MAC5788 - Relatório do EP2

Viviane Bonadia NUSP 9167607, Diego Araújo NUSP 7157092,
Ignasi Andrés NUSP 8193481
Universidade de São Paulo
São Paulo, Brazil
{vbonadia, diegoamc, ignasi} @ime.usp.br

24 de Abril de 2015

1 Introdução

Neste trabalho vamos apresentar uma implementação do problema de planejamento dos robôs visto na aula. Em primeiro lugar discutiremos como o planejamento pode ser visto como um problema de busca. Na seção 3 descreveremos brevemente o algoritmo A*, e como pode ser adaptado para funcionar raciocinando sobre as ações. Na seção 4 explicaremos concisamente a implementação do algoritmo, e por fim apresentaremos os resultados.

2 Planejamento Automatizado

O planejamento consiste na elaboração de um plano de ação, ou seja um conjunto de ações, para atingir um determinado objetivo [Russell et al., 1995]. O planejamento automatizado é a sub-área da inteligência artificial que estuda este processo de raciocínio de maneira computacional.

Os problemas de busca precisam ter o grafo completo de estados, as possíveis ações para escolher a melhor ação e saber onde essa ação leva o agente. Porém, em um problema de planejamento isso não é sempre possível pois o tamanho do grafo e dos estados pode ser muito grande. Isso se deve ao fato de que um estado, em geral, é composto de várias variáveis. Portanto, o grafo é criado a medida que as melhores ações e seus estados sucessores são selecionados.

O planejamento automatizado pode ser usado em ambientes que podem ser modelados, mas se usarmos uma representação explícita o tamanho do

problema pode ser muito grande e conter estados que não são possíveis, ou que nunca serão visitados.

Para modelar os ambientes, podemos usar um conjunto de variáveis para representar o estado do mundo. Essas variáveis podem ser de dois tipos, segundo os valores que possam tomar [Geffner and Bonet, 2013]:

- Booleanos: as variáveis somente podem tomar os valores de verdadeiro ou falso.
- Multivaloradas: as variáveis podem tomar valores de um conjunto finito de dados.

No primeiro caso, por exemplo, podemos ter o predicado *(robot-at room1)*, que toma o valor verdadeiro se o robot se acha na localização indicada (room1). No segundo caso podemos ter *(robot-at ?x)*, onde $?x$ pode tomar valores no intervalo de $\{room1, room2\}$. As duas representações podem ser equivalentes, pois podemos instanciar o predicado *(robot-at ?x)* com os dois valores e ter as seguintes variáveis booleanas: $\{(robot - at\ room1), (robot - at\ room2)\}$.

A linguagem STRIPS é a base das linguagens usadas atualmente para representar problemas de planejamento. Esta linguagem usa variáveis booleanas para representar o estado do mundo. Um problema de planejamento representado em STRIPS é uma tupla $P = \langle F, O, I, G \rangle$, onde:

- F: representa o conjunto de proposições, ou variáveis booleanas que são usadas para representar os estados do mundo.
- O: representa o conjunto de operadores ou ações que podem ser aplicadas.
- $I \subseteq F$: representa a situação inicial, e é composto por um subconjunto das proposições F.
- $G \subseteq F$: representa a meta, e também é composta por um subconjunto de F.

Em STRIPS as ações $a \in O$ estão representadas por três listas de proposições de F: a lista de Precondições ($Pre(o)$), a lista de efeitos positivos (Eff^+), e a lista de efeitos negativos (Eff^-).

PDDL é uma linguagem que usa a representação STRIPS com uma sintaxe baseada em Lisp. Nosso problema de planejamento está escrito em PDDL e armazenado em dois arquivos. O primeiro é o arquivo do domínio

(*domain.pddl*) que contém informações gerais sobre o problema, como a descrição das ações e o tipo de predicados que existem, assim como algumas constantes. O segundo é o arquivo do problema (*problem.pddl*), que contém informações relativas a uma instância particular do problema, como o número de objetos, a situação inicial e a situação meta.

Por exemplo, no caso do robô que leva as caixas de uma sala para outra, o arquivo do domínio contém a descrição das ações e dos predicados, usando variáveis multivaloradas como $?x, ?y$ para não instanciar todo o conjunto. O planejador desenvolvido instancia todas as ações de duas maneiras:

- Instanciando todo o conjunto de ações antes de realizar a busca,
- Instanciando as ações sob demanda.

Na seção 3 esses conceitos serão detalhados.

3 Algoritmo A*

O algoritmo que resolve o problema de planejamento é o A^* funcionando em grafo, adaptado do código do Coursera [1].

