Universidade de São Paulo
Instituto de Matemática e Estatística
MAC 5789 - Laboratório de Inteligência Artificial

Exercício Programa 3: SAT-Plan vs Blackbox

Autor:

Walter Perez Urcia

São Paulo

Junio 2015

Resumo

Neste trabalho o objetivo foi construir um sistema de planejamento clássico de dois formas. O primeiro do tipo SATPLAN utilizando um SAT-solver e o segundo usando o algoritmo BlackBox. A primeira parte do trabalho consiste em fazer um parser que traduz um problema descrito na linguagem STRIPS para uma CNF que será a entrada para o SAT-solver em ambas implementações. A seguinte parte é fazer que o sistema dé uma solução para o problema. Por ultimo decodificar a solução e mostrá-la. A continuação presentamos a implementação, experimentos, resultados e conclusões.

Sumário

Lista de Figuras

1 Dados de entrada

O sistema de planejamento vai receber dois archivos de entrada. O primeiro é o problema descrito na linguagem STRIPS e o segundo é uma possível situação (com estado inicial e meta) para fazer a busca de solução.

1.1 Linguagem STRIPS

O archivo de entrada com o problema descrito vai ter a seguinte forma:

Bloco 1: Estructura de archivo STRIPS

```
1    ( define ( domain [domain_name] )
2      ( :requirements [list_of_requirements] )
3      ( :types [list_of_types] )
4      ( :predicates
5        [list_of_predicates]
6      )
7      [list_of_actions]
8    )
```

Na linha 1 vai o nome do dominio e na linha 2 uma lista de requerimentos (mas não vai ser usada neste trabalho). As seguinte linhas são as principais para descrever o problema. Linha 3 tem uma lista dos tipos de dados que terá o problema e os tipos que vão receber as ações e fluentes. Por exemplo, para o problema dos blocos (ler [1]) temos:

```
1    ( :types block )
```

Da mesma forma temos a linha 5 no bloco 1 que contem uma lista de fluentes com seus parámetros e tipos de cada um. Por exemplo para o problema anterior:

Bloco 2: Lista de fluentes do problema

```
1    ( :predicates
2      ( on ?x - block ?y - block )
3      ( ontable ?x - block )
4      ( clear ?x - block )
5      ( handempty )
6      ( holding ?x - block )
7    )
```

No bloco 3 se mostra a definição da ação *pickup* do problema. Além, cada ação sempre vai ter uma lista de parâmetros com seus tipos, seus precondições e seus efeitos. Por último, a lista de ações vai na linha 7 no bloco 1.

Bloco 3: Descrição de ação

```
1    ( :action pick-up
2      :parameters ( ?x - block )
3      :precondition ( and
4        ( clear ?x )
5        ( ontable ?x )
```

```
6         ( handempty )
7       )
8       :effect ( and
9         ( not ( ontable ?x ) )
10        ( not ( clear ?x ) )
11        ( not ( handempty ) )
12        ( holding ?x )
13      )
14    )
```

1.2 Estado inicial e meta

Como se mostrou em 1.1 se tem fluentes. Estas fluentes vai ser usadas para definir o estado inicial e a meta. Por exemplo para o problema dos blocos podemos ter:

Bloco 4: Archivo com estado inicial e meta

```
1   # Estado inicial
2   clear_a ; clear_b ; ontable_a ; ontable_ b ; handempty
3   # Meta
4   on_a_b
```

As linhas 1 e 3 são só referenciáveis e não ter que estar no archivo de entrada. Com ambos archivos o sistema tem que dar uma solução. A continuação se explicará a implementação do sistema de planejamento.

2 Implementação do sistema

Os pasos para fazer o sistema de planejamento clássico (implementado em python) são os seguintes:

- Parser de STRIPS para Json
- Pre processamento do archivo com o estado inicial e a meta
- Parser de Json para CNF
- Adicionamento de ações (aumento em tamanho do plano de solução)
- Interprete de CNF para Json

O pseudocódigo 5 mostra o algoritmo usado pelo sistema de planejamento para a busca de solução para um problema dado (para SAT-Plan e Blackbox).

