Planejamento em Inteligência Artificial

Silvio do Lago Pereira

17 de novembro de 2003

1 Introdução

Essencialmente, a *tarefa de planejamento* em Inteligência Artificial consiste em encontrar uma seqüência de ações que, quando executada a partir de uma situação inicial do mundo, leva a uma determinada situação meta desejada.

Nessa seção, definimos o problema de planejamento clássico[1] bem como algoritmos básicos para solucioná-lo; na seção 2, consideramos planejamento como busca heurística e analisamos diferentes heurísticas que podem ser utilizadas; na seção 3, apresentamos a linguagem Pddl e suas extensões; na seção 4, analisamos resultados obtidos a partir de experimentos realizados com planejadores implementados como algoritmos de busca heurística; na seção 5, estendemos o planejamento clássico com as idéias de ações durativas e consumo de recursos métricos e, finalmente, na seção 6, modelamos o problema proposto pelo Roadef'2005 como uma tarefa planejamento temporal métrico.

1.1 O problema de planejamento

Formalmente, uma *situação* do mundo é representada por um conjunto Σ de átomos que denotam proposições verdadeiras nessa situação. Uma *ação* é representada por uma tupla α, da forma $\langle pre(\alpha), add(\alpha), del(\alpha)\rangle$, onde $pre(\alpha)$ é um conjunto de átomos que denotam proposições que devem ser verdadeiras na situação em que a ação α é executada (*i.e.* precondições da ação); $add(\alpha)$ é um conjunto de átomos que denotam proposições que passam a ser verdadeiras, após a execução da ação α (*i.e.* efeitos positivos da ação), e $del(\alpha)$ é um conjunto de átomos que denotam proposições que passam a ser falsas após a execução da ação α (*i.e.* efeitos negativos da ação).

Dada uma situação Σ e uma ação α, dizemos que α é *aplicável* a Σ se e só se $pre(\alpha) \subseteq \Sigma$. Ademais, se uma ação α é aplicável a uma situação Σ, então a situação

[1] No planejamento clássico são feitas as seguintes suposições: *tempo atômico*, *efeitos determinísticos*, *onisciência* e *causa de mudança única*.

resultante da execução dessa ação é representada pelo conjunto $\Sigma + add(\alpha) - del(\alpha)$, denotado por $res(\alpha, \Sigma)$.

Um *problema de planejamento* é definido por uma tupla $\mathcal{P}$, da forma $\langle \mathcal{A}, \mathcal{I}, \mathcal{G} \rangle$, onde $\mathcal{A}$ é o conjunto de *ações do domínio* de planejamento considerado, $\mathcal{I}$ é uma *situação inicial* e $\mathcal{G}$ é a descrição da meta de planejamento. Dizemos que uma situação Σ é uma *situação meta* para um problema $\mathcal{P}$ se e só se $\Sigma \models \mathcal{G}$.

Dado um problema de planejamento $\mathcal{P} = \langle \mathcal{A}, \mathcal{I}, \mathcal{G} \rangle$ e uma seqüência de ações $\Pi = \langle \alpha_1, \alpha_2, \ldots, \alpha_n \rangle$, dizemos que Π é uma solução (*i.e. plano*) para o problema de planejamento $\mathcal{P}$ se e só se:

- Π contém apenas ações do domínio considerado, *i.e.*, $\alpha_i \in \mathcal{A}$, para $1 \leq i \leq n$;
- Π é executável, *i.e.*, $pre(\alpha_1) \subseteq \mathcal{I}$ e, para $1 < i \leq n$, $pre(\alpha_i) \subseteq res(\alpha_{i-1}, \Sigma)$;
- Π leva a uma situação meta, *i.e.*, $res(\alpha_n, res(\alpha_{n-1}, res(\ldots, res(\alpha_1, \mathcal{I})))) \models \mathcal{G}$.

1.2 Busca no espaço de estados

Um *espaço de estados* consiste de um conjunto finito de estados S, um conjunto finito de ações A e uma função de transição f que descreve como as ações mapeiam um estado em outro. Um espaço de estados juntamente com um estado inicial s_0 e um conjunto G de estados metas é denominado *modelo de estados*.

Um modelo de estados é uma tupla $\langle S, s_0, G, A, f \rangle$, onde S é um conjunto finito não-vazio de estados, $s_0 \in S$ é o estado inicial, $G \subseteq S$ é um conjunto não-vazio de estados metas, $A(s) \subseteq A$ denota as ações aplicáveis em cada estado $s \in S$, e $f : A(s) \times S \mapsto S$ denota a função de transição de estados.

Claramente, um problema de planejamento $\mathcal{P} = \langle \mathcal{A}, \mathcal{I}, \mathcal{G} \rangle$ define um modelo de estados. Esse modelo de estados pode ser representado por um grafo $\mathbb{G}$ cujos vértices representam situações do mundo e cujos arcos representam ações que transformam uma situação em outra. Sendo assim, encontrar uma solução para um problema de planejamento $\mathcal{P}$ equivale a encontrar um caminho em $\mathbb{G}$, rotulado com ações em $\mathcal{A}$, que leve do vértice que representa a situação inicial $\mathcal{I}$ a um vértice que representa uma situação meta $\Sigma_n \in \mathcal{G}$.

Uma vantagem em modelar o problema de planejamento como busca em grafos é que isso nos permite aplicar diversos algoritmos de *força bruta* ou *heurísticos* bem conhecidos na literatura da área.

A busca no grafo que representa o espaço de estados para um problema de planejamento pode ser feita de duas maneiras distintas: *progressiva*, *i.e.* a partir do vértice que representa a situação inicial, tentamos encontrar um vértice representando uma situação meta; ou *regressiva*, *i.e.* a partir de um nó que representa um estado meta, tentamos encontrar o vértice que representa o estado inicial.

1.2.1 Planejamento como busca progressiva

O algoritmo não-determinístico $Prog$, apresentado na tabela 1, realiza uma busca progressiva num espaço de estados. Ele é chamado inicialmente com o estado corrente Σ igual ao estado inicial do problema e com Π sendo uma seqüência vazia de ações. No início de cada iteração, o algoritmo (*i*) verifica se o estado corrente satisfaz a meta de planejamento e, em caso afirmativo, devolve o plano Π como solução do problema. Senão, (*ii*) o algoritmo *escolhe não-deterministicamente* uma ação aplicável ao estado corrente que seja apropriada e (*iii*) anexa-a ao final do plano, continuando a busca no estado resultante de sua aplicação. Finalmente, se uma tal ação não pode ser escolhida, (*iv*) o algoritmo devolve `falha`.

`Algoritmo` $Prog(\mathcal{A}, \Sigma, \mathcal{G}, \Pi)$

`Entrada:` A descrição das ações do domínio $\mathcal{A}$
A descrição do estado corrente Σ
A descrição da meta de planejamento $\mathcal{G}$
Um plano parcialmente especificado Π
`Saída:` `falha` ou um plano solução Π

`Início`
`Se` $\Sigma \models \mathcal{G}$ `então devolva` Π
`Escolha` $\alpha \in \mathcal{A}$ `tal que` $pre(\alpha) \subseteq \Sigma$
`Se` *tal escolha é possível* `então devolva` $Prog(\mathcal{A}, res(\alpha, \Sigma), \mathcal{G}, \Pi \circ \alpha)$
`senão devolva falha`
`Fim`

Tabela 1: *Busca progressiva no espaço de estados.*

1.2.2 Planejamento como busca regressiva

A idéia de regressão consiste em utilizar ações relevantes a uma determinada situação para regredir dessa situação a uma situação prévia. Dizemos que um operador α é *relevante* a uma situação Σ se e só se se $add(\alpha) \cap \Sigma \neq \emptyset$ e $del(\alpha) \cap \Sigma = \emptyset$. Ademais, a *aplicação inversa* da ação relevante α a uma situação Σ regride a um estado prévio representado pelo conjunto $\Sigma - add(\alpha) + del(\alpha)$, denotado por $rev^{-1}(\alpha, \Sigma)$.

