Inteligência Artificial
ACH2016

PDDL e planejamento como busca no espaço de estados

Profa. Karina Valdivia Delgado
EACH-USP

Slides baseados em:

Slides do Prof. Edirlei Soares de Lima

Slides do Prof. Paulo Santos (FEI)

Slides da Profa. Leliane Nunes de Barros (IME)

RUSSEL, S.; NORVIG, P. Artificial Intelligence: A modern approach. Third Edition, 2010. Capítulo 11

PDDL

Planning Domain Definition Language (PDDL)

- Vários formalismos de planejamento foram sistematizados em uma linguagem padrão, chamada PDDL.
  -Baseada no modelo STRIPS e outros como ADL e redes hierárquicas.
  -Permite que diferentes sistemas permutem problemas benchmark e comparem seus resultados.
- Possui varias versões com a incorporação de novas características.
  - 1.2 - Versão básica (Manual)
  - 2.1 - Funções e métricas (Manual)
  - 3.1 - Versão atual (Manual)

Planning Domain Definition Language (PDDL)

- Linguagem adotada como padrão para descrever domínios de planejamento. Permite incluir:
  - Tipos
  - Funções
  - Variáveis numéricas
  - Ações durativas
  - Funções de otimização

- Proposta inicial para a competição de planejamento: AIPS 2002 Planning Competition .

Planning Domain Definition Language (PDDL)

- **Componentes da linguagem PDDL**:
  - **Objetos e tipos de objetos**: objetos e tipos de objetos que compõem o problema de planejamento.
  - **Predicados**: propriedades dos objetos – podem ser verdadeiros ou falsos.
  - **Estado Inicial**: estado do mundo onde o processo de planejamento se inicia.
  - **Objetivos:** predicados que devem ser verdade para concluir o processo de planejamento.
  - **Operadores (chamados de ações em PDDL):** ações que podem ser executadas e modificam o estado do mundo.

PDDL – Domain & Problem

- Tarefas de planejamento especificadas em PDDL são separadas em dois arquivos:
  - **Domain File:** tipos, predicados e ações.
  - **Problem File:** objetos, estado inicial e objetivos.

PDDL - Exemplo

“Existe um robô que pode se mover entre duas salas, pegar e soltar caixas com ambas as suas mãos. Inicialmente, o robô e 4 caixas estão na sala 1. O objetivo é que o robô leve as caixas para a sala 2.”

PDDL - Domain File

- Sintaxe:

```
(define (domain <domain name>)
     (:requirements :strips :equality
:typing)
     (:types <list of types>)
     (:constants <list of constants>)
     <PDDL code for predicates>
     <PDDL code for first action>
     [...]
     <PDDL code for last action>
)
```

PDDL - Domain File

- Sintaxe:

```
(define (domain <domain name>)
     (:requirements :strips :equality
:typing)
     (:types <list of types>)
     (:constants <list of constants>)
     <PDDL code for predicates>
     <PDDL code for first action>
     [...]
     <PDDL code
)
```

Tipos: sala, caixa, braço
PDDL:

```
(:types room box arm)
```

PDDL - Domain File

- Sintaxe:

```
(define (domain <domain name>)
     (:requirements :strips :equality :typing)
     (:types <list of types>)
     (:constants <list of constants>)
     <PDDL code for predicates>
     <PDDL code for first action>
     [...]
     <PDDL code
 )
```

Constantes – objetos que podem ser usados no arquivos de domínio.
Ex: braço esquerdo e direito
PDDL:

```
(:constants left right - arm)
```

PDDL - Domain File

- Sintaxe:

```
(define (domain <domain name>)
     (:requirements :strips :equality :typing)
     (:types <list of types>)
     (:constants <list of constants>)
     <PDDL code for predicates>
     <PDDL code for first action>
     [...]
     <PDDL code
 )
```

Predicados ...

PDDL - Domain File

- Sintaxe:

```
(define (domain <domain name>)
     (:requirements :strips :equality :typing)
     (:types <list of types>)
```

Predicados:

robot-at(x) – verdadeiro se o robô estiver na sala x
box-at(x, y) – verdadeiro se a caixa x estiver na sala y
free(x) – verdadeiro se o braço x não estiver segurando uma caixa
carry(x, y) – verdadeiro se o braço y estiver segurando a caixa x

PDDL:

```
(:predicates
   (robot-at ?x - room)
   (box-at ?x - box ?y - room)
   (free ?x - arm)
   (carry ?x – box ?y - arm)
 )
```

PDDL - Domain File

- Sintaxe:

```
(define (domain <domain name>)
     (:requirements :strips :equality :typing)
     (:types <list of types>)
     (:constants <list of constants>)
     <PDDL code for predicates>
     <PDDL code for first action>
     [...]
     <PDDL code for last action>
 )
```

Ações ...

PDDL - Domain File

- Sintaxe:

```
(define (domain <domain name>)
     (:requirements :strips :equality :typing)
```

Descrição: O robô se move da sala x para a sala y.
Precondição: robot-at(x) ser verdade.
Efeito: robot-at(y) se torna verdade. robot-at(x) se torna falso.