Pseudocódigo 5: Algoritmo de busca

```
1  Function Solve
2    While true
3      Add axioms for one more level
4      cnf = generate CNF
5      model = SAT_Solver( cnf )
6      If model exists then
7        Sol = extract solution
8        break
```

```
 9       End If
10     End While
11     Return Sol
12   End Function
```

A linha 3 será explicada nas seções 2.5 e 2.6 para o algoritmo SAT-Plan e Blackbox respectivamente. Além,as linhas 4 e 7 seram explicadas nas subseções 2.4 e 2.7 respectivamente. Por último, o SAT-solver da linha 5 já está implementado e só sera usado. Toda a implementação está no archivo *solver.py*, mas as funções específicas de cada algoritmo estão nos archivos *satplan.py* e *blackbox.py* para cada algoritmo do mesmo nome.

2.1 Definições previas

2.1.1 SAT-Plan

Este algoritmo muda todas as ações e fluentes em STRIPS para sentencias lógicas e usando um SAT-solver busca se existe uma solução para o problema, caso contrário adiciona um novo nível ao plano de solução.

2.1.2 Blackbox

A diferença de SAT-Plan, este algoritmo muda as ações e fluentes a um grafo, desta forma, não é necessário adicionar todas as ações e fluentes para um novo nível. Além, permite que mais de uma ação por nível seja executada adicionando o conceito de mutex (ou exclusão mutua), ou seja, faz relações entre ações que não podem ser feitas juntas (em 2.6 será explicado melhor). Com esto pode reduzir o número de cláusulas do problema. Por último, muda o grafo para uma estructura CNF e usa um SAT-Solver para ver se existe uma solução, caso contrário adiciona um novo nível com só as ações e fluentes possíveis.

2.2 Parser de STRIPS para Json

O archivo STRIPS tem uma estrutura que é muito dificil de usar para fazer cálculos e operações, ainda pior de mudar para CNF diretamente. Mas é possível obter todas as partes necessárias usando expressões regulares e mudar para o padrão Json que tem a forma de um dicionário e é mais fácil de usar. A figura 6 mostra o método principal do parser que recebe o archivo STRIPS.

Código 6: Parser STRIPS/Json

```
1  def convertToJson( filename ) :
2    s = open( filename , 'r' ).read()
3    for ( original , replaceable ) in REPLACEABLE_WORDS.iteritems() :
4      s = s.replace( original , replaceable )
5    lst = {}
6    for ( pattern , key ) in EXTRACT_RULES.iteritems() :
7      matches = extractMatches( pattern , s )
```

```
8       lst[ key ] = curateFunctions[ key ]( matches )
9     newname = os.path.splitext( filename )[ 0 ] + '.json'
10    with open( newname , 'w' ) as jsonfile :
11      json.dump( lst , jsonfile , indent = 4 , sort_keys = True )
12    return lst
```

Primeiro são substituidas algumas cadeias para fazer mais fácil a extração de cada uma das partes e depois executa a função *curateFunctions* respetiva para cada parte (ações, fluentes, tipos, etc.). Por último, salva o novo objeto json em um archivo para não ter que ler o archivo STRIPS outra vez. Esta função está no archivo *converter.py* e as funções de extração para cada parte estão no archivo *extractor.py*.

2.3 Pre processamento

O pre-processamento tem os seguintes pasos:

- Obter o estado inicial e a meta
- Obter todas as variáveis do problema
- Evaluar as fluentes e as ações com as variáveis encontradas
- Adicionar ao estado inicial os fluentes não especificadas no archivo
- Identificar para cada ação os fluentes que não são afeitados por ela

Os primeiros dois passos agora são mais fáceis porque já temos o parser de 2.2 e cada fluente está separado pelo caracter ";"uma de outra. Além, as variáveis sempre estão separadas pelo caracter "_". Para poder fazer o seguinte paso temos a função no código 7.

