Planning

Chapter 10

Outline

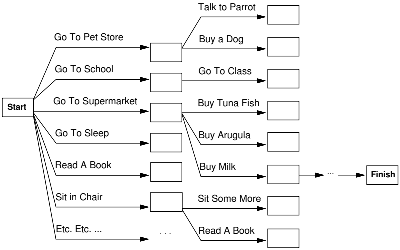
- Search vs. planning
- Using PDDL for planning

Search vs. planning

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

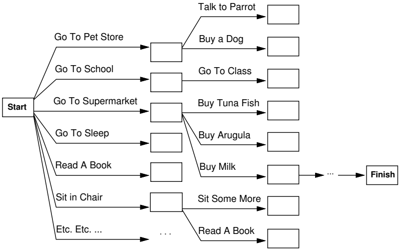
Search vs. planning

- Consider the task get milk, bananas, and a cordless drill
- Standard search algorithms seem to fail miserably:



Search vs. planning

- Consider the task get milk, bananas, and a cordless drill
- Standard search algorithms seem to fail miserably:



- Problems:
  - *Enormous* search space
  - Actions are complex objects:
    - They have preconditions and they change the world
  - Simple goal test is inadequate

Search vs. planning contd.

Planning systems do the following:

1. open up action and goal representation to allow selection
2. divide-and-conquer by subgoaling

Search vs. planning contd.

Planning systems do the following:

1. open up action and goal representation to allow selection
2. divide-and-conquer by subgoaling

Compare:

	Search	Planning
States	Data structures	Logical sentences (positive ground literals)
Actions	Code	Preconditions/outcomes in a schema
Goal	Test	Logical sentence (conjunction of literals)
Plan	Sequence from S_0	Sequence from S_0

PDDL

PDDL (= Planning Domain Definition Language) is a planning system derived from STRIPS, which goes back to 1971(!).

PDDL

PDDL (= Planning Domain Definition Language) is a planning system derived from STRIPS, which goes back to 1971(!).

- The world is assumed to be fully observable, deterministic, static and with a single agent.

PDDL

PDDL (= Planning Domain Definition Language) is a planning system derived from STRIPS, which goes back to 1971(!).

- The world is assumed to be fully observable, deterministic, static and with a single agent.
  - Describe the state of the world by a set of positive facts (ground atomic formulas – like a relational database).
  - Facts that can change their truth value are called *fluents*

PDDL

PDDL (= Planning Domain Definition Language) is a planning system derived from STRIPS, which goes back to 1971(!).

- The world is assumed to be fully observable, deterministic, static and with a single agent.
    - Describe the state of the world by a set of positive facts (ground atomic formulas – like a relational database).
    - Facts that can change their truth value are called *fluents*
- *Action schemas* describe general actions.

PDDL

PDDL (= Planning Domain Definition Language) is a planning system derived from STRIPS, which goes back to 1971(!).

- The world is assumed to be fully observable, deterministic, static and with a single agent.
  - Describe the state of the world by a set of positive facts (ground atomic formulas – like a relational database).
  - Facts that can change their truth value are called *fluents*
- *Action schemas* describe general actions.
  - Schemas are *instantiated* to specific action instances
  - An action instance transforms the world description.

PDDL

PDDL (= Planning Domain Definition Language) is a planning system derived from STRIPS, which goes back to 1971(!).

- The world is assumed to be fully observable, deterministic, static and with a single agent.
  - Describe the state of the world by a set of positive facts (ground atomic formulas – like a relational database).
  - Facts that can change their truth value are called *fluents*
- *Action schemas* describe general actions.
  - Schemas are *instantiated* to specific action instances
  - An action instance transforms the world description.
- Given an *initial world description*, find a sequence of action instances that achieves a given *goal*.
- No explicit mention is made of time.

World States

- The world or domain is described as a variable-free set of atomic formulas.
- Example:
  $\{Block(a), Block(b), \ldots,$
  $\quad On(a, b), OnTable(b), \ldots, Clear(c), \ldots\}$
- Uses *database semantics*: If a fact doesn't appear in the list, it is assumed to be false.
  - E.g. If $On(b, c)$ isn't in the domain description $\neg On(b, c)$ is assumed to hold.
- Constants are assumed to denote distinct individuals, i.e. $a \neq b$.