O algoritmo não-determinístico $Regr$, apresentado na tabela 2, realiza uma busca regressiva num espaço de estados. Ele é chamado inicialmente com o estado corrente Σ igual a um estado meta do problema e com Π sendo uma seqüência vazia de ações. No início de cada iteração, o algoritmo (*i*) verifica se o estado corrente é satisfeito pelo estado inicial e, em caso afirmativo, devolve o plano Π como solução do problema. Senão, (*ii*) o algoritmo *escolhe não-deterministicamente* uma ação *relevante* ao estado corrente que seja apropriada e (*iii*) a insere no início do plano, continuando a busca

no estado anterior, resultante de sua aplicação inversa. Finalmente, se uma tal ação não pode ser escolhida, (*iv*) o algoritmo devolve `falha`.

```
Algoritmo Regr(A, I, Σ, Π)

Entrada: A descrição das ações do domínio A
         A descrição do estado inicial I
         A descrição do estado corrente Σ
         Um plano parcialmente especificado Π
Saída: falha ou um plano solução Π

Início
   Se I ⊨ Σ então devolva Π
   Escolha α ∈ A tal que add(α) ∩ Σ ≠ ∅ ∧ del(α) ∩ Σ = ∅
   Se tal escolha é possível então devolva Regr(A, I, res^-1(α, Σ), α ∘ Π)
   senão devolva falha
Fim
```

Tabela 2: *Busca regressiva no espaço de estados*

Consistência da regressão. Note que, caso o algoritmo *Regr* não seja chamado com Σ igual a uma situação meta específica (*i.e.* completamente especificada), a busca regressiva não será simétrica à busca progressiva. Se tivermos inicialmente $\Sigma = \mathcal{G}$, então cada estado do espaço de busca estará representando um conjunto de situações do mundo e não apenas uma situação específica. Nesse caso, a aplicação inversa de uma ação relevante poderá resultar em estados contendo situações inconsistentes no mundo considerado. Para evitar tais estados, uma verificação adicional deverá ser acrescentada ao algoritmo apresentado. Tal verificação consiste em observar a existência de mutexes nos estados gerados por regressão e podar esses estados.

Essencialmente, um *mutex* é um par de átomos incompatíveis (*i.e.* que não podem co-existir numa mesma situação do mundo) e o conjunto completo desses pares num determinado domínio pode ser construído, automaticamente, a partir da compilação da descrição das ações desse domínio. Conforme *Bonet & Geffner* [1], dado um conjunto de operadores $\mathcal{A}$ e um estado inicial $\mathcal{I}$, um conjunto $\mathbb{M}$ de pares de átomos é um conjunto de mutexes se e só se, para todo par de átomos $\mu = \{p, q\} \in \mathbb{M}$:

- (*i*) $\mu \notin \mathcal{I}$ e
- (*ii*) $\forall \alpha \in \mathcal{A}$ tal que $p \in add(\alpha)$, ou (a) $q \in del(\alpha)$ ou, então, (b) $q \notin add(\alpha)$ e existe $r \in pre(\alpha)$ tal que $\{r, q\} \in \mathbb{M}$.

1.2.3 Implementação dos algoritmos não-determinísticos

Os algoritmos não-determinísticos apresentados podem ser implementados através de busca exaustiva ou heurística. No caso de busca exaustiva, essencialmente podemos usar busca em profundidade ou busca em largura. No caso da *busca em profundidade*, o algoritmo resultante não é completo (caso não haja verificação de estados repetidos), mas é bastante eficiente em termos de espaço consumido (guarda apenas os nós do caminho percorrido). Por outro lado, no caso da *busca em largura*, o algoritmo não apenas é completo, mas também encontra uma solução ótima em número de passos, sendo sua desvantagem o fato de consumir muito espaço para armazenar a fila de nós ainda não visitados. Um alternativa interessante seria a *busca em profundidade iterativa*, que reúne num só algoritmo tanto as vantagens da busca em largura quanto da busca em profundidade.

2 Planejamento como busca heurística

Em vez de busca exaustiva, conforme vimos na seção anterior, podemos resolver problemas de planejamento usando busca heurística. Nesse caso, a eficiência de busca e a qualidade da solução dependerão, essencialmente, da qualidade da função heurística empregada. Nessa seção, apresentamos os principais algoritmos de busca heurística, bem como algumas heurísticas que podem ser usadas com esses algoritmos.

2.1 Algoritmos de busca heurística

Busca heurística é um tipo de busca baseada na noção de "melhor" nó, considerada cada vez que um novo nó deve ser escolhido para ser expandido.

2.1.1 Preferência para o menor custo

O algoritmo para busca heurística mais simples é aquele que sempre escolhe expandir primeiro o nó cujo custo calculado é o menor possível.

Como em problemas de planejamento, em geral, as ações têm todas o mesmo custo (*i.e. custo uniforme*), o custo acumulado de um determinado nó equivale ao número de ações executadas, a partir do estado inicial, para atingir esse nó. O algoritmo apresentado na tabela 3 baseia-se justamente nessa noção de melhor nó.

```
Algoritmo MenorCusto(P)

Entrada: Um problema de planejamento P = ⟨A, I, G⟩
Saída: Um plano solução para P ou falha

Início
   Fila ← {(I, 0)}
   Enquanto Fila ≠ ∅ faça
      Início
         (Estado, Custo) ← RemoveMenorCusto(Fila)
         Se Estado ⊨ G então devolva ExtraiPlano(Estado)
         Para cada E sucessor de Estado faça
            Fila ← Fila ∪ {(E, Custo + 1)}
      Fim
    Devolva falha
Fim
```

Tabela 3: *Busca orientada pelo menor custo.*

2.1.2 Preferência para a menor estimativa

Ao escolher o próximo nó a ser expandido, em vez de considerar o custo de o caminho já percorrido, um algoritmo de busca heurística pode considerar uma estimativa do custo do caminho que ainda falta percorrer até um estado meta, a partir do nó corrente.

A melhor estimativa do custo do caminho que ainda precisa ser percorrido é justamente o número mínimo de ações que precisam ser executadas, a partir do estado corrente, para que um estado meta seja atingido. Em geral, porém, não há estimativas tão precisas pois, do contrário, não haveria necessidade de se realizar busca. Entretanto, para que o algoritmo encontre uma solução, quando houver uma, aé necessário que a estimativa seja otimista (*i.e. admissível*), ou seja, a função heurística não pode super-estimar o número de ações realmente necessárias.

No algoritmo apresentado na tabela 4, h é a função heurística que informa a estimativa do número de ações que ainda precisam ser executadas, a partir de um determinado nó, para que um estado meta seja atingido.

```
Algoritmo MenorEstimativa(P)

Entrada: Um problema de planejamento P = ⟨A, I, G⟩
Saída: Um plano solução para P ou falha

Início
   Fila ← {(I, h(Σ))}
   Enquanto Fila ≠ ∅ faça
      Início
         Estado ← RemoveMenorEstimativa(Fila)
         Se Estado ⊨ G então devolva ExtraiPlano(Estado)
         Para cada E sucessor de Estado faça
            Fila ← Fila ∪ {(E, h(E))}
      Fim
    Devolva falha
Fim
```

Tabela 4: *Busca orientada pela menor estimativa.*

2.1.3 Algoritmo A^*

Um algoritmo interessante, derivado da junção dos dois algoritmos anteriores, é o algoritmo A^*. Esse algoritmo, apresentado na tabela 5, ao escolher o próximo nó a ser expandido, considera não apenas o custo do caminho já percorrido, desde o estado inicial, mas também a estimativa do custo do caminho que ainda falta percorrer, até um estado meta. Quando utilizado com uma heurística admissível, a algoritmo A^* é

correto, completo e sempre encontra soluções ótimas.

```
Algoritmo A*(P)

Entrada: Um problema de planejamento P = ⟨A, I, G⟩
Saída: Um plano solução para P ou falha

Início
   Fila ← {(I, 0, h(Σ))}
   Enquanto Fila ≠ ∅ faça
      Início
         (Estado, Custo) ← RemoveMelhor(Fila)
         Se Estado ⊨ G então devolva ExtraiPlano(Estado)
         Para cada E sucessor de Estado faça
            Fila ← Fila ∪ {(E, Custo + 1, h(E))}
      Fim
    Devolva falha
Fim
```

Tabela 5: *A^*: considera custo mais estimativa.*

2.1.4 Algoritmo *Hill-Climbing*

Quando o espaço de busca a ser explorado é muito extenso, uma boa aproximação é empregar um algoritmo conhecido como *Hill-Climbing*. Esse algoritmo faz uma escolha local por um nó sucessor que seja melhor que o nó corrente e despreza as demais opções, armazenando apenas os nós que compõem o caminho que leva ao nó corrente. Evidentemente, esse algoritmo não é capaz de retroceder nas escolhas feitas e o sucesso da busca dependerá, essencialmente, da qualidade da função heurística empregada. De qualquer forma, ainda que as estimativas sejam bastante precisas, esse algoritmo não é completo e não podemos garantir que uma solução será encontrada, mesmo quando houver uma. Há duas versões do algoritmo *Hill-Climbing*: a primeira delas escolhe qualquer sucessor que tenha avaliação melhor que o nó corrente; a segunda, apresentada na tabela 6, escolhe o melhor entre todos os sucessores que tenham avaliação melhor que o nó corrente.