Código 7: Função que evalua com as variáveis encontradas

```
1  def evaluateWith( self , prop , isAction = False , variables = None ) :
2    if variables == None : variables = self.var
3    if isAction :
4      # Preprocess action
5    lst = self.addVariable( prop.copy() , variables , isAction )
6    if not isAction :
7      # Post process fluent
8    else :
9      # Post process action
10   return lst
```

Na linha 5 é uma função recursiva que adiciona um valor a uma variável em cada chamada. No caso dos fluentes só tem que ser substituidos seus parâmetros com alguns valores, mas para as ações também tem que ser substituidas os valores para seus precondições e efeitos.
O paso 4 é verificar quais fluentes do problema não estão no estado inicial e adicionar seus negações, ou seja, para o fluente P que não está no estado inicial, adicionar $\neg P$.
Por último, para cada ação verificar quais fluentes não são afeitadas por ela para ter essa informação ao momento de gerar o archivo CNF.

2.4 Parser de Json para CNF

Para gerar o CNF a partir de Json temos que definir uma forma de levar uma ação o fluente a uma representação numérica. Depois de fazer o pre-processamento já temos todos os fluentes e ações evaluadas para todas as combinações das variáveis, então se dizemos que N_f = número de fluentes e N_a = número de ações, então temos que o número total de proposições ao inicio será $total = N_f + N_a$. Além, fluentes vai ter um ID de 1 a N_f e as ações de $N_f + 1$ a $N_f + N_a - 1$, mas esto já não vai cumplir para um tamanho do plano mais grande. Então sabemos que o tempo em que ocorre a proposição é importante para dar um ID único, então para qualquer proposição temos que a seguinte função da seu representação numérica.

$$ID = prop.time * total + pos$$

Onde pos será seu posição na lista de fluentes ou na lista de ações dependendo do tipo de proposição. A função no código 8 faz o que queremos obter.

Código 8: Função para obter representação numérica de uma proposição

```
1  def getID( self , prop ) :
2    if prop == None : return ''
3    time = prop[ 'time' ]
4    pos = 0
5    if prop[ 'isaction' ] :
6      pos = getAllValues( self.actions , 'name' ).index( prop[ 'name' ] )
7      pos += self.idactions
8    else :
9      pos = self.predicates.index( prop[ 'name' ] )
10     pos += self.idpredicates
11   ID = pos + time * self.total
12   if not prop[ 'state' ] : ID = -ID
13   return ID
```

Uma vez que temos uma forma de obter um identificador para cada proposição podemos gerar o archivo CNF para cada tipo de cláusula no problema: estado inicial, axiomas e meta. Por cada fluente no estado inicial vamos ter uma linha no archivo CNF com seu ID para o tempo 0. Da mesma forma para cada fluente na meta, mas estas vão ter tempo n (tamanho do plano). Mas para cada algoritmo a forma de mudar para o archivo CNF é diferente e será explicada nas subseções 2.4.1 e 2.4.2.

2.4.1 Para SAT-Plan

Neste algoritmo temos todos os axiomas da forma $A_1 \wedge A_2 \wedge \ldots \wedge A_k \rightarrow B$ (com $k \geq 1$). Cada axioma é mudado para a forma $\neg A_1 \vee \neg A_2 \vee \ldots \vee \neg A_k \vee B$ e colocado em uma linha no archivo CNF.

2.4.2 Para Blackbox

Em caso do algoritmo Blackbox como se diz na subseção 2.1.2, não se tem todos os axiomas se não só os possíveis até esse momento. Além, para mudar o grafo para um archivo CNF, não é necessário colocar os fluentes, só as relações entre todas as ações de cada nível do grafo da seguinte forma:

- Se adiciona uma ação A_F que sempre vai ser $True$ e tem como precondições as metas
- Desde o último nível de ações (tendo em conta também o nível com a ação A_F) se busca quais são seus precondições, ou seja $Pre(A_i)$.
- Para cada precondição se busca quais são as ações que a adicionam, ou seja para cada $p \in Pre(A_i)$ se busca as ações A_j tal que $p \in Eff(A_j)$.
- Então para cada precondição de cada ação A_i temos suas pre-ações (denotada por $PreAct(A_i)$ e adicionamos tres tipos de cláusulas:
  - $A_i \rightarrow Disj(PreAct(A_i))$, onde $Disj(X)$ é a disjunção dos términos en X
  - $Disj(\neg PreAct(A_i))$, onde a negação de $PreAct$ é a negação de cada um de seus términos e então $Disj$ disso é a disjunção de as negações das pre-ações de A_i
  - Por úlitmo, todas as relações de mutex das pre-ações de A_i na forma $\neg B_j \vee \neg B_k$ onde B_j e B_k tem uma relação de mutex e ao menos um delos está em $PreAct(A_i)$

2.5 Adicionamento de níveis para SAT-Plan

Em cada iteração do algoritmo de busca especificado ao inicio da seção 2 vai ser adicionado um nível com os axiomas (cláusulas) para todas as possíveis ações que existem no problema.
Na definicão dada no bloco 3 na subseção 1.1 temos que cada ação tem precondições e efeitos.

