Universidade Federal do Amazonas
Intituto de Computação da UFAM

Planejador para empilhar blocos de diferentes dimensões

Carlos Andres Ramos Losada Junior
Davi Israel Abtibol Carvalho
Gabriel Toshiyuki Batista Toyoda

2024

Introdução

Este código implementa um planejador de blocos em Prolog, que determina a sequência de movimentos para reorganizar blocos de um estado inicial para um estado final desejado. O código define:

Blocos e Locais

- `block(X)`: Define os blocos 'a', 'b', 'c' e 'd'.
- `place(X)`: Define os locais numerados de 0 a 5, representando as posições possíveis para os blocos.

Propriedades dos Blocos

- `height(Block, Height)`: Define a altura de cada bloco como 1.
- `length(Block, Length)`: Define o comprimento de cada bloco, variando de 1 a 3.

Estados

- `final(List)`: Define o estado final desejado, listando as posições dos blocos e as posições livres.
- `state(List)`: Define o estado inicial dos blocos, similar ao estado final.
- `state0`, `state3`, `state4`, `state6`, `state7`: Definem estados intermediários para testes.

Predicados Auxiliares

- `clearInterval(X1, X2, Y, List)`: Gera uma lista de posições livres (representadas por `clear(X,Y)`) em um intervalo de $X1$ a $X2$, na altura Y.

Regras de Movimento (Predicado `can`)

O predicado `can(move(Block, From, To), List)` verifica se um movimento é possível, considerando:

- Se o bloco existe e suas dimensões.

- Se as posições inicial e final são válidas.
- Se a posição final está livre.
- Se o movimento respeita as regras de estabilidade, garantindo que os blocos sejam suportados adequadamente.
- Existem quatro versões do predicado `can`, cada uma lidando com um comprimento de bloco específico (1, 2, ímpar maior que 2, par maior que 2).

Efeitos dos Movimentos (Predicados `adds` e `deletes`)

- `adds(Move, List)`: Define as novas relações (posições de blocos e posições livres) que são adicionadas após um movimento.
- `deletes(Move, List)`: Define as relações que são removidas após um movimento.

Observações

- O código assume que um bloco só pode ser movido se estiver no topo de uma pilha.
- A estabilidade é verificada considerando se os blocos têm suporte suficiente nas laterais.
- O código não implementa a lógica completa do planejador, faltando predicados como `plan`, `satisfied`, `select`, `achieves`, `preserves` e `regress`, que são necessários para gerar o plano de movimento.

Código em Prolog disponível no seguinte repositório do GitHub: `https://github.com/CarlosARL/T1_MundoDosBlocos_02`

O código Prolog fornecido implementa um planejador de ações para um domínio de blocos, utilizando uma estratégia de análise regressiva (backward chaining) e o princípio Means-Ends Analysis. Ele busca encontrar uma sequência de ações válidas que transformem um estado inicial em um estado final desejado, considerando restrições do ambiente e dos blocos.

Funcionalidades Principais

- Representação de Estados: Os estados são representados por listas de termos Prolog, que descrevem as posições dos blocos e as posições livres na área de trabalho.

- Ações: O código define ações como "mover um bloco de uma posição para outra", considerando as restrições de movimento, como a necessidade de ter espaço livre no destino.
- Planejamento: A função `plan/3` implementa a lógica principal de planejamento, utilizando recursão para explorar diferentes sequências de ações.
- `Satisfied/2`: Verifica se um determinado estado satisfaz todas as metas especificadas.
- `Select/3`: Seleciona uma meta que ainda não foi alcançada no estado atual.
- `Achieves/2`: Verifica se uma ação específica contribui para atingir uma meta.
- `Preserves/2`: Garante que uma ação não invalida nenhuma das metas já alcançadas.
- `Regress/3`: Realiza a regressão de metas, ou seja, determina as submetas necessárias para alcançar um determinado objetivo.
- Impossibilidade: Define condições impossíveis de serem satisfeitas, como um bloco ocupar duas posições simultaneamente.
- Utilidades: Funções auxiliares, como `addnew/3` (adiciona elementos a uma lista evitando duplicatas) e `delete_all/3` (remove elementos de uma lista que estão presentes em outra).

Exemplo de Uso

A função `to_plan/1` demonstra como usar o planejador para encontrar uma sequência de ações que leve o sistema do estado `state6` para o estado `state7`. A variável `L` será instanciada com a lista de ações encontradas pelo planejador.