Action Schema

- An action schema consists of
  - the action name,
  - a list of variables used in the schema,
  - a precondition, and
  - an effect.

Action Schema

- An action schema consists of
  - the action name,
  - a list of variables used in the schema,
  - a precondition, and
  - an effect.
- *Precondition*: Specifies those conditions that must be met before the operator can be applied.
  - Given as a conjunction of literals

Action Schema

- An action schema consists of
  - the action name,
  - a list of variables used in the schema,
  - a precondition, and
  - an effect.
- *Precondition*: Specifies those conditions that must be met before the operator can be applied.
  - Given as a conjunction of literals
- *Effect*: Defines the effect of the action.
  - Given as a conjunction of literals

Action Schema

- An action schema consists of
  - the action name,
  - a list of variables used in the schema,
  - a precondition, and
  - an effect.
- *Precondition*: Specifies those conditions that must be met before the operator can be applied.
  - Given as a conjunction of literals
- *Effect*: Defines the effect of the action.
  - Given as a conjunction of literals
- E.g.: $Action(\ Fly(p, from, to)$
  PRECOND:
  $At(p, from) \land Flight(p) \land Airport(from) \land Airport(to)$
  EFFECT: $\neg At(p, from) \land At(p, to) \quad)$

PDDL Operators

More examples:

- *Move*(x, y, z): Move x from being on y to being on z.
  - PRECOND:

PDDL Operators

More examples:

- $Move(x, y, z)$: Move x from being on y to being on z.
    - PRECOND: $On(x, y) \wedge Clear(x) \wedge Clear(z)$
    - EFFECT:

PDDL Operators

More examples:

- $Move(x, y, z)$: Move x from being on y to being on z.
  - PRECOND: $On(x, y) \land Clear(x) \land Clear(z)$
  - EFFECT: $On(x, z) \land Clear(y) \land \neg On(x, y) \land \neg Clear(z)$

PDDL Operators

More examples:

- *Move*(x, y, z): Move x from being on y to being on z.
  - PRECOND: $On(x, y) \land Clear(x) \land Clear(z)$
  - EFFECT: $On(x, z) \land Clear(y) \land \neg On(x, y) \land \neg Clear(z)$
- *Stack*(x, y): Move x from being on the table to being on y.
  - PRECOND: $OnTable(x) \land Clear(x) \land Clear(y) \land x \neq y$
  - EFFECT: $On(x, y) \land \neg OnTable(x) \land \neg Clear(y)$

PDDL Operators

More examples:

- $Move(x, y, z)$: Move x from being on y to being on z.
  - PRECOND: $On(x, y) \land Clear(x) \land Clear(z)$
  - EFFECT: $On(x, z) \land Clear(y) \land \neg On(x, y) \land \neg Clear(z)$
- $Stack(x, y)$: Move x from being on the table to being on y.
  - PRECOND: $OnTable(x) \land Clear(x) \land Clear(y) \land x \neq y$
  - EFFECT: $On(x, y) \land \neg OnTable(x) \land \neg Clear(y)$
- $Unstack$: (Exercise)

Planning with PDDL

- The initial state is completely specified
  - i.e. all facts initially true are given.
  - recall: A fact not mentioned is assumed to be false

Planning with PDDL

- The initial state is completely specified
  - i.e. all facts initially true are given.
  - recall: A fact not mentioned is assumed to be false
- There is also a *goal* to be achieved.
  - For example, put a red block on b:

$$On(x, b) \wedge Colour\_of(x, red).$$

Planning with PDDL

- The initial state is completely specified
  - i.e. all facts initially true are given.
  - recall: A fact not mentioned is assumed to be false
- There is also a *goal* to be achieved.
  - For example, put a red block on b:

$$On(x, b) \wedge Colour\_of(x, red).$$

- To establish a goal, a sequence of action instances needs to be found that leads from the initial state to the goal.