2.1.5 Algoritmo *Enforced Hill-Climbing*

Quando todos os sucessores do nó corrente na busca têm o mesmo valor heurístico desse nó, o algoritmo *hill-climbing* termina com falha. Para evitar que isso aconteça, a algoritmo *Enforced Hill-Climbing* tenta sair de *platôs* realizando uma busca local em lagura, até que um nó com avaliação melhor que aquela do nó corrente seja encontrado ou até que um limite de profundidade pré-estabelecido seja seja alcançado. Nesse caso, a função $MelhorSucessor(\Sigma)$, utilizada no algoritmo apresentado na tabela 6, deve

ser modificada de modo a considerar não apenas os sucessores imediatos do estado Σ, mas também sucessores em gerações futuras, até o limite imposto (se houver). Note que, no pior caso, o algoritmo *Enforced Hill-Climbing* pode degenerar a uma busca completa em largura.

```
Algoritmo HillClimbing(P)

Entrada: Um problema de planejamento P = ⟨A, I, G⟩
Saída: Um plano solução para P ou falha

Início
   Σ ← I
   Enquanto Σ ⊭ G faça
      Início
         Σ' ← MelhorSucessor(Σ)
         Se h(Σ') ≥ h(Σ) então devolva falha
         Σ' ← Σ
      Fim
    Devolva ExtraiPlano(Σ)
Fim
```

Tabela 6: *Hill-Climbing: seleciona o melhor localmente.*

2.2 Heurísticas para planejamento

Nessa subseção apresentamos algumas funções heurísticas que podem ser utilizadas para resolver problemas de planejamento.

2.2.1 Número de submetas ainda não satisfeitas

Uma heurística bastante simples, apresentada na tabela 7, consiste em considerar como estimativa do custo de atingir um estado meta, a partir de um nó Σ, o número de submetas que ainda precisam ser atingidas. Por exemplo, considere os estados $\Sigma_1 := \{p, g_2, q\}$ e $\Sigma_2 := \{g_3, p, g_2, q, g_1\}$. Então, se as submetas de planejamento forem $\{g_1, g_2, g_3\}$, teremos $h(\Sigma_1) = 2$ e $h(\Sigma_2) = 0$.

Devemos observar que essa heurística não é admissível para alguns domínios. Particularmente, para o domínio do *mundo dos blocos*, se tivermos uma situação $\Sigma := \{on(a, b), clear(a)\}$, e a meta de planejamento for $\mathcal{G} := \{clear(b), ontable(a)\}$, teremos $h(\Sigma) = 2$. Essa estimativa indica que ainda seria necessário executar pelo duas ações para atingir um estado meta a partir do estado Σ. Entretanto, como é evi-

```
Algoritmo h_1(Σ, G)

Entrada: Um estado Σ e a descrição da meta de planejamento G
Saída: O número de submetas ainda não satisfeitas

Início
   Devolva G − Σ
Fim
```

Tabela 7: *Heurística: número de submetas ainda não satisfeitas.*

dente que a ação $unstack(a, b)$ realiza as duas submetas simultaneamente, concluímos que a estimativa é pessimista e, portanto, não-admissível.

2.2.2 Número de átomos distintos

Outra heurística também bastante simples, e obviamente também não-admissível, consiste em considerar o número de átomos distintos entre o estado corrente e o estado que se deseja atingir. Conforme veremos na seção 4, quando usada com estados representados por vetores de bits (*c.f.* seção 4.1), essa heurística parece ser bem mais informativa que a heurística baseada apenas no número de submetas que ainda não foram satisfeitas. No algoritmo apresentado na tabela 8, o símbolo $\oplus$ representa a operação *ou-exclusivo* entre dois vetores de bits.

```
Algoritmo h_2(Σ, G)

Entrada: Um estado Σ e a descrição da meta de planejamento G
Saída: O número de átomos distintos entre dois estados

Início
   Devolva G ⊕ Σ
Fim
```

Tabela 8: *Heurística: número de átomos distintos entre dois estados.*

2.2.3 Grafo de planejamento relaxado

Relaxando as ações de um domínio, tornamos mais fácil encontrar soluções aproximadas para os problemas de planejamento nesse domínio. Conseqüentemente, podemos calcular heurísticas mais precisas que aquelas apresentadas nos tópicos anteriores.

A *relaxação* de uma ação consite em desconsiderar os efeitos negativos dessa ação.

Assim, se α é uma ação da forma $\langle pre(\alpha), add(\alpha), del(\alpha)\rangle$, a *ação relaxada* correspondente, denotada por α^+, será da forma $\langle pre(\alpha), add(\alpha), \emptyset\rangle$. Dado um problema de planejamento $\mathcal{P} = \langle \mathcal{A}, \mathcal{I}, \mathcal{G}\rangle$, o problema de planejamento relaxado correspondente é $\mathcal{P}^+ = \langle \mathcal{A}^+, \mathcal{I}, \mathcal{G}\rangle$, onde $\mathcal{A}^+$ são as ações relaxadas correspondentes àquelas em $\mathcal{A}$. O algoritmo apresentado na tabela 9, recebe um problema relaxado e constrói uma solução relaxada para esse problema, conforme descrito em [6]. A função $level(p)$ informa o primeiro nível do grafo relaxado em que o átomo p ocorre. A heurística calculada é o número de ações nessa solução relaxada.

```
Algoritmo h_3(Σ, G)

Entrada: Um estado Σ e a descrição da meta de planejamento G
Saída: Tamanho da solução para o problema relaxado

Início
    P_0 ← Σ
    i ← 0

    Enquanto P_i ⊭ G faça
       Início
          A_i ← {α ∈ A | pre(α) ⊆ P_i}
          P_{i+1} ← P_i ∪ ⋃_{α∈A_i} add(α)
          Se P_{i+1} = P_i então devolva ∞
          i ← i + 1
       Fim

    para t ← 1 até i faça
       G_t ← {g ∈ G | level(g) = t}

    n ← 0

    para t ← i até 1 faça
       para cada g ∈ G_t faça
          Início
             Selecione α ∈ A tal que level(α) = t − 1 e g ∈ add(α)
             n ← n + 1
             para cada p ∈ pre(α) faça
                G_{level(p)} ← G_{level(p)} ∪ {p}
          Fim

    Devolva n
Fim
```

Tabela 9: *Heurística: tamanho da solução para o problema relaxado.*

2.3 Considerações finais sobre busca heurística

Conforme mostramos, planejamento pode ser modelado como busca heurística. Nesse caso, tanto a eficiência da busca por uma solução, quanto a qualidade da solução encontrada dependerão, essencialmente, da qualidade da heurística empregada. Nessa seção, apresentamos três funções heurísticas e os desempenhos da buscas orientadas por essas ehurísticas serão analisados na seção 4.4.

3 Linguagens para especificação de ações

Conforme definimos na seção 1, dada uma descrição das ações que um agente pode executar, da situação inicial do mundo e dos objetivos desse agente, o planejamento consiste em determinar uma seqüência de ações que, quando executada num mundo satisfazendo a descrição da situação inicial, atinge uma situação meta onde os objetivos especificados são satisfeitos. Nos algoritmos de planejamento, tais descrições devem ser feitas empregando-se um formalismo de representação de conhecimento. Tal formalismo, em geral, é baseado no estilo de representação STRIPS.

3.1 STRIPS, ADL, PDDL e suas extensões

A representação STRIPS[2] foi proposta por *Fikes* e *Nilsson* [3] como uma alternativa ao cálculo de situações e tem sido amplamente usada como forma de representação de conhecimento nos sistemas de planejamento desde então desenvolvidos. Sua principal vantagem com relação ao cálculo de situações é eliminar a necessidade de axiomas de persistência temporal, permitindo um acréscimo de eficiência computacional. Os principais conceitos relacionados à representação STRIPS já foram apresentados na subseção 1.1.

ADL[3] é uma extensão de STRIPS, proposta por *Pednault* [9], que permite representar efeitos condicionais e quantificação universal em universos estáticos. Diferentemente do STRIPS original, que só permitia ações proposicionais, em ADL ações são representadas por esquemas, *i.e.* operadores cujos objetos manipulados são identificados por variáveis.

PDDL[4] é uma tentativa de definir um padrão para representação de conhecimento em sistemas de planejamento [5]. A versão 1.0 dessa linguagem suporta as seguintes características:

- ações básicas do STRIPS;
- efeitos condicionais do ADL;
- quantificação universal em universos dinâmicos (extensão de ADL);
- axiomas de domínios em teorias estratificadas;
- especificação de restrições de segurança;
- especificação de ações hierárquicas.

PDDL 2.1 [4], a versão atual da linguagem, tem como principais características:

[2] *STanford Research Institute Problem Solver.*
[3] *Action Description Language.*
[4] *Planning Domain Definition Language.*

- ações concorrentes;
- tratamento de variáveis métricas;
- representação explícita de tempo e duração das ações;
- otimização de função-objetivo especificada como parte do problema.

Um exemplo de uso do PDDL 2.1 é apresentado na seção 6, mais especificamente na subseção 6.1, onde a sintaxe básica dos operadores é apresentada.

