Augusto Fredigo Hack

PLANEJADOR AUTOMÁTICO NO ESPAÇO DE PLANOS DE ORDEM PARCIAL

Trabalho de Conclusão de Curso submetida ao Sistemas de Informação para a obtenção do Grau de Bacharel. Universidade Federal de Santa Catarina: Orientadora Profa. Dra. Jerusa Marchi

Florianópolis, SC

2017

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Hack, Augusto Fredigo
Planejador automático no espaço de planos de ordem parcial / Augusto Fredigo Hack ; orientadora, Jerusa Marchi, 2017.
109 p.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Centro Tecnológico, Graduação em Sistema de Informação, Florianópolis, 2017.

Inclui referências.

1. Sistema de Informação. 2. Planejamento automático. 3. Inteligência artificial. I. Marchi, Jerusa. II. Universidade Federal de Santa Catarina. Graduação em Sistema de Informação. III. Título.

Este trabalho é dedicado aos meus amigos
e amados familiares.

AGRADECIMENTOS

Agradeço a todos que me apoiaram durante a elaboração deste trabalho, em especial à professora Jerusa pela sua disposição e apoio durante o tempo que me foi necessário para terminá-lo.

Though many think they are understanding something when really they are merely repeating the word in murmur or running over them in their mind.

(Thomas Hobbes, 1668)

RESUMO

Planejamento automático é um área de estudo da inteligência artificial, com o objetivo de simular raciocínio em máquinas, permitindo que estas atinjam objetivos de maneira abstrata. Planejadores são capazes de elaborar planos a partir de descrições de domínios e problemas, onde quais ações e a ordem de execução necessária para que o objetivo seja atingido é descoberta pela máquina. Planejadores são construídos para auxiliar humanos, automatizar tarefas abstratas e como uma forma de estudo do raciocínio. Para que planos possam ser elaborados diversas técnicas foram desenvolvidas, todas elas baseadas em busca. A busca pode ser realizada em diferentes espaços e representações, é objetivo desse trabalho a construção de um planejador que faz a busca no espaço de planos.

Palavras-chave: Planejamento. Inteligência artificial. Espaço de planos.

ABSTRACT

Planning is the act of elaborating plans that achieve abstract goals. Automated planning is an A.I. research area with the goal of simulating rational thought on machines. Plans are computed by planners using search techniques, based on problem and domain descriptions. Planning is used to support human decision making or to create automated agents. A plan is a complete description of how to achive a goal, with the required actions and their dependecies.

Keywords: Automated planning. Plan space. Artificial intelligence.

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

Planejar é um processo natural aos humanos, um processo abstrato e deliberado onde ações são escolhidas a partir do seu resultado esperado. Planejamento, o ato de planejar, é essencial para resolver problemas complexos, onde para atingir um objetivo é necessário considerar as possíveis ações e suas interações, onde a ordem de execução é importante para que o resultado estipulado seja atingido.

A atividade de planejar é utilizada em diversos contextos, com diferentes objetivos e com diferentes estratégias para atingi-los. Dentre o universo do que se pode planejar temos como exemplos, planejamento de trajetória (*Path planning*) e busca e perseguição (*Hide and Seek*).

Planejamento de trajetória tem como objetivo calcular a trajetória a ser executada por um máquina, como um braço mecânico. A trajetória é calculada a partir de um objetivo, como pegar um objeto próximo à máquina.

Busca e perseguição é uma forma de busca que pode ser realizada por um ou mais agentes colaborativamente, em um ambiente conhecido ou desconhecido, onde os agentes devem levar em consideração a possibilidade de movimentação de seus oponentes.

Planejamento de trajetória e busca e perseguição são exemplos de planejamento de domínio específico, onde conhecimento de domínio é utilizado para solucionar problemas de forma mais eficiente. Como contraste há o planejamento independente de domínio, onde o planejamento não dependente de conhecimento específico. As duas formas de planejamento não são conflitantes, mas possuem diferentes benefícios. A primeira explora conhecimento sobre o problema para ser mais eficiente, enquanto a segunda pode ser utilizada em mais contextos, ao custo de complexidade computacional.

Planejamento automático (*Automated Planning*) é uma área de pesquisa, onde criam-se algoritmos capazes de sintetizar planos de maneira independente de domínio. Planos são compostos por um conjunto de ações, que devem ser executadas por um ou mais agentes, em paralelo ou em sequência, que estima-se, atingirá o objetivo estipulado. Planejamento automático é um subcampo da Inteligência Artificial, que auxilia humanos na tomada de decisão em problemas complexos, ou automatiza a criação e execução de planos para agentes autônomos

(MCDERMOTT, 1995).

1.1 OBJETIVO

O objetivo desse trabalho é desenvolver um planejador automático genérico, que solucionará problemas clássicos de planejamento. O planejador será capaz de gerar planos parciais, aplicando técnicas de busca no espaço de planos parciais.

A formulação clássica do problema de planejamento é mais simples, pois nesta formulação considera-se apenas ações discretas. Para ações discretas assume-se que não há possibilidade de falha na sua execução, e que não há benefícios entre a escolha de diferentes ações.

Planos são denominados parciais quando a ordem entre as ações não é total, diferente de planos clássicos onde ações são totalmente ordenadas. O ordenamento de ações em um plano sempre será necessário para eliminar conflitos, no entanto o uso de ordem total é desnecessariamente restritivo, já que proíbe a execução de ações em paralelo.

Planos parciais foram escolhidos porque estes possuem mais de uma linearização, portanto um plano de ordem parcial pode representar um grande número de planos clássicos. O ganho em expressividade inerente aos planos parciais permite que agente(s) tenham maior liberdade durante a sua execução.

O espaço de busca de planos parciais foi escolhido pela sua naturalidade. O objetivo é explorar esta técnica que deixou de ser popular em meados da década de 70.

Objetivos específicos:

- Desenvolver um planejador para o problema clássico do planejamento
- Gerar planos parciais, onde não existe uma ordem total entre as ações

1.2 ORGANIZAÇÃO DO TRABALHO

Este trabalho está organizado da seguinte forma:

- Capítulo 1: Introdução ao trabalho.
- Capítulo 2: Fundamentação teórica. Este capítulo apresenta uma breve descrição histórica dos planejadores clássicos, uma descrição formal do problema do planejamento, seis diferentes representações utilizadas para planejar, e por fim algoritmos de busca que podem ser utilizados para elaborar planos.
- Capítulo 3: Apresentação do *PNKE*, planejador desenvolvido como parte do trabalho, que constrói planos parciais utilizando a representação baseada em variáveis de estado parcialmente instanciadas.
- Capítulo 4: Avaliação do planejador *PNKE* e trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo contém um breve levantamento histórico, contendo tanto a evolução da inteligência artificial quanto de alguns planejadores automáticos. O levantamento histórico será feito até meados da década de 70, onde a busca no espaço de planos parciais deixou de ser popular.

Após a descrição histórica será introduzido o formalismo baseado em um sistema de transição. Este formalismo descreve o planejamento clássico, e servirá para apresentar propriedades que se mantem válidas independente da representação utilizada por um planejador.

Após o modelo de transição serão apresentadas cinco representações para o planejamento clássico. Em grande parte estas representações são equivalentes, mas possuem diferentes características de complexidade computacional. Detalhes sobre a complexidade computacional de algumas destas representações estão disponíveis no apêndice B.

Por fim, são apresentados algoritmos de buscas, os quais podem ser utilizados para solucionar problemas de planejamento clássico.

2.1 PERSPECTIVA HISTÓRICA

O conceito de planejamento é primeiro apresentado por uma perspectiva histórica. Aqui será introduzido de maneira breve algumas das motivações que impulsionaram o desenvolvimento da inteligência artificial, com um foco especial para a aplicação de lógica ao problema de planejamento.

2.1.1 A lógica da inteligência

A inteligência artificial é um campo jovem, fortemente influenciado por outras áreas de conhecimento, como a engenharia elétrica, matemática e lógica. Essas influências deixam o campo interessante não apenas pela sua utilidade e objetivos ambiciosos, mas também pela sua natureza multidisciplinar. Dentre estas disciplinas, a lógica tem um

papel especialmente importante.

Um dos trabalhos pioneiros na área da lógica é o livro Organon (ARISTÓTELES, -350b), do filósofo grego Aristóteles, que desenvolveu uma técnica para o estudo do diálogo. Foram nos capítulos *Analytica Priora* e *Analytica Posteriora*, deste livro, que Aristóteles deu início ao estudo da lógica. No primeiro desenvolvendo a ideia das premissas, sentenças tidas como fato sem a necessidade de prova, no segundo como novas sentenças podem ser construídas a partir de outras. Juntos, estes livros formaram o que veio a ser conhecido como silogismo.

Um dos principais interesses de Aristóteles para desenvolver esse trabalho foi o estudo da retórica, relação que continuou existindo e é clara no trabalho de George Boole, "As leis do pensamento" (*The Laws of Thought*) (BOOLE, 1854), onde Boole se dedicou à criação da aritmética da lógica, uma fundamentação para a lógica como uma linguagem para expressar o pensamento.

Influenciado pela lógica booleana, Claude Shannon desenvolveu, em *A symbolic analysis of relay and switching circuits* (SHANNON, 1938), um método para simplificar a esquematização de circuitos lógicos, não só criando uma técnica mais eficiente para que circuitos lógicos fossem desenvolvidos, mas também relacionando as leis do pensamento com o desenvolvimento de computadores (LAIRD; ROSENBLOOM, 1992).

A influência da lógica no desenvolvimento de circuitos digitais foi mais um catalisador para pesquisas em inteligência artificial, se circuitos lógicos são modelados utilizando lógica booleana, uma linguagem para expressar o pensamento, então pode-se imaginar que circuitos lógicos permitem que computadores raciocinem simbolicamente. Além da sua importância no âmbito filosófico da inteligência artificial, a lógica desempenha um papel importante como ferramenta para resolver problemas complexos, graças ao trabalho de lógicos como Gotllob Frege e Gerhard Gentzen, que formalizaram quantificadores e a dedução natural. Quantificadores são construções que permitem referir-se a um universo de objetos, e a dedução natural um sistema para tomada de decisões com base em lógica (PELLETIER, 2000).