2.5.1 Axiomas de precondições

Se dizemos que $Pre(A)^t$ é a conjunção das precondições de A^t então para cada ação adicionamos axiomas da forma $A^t \rightarrow P_i^t$ (com $P_i^t \in Pre(A)^t$). Por ser precondições sempre ter que ser ao mesmo tempo da ação.

2.5.2 Axiomas de efeitos

Se dizemos que $Eff(A)^{t+1}$ é a conjunção dos efeitos de A^t então para cada ação adicionamos axiomas da forma $A^t \rightarrow P_i^{t+1}$ (com $P_i^{t+1} \in Eff(A)^{t+1}$). Como são efeitos de uma ação, os fluentes ocorreram no seguinte instante de tempo.

2.5.3 Axiomas de persistencia

Para cada ação existem fluentes que não são afeitadas por ela. Se dizemos $Pers(A)^t$ é a conjunção dessas fluentes para A^t, então temos que adicionar dois axiomas:

- $({P_i}^t \wedge A^t) \rightarrow {P_i}^{t+1}$
- $(\neg {P_i}^t \wedge A^t) \rightarrow \neg {P_i}^{t+1}$

2.5.4 Axiomas de continuidade

Para cada instante de tempo sempre deve ser executada alguma ação e portanto temos que adicionar o axioma: ${A_1}^t \vee {A_2}^t \vee \ldots \vee {A_{N_a}}^t$

2.5.5 Axiomas de não paralelismo

O axioma anterior permite que se façam mais de uma ação por instante de tempo, mas isso não é possível para solucionar o problema, então para cada par de ações adicionamos: $\neg {A_i}^t \vee \neg {A_j}^t$, para $i \neq j$.
Adicionar todos os axiomas anteriores garantir que sempre se execute uma ação e só uma. Além, que sempre será executada uma ação que cumple seus precondições e por último extende seus efeitos ao próximo instante de tempo.

2.6 Adicionamento de níveis para Blackbox

Neste algoritmo, não é necessário adicionar todos os axiomas porque se terá uma estructura de grafo para saber quais fluentes e ações são possíveis. Então em cada iteração do algoritmo vai ser adicionados só as ações possíveis, ou seja, aquelas que tenham seus precondições como nós do grafo. Além, para todas as ações adicionadas, seus efeitos vai ser adicionados ao grafo.
As aristas do grafo são de tres tipos:

- De precondição
- De efeito
- Mutex

Os primeiros dois tipos são iguais aos especificados na seção 2.5. A continuação será explicado o tercer tipo para ações em um mesmo nível.

2.6.1 Mutex de ações inconsistentes

Se um efeito de uma ação é a negação de um efeito de outra, então se adiciona uma relação de mutex entre elas. Ou seja para os casos que temos $p \in Eff(A_i)$ e $\neg p \in Eff(A_j)$ para todo i e j.

2.6.2 Mutex de interferência

Se uma ação elimina uma precondição de outra, ou seja para os casos onde $p \in Pre(A_i)$ e $\neg p \in Eff(A_j)$ para todo i e j também se adiciona uma relação de mutex entre elas.

2.6.3 Mutex de necessidades que competem

Para cada par de ações i e j, se adiciona uma relação de mutex se $p_1 \in Pre(A_i)$, $p_2 \in Pre(A_j)$ e existe uma relação de mutex entre as precondições p_1 e p_2.

2.6.4 Mutex entre fluentes

Além de ter mutex entre ações também existem mutex entre fluentes, os que são necessários para adicionar os mutex em 2.6.3. Só existem dois condições para que dois fluentes tenham uma relação de mutex:

- Um é a negação de outro (se temos p e $\neg p$ à vez)
- Se todas as maneiras de satisfazê-los são mutex

Adicionar todos os axiomas anteriores garantir que pode ser executadas mais de uma ação por nível se e só se não tenham uma relação de mutex entre elas.