Código em Prolog disponível no seguinte repositório do GitHub: `https://github.com/CarlosARL/T1_MundoDosBlocos_02`

Esta questão aborda a criação manual de planos para atingir estados finais desejados, utilizando a linguagem de planejamento desenvolvida. Cada subseção (3.1 a 3.4) apresenta um cenário específico, com:

- Estado inicial: Descrição do estado inicial do mundo (blocos em suas posições iniciais).
- Estado final: Descrição do estado desejado (blocos em suas posições finais).
- Ações estilo backtracking: Sequência de ações que, se executadas na ordem inversa, levarão o estado inicial ao estado final.

- Ações reais: Sequência de ações que, se executadas na ordem direta, levarão o estado inicial ao estado final.

Questão 3.1

- Estado inicial: `[on(a, at(1,0)), on(b, at(5,0)), on(c, at(0,0)), on(d,at(2,0)), clear(0,2), clear(1,1), clear(2,1), clear(3,1), clear(4,1), clear(5,1)]`
- Estado final: `[on(a, at(3,0)), on(b, at(5,0)), on(c, at(0,0)), on(d,at(3,1)), clear(2,0), clear(4,0), clear(0,1), clear(1,1), clear(3,2), clear(4,2), clear(5,2)]`
- Ações estilo backtracking: `move(a, at(0,1), at(5,0)), move(d, at(2,0), at(0,1)), move(a, at(5,0), at(3,0)), move(d, at(0,1), at(3,1))`
- Ações reais: `move(d, at(3,1), at(0,1), move(a, at(3,0), at(5,1)), move(d, at(0,1), at(2,0)), move(a, at(5,1), at(0,1))`

Questão 3.2

- Estado inicial: `[on(a, at(4,1)), on(b, at(5,1)), on(c, at(4,2)), on(d, at(3,0)), clear(0,0), clear(1,0), clear(2,0), clear(3,1), clear(4,3), clear(5,3)]`
- Estado final: `[on(a, at(0,1)), on(b, at(5,0)), on(c, at(0,0)), on(d, at(2,0)), clear(0,2), clear(1,1), clear(2,1), clear(3,1), clear(4,1), clear(5,1)]`
- Ações estilo backtracking: `move(c, at(0,0), at(4,2)), move(a, at(0,1), at(4,1)), move(b, at(1,1), at(5,1)), move(d, at(2,0), at(3,0)), move(b, at(1,1), at(5,0))`
- Ações reais: `move(b, at(1,1), at(5,0)), move(d, at(2,0), at(3,0)), move(b, at(1,1), at(5,1)), move(a, at(0,1), at(4,1)), move(c, at(0,0), at(4,2))`

Questão 3.3

- Estado inicial: `[on(a, at(4,2)), on(b,at(5,2)), on(c, at(4,1)), on(d, at(3,0)), clear(0,0), clear(1,0), clear(2,0), clear(3,1), clear(4,3), clear(5,3)]`
- Estado final: `[on(a, at(0,1)), on(b, at(5,0)), on(c, at(0,0)), on(d, at(2,0)), clear(0,2), clear(1,1), clear(2,1), clear(3,1), clear(4,1), clear(5,1)]`
- Ações estilo backtracking: `move(a, at(4,2), at(3,1)), move(b, at(5,2), at(2,0)), move(c, at(4,1), at(0,0), move(a, at(3,1), at(0,1)), move(b, at(2,0), at(1,1)), move(d, at(3,0), at(2,0), move(b, at(1,1), at(5,0))`

- Ações reais: `move(b, at(5,0), at(1,1)), move(d, at(2,0), at(3,0)), move(b, at(1,1), at(2,0)), move(a, at(0,1), at(3,1)), move(c, at(0,0), at(4,1), move(b, at(2,0), at(5,2)), move(a, at(3,1), at(4,2))`

Questão 3.4

- Estado inicial: `[on(a, at(4,2)), on(b,at(5,2)), on(c, at(4,1)), on(d, at(3,0)), clear(0,0), clear(1,0), clear(2,0), clear(3,1), clear(4,3), clear(5,3)]`
- Estado final: `[on(a, at(0,1)), on(b, at(5,0)), on(c, at(0,0)), on(d, at(2,0)), clear(0,2), clear(1,1), clear(2,1), clear(3,1), clear(4,1), clear(5,1)]`
- Ações estilo backtracking: `move(b, at(5,0), at(4,0)), move(d, at(0,1), at(3,1)), move(a, at(2,0), at(1,1)), move(d, at(3,0), at(0,2)), move(b, at(4,0), at(5,0)), move(d, at(0,2), at(2,0)), move(a, at(1,1), at(0,1))`
- Ações reais: `move(a, at(0,1), at(1,1)), move(d, at(2,0), at(0,2)), move(b, at(5,0), at(4,0)), move(d, at(0,2), at(3,1)), move(a, at(1,1), at(2,0)), move(d, at(3,1), at(0,1)), move(b, at(4,0), at(5,0))`