2.1.2 Inteligência Artificial

O termo Inteligência artificial (*Artificial Intelligence*, IA) foi cunhado pelo cientista da computação John McCarthy em 1955 (MCCARTHY et al., 1955), época na qual pesquisas e desenvolvimento de técnicas para a criação de sistemas computacionais inteligentes teve início. A promessa inicial da comunidade de inteligência artificial era de que nos anos seguintes máquinas seriam capazes de realizar tarefas complexas, que até então apenas humanos realizavam.

Para se definir o que é inteligência artificial primeiro é necessário saber o que é inteligência. Apesar de ser um conceito familiar a todos, definir o que é inteligência não é uma tarefa simples. Inteligência está intimamente atrelada à percepção humana, por isso a sua definição é abrangente e imprecisa. Desde Aristóteles que definiu *nous* (intelecto, mente) como 'a parte da alma pela qual se sabe e entende'(ARISTÓTELES, -350a) até hoje não há consenso nem definição precisa do que é inteligência, pode-se no entanto considerar que a inteligência artificial é *'a capacidade de uma máquina tomar decisões que pareçam racionais ou humanas'* (RUSSELL; NORVIG, 1995).

Além de se definir o que é inteligência também é necessário que haja uma forma de se testar a inteligência, ou no caso da inteligência artificial testar a capacidade de uma máquina parecer racional. Alan Turing já vislumbrava a possibilidade de que as máquinas se tornassem inteligentes e propôs um teste de inteligência para elas, chamado de teste de Turing em sua homenagem. Para o teste é necessário a participação de dois humanos e da máquina, um dos humanos tem como responsabilidade decidir qual dos outros também é humano, a máquina é considerada inteligente se ela for capaz de se passar por um humano (TURING, 1950).

Apesar dos avanços na área de IA, a promessa de uma inteligência artificial tão ou mais capaz do que a humana se provou mais difícil do que inicialmente imaginado, ainda não há uma IA capaz de raciocinar de forma genérica sobre qualquer problema. Mesmo não tendo atingido o seu objetivo, a IA beneficiou de maneira profunda a computação e áreas relacionadas. Entre os resultados obtidos pela IA temos a criação de famílias inteiras de linguagens programação, como Prolog e Lisp, desenvolvimento de técnicas importantes como a tipagem dinâmica, coleta de lixo (*garbage collector*) e casamento de padrões (*pattern matching*). IA também trouxe avanços na compreensão e no desenvolvimento de téc-

nicas para soluções de problemas de alta complexidade computacional, como o problema do caixeiro viajante (LARRANAGA et al., 1999).

A inteligência artificial continua evoluindo e atacando problemas complexos, como reconhecimento de imagens e descrição do seu conteúdo (KARPATHY; JOULIN; FEI-FEI, 2014), técnicas de IA também estão sendo usadas em aplicações comerciais de sucesso, como é o caso de sistemas de recomendações que são utilizados por lojas de comércio eletrônico ou de conteúdo online (KOREN, 2009), sistemas de reconhecimento de voz para busca (WONG, 2014), veículos autônomos (GOMES, 2014) entre várias outras aplicações.

A inteligência artificial possui dois grandes dogmas, com profundas diferenças em como abordar os problemas. Uma abordagem possui um maior foco no resultado das aplicações, usando técnicas como aproximações baseadas em modelos estatísticos, sistemas especialistas e uso de heurísticas para solucionar problemas, essa abordagem é denominada de IA Fraca (GARDNER, 1996). Outra perspectiva é a criação de sistemas com capacidade cognitiva, uma tentativa de criar inteligência virtual, com capacidade de aprendizado, planejamento e tomada de decisão, uma abordagem que prioriza solução de problemas de forma generalizada, essa linha de pensamento é denominada IA Forte (BRACHMAN; LEVESQUE, 2004).

2.1.3 Evolução histórica dos planejadores

Em 1956 três pioneiros na área de inteligência artificial iniciaram os seus trabalhos para desenvolver máquinas capazes de raciocinar. Allen Newell, Herbert A. Simon e posteriormente Cliff Shaw foram inspirados pelo trabalho de Claude Shannon, enquanto trabalhavam na *RAND Corporation*, para desenvolver o primeiro software capaz de raciocinar simbolicamente. É interessante notar que para os autores e muitos dos pesquisadores da época, agir de forma inteligente é o mesmo que ser criativo para resolução de problemas (*problem solving*), por isso muitos dos trabalhos da época são resolvedores de problemas, o que alguns autores consideram como uma disciplina mais genérica do que o planejamento(GARDNER, 1996), e outros mencionam como apenas um outro nome para o planejamento (FIKES; NILSSON, 1971).

Newell, Simon e Shaw construíram durante o natal daquele ano o primeiro software de inteligência artificial, chamado *Logic Theorist*

Machine (LT) (MICHALEK, 2001), um software criado para ser capaz de provar teoremas de lógica simbólica do livro *Principia Mathematica* de Whitehead e Russel. Newell e Simon usaram labirintos como analogia para definir a estratégia utilizada pelo LT para encontrar provas para os teoremas. Como encontrar a saída de um labirinto, encontrar provas para problemas lógicos é uma tarefa não trivial. É necessário explorar diferentes trajetórias, sem conhecimento de qual delas é a mais promissora, até que a saída seja encontrada. Na analogia com labirintos, os caminhos são todas as sentenças lógicas válidas, a trajetória da entrada à saída equivale a prova de um teorema, e provas são encontradas por busca, tal como labirintos são explorados para encontrar uma saída. Hoje chama-se o labirinto de espaço de busca (*search space*), que é composto por todos os valores possíveis de um problema, e o explorar o espaço de busca chama-se simplesmente de busca.

O Logic Theorist usava heurísticas extensivamente, para melhorar a sua performance e garantir sua execução dentro das limitações de hardware da época. Entre as heurística estavam testes de similaridade e simplicidade, onde o objetivo era eliminar trabalho duplicado e preferir caminhos mais simples. Além das heurísticas o LT também tentava aprender com os seus sucessos, onde um caminho que fora bem sucedido em um problema era reutilizado com os novos problemas. O aprendizado no entanto, ao invés de melhorar a performance do LT, teve o efeito de adicionar dependências de ordem em que os problemas eram resolvidos (MINSKY, 1961), prejudicando ao invés de ajudar a busca por soluções.

Para o LT encontrar uma solução é o mesmo que fazer uma busca pelo espaço de sentenças válidas, o tempo necessário para encontrá-la é diretamente proporcional ao número de sentenças geradas a cada etapa (*branch factor*), por isso Newell e Simon preferiram utilizar indução para a busca, iniciado-a a partir dos axiomas do *Principia* em direção aos teoremas, percorrendo o labirinto da busca de maneira reversa, da saída à entrada. A observação é que para a prova de teoremas a indução possui um *branch factor* menor do que a dedução, ou seja, iniciar a busca pelos axiomas irá explorar um número menor de sentenças, em relação a busca iniciada a partir dos teoremas, resultando em um espaço de busca menor e numa execução mais rápida (NEWELL; SIMON, 1956).

Logic Theorist teve grande influência nos planejadores que o sucederam, como pode ser notado na figura 1. Apesar de sua importância o Logic Theorist teve resultados limitados, ele foi capaz de gerar provas para apenas dois terços (trinta e oito) dos teoremas do capítulo dois.

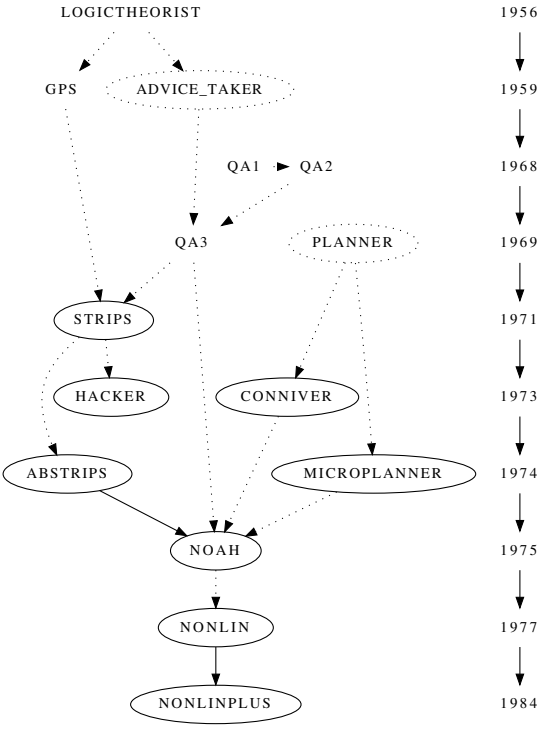

Figura 1 – Influências entre planejadores durante a história

Além de ser um software de domínio específico, onde as regras de dedução para a lógica de primeira ordem e os teoremas faziam parte de seu código, o que não permite o seu uso para solucionar novos problemas. Durante o mesmo período Hao Wang desenvolveu um outro resolvedor de teoremas, que provou 400 teoremas do *Principia*, utilizando um sistema formal equivalente, porém mais sucinto ao LT (MINSKY, 1961; WANG, 1960). Apesar de resultados limitados, o LT continua tendo grande importância, pelo uso pioneiro de heurísticas, buscas, e pelo desenvolvimento da linguagem de processamento de listas (*Information Processing Language*, *IPL*) uma grande influenciadora para o desenvolvimento de LISP.

