Picat: uma Linguagem para Planejamento em Pesquisa Operacional

Claudio Cesar de Sá
Departamento de Ciência da Computação – Universidade do Estado de Santa Catarina (UDESC)
Rua Paulo Malschitzki, 200 - Campus Universitário Prof. Avelino Marcante – 89.219-710 – Joinville – SC – Brasil
claudio.sa@udesc.br

Lucas Hermman Negri
Instituto Federal de Educação, Ciência e Tecnologia de Mato Grosso do Sul (IFMS)
Rua José Tadao Arima, 222 – 79.200-000 – Aquidauana – MS – Brasil
lucas.negri@ifms.edu.br

Alexandre Gonçalves Silva
Departamento Informática e Estatística – Universidade Federal de Santa Catarina (UFSC)
Campus Universitário – Trindade – 88.010-970 – Florianópolis – SC – Brasil
alexandre.goncalves.silva@ufsc.br

RESUMO

Este artigo apresenta a linguagem de programação Picat, destinada à resolução de problemas gerais utilizando diferentes paradigmas de programação, tais como os paradigmas funcional, lógico, por restrições e procedural. Além deste suporte a múltiplos paradigmas, a linguagem possui por padrão o suporte a problemas SAT e de planejamento, sendo este último o foco deste artigo. Considerando que muitos *solvers* da pesquisa operacional não apresentam uma modelagem imediata aos problemas de planejamento, este trabalho apresenta a linguagem Picat como uma alternativa atrativa para o ensino e também como ferramenta para planejamento. Deste modo, a linguagem Picat apresenta perspectivas de uso em disciplinas introdutórias de programação, bem como na resolução de problemas de planejamento.

PALAVRAS CHAVE. Linguagem de programação, paradigmas de programação, pesquisa operacional.

Tópicos: EDU - PO na Educação, OC - Otimização Combinatória

ABSTRACT

This article presents the multi-paradigm programming language Picat, aimed at solving general problems with multiple programming paradigms such as functional programming, constraint programming, logic programming, and procedural programming. Besides supporting multiple programming paradigms, the language also supports SAT and planning problems by default. In particular, the planning area is treated. Considering that many operational research solvers do not present an immediate modeling of planning problems, this work presents the Picat language as an attractive alternative to teaching and as a planning tool.

KEYWORDS. Programming language, programming paradigms, operations research.

Paper topics: EDU – Education in OR, OC – Combinatorial optimization

1. Introdução

A área de planejamento é recorrente em vários problemas reais. Esta teve suas raízes na década de 60, com implementações computacionais dos primeiros algoritmos *planejadores*. Um planejador elabora um plano, dado um estado inicial, um estado final e um conjunto de ações que possibilitam transições entre estes estados. A modelagem do problema consiste em definir os estados e as movimentações possíveis, remetendo ao conceito de *esquemas*.

A complexidade dos problemas desta área são classe PSPACE, haja visto que há dois encaminhamentos sobre encontrar um plano satisfatível[1]

- A existência de um plano dadas as condições previstas nas restrições exigidas para um plano. Este fato torna o problema computacionalmente semi-decidível. Há casos em que nem sempre se encontra um plano;
- A existência de um plano em até k-passos. Este fato o torna difícil pela otimização exigida. Neste caso, o problema torna-se PSPACE-difícil.

Na década de 90, a área de planejamento recebeu atenção da indústria com várias aplicações de sucesso. A linguagem STRIPS de Nillson foi precursora [Russell e Norvig, 2010] e a comunidade trabalhou sobre uma linguagem que fosse padrão em escrever seus códigos (modelos) e testarem sobre os diversos *planners* existente até então. Nesta direção, em 1998 foi definida a linguagem PDDL (*Planning Domain Definition Language*) pela comunidade de planejamento para que houvesse um padrão na definição de seus problemas e estes serem testados e comparados nos diversos planejadores. Neste sentido, tanto a linguagem PDDL apresenta seu próprio planejador, como há tradutores para outras sintaxes de outros planejadores. Ou seja, para que pudessem comparar a eficiência dos diversos planejadores, era necessário que todos tivessem um único modelo de entrada. E assim, desde 1998 existe a IPC (*International Planning Competition*), uma competição entre os planejadores sob diversos modelos escritos em PDDL [Russell e Norvig, 2010].

Suportando uma sintaxe muito próxima ao PDDL, a linguagem Picat [Zhou et al., 2015b] se destaca com um módulo de planejamento clássico com muita flexibilidade. Além deste módulo, Picat tem um módulo com um *solver* SAT, e este também se destacado em *benchmarks* da área de programação por restrições, e problemas de planejamento. Nesta direção, Picat é atrativa pelo fato de exibir uma sintaxe multiparadigma, com módulos (bibliotecas) nas áreas de planejamento, SAT e programação por restrições. Estes atributos vão permitir que a linguagem exiba uma flexibilidade ímpar em seus modelos computacionais, e com isto permitindo heurísticas diversas na construção de planos.