Planning with PDDL

- An *action instance* a is an action along with bindings for its free variables.
- E.g. recall the schema:
  $Action(\ Fly(p, from, to)$
  PRECOND:
  $At(p, from) \wedge Flight(p) \wedge Airport(from) \wedge Airport(to)$
  EFFECT: $\neg At(p, from) \wedge At(p, to) \quad)$

Planning with PDDL

- An *action instance* a is an action along with bindings for its free variables.
- E.g. recall the schema:

$Action(\ Fly(p, from, to)$
  PRECOND:
    $At(p, from) \land Flight(p) \land Airport(from) \land Airport(to)$
  EFFECT: $\neg At(p, from) \land At(p, to) \quad)$

This has instance:

$Action(\ Fly(AC118, YVR, YYZ)$
  PRECOND:
    $At(AC118, YVR) \land Flight(AC118) \land$
      $Airport(YVR) \land Airport(YYZ)$
  EFFECT: $\neg At(AC118, YVR) \land At(AC118, YYZ) \quad)$

Planning with PDDL

- An action instance a is *possible* in state s iff every precondition in $PRECOND(a)$ holds in s.
- If we describe s by listing those atoms that hold in s, then this can be expressed as
  - $PRECOND^{+}(a) \subseteq s$
  - $PRECOND^{-}(a) \cap s = \emptyset$

  where

  - $PRECOND^{+}(a)$ is the set of positive literals and
  - $PRECOND^{-}(a)$ is the set of negated literals

  in the precondition.
- Equivalently, we can write:

$$PRECOND(a) \subseteq s \cup \{\neg p \mid p \notin s\}.$$

Planning with PDDL

- Let
  - $ADD(a)$ be the set of positive literals in $EFFECT(a)$ and
  - $DEL(a)$ be the set of atoms given by the negative literals in $EFFECT(a)$.
- The result of executing an action instance a that is possible in s is the state:

$$RESULT(a, s) = (s - DEL(a)) \cup ADD(a).$$

Planning with PDDL

- Given an instantiated action sequence $a_1, \ldots, a_n$, and a situation s, we set

  $$S_0 = s$$

  and

  $$s_i = RESULT(a, s_{i-1}) \quad \text{for} \quad i = 1, \ldots, n.$$

- The action sequence *succeeds* if every individual action succeeds.
- The action sequence *achieves* the goal G if s_n entails G.

Planning with PDDL

- Planning can be done in either a "forward" or "backward" manner.
- Known as *progressive* and *regressive* planning respectively.
- Originally regressive planners were most used, due to their focus on the goal
- With better heuristics and increased computational power, progressive planners have come to dominate.

Progressive Planning in PDDL

- The most intuitive way to try to obtain a plan is to:
  - begin at the initial state and
  - find a sequence of actions that lead to the goal.
- This is called a *progressive planner* since it progresses the initial state forward until a state satisfying the goal is found.

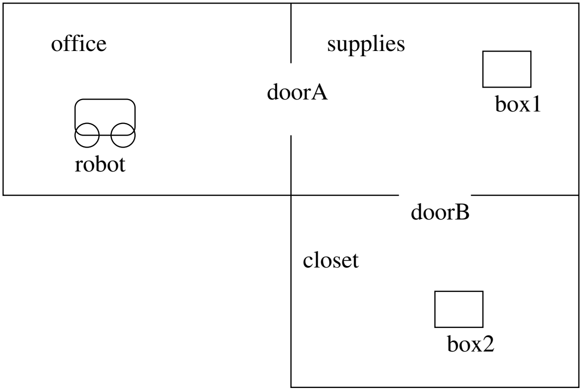
Progressive Planning

Depth-First Progressive Planner:

Input: A world description S and goal formula *Goal*
Output: A plan or *fail*

```
ProgPlan(S, Goal)
    if Goal ⊆ S then return empty plan
    for each operator instance ⟨Act, Pre, Add, Del⟩
                                such that S satisfies Pre do {
        let S′ = (S \ Del) ∪ Add
        let Plan = ProgPlan(S′, Goal)
        if Plan ≠ fail then return Plan·Act
    }
    return fail
```