Como evolução do LT, os seus autores criaram, em 1958, o

General Problem Solver (GPS), software criado com o objetivo de simular o comportamento de alunos de graduação na resolução de problemas. Além das técnicas já adotadas no LT, como a busca guiada por heurística, os autores introduziram a análise de meios e fins e o uso de tabelas de diferença. Tabelas de diferença foram introduzidas como uma técnica para generalizar o software, permitindo descrever como computar novos estados da busca independente do código fonte. Análise de meios e fins como uma nova forma de guiar o processo de busca, considerando apenas as tabelas relevantes para resolver o problema. GPS é portanto mais genérico e eficiente do que o LT (RUSSELL; NORVIG, 1995; NEWELL; SHAW; SIMON, 1959).

Na mesma época em que o sucessor do Logic Theorist era desenvolvido, John McCarthy propôs a criação do *Advice Taker*, com uma preocupação especial na representação do conhecimento, como tinha sido notado pelos autores do LT uma das fraquezas do software era a sua rigidez, a impossibilidade do usuário definir novas regras para o sistema, isso acontece porque todo o conhecimento e heurísticas do LT foram codificadas diretamente no programa, dificultando a adição de mais conhecimento (NEWELL; SHAW; SIMON, 1959). Para sanar essa deficiência McCarthy propôs a utilização de uma linguagem declarativa, para definição do problema e das heurísticas, e a possibilidade de adicionar conhecimento ao programa (MCCARTHY, 1968). Além da preocupação com a adição de conhecimento ao programa, McCarthy também introduziu a ideia de descrever problemas com predicados e fórmulas, e a manipulação de sentenças como técnica para solucioná-los. A motivação de McCarthy era o desenvolvimento de softwares que fossem capazes de aprender, como é descrito pelo próprio no texto do *Advice Taker*:

> "In order for a program to be capable of learning something it must first be capable of being told it."

Em 1963 as ideias introduzidas pelo *Advice Taker* foram formalizadas por McCarthy no artigo *Situações, ações e leis de causalidade* (*Situations, Actions, and Causal Laws*). O ponto de vista de McCarthy é que a inteligência humana não depende de uma representação completa do mundo, mas apenas que uma representação parcial é suficiente para compreender os efeitos de ações e para planejar. A ideia central do formalismo é a situação, uma forma de se entender a progressão de fatos no tempo, onde cada situação possui fatos, e a partir dos fatos sabe-se quais ações podem ser executadas. O cálculo situacional é portanto um

sistema formal que pode ser utilizado para descrever conhecimento e escolher ações a partir de situações (MCCARTHY; LABORATORY, 1963).

No ano de 1969 McCharty e Hayes publicaram o artigo *Some Philosophical Problems From The Standpoint of Artificial Intelligence* abordando vários assuntos relacionados a inteligência artificial, como o cálculo situacional e o *frame problem*. O *frame problem* é uma consequência do uso de sistemas lógicos para solucionar problemas de planejamento, nesta abordagem ações são descritas por axiomas lógicos parametrizados pela situação. A situação é a coleção de todos os fatos conhecidos pelo agente sobre o mundo, fatos esses que são utilizados para definir quais ações são elegíveis para execução. A execução de uma ação não só depende da situação atual como também cria uma nova situação, onde alguns dos fatos conhecidos sobre o mundo são alterados enquanto a grande maioria dos fatos se mantém preservados. Preservar os fatos entre situações é necessário para manter a consistência interna do resolvedor automático. O *frame problem* é portanto, a necessidade de descrever as ações de maneira a preservar todos os fatos do mundo que se mantem inalterados pela execução desta (MCCARTHY; HAYES, 1969).

Nesse mesmo ano, Cordell Green aplicou as ideias do *Advice Taker* no desenvolvimento de um software de perguntas e respostas chamado QA3 (*Question Answering System*). O QA3 utiliza uma base de conhecimento que pode ter expandida pelo usuário, para isso Green utilizou a estratégia proposta por McCarthy e representou o conhecimento utilizando expressões em lógica de primeira ordem. As expressões lógicas são salvas na base, que é consultada para gerar respostas às perguntas do usuário. Pelo fato do QA3 ter sido construído utilizando o formalismo da lógica de primeira ordem, ele também era capaz de raciocinar simbolicamente e solucionar problemas de planejamento. No QA3 ações são definidas como axiomas, situações são denominadas estados e são declaradas como fatos na base de conhecimento. A construção de um plano é equivalente a uma prova construtiva de um teorema, onde o software utilizava o princípio da resolução para fazer suas deduções. Como os planos gerados pelo QA3 eram baseados na prova de teoremas, o problema do *frame problem* também se aplicava a ele (GREEN, 1969).

No ano seguinte do desenvolvimento do QA3, Nilsson e Fikes, também do *Stanford Research Institute* (SRI), desenvolveram o STRIPS (*Stanford Research Institute Problem Solver*), um dos planejadores mais influentes até hoje. O STRIPS foi desenvolvido como o planejador res-

ponsável por controlar o robô SHAKEY, e dentre as suas contribuições está a união das ideias do GPS e QA3. Os autores do STRIPS notaram que tanto o GPS quanto o QA3 não estavam separando o problema do planejamento de maneira adequada (FIKES; NILSSON, 1971), *misturando a busca por modelos de mundo com a busca por ações válidas.* A consequência para o QA3 e GPS foi o uso de representações mais complexas do que o necessário. No caso do GPS formular as tabelas de diferença era trabalhoso (MCCARTHY; HAYES,), enquanto no QA3 o problema era a adição de axiomas de quadro para solucionar o *frame problem* (GREEN, 1969).

STRIPS é um planejador no mundo de estados, para sua representação interna STRIPS utiliza a mesma estratégia do QA3, um conjunto de cláusulas bem formadas em lógica de primeira ordem. Durante o planejamento a aplicação de uma ação gera um novo estado de mundo, o que permite representar os seus efeitos. A aplicação de uma ação, portanto, gera um novo estado onde o conjunto de cláusulas verdadeiras é alterado.

Em STRIPS ações são definidas em termos de *operators schema*, ou apenas operadores. Cada operador possui um nome e é parametrizado por variáveis. A parametrização do operador permite que este represente uma coleção de ações, onde uma ação é uma instância de um operador com todas variáveis atribuídas. A descrição do operador contem três listas, a primeira descreve quais cláusulas lógicas devem ser verdadeiras para que a ação possa ser aplicada, as últimas duas quais são os efeitos positivos e negativos da ação. Esta nova notação, baseada em uma linguagem para descrição de operadores, dá a STRIPS os mesmos benefícios que GPS teve em relação ao LT e uma solução para o *frame problem*. Os operadores, como as tabelas de diferença, permitem generalizar o software, já que pode-se descrever como computar novos estados durante a busca independente do código fonte. Além disso o *frame problema* é solucionado, já que por definição os operadores em STRIPS preservam todas as cláusulas intocados por uma ações (MCDERMOTT, 1995) (FIKES; NILSSON, 1971).

STRIPS foi um avanço em relação aos seus predecessores, no entanto STRIPS possui suas próprias falhas. A suposição de linearidade (*linear assumption*) adotada por STRIPS, onde assume-se que não há conflitos entre os objetivos, e que estes podem ser satisfeitos na ordem em que foram especificados é um destes problemas. A *linear assumption* foi descrita por Sussman (SUSSMAN, 1973) e detalhada por Tate(TATE, 1974), o seu uso resulta em planos mais extensos do que o necessário,

ou na impossibilidade de resolver problemas onde há interação entre as precondições dos objetivos.

Nets Of Action Hierarquies (NOAH) é um planejador capaz de gerar planos para um domínio onde há ações com interações destrutivas. No NOAH isso foi possível por causa da busca no espaço planos e do uso de uma nova heurística denominada *least-commitment*, onde a escolha da ordem entre as ações é adiada ao máximo, isso significa que para o NOAH ações podem ser executadas em paralelo e a ordem entre ações só é imposta ao plano quando há conflito de efeito entre estas.

2.2 PLANEJADORES E SISTEMAS FORMAIS

Diferentes sistemas formais podem ser utilizados para representar planejamento. Nesta seção serão apresentados seis formalismos: o modelo de transição restrito; representação baseada em lógica proposicional; representação baseada em lógica de primeira ordem; representação baseada em conjuntos; representação clássica; e representação baseada em variáveis de estado.

Alguns destes formalismos não são utilizados para o desenvolvimento de planejadores, como é o caso do modelo de transição e representações baseadas em lógicas. O primeiro é apresentado para introduzir planejamento de forma mais formal, onde espera-se que o leitor ganhe intuição quanto aos conceitos de estados, ações e plano. Os modelos baseados em lógica são apresentados pela sua relevância histórica, já que estes compõe as técnicas utilizadas por planejadores como LT e QA3. Além da sua importância histórica os formalismos baseados em lógica também são apresentados para fins de comparação, onde espera-se que o leitor compreenda a motivação das outras representações para solucionar o *frame problem*.

Para as três últimas representações também há uma tabela no apêndice B, onde estas são comparadas em função da sua complexidade computacional.

Todos os formalismos e técnicas discutidas nesta seção são baseadas na obra *Automated Planning: Theory and Practice* (GHALLAB; NAU; TRAVERSO, 2004).

2.2.1 Modelo de transição restrito

Um modelo de transição é uma tupla de quatro elementos $\Sigma = \langle E, A, O, \gamma \rangle$, neste modelo O representa eventos não determinísticos que podem ocorrer no ambiente. Eventos são úteis para modelar sistemas dinâmicos, desnecessário para problemas clássicos onde assumem-se um sistema estático. Portanto para este trabalho será adotado uma versão restrita do modelo de transição definido como:

Definição 1. *Modelo de transição restrito* Σ *é a tripla*

$$\Sigma = \langle E, A, \gamma \rangle$$

Onde E representa o conjunto finito de estados possíveis e A é conjunto finito de ações do sistema, por fim γ é função de transição de estados.

Definição 2. *A função de transição mapeia o estado resultado após a aplicação de uma ação*

$$\gamma : E \times A \mapsto E$$

As ações possuem condições e efeitos, as condições representam em quais circunstâncias a ação pode ser executada, os seus efeitos representam o que será alterado no mundo após a sua aplicação.