Questão 3.5

O objetivo é, a partir de um estado desejado, determinar quais objetivos precisam ser satisfeitos em um estado anterior para que, através de uma ação específica, se possa alcançar o estado desejado.

Algoritmo de regressão de objetivos

- Estado atual: Se a lista de objetivos já está satisfeita no estado atual, nada precisa ser feito.
- Seleção de objetivo: Caso contrário, seleciona-se um objetivo que não está no estado inicial e a ação que o atinge.
- Regressão: Os objetivos são regredidos através da ação, gerando novos objetivos.
- Busca de plano: Um novo plano é então buscado, a partir do estado inicial, para alcançar os novos objetivos.

Heurísticas e Eficiência

- Para encontrar as ações que formam o plano, utiliza-se a heurística de amplitude em primeiro lugar, priorizando planos menores.

- Predicados como "impossible"são usados para eliminar ações impossíveis, otimizando o processo.

Situação 2

- Para esta situação, começamos pelo estado final representado pela seguinte lista de predicados: `[on(a, at(4,2)), on(b, at(5,2)), on(c, at(4,1)), on(d, at(3,0)), clear(0,0), clear(1,0), clear(2,0), clear(3,1), clear(4,3), clear(5,3)]`
- Porém, desejamos alcançar o estado objetivo: `Goal: [on(a, at(0,1)), on(b, at(1,1)), on(c, at(0,0)), on(d, at(3,0)), clear(0,2), clear(1,2), clear(2,0), clear(3,1), clear(4,1), clear(5,1)]`
- Portanto, precisamos encontrar um plano para atingir esse estado por meio de backtracking. A seguir, listarei a sequência de passos:
  - Ação 1: `move(b, at(2,0), at(5,2))`
    - Removidos: `on(b, at(2,0)), clear(2,1), clear(5,2)`
    - Adicionados: `on(b, at(5,2)), clear(2,0), clear(5,3)`
    - Estado atual: `[on(a, at(4,2)), on(b, at(2,0)), on(c, at(4,1)), on(d, at(3,0)), clear(0,0), clear(1,0), clear(2,1), clear(3,1), clear(4,3), clear(5,2)]`
  - Ação 2: `move(a, at(2,1), at(4,2))`
    - Removidos: `on(a, at(2,1)), clear(2,2), clear(4,2)`
    - Adicionados: `on(a, at(4,2)), clear(2,1), clear(4,3)`
    - Estado atual: `[on(a, at(2,1)), on(b, at(2,0)), on(c, at(4,1)), on(d, at(3,0)), clear(0,0), clear(1,0), clear(2,2), clear(3,1), clear(4,2), clear(5,2)]`
  - Ação 3: `move(c, at(0,0), at(4,1))`
    - Removidos: `on(c, at(0,0)), clear(0,1), clear(1,1), clear(4,1), clear(5,1)`
    - Adicionados: `on(c, at(4,1)), clear(0,0), clear(1,0), clear(4,2), clear(5,2)`
    - Estado atual: `[on(a, at(2,1)), on(b, at(2,0)), on(c, at(0,0)), on(d, at(3,0)), clear(0,1), clear(1,1), clear(2,2), clear(3,1), clear(4,1), clear(5,1)]`
  - Ação 4: `move(a, at(0,1), at(2,1))`
    - Removidos: `on(a, at(0,1)), clear(0,2), clear(2,1)`

    - Adicionados: `on(a, at(2,1)), clear(0,1), clear(2,2)`
    - Estado atual: `[on(a, at(0,1)), on(b, at(2,0)), on(c, at(0,0)), on(d, at(3,0)), clear(0,2), clear(1,1), clear(2,1), clear(3,1), clear(4,1), clear(5,1)]`
  - Ação 5: `move(b, at(1,1), at(2,0))`
    - Removidos: `on(b, at(1,1)), clear(1,2), clear(2,0)`
    - Adicionados: `on(b, at(2,0)), clear(1,1), clear(2,1)`
    - Estado atual: `[on(a, at(0,1)), on(b, at(1,1)), on(c, at(0,0)), on(d, at(3,0)), clear(0,2), clear(1,2), clear(2,0), clear(3,1), clear(4,1), clear(5,1)]`

- Note que o estado atual agora é igual ao estado objetivo, seguindo as ações de 5 até 1, nessa ordem.