PDDL:

```
 (:action move
    :parameters (?x ?y - room)
    :precondition (robot-at ?x)
    :effect (and (robot-at ?y) (not (robot-at ?x)))
 )
```

PDDL - Domain File

```
(define (domain robot)
  (:requirements :strips :equality :typing)
  (:types room box arm)
  (:constants left right - arm)
  (:predicates
    (robot-at ?x - room)
    (box-at ?x - box ?y - room)
    (free ?x - arm)
    (carry ?x - box ?y - arm)
  )
  (:action move
    :parameters (?x ?y - room)
    :precondition (robot-at ?x)
    :effect (and (robot-at ?y) (not (robot-at ?x)))
  )
  (:action pickup
    :parameters (?x - box ?y - arm ?w - room)
    :precondition (and (free ?y) (robot-at ?w) (box-at ?x ?w))
    :effect (and (carry ?x ?y) (not (box-at ?x ?w)) (not(free ?y)))
  )
  (:action putdown
    :parameters (?x - box ?y -arm ?w - room)
    :precondition (and (carry ?x ?y) (robot-at ?w))
    :effect (and (not(carry ?x ?y)) (box-at ?x ?w) (free ?y))
  )
)
```

PDDL – Problem File

- Sintaxe:

```
(define (problem <problem name>)
     (:domain <domain name>)
     <PDDL code for objects>
     <PDDL code for initial state>
     <PDDL code for goal specification>
  )
```

PDDL – Problem File

- Sintaxe:

```
(define (problem <problem name>)
     (:domain <domain name>)
     <PDDL code for objects>
     <PDDL code for initial state>
)
```

Objetos:
Salas: room1, room2
Caixas: box1, box2, box3, box4
Braços: left, right
PDDL:

```
  (:objects
     room1 room2 - room
     box1 box2 box3 box4 - box
     left right - arm
   )
```

PDDL – Problem File

- Sintaxe:

```
(define (problem <problem name>)
     (:domain <domain name>)
     <PDDL code for objects>
     <PDDL code for initial state>
     <PDDL code for goal specification>
)
```

```
Estado inicial ...
```

PDDL – Problem File

- Sintaxe:

```
(define (problem <problem name>)
     (:domain <domain name>)
     <PDDL code for objects>
```

Estado Inicial: todas as caixas e robô estão na primeira sala.
PDDL:

```
(:init
   (robot-at room1)
   (box-at box1 room1)
   (box-at box2 room1)
   (box-at box3 room1)
   (box-at box4 room1)
   (free left)
   (free right)
)
```

PDDL – Problem File

- Sintaxe:

```
(define (problem <problem name>)
     (:domain <domain name>)
     <PDDL code for objects>
     <PDDL code for initial state>
     <PDDL code for goal specification>
  )
```

Objetivo...

PDDL – Problem File

- Sintaxe:

```
(define (problem <problem name>)
     (:domain <domain name>)
     <PDDL code for objects>
     <PDDL code for initial state>
```

Objetivo: todas as caixas estão na segunda sala.

PDDL:

```
 (:goal
    (and (box-at box1 room2)
         (box-at box2 room2)
         (box-at box3 room2)
         (box-at box4 room2)
     )
 )
```

PDDL – Problem File

```
(define (problem robot1)
(:domain robot)
  (:objects
    room1 room2 - room
    box1 box2 box3 box4 - box
    left right - arm
  )
  (:init
    (robot-at room1)
    (box-at box1 room1)
    (box-at box2 room1)
    (box-at box3 room1)
    (box-at box4 room1)
    (free left)
    (free right)
  )
  (:goal
    (and
      (box-at box1 room2)
      (box-at box2 room2)
      (box-at box3 room2)
      (box-at box4 room2)
    )
  )
)
```

Mundo dos blocos em PDDL

```
(define (domain BLOCKS)
  (:requirements :strips) ...
  (:action pick_up
             :parameters (?x)
             :precondition (and (clear ?x) (ontable ?x) (handempty))
             :effect (and (not (ontable ?x)) (not (clear ?x)) (not (handempty)) (hol
  (:action put_down
             :parameters (?x)
             :precondition (holding ?x)
             :effect  (and (not (holding ?x)) (clear ?x) (handempty) (ontable ?x)))
  (:action stack
             :parameters (?x ?y)
             :precondition (and (holding ?x) (clear ?y))
             :effect  (and (not (holding ?x)) (not (clear ?y)) (clear ?x)(handempty)
                           (on ?x ?y))) ...
(define (problem BLOCKS_6_1)
   (:domain BLOCKS)
   (:objects F D C E B A)
   (:init (CLEAR A) (CLEAR B) ...  (ONTABLE B) ... (HANDEMPTY))
   (:goal (AND (ON E F) (ON F C) (ON C B) (ON B A) (ON A D))))
```