2.7 Interprete de CNF para Json

Por último, uma vez encontrado uma solução com o SAT-solver temos que voltar do archivo CNF com representações numéricas para as representações literais das ações e fluentes. Para isso, só temos que fazer o contrario que está especificado em 2.4 para cada proposição que foi verdadera em cada instante de tempo. O código 9 mostra a implementação desta ideia.

Código 9: Obtenção do representação literal

```
1  getProposition( self , ID ) :
2    isnegation = False
3    if ID < 0 :
4      isnegation = True
5      ID = -ID
6    pos = ( ID % self.total ) - 1
7    resp = ''
8    if pos >= len( self.predicates ) :
9      pos -= len( self.predicates )
10     resp = self.actions[ pos ][ 'name' ]
11   else :
12     resp = self.predicates[ pos ]
13   resp = ( "~" if isnegation else "" ) + resp
14   return resp
```

A função anterior calcula o resíduo de ID com o total de proposições e determina se é uma ação o uma fluente, depois retorna a representação literal dependendo o caso.

Com esto já se pode salvar um archivo com as ações e fluentes para cada instante de tempo no plano de solução onde cada dois linhas representam um instante, sendo a primeira fluentes verdaderos e a segunda a ação executada.

3 Experimentos e resultados

Dado que o sistema de planejamento é geral, foram feitos dois experimentos usando ambos algoritmos: para o mundo dos blocos (subseção 3.1) e para o mundo dos satélites (subseção 3.2). Os dados de entrada são os proporcionados na página no curso (ver [2]). Para todas as figuras, uma linha vermelha é para o algoritmo Blackbox e uma linha azul para o algoritmo SAT-Plan.

3.1 Experimentos com mundo dos blocos

A figura 1 mostra os tamanhos do plano de solução do sistema de planejamento para os problemas proporcionados (com 4 a 8 blocos) e mostra que enquanto o número de blocos aumenta, o tamanho do plano não muda muito. Além, para ambos algoritmos, o plano de solução é igual (por isso, só se ve a linha vermelha).

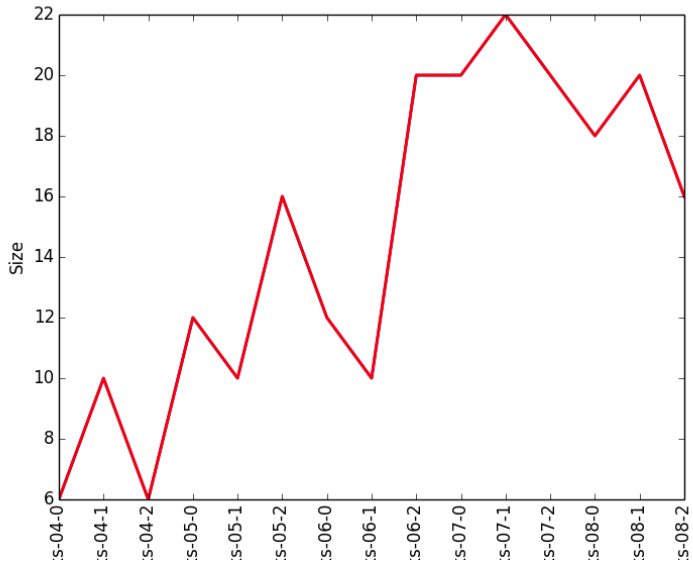

Figura 1: Tamanhos do plano de solução

Mas na figura 2 se poder ver que os tempos de execução aumentam consideravelmente enquanto o número de blocos aumenta desde segundos até mais de uma hora para 8 blocos em caso de SAT-Plan. No caso de Blackbox os tempos são muito pequenos comparados com SAT-Plan para todos os casos.