Definição 3. *Efeito de uma ação é igual ao estado resultado com os efeitos negativos removidos e efeitos positivos adicionados*

$$\gamma(e, a) = (e \cap efeito^{-}(a)) \cup efeito^{+}(a)$$

A função de transição pode ser invertida, onde ao invés de definir o estado obtido pela aplicação de uma ação tem-se o estado onde a ação pode ser aplicada.

Definição 4. *O inverso da função transição é*

$$\gamma^{-1}(e, a) = (e \setminus efeito^{+}(a) \cup efeito^{-}(a))$$

O modelo de transição restrito é suficiente para descrever o domínio de um problema de planejamento, para um que o plano possa ser gerado, além dos detalhes do sistema de transição é necessário a descrição de uma instância deste domínio.

Definição 5. *Instância de um problema de planejamento é uma tripla*

$$\mathcal{P} = \langle \Sigma, e_0, e_o \rangle$$

Onde Σ é o modelo de transição restrito, e_0 é o estado inicial dessa instância e e_o o estado objetivo que deve ser atingido pelo o plano.

As ações podem ser relevantes ou aplicáveis, onde as ações relevantes satisfazem algum objetivo pendente do plano atual e ações aplicáveis possuem todos as suas precondições satisfeitas.

Ações relevantes permitem a elaboração de planos de trás para frente, a partir dos objetivos de um plano.

Definição 6. *Ações relevantes possuem um ou mais efeitos que contribuem de maneira positiva para que o objetivo seja atingido.*

Enquanto ações aplicáveis permitem a elaboração da frente para trás, a partir do estado atual de um plano.

Definição 7. *Ações aplicáveis possuem alguma transição válida a partir do estado considerado.*

A função Γ descreve o conjunto de possíveis estados sucessores.

Definição 8. *O conjunto de estados sucessores é obtido pela aplicação de todas as ações aplicáveis a um determinado estado*

$$\Gamma(e) = \{\gamma(e, a) | a \in \textit{ações aplicáveis em } e\}$$

Estados sucessores são computados pela aplicação sucessiva de ações válidas, $\Gamma^2 = \Gamma(\Gamma(e))$, $\Gamma^3 = \Gamma(\Gamma(\Gamma(e)))$, a união dos estados sucessores formam o conjunto transitivo contendo todos os estados que podem ser atingidos pela aplicação de ações.

Definição 9. *O conjunto de estados acessíveis é o conjunto transitivo de estados sucessores*

$$\hat{\Gamma} = \Gamma(e) \cup \Gamma^1(e) \cup \cdots \cup \Gamma^n(e)$$

Como o problema clássico de planejamento possui número finito de estados o conjunto $\hat{\Gamma}$ possui ponto fixo. Um problema de planejamento pode ser resolvido quando o conjunto de estados acessíveis contém o estado objetivo.

Definição 10. *Um problema é solucionável quando o estado objetivo está contido em* $\hat{\Gamma}$.

Esta definição formal é suficiente para descrever o planejamento clássico, a seguir serão introduzidas representações para computar soluções.

2.2.2 Lógica proposicional

Planejamento pode ser modelado como um sistema de pergunta e resposta, como feito pelo **QA3** e seus antecessores. Um sistema de pergunta e resposta pode ser construído em cima de alguma lógica, como a lógica proposicional, também chamada de lógica sentencial.

Lógica proposicional é um sistema composto de três partes, uma linguagem que possui proposições ou sentenças, alguma definição semântica ou interpretação para os símbolos utilizados nas proposições, e um sistema de prova com regras de dedução.

Em lógica proposicional as sentenças são compostas por átomos simples ou compostos. Átomos simples são usualmente denotados por letras em caixa alta como P, Q, e $\neg R$, cada qual representa algum fato. Átomos compostos por sua vez utilizam átomos simples, conectores (**e** ($\wedge$), **ou** ($\vee$), **implicação** ($\rightarrow$)), parênteses e outros átomos compostos para compor sentenças mais complexas, e.g. $P \wedge \neg Q \rightarrow R$.

Sentenças com conector **implicação** são utilizadas para representar ações, portanto a ação *soltar* pode ser representada da seguinte forma:

$$\mathrm{T1} \wedge \mathrm{RC} \wedge \mathrm{RL} \rightarrow \mathrm{T2} \wedge \neg \mathrm{RC} \wedge \mathrm{CL}$$

Os átomos simples **RC**, **RL**, e **CL**, representam respectivamente os fatos de que o **robô** está segurando a **caixa**, o **robô** está no **local**, e a **caixa** está no **local**. Esta ação pode ser utilizada no instante $T1$, caso o *robô* tenha consigo a *caixa*. Após o uso desta ação o *robô* não estará mais segurando a caixa, a caixa estará solta na mesma localização que o robô no tempo $T2$.

Desta forma é possível descrever problemas de planejamento e utilizar técnicas de lógica proposicional para solucioná-los. Apesar de simples existem desvantagens no uso de lógica proposicional para planejar, a mais evidente é a verbosidade desta técnica, onde cada fato

precisa ser descrito com o seu próprio átomo e todas as combinações de ações possíveis precisam ser descritas em fórmulas lógicas, uma atividade tediosa para um humano e fácil de introduzir erros, estes erros podem tornar o problema insolucionável no melhor caso ou gerar planos inválidos no pior caso.

Outra deficiência desta representação de planejamento é a necessidade de descrever o tempo explicitamente e adicionar o equivalente ao *frame axioms* [1] em cada cláusula (não demonstrado no exemplo apresentado acima).

2.2.3 Lógica de primeira ordem

Uma alternativa a lógica proposicional é o uso de lógica de primeira ordem ou lógica de predicados, o principal benefício da lógica de primeira ordem em relação a lógica proposicional é sua expressividade. Lógica de primeira ordem estende a lógica proposicional com a adição de quantificadores, variáveis, predicados e funções. Permitindo representar os mesmo problemas que lógica proposicional, porém com menos cláusulas.

As fórmulas bem formadas em lógica de primeira ordem são mais complexas do que as fórmulas em lógica proposicional. Ao invés de átomos simples agora há termos formados por constantes (c, b), variáveis (x, y) ou predicados ($P(x,y)$), os termos por sua vez são utilizados para formar *fórmulas atômicas*, as fórmulas atômicas podem ser compostas em fórmulas pelo uso de conectores ($\wedge, \vee, \rightarrow$) e quantificadores ($\forall, \exists$).

Estes são alguns exemplos de fórmulas em lógica de primeira ordem:

$$caixa1$$
$$Sobre(caixa1, x, t)$$
$$\forall x, y, t\,(Sobre(x, y, t) \rightarrow \neg Sobre(y, x, t))$$

Com lógica de primeira ordem é possível utilizar quantificadores e variáveis para descrever fatos para coleções de objetos, reduzindo o tamanho da especificação em relação a representação proposicional. Essa

[1] Axiomas para descrever os estados que se mantem inalterados após a aplicação de uma ação

representação é portanto mais fácil de ser utilizada por usuários e reduz a chance de erro humano.

Lógica de primeira ordem possui benefícios de notação em relação a lógica sentencial, mas contém outras dificuldades para representar problemas, em especial é necessária atenção redobrada ao elaborar-se a descrição de problemas, para garantir que apenas interpretações válidas possam ser derivadas. Por exemplo, a fórmula $Sobre(x, robo1)$ pode ser resolvida com a substituição $x = robo1$, já que a fórmula não está limitando os valores possíveis para x, esta interpretação é logicamente correta, porém indesejável. Além das interpretações inesperadas, lógica de primeira ordem também precisa expressar explicitamente tempo e também é sucessível ao *frame problem*.

Apesar de lógica de primeira ordem permitir o uso de funções, em planejadores o seu uso é indesejado, já que funções e variáveis em lógica de primeira ordem fazem com que o espaço de possíveis estados seja infinito.

2.2.4 Representação baseada em conjuntos

A representação baseada em conjuntos depende do uso de *proposições* para representar estados, tal como a representação baseada em lógica proposicional. No entanto a definição de ações não é baseada em expressões lógicas. Na representação baseada em conjuntos ações são representadas por uma tripla $\langle condição(a), efeito^{-}(a), efeito^{+}(a)\rangle$. Além da representação das ações, também há um conjunto enumerável de possíveis estados S e uma função de transição de estados:

$$\gamma(s,a) = \begin{cases} s - efeito^{-}(a) \cup efeito^{+}(a) & \text{se } condição(a) \subset s \\ \text{indefinido} & \text{caso contrário} \end{cases}$$

A $condição(a)$ representa as precondições necessárias para que a ação possa ser aplicada, $efeito^{-}(a)$ e $efeito^{+}(p)$ são os efeitos resultantes da sua aplicação. Tanto as precondições quanto os efeitos são conjunto de proposições. A aplicação de uma ação é equivalente a alteração do conjunto de proposição no estado atual, removendo os estados em $efeito^{-}(a)$ e adicionado $efeito^{+}(a)$, mantendo os outros

estados intactos. Como consequência esta representação não é suscetível ao *frame problem*.

Ao passo em que há a vantagem de não precisar descrever os *frame axioms*, esta representação tal como a representação em lógica proposicional também sofre de problemas de expressividade, já que toda ação possível precisa ser descrita e ter a sua tupla de condições e efeitos definida.

2.2.5 Representação clássica

A representação clássica foi introduzida pelo planejador **STRIPS**, ao invés da definição direta das ações possíveis em um determinado problema, a representação clássica utiliza operadores. Operadores são definidos em tuplas de três elementos $\langle nome(o), condição(o), efeito(o)\rangle$, onde $nome(o)$ é uma expressão sintática na forma $op(x_1, x_2, ..., x_n)$, op é o nome do operador que deve ser único, e as variáveis definidas entre os parênteses permitem a parametrização do operador, utilizadas para definir os predicados em $condição(o)$ e $efeito(o)$. Efetivamente operadores são equivalentes à representação em primeira ordem, com quantificadores existenciais e variáveis. A representação de estados e a função de transição são consistentes com a representação baseada em conjuntos.