A principal diferença entre o código do nosso planejador e do mostrado no Coursera, é que nosso planejador raciocina diretamente sobre ações, implementando as seguintes funções:

1. *goalTest(s)* Conforme explicado na seção 4.2.2, esta função verifica se os predicados da meta estão contidos no estado *s*.
2. *groundAllActions(Problem)* Esta função está descrita na seção 4.2.1. Como indicado anteriormente, as ações podem ser instanciadas no instante inicial ou sob demanda. Esta função realiza a instanciação no instante inicial, e devolve o conjunto completo de ações instanciadas.
3. *matchApplicableActions(A,s)*, descrita na mesma seção que a função anterior, seleciona do conjunto de ações A, quais são as ações que tem as precondições satisfeitas e podem ser executadas no estado *s*. Se as ações foram instanciadas todas no instante inicial, a cada iteração do A^* temos de executar esta função.
4. *groudApplicableActions(Problem, s)*, esta função realiza a instanciação sob demanda, a cada iteração do A^*. Esta função realiza duas tarefas, em primeiro lugar seleciona as ações aplicáveis e depois as instancia.

[1] https://class.coursera.org/aiplan-003/wiki/week2

Para implementar esta função, nos baseamos na função *AddAplicables* dos slides do Coursera.

5. *expand(a,s)*, descrita na seção 4.2.3, realiza a expansão do nó, executando a ação a no estado s. Para realizar a expansão, a função elimina do estado os efeitos negativos da ação ($Eff^{-}(a)$) e acrescenta os positivos ($Eff^{+}(a)$).

As heurísticas usadas ($h = 1$ e $h = 0$) não fazem diferença no resultado final, e pelo fato de serem constantes, transformam a busca em uma busca em largura.

4 Implementação

Nesta seção, apresentaremos como executar o programa e os seus detalhes de implementação, ressaltando decisões importantes que tomamos durante o desenvolvimento. Explicaremos cada classe individualmente, com ênfase nos seus principais métodos.

4.1 Como executar o programa?

É preciso ter o Ruby (versão acima da 1.9.3) instalado no computador. Além disso, nosso EP possui uma dependência externa, a biblioteca PQueue[2]. Ela é responsável por criar e gerenciar a fila de prioridades que usamos durante o algoritmo. Para instalá-la no Linux, basta rodar:

```
gem install pqueue
```

Dependendo de como a instalação do Ruby foi feita, esse comando precisará de permissão de *root*.

Com as dependências instaladas, na pasta raíz do projeto execute:

```
ruby robot_problem.rb problem4Box2Room.pddl
```

Esse comando rodará o EP para um problema com quatro caixas e duas salas e fará a proposicionalização das ações por demanda. Os problemas que escrevemos em PDDL estão no diretório *problems*. Para proposicionalizar todas as ações antes de realizar a busca, rode:

```
ruby robot_problem.rb problem4Box2Room.pddl all
```

[2]https://github.com/rubyworks/pqueue

4.2 Detalhes do código

Os arquivos mais simples estão descritos a seguir. Arquivos mais complexos, como a classe *Domain* são explicados na próximas subseções.

- **requirements.rb**: Arquivo que importa todos os outros.
- **robot_problem.rb**: Arquivo que roda o problema. Ele analisa os arquivos PDDL do domínio e do problema, roda o algoritmo, mede o tempo de execução e escreve os resultados em um arquivo de texto dentro do diretório *results*, usando o nome do arquivo PDDL do problema.
- **parser.rb**: Analisador de Lisp[3] que adaptamos para analisar PDDL.
- **action.rb**: Representa uma ação do domínio. Ela pode estar instanciada ou não.
- **node.rb**: Representa um nó. Por questões de facilidade de programação, as heurísticas e função de avaliação também ficam definidas nele.

4.2.1 Domain.rb

Em sua inicialiazação, essa classe recebe o resultado da análise feita pelo *parser.rb* do arquivo *domain.pddl*. A partir disso, instanciamos seus atributos. É nessa classe que ficam os métodos de proposicionalização das ações:

- **ground_all_actions**: Recebe o problema de parâmetro e devolve um vetor com todas as ações proposicionalizadas do domínio.
- **groud_applicable_actions**: Recebe o problema e um estado como parâmetros e devolve um vetor com as ações que são aplicáveis no estado.
- **add_applicable**: Recebe o problema, uma ação não proposicionalizada, um conjunto de pré-condições, um conjunto de substuições que foram realizadas e um estado. Recursivamente substitui as variáveis por objetos e decide se a substituição realizada é válida. Se não for, para e testa para o próximo objeto. Se for, continua até não existirem mais variáveis nas pré-condições. Nesse momento, cria uma nova ação e a inclui em um vetor com as ações válidas. Essa função é utilizada pela **groud_applicable_actions** para adicionar as ações aplicáveis de um tipo específico de ação.

