Planning for Artificial Intelligence

Lukáš Chrpa

Classical Planning and State-space Search

Classical Planning Representation

(revision)

STRIPS Planning Task

- A **planning task** in **STRIPS** is a quadruple (P,A,I,G), where
  - **P** is a finite set of **atoms** (or facts or propositions)
  - **A** is a finite set of **actions**, where each action $a \in A$ is a triple (pre(a),del(a),add(a)), all subsets of P, where
    - pre(a) is a **precondition** of a
    - del(a) is a set of **delete effects** of a
    - add(a) is a set of **add effects** of a
  - **I**$\subseteq$P is an **initial state**
  - **G**$\subseteq$P is a **goal**

STRIPS Planning Task cont.

- **States** are **collections of atoms**, i.e., $S \subseteq 2^P$
- An action a is **applicable** in a state s iff **pre(a)⊆s**
  - (otherwise a is **inapplicable** in s)
- A state s’ is the **result** of application of an applicable action a in a state s iff **s’=(s\del(a))**∪**add(a)**

SAS Planning Task

- A **planning task** in **SAS** is quadruple (V,A,I,G), where
  - **V** is a set of **variables**, where each variable v∈V has its own domain dom(v)
  - **A** is a set of **actions**, where each action a∈A is a pair (pre(a),eff(a)), both partial assignments over V, where
    - pre(a) is a **precondition** of a
    - eff(a) stands for **effects** of a
  - **I** is an **initial state** (a complete assignment over V)
  - **G** is a **goal** (partial assignment over V)

SAS Planning Task cont.

- Let q[v] denote the value of a variable v in a (partial) assignment q
- **States** are complete assignments over V
- An action a is **applicable** in a state s iff **pre(a)[v]=s[v]** whenever pre(a)[v] is specified
  - (otherwise a is **inapplicable** in s)
- A state s' is the **result** of application of an applicable action a in a state s iff **s'[v]=eff(a)[v]** whenever eff(a)[v] is specified or **s'[v]=s[v]** otherwise

Solution Plans

- Let $\gamma(s,a)=s'$ iff s' is the result of application of an action a in a state s (a is applicable in s)
  - $\gamma(s,a)$ is undefined iff a is inapplicable in s
- Let γ^* be defined recursively
  - $\gamma^*(s,\langle\rangle)=s$
  - $\gamma^*(s,\langle a_1,a_2,\ldots,a_n\rangle)=\gamma^*(\gamma(s,a_1),\langle a_2,\ldots,a_n\rangle)$
- We say that π, a sequence of actions over A, is a **solution plan** (or a **plan**) of the planning task iff $\gamma^*(I,\pi) \models G$ ($G\subseteq\gamma^*(I,\pi)$ for STRIPS)

STRIPS vs SAS

- Is SAS more expressive than STRIPS ?

STRIPS vs SAS

- Is SAS more expressive than STRIPS ?
  - No, they are equally expressive
- Why ?

STRIPS vs SAS

- Is SAS more expressive than STRIPS ?
  - No, they are equally expressive
- Why ?
  - STRIPS → SAS
    - Each atom (proposition) can be converted to a state variable with domain {true,false}
  - SAS → STRIPS
    - Each possible variable assignment can be converted to an atom (proposition)
  - Converting actions from STRIPS to SAS (and vice versa) → To think about at home

Convention

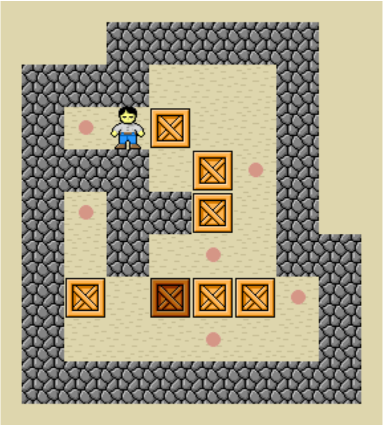
- By an **atom** or a **fact** we mean
  - A **proposition** (STRIPS representation)
  - A **variable assignment** (SAS representation)
- An **action** is in literature often denoted as an **operator**
  - We assume that actions are **well defined** (add and del effects are disjoint)
- By **lifted** representation we mean the one using **free variables**
  - STRIPS or SAS representation is then obtained by **grounding**, i.e., substituting free variables by specific objects
- Some concepts (and algorithms) will be presented in STRIPS while some other concepts (and algorithms) in SAS