Esta representação resolve o *frame problem* da mesma forma que a representação baseada em conjuntos, com a vantagem de não ser necessário enumerar todas as possíveis ações.

Esta é a primeira representação a introduzir um nome para as ações, onde não é necessário um mecanismo para deduzir quais ações foram executadas a partir do plano gerado, como é o caso das representações baseadas em lógica. Também é a primeira representação a introduzir uma separação entre as ações e esquemas de ações, chamados nos parágrafos acima de operadores. Os operadores representam um conjunto de possíveis ações, um operador pode ter um número incompleto de suas variáveis definidas, nesse caso diz-se que a ação está parcialmente instanciada (*partially grounded*), uma ação é um operador completamente instanciado.

2.2.6 Representação de variáveis de estado

A representação de variáveis de estados difere das representações anteriores onde não são utilizados proposições ou predicados para representar estados, mas variáveis que podem ser parametrizadas e ter o seu valor alterado.

Antes de definir os operadores nesta representação é necessário introduzir os conceitos de variáveis de estados, variáveis de objeto, objetos, e relações.

As variáveis de estado são expressões sintáticas utilizadas para descrever estados. A expressão é composta por um nome, que identifica o estado, parametrizado por variáveis de objeto.

$$nome(o_1, o_2, ..., o_n) = v$$

Variáveis de objetos são variáveis que podem ter seu valor associado a algum objeto. Objetos possuem uma interpretação natural, o conceito de objeto é equivalente às constantes na representação de lógica de primeira ordem, objetos são descritos em um conjunto enumerável relativo a cada problema. Relações, tal como variáveis de estado, são expressões sintáticas, porém relações são rígidas e não podem ter seu valor alterado, por isso relações não possuem a igualdade em sua forma sintática. Na proposição $sobre(caixa2, estado1) = caixa1$ utiliza-se a variável de estado $estado1$ para indicar que o objeto $caixa2$ está sobre a $caixa1$ naquele instante.

Nesta representação operadores são tuplas de três elementos $\langle nome(o), condição(o), efeito(o)\rangle$. $nome(o)$ é uma expressão sintática, tal como a utilizada pela representação clássica, $condição(o)$ e $efeito(o)$ no entanto são diferentes. As condições são definidas em termos de relações e variáveis de estados. Os efeitos definem novos valores para as variáveis de estado, podendo ser negadas pelo uso de expressões como $sobre(caixa2, estado + 1) = nulo$. Efeitos não podem conter relações porque estas são invariantes do problema que não podem ser alteradas.

Esta representação não está sujeita ao *frame problem*, variáveis que não são descritas no efeito do operador são inalteradas. A vantagem desta representação está no fato de a negação ser definida implicitamente nos operadores, onde as variáveis de estado tem o seu novo valor descrito em apenas uma expressão, enquanto representações que dependem de

proposições precisam de duas expressões, uma para negar seu valor antigo e outra para afirmar o novo valor.

2.3 FORMAS DE SE PLANEJAR

A dificuldade do planejamento esta diretamente relacionada ao número de estados válidos de um problema. Considere o mundo dos containers como exemplo, neste domínio de problema um porto é organizado de forma autônoma, containers são movimentados entre locais por robôs, estes carregados e descarregados com o auxílio de guindastes. Um problema com apenas 1 porto, 3 robôs, 100 containers, 5 locais onde cada local com 1 guindaste e 3 pilhas de containers, possui $\mathbf{10^{277}}$ estados possíveis.

Por causa do grande número de estados possíveis é inviável enumerá-los, sendo necessário o uso de técnicas onde o espaço solução seja explorado de maneira gradual.

2.3.1 Busca

Busca é uma técnica para explorar os possíveis estados de um problema sem enumerá-los. O algoritmo 1 descreve a busca de forma genérica:

Algorithm 1: Busca

```
   input  : Estado inicial
   output : Uma solução ou erro
1  agenda ← {estado inicial};
2  while agenda não está vazia do
3  |  estado ← Proximo(agenda);
4  |  if Satisfaz(estado) then
5  |  |  return estado;
6  |  for novo estado in ExpandirEstado(estado) do
7  |  |  Incluir(agenda, novo estado);
8  return erro;
```

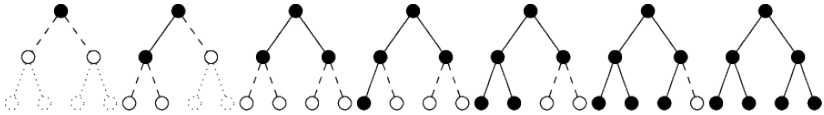

Figura 2 – Visualização de uma busca em largura

A busca é iniciada a partir de um estado conhecido, chamado de estado inicial (linha 1). A cada iteração verifica-se se o estado atual é uma solução para o problema (linha 4), se o estado for uma solução esta é retornada, do contrário zero ou mais estados são gerados e agendados para serem explorados (linha 6 e 7). A busca prossegue enquanto há estados para serem explorados e uma solução não foi encontrada (RUSSELL; NORVIG, 1995).

Algoritmos de buscas podem ser corretos, completos e ótimos. Uma busca correta nunca reportará um falso positivo, ou seja, a busca só para em falha ou com uma solução válida. Uma busca completa garante que caso existam soluções alguma destas será encontrada. Uma busca correta não é necessariamente completa, é possível que a busca sempre encontre soluções corretas mas nem sempre encontre soluções. Uma busca ótima garante que para problemas com mais de uma solução, a busca sempre irá informar a melhor delas, onde a definição de melhor varia de acordo com a busca. Em problemas de planejamento, especialmente planejamento clássico, o plano com o menor número de ações pode ser considerada a melhor solução.

O pseudo algoritmo 1 pode ser utilizado para explorar estados em camadas, expandindo sempre os estados mais recentes como representado pela figura 2, esta busca é denominada de busca em largura. Para que o pseudo algoritmo 1 seja equivalente a busca me largura é necessário utilizar uma fila (*first-in first-out*) como estrutura de dados de agenda. A busca em largura é correta desde que a expansão seja correta. Além de correta a busca em largura é ótima, já que os estados mais próximos do estado inicial são expandidos primeiro.

A desvantagem da busca em largura é o seu elevado uso de memória. Assumindo que o número de novos estados gerados a partir de cada estado conhecido, ou o *branch fractor* da busca, é igual a n, após a expansão de cada *camada* a agenda cresce por $n * n$ novos estados, após i camadas a agenda terá no pior caso n^i elementos, uma complexidade exponencial em termos de espaço (RUSSELL; NORVIG, 1995).

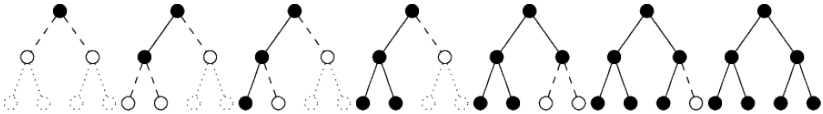

Figura 3 – Visualização de uma busca em profundidade

O mesmo pseudo algoritmo pode ser usado para explorar o espaço de busca em profundidade. Ao invés de expandir os estados em camadas, a busca em profundidade faz *sprints*, distanciado-se o mais rápido possível do estado inicial. Repare na figura 3 como a busca em profundidade expande o estado inferior esquerdo mais rápido do que na busca em largura, representada pela figura 2.

Para a busca em profundidade é necessário utilizar uma pilha (`first-in last-out`) como estrutura de dados para a agenda, priorizando os elementos mais recentes na busca. Como vantagem a busca em profundidade pode explorar caminhos mais longos com menos memória. Diferente da busca em largura que expande toda uma camada antes de iniciar a próxima, a busca em profundidade dá prioridade a estados mais recentes, portanto a cada iteração da busca n novos estados são adicionados a agenda **e** a busca progride para a próxima camada, resultando em um aumento de n elementos por camada e em uma complexidade de espaço igual a $n * i$ no pior caso (RUSSELL; NORVIG, 1995).

A vantagem da busca em profundidade no que diz respeito ao uso de memória não é, no entanto, livre de desvantagens. Por priorizar a expansão de estados mais recentes a busca é correta, desde que a expansão seja correta, mas não é completa nem ótima. A busca não é completa para espaços de busca infinitos ou que contenham ciclos. Em espaços infinitos a busca prosseguirá sempre expandido o estado mais recente, sem jamais expandir estados antigos, caso o caminho escolhido não contenha uma solução, a busca jamais terminará. Em espaços com ciclos a busca entrará em *loop*, e não haverá progresso. A busca em profundidade também não é ótima, caso uma solução seja encontrada, não há garantia que esta é de menor número de etapas, já que caminhos anteriores não foram explorados.

Uma forma de se unir as qualidades da busca em largura e profundidade, obtendo uma busca completa, ótima e eficiente no que diz respeito ao uso de memória, é a busca em profundidade interativa. Na busca em profundidade interativa utiliza-se uma pilha como estrutura para agenda, como na busca em profundidade, e como na busca em

largura os estados mais próximos do estado inicial são expandidos primeiro. Para isso a profundidade da busca é limitada, a busca é iniciada com limite l igual a 1, e todos os estados que estão a uma distância de l camadas são explorados, caso em l camadas uma solução não seja encontrada o limite é aumentado em 1 e a busca é reiniciada.

É importante notar que a busca em profundidade interativa também possui vantagens e desvantagens, a busca interativa precisa ser reiniciada após cada iteração, sempre começando a partir do estado inicial sem conhecimento de expansões anteriores, como resultado a busca interativa irá expandir os mesmos estados l vezes, o que leva a um aumento considerável no tempo de execução do algoritmo.

2.3.2 Busca com fronteira

A busca em profundidade não é completa para espaços que contenham ciclos. Isso ocorre porque a busca não tem conhecimento sobre estados que já foram visitados, adicionando à agenda estados que já foram explorados em uma iteração anterior.