[3] http://rosettacode.org/wiki/S-Expressions#Ruby

- **match_applicable_actions**: Recebe um conjunto de ações e um estado. Retorna um vetor com as ações aplicáveis no estado.

4.2.2 Problem.rb

Essa classe é uma abstração do problema. Em sua inicialiazação, ela recebe o resultado da análise feita pelo *parser.rb* de algum dos arquivos PDDL de problemas, localizados no diretório *problems*. A partir disso, instanciamos seus atributos. É nessa classe que fica o teste de meta:

- **goal_test**: Recebe um nó e verifica se a meta está contida no estado. Retorna *true* em caso afirmativo e *false* caso contário.

4.2.3 Search.rb

Essa classe executa o algoritmo. Ela possui alguns detalhes interessantes que valem ser ressaltados.

- **path_to**: Recebe um nó e imprime informações dele, subindo na árvore de busca gerada até a raíz. É nesse método que fazemos os cálculos das estatísticas mostrados nos arquivos de saída.

- **expand**: Recebe uma ação e um estado. Usa a representação de conjuntos vista em aula, ou seja, faz união do estado com os efeitos positivos e depois deleta, do estado resultante, os efeitos negativos.

- **a_star_tree_search**: Recebe o conjunto de ações, o problema, o domínio, a heurística que será utilizada e uma variável *booleana* que indica se a proposicionalização das ações deve ser feita previamente. O nó raíz, a *fringe* e um dicionário de nós visitados são inicializados. Vale notar que a *fringe* é uma fila de prioridade. Para implementá-la, utilizamos a biblioteca PQueue. Em sua inicialização, ela recebe o vetor inicial (que contém apenas o nó raíz) e o critério usado para definir a prioridade dos nós. No algoritmo A*, esse critério é o menor valor da função de avaliação. Depois, obtemos o conjunto de ações aplicáveis para o estado. Isso depende se a proposicionalização foi feita previamente. Em seguida, selecionamos uma ação do conjunto de ações aplicáveis e expandimos o estado utilizando essa ação. Se o estado resultante ainda não foi visitado, criamos um novo nó e o incluímos na *fringe* e no dicionário de estados. É importante ressaltar que, como os custos dos caminhos são unitários, não há necessidade de verificar se o nó criado tem custo menor do que um outro nó, com o mesmo estado, que já tenha sido encontrado em alguma iteração anterior.

5 Resultados

Os algoritmos descritos na seção 4.2 foram testados no problema do robô que transporta caixas. Neste problema temos um robô equipado com dois braços que podem segurar uma caixa cada braço, uma certa quantidade de caixas e uma certa quantidade de salas. A resolução do problema consiste em transportar as caixas que estão em uma determinada sala (esta informação está descrita no estado inicial do problema) para uma outra sala (descritas no estado meta).

No menor caso do problema testado, temos dois quartos, nos quais podemos armazenar as caixas e quatro caixas. Inicialmente o primeiro quarto contém as caixas e o segundo está vazio e a meta é levar todas as caixas do primeiro quarto até o segundo.

Nos testes realizados, crescemos este problema variando o número de caixas que devem ser transportadas de um quarto para o outro. Analisamos o tamanho do plano solução obtido, o tempo de execução gasto pelo algoritmo, o número total de estados gerados e visitados durante a busca e o fator de ramificação médio. Para cada tamanho de problema, rodamos o algoritmo com todas as ações previamente instanciadas e com as ações geradas sob demanda.

Executamos o algoritmo com problemas de 2 até 10 caixas. A figura 1 apresenta o tamanho do plano solução obtido para cada um dos testes realizados. Tanto no caso em que as ações são previamente instanciadas quanto no caso em que são geradas sob demanda, para a mesma quantidade de caixas, o tamanho do plano foi o mesmo. Já o tempo de execução não foram os mesmos para a mesma quantidade de caixas. A Figura 2 apresenta os resultados obtidos. A Tabela 1 descreve com mais detalhes os resultados da Figura 2.