3.2 Ações proposicionais

Para facilitar o uso das ações em tempo de planejamento, em geral, os planejadores expandem a descrição de ações baseadas em esquemas numa descrição equivalente composta apenas por ações proposicionais, completamente instanciadas. Esse trabalho de expansão é feito por um pré-processador, denominado *parser*, que substitui cada variável existente num esquema por cada um dos objetos do seu domínio correspondente. Um bom *parser*, além de expandir esquemas, deve realizar outras tarefas importantes para melhorar a eficiência de planejamento. Entre essas tarefas podemos citar:

- analisar a descrição do problema específico a ser resolvido e descartar ações que não contribuem para sua solução (*e.g.* $move(a, d, c)$ num problema onde o bloco d nunca é citado);
- analisar as precondições dos operadores e descartar aqueles que têm precondições insatisfatíveis (*e.g.* $move(a, a, b)$, que tem precondição $on(a, a)$);
- compilar conjuntos de mutexes para auxiliar planejadores que fazem busca regressiva a podar estados inconsistentes;
- construir estruturas de dados que facilitem determinar operadores aplicáveis ou relevantes, necessários durante a busca de uma solução.

3.3 Considerações finais sobre especificação de ações

Nessa seção, apresentamos os conceitos gerais relacionados à representação de ações em domínios de planejamento. Conforme vimos, a complexidade dos problemas de planejamento que somos capazes de especificar e resolver com um sistema de planejamento depende muito da expressividade da linguagem de especificação de ações utilizada por esse sistema de planejamento. Das linguagens para representação de ações existentes, PDDL 2.0 é aquela mais expressiva e com ela podemos especificar não apenas problemas de planejamento clássico, mas também problemas de planejamento envolvendo ações durativas e consumo de recursos métricos.

4 Experimentos com busca heurística

Nessa seção, descrevemos os algoritmos implementados para busca e para cálculo das heurísticas. Além disso, descrevemos os testes realizados com esses algoritmos e analisamos os resultados obtidos.

4.1 O algoritmo de busca

O algoritmo utilizado para a busca principal é o A^*, apresentado na subseção 2.1.3. Nessa subseção, discutimos os detalhes de sua implementação.

4.1.1 Representação de estados e ações

Para implementar um sistema de planejamento baseado em busca, precisamos de uma estrutura de dados para representar estados e ações. No caso de planejamento como busca progressiva, essa estrutura de dados deve permitir que operações básicas, tais como verificação de *aplicabilidade* de ações e *progressão* de estados, sejam efetuadas eficientemente. Ademais, ainda é desejável que tal estrutura seja compacta, uma vez que memória é um recurso escasso em sistemas de planejamento.

Estados como vetores de bits. Seja $\mathbb{A}$ o conjunto dos átomos empregados na descrição de um determinado problema de planejamento. Seja $b : \mathbb{A} \mapsto n$ uma função bijetora que mapeia cada átomo $a \in \mathbb{A}$ a um inteiro no intervalo $[0, n-1]$, onde n é a cardinalidade do conjunto $\mathbb{A}$. Então, um estado Σ pode ser representado po um vetor de bits $\langle b_{n-1} b_{n-2} \ldots b_0 \rangle$ onde $b_i = 1$ se e só se, para $0 \leq i < n$, $\exists a \in \Sigma$ e $b(a) = i$.

A função de mapeamento $b : \mathbb{A} \mapsto n$ pode ser implementada, trivialmente, por meio de uma *tabela de hashing* que armazene os átomos e seus respectivos números, atribuídos seqüencialmente a cada inserção na tabela.

Considere, por exemplo, a *Anomalia de Sussman* e $t(x)$ denotando que o bloco x está sobre a mesa, $c(x)$ denotando que o bloco x está livre e $o(x, y)$ denotando que o bloco x está sobre o bloco y. O conjunto dos átomos na descrição desse problema é $\mathbb{A} = \{t(a), t(b), t(c), c(a), c(b), c(c), o(a, b), o(a, c), o(b, a), o(b, c), o(c, a), o(c, b)\}$. Sem perda de generalidade, suponha que $b(t(a)) = 11$, $b(t(b)) = 10$, $\ldots$, $b(o(c, a)) = 1$ e $b(o(c, b)) = 0$. Então, a situação inicial $\mathcal{I} = \{t(a), t(b), c(b), c(c), o(c, a)\}$ pode ser representada pelo vetor $\langle 110011000010 \rangle$. Em geral, num domínio com m objetos e aridade máxima k (onde f_i é o número de fluentes com aridade i), o número de bits necessários para representar um estado é $O(\sum_{0 \leq i \leq k} f_i . m^i)$.

Precondições e efeitos como vetores de bits. Seja α uma ação definida por $\langle pre(\alpha), add(\alpha), del(\alpha) \rangle$. O conjunto $pre(\alpha)$ pode ser representado por um vetor de bits $\langle p_{n-1} p_{n-2} \ldots p_0 \rangle$ tal que $p_i = 1$ se e só se $\exists p \in pre(\alpha)$ e $b(p) = i$. Analogamente,

os conjuntos $add(\alpha)$ e $del(\alpha)$ podem ser representados, respectivamente, pelos vetores $\langle a_{n-1}a_{n-2}\ldots a_0\rangle$ e $\langle d_{n-1}d_{n-2}\ldots d_0\rangle$.

Operações com vetores de bits. Uma grande vantagem da representação de conjuntos por meio de vetores de bits é que as operações com esses conjuntos podem ser diretamente implementadas por instruções de máquina que realizam operações lógicas bit-a-bit. Por exemplo, usando os operadores bit-a-bit `~` (*i.e. negação*), `&` (*i.e. conjunção*) e `|` (*i.e. disjunção*), podemos implementar os algoritmos para determinar aplicabilidade e estado resutante apresentados, respectivamente, nas tabelas 10 e 11.

```
Algoritmo Aplic(α, Σ)

Entrada: Uma ação α e um estado Σ representados por vetores de bits
Saída: verdade ou falso

Início
    Devolva (pre(α) & Σ) == pre(α)
Fim
```

Tabela 10: *Verifica a aplicabilidade de uma ação a um estado.*

```
Algoritmo Res(α, Σ)

Entrada: Uma ação α e um estado Σ representados por vetores de bits
Saída: O estado resultante da aplicação de α a Σ

Início
    Devolva (Σ | add(α)) & ~del(α)
Fim
```

Tabela 11: *Determina o estado resultante da aplicação de uma ação a um estado.*

4.1.2 Estados visitados e extração do plano

Para guadar estados já visitados, e evitar caminhos já percorridos, cada novo nó gerado na árvore de busca é armazenado numa tabela de *hashing*, juntamente com um ponteiro para o seu nó pai. Ademais, sempre que um nó é inserido nessa tabela, um ponteiro para esse nó é devolvido. Iniciamos a busca inserindo o nó $\langle \mathcal{I}, \texttt{null}\rangle$ e recebendo seu ponteiro `p`. A partir daí, geramos os sucessores Σ_i de $\mathcal{I}$ e inserimos na tabela os nós $\langle \Sigma_i, \texttt{p}\rangle$ correspondentes (analogamente para cada novo nó expandido).

Com essa tabela, evitamos que dois estados idênticos sejam explorados duas ou mais vezes. Além disso, como cada nó guarda um ponteiro para seu nó pai, quando

um nó representando um estado meta é atingido, o caminho do estado inicial até esse nó pode ser facilmente reconstituído. Dessa forma, evitamos que um plano tenha que ser explicitamente construído durante a busca e economizamos não apenas tempo, mas também espaço.

4.1.3 Armazenamento dos nós ainda não expandidos

Como dissemos, cada vez que um novo estado é gerado, ele é armazenado numa tabela de hashing e um ponteiro para esse estado é devolvido. Isso significa que a estrutura que representa um estado é única e toda e qualquer referência a ela pode e deve ser feita por meio de seu ponteiro. Sendo assim, quando geramos os estados sucessores de um determinado estado, apenas seus ponteiros precisam ser armazenados na fila de nós a serem explorados. Isso faz com que o espaço necessário na fila seja reduzido ao mínimo; o que é particularmente importante quando o problema de planejamento é muito grande. Além disso, para melhorar a eficiência de tempo de inserção e remoção nessa fila, utilizamos um estrutura de *heap*, que reduz o tempo dessas operações de $O(n)$ para $O(lg\ n)$.

4.2 Algoritmos para cálculo de heurística e variações

Os algoritmos para cálculo das heurísticas são aqueles já apresentados na subseção 2.2. Nessa seção, vamos apenas discutir variações da heurística h_3, apresentada na tabela 9, que é baseada no grafo de planejamento relaxado.