Algorithm 2: Busca com fronteira

```
   input  : Estado inicial
   output : Uma solução ou erro
 1 agenda ← {estado inicial};
 2 vistados ← {};
 3 while agenda não está vazia do
 4 |   estado ← Proximo(agenda);
 5 |   Incluir(vistados, estado);
 6 |   if Satisfaz(estado) then
 7 |   |   return estado;
 8 |   for novo estado in ExpandirEstado(estado) do
 9 |   |   if novo estado não está em vistados then
10 |   |   |   Incluir(agenda, novo estado);
11 return erro;
```

A solução para este problema é a manutenção de uma estrutura de dados que contenha todos os estados que já foram visitados, evitando reexpandi-los. Exemplificado pelo algoritmo 2, a busca com fronteira faz

esse papel, onde utiliza-se um conjunto de estados visitados, também chamado de conjunto fechado, para verificar se o estado já foi explorado (linha 9). Note que para que este algoritmo funcione corretamente, é necessário que a estrutura de agenda não duplique valores (RUSSELL; NORVIG, 1995).

O uso de fronteira não é útil apenas para evitar ciclos na busca em profundidade, mas também como uma otimização para espaços que possuem mais de uma trajetória, i.e. busca em espaços que não são árvores, como grafos.

Restrições para o uso de busca com fronteira são o uso de memória, já que todos os estados visitados serão mantidos em memória, e a necessidade de definir igualdade para os elementos da busca.

2.3.3 Busca guiada A*

Explorar o espaço de soluções de forma gradual, como é feito pelas buscas, é melhor do que a enumeração de todos os estados possíveis, no entanto apenas a expansão gradual não é suficiente para solucionar problemas de forma eficiente. A eficiência de uma busca está diretamente relacionada ao número de estados explorados, quanto menos estados explorados para encontrar uma solução, mais eficiente a busca. Como um espaço de busca pode conter um número muito maior de estados não solução do que estados solução, se os estados forem explorados de forma cega, sem conhecimento sobre o espaço de busca, muito trabalho será gasto em estados que não levam a uma solução, fazendo com que a busca seja ineficiente.

O primeiro passo para evitar estados que não levam a uma solução é evitar estados impossíveis, para isso é necessário que a função de expansão seja correta. Na busca por provas de teoremas feita pelo *Logic Theorist*, os novos estados são cláusulas lógicas, portanto a expansão precisa gerar novas cláusulas válidas, enquanto para problemas de planejamento o objetivo é gerar planos válidos, portanto a expansão deve basear-se nas regras descritas pelo domínio do problema.

A expansão correta não é suficiente para que a busca seja eficiente, também é necessário diminuir o número de estados visitados. Para isso a expansão de estados não pode ser cega, é necessário diferenciar os possíveis estados e escolher as melhores trajetórias. A busca guiada

A*, descrita pelo algoritmo 3, faz esta escolha pelo uso de uma função heurística, que da nota a cada estado depois de expandido (linha 3 e 10). Os estados com maiores notas são tidos como mais favoráveis a levarem para uma solução, a nota é então usada para salvar o estado em ordem de prioridade na agenda (linha 4 e 11), isso permite que as trajetórias mais favoráveis sejam escolhidas a cada iteração (linha 6) (RUSSELL; NORVIG, 1995).

Uma função heurística nada mais é uma função que mapeia um estado para uma nota $h : E \to [0,1]$, a nota é um indicativo da qualidade daquele estado, e é utilizada para evitar a expansão de estado ruins. Como exemplo de heurística temos a análise de meios e fins realizada pelo planejador GPS, onde o planejador escolhe ações conforme estas contribuem para a solução.

Algorithm 3: Busca guiada A*

```
   input  : Estado inicial
   output : Uma solução ou erro
 1 agenda ← {};
 2 // estados são ordenados em função de suas notas
 3 nota ← Heurística(estado inicial);
 4 IncluirOrdenado(agenda, nota, estado inicial);
 5 while agenda não está vazia do
 6 |   estado ← ProximoOrdenado(agenda);
 7 |   if Satisfaz(estado) then
 8 |   |   return estado;
 9 |   for novo estado in ExpandirEstado(estado) do
10 |   |   nota ← Heurística(novo estado);
11 |   |   IncluirOrdenado(agenda, nota, novo estado);
12 return erro;
```

Busca guiada por heurística pode ser completa e ótima dependendo da heurística utilizada. Idealmente a função heurística guiaria a busca exatamente pela trajetória correta, levando diretamente ao melhor estado resultado, infelizmente tal heurística equivale a conhecer a solução para o problema e a busca não seria necessária, portanto para os problemas de interesse não há heurística perfeita. Para que a busca seja completa em espaços finitos é apenas necessário que a busca não descarte expansões que levam à solução, para espaços infinitos é necessário que eventualmente a busca seja guiada para uma trajetória

que inclua uma solução. Para que a heurística seja ótima é necessário que a nota de um estado nunca seja superestimada, do contrário a busca pode ser encerrada por expandir estados soluções não ótimos (RUSSELL; NORVIG, 1995).

2.3.4 Busca e planejamento

Para que a busca possa ser utilizada para resolver problemas de planejamento é necessário descrever as funções **ExpandirEstado** e **Satisfaz** junto com o estado inicial da busca. A função **ExpandirEstado** determina o espaço de busca, e a função **Satisfaz** como identificar se um estado é uma solução para o problema. As diferentes maneiras de se definir o espaço de busca serão discutidas a seguir.

2.3.5 Busca para frente

Planejamento pode ser solucionado com busca, utilizando o estado inicial do problema como estado inicial da busca. A função de expansão é igual ao conjunto de estados sucessores, descrito pela função $\Gamma(e)$. A função **Satisfaz** é baseada na representação utilizada pelo planejador.

Repare que essa busca baseia-se na transição de estados e na representação de mundo adotada pelo planejador, onde cada estado da busca é igual ao estado esperado do mundo, esta forma de planejamento denomina-se de busca no espaço de estados. Deve-se notar que neste estilo de busca os algoritmos 1 e 3 não retornariam um plano ao fim da busca, mas o estado objetivo, desta forma estes algoritmos precisam ser expandidos. A cada iteração além de se ter o estado atual da busca também é necessário ter o estado atual do plano, contendo todas as ações utilizadas até então.

O algoritmo 4 depende de pontos de escolha (linha 9). Pontos de escolha (*backtracking*) representam a execução não determinística do algoritmo, execuções estas que podem ser revisitadas. Um ponto de escolha é revisitado quando todas as suas subalternativas falharem, i.e. a busca chega a uma trajetória sem saída. Quando um ponto de escolha é revisitado o estado do programa é restaurado, limpando as alterações decorrentes das subexecuções, e a busca prossegue com outra alternativa.

Algorithm 4: Busca para frente

```
   input  : Operadores, estado inicial, estado objetivo
   output : Um plano ou erro
 1 estado ← estado inicial;
 2 plano ← plano vazio;
 3 while do
 4     if Satisfaz(estado, objetivo) then
 5         return plano;
 6     ações aplicáveis ← {a | a é ação aplicável em estado};
 7     if ações aplicáveis = ∅ then
 8         return erro;
 9     ação ← Ponto de escolha(ações aplicáveis);
10     estado ← γ(estado, ação);
11     plano ← plano + ação;
```

Durante uma busca existem vários pontos de escolha, quando um ponto de escolha não possuir mais alternativas a escolha anterior é revisitada.

É importante notar que a busca para frente opera sobre ações completamente instanciadas e não sobre operadores, portando para algumas representações é necessário instanciar os operadores. Como o algoritmo de busca para frente é uma variação da busca em profundidade, a busca não é ótima, no entanto a busca para frente é completa para problemas clássicos de planejamento, desde que a busca utilize uma estrutura de fronteira para evitar ciclos, já que a formulação clássica possui um número finito de estados possíveis.

2.3.6 Busca reversa

A busca pode ser iniciada pelo estado objetivo ao invés do estado inicial, progredindo de trás para frente. Na busca reversa, descrita no algoritmo 5, a expansão *não* pode ser feita com ações aplicáveis, já que durante a busca não se sabe quais requisitos estão satisfeitos, apenas quais efeitos são necessários, portanto na busca reversa as ações relevantes são utilizadas, a expansão é portanto equivalente a função $\hat{\Gamma}(e)$.

Existem nuances importantes na definição de ação relevante e

Algorithm 5: Busca reversa

```
input  : Operadores, estado inicial, estado objetivo
output : Um plano ou erro
plano ← plano vazio;
while do
    if Satisfaz(estado inicial, objetivo) then
        return plano;
    ações relevantes ← {a | a é relevante para objetivo};
    if ações relevantes = ∅ then
        return erro;
    ação ← Ponto de escolha(ações relevantes);
    objetivo ← γ⁻¹(objetivo, ação);
    plano ← ação + plano;
```

no teste de satisfação para que a busca reversa seja correta. Para que um plano seja considerado correto é necessário que as ações sempre tenham seus requisitos satisfeitos, caso contrário a ação não poderá ser executada. A cada iteração da busca reversa uma nova ação é adicionada, esta ação irá satisfazer algum objetivo pendente e adicionar os seus requisitos como novos objetivos, isto é necessário para garantir que todas as ações do plano tenham os seus requisitos satisfeitos. A busca reversa é considerada completa quando o estado inicial satisfazer todos os objetivos daquela iteração da busca. A busca reversa, tal como a busca para frente, é completa para o planejamento clássico com o uso de fronteira.

2.3.7 Busca reversa com ações parciais

A eficiência de uma busca é diretamente relacionada ao número de estados expandidos a cada iteração, seu *branch factor*. Na busca para frente o *branch factor* é igual ao tamanho do conjunto de ações aplicáveis, enquanto para a busca reversa é igual ao tamanho do conjunto de ações relevantes. A busca reversa pode ser alterada para utilizar ações parcialmente instanciadas, a fim de diminuir o *branch factor* da busca, onde apenas os efeitos necessários para satisfazer algum objetivo são instanciados.