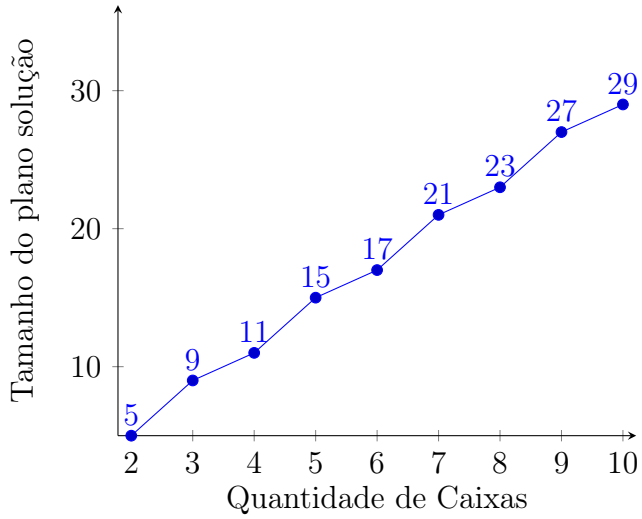

Figura 1: Tamanho do Plano

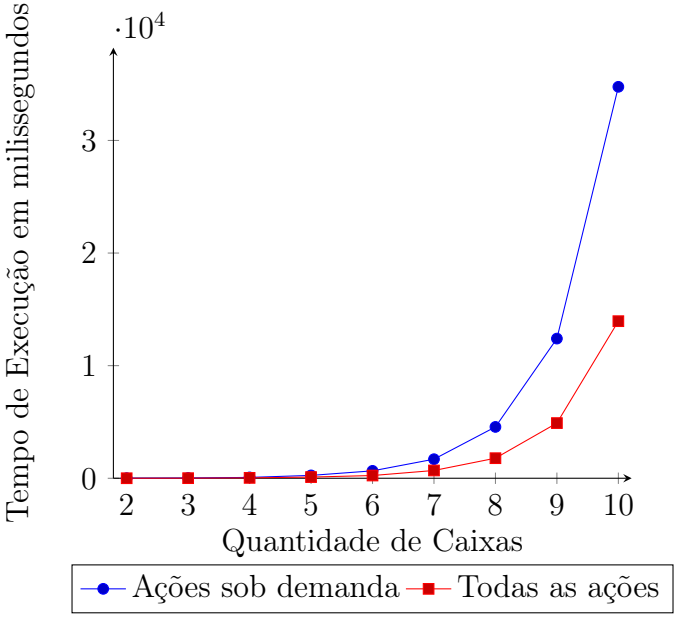

Figura 2: Tempo de Execução

Quantidade de caixas	Ações sob demanda	Todas as ações
2	8.8	2.4
3	23.01	8.72
4	77.63	35.55
5	255.23	106.76
6	659.28	243.69
7	1694.17	691.84
8	4560.15	1782.61
9	12403.77	4900.63
10	34762.09	13949.89

Tabela 1: Tempo de execução em milissegundos

A quantidade de nós gerados e explorados também apresentou resultados diferentes quando as ações foram previamente instanciadas ou sob demanda. A Figura 3 apresenta o fator de ramificação para cada um dos testes realizados. O fator de ramificação é calculado a partir da divisão do número de nós gerados pelo número de nós explorados. A Figura 4 apresenta a quantidade de nós gerados durante a busca e, a Tabela 2 e a Tabela 3 descrevem com mais detalhes os valores apresentados no gráfico da Figura 4.