DWR domain in PDDL

```
;; Specification in PDDL1 of the DWR domain
;; disponível em: http://projects.laas.fr/planning/
(define (domain dock-worker-robot)
(:requirements :strips :typing )
(:types
  location      ; there are several connected locations in the harbor
  pile          ; is attached to a location
                ; it holds a pallet and a stack of containers
  robot         ; holds at most 1 container, only 1 robot per location
  crane         ; belongs to a location to pickup containers
  container)
 (:predicates
   (adjacent ?l1  ?l2 - location)       ; location ?l1 is adjacent ot ?l2
   (attached ?p - pile ?l - location)  ; pile ?p attached to location ?l
   (belong ?k - crane ?l - location)   ; crane ?k belongs to location ?l
   (at ?r - robot ?l - location)       ; robot ?r is at location ?l
   (occupied ?l - location)             ; there is a robot at location ?l
   (loaded ?r - robot ?c - container ) ; robot ?r is loaded with container ?c
   (unloaded ?r - robot)                ; robot ?r is empty
   (holding ?k - crane ?c - container) ; crane ?k is holding a container ?c
   (empty ?k - crane)                         ; crane ?k is empty
   (in ?c - container ?p - pile)        ; container ?c is within pile ?p
   (top ?c - container ?p - pile)      ;  container ?c is on top of pile ?p
   (on ?k1 - container ?k2 - container); container ?k1 is on container ?k2
   )
```

DWR domain in PDDL

```
;; there are 5 operators in this domain:

;; moves a robot between two adjacent locations
 (:action move
    :parameters (?r - robot ?from ?to - location)
    :precondition (and (adjacent ?from ?to)
                  (at ?r ?from) (not (occupied ?to)))
    :effect (and (at ?r ?to) (not (occupied ?from))
                           (occupied ?to) (not (at ?r ?from)) ))

;; loads an empty robot with a container held by a nearby crane
 (:action load
    :parameters (?k - crane ?c - container ?r - robot)
    :vars (?l - location)
    :precondition (and (at ?r ?l) (belong ?k ?l)
                  (holding ?k ?c) (unloaded ?r))
    :effect (and  (loaded ?r ?c) (not (unloaded ?r))
              (empty ?k) (not (holding ?k ?c))))


;; unloads a robot holding a container with a nearby crane
 (:action unload
    :parameters (?k - crane ?c - container ?r - robot)
    :vars (?l - location)
    :precondition (and (belong ?k ?l) (at ?r ?l)
                  (loaded ?r ?c) (empty ?k))
    :effect (and (unloaded ?r) (holding ?k ?c)
            (not (loaded ?r ?c))(not (empty ?k))))
```

DWR domain in PDDL

```
;; takes a container from a pile with a crane
 (:action take
      :parameters (?k - crane ?c - container ?p - pile)
      :vars (?l - location ?else - container)
      :precondition (and (belong ?k ?l)(attached ?p ?l)
                         (empty ?k) (in ?c ?p)
                         (top ?c ?p) (on ?c ?else))
      :effect (and (holding ?k ?c) (top ?else ?p)
                   (not (in ?c ?p)) (not (top ?c ?p))
                   (not (on ?c ?else)) (not (empty ?k))))

;; puts a container held by a crane on a nearby pile
 (:action put
      :parameters (?k - crane ?c - container ?p - pile)
      :vars (?else - container ?l - location)
      :precondition (and (belong ?k ?l) (attached ?p ?l)
                         (holding ?k ?c) (top ?else ?p))
      :effect (and (in ?c ?p) (top ?c ?p) (on ?c ?else)
                   (not (top ?else ?p)) (not (holding ?k ?c))
                   (empty ?k))))
```

DWR problem in PDDL

```
;; a simple DWR problem with 1 robot and 2 locations
(define (problem dwrpb1)
  (:domain dock-worker-robot)

  (:objects
   r1 - robot
   l1 l2 - location
   k1 k2 - crane
   p1 q1 p2 q2 - pile
   ca cb cc cd ce cf pallet - container)

  (:init
   (adjacent l1 l2) (adjacent l2 l1) (attached p1 l1) (attached q1 l1)
   (attached p2 l2) (attached q2 l2) (belong k1 l1)(belong k2 l2)
   (in ca p1)(in cb p1)(in cc p1)(in cd q1)(in ce q1)(in cf q1)(on ca pallet)
   (on cb ca)(on cc cb)(on cd pallet)(on ce cd)(on cf ce)(top cc p1)
   (top cf q1)(top pallet p2)(top pallet q2)(at r1 l1)(unloaded r1)
   (occupied l1)(empty k1)(empty k2))

;; the task is to move all containers to locations l2
;; ca and cc in pile p2, the rest in q2
  (:goal
```

Planejamento como busca no espaço de estados

Como resolver problemas de planejamento clássico?

- Algoritmo STRIPS (70s):
- Busca no espaço de estados: planejamento progressivo e regressivo
- Busca no espaço de planos: Planejamento de Ordem Parcial (80’s) (UCPOP, 1992)
- Busca no grafo de planejamento: Graphplan (1995)
- Busca no espaço de interpretações lógicas: SatPlan (1996)
- Busca heurística: planejamento como uma busca heurística no espaço de estados (1996 ...)
- Planejamento como Verificação de Modelos (1998)

Planejamento como busca no espaço de estados

- A ideia é aplicar os algoritmos de busca para o problema de planejamento:

  -O espaço de busca é um subconjunto do espaço de estados

  -Nós correspondem a estados do mundo

  -Arcos correspondem a transição de estados

  -Um caminho no espaço de busca corresponde a um plano

Planejamento como busca no espaço de estados

- São possíveis ambos: encadeamento para frente e para trás
- Planejamento progressivo
  - Busca por encadeamento para frente:
  - estado inicial -> objetivo
  - Considerar os efeitos de TODAS as ações aplicáveis em um dado estado
- Planejamento regressivo
  - Busca por encadeamento para trás:
  - objetivo -> estado inicial
  - Usa o inverso das ações relevantes para procurar para trás pelo estado inicial.