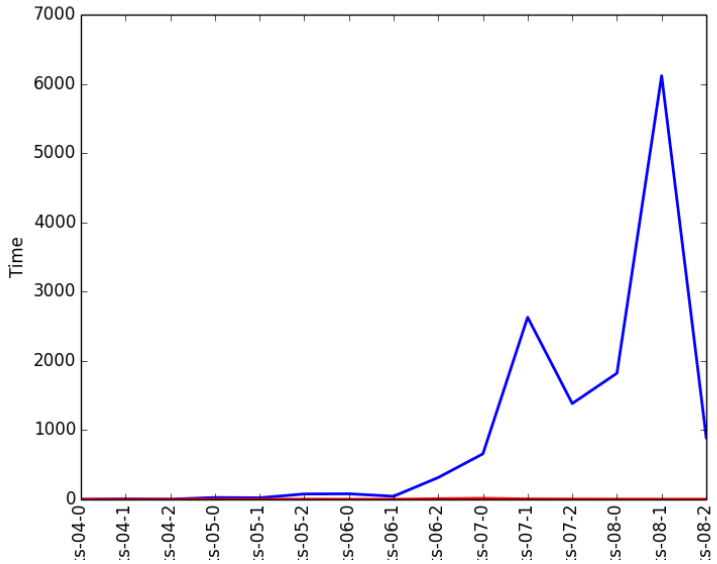

Figura 2: Tempo de execução (segundos)

Além disso, o número de proposições necessárias para solucionar o problema sempre aumenta apesar que o tamanho do plano não muda muito para ambos algoritmos. Mas para o algoritmo Blackbox (linha vermelha) não são necessárias muitas proposições, isto devido a que não coloca as cláusulas de fluentes, se não só de ações e isto adiciona menos proposições comparado com SAT-Plan.

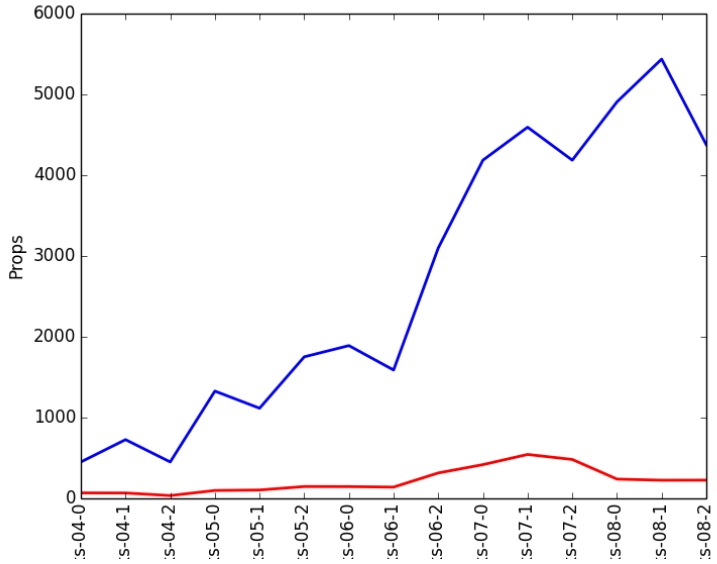

Figura 3: Número de proposições

Por último, o número de cláusulas aumenta exponencialmente devido que se tem

mais variáveis e portanto mais valores para evaluar as ações e fluentes, o que também adiciona mais axiomas em cada iteração dos algoritmos. Mas para o algoritmo Blackbox, o número de cláusulas é consideravelmente menor comparado com SATPlan pela mesma razão de número de proposições.

