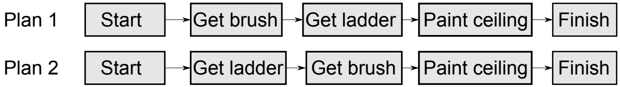
1. get A atop B
2. get B atop C

By removing C from A and moving A atop B we accomplish subgoal 1.

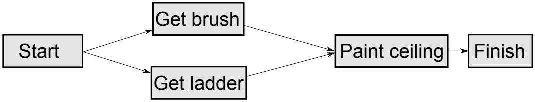
But then we cannot accomplish subgoal 2 without undoing subgoal 1!

Total vs. Partial order

Total order: Plan is always a strict sequence of actions



Partial order: Plan steps may be unordered



Partial Order Planning

Search in plan space and use least commitment whenever possible

In state space search:

- Search space is a set of states of the world
- Actions cause transitions between states
- Plan is a path through state space

In plan space search:

- Search space is a set of partially ordered plans
- Plan operators cause transitions
- Goal is a legal plan

Least commitment: only make choices that are relevant to solving the current part of the problem

Plan operators: add actions, specify orderings, bind variables

POP process

Start with an empty plan consisting of

- *Start* step with the initial state description as its effect
- *Finish* step with the goal description as its precondition

Proceed by

- adding actions to achieve preconditions
- adding causal links from an existing action to achieve preconditions
- order on action w.r.t. another to remove possible conflicts

Gradually move from incomplete/vague plans to complete, correct plans

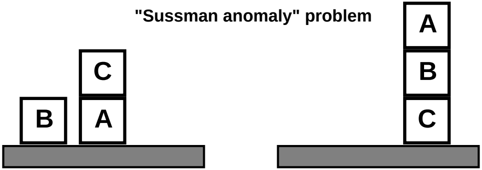
Backtrack if an open condition is unachievable or if a conflict is unresolvable

A plan is complete iff every precondition is achieved

A precondition is achieved iff it is the effect of an earlier step and no possibly intervening step undoes it

Example: Blocks world

"Sussman anomaly" problem



Start State

Goal State

Clear(x) On(x,z) Clear(y)

PutOn(x,y)

~On(x,z) ~Clear(y)
Clear(z) On(x,y)

Clear(x) On(x,z)

PutOnTable(x)

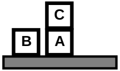
~On(x,z) Clear(z) On(x,Table)

+ several inequality constraints

Example contd.

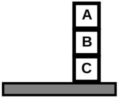
START

On(C,A) On(A,Table) Cl(B) On(B,Table) Cl(C)

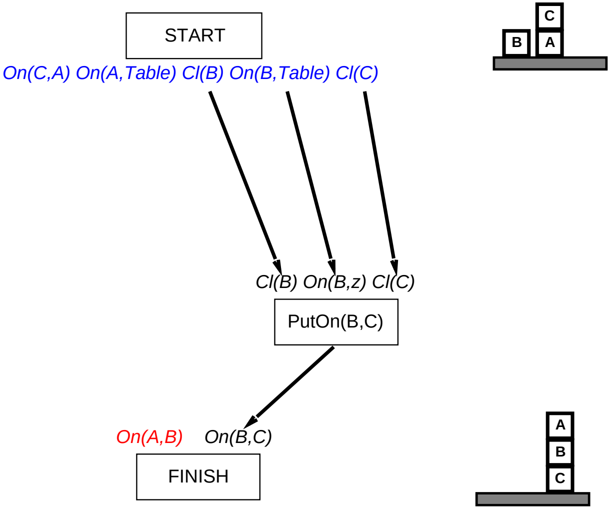


On(A,B) On(B,C)

FINISH



Example contd.



START

On(C,A) On(A,Table) Cl(B) On(B,Table) Cl(C)

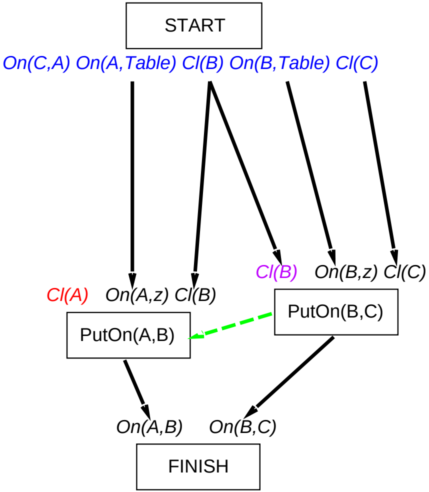
Cl(B) On(B,z) Cl(C)

PutOn(B,C)

On(A,B) On(B,C)

FINISH

Example contd.