Example



Goal: Get some box into the office

Example

Initial world DB:

$Box(box1),\ Box(box2),$

$InRoom(box1, supplies),\ InRoom(box2, closet),$
$\quad InRoom(robot, office),$

$Connected(office, supplies),\ Connected(supplies, office),$
$\quad Connected(closet, supplies),\ Connected(supplies, closet)$

Example

Action schema:

$goThru(r1, r2)$

- PRECOND: $InRoom(robot, r1)$, $Connected(r1, r2)$
- EFFECT: $InRoom(robot, r2)$, $\neg InRoom(robot, r1)$,

$pushThru(x, r1, r2)$

- PRECOND: $InRoom(robot, r1)$, $InRoom(x, r1)$,
  $Connected(r1, r2)$
- EFFECT: $InRoom(robot, r2)$, $InRoom(x, r2)$,
  $\neg InRoom(robot, r1)$, $\neg InRoom(x, r1)$

Progressive Planning Example

With *goThru(office, supplies)*, obtain first progressed DB:

$Box(box1),\ Box(box2),$

$InRoom(box1, supplies),\ InRoom(box2, closet),$
$\quad InRoom(robot, supplies),$

$Connected(office, supplies),\ Connected(supplies, office),$
$Connected(closet, supplies),\ Connected(supplies, closet)$

With *pushThru(box1, supplies, office)*, obtain the DB:

$Box(box1),\ Box(box2),$

$InRoom(box1, office),\ InRoom(box2, closet),$
$\quad InRoom(robot, office),$

$Connected(office, supplies),\ Connected(supplies, office),$
$Connected(closet, supplies),\ Connected(supplies, closet)$

Regressive Planning with PDDL

- Idea: Begin with the goal state, and work backwards to try to get to the initial state.
- The *search space* can be defined in a "backwards chaining" fashion:
- Idea: Work backwards, repeatedly simplifying the goal until we get a goal satisfied in the initial state.
- Called *goal regression*

Regressive Planning

Depth-First Regressive Planner:

Input: The initial world description *Init* and a goal formula *Goal*
Output: A plan or *fail*

RegrPlan(Init, Goal)
    if Goal $\subseteq$ Init then return empty plan
    for each operator instance $\langle Act, Pre, Add, Del \rangle$
                                        such that $Del \cap Goal = \emptyset$ {
        let $Goal' = (Goal \cup Pre) \setminus Add$
        let $Plan = RegrPlan(Init, Goal')$
        if Plan $\neq$ fail then return Plan·Act
    }
    return fail

Regressive Planning Example

- Planner is called with the initial world DB and the goal:
  $Box(x), InRoom(x, office)$
- The goal is not satified by the initial world DB.
- The action instance
  $pushThru(box1, supplies, office)$
  has a delete list that does not intersect with the goal.
- Get regressed subgoal:
  $Box(box1), InRoom(robot, supplies), InRoom(box1, supplies),$
  $Connected(supplies, office)$
- The action instance: $goThru(office, supplies)$
  yields the regressed goal:
  $Box(box1), InRoom(robot, office), InRoom(box1, supplies),$
  $Connected(supplies, office),\ Connected(office, supplies)$
- This is satisfied in the initial state.

Regressive Planning: Another Example

- A, B, and C are on the table.
- The goal is $On(A, B)$ and $On(B, C)$.
- Initial state:

  $$\begin{aligned} Init = \{ & OnTable(A), OnTable(B), OnTable(C), \\ & Clear(A), Clear(B), Clear(C)\} \end{aligned}$$

Regressive Planning: Another Example

- A, B, and C are on the table.
- The goal is $On(A, B)$ and $On(B, C)$.
- Initial state:
  $$\begin{aligned} Init = \{ & OnTable(A), OnTable(B), OnTable(C), \\ & Clear(A), Clear(B), Clear(C)\} \end{aligned}$$
- Initial call: $RegrPlan(Init, \{On(A, B), On(B, C)\})$