1.1. Motivação

Considerando que problemas das áreas de planejamento e escalonamento (um planejamento com precedências de ações – tarefas, temporalidade e recursos) apresentam respostas eficientes mediante heurísticas em suas implementações, a flexibilidade é um item mandatório em um planejador com o objetivo de contornar a complexidade inicial destes problemas da classe PSPACE. Assim, planejadores e linguagens (mais flexíveis que *planners*) são ferramentas úteis no desenvolvimento de soluções de problemas complexos. Neste sentido, este artigo discute a linguagem Picat sob o ponto de vista de ser uma linguagem para construir modelos de problemas de planejamento. Utiliza-se um problema clássico para apresentar o potencial desta linguagem de programação. Outrossim, esta linguagem exibe facilidades em seu aprendizado e possibilidades de incorporar *solvers*, tais como o Gurobi, na resolução de seus modelos. Ou seja, uma linguagem versátil de propósitos gerais, com interseção ao ensino da pesquisa operacional (PO).

[1] Satisfazível ou consistente.

1.2. Organização do Artigo

Este artigo está organizado da seguinte forma: inicia-se (seção 1) com uma breve introdução à área de planejamento, apresentando também a motivação para o estudo da linguagem Picat. As principais características da linguagem Picat são descritas na seção 2, que acompanha exemplos do uso dos paradigmas de programação suportados pela linguagem, sendo eles o imperativo, o lógico e o funcional. Na seção 3 apresenta-se a configuração geral de um problema de planejamento, enquanto que na seção 4 é discutida a implementação de uma instância deste problema em Picat. As conclusões são apresentadas na seção 5.

2. A Linguagem Picat

O Picat é uma linguagem multiparadigma projetada para aplicações gerais de programação [Zhou et al., 2015b]. Esta foi criada em 2013 por Neng-Fa Zhou e Jonathan Fruhman utilizando o B-Prolog como base na implementação, onde regras lógicas são utilizadas na programação destas linguagens. Alguns dos elementos do Picat seguem a base teórica da linguagem Prolog [Zhou et al., 2015b], isto é, a lógica de primeira-ordem [Enderton, 2001], onde os objetos são chamados por *termos*. Os destaques do Picat é a sua natureza declarativa, funcional, tipagem dinâmica e sintaxe simples com elementos da programação imperativa. O nome Picat é um anacrônico P.I.C.A.T., onde cada letra representa uma de suas funcionalidades:

***Pattern-matching*:** Utiliza o conceito de *casamento de padrão* para selecionar um predicado, o qual define uma regra de programa. Um predicado define uma relação, podendo ter nenhum valor de retorno ou vários argumentos como respostas. Quanto as funções, estas são predicados especiais que sempre retornam um único valor. Análogo às funções clássicas das linguagens de programação. Tanto os predicados e funções são definidos por regras e seguem as *regras do casamento* estabelecidas pela unificação.

***Intuitive*:** O Picat oferece atribuições explícitas e laços de repetição (*loops*). Uma variável atribuída pode imitar várias variáveis lógicas, alterando seu valor seguindo o estado da computação. As atribuições são úteis para associar os termos, bem como utilizadas nas estruturas de laços repetitivos.

***Constraints*:** Picat suporta a programação por restrições. Dado um conjunto de variáveis, cada variável possui um domínio de valores possíveis e restrições para limitar os valores a serem atribuídos às variáveis.

***Actors*:** Atores são chamadas orientadas a eventos. Em Picat, as regras descrevem ações e comportamentos dos atores, os quais podem receber um objeto e disparar uma ação. Os eventos são postados via canais de mensagem e um ator pode ser conectado a um canal, permitindo verificar e/ou processar seus eventos postados no canal. Neste ponto, a linguagem Picat se prepara para suportar interfaces de entrada tais como *mouse*, teclado, *touchpad*, etc.

***Tabling*:** Considerando que operações entre variáveis podem ser armazenadas parcialmente em uma tabela na memória, um programa pode acessar valores já computados. Assim, evita-se a repetição de operações já realizadas, mais precisamente as definidas recursivamente. Esta técnica é conhecida como *memoization*, utilizada na programação dinâmica (PD).

2.1. Elementos da Linguagem

Os elementos desta linguagem seguem os conceitos da lógica de primeira-ordem [Kowalski, 1974] e linguagens imperativas clássicas. A terminologia do Picat tem heranças do Prolog, uma linguagem com 40 anos de existência e ainda largamente utilizada na indústria e academia. Assim, um conhecimento elementar de lógica [Enderton, 2001] auxilia significativamente no processo de aprendizagem do aluno. Sob este ponto vista, estudantes e pesquisadores com conhecimentos prévios em linguagens de programação, matemática, e, preferencialmente, lógica formal, terão uma adaptação rápida a esta linguagem.