Planejamento como busca no espaço de estados

- São possíveis ambos: encadeamento para frente e para trás
- Planejamento progressivo
  - Busca por encadeamento para frente:
  - estado inicial -> objetivo
  - Consider
    aplicáve
- Planejamento regressivo
  - Busca por encadeamento para trás:
  -
  - Usa o ir
    pelo est

O uso de heurísticas torna viável a busca para frente

-Há menos caminhos partindo do objetivo do que do estado inicial
-Funciona quando sabemos regredir a partir de um estado para a descrição do estado predecessor. PDDL foi projetada para facilitar a regressão de ações.

Algoritmo progressivo

- Formulação como busca em um espaço de estados:
  - -Estado inicial: estado inicial do problema de planejamento. Literais não presentes são falsos
  - -Ações: ações são aplicáveis em um estado s se as precondições são satisfeitas

$$a_i \in \{a \mid a \in A \wedge pre(a) \subseteq s\}$$

Cálculo do estado sucessor $s' = \gamma(a, s)$

$$s' = s \cup eff+(a_i) \setminus eff-(a_i)$$

Algoritmo progressivo

- ...
  -Teste de objetivo: verifica se o estado satisfaz o objetivo do problema de planejamento
  -Custo do passo: cada ação custa 1

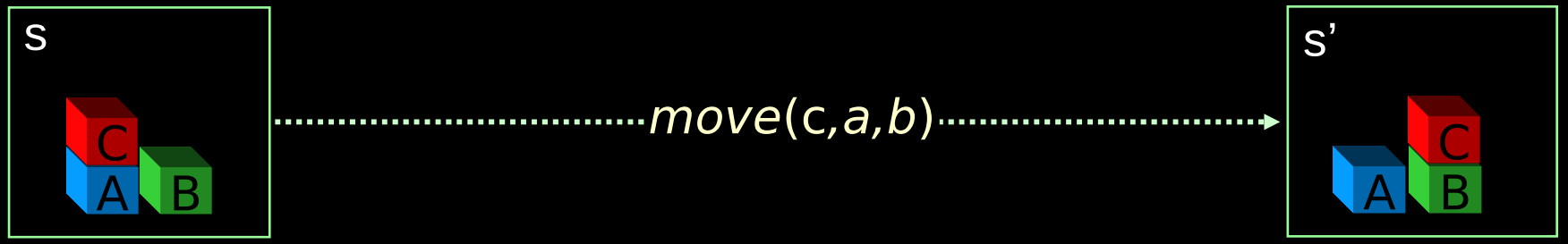
Plano: A concatenação da ação selecionada é feita no fim do plano π

$$\pi \cdot a_i$$

Cálculo do estado sucessor: Exemplo

operator: (*move*(X,Y,Z),
precond: { *limpo*(X), *limpo*(Z), *sobre*(X,Y) },
effects: { *limpo*(Y), *sobre*(X,Z), ¬*limpo*(Z), ¬*sobre*(X,Y) })

substituição = {X/c, Y/a, Z/b}

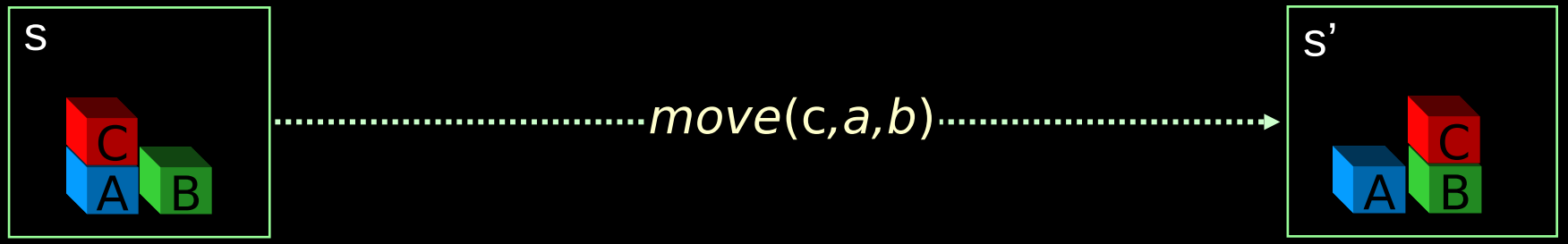


limpo(b)
limpo(c)
sobre (a,mesa)
Sobre(b,mesa)
sobre(c,a)

Cálculo do estado sucessor: Exemplo

action: (*move*(c,a,b),
precond: { *limpo*(c), *limpo*(b), *sobre*(c,a) },
effects: { *limpo*(a), *sobre*(c,b), ¬*limpo*(b), ¬*sobre*(c,a) })

s' = s + efeitos(positivos) – efeitos(negativos)