States, Mutexes, Invariants

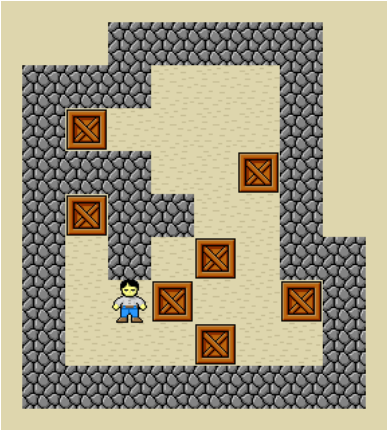
States

- The set of states **S** is derived from the propositions (STRIPS) or state variables (SAS) of a given planning task
- A state $s_g \in S$ is a **goal state** iff $s_g \models G$
- A state $s' \in S$ is **reachable** from a state $s \in S$ iff $\exists \pi \in A^*: \gamma^*(s,\pi)=s'$
- A state $s' \in S$ is **unreachable** from a state $s \in S$ iff $\nexists \pi \in A^*: \gamma^*(s,\pi)=s'$
  - By denoting a state (un)reachable without mentioning from which state we mean (un)reachable from the initial state I
- A state $s \in S$ is a **dead-end state** iff $\nexists \pi \in A^*: \gamma^*(s,\pi) \models G$

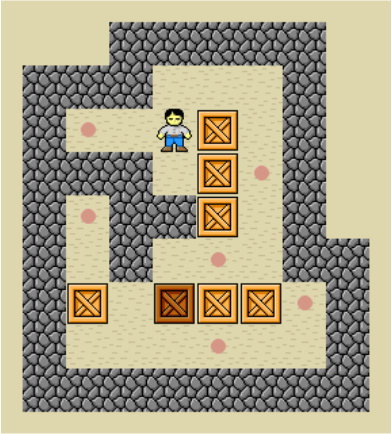
Sokoban Example



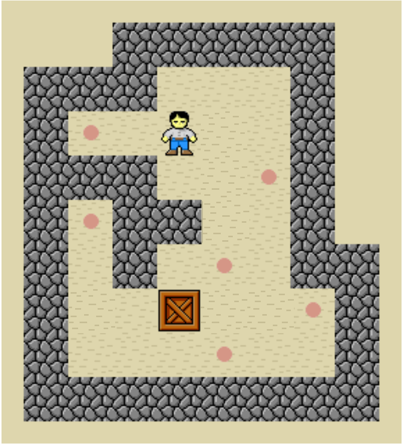
Initial state



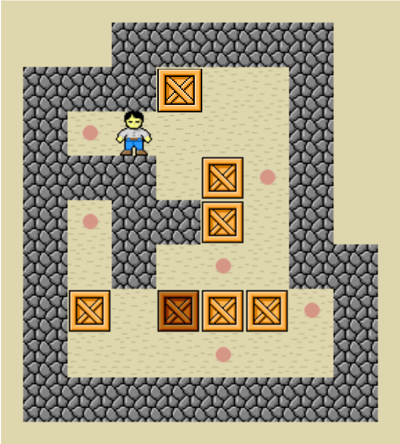
Goal state



Reachable state



Unreachable state



Dead-end state

State Invariants

- An **invariant** is a **property** of an object which **remains unchanged**, **after operations** of certain type are applied to the object
- We say that a state s **has a property** p iff p holds in s
- We say that a state s **has an invariant** p iff each reachable state s’ from s has a property p
- We say that a planning task **has an invariant** p iff the initial state I has the invariant p

Mutual Exclusivity (Mutex)

- We say that atoms (or facts) p and q are **mutually exclusive** (or **mutex**) in a given planning task iff for each reachable state s, $\{p,q\} \not\subseteq s$
- We say that a set of atoms $\{p_1,p_2,\ldots,p_n\}$ forms a **mutex group** in a given planning task iff for each reachable state s, $|\{p_1,p_2,\ldots,p_n\} \cap s| \leq 1$
- We say that a set of atoms $\{p_1,p_2,\ldots,p_n\}$ forms a **facts alternating mutex (FAM) group** in a given planning task iff for each reachable state s, $|\{p_1,p_2,\ldots,p_n\} \cap s| = 1$
- Any relation between mutex group and FAM group ?
- Any relation between mutexes and mutex group ?