Qualquer elemento da linguagem do Picat é chamado de *termo* e este se divide entre *variáveis* e *valores*. Os valores são instâncias das variáveis, e podem ser do tipo *atômico* ou *estruturados* como: listas, vetores, mapas, arquivos, etc.

2.2. Exemplo

Para uma apresentação concisa da linguagem Picat, escolheram-se exemplos clássicos vistos nos cursos introdutórios de programação. Estes exemplos foram abordados por diferentes paradigmas de programação, sendo eles o imperativo, o funcional e o lógico. O detalhamento dos recursos da linguagem encontra-se em [Zhou e Fruhman, 2017].

2.2.1. Paradigma Imperativo

O exemplo abordado é a soma dos primeiros números naturais de 0 até N. Formalmente esta soma é dada por:

$$S(N) = 0 + 1 + 2 + 3 + 4 + ... + (N-1) + N,$$

ou seja, sob uma visão imperativa temos um laço de repetição que soma os valores de 0 até n. Picat dispõem de estruturas de repetição, como ilustrado no exemplo abaixo:

```
%% Usando laços para soma de 0 até N
main => soma_01(7), soma_02(7).

soma_01 (N) =>   %% uso da estrutura foreach
     S := 0,
     foreach(Aux in 1 .. N)
          printf("%d ", Aux),
          S := S + Aux
     end,
     printf("\nSOMA de 1 ate %d: %d\n", N, S).

soma_02 (N) => %% uso da estrutura while-do
     S := 0,
     Aux := 1,
     while (Aux <= N)
          printf("%d ", Aux),
          S := S + Aux,
          Aux := Aux + 1
     end,
     printf("\nSOMA de 1 ate %d: %d\n", N, S).
```

A execução na console para a função `main` é dada por:

```
$ Picat lacos_soma_N.pi
1 2 3 4 5 6 7
SOMA de 1 ate 7: 28
1 2 3 4 5 6 7
SOMA de 1 ate 7: 28
```

2.2.2. Visão Lógica e Funcional

Este problema pode ser reformulado sob uma visão matemática, mais especificamente, pela *indução finita* definida por:

$$S(n) = \begin{cases} 0 & \text{para } n = 0 \\ S(n-1) + n & \text{para } n \geq 1 \end{cases}$$

Sob esta visão recursiva, escreve-se em Picat sob o paradigma funcional e lógico, dado pelo código a seguir:

```
main =>
        soma_p(7, S1),
        printf("Visao logica: %d\n", S1),
        printf("Visao funcional 1: %d\n", soma_f1(7)),
        printf("Visao funcional 2: %d\n", soma_f2(7)).

% Soma como predicado -- visão lógica
soma_p(0, S) => S = 0.
soma_p(N, S), N > 0 =>
                        soma_p(N - 1, Parcial),
                        S = N + Parcial.

% Soma como funcao -- visão clássica
soma_f1(0) = S => S = 0.
soma_f1(N) = S, N > 0 => S = N + soma_f1(N - 1).

% Soma como funcão de fatos -- próximo a Haskell
soma_f2(0) = 0.
soma_f2(N) = N + soma_f2(N - 1).
```

A execução na console para a função `main` é dada por:

```
$ Picat soma_ate_N.pi
Visao logica: 28
Visao funcional 1: 28
Visao funcional 2: 28
```

2.2.3. Resumo

Assim, sejam as bases de conhecimento que um estudante/pesquisador apresentar ao aprender uma nova linguagem de programação, os vários paradigmas permitem uma liberdade de escolha na modelagem de um problema. Esta liberdade, sob o aspecto pedagógico, permite que um aprendiz se sinta confortável em seguir um paradigma ou outro. Finalmente, permite que estes sejam combinados conforme a visão do problema.

3. Planejamento

Um plano é uma atividade de organizar ações fundamentadas numa construção e execução de estruturas simbólicas, como programas computacionais. Os estudos sobre planos iniciaram-se na década de 50. Uma das primeiras abordagens, feita por Karl Lashley, em 1951, como neurofisiologista, estudava o comportamento da atividade neural no processo cognitivo da estrutura fonética da linguagem. A ideia era que elementos básicos da linguagem, os fonemas, possuíssem estruturas neurais "simples" e não decomponíveis. Uma ordem serial dessas estruturas geraria uma sequência de ações primitivas pela aplicação de esquemas ("schemata") habituais [Agre, 1995]. Segundo Lashley, uma frase falada se traduz numa sequência neural primitiva, no caso um esquema. Contudo, a primeira definição aceita de plano vem do livro "*Plans and the Structure of Behavior*" de George Miller, Eugene Galanter e Karl Pribam em 1961:

> *"Um plano é qualquer processo hierárquico de um organismo tal que possa controlar a ordem na qual uma sequência de operações possa ser executada" [George Miller e Pribam, 1960], página 16.*

Outro trabalho mais conhecido no contexto da inteligência artificial (IA) refere-se aos estudos desenvolvidos por Allen Newell e Hebert Simon sobre a modelagem computacional na solução de problemas por meio de buscas. Os conceitos de planejamento tinham a conotação de dirigir uma sequência de operações, de modo que pudessem desencadear uma pesquisa num espaço

de estados. Eles suportavam a ideia de que a cognição humana poderia ser estruturada de forma hierárquica. Há uma decomposição hierárquica sobre a sequência geradora de ações, de modo que disparem suas buscas locais.