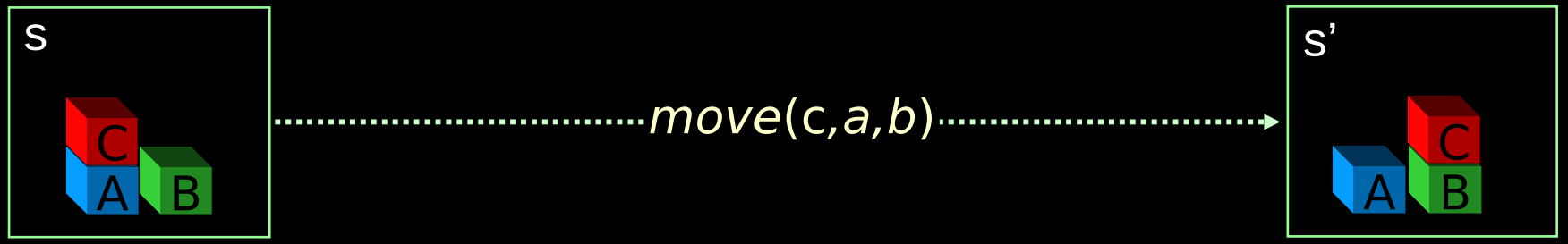


limpo(b)
limpo(c)
sobre (a,mesa)
Sobre(b,mesa)
sobre(c,a)

Cálculo do estado sucessor: Exemplo

action: (*move*(c,a,b),
precond: { *limpo*(c), *limpo*(b), *sobre*(c,a) },
effects: { *limpo*(a), *sobre*(c,b), ¬*limpo*(b), ¬*sobre*(c,a) })

s' = s + efeitos(positivos) – efeitos(negativos)



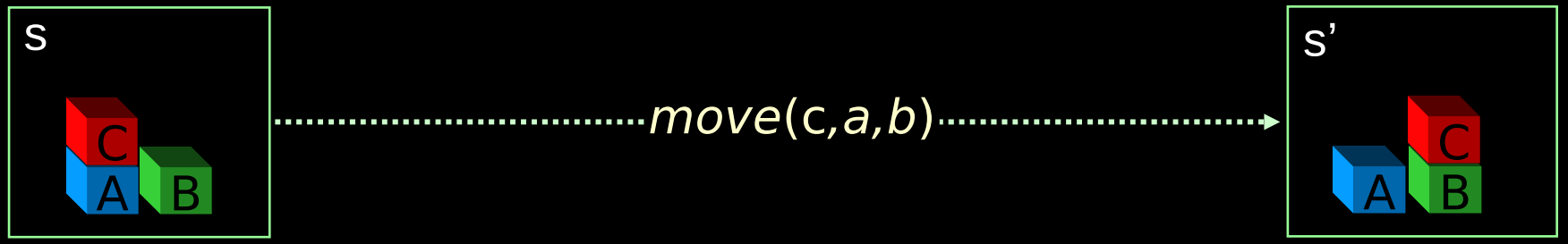
limpo(b)
limpo(c)
sobre (a,mesa)
Sobre(b,mesa)
sobre(c,a)

limpo(b)
limpo(c)
sobre(a,mesa)
sobre(b,mesa)
sobre(c,a)

Cálculo do estado sucessor: Exemplo

action: (*move*(c,a,b),
precond: { *limpo*(c), *limpo*(b), *sobre*(c,a) },
effects: { *limpo*(a), *sobre*(c,b), ¬*limpo*(b), ¬*sobre*(c,a) })

s' = s + efeitos(positivos) – efeitos(negativos)



limpo(b)
limpo(c)
sobre (a,mesa)
Sobre(b,mesa)
sobre(c,a)

limpo(b)
limpo(c)
sobre(a,mesa)
sobre(b,mesa)
sobre(c,a)

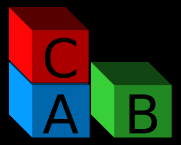
limpo(b)
limpo(c)
sobre(a,mesa)
sobre(b,mesa)
sobre(c,a)
limpo(a)
sobre(c,b)

Cálculo do estado sucessor: Exemplo

action: (*move*(c,a,b),
precond: { *limpo*(c), *limpo*(b), *sobre*(c,a) },
effects: { *limpo*(a), *sobre*(c,b), ¬*limpo*(b), ¬*sobre*(c,a) })

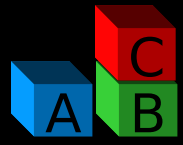
s' = s + efeitos(positivos) – efeitos(negativos)

s



move(c,*a*,*b*)

s'



limpo(b)
limpo(c)
sobre (a,mesa)
Sobre(b,mesa)
sobre(c,a)

limpo(b)
limpo(c)
sobre(a,mesa)
sobre(b,mesa)
sobre(c,a)