Regressive Planning: Another Example

- A, B, and C are on the table.
- The goal is $On(A, B)$ and $On(B, C)$.
- Initial state:
  $$\begin{aligned} Init = \{&OnTable(A), OnTable(B), OnTable(C), \\ &Clear(A), Clear(B), Clear(C)\}\end{aligned}$$
- Initial call: $RegrPlan(Init, \{On(A, B), On(B, C)\})$
- Action instance: $Stack(A, B)$

Regressive Planning: Another Example

- A, B, and C are on the table.
- The goal is $On(A, B)$ and $On(B, C)$.
- Initial state:
  $$Init = \{OnTable(A), OnTable(B), OnTable(C), Clear(A), Clear(B), Clear(C)\}$$
- Initial call: $RegrPlan(Init, \{On(A, B), On(B, C)\})$
- Action instance: $Stack(A, B)$
- Call:
  $RegrPlan(Init, \{OnTable(A), Clear(A), Clear(B), On(B, C)\})$

Regressive Planning: Another Example

- A, B, and C are on the table.
- The goal is $On(A, B)$ and $On(B, C)$.
- Initial state:
  $$\begin{aligned} Init = \{ & OnTable(A), OnTable(B), OnTable(C), \\ & Clear(A), Clear(B), Clear(C)\} \end{aligned}$$
- Initial call: $RegrPlan(Init, \{On(A, B), On(B, C)\})$
- Action instance: $Stack(A, B)$
- Call:
  $RegrPlan(Init, \{OnTable(A), Clear(A), Clear(B), On(B, C)\})$
- Action instance: $Stack(B, C)$

Regressive Planning: Another Example

- A, B, and C are on the table.
- The goal is $On(A, B)$ and $On(B, C)$.
- Initial state:
  $$\begin{aligned} Init = \{ & OnTable(A), OnTable(B), OnTable(C), \\ & Clear(A), Clear(B), Clear(C)\} \end{aligned}$$
- Initial call: $RegrPlan(Init, \{On(A, B), On(B, C)\})$
- Action instance: $Stack(A, B)$
- Call:
  $RegrPlan(Init, \{OnTable(A), Clear(A), Clear(B), On(B, C)\})$
- Action instance: $Stack(B, C)$
- Call: $RegrPlan(Init, \{OnTable(A), Clear(A), Clear(B),$
  $OnTable(B), Clear(C)\})$

Regressive Planning: Another Example

- A, B, and C are on the table.
- The goal is $On(A, B)$ and $On(B, C)$.
- Initial state:
  $Init = \{OnTable(A), OnTable(B), OnTable(C),$
  $\qquad Clear(A), Clear(B), Clear(C)\}$
- Initial call: $RegrPlan(Init, \{On(A, B), On(B, C)\})$
- Action instance: $Stack(A, B)$
- Call:
  $RegrPlan(Init, \{OnTable(A), Clear(A), Clear(B), On(B, C)\})$
- Action instance: $Stack(B, C)$
- Call: $RegrPlan(Init, \{OnTable(A), Clear(A), Clear(B),$
  $\qquad OnTable(B), Clear(C)\})$
  which is satisfied in the initial state.

Heuristics for Planning

- Neither forward nor backward search is efficient without a good heuristic.
- Recall: finding an *admissable heuristic* via defining a *relaxed problem*.

Heuristics for Planning

- Neither forward nor backward search is efficient without a good heuristic.
- Recall: finding an *admissable heuristic* via defining a *relaxed problem*.
- Heuristics:
  - Ignore some or all of the preconditions
  - Ignore the delete list

Heuristics for Planning

- Neither forward nor backward search is efficient without a good heuristic.
- Recall: finding an *admissable heuristic* via defining a *relaxed problem*.
- Heuristics:
  - Ignore some or all of the preconditions
  - Ignore the delete list
- Problem:
  - The simplified planning problem is still NP-hard
  - Resolve by using a greedy algorithm