Em resumo, um plano contém um repertório de padrões habituais sobre uma sequência de ações conhecidas previamente a serem executadas em situações futuras. O exemplo clássico sobre planejamento é o programa STRIPS, desenvolvido por Nilsson e Fikes [Russell e Norvig, 2010].

3.1. Definições

Um plano pode ser gerado a partir de uma *representação* de três pré-requisitos: do *domínio*, dos *objetivos* e dos *operadores*. A representação de um domínio visa uniformizar o conhecimento sobre o conjunto de seus objetos e suas propriedades invariantes. Essas propriedades não mudam, assim como as ações planejadas e executadas pelo sistema.

A uniformização tem se mostrado útil nas implementações e interações com usuários [Wilkins, 1983]. A representação do objetivo deve ser congruente ao domínio modelado. O objetivo é um estado particular construído a partir dos objetos identificados no domínio. Os operadores descritos para um domínio representam as ações que o sistema pode realizar sobre os objetos. Nas ações possíveis, alguns pontos são avaliados: os objetos participantes, quais tentam atingir a meta, seus efeitos e seus pré-requisitos. Um operador possui uma descrição de como as ações mudam o estado em um domínio. Exemplo: PUTON(X,Y), colocar o bloco “X” sobre o bloco “Y”.

Após o efeito de uma ação, um planejador defronta-se com o problema do “frame”. Esse problema descreve a manutenção das consistências dos estados e uma análise da evolução do sistema. O número de relacionamentos internos começa a crescer a cada ação. Há uma estrutura de dados que deve ser mantida na avaliação desses estados.

Como se observa pelas definições acima, o problema de planejamento se reduz em encontrar um caminho de um estado inicial a um final desejado. Há uma busca por meio dos estados possíveis e pela aplicação sucessiva e sistemática dos operadores sobre os objetos do domínio. Para que se estabeleça uma busca eficiente é necessário um esquema de *controle* para evitar problemas como:

- A “visita” a estados já explorados anteriormente, isto é, a *ciclicidade*;
- A explosão combinatorial sobre estados gerados, mas inconsistentes com o histórico (“frame problem”);
- A sistematização sobre a completude do processo. Ou seja, visitar todos estados que apresentem uma boa perspectiva de solução;
- A solução em um tempo factível.

Esses são alguns itens que a área tem assumido no contexto de controle para planejamento convencional ou de “planejamento como um programa” [Agre e Chapman, 1991]. Quanto melhor o planejamento, menor é a necessidade de controle [Wilkins, 1983]. Dentro dessa sistematização, há necessidade de estruturas de dados que armazenem de forma esquemática os estados permitidos pelo sistema. Normalmente, essas estruturas apresentam-se sob a forma de *árvores* ou *grafos*. Os nós dessas estruturas representam os possíveis estados do sistema, que o plano deve avaliar ou “*visitar*". Para que a visita aos estados do sistema ocorra, é necessário *técnicas de buscas* (ou visitas) sistemáticas sobre essa representação do problema. As descrições dessas técnicas de buscas são encontradas em [Russell e Norvig, 2010].

3.2. Um Modelo Formal de Plano

Considerando as definições acima, podemos construir um modelo para definir um plano. Um plano é definido por um conjunto de pares estados e ações, dado por:

$$plano = \{(e_0, a_0),\ (e_1, a_1),\ ...,\ (e_f, a_n)\}$$

onde e_0 é o estado inicial e e_f é um dos estados finais, tal que $e_f \in E_f$, e a_i é uma ação, tal que $a_i \in A$. Assim um planejador define uma tupla dada por:

$$(e_0,\ E_f,\ E,\ A,\ \Pi)$$

onde E é o conjunto de estados e $E \supseteq E_f$, A o conjunto de ações possíveis em qualquer estado, e Π uma função de transição $E \times A \mapsto E$. Um plano gerado é considerado *completo*, se e_f for alcançado por uma sequência de a_i, repetidos ou não. Este é o ponto da complexidade em gerar planos, pois uma determinada ação pode ser aplicada sucessivamente um número de vezes. Adicionalmente, se um dado estado se repetir, sob uma nova ação aplicada, ou a partir deste estado, então o problema admite múltiplos planos. Sob esta enumerabilidade, um plano é considerado semi-decidível (semi-computável), pois nem sempre um e_f é alcançado [Russell e Norvig, 2010], e se para cada estado há um sub-conjunto de ações possíveis, então tem-se um conjunto de planos factíveis.