Mutexes, Dead-ends and Invariants

- Is a property of being a dead-end state an invariant ?
- Is mutex an invariant ?
- Is a mutex (or FAM) group an invariant ?

- Some other examples of invariants (even domain-specific) → To think about at home

Planning – what we can look for

1) Deciding plan (non)existence

2) Finding any (satisficing) plan (if it exists)

3) Finding an optimal plan (if it exists)

   ……..

- The tasks are **very different** and techniques addressing them are often **disjoint**

Optimal plans

- Optimizing for **plan length**: A plan π of some planning task is **optimal** iff for each plan π' of the same planning task it is the case that $|\pi| \leq |\pi'|$

- **Action cost** is a function $c: A \rightarrow N_0$

- Optimizing for **total action cost:** A plan π of some planning task is **optimal** iff for each plan π' of the same planning task it is the case that $\sum_{a \in \pi} c(a) \leq \sum_{a' \in \pi'} c(a')$

Complexity

- Deciding plan existence in classical planning is **PSPACE-complete**
  - With plans of polynomial length it is **NP-hard**
- Some classes of planning tasks can be easy (in **P**)
- Sometimes there are differences in complexity for satisficing (any plan) and optimal planning
  - For BlocksWorld, finding **any plan** is in **P** while finding an **optimal plan** is **NP-hard**

Towards Solving Planning Tasks

How to address Planning Tasks ?

- **State-space search**
  - The most widespread
- Symbolic search
  - Representing sets of states by Binary Decision Diagrams
- Translate the problem to a different formalism
  - Boolean Satisfiability (SAT)
  - Constraint Satisfaction Problem (CSP)
- Plan-space search

…..

State-space search

- Search direction
  - **Progressive** (from the initial to the goal state)
  - Regressive (from the goal to the initial situation)
  - Bidirectional
- Uniformed (blind) vs **informed (heuristic)**
- Systematic vs Local
- Additional knowledge (e.g. symmetry pruning, invariants etc.)

Progressive search

$s := I$

$\pi := \langle\rangle$

while $s \mid\neq G$ **do**

  non-deterministically select $a \in A$ s.t. a is applicable in s

  if no such a exists **then return** no solution

  $s := \gamma(s, a)$

  $\pi := \pi.a$

return π

Regressive search (in STRIPS)

$s := G$

$\pi := \langle\rangle$

while $I \not\models s$ **do**

  non-deterministically select $a \in A$ s.t. $s \cap add(a) \neq \emptyset$ and $s \cap del(a) = \emptyset$

  if no such a exists **then return** no solution

  $s := (s \setminus add(a)) \cup pre(a)$

  $\pi := a.\pi$

return π

Progressive search

$s := I$

$\pi := \langle\rangle$

while s $\models\!\!\!\!/$ G **do**

    non-deterministically select $a \in A$ s.t. a is applicable in s

    if no such a exists **then return** no solution

    $s := \gamma(s, a)$

    $\pi := \pi.a$

return π

Uninformed (blind) search

- **Depth-first search**
  - Successor nodes are pushed into a **stack**
  - Memory efficient
  - Does not guarantee optimality
- **Breadth-first search**
  - Successor nodes are pushed into a (priority) **queue**
  - Memory consuming
  - Guarantees optimality
- Iterative deepening

Heuristic Function

- Let S be a set of states for a given planning task Π. A **heuristic function** (or **heuristics**) for Π is a function $h:S \rightarrow N_0 \cup \{\infty\}$
- The value h(s) **estimates** distance from s to the nearest goal state
- h(s) is called **heuristic estimate** or **heuristic value** for s
- A **perfect** (or optimal) **heuristic**, denoted as h*, maps each state to the length (or cost) of the optimal plan to the nearest goal state
  - If $h^*(s)=\infty$ then no goal state is reachable from s (s is a dead-end state)