4.2.1 Reutilização de ações já escolhidas num mesmo nível

Após construído o grafo de planejamento relaxado, uma busca regressiva é empregada para se extrair uma solução relaxada para o problema considerado. Para garantir que ações já escolhidas num determinado nível de ações sejam reutilizadas para suportar várias submetas num mesmo nível de átomos do grafo, modificamos o algoritmo da tabela 9 da seguinte maneira:

$n \leftarrow 0$
Para $t \leftarrow i$ **até** 1 **faça**
  Início
    $\mathcal{S} \leftarrow \emptyset$ /* *ações selecionadas no nível* t */
    Para cada $g \in G_t$ **faça**
      Início
        Se *não existe* $\alpha \in \mathcal{S}$ *tal que* $g \in add(\alpha)$ **então**
          Início
            Selecione $\alpha \in \mathcal{A}$ *tal que* $level(\alpha) = t - 1$ *e* $g \in add(\alpha)$
            Para cada $p \in pre(\alpha)$ **faça** $G_{level(p)} \leftarrow G_{level(p)} \cup \{p\}$
            $\mathcal{S} \leftarrow \mathcal{S} \cup \{\alpha\}$

```
                $n \leftarrow n + 1$
            Fim
        Fim
    Fim
```

Observe que quando uma ação é reutilizada num nível t, as precondições dessa ação não precisam ser resdistribuídas em níveis anteriores, nem o número de ações n deve ser incrementado (isso já foi feito no momento em que a ação foi selecionada pela primeira vez).

4.2.2 Escolha de ações com o menor número de precondições

Outra variação possível da heurística baseada no problema relaxado é a seguinte: durante a busca regressiva por uma solução relaxada, ao selecionar uma ação para suportar uma sub-meta, escolha sempre a ação que tiver o menor número de precondições. Para implementar essa variaçào, modificamos o algoritmo da tabela 9 da seguinte maneira:

```
n ← 0
Para t ← i até 1 faça
    Início
       Para cada g ∈ G_t faça
          Início
             𝔸 ← {α | α ∈ 𝒜 ∧ level(α) = t − 1 ∧ g ∈ add(α)}
             Selecione α ∈ 𝔸 tal que |pre(α)| é mínimo
             Para cada p ∈ pre(α) faça G_level(p) ← G_level(p) ∪ {p}
             n ← n + 1
          Fim
    Fim
```

4.2.3 Uma terceira variação

As duas variações da heurística h_3 (apresentadas nos tópicos anteriores) foram denominadas, respectivamente, h_3+*reuso* e h_3+*minprec*. Juntando essas duas variações, obtivemos uma terceira variação, que denominamos h_3+*reuso*+*minprec*. Os resultamos obtidos com essas três variações da heurística h_3 podem ser vistos nos gráficos apresentados na subseção 4.4.

4.3 Domínios de teste e metodologia

O domínio escolhido para testes foi o *mundo dos blocos*[5]. Para esse domínio, resolvemos parte dos problemas propostos no *International Planning Competition - 2002*,

[5]Outros domínios foram testados (*e.g.* satélite e hanói), mas como o número de problemas resolvidos nesses domínios foi muito pequeno, nenhuma análise mais consistente poderia ser feita.

com o número de blocos variando de 4 a 12.

Para que os resultados fossem mais expressivos, seria necessário que os experimentos fossem realizados com diversas instâncias de um mesmo tamanho. Assim, poderíamos traçar as curvas representando consumo de tempo e espaço para cada tamanho de instância, com base em valores médios e não com base em um único valor obtido para um instância específica. No entanto, ao contrário com o que ocorre com os domínio artificiais [7], não é fácil gerar aleatoriamente grande quantidade de problemas solucionáveis para o mundo dos blocos. Por esse motivo, os resultados apresentados na próxima seção são baseados apenas nos valores obtidos para as instâncias específicas propostas no IPC-2.

Os experimentos foram realizados em duas etapas:

- primeiro comparamos o desempenho das heurísticas h_1 (*i.e* número de submetas ainda não satisfeitas), h_2 (*i.e.* número de átomos distintos entre dois estados) e h_3 (*i.e.* tamanho da solução relaxada);
- depois, comparamos apenas o desempenho relativo das variações da heurística h_3, a saber: h_3+*reuso*, h_3+*minprec* e h_3+*reuso*+*minprec*.

A fim de tornar mais fácil discernir as curvas obtidas para cada heurística, adotamos uma escala logarítmica nos gráficos a seguir, que apresentam os resultados dos testes realizados.

4.4 Análise dos resultados

O gráfico apresentado na figura 1 mostra o tamanho do espaço de busca explorado com cada uma das três heuríticas básicas h_1, h_2 e h_3. Como podemos observar nesse gráfico, embora as heurísticas h_1 e h_2 tenham um comportamento bastante semelhante, a heurística h_2 explora um espaço de busca bem menor que a heurística h_1 e, portanto, consegue resolver problemas maiores que aqueles resolvidos por h_1. Particularmente, nos testes que realizamos, h_1 resolve problemas com no máximo 8 blocos, enquanto h_2 chega a resolver problemas com até 12 blocos, o mesmo tamanho de problema que conseguimos resolver com a heurística h_3, baseada no problema relaxado.

Ainda com base na figura 1, podemos observar que a heurística h_3 é aquela que explora o menor espaço de busca e que, portanto, deve ser aquela que resolve os problemas mais rapidamente. Entretanto, analisando o gráfico apresentado na figura 2, observamos que o tempo gasto pela heurística h_2, pelo menos para os problemas que conseguimos resolver com ambas as heurísticas, é bastante próximo daquele gasto pela heurística h_3, a despeito dessa última heurística explorar um espaço de busca bem menor. Uma justificativa para esse fato é que o cálculo da heurística h_3 é muito mais custoso que o cálculo de heurística h_2. Sendo assim, para problemas pequenos, o tempo que se ganha com a redução do espaço de busca proporcionada pelo uso da

heurística h_3, perde-se com o cálculo dessa heurística. Evidentemente, para problemas muito grandes, a redução do espaço de busca certamente irá compensar o tempo gasto para calcular uma heurística mais precisa e, portanto, para esses problemas, a heurística h_3 deverá tem um desempenho muito melhor que as outras duas.

De modo geral, podemos dizer que a heurística h_3 é bem mais informativa que as outras duas e resolve os problemas de planejamento de modo mais eficiente tanto em termos do espaço de busca explorado quanto em termos do tempo de CPU consumido.

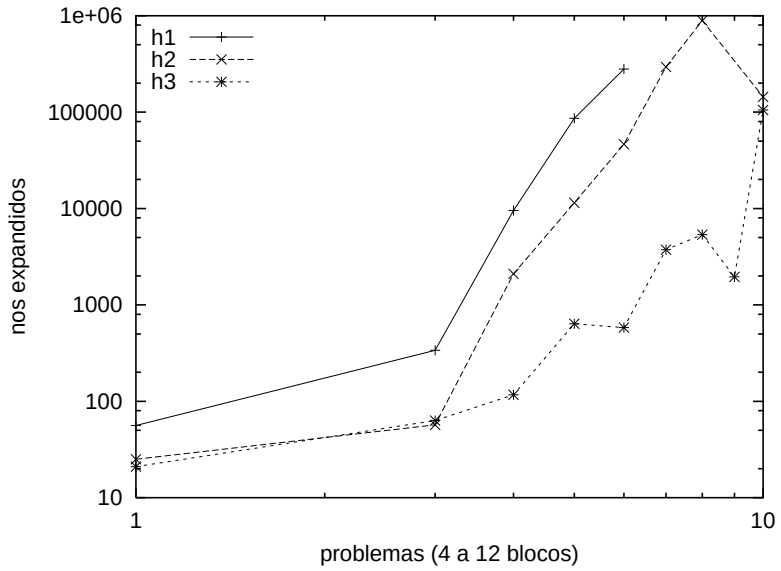

Figura 1: Tamanho do espaço de busca explorado com as heurísticas h_1, h_2 e h_3

Os gráficos apresentados nas figuras 3 e 4 mostram, respectivamente, o espaço de busca explorado e o tempo de CPU consumido por cada uma das variações da heurística h_3.

Como podemos observar na figura 3, a versão básica da heurística h_3 e a versão h_3-$minprec$, que escolhe ações com o número mínimo de precondições, exploram exatamente o mesmo espaço de busca. Acreditamos que esse fato seja devido a características especificadas do domínio considerado ($i.e$ mundo dos blocos) e que, em outros domínios, poderíamos observar um comportamento diferenciado para essas duas heurísticas.