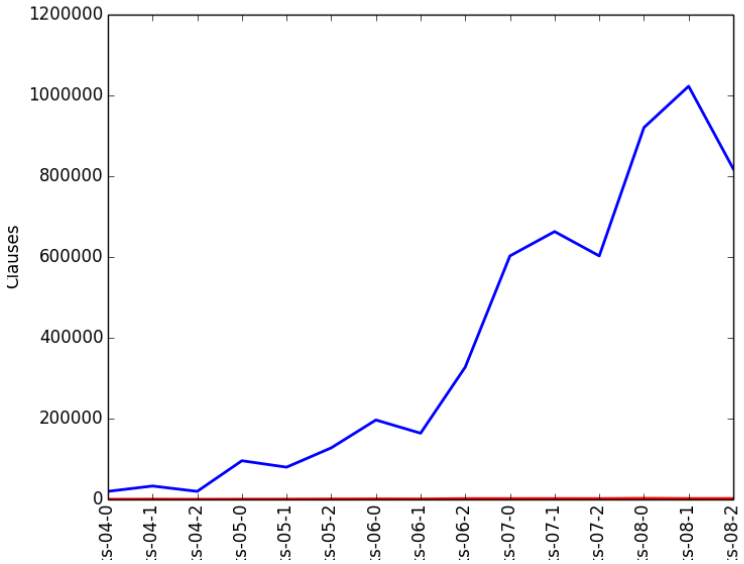

Figura 4: Número de cláusulas

3.2 Experimentos com mundo dos satelites

A figura 5 mostra os tamanhos do plano de solução do sistema de planejamento para o mundos dos satélites com ambos algoritmos, se pode ver que o tamanho do plano não tem uma relação direta com o número de variáveis e também que ambos algoritmos tem o mesmo tamanho de plano de solução para os problemas.

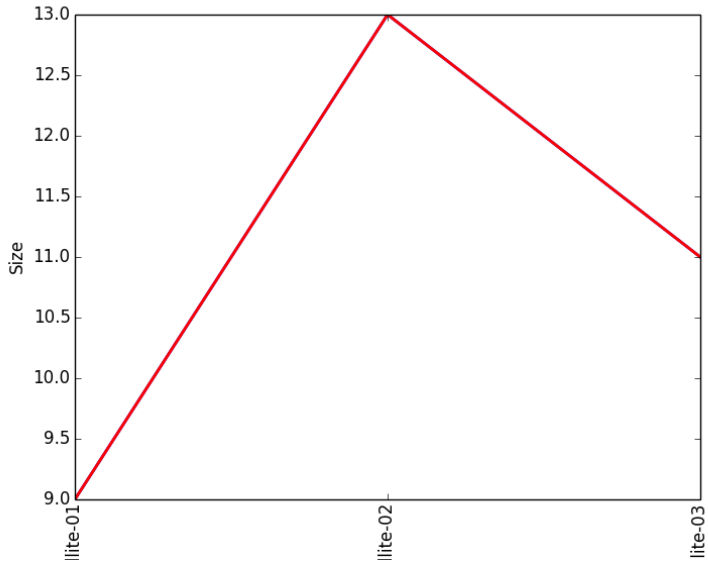

Figura 5: Tamanhos do plano de solução

Mas na figura 6 se poder ver que os tempos de execução aumentam consideravelmente enquanto o número de variáveis aumenta. Como na subseção 3.1 aqui também tem tempos de execução consideravelmente baixos comparados com SAT-Plan.

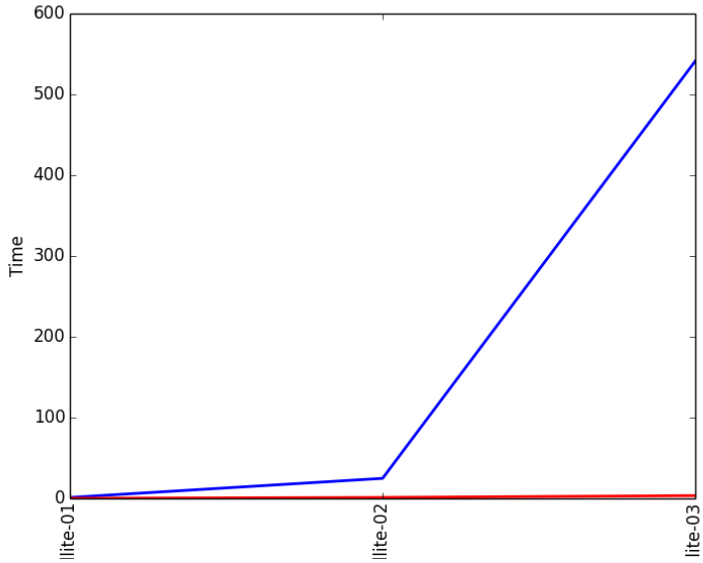

Figura 6: Tempo de execução (segundos)

Além disso, o número de proposições aumenta muito em cada incremento de número de variáveis para SAT-Plan, mas não muda muito para Blackbox.