O algoritmo 6 descreve a busca reversa com ações parcialmente instanciadas, neste algoritmo são escolhidos operadores relevantes ao invés de ações relevantes (linha 6). Como a representação dos operadores utiliza variáveis, cada operador também é acompanhado de uma substituição (linha 7), onde a substituição define o valor das variáveis para que o(s) efeito(s) do operador possa(m) ser unificado(s) com o objetivo que está sendo satisfeito.

Algorithm 6: Busca reversa com ações parciais

```
   input  : Operadores, estado inicial, estado objetivo
   output : Um plano ou erro
 1 plano ← plano vazio;
 2 while do
 3     if Satisfaz(estado inicial, objetivo) then
 4         return plano;
 5     substituições ← {
 6         (op, σ) | op é um operador relevante em estado,
 7         σ é a substituição que unifica o operador com estado
 8     };
 9     if substituições = ∅ then
10         return erro;
11     (operador, σ) ← Ponto de escolha(substituições);
12     objetivo ← γ^{-1}(σ(objetivo), σ(ação));
13     plano ← σ(operador) + σ(plano);
```

É importante notar que nem todas as representações possuem operadores, em especial a representação baseada em estados e as representações baseadas em lógica não podem ser utilizadas com esta forma de busca.

2.3.7.1 STRIPS

Uma das primeiras tentativas para reduzir o *branch factor* da busca em problemas de planejamento foi o algoritmo 7, desenvolvido para o planejador STRIPS. Esta é uma busca reversa, utilizando a representação baseada operadores (inventada para este planejador), com duas diferenças em relação ao algoritmo 5 (busca reversa). Primeiro, a busca satisfaz as precondições de um operador por vez (linha 9). Segundo,

o planejador compromete-se em utilizar o primeiro operador que tiver suas precondições satisfeitas, adicionando o subplano necessário para satisfazer as precondições do operador (linha 12), seguido do operador que pôde ser satisfeito (linha 13), na versão final do plano (linha 14).

Algorithm 7: Algoritmo STRIPS

```
   input  : A definição do problema, com os operadores, estado
            atual e objetivos
   output : Um plano ou erro
 1 while do
 2     if Satisfaz(estado, objetivos) then
 3         return Plano;
 4     ações ← AcoesRelevantes(operadores,estado,objetivos);
 5     if not ações then
 6         return erro;
 7     ação ← Ponto de escolha(ações);
 8     pendências ← Precondicoes(ação);
 9     subplano ← Strips(operadores, estado, pendências);
10     if not subplano then
11         return erro;
12     estado ← Transicao(estado, subplano);
13     estado ← Transicao(estado, ação);
14     Plano ← Plano + subplano + ação;
```

Essa estratégia adotada pelo STRIPS elimina uma grande porção do espaço de busca, tornando a busca mais eficiente, no entanto o algoritmo não é ótimo nem completo. A heurística adotada por STRIPS é a *linear assumption*, descrita por Sussman, onde assume-se que os objetivos podem ser satisfeitos na ordem de definição, i.e. não há conflito entre os pré-requisitos de cada objetivo.

Como exemplo da limitação da *linear assumption* vamos utilizar a *Sussman anomaly*, esta anomalia demonstra como a heurística não é ótima. A anomalia é descrita em termos do mundo dos blocos, um problema similar a torre de hanoi. O mundo dos blocos possui uma mesa e alguns blocos, os blocos podem estar empilhados ou postos sozinhos na mesa, a única ação possível é mover um bloco do topo de uma pilha à outra. Considere que um bloco sozinho é também uma pilha, portanto um bloco pode ser retirado do topo de uma pilha e posto sobre a mesa.

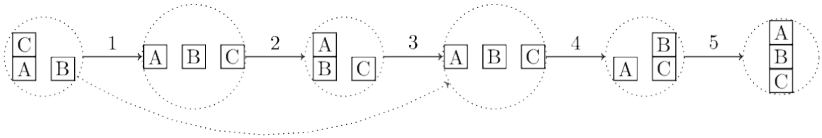

Figura 4 – Sussman anomaly

A anomalia de Sussman, ilustrada na imagem 4, utiliza uma instância com apenas três blocos. Nesta instâncias os blocos são nomeados A, B e C. O bloco C está sobre o bloco A que está sobre a mesa, e bloco B está sobre a mesa. O objetivo é empilhá-los em ordem alfabética.

A anomalia ocorre quando o objetivo de empilhar o bloco A sobre o bloco B é definido antes do objetivo do bloco B sobre o C. O plano gerado irá satisfazer os objetivos em ordem, para isso o bloco C é movido para a mesa, passo 1, liberando o bloco A para ser movido, então o bloco A é movido sobre o bloco B, passo 2, o primeiro objetivo foi satisfeito. Com o bloco A sobre o B é hora de satisfazer o segundo objetivo, o bloco B sobre o C. O passo 3 limpa o bloco B, permitindo que este seja movido, no passo 4 bloco B é posto sobre o bloco C, satisfazendo o segundo objetivo. No entanto o planejador não percebe o conflito entre os objetivos, onde o passo 3 desfaz o passo 2, portanto o primeiro objetivo precisa ser satisfeito novamente, e o passo 5 é adicionado ao plano.

A estratégia adotada por Sussman no seu planejador HACKER para solucionar conflitos de pré-requisitos foi chamada de *protection*, onde as ações para solucionar um objetivo são marcadas, caso novas ações adicionadas ao plano ameacem um objetivo estratégias de correção de conflitos são utilizadas, como por exemplo a inversão das cláusulas de objetivo (SUSSMAN, 1973).

2.3.8 Busca no espaço de planos

O modelo de busca no espaço de estados foi uma evolução natural dos planejadores. De certa forma esta evolução foi guiada pelo trabalhos em cálculo situacional de McCarthy e o desenvolvimento inicial em *problem solvers* como o GPS. No entanto essa não é a única forma de se resolver problemas de planejamento, busca no espaço de

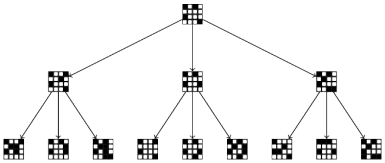

Figura 5 – Busca no espaço de estados

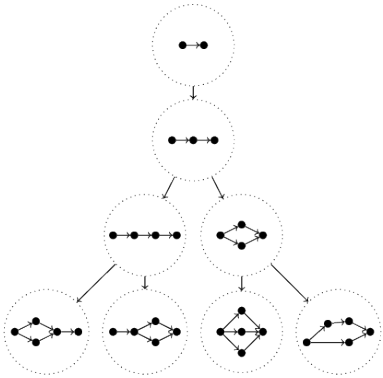

Figura 6 – Busca no espaço de planos

planos é uma alternativa.

A diferença entre planejamento no espaço de estados em relação ao espaço de planos se dá principalmente pelas estruturas de dados utilizadas durante a busca. A primeira busca, baseada no espaço de estados, é realizada por alterações graduais as proposições que compõem o estado esperado do mundo, esta forma de busca está representada visualmente pela figura 5, nesta figura cada nó no grafo representa um estado do mundo, e cada transição a aplicação de uma nova ação. As alterações são realizadas até que um estado que contenha todas as cláusulas do objetivo seja encontrado, o plano é igual as transições do estado estado inicial ao final, i.e. o caminho percorrido pelo grafo de estados do problema.

Planejamento no espaço de planos é essencialmente diferente, como pode ser notado na figura 6. O planejador inicia a busca a partir de um plano incompleto, contendo apenas ações que equivalem aos estados inicial e final, e é gradualmente refinado. Nesta representação cada nó no grafo representa um plano válido porém incompleto, cada transição representa um refinamento igual a adição de uma nova ação, a solução para o problema é completamente contida no último nó do grafo de busca.

Planos gerados pela busca no espaço de estados possuem implicitamente uma ordem total entre suas ações, isso ocorre porque a única transição válida para a busca no espaço de estados é a adição de

uma nova ação, onde necessariamente a nova ação será executada após todas as ações que já fazem parte do plano, quando uma solução for encontrada o plano já terá uma ordem total definida. Planejamento no espaço de planos é mais geral neste aspecto, como a busca possui uma representação completa do plano parcial é possível controlar a ordem das ações, isto permite representar diferentes ordens de execução serial ou paralela de ações.

Por definição a busca no espaço de planos parciais não terá uma ordem total, no entanto nem sempre é possível gerar planos apenas pela agremiação de ações, já que ações são utilizadas para satisfazer pré-requisitos pendentes e ações podem interferir destrutivamente. A causalidade é utilizada para ordenar uma ação que é pré-requisito de outra, onde necessariamente uma ação deve ser executada antes de outra. A relação de causalidade é também utilizada para resolver interações destrutivas entre ações.

A relação de causalidade ações não deve ser adicionada de maneira irrestrita, caso contrário o plano será reduzido para um de ordem total, por isso é comum adotar-se a técnica de *least commitment*, onde as restrições de ordem são adicionadas apenas quando estritamente necessário, para remover conflitos e garantir que o plano seja válido (WELD, 1994).

A busca no espaço de planos pode ser exemplificada pelo algoritmo genérico denominado *Plan Space Planning*. Nesta forma de planejar o objetivo é codificado como uma ação, esta ação não possui efeitos, apenas pré-requisitos, e ela é necessariamente a última ação do plano, além da ação objetivo também é adiciona uma ação para o estado inicial, que possui apenas efeitos e nenhum pré-requisito, o plano apenas com as ações inicial e final, onde a ação inicial acontece antes da ação final, é o ponto de partida da busca.

A busca *PSP* irá então selecionar uma falha no plano atual, como um pré-requisito pendente ou alguma interação destrutiva entre ações. Caso haja mais de uma estratégia para solucionar a falha no plano uma destas estratégias é escolhida de maneira não determinística, a busca continua até que uma solução seja encontrada, um plano que não contenha falhas, ou não haja nenhuma nova possibilidade de refinamento, onde não há solução para o problema.