A figura 3 ainda mostra também que as variações de h_3 que tentam reutilizar ações já selecionadas anteriormente (h_3+$resuso$ e h_3+$resuso$+$minprec$) exploram um espaço de busca maior que a versão básica da heurística. Uma justificativa para esse fato é que as variações de h_3 propostas, em geral, produzem estimativas inferiores àquelas obtidas com a versão básica de h_3. (Note que o reuso de ações e a seleção de ações com o menor número de precondições fazem com que a estimativa do número

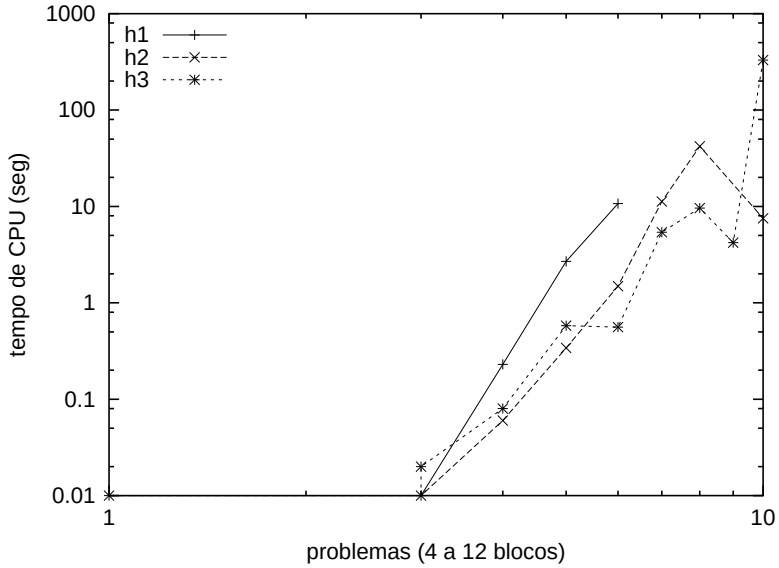

Figura 2: Tempo de CPU consumido com as heurísticas h_1, h_2 e h_3

de ações na solução relaxada seja reduzida). Como sabemos, quanto maior for a estimativa fornecida pela função heurística, menor será o espaço de busca explorado pelo algoritmo de busca que a utiliza (evidentemente, se a heurística super-estima a estimativa, eventualmente, o espaço de busca será tão reduzido que uma solução poderá não ser encontrada, ainda que exista). Sendo assim, tornar uma heurística mais precisa (*i.e.* admissível) garante apenas que o algoritmo de busca seja completo (*i.e.* encontre uma solução sempre que tal solução existir), mas não que a busca se torne, necessariamente, mais eficiente (como esperávamos).

Observando as curvas na figura 4, que representam os tempos gastos com cada uma das variações da heurística, podemos notar que o consumo de tempo é diretamente proporcional ao espaço de busca explorado (observe que as curvas de tempo apresentadas na figura 4 são praticamente idênticas às curvas de espaço de busca apresentadas na figura 3). As diferenças que se observam entre as curvas de tempo são devidas aos custos de se calcular as diversas heurísticas. Evidentemente, h_3 é a heurística mais barata para se calcular (curva inferior), enquanto a heurística h_3+$resuso$+$minprec$ é a mais cara (curva superior). Esse custo adicional que se observa é devido a tempo que se gasta para verificar se uma ação já selecionada pode ser reutilizada e também para selecionar a ação com o menor número de precondições capaz de suportar uma determinada sub-meta, durante a extração de uma plano relaxado.

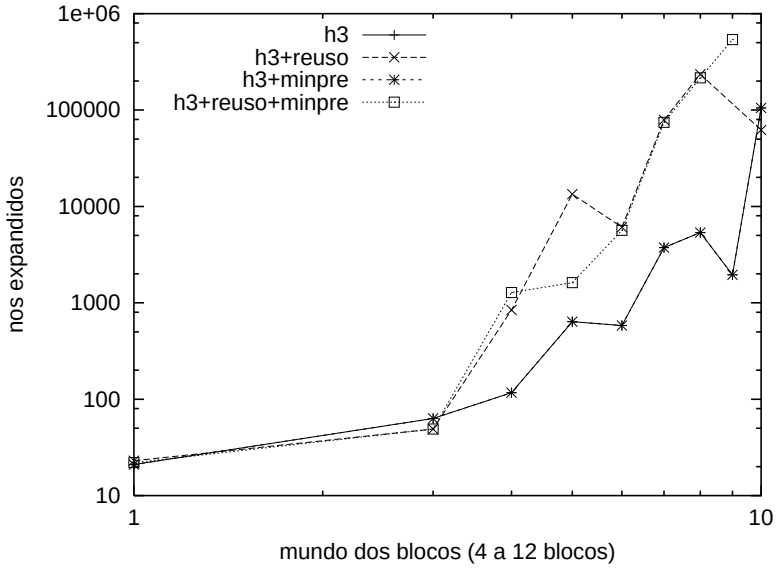

Figura 3: Tamanho do espaço de busca explorado com as variações da heurística h_3

4.5 Considerações finais sobre os experimentos com heurísticas

Conforme mostraram os experimentos realizados, as variações propostas para o cálculo da heurística baseada em grafo de planejamento relaxado, apesar de admissíveis, não são mais informativas que a versão básica descrita em [6], pelo menos para problemas no *mundo dos blocos*, que foi o domínio considerado. Acreditamos que para tornar a heurística h_3 mais eficiente teríamos que encontrar uma forma de aumentar a estimativa que ela fornece (para reduzir o espaço de busca explorado), sem no entanto torná-la não-admissível (para garantir que pelo menos uma solução seja encontrada, ainda que não seja a melhor).

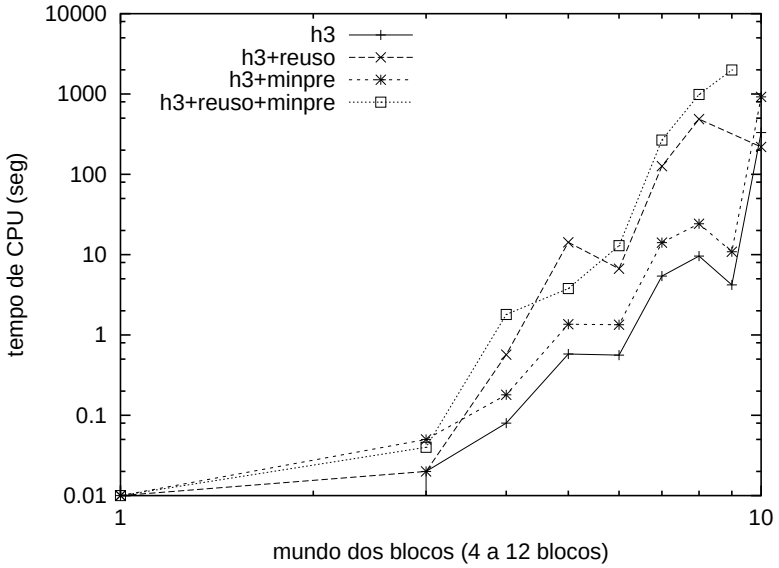

Figura 4: Tempo de CPU consumido com as variações da heurística h_3

5 Planejamento temporal métrico

Diferentemente de planejamento clássico, planejamento temporal métrico envolve raciocínio sobre ações durativas, que executam concorrentemente e consomem recursos. As principais dificuldades que surgem desses novos requisitos são:

- maior espaço de busca a ser explorado;
- tarefa engloba características de planejamento e escalonamento;
- solução não depende só de satisfação de metas, mas também de otimização;
- heusrística também deve levar em conta restrições de tempo e recursos.

A proposta do artigo de *Do & Kambhampati* [2] é implementar um planejador baseado em busca heurística progressiva no espaço de estados do problema. Esse tipo de busca é necessário porque, para tratarmos recursos métricos, precisamos ter informação completa sobre os estados do mundo.

5.1 Modelo de estados

Nessa subseção descrevemos as representações para estados e ações, bem como o algoritmo de busca progressiva no espaço de estados do problema, conforme especificado pelo sistema de planejamento temporal métrico Sapa [2].

5.1.1 Representação de estados

No planejamento temporal métrico, os estados do mundo são representados por tuplas da forma $S = \langle A, M, P, E, t \rangle$, onde:

- A é um conjunto de pares $\langle a_i, t_i \rangle$, $t_i < t$, indicando que o átomo a_i foi atingido mais recentemente no instante t_i e persiste até o instante t;
- M é um conjunto de valores de todas as variáveis métricas do problema;
- P é um conjunto de condições persistentes num certo intervalo de tempo;
- E é um conjunto de pares $\langle a_i, t_i \rangle$, $t_i > t$, indicando que o átomo a_i será atingido no instante futuro t_i. A ocorrência de um tal evento pode alterar o valor de um predicado, atualizar o valor de uma variável métrica ou finalizar a persistência de alguma condição;
- t é um rótulo temporal.

Estado inicial e meta. O estado inicial, rotulado com o instante 0, tem uma agenda vazia de eventos E e um conjunto vazio de condições persistentes P. Porém, é completamente especificado em termos de valores de átomos A e variáveis métricas M. Uma meta é representada por um conjunto $G = \{\langle a_1, t_1 \rangle \ \ldots, \langle a_n, t_n \rangle\}$, onde cada elemento especifica uma submeta a_i e o instante t_i em que ela deverá ser atingida.