Situação 3

- Para esta situação, começamos pelo estado final representado pela seguinte lista de predicados: `[on(a, at(0,1)), on(b, at(1,1)), on(c, at(0,0)), on(d, at(3,0)), clear(0,2), clear(1,2), clear(2,0), clear(3,1), clear(4,1), clear(5,1)]`

- No entanto, queremos alcançar o estado objetivo: `Goal: [on(a, at(3,0)), on(b, at(5,0)), on(c, at(0,0)), on(d, at(3,1)), clear(0,1), clear(1,1), clear(2,0), clear(3,2), clear(4,2), clear(5,2)]`

- Portanto, precisamos encontrar um plano para atingir esse estado por meio de backtracking. A seguir, listarei a sequência de passos:
  - Ação 1: `move(d, at(2,0), at(3,0))`
    - Removidos: `on(d, at(2,0)), clear(2,1), clear(3,1), clear(4,1), clear(5,0)`
    - Adicionados: `on(d, at(3,0)), clear(2,0), clear(5,1)`
    - Estado atual: `[on(a, at(0,1)), on(b, at(1,1)), on(c, at(0,0)), on(d, at(2,0)), clear(0,2), clear(1,2), clear(2,1), clear(3,1), clear(4,1), clear(5,0)]`
  - Ação 2: `move(b, at(5,0), at(1,1))`
    - Removidos: `on(b, at(5,0)), clear(5,1), clear(1,1)`
    - Adicionados: `on(b, at(1,1)), clear(5,0), clear(1,2)`
    - Estado atual: `[on(a, at(0,1)), on(b, at(5,0)), on(c, at(0,0)), on(d, at(2,0)), clear(0,2), clear(1,1), clear(2,1), clear(3,1), clear(4,1), clear(5,1)]`

- Ação 3: `move(a, at(5,1), at(0,1))`
  - Removidos: `on(a, at(5,1)), clear(5,2), clear(0,1)`
  - Adicionados: `on(a, at(0,1)), clear(5,1), clear(0,2)`
  - Estado atual: `[on(a, at(5,1)), on(b, at(5,0)), on(c, at(0,0)), on(d, at(2,0)), clear(0,1), clear(1,1), clear(2,1), clear(3,1), clear(4,1), clear(5,2)]`
- Ação 4: `move(d, at(0,1), at(2,0))`
  - Removidos: `on(d, at(0,1)), clear(2,0), clear(3,0), clear(4,0), clear(0,2), clear(1,2), clear(2,2)`
  - Adicionados: `on(d, at(2,0)), clear(0,1), clear(1,1), clear(2,1), clear(3,1), clear(4,1)`
  - Estado atual: `[on(a, at(5,1)), on(b, at(5,0)), on(c, at(0,0)), on(d, at(0,1)), clear(0,2), clear(1,2), clear(2,2), clear(2,0), clear(3,0), clear(4,0), clear(5,2)]`
- Ação 5: `move(a, at(3,0), at(5,1))`
  - Removidos: `on(a, at(3,0)), clear(3,1), clear(5,1)`
  - Adicionados: `on(a, at(5,1)), clear(3,0), clear(5,2)`
  - Estado atual: `[on(a, at(3,0)), on(b, at(5,0)), on(c, at(0,0)), on(d, at(0,1)), clear(0,2), clear(1,2), clear(2,2), clear(3,1), clear(4,0), clear(5,1)]`
- Ação 6: `move(d, at(3,1), at(0,1))`
  - Removidos: `on(d, at(3,1)), clear(0,1), clear(1,1), clear(3,2), clear(4,2), clear(5,2)`
  - Adicionados: `on(d, at(0,1)), clear(3,1), clear(5,1), clear(0,2), clear(1,2), clear(2,2), clear(4,0)`
  - Estado atual: `[on(a, at(3,0)), on(b, at(5,0)), on(c, at(0,0)), on(d, at(3,1)), clear(0,1), clear(1,1), clear(2,0), clear(3,2), clear(4,2), clear(5,2)]`

- Note que o estado atual agora é igual ao estado objetivo, seguindo as ações de 6 até 1, nessa ordem.