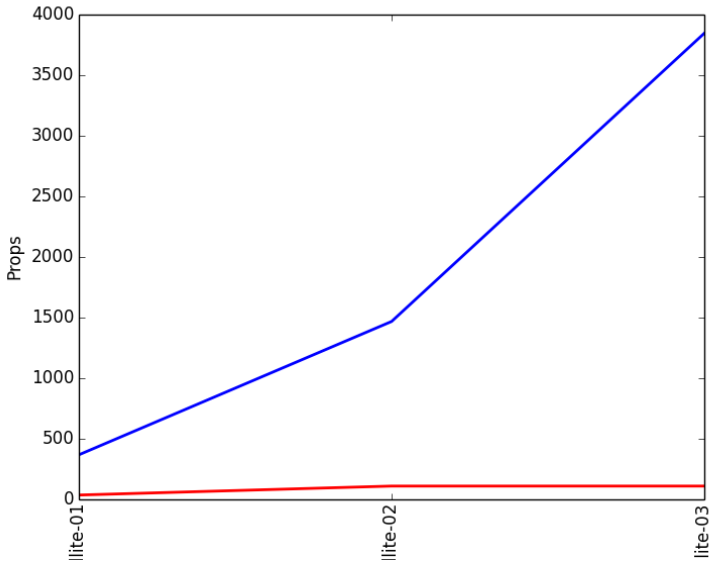

Figura 7: Número de proposições

Por último, são necessárias mais de dois milhões de cláusulas para solucionar o tercer archivo de dados de entrada com SAT-Plan e menos de dois mil para Blackbox.

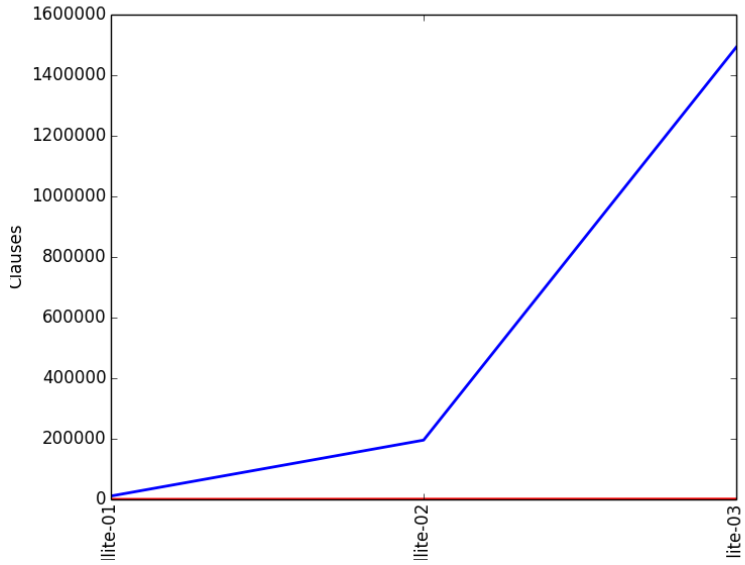

Figura 8: Número de cláusulas

4 Conclusões

Pode-se concluir em geral que:

- O tamanho do plano de solução pode ser similar para diferentes valores de variáveis independentemente do algoritmo usado
- Blackbox tem um melhor tempo de execução comparado com SAT-Plan e encontra um plano de solução do mesmo tamanho que SAT-Plan devido a que só usa proposições de ações possíveis
- Enquanto o número de valores de variáveis aumenta, o número de cláusulas também vai aumentar, da mesma forma o número de proposições e o tempo de execução para ambos algoritmos
- O número de cláusulas é exponencial em número de variáveis para o algoritmo SAT-Plan e afeitando diretamente ao tempo de execução, mas para Blackbox, o número de cláusulas não tem cambios consideráveis porque tem menos cláusulas e só usa os axiomas de ações
- Blackbox é uma melhor opção que SAT-Plan porque reduz consideravelmente o tempo de execução e a memoria usada na busca de uma solução ao problema
- A diferença mais importante entre Blackbox e SAT-Plan é a forma que muda de sua estrutura (axiomas em SAT-Plan e grafo em Blackbox) para um archivo CNF porque o primeiro usa menos cláusulas no archivo para encontrar a solução

Referências

[1] Leliane Nunes de Barros. Planejamento clássico como um problema SAT, 2015.

[2] MAC5789 - PACA IME. Laboratório de Inteligência Artificial, 2015.