4. Experimentação do módulo *Planner*

Para o uso deste módulo na linguagem Picat basta inserir o comando: `import planner.` no início do código fonte. Neste módulo há vários predicados prontos para uso em problemas de planejamento. Basicamente há dois tipos de esquemas de construção de planos: com buscas *limitadas* ou *não-limitadas*. As construções de planos *limitadas* utilizam parâmetros extras, afim de avaliar uma determinada heurística de construção. Estas medidas ou limites podem ser estabelecidas por critérios como: o custo de um plano, o valor da pilha de chamadas que estabelece um limite quanto a recursos utilizados.

Quanto as *não-limitadas*, utilizam uma estratégia de busca ilimitadas, onde um limite pode ser estabelecido, bem como o custo do plano. As estratégias *limitadas* utilizam busca baseada em *aprofundamente iterativo* e *branch-bound* [Zhou et al., 2015a]. Contudo, o diferencial do *planner* do Picat encontra-se no armazenamento dos estados calculados. Aqui, Picat utiliza um armazenamento dinâmico e que não recalcula estados existentes na tabela. Ou seja, a técnica de *memoization* é utilizada na construção destes planos. Assim, as ramificações a partir de estados já conhecidos ou já expandidos, não são exploradas novamente. A eficiência deste planner pode ser ilustrada em [Zhou et al., 2015a] onde vários problemas são resolvidos em Picat e em um planner conhecido como *Symba* [Zhou et al., 2015a]. O número de instâncias otimamente resolvidas é maior no Picat para todos os 9 problemas tratados.

Para modelagem do problema e uso do *planner*, há dois predicados pré-definidos obrigatórios:

***action*:** neste predicado são especificadas as pré-condições, o efeito da ação e seu custo. Em geral este é definido com regras não-determinísticas via casamento de padrões. A exploração ordem destas regras de ações seguem um padrão do Prolog. Isto é, tenta as regras na ordem que vai encontrando de cima para baixo.

***final*:** Aqui se especifica um ou vários critérios de parada ou estado final para o plano. Quanto ao *estado inicial*, este é um termo aterrado definido no predicado de estratégia do plano.

No exemplo aqui apresentado, este conceito e outros permitem ilustrar a facilidade e flexibilidade de se resolver problemas com este módulo. Outros exemplos podem ser encontrados em [Zhou et al., 2015a].

4.1. Exemplo de Uso

O exemplo utilizado para ilustrar o módulo de planejamento do Picat é o de um robô varredor (*cleaner*) que busca todos os símbolos `'@'` de sujeira em um ambiente reticulado. Ao encontrar o símbolo `'@'`, este será substituído por `'.'`. As ações permitidas de movimento a partir de uma célula qualquer pelo agente `'c'` são: para cima (*up_move*), para baixo (*down_move*), para esquerda (*left_move*) e para direita (*right_move*). Estes movimentos são ilustrados na figura 1.

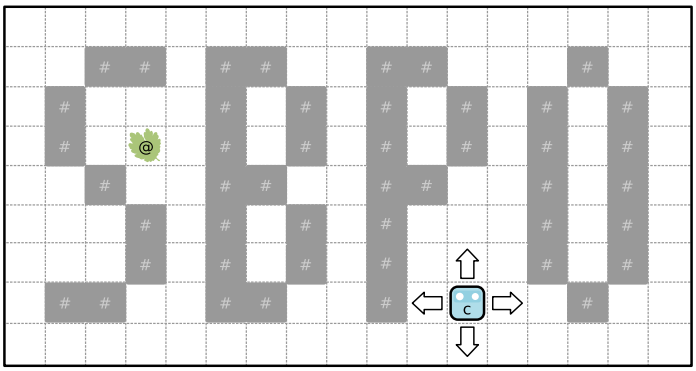

Figura 1: Exemplo de ambiente matricial contendo paredes `#`, sujeira `@`, além de um robô varredor `'c'` e seus quatro movimentos possíveis.

Caso o agente `'c'` encontre `'#'` em seu próximo movimento, este símbolo equivale a bloqueio de uma parede ou muro, e o próximo movimento não pode se suceder nesta direção.

O agente `'c'` também não pode se movimentar além das extremidades dos limites do ambiente (domínio) definido pela entrada.