START

On(C,A) On(A,Table) Cl(B) On(B,Table) Cl(C)

Cl(A) On(A,z) Cl(B)

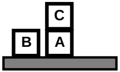
PutOn(A,B)

Cl(B) On(B,z) Cl(C)

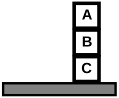
PutOn(B,C)

On(A,B) On(B,C)

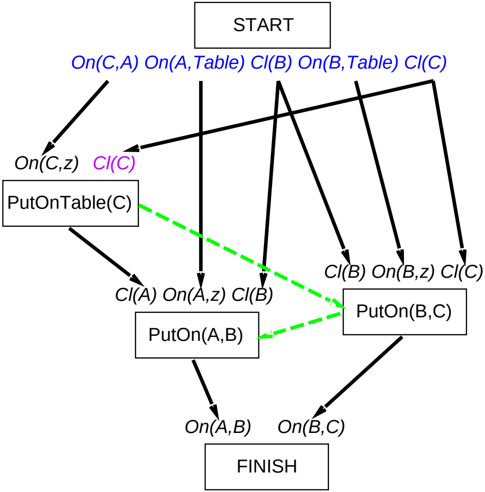
FINISH

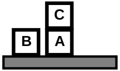


PutOn(A,B)
clobbers Cl(B)
=> order after
PutOn(B,C)



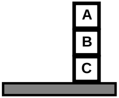
Example contd.





PutOn(A,B) clobbers Cl(B) => order after PutOn(B,C)

PutOn(B,C) clobbers Cl(C) => order after PutOnTable(C)



Discussion of Partial Order Planning

Advantages:

- Plan steps may be executed unordered
- Handles concurrent plans
- Least commitment can lead to shorter search times
- Sound and complete
- Typically produces the optimal plan

Disadvantages:

- Complex plan operators lead to high cost for generating actions
- Larger search space because of concurrent actions

GraphPlan

Planning graph: a polynomial size approximation of a tree with all possible actions from an initial state S_0 to successor states

Can be used as an admissible heuristic to determine if G is reachable from S_0

GraphPlan extracts a plan directly from a such a planning graph

Main idea:
- Construct a graph that encodes constraints on possible plans
- If a valid plan exists it will be part of this planning graph,
  so search only within this graph

Planning graph

Two types of nodes, arranged in alternating levels:

- Propositions
- Actions

Three types of edges between levels:

- Precondition: edge from P to A if P is a precondition of A
- Add: edge from A to P if A has P as effect
- Delete: edge from A to $\neg P$ if A deletes P

Action level includes actions whose preconditions are satisfied in the previous level, plus persistence actions (“no-op”)

Example: Eat cake

Initial state:

$Have(Cake)$

Goal state:

$Have(Cake) \land Eaten(Cake)$

Eat action:

$Action(Eat(Cake)$
$\quad \mathrm{PRECOND} : Have(Cake)$
$\quad \mathrm{EFFECT} : \neg Have(Cake) \land Eaten(Cake))$

Bake action:

$Action(Bake(Cake)$
$\quad \mathrm{PRECOND} : \neg Have(Cake)$
$\quad \mathrm{EFFECT} : Have(Cake))$

Constructing the planning graph

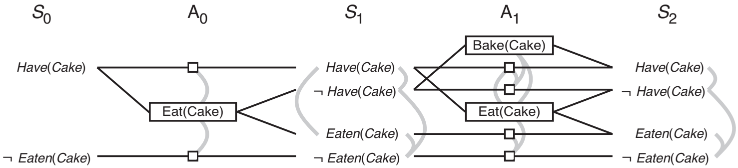
Level S_0 is initialized with all the literals from the initial state

Add an action at level A_i if all its preconditions are present in level S_i

Add a proposition in level S_{i+1} if it is the effect of some action in level A_i (including persistence actions)

Maintain a set of exclusion relations (mutexes) to eliminate incompatible propositions and actions

Example: Eat cake



Small squares indicate persistence action (“no-op”)

Action level contains all actions whose preconditions are satisfied

Edges between nodes on same level indicate mutual exclusion

Mutual exclusion

Two actions are mutually exclusive (mutex) at some stage if no valid plan could contain both at that stage

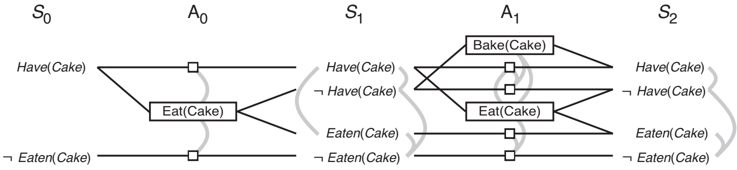
Two actions at the same level can be mutex because of:

- Inconsistent effects: an effect of one negates the effect of the other
- Interference: one negates a precondition of the other
- Competing needs: the actions have mutex preconditions

Two propositions at the same level are mutex if:

- One is a negation of the other
- Inconsistent support: All ways of achieving them are pairwise mutex