Heuristics for Planning

- Neither forward nor backward search is efficient without a good heuristic.
- Recall: finding an *admissable heuristic* via defining a *relaxed problem*.
- Heuristics:
  - Ignore some or all of the preconditions
  - Ignore the delete list
- Problem:
  - The simplified planning problem is still NP-hard
  - Resolve by using a greedy algorithm
- Also: domain-specific heuristics

Heuristics for Planning

Other possibilities:

- State abstraction: Combine states by ignoring some fluents
- Problem decomposition:
  - Divide a problem into parts;
  - solve each part independently;
  - combine the parts

Heuristics for Planning

Other possibilities:

- State abstraction: Combine states by ignoring some fluents
- Problem decomposition:
  - Divide a problem into parts;
  - solve each part independently;
  - combine the parts

Other types of planners:

- Partial-order planners
- GRAPHPLAN

PDDL: Summary

- Very successful, mainly because it is simple (basically STRIPS).

PDDL: Summary

- Very successful, mainly because it is simple (basically STRIPS).
- Very limited representation language:
    1. All information must be specified.

PDDL: Summary

- Very successful, mainly because it is simple (basically STRIPS).
- Very limited representation language:
  1. All information must be specified.
  2. Actions with state-dependent effects must be split.
     - E.g., a *move* doesn't change the colour of an object usually, but it does if an object moves into the path of a spray gun.
     - In PDDL need actions *move_object_into_path_of_spray_gun* and *move_object_elsewhere*.

PDDL: Summary

- Very successful, mainly because it is simple (basically STRIPS).
- Very limited representation language:
  1. All information must be specified.
  2. Actions with state-dependent effects must be split.
     - E.g., a *move* doesn't change the colour of an object usually, but it does if an object moves into the path of a spray gun.
     - In PDDL need actions *move_object_into_path_of_spray_gun* and *move_object_elsewhere*.
  3. We can't reason *about* actions.

PDDL: Summary

- Very successful, mainly because it is simple (basically STRIPS).
- Very limited representation language:
  1. All information must be specified.
  2. Actions with state-dependent effects must be split.
     - E.g., a *move* doesn't change the colour of an object usually, but it does if an object moves into the path of a spray gun.
     - In PDDL need actions *move_object_into_path_of_spray_gun* and *move_object_elsewhere*.
  3. We can't reason *about* actions.
  4. Single agent. No exogenous actions.

PDDL: Summary

- Very successful, mainly because it is simple (basically STRIPS).
- Very limited representation language:
  1. All information must be specified.
  2. Actions with state-dependent effects must be split.
     - E.g., a *move* doesn't change the colour of an object usually, but it does if an object moves into the path of a spray gun.
     - In PDDL need actions *move_object_into_path_of_spray_gun* and *move_object_elsewhere*.
  3. We can't reason *about* actions.
  4. Single agent. No exogenous actions.
  5. Offline. No sensing.

PDDL: Summary

- Very successful, mainly because it is simple (basically STRIPS).
- Very limited representation language:
  1. All information must be specified.
  2. Actions with state-dependent effects must be split.
     - E.g., a *move* doesn't change the colour of an object usually, but it does if an object moves into the path of a spray gun.
     - In PDDL need actions *move_object_into_path_of_spray_gun* and *move_object_elsewhere*.
  3. We can't reason *about* actions.
  4. Single agent. No exogenous actions.
  5. Offline. No sensing.
  6. No concurrency, non-determinism.

PDDL: Summary

- Very successful, mainly because it is simple (basically STRIPS).
- Very limited representation language:
  1. All information must be specified.
  2. Actions with state-dependent effects must be split.
     - E.g., a *move* doesn't change the colour of an object usually, but it does if an object moves into the path of a spray gun.
     - In PDDL need actions *move_object_into_path_of_spray_gun* and *move_object_elsewhere*.
  3. We can't reason *about* actions.
  4. Single agent. No exogenous actions.
  5. Offline. No sensing.
  6. No concurrency, non-determinism.
- General planning comment: Things get tricky very quickly.
  - E.g: $On(B, table)$, $On(C, A)$, Goal: $On(A, B)$, $On(B, C)$.