Um exemplo de entrada neste programa é ilustrado na figura 2, via um arquivo texto que contém a descrição do ambiente.

```
1  9 17
2  . . . . . . . . . . . . . . . . .
3  . . # # . # # . . # # . . . # . .
4  . # . . . # . # . # . # . # . # .
5  . # . @ . # . # . # . # . # . # .
6  . . # . . # # . . # # . . # . # .
7  . . . # . # . # . # . . . # . # .
8  . . . # . # . # . # . . . # . # .
9  . # # . . # # . . # . c . . # . .
10 . . . . . . . . . . . . . . . . .
```

Figura 2: Exemplo de uma entrada

Este exemplo tem o número de estados dado por $m \cdot n$, onde m é o número de linhas e n número de colunas nesta matriz. Em cada estado o agente pode ter 4 movimentos possíveis. Logo, o número total de movimentos é dado por $4 \cdot m \cdot n$. Contudo, a complexidade de um plano bem sucedido, vai depender de cada uma das trajetórias possíveis entre um ponto inicial e seu destino, deste agente. Se considerarmos uma distância euclidiana k entre os 2 pontos de origem e destino, uma aproximação ilustrativa, estima-se que a complexidade para este plano é da ordem de $O((4 \cdot m \cdot n)^k)$. Esta complexidade é aproximada para uma linha direta entre 2 pontos com k células entre eles. Como era de se esperar, a complexidade é exponencial para um simples caso em que nenhuma heurística é aplicada e as 4 direções são testadas a cada passo.

Neste exemplo, ao assumir a ausência de informações prévias sobre o ambiente, tem-se que nenhum algoritmo polinomial como o de Dijkstra possa ser aplicado, e a complexidade do plano torna-se PSPACE-*hard*. Assim heurísticas sobre algoritmos de planos se fazem necessário tornando esta uma das áreas produtivas de pesquisas em algoritmos de buscas. Um plano é construído a partir de buscas exaustivas com ou sem heurísticas. Felizmente, em casos práticos, planos quase-ótimos tem sido encontrados em tempo viável. Ilustra-se o caso do algoritmo simplex, o qual tem o seu pior caso sendo um NP-hard, mas na prática, na maioria dos casos é um algoritmo eficiente [Russell e Norvig, 2010].

4.2. Saída

Os resultados do experimento (*planner* do Picat) podem ser observados na figura 3.

```
CPU time 0.124 seconds.
PLAN: [down_move,left_move,left_move,left_move,left_move,left_move,
left_move,left_move,up_move,up_move,up_move,up_move,up_move,left_move]
Total of states: 14
Total of moviments: 13
```

Figura 3: Parte textual final dos resultados da execução do programa.

A execução deste programa ilustra toda trajetória do agente 'c' durante a busca pelos símbolos '@'. Para acompanhar as ações do plano, veja a figura 4.