Satisfação de meta. Um estado $S = \langle A, M, P, E, t \rangle$ satisfaz uma meta G, denotado por $S \models G$, se para cada $\langle a_i, t_i \rangle \in G$:

- $\exists \langle a_i, t_j \rangle \in A$ tal que $t_j < t_i$ e não existe evento em E que remova a_i, ou
- $\exists e \in E$ que adiciona a_i num instante $t_e < t_i$.

5.1.2 Representação de ações

A representação de ações utilizada permite descrever:

- ações durativas;
- precondições instantâneas ou persistentes;
- efeitos instantâneos em qualquer instante durante a execução da ação.

Toda ação α tem duração $dur(\alpha)$, início $start(\alpha)$ e final $end(\alpha) \equiv start(\alpha) + dur(\alpha)$. Ademais, os efeitos de uma ação α são particionados em três conjuntos disjuntos:

- efeitos que ocorrem no instante $start(\alpha)$;
- efeitos que ocorrem no instante $end(\alpha)$;
- efeitos que ocorrem num instante $start(\alpha) + d$ $(0 < d < dur(\alpha))$.

Aplicabilidade. Uma ação α é aplicável a um estado $S = \langle A, M, P, E, t \rangle$ se:

- toda precondição instantânea de α é satisfeita por A e M;
- efeitos de α não interferem com condições em P, nem com eventos em E;
- nenhum evento em E interfere com precondições persistentes de α.

Avanço do tempo. Uma ação especial $AdvanceTime$ avança o tempo de um estado S para o instante t_e do evento mais próximo em E. Essa ação é aplicável em qualquer estado S em que $E \neq \emptyset$ e sua aplicação produz um estado S' atualizado com todos os efeitos dos eventos em E, que ocorrem concorrentemente no instante t_e.

5.1.3 Algoritmo de busca

A busca é realizada, essencialmente, da seguinte forma:

$StateQueue \leftarrow \{S_{init}\}$
`while` $StateQueue \neq \emptyset$ `do`
    $State \leftarrow Dequeue(StateQueue)$
    `choose` $Applicable(Action, State)$ `in`
        $State' \leftarrow Apply(Action, State)$
        `if` $State' \models G$ `then` $PrintSolution(State')$
        `else` $Enqueue(State', StateQueue)$

O planejador SAPA [2] implementa essa busca usando o algoritmo A^* guiado por funções heurísticas apropriadas, descritas na próxima seção.

5.2 Heurísticas

Para ser suficientemente informativa, uma função heurística para planejamento temporal métrico deve estimar a distância até um estado meta não apenas com base no número de ações necessárias para atingir tal estado, mas também levando em conta a duração total do plano (*i.e. makespan*), folgas nos prazos para realização de sub-metas (*i.e. slack*) e consumo de recursos métricos. Funções heurísticas que consideram esses aspectos adicionais servem para guiar não apenas a tarefa e planejamento, mas também a tarefa de escalonamento.

O sistema SAPA utiliza uma heurística estimada em duas fases:

- Primeiro, usando a idéia de problema relaxado; ignorando não apenas os efeitos negativos das ações, mas também o consumo de recursos métricos;
- Depois, ajustando a estimativa obtida na fase anterior; usando informações a respeito de tempo de execução das ações e consumo de recursos métricos.

5.2.1 Grafo de planejamento temporal métrico relaxado

A construção do grafo para o problema relaxado é baseada na *marcação* de átomos e ações. Cada átomo a_i é marcado, e aparece no nível de átomos i, se pode ser atingido no intante t_i. Analogamente, cada ação α é marcada, e aparece no nível de ações i, se pode ser executada no intante t_i. No início, apenas átomos em $A \in S_{init}$ são marcados no nível 0; o nível de ações é vazio e a fila de eventos E contém eventos ainda não executados que adicionam átomos novos. A partir daí, uma ação é marcada se todas as suas precondições já foram marcadas. Quando uma ação α é marcada, todos seus efeitos positivos instantâneos são também marcados. Além disso, todo efeito retardado e de α produzindo a é colocado em E se (1) a não foi marcado e (2) não há evento que produz a em E agendado para ocorrer antes de r. Quando um evento e é adicionado a E, removemos de E todo evento e' que ocorre após e, adicionando o mesmo átomo a.

Quando não há mais ações aplicáveis em um estado S, sinalizamos `falha` se (1) a fila de eventos E está vazia ou (2) há uma submeta não marcada com prazo menor que o primeiro evento agendado em E. Se nenhuma dessas condições ocorre, aplicamos $AdvanceTime$ e ativamos todos os eventos que ocorrem no mesmo instante do próximo evento em E. O processo é repetido até que todas as submetas tenham sido marcadas ou até que uma das condições de falha seja detectada.

No sistema SAPA, o grafo de planejamento temporal relaxado é usado para:

- podar estados que não levam a um estado meta;
- determinar um limite inferior de tempo para atingir submetas;
- estimar o número mínimo de passos até um estado meta.

5.2.2 Heurísticas baseadas em tempo

Observando que, no grafo relaxado, todas as submetas são atingidas no menor tempo possível, podemos derivar uma série de heurísticas admissíveis que podem ser usadas para otimizar o plano com relação ao tempo necessário para sua execução.

A *distância estimada* de um estado S a um estado meta S' é:

- **Max-span**: *a diferença absoluta entre os rótulos temporais de S e S'.*

- **Min-slack**: *o mínimo das folgas*[6] *estimadas para cada submeta individual.*
- **Max-slack**: *o máximo das folgas estimadas para cada submeta individual.*
- **Sum-slack**: *a soma das folgas estimadas para cada submeta individual.*

5.2.3 Heurísticas baseadas em ações

As heurísticas apresentadas na subseção visam otimizar a solução de acordo com funções objetivos baseadas em tempo. Para otimizar o tamanho dos planos, podemos usar a heurísticas definida a seguir.

A *distância estimada* de um estado S a um estado meta S' é:

- **Sum-action**: *o número de ações no plano relaxado que leva de S a S'.*
- **Sum-duration**: *a soma das durações das ações no plano relaxado que leva de S a S'*

5.2.4 Uso de restrições métricas para ajuste das heurísticas

As heurísticas anteriores não levam em conta o consumo de recursos métricos. Como, no grafo relaxado, as ações que produzem e consomem recursos métricos podem ser executadas concorrentemente, é muito difícil raciocinar sobre quantidades de recursos em cada instante de tempo. Para contornar essa dificuldade, procedemos da seguinte maneira: para cada recurso ρ, encontramos a ação α_ρ que produz a maior quantidade desse recurso. Sejam Δ_ρ a quantidade produzida por α_ρ e $init(\rho)$ a quantidade desse recurso disponível no estado inicial do grafo relaxado. Sejam também $pro(\rho)$ e $con(\rho)$, respectivamente, o total produzido e consumido do recurso ρ, pelas ações existentes no plano relaxado. Então, se $con(\rho) > init(\rho) + pro(\rho)$, podemos ajustar as heurísticas da seguinte forma:

- **Adjust-sum-action**: $h \leftarrow h + \sum_\rho \lceil \frac{con(\rho) - [init(\rho) + pro(\rho)]}{\Delta_\rho} \rceil$
- **Adjust-sum-duration**: $h \leftarrow h + \sum_\rho \frac{con(\rho) - [init(\rho) + pro(\rho)]}{\Delta_\rho} . dur(\alpha_\rho)$

5.3 Considerações finais sobre planejamento temporal métrico

Testes realizados pelos autores mostraram que as heurísticas baseadas apenas em tempo, apesar de admissíveis, são pouco informativas e tornam a busca muito ineficiente para problemas grandes. Por outro lado, as heurísticas em ações, apesar de não serem admissíveis, são capazes de guiar eficientemente a busca de soluções, até

[6] A folga (*i.e. slack*) para uma submeta g é estimada como a diferença absoluta entre o instante em que g é atingida no grafo relaxado e o prazo estabelecido para seu atingimento.

mesmo para problemas grandes. Ademais, os testes também mostraram a heurística *sum-action* é sempre menos eficiente que sua versão ajustada *adjust-sum-action* ; ao contrário do que ocorre com a heurística *adjust-sum-duration* que, em alguns casos, chega a examinar o triplo do número de nós que são examinados com sua versão sem ajuste (*sum-duration*). Os autores ainda não sabem explicar porque a heurística *adjust-sum-duration*, ao contrário do que se esperava, é tão menos informativa que sua versão básica.