Example: Eat cake



Eat(*Cake*) and persistence of *Have*(*Cake*) are mutex because they disagree on the effect *Have*(*Cake*) (inconsistent effect)

Eat(*Cake*) and persistence of *Have*(*Cake*) are also mutex because *Eat*(*Cake*) interferes with the persistence action by negating its precondition (interference)

Bake(*Cake*) and *Eat*(*Cake*) are mutex because they compete on the value of the *Have*(*Cake*) precondition (competing needs)

Observations

Number of propositions always increases
- because all the ones from the previous level are carried forward

Number of actions always increases
- because the number of satisfied preconditions increases

Number of propositions that are mutex decreases
- because there are more ways to achieve same propositions

Number of actions that are mutex decreases
- because of the decrease in mutexes between propositions

After some time, all levels become identical: graph "levels off"

Because there is a finite number of propositions and actions, mutexes will not reappear

Valid plan

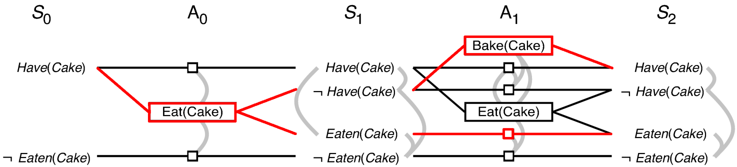
A valid plan is a subgraph of the planning graph such that:
- All goal propositions are satisfied in the last level
- No goal propositions are mutex
- Actions at the same level are not mutex
- Each action's preconditions are made true by the plan

Basic algorithm:

1. Grow the planning graph until all goal propositions are reachable and not mutex
2. If the graph levels off first, return failure (no valid plan exists)
3. Search the graph for a solution (CSP or backward search)
4. If no valid plan is found, add a level and try again

Example: Eat cake

Valid plan:



Planning and acting in the real world

Planners used in the real world are more complex

Durations and resource constraints
- E.g. limited number of staff, or time
- Plan first, schedule later: job shop scheduling

Very large state spaces
- Actions are often low level
- Decompose problem: hierarchical planning

Uncertainty
- Non-correct and non-complete information
- Contingency planning, replanning

Hierarchy and abstraction

Imagine a robot docking with the battery charger: what kinds of actions should we use for its planning?

- Current(0.5amp, left-wheel-motor), Current(0.2amp, right-wheel-motor), . . .
- Go-to-wall(east-wall), Follow-wall(east-wall), Dock

Low-level description yields really long plans, but they are always executable

High-level description yields short, understandable plans, but might be unrealistic

Hierarchical Task Network (HTN)

States the same as in classical planning

Two types of actions

- Primitive actions can be executed directly
  e.g. go-forward
- High-level actions can be refined into a sequence of actions
  e.g. dock-with-charger

High-level actions can be refined to primitive actions (an implementation) or other HLAs, both with possible precedence constraints

Refinements given by a plan library

- defined by domain experts
- learned through experience

Searching for solutions

An instance of a planning problem starts with an initial state (Act)

Creating a plan is done by repeatedly applying refinements recursively to the high-level actions, until we reach the level of the primitive tasks (which can be executed directly)

Backtracking is done if necessary (e.g. if the internal constraints in the task network cannot be satisfied)

Easy if HLAs only contain one implementation (preconditions and effects can be used directly)

However, they usually don't:

- search among each possible implementations
- reason directly about the HLAs

The real world

Things are usually not as expected:

Incomplete information:

- Unknown preconditions, e.g., $Intact(Spare)$
- Disjunctive effects, e.g., $Inflate(x)$ causes $Inflated(x)$ according to the knowledge base, but in reality it actually causes $Inflated(x) \lor SlowHiss(x) \lor Burst(x) \lor BrokenPump \lor \ldots$

Incorrect information:

- Current state incorrect, e.g., spare NOT intact
- Missing/incorrect postconditions in operators

Qualification problem: can never finish listing all the required preconditions and possible conditional outcomes of actions

Solutions

Conditional planning:

1. Plans include observation actions which obtain information
2. Sub-plans are created for each contingency (each possible outcome of the observation actions)

E.g. Check the tire. If it is intact, then were ok, otherwise there are several possible solutions: inflate, call AAA....

Expensive because it plans for many unlikely cases

Monitoring/Replanning:

1. Assume normal states, outcomes
2. Check progress during execution, replan if necessary

Unanticipated outcomes may lead to failure (e.g., no AAA card)

In general, some monitoring is unavoidable

Summary

Planning is very related to search, but allows the actions/states have more structure

We typically use logical inference to construct solutions

State-space vs.plan-space planning

Least-commitment: we build partial plans, order them only as necessary

Planning graphs can be used as a heuristic or searched directly for a planning solution (GraphPlan)

In the real world, it is necessary to consider failure cases - replanning

Hierarchy and abstraction make planning more efficient