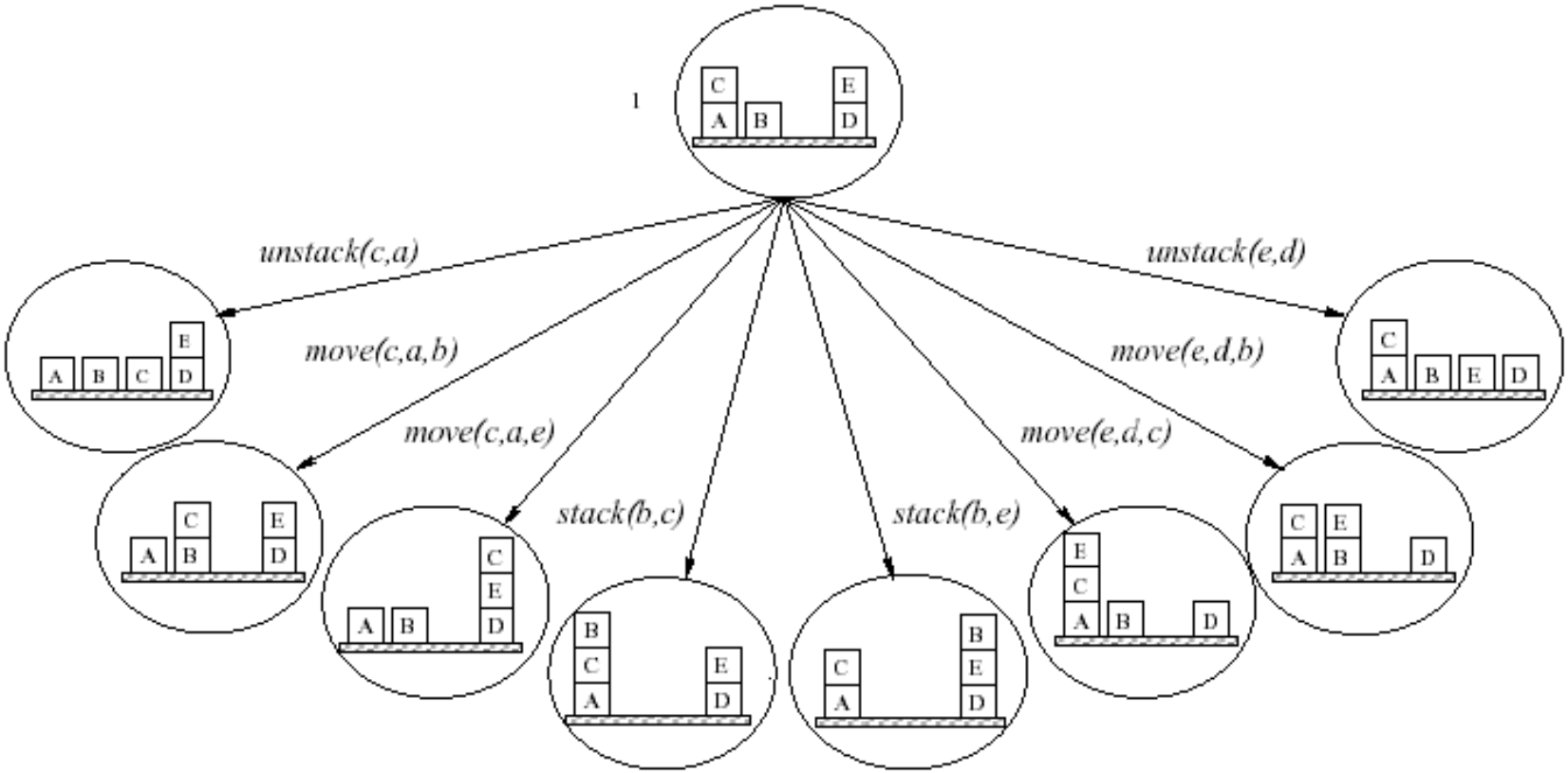
limpo(b)
limpo(c)
sobre(a,mesa)
sobre(b,mesa)
sobre(c,a)
limpo(a)
sobre(c,b)

~~*limpo(b)*~~
limpo(c)
sobre(a,mesa)
sobre(b,mesa)
~~*sobre(c,a)*~~
limpo(a)
sobre(c,b)

Algoritmo progressivo

- Sem símbolos funcionais ... qualquer algoritmo (completo) de busca é um algoritmo de planejamento completo.
- Ineficiente:
  1) Muitas ações irrelevantes são exploradas
  2) A abordagem fracassa sem uma boa heurística.

Planejamento progressivo



Todas as ***ações aplicáveis*** são usadas para gerar sucessores de s

Algoritmo regressivo

Uma vez que na linguagem adotada o Estado Inicial é completo mas a **Meta G é parcial**, isso significa que teremos muitos Estados Meta completos para considerarmos como **raiz** da busca regressiva:

1) Busca paralela a partir de todos os estados meta e escolher a solução ótima (o menor plano) ou

2)Busca com conjunto de estados

Algoritmo regressivo

Como determinar predecessores? É preciso aplicar a função de transição inversa

$$s' = \gamma^{-1}(a, s)$$

-Quais são os estados a partir dos quais a aplicação de uma dada ação leva ao objetivo?

- Vantagem principal: somente ações relevantes são consideradas (fator de ramificação < que progressão).

Algoritmo regressivo

- Considere o problema de ter 20 itens no aeroporto SFO:

Objetivo: Em(c1, sfo) ^ Em(c2, sfo) ^ … ^ Em(c20, sfo)

Ação(**Descarregar**(c1,A,sfo)
PRÉ-CONDIÇÃO: Dentro(c1,A) ^ Em(a,sfo)
^ Carga(c1) ^ Avião(A) ^ Aeroporto(sfo)
EFEITO: Em(c1,sfo) ^ ¬Dentro(c1,A))

Descarregar é uma ação relevante?

er estado

Dentro(c1,A) ^ Em(A,sfo)^ Em(c2, sfo) ^ … ^ Em(c20, sfo)

Note que o subobjetivo Em(c1,sfo) não deve estar presente neste estado.