Properties of Heuristic Function

- Heuristic function h for Π (over S) is
  - **safe** if for each $s \in S$ s.t. $h(s)=\infty$ it holds that s is a dead-end state ($h^*(s)=\infty$)
  - **goal aware** if $h(s_G)=0$ for each goal state s_G
  - **admissible** if for each $s \in S$ it holds that $h(s) \leq h^*(s)$
  - **consistent** if goal aware and for each $s,s' \in S$ s.t. s' is a successor of s it holds that $h(s) \leq h(s') + cost(s,s')$
- Relationships ?

Practical remarks

- Heuristic function should be **safe** and **goal aware**
- Heuristic function has to be **admissible** for **optimal planning**
- **Informativeness** of heuristic function
  - shape of its landscape (e.g. monotonic, local minima)
- **Complexity/hardness** of computation of heuristic values
- Complexity of **implementation** of heuristic function
- Often “it works well in practice” (for some classes of domains) is the only analysis we do have

Terminology

- A **search node** is a pair $n=(s,\pi_s)$, where s is a state and π_s is a sequence of actions from I to s
- A $n'=(s',\pi_{s'})$ is a **successor node** of a node $n=(s,\pi_s)$ iff there is an action a s.t. $s'=\gamma(s,a)$ and $\pi_{s'}=\pi_s.a$
- A **search-space** is composed from search nodes and edges, where a (directed) edge from n to n' exists only if n' is a successor node of n
- The **g value** of a node $n=(s,\pi_s)$, denoted as **g(n)** is the length (or cost) of π_s
- The **f value** of a node $n=(s,\pi_s)$ is **f(n)=g(n)+h(s)**

Greedy Best-First Search (GBFS)

```
open:=new priority_queue() //ordered by the h-value
closed:=∅
open.push((I,⟨⟩))
while !open.empty() do
        n:=open.pop()
        if n.state()∉closed then
                closed:=closed∪{n.state()}
                if n.state() |= G then return n.plan()
                foreach n’ being a successor of n do
                        if h(n’.state())≠∞ then open.push(n’)
return no solution
```

Properties of GBFS

- Widely used for satisficing planning
- **complete** if h is safe (with duplicate detection)
- **suboptimal** (even if h is admissible)

A*

```
open:=new priority_queue() //ordered by the f-value
closed:=∅
dist:=∅
open.push((I,⟨⟩))
while !open.empty() do
    n:=open.pop()
    if n.state()∉closed or g(n)<dist(n.state()) then
        closed:=closed∪{n.state()}
        dist(n.state()):=g(n)
        if n.state() |= G then return n.plan()
        foreach n' being a successor of n do
            if h(n'.state())≠∞ then open.push(n')
return no solution
```

Properties of A*

- Often used for optimal planning and rarely for satisficing planning
- **complete** if h is safe
- **optimal** if h is admissible
- Does not reopen nodes if h is consistent

Weighted A*

- The **f** value is modified to **f(n)=g(n)+W*h(s)**, where the **weight** W≥0
- With W=
  - 0 – we get breadth-first search
  - 1 – we get A*
  - ∞ – we get GBFS
- Commonly used for satisficing planning
- If h is admissible and W>1, then plans are **bounded suboptimal** by at most the factor W

Local Search

- Local search techniques are more **memory efficient** than systematic search ones
- **Hill Climbing**
  - only the current state is kept in memory
  - in each step, **the successor node with minimum h value** is selected and the **h value must be lower than for the current state**
  - can be easily stuck in local minima
- **Enforced Hill Climbing**
  - performs **Breadth-First Search to find a node with lower h value** then the current state
  - can get stuck in dead-end states

Bidirectional Search

- Combines progressive and regressive search
- Uninformed bidirectional search might consists of two interleaving BFS (from I and from G)
- Heuristic bidirectional search
  - **Front-to-back**
    - heuristic values are computed to the goal, or to the initial state (depending on direction)
  - **Front-to-front**
    - heuristic values are computed to the best node in the open list of the opposite search