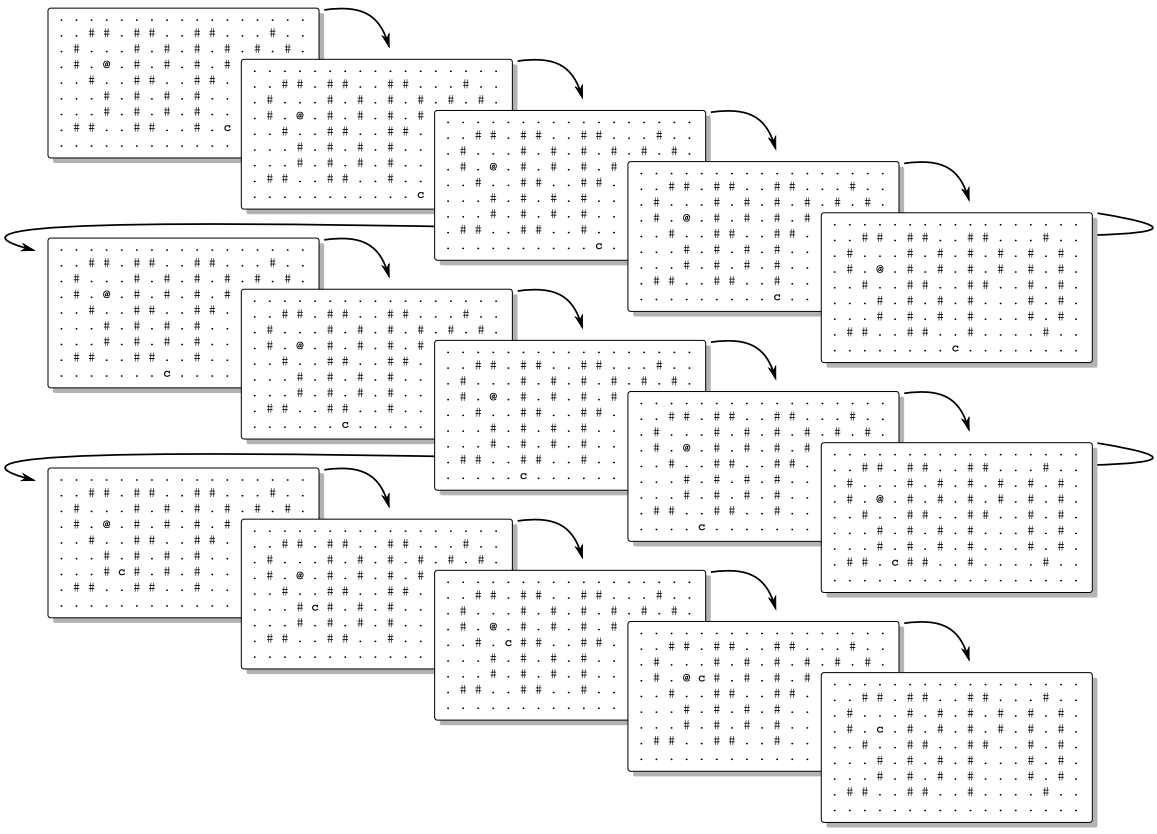

Figura 4: Ações do plano, do estado inicial até o estado final, exibidas, na tela, pelo programa.

O código completo encontra-se em: `https://github.com/claudiosa/CCS/tree/master/picat/sbpo_2017`.

4.3. Comentários sobre o Código

Ao se utilizar o *planner* do Picat, a importação deste módulo é feita pela linha: `import planner.` [Zhou e Fruhman, 2017] Em seguida há uma cláusula `main` que organiza este código nas seguintes seções:

1. Ler o arevquivo de entrada
2. Chamada ao *planner*
3. Imprimir os resultados.

Há várias cláusulas construídas neste experimento, contudo, para o uso do *planner*, apenas dois predicados necessitam ser definidos: *final* e *action*. Para este experimento, o código de *final* é dado por:

```
final( Grid ) =>
          array_matrix_to_list(Grid) = List,
          not member('@', List ).
```

Basicamente, *final* recebe o estado corrente do sistema, representado pela instância em *Grid* (esta é o tabuleiro completo e suas peças), converte o mesmo para uma lista, a qual é testada com o predicado *member*. Este verifica a não existência de nenhum símbolo '`@`' no tabuleiro. Quando não ocorrer nenhum símbolo '`@`' no tabuleiro, o *planner* apresenta uma resposta.

Quanto a `action`, este modela os propósitos do problema, seguindo os conceitos de pré e pós condição, definidos em *planners* tais como STRIPS e PDDL. No código aqui desenvolvido foram definidos 4 *actions*, que são os movimentos do robô *cleaner*: $\{\leftarrow, \rightarrow, \downarrow, \uparrow\}$.

Na cláusula `action` são definidos 4 argumentos, que definem em ordem os elementos do *planner*:

1º argumento: o estado corrente, no caso deste problema são as posições das barreiras–obstáculos, da sujeira e do agente. Tudo isto é sumarizado pela variável tabuleiro ou `Grid`;

2º argumento: o novo estado, caso esta ação seja bem-sucedida. Este novo estado segue para o histórico do planner; neste problema é sumarizado pela variável `New_Grid`;

3º argumento: define uma ação, neste problema, uma descrição atômica textual era o suficiente. Para as ações possíveis foram definidas por: `down_move`, `left_move`, `up_move`, e `right_move`.

4º argumento: define um custo para ação. Importante caso se deseje priorizar e/ou ordenar algumas sub-sequências de ações. Neste problema todas ações eram equiprováveis.

Assim, um exemplo de código para `action`, que o robô se mova uma célula acima de sua posição corrente (as demais ações seguem a mesma ideia) é dado por:

```
action(Grid, New_Grid, Action, Action_Cost ) ?=>
     /** UP MOVE **/
     Grid_TEMP = copy_term(Grid),    %% Uma variável temporária Grid_TEMP
     Action_Cost = 1,                %% Define um custo para esta ação
     Action = up_move,               %% Descricao da ação -- para saída
     position_c_XY( [X,Y], Grid ),   %% Obtém a posição do agente
     %%% PRÉ-CONDIÇÕES:
     (X > 1 , (Grid[X-1,Y] == '.' ; Grid[X-1,Y]=='@') ),
     %% AÇÕES
          Grid_TEMP[X,Y] := '.' ,
          Grid_TEMP[X-1,Y] := 'c',
     %%% PÓS-CONDIÇÕES
     %% atualização do tabuleiro em New_Grid
     New_Grid = Grid_TEMP .
```

Observações:

- A cláusula *action* é *backtrackable*, definida por `?=>`, isto significa que se a mesma falhar, outras abaixo desta serão testadas até encontrar uma ação consistente. A vantagem do Picat sob o Prolog, é que seu mecanismo de *backtracking* é controlável [Zhou e Fruhman, 2017];
- As pré-condições avaliam se a ação é factível. No caso verifica se a direção é permitida, e os limites do tabuleiro;
- A ação neste exemplo é dado por: se célula está livre um movimento é possível, caso haja sujeira '@', igualmente possível;

- A pós-condição, é a atualização em `New_Grid` pelo movimento ou ação causada pelo robô.