Algoritmo regressivo

- Considere o problema de ter 20 itens no aeroporto SFO:

Objetivo: Em(c1, sfo) ^ Em(c2, sfo) ^ … ^ Em(c20, sfo)

Ação(**Descarregar**(c1,A,sfo)
PRÉ-CONDIÇÃO: Dentro(c1,A) ^ Em(a,sfo)
^ Carga(c1) ^ Avião(A) ^ Aeroporto(sfo)
EFEITO: Em(c1,sfo) ^ ¬Dentro(c1,A))

Funciona somente se as precondições forem satisfeitas, i.e. qualquer estado predecessor deve incluir estas precondições

Predecessor:
Dentro(c1,A) ^ Em(A,sfo)^ Em(c2, sfo) ^ … ^ Em(c20, sfo)

Note que o subobjetivo Em(c1,sfo) não deve estar presente neste estado.

Algoritmo regressivo

Uma ação é relevante para um objetivo s se:

1) ela alcança um dos elementos da conjunção do objetivo e

2) não deve desfazer os outros literais desejados (i.e, a ação deve ser consistente)

$$a_i \in \{a \mid a \in A \wedge \mathtt{eff}^+(\alpha) \cap s \neq \emptyset \wedge \mathtt{eff}^-(a) \cap s = \emptyset\}$$

Algoritmo regressivo: Processo geral para a construção de predecessores

Dada uma descrição de objetivo G, seja A uma ação relevante e consistente. O predecessor correspondente é descrito a seguir:

- Quaisquer efeitos positivos de A que aparecem em G são eliminados.
- Quaisquer efeitos negativos de A que aparecem em G são adicionados.
- Cada literal da precondição de A é adicionado, a menos que já apareça.

$$s' = \gamma^{-1}(a, s)$$

$$s' = s \cup pre(a) \cup eff^{-}(a) \setminus eff^{+}(a)$$

Algoritmo regressivo

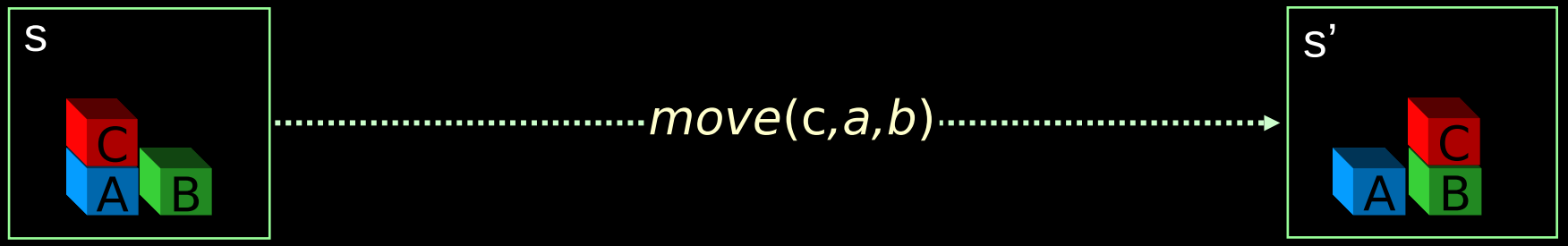
Plano: A concatenação de uma ação selecionada é feita no início do plano π

$$-a_i.\pi$$

Cálculo do estado predecessor: Exemplo

action: (*move*(c,a,b),
precond: { *limpo*(c), *limpo*(b), *sobre*(c,a) },
effects: { *limpo*(a), *sobre*(c,b), ¬*limpo*(b), ¬*sobre*(c,a) })

$s' = s \cup pre(a) \cup eff^{-}(a) \setminus eff^{+}(a)$



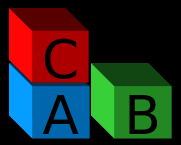
limpo(c)
sobre(a,mesa)
sobre(b,mesa)
limpo(a)
sobre(c,b)

Cálculo do estado predecessor: Exemplo

action: (*move*(c,a,b),
precond: { *limpo*(c), *limpo*(b), *sobre*(c,a) },
effects: { *limpo*(a), *sobre*(c,b), ¬*limpo*(b), ¬*sobre*(c,a) })

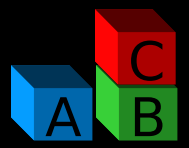
$s' = s \cup pre(a) \cup eff^{-}(a) \setminus eff^{+}(a)$

s



move(c,*a*,*b*)

s'



limpo(c)
sobre(a,mesa)
sobre(b,mesa)
limpo(a)
sobre(c,b)
limpo(b)
limpo(c)
sobre(c,a)

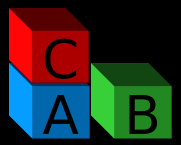
limpo(c)
sobre(a,mesa)
sobre(b,mesa)
limpo(a)
sobre(c,b)

Cálculo do estado predecessor: Exemplo

action: (*move*(c,a,b),
precond: { *limpo*(c), *limpo*(b), *sobre*(c,a) },
effects: { *limpo*(a), *sobre*(c,b), ¬*limpo*(b), ¬*sobre*(c,a) })