6 ROADEF'2005 sob a perspectiva de planejamento

O desafio proposto pelo ROADEF'2005 [8] está relacionado à programação da produção numa fábrica de veículos. O problema consiste em, dados os veículos que deverão ser montados num dia, determinar a melhor ordem de montagem tal que certas restrições sejam satisfeitas. Essas restrições podem ser de dois tipos:

- *pintura*: para minimizar a quantidade de solvente gasto na limpeza da máquina (necessária a cada mudança de cor ou a cada n veículos pintados, mesmo que não haja mudança de cor), os veículos devem aparecer na seqüência de montagem agrupados por cores;
- *montagem*: para não sobrecarregar a linha de produção, veículos que demandam operações especiais (*e.g.*, instalação de teto solar ou ar condicionado) devem estar uniformemente distribuídos na seqüência de montagem. Cada operação especial tem associada uma razão da forma N/P, indicando que numa seqüência de P veículos, podemos ter no máximo N veículos requerendo tal operação. Essas restrições podem ser de *alta* ou *baixa* prioridade.

Enquanto a pintura é uma restrição *rígida*, montagem é uma restrição *frouxa* (*i.e.*, não é possível garantir que todas as restrições de montagem possam ser satisfeitas). Sendo assim, durante a busca da melhor ordem de montagem dos veículos, nosso objetivo será atender às restrições de pintura, minimizando o número de violações das restrições de montagem. Ademais, não deverá haver nenhuma compensação entre objetivos, ou seja, uma restrição menos importante não deverá ser satisfeita às custas da violação de uma outra restrição mais importante. Além disso, ao iniciar a busca pela melhor seqüência de montagem para um determinado dia, deveremos levar em conta os últimos veículos na seqüência de montagem do dia anterior e, portanto, a contabilização do número total de violações deverá considerar também esses veículos (embora a ordem deles não possa mais ser modificada).

6.1 Descrição do problema de montagem de veículos

Um problema nesse domínio é descrito através de quatro arquivos: o primeiro deles determina o limite de pinturas, antes que a máquina tenha que ser lavada; o segundo indica a prioridade relativa das restrições consideradas; o terceiro estabelece as razões associadas às operações de montagem; e o último descreve os veículos a serem montados. Por exemplo:

1. `paint_batch_limit_.txt`

```
limitation;
10;
```

2. `optimization_objectives.txt`

```
rank;objective name;
1;paint_color_batches;
2;high_priority_level_and_easy_to_satisfy_ratio_constraints;
3;low_priority_level_ratio_constraints;
```

3. `ratios.txt`

```
Ratio;Prio;Ident;
2/4;1;HPRC2;
1/4;1;HPRC3;
...
```

4. `vehicles.txt`

```
Date;SeqRank;Ident;Color;HPRC1;HPRC2;HPRC3;HPRC4;HPRC5;LPRC1;LPRC2;LPRC3;...
2003 26 1;1050;039032521147;2;0;0;0;0;0;0;0;1;0;0;0
2003 27 1;   1;039032440804;3;0;0;0;0;0;0;0;0;0;0;0
2003 27 1;   2;039032620549;5;0;0;0;0;0;0;0;0;0;0;0
```

6.2 Modelagem do problema em termos de planejamento

Evidentemente, em termos de planejamento, uma solução para o problema proposto deverá ser um plano cuja ordem das ações corresponda à ordem de montagem dos veículos. Uma forma natural de conseguir esse mapeamento é associar a cada veículo um operador correspondente. Assim, por exemplo, o plano $\langle do913, do127, \ldots \rangle$ indicaria que o veículo cujo identificador é 913 seria o primeiro a ser montado, aquele cujo identificador é 127 seria o segundo, *etc*.

Para entender como codificar esses operadores considere, por exemplo, a montagem de um veículo com as seguintes características: `Ident = 913, AirCond = false, SunRoof = true`. Além disso, suponha que a razão associada à operação de instalação de teto solar seja 2/5 (como esse veículo particular não vai ter ar condicionado, a razão associada a essa última operação é irrelevante) e que a "unidade de tempo" corresponda ao período de montagem de um veículo. Então, usando PDDL 2.0 [4], podemos codificar o seguinte operador para o referido veículo:

```
(:durative-action do913
  :parameters ()
  :duration   (= ?duration 5)
  :condition  (and (at start (vehicle 913))
                   (at start (>= (sunroof) 1)))
  :effect     (and (at start (not (vehicle 913)))
                   (at start (decrease (sunroof) 1))
                   (at end (increase (sunroof) 1))))
```

Note que os operadores devem ser instâncias específicas para cada um dos veículo e não esquemas a serem instanciados pelo planejador. Nesse operador específico, o

átomo `(vehicle 913)`, requerido como precondição, é também um efeito negativo do operador. Portanto, fornecendo esse átomo no estado inicial, e garantindo que nenhum operador o produza como efeito, garantimos também que a ação `do913` ocorrerá uma única vez no plano. Além disso, iniciando a variável numérica `sunroof` com o valor 2, e dando à ação a duração 5, conseguimos modelar a razão 2/5 (supondo que as ações possam ser executadas concorrentemente). Observe que uma das condições do operador `do913` é que a variável `sunroof` seja pelo menos 1. Se tal condição não é satisfeita, significa que já existem dois veículos com teto solar sendo montados entre os últimos 5 veículos da seqüência. Ademais, caso o operador `do913` seja aplicável, o valor da variável `sunroof` é imediatamente decrementado, sendo incrementado novamente apenas no final da duração da ação, após 5 veículos terem sido montados.

Assim, em termos de planejamento, um problema de seqüenciamento de veículos nesse domínio poderia ser descrito da seguinte maneira:

```
(define (problem day-1)
  (:domain assembly)
  (:init (vehicle 127)
         (vehicle 913)
          ...
         (= (sunroof) 2)
         (= (aircond) 1)
          ...)
  (:goal (and (not (vehicle 127))
              (not (vehicle 913))
               ...)))
```

6.2.1 Tratamento de penalidades

Usando essa modelagem sugerida para os operadores, apenas seqüências de montagem sem nenhuma penalidade seriam consideradas soluções para o problema proposto. Nesse caso, se uma tal seqüência não for possível, o planejador não será capaz de encontrar uma solução. Para contornar esse problema, para cada operador do domínio, podemos gerar um outro operador com precondições relaxadas, que contabiliza a penalidade dessa relaxação. Por exemplo, para o operador `do913`, teríamos:

```
(:durative-action do913R
  :parameters ()
  :duration   (= ?duration 5)
  :condition  (at start (vehicle 913))
  :effect     (and (at start (not (vehicle 913)))
                   (at start (increase (penalty sunroof) 1000))))
```

Ademais, para levar em conta os dados relativos aos últimos veículos montados no dia anterior, bastaria iniciar as variáveis `(penalty ...)` com valores apropriados.

6.3 Considerações finais sobre o problema do ROADEF'2005

Conforme mostramos, a modelagem do problema de seqüenciamento de veículos como uma tarefa planejamento requer que o planejador seja capaz de considerar a execução concorrente dos operadores, permitindo uma nova ação seja iniciada antes do término de uma ação anterior (*vide* seção 5 – *Planejamento temporal métrico*).

Caso a execução concorrente são seja possível, então a modelagem do problema será muito difícil pois, conforme a seção 3 de [4], PDDL não permite que variáveis númericas sejam usadas em termos da linguagem (*i.e.*, argumentos de predicados ou de ações do domínio). Conseqüentemente, informações sobre o sufixo da seqüência de veículos parcialmente construída durante a busca não pode ser facilmente representada e ficará muito difícil resolver o problema usando planejamento.

Referências

[1] Bonet, B. & Geffner, H. *Planning as Heuristic Search*, Artificial Intelligence, Special issue on Heuristic Search. Vol 129 (1-2), 2001.

[2] Do, M. B. & Kambhampati, S. *Sapa: A Domain-Independent Heuristic Metric Temporal Planner*, ECP, 2001.

[3] Fikes, R. E., and Nilsson, N. J. *STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving*, Artificial Intelligence, vol. 2 (3/4), pages 189-208, 1971.

[4] Fox, M. & Long, D. *PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains*, University of Durham, UK, 2003.

[5] Ghallab, M. *et alli. PDDL—The Planning Domain Definition Language*, AIPS-98 Planning Committee, 1998.

[6] Hoffmann. J. *The Metric-FF Planning System: Translating "Ignoring Delete Lists" to Numeric State Variables*, In: Journal of Artificial Intelligence Research, special issue on the 3rd International Planning Competition, 2002.

[7] Korf, R. E. *Planning as Search: A Quantitative Approach*, Artificial Intelligence, vol. 33, pages 65-88, 1987.

[8] Nguyen, A. *Challenge* Roadef *'2005: Car Sequencing Problem*, Renault, France, 2003.

[9] Pednault, E. P. D. *Formulating Multiagent, Dynamic-world Problems in the Classical Planning Framework*, In Georgeff, M. P. and Lansky, A. L., editors, Reasoning About Actions and Plans: Proceedings of the 1986 Workshop, pages 47-82, Timberline, Oregon. Morgan kaufmann, 1986.