Os detalhes de instanciações e casamento de variáveis seguem a programação em lógica. As atribuições existentes são mecanismos disponíveis no Picat afim de dar legibilidade de código [Zhou e Fruhman, 2017].

5. Conclusões

A linguagem Picat é jovem (menos de 4 anos), porém oferece um avanço significativo desde seu principal antecessor, o Prolog [Scott, 2000]. A linguagem Prolog tem sido largamente utilizada na indústria e academia, sendo que outras linguagens a sucederam buscando aprimoramentos a partir desta, tais como: Mercury, Erlang, Oz, Goedel e Curry [Sestoft, 2012; Sebesta, 2003; Tate, 2010]. Contudo, cada uma destas linguagens tiveram seus objetivos bem definidos, enquanto Picat é de propósito geral, portável e multiparadigma quanto a sua sintaxe [Zhou et al., 2015b].

Neste artigo, a linguagem Picat foi apresentada com o objetivo de ser utilizada na área de PO, haja visto que a mesma apresenta módulos para áreas de programação por restrições, planejamento e problemas SAT. Além de apresentar a sintaxe desta linguagem via exemplos, a área de planejamento é revisada sob a visão da IA.

Como motivação do uso desta linguagem em planejamento, destacam-se:

- A flexibilidade de implementação que uma linguagem de programação oferece frente aos *solvers* utilizados na área de PO;
- Linguagens orientadas a modelos, tais como Minizinc, Cometa, etc, são eficazes na descrição precisa dos modelos e seu uso com solvers. Contudo, não resolvem problemas que envolvam a dinâmica de *ações* versus *estados*, um inicial, intermediários e finais;
- Enquanto linguagens de modelagem utilizam uma representação *fatorada* do problema, em Picat a representação é *estruturada* [Zhou et al., 2015a];
- A linguagem enfatiza uma visão moderna e controlável quanto ao seu mecanismo de *backtracking*, tornando-o mais legível, motivado pela clareza de construir regras declarativas para predicados e funções.

Assim, Picat demonstra um potencial de uso devido uma variedade de características dos diversos paradigmas de programação, módulos prontos para problemas da PO, com suporte à *solvers* externos. Além desta disponibilidade como *front-end* para alguns *solvers*, neste artigo ilustra-se o uso do módulo do seu próprio *planner*. Neste quesito, o *planner* do Picat torna-o atrativo como uma linguagem de uso a PO, combinatória, IA, pois apresenta flexibilidade e legibilidade.

Referências

Agre, P. E. e Chapman, D. (1991). What are plans for. In Maes, P., editor, *Designing Autonomous Agents - Theory and Practice from Biology to Enginnering and Back*, Special Issues of Robotics and Autonomous Systems, p. 17–34. Bradford Book - MIT Press.

Agre, P. E. (1995). Computational research on interaction and agency. *Artificial Intelligence*, 72: 1–52.

Enderton, H. (2001). *A Mathematical Introduction to Logic*. Harcourt/Academic Press. ISBN 9780122384523.

George Miller, E. G. e Pribam, K. (1960). *Plans and the Structure of Behavior*. Holt, New York. Apud in Agre [1995].

Kowalski, R. A. (1974). Predicate logic as programming language. In *IFIP Congress'74*, p. 569–574.

Russell, S. J. e Norvig, P. (2010). *Artificial Intelligence - A Modern Approach (3. internat. ed.)*. Pearson Education. ISBN 978-0-13-207148-2.

Scott, M. L. (2000). *Programming Language Pragmatics*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. ISBN 1-55860-442-1.

Sebesta, R. (2003). *Conceitos de Linguagens de Programação*. Bookman Editora. ISBN 9788536301716.

Sestoft, P. (2012). *Programming Language Concepts*. Undergraduate Topics in Computer Science. Springer London. ISBN 9781447141563.

Tate, B. A. (2010). *Seven Languages in Seven Weeks: A Pragmatic Guide to Learning Programming Languages*. Pragmatic Bookshelf, 1st edition. ISBN 193435659X, 9781934356593.

Wilkins, D. (1983). Domain-independent planning: Representation and plan generation. Technical Report Technical Note No. 266R, SRI International, Artificial Inteligence Center, Computer Science and Technology Division, 333 Ravenswood Ave., Menlo Park, CA 94025. The research reported herein was supported by the Air Force Office of Scientific Research, Contract F4920-79-C0188, SRI Project 8871.

Zhou, N., Barták, R., e Dovier, A. (2015a). Planning as tabled logic programming. *CoRR*, abs/1507.03979.

Zhou, N. e Fruhman, J. (2017). A user's guide to picat. Internet. `http://picat-lang.org/download/picat_guide.pdf`, acessada: 14 de junho de 2017.

Zhou, N., Kjellerstrand, H., e Fruhman, J. (2015b). *Constraint Solving and Planning with Picat*. Springer Briefs in Intelligent Systems. Springer. ISBN 978-3-319-25881-2.