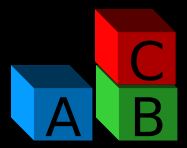
$s' = s \cup pre(a) \cup eff^{-}(a) \setminus eff^{+}(a)$

s



move(c,*a*,*b*)

s'



limpo(b)
sobre(c,a)
limpo(c)
sobre(a,mesa)
sobre(b,mesa)
limpo(a)
sobre(c,b)
limpo(b)
limpo(c)
sobre(c,a)

limpo(c)
sobre(a,mesa)
sobre(b,mesa)
limpo(a)
sobre(c,b)
limpo(b)
limpo(c)
sobre(c,a)

limpo(c)
sobre(a,mesa)
sobre(b,mesa)
limpo(a)
sobre(c,b)

Cálculo do estado predecessor: Exemplo

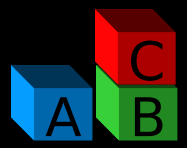
action: (*move*(c,a,b),
precond: { *limpo*(c), *limpo*(b), *sobre*(c,a) },
effects: { *limpo*(a), *sobre*(c,b), ¬*limpo*(b), ¬*sobre*(c,a) })

$s' = s \cup pre(a) \cup eff^{-}(a) \setminus eff^{+}(a)$

s

move(c,*a*,*b*)

s'



limpo(b)
sobre(c,a)
limpo(c)
sobre(a,mesa)
sobre(b,mesa)
~~*limpo(a)*~~
~~*sobre(c,b)*~~
limpo(b)
limpo(c)
sobre(c,a)

limpo(b)
sobre(c,a)
limpo(c)
sobre(a,mesa)
sobre(b,mesa)
limpo(a)
sobre(c,b)
limpo(b)
limpo(c)
sobre(c,a)

limpo(c)
sobre(a,mesa)
sobre(b,mesa)
limpo(a)
sobre(c,b)
limpo(b)
limpo(c)
sobre(c,a)

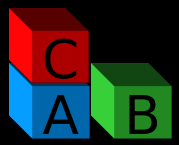
limpo(c)
sobre(a,mesa)
sobre(b,mesa)
limpo(a)
sobre(c,b)

Cálculo do estado predecessor: Exemplo

action: (*move*(c,a,b),
precond: { *limpo*(c), *limpo*(b), *sobre*(c,a) },
effects: { *limpo*(a), *sobre*(c,b), ¬*limpo*(b), ¬*sobre*(c,a) })

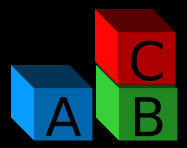
$s' = s \cup pre(a) \cup eff^{-}(a) \setminus eff^{+}(a)$

s



move(c,*a*,*b*)

s'



limpo(b)
sobre(c,a)
sobre(a,mesa)
sobre(b,mesa)
limpo(c)

limpo(b)
sobre(c,a)
limpo(c)
sobre(a,mesa)
sobre(b,mesa)
limpo(a)
sobre(c,b)
limpo(b)
limpo(c)
sobre(c,a)

limpo(c)
sobre(a,mesa)
sobre(b,mesa)
limpo(a)
sobre(c,b)
limpo(b)
limpo(c)
sobre(c,a)

limpo(c)
sobre(a,mesa)
sobre(b,mesa)
limpo(a)
sobre(c,b)

Algoritmo regressivo

- Qualquer dos algoritmos de busca pode ser usado para executar a busca regressiva.
- O término acontece quando é gerada uma descrição de predecessor que é satisfeita pelo estado inicial do problema de planejamento.
  - No caso de 1a ordem, a satisfação pode requerer uma substituição de variáveis na descrição do predecessor.

Algoritmo regressivo

- Considere o problema de ter 20 itens no aeroporto SFO:

Objetivo: Em(c1, sfo) ^ Em(c2, sfo) ^ … ^ Em(c20, sfo)

Ação(**Descarregar**(c1,A,sfo)
PRÉ-CONDIÇÃO: Dentro(c1,A) ^ Em(a,sfo)
^ Carga(c1) ^ Avião(A) ^ Aeroporto(sfo)
EFEITO: Em(c1,sfo) ^ ¬Dentro(c1,A))

Funciona somente se as precondições forem satisfeitas, i.e. qualquer estado predecessor deve incluir estas precondições

Predecessor:
Dentro(c1,A) ^ Em(A,sfo)^ Em(c2, sfo) ^ … ^ Em(c20, sfo)

Note que o subobjetivo Em(c1,sfo) não deve estar presente neste estado.

Algoritmo regressivo

- Considere o problema de ter 20 itens no aeroporto SFO:

Objetivo: Em(c1, sfo) ^ Em(c2, sfo) ^ … ^ Em(c20, sfo)

Ação(**Descarregar**(c1,A,sfo)
PRÉ-CONDIÇÃO: Dentro(c1,A) ^ Em(a,sfo)
^ Carga(c1) ^ Avião(A) ^ Aeroporto(sfo)
EFEITO: Em(c1,sfo) ^ ¬Dentro(c1,A))

Funciona somente se as precondiçõ tado
predecessor deve incluir estas prec

Predecessor:
Dentro(c1,A) ^ Em(A,sf

Note que o subobjetivo Em(c1,sfo) n o.

Algoritmo regressivo “Lifted”: usa operadores parcialmente instanciados no lugar de ações.

Planejamento Progressivo x Regressivo

- A busca regressiva usa conjuntos de estados em vez de estados individuais
  - torna mais difícil chegar a uma boa heurística

- A maioria de sistemas atuais preferem a busca para frente utilizando heurísticas.