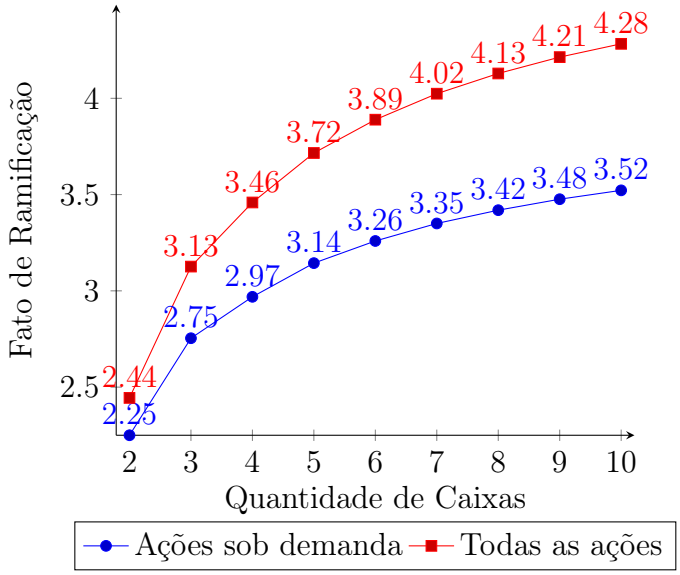

Figura 3: Fator de Ramificação

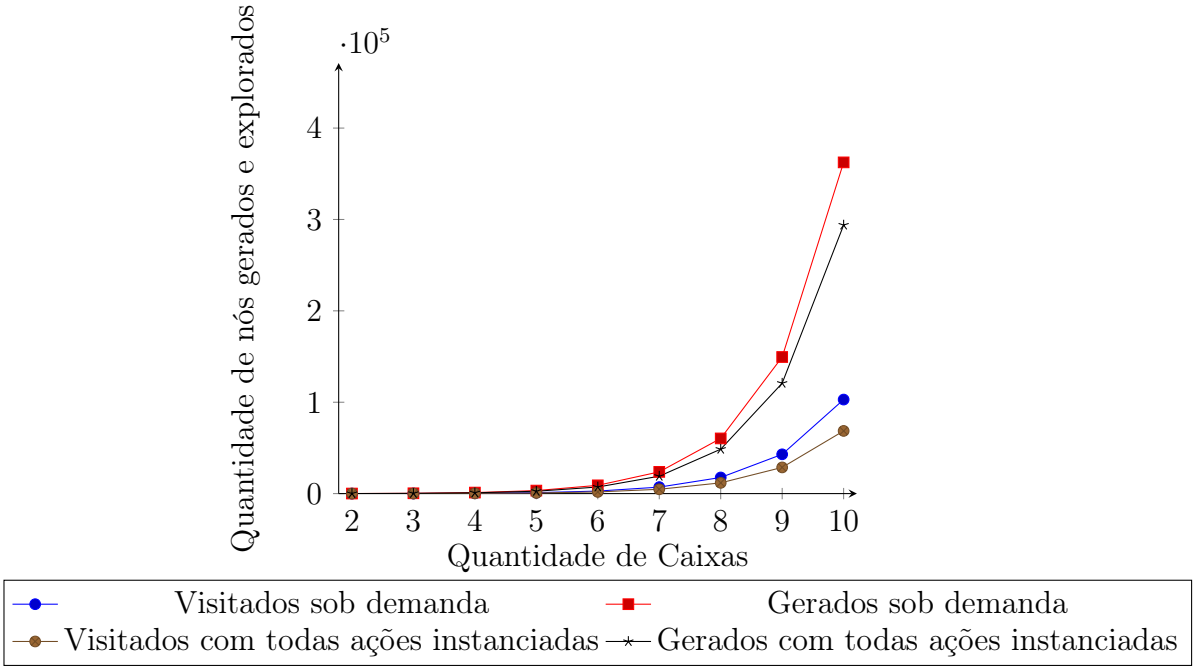

Figura 4: Nós Gerados e Explorados

Quantidade de caixas	Visitados	Gerados
2	40	90
3	130	358
4	382	1134
5	1054	3314
6	2782	9066
7	7102	23790
8	17662	60390
9	43006	149482
10	102910	362466

Tabela 2: Nós Visitados e Gerados para ações criadas sob demanda

Quantidade de caixas	Visitados	Gerados
2	27	66
3	87	272
4	255	882
5	703	2612
6	1855	7214
7	4735	19056
8	11775	48618
9	28671	120812
10	68607	293862

Tabela 3: Nós Visitados e Gerados para ações instanciadas

6 Conclusões

Observando os resultados, podemos ver que o algoritmo que instancia todas as ações antes de começar a busca é realmente melhor. Segundo a figura (fig. 2) o tempo de execução do algoritmo que gera as ações sob demanda é muito maior e cresce muito mais rápido que o tempo de execução do algoritmo que instancia todas as ações. Mesmo que no início do programa possa ter uma sobrecarga por causa do fato de calcular o conjunto de ações, ele se compensa posteriormente pois não tem que criá-las de novo. Mas, por outro lado, gerar as ações sob demanda resulta em um fator de ramificação menor (fig. 3).

Por último, temos que ressaltar que o tamanho do plano cresce de forma linear (fig. 1), pois as soluções tem uma estrutura semelhante (levar duas caixas para a outra sala varias vezes), que se repete dependendo do tamanho do problema. A diferença entre os planos de tamanho ímpar e par e sempre de duas ações, por exemplo os pares 5 e 6, 7 e 8, e assim por diante. Essa diferença é causada por ter que recolher mais uma caixa (ação *pickup*) e descarregá-la depois (ação *putdown*).

Referências

[Geffner and Bonet, 2013] Geffner, H. and Bonet, B. (2013). *A Concise Introduction to Models and Methods for Automated Planning*, volume 8. Morgan & Claypool Publishers.

[Russell et al., 1995] Russell, S. J., Norvig, P., Canny, J. F., Malik, J. M., and Edwards, D. D. (1995). *Artificial intelligence: a modern approach*, volume 74. Prentice hall Englewood Cliffs.