Isto conclui a apresentação dos principais algoritmos utilizados para solucionar problemas clássicos de planejamento, dentre os algoritmos apresentados nota-se que os planos encontrados pela busca no

Algorithm 8: Algoritmo PSP

```
   input  : plano parcial
   output : plano refinado ou erro
 1 falhas ← Objetivos(plano) + Ameaças(plano);
 2 if not falhas then
 3 |  return plano;
 4 falha ← EscolherFalha(falhas);
 5 resolvedores ← ResolvedoresValidos(falha);
 6 if not resolvedores then
 7 |  return erro;
 8 resolvedor ← Ponto de escolha(resolvedores);
 9 resolução ← ResolverFalha(resolvedor, falha, plano);
10 plano ← Refinar(resolução, plano);
11 plano ← PSP(plano);
12 return plano;
```

espaço de planos é a de maior benefício, por isso o planejador *PNKE* realizará a busca neste espaço.

3 DESENVOLVIMENTO

Para este trabalho foi desenvolvido um planejador automático denominado *PNKE*. Um planejador independente de domínio, que elabora planos parcialmente ordenados para problemas clássicos de planejamento, utilizando a representação baseada em variáveis de estado.

Como descrito nas seções anteriores os espaços de busca para problemas de planejamento são imensos, onde mesmo problemas com poucos elementos (ações, objetos, predicados, etc.) compõem espaços de busca grandes o suficiente para tornar a enumeração impossível, portanto é necessário o uso de técnicas de busca.

A seguir serão descritos os detalhes de implementação da busca no *PNKE*, baseada no algoritmo genérico *PSP*, tal como as razões para as escolhas feitas durante o projeto. Os detalhes de implementação discutidos incluem: a técnica utilizada para implementar pontos de escolha; a representação em memória para os estados da busca; os algoritmos utilizados para resolver falhas.

3.1 PONTOS DE ESCOLHA

A primeira observação importante é que o algoritmo *PSP* depende do uso de pontos de escolha, uma funcionalidade que não é oferecida como parte de um grande número de linguagens de programação.

A busca com pontos de escolha pode ser facilmente simulada pelo uso de recursão, descrita pelo algoritmo 9. A busca é iniciada pela expansão do estado atual em novos estados válidos (linha 1), então um destes é escolhido (linha 6) para continuar com a busca em uma chamada recursiva (linha 7), se todos os estados da sub-chamada falharem um erro é retornado (linha 8). O erro é utilizado para informar a chamada pai de que uma nova escolha deve ser feita, permitindo revistar escolhas anteriores tal como no algoritmo *PSP*. Se ao tratar a falha houverem estados para escolha, um destes é utilizado para continuar com a busca (linha 3), do contrário o erro é propagado para as chamadas anteriores (linha 8).

Algorithm 9: Ponto de escolha com recursão

```
   input  : Estado atual, funções Expandir e Sucesso
   output : Uma solução ou erro
1  opções ← Expandir(estado atual);
2  resultado ← erro;
3  while opções não estiver vazio do
4  |  if Sucesso(resultado) then
5  |  |  return resultado;
6  |  escolha ← Proximo(opções);
7  |  resultado ← busca(escolha);
8  return erro;
```

Recursão é uma solução elegante para pontos de escolha, porém o seu uso resulta em pressão sobre a *stack* do processo. Como a *stack* é um recurso limitado que quando esgotado leva a falha do software, o seu uso é indesejado. Para evitar problemas de exaustão da *stack* é necessário o uso de alguma técnica alternativa.

Em linguagens com avaliação preguiçosa (*lazy evaluation*) pode-se utilizar estruturas preguiçosas para modelar pontos de escolha [1] (WADLER, 1985). Para linguagens com semântica ansiosa pode-se utilizar funções geradoras ou corrotinas [2].

PNKE foi desenvolvido utilizando a linguagem Python, uma linguagem com avaliação ansiosa, portanto as opções são funções geradas ou corrotinas. Corrotinas foram utilizadas para representar pontos de escolha porque há suporte nativo da linguagem, o que permite utilizar o algoritmo 1 para a busca, onde cada elemento adicionado à agenda possui uma corrotina para gerar os novos estados da busca durante a etapa de expansão.

[1] Linguagens com avaliação preguiçosa oferecem uma abstração sobre a ideia de funções geradoras (BLOSS; HUDAK; YOUNG, 1988). Em linguagens preguiçosas é possível descrever todos pontos de escolha em uma lista e deferir a sua avaliação apenas para quando houver *backtracking*.

[2] Funções geradoras e corrotinas extraem o estado necessário para o ponto de escolha da stack principal do programa, no primeiro caso utilizando estruturas salvas no *heap*, no segundo a execução é movida para *light weight threads*, também alocadas no *heap*.

3.2 REPRESENTAÇÃO DE PLANOS

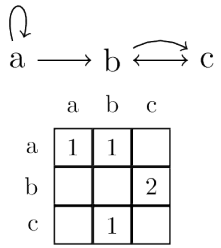

Figura 7 – Representação matricial de um grafo

Tendo uma técnica para implementar pontos de escolha é necessário uma representação dos estados da busca. Como a busca realizada pelo *PNKE* é no espaço de planos parciais, que são grafos, uma representação de grafos é necessária. A representação comum baseada em objetos nó, onde cada objeto possui referências aos nós filhos, sofre de baixa localidade de memória [3] (DREPPER, 2007), por isso uma representação disposta de forma contígua em memória é mais vantajosa.

Grafos podem ser representados utilizando matrizes, onde estas são dispostas de forma contígua em memória. Nesta representação cada nó do grafo possui uma linha **e** uma coluna, i.e. cada linha e coluna da matriz é etiquetada pelos nós do grafo. As arestas partem do elemento representado pela linha para o elemento representado pela coluna, o valor de cada célula representa o número de arestas que conecta estes nós, como demonstrado pela figura 7.

Planos são grafos direcionados acíclicos, onde arestas não seguem na direção inversa e nós não podem ter auto referências, portanto apenas uma matriz triangular é suficiente para representar planos. Como as arestas utilizadas pelos planos representam ordem entre as ações, não há a necessidade para mais de uma aresta, dessa forma cada célula só precisa ter um bit de tamanho, setado para verdadeiro caso as ações estejam conectadas (KNUTH, 2011).

A representação baseada em matriz é uma melhoria em relação ao uso de objetos nós, no que diz respeito a localidade dos dados em

[3] Localidade de memória refere-se a disposição dos dados na memória principal, onde há benefício de performance ao manter-se os valores em uma mesma *cacheline* ou em endereços de memória consecutivos.

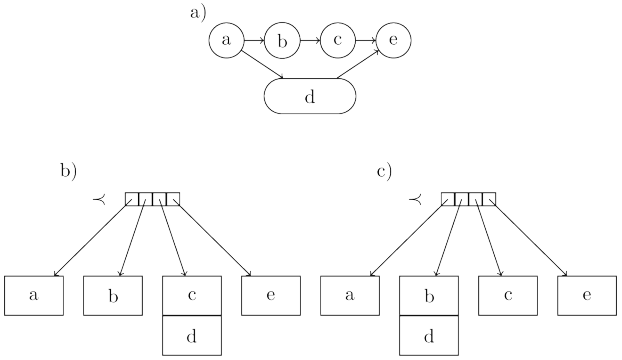

Figura 8 – Representação esparsa de um plano parcial

memória, no entanto uma matriz triangular representa todas as possíveis arestas em um grafo, o que não é ótimo para grafos esparsos, resultando em uso excessivo memória. Como planos são grafos esparsos, para reduzir o desperdício de memória uma representação baseada em matriz esparsa foi utilizada.

Na representação esparsa adotada, representada pela figura 8, nós são agrupados em camadas. Em contraste com uma matriz triangular que mantém a ordem entre todos os nós do grafo, esta representação mantém ordem apenas entre **camadas**. A ordem é representada por uma lista, que mantém relação "ocorre antes de", onde os elementos na posição p ocorrem antes de alguma das camadas sucessoras.

Para exemplificar a invariante da estrutura repare na figura 8 o nó **d**, este nó não está ordenado em relação aos nós **b** e **c**, e portanto poderia ser adicionado tanto a camada que contém **b** ou **c**, preservando em ambos os casos as relações de ordem presentes no grafo original.

Essa representação não é fiel ao grafo original, já que nós com maior liberdade são forçados em camadas que com relações mais restritivas, mas esta estrutura é suficiente para representar a relação de dependências entre ações, necessárias para manter a consistência de um plano durante a busca.

3.2.1 Otimizações

Cada iteração do processo de busca irá alterar minimamente o plano parcial que está sendo expandido, seguindo a ideia de *least commitment*. Cada etapa da busca irá adicionar uma nova ação ou ordem entre as ações existentes, mantendo boa parte do plano anterior intacto.

Como consequência a representação esparsa pode ser utilizada como uma estrutura de dados persistente. Onde a estrutura nunca é limpa ou sobrescrita, seu valores são preservados e compartilhados com os novos planos. Isso é possível pela uso de cópia rasa (*shallow copy*), onde as ações que se mantiveram intactas entre os planos são compartilhadas como pode ser observado na figura 9.

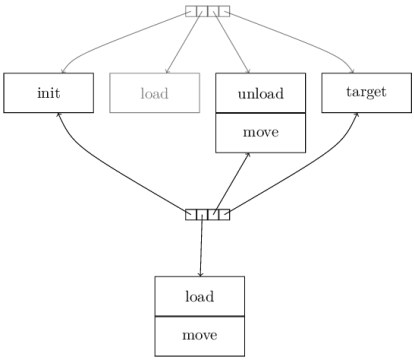

Figura 9 – Plano baseado em cópia rasa das ações do plano anterior

Além do uso de cópias rasas, o processo de busca também é um bom candidato para o uso de técnicas de programação dinâmica, onde o objetivo é não recomputar resultados, como é o caso dos efeitos de operadores, precondições e efeitos das ações no plano atual. Isso foi feito no *PNKE* pelo uso de índices, utilizados para manter a lista de ações com precondições pendentes e dos seus efeitos, evitando a necessidade de percorrer por todos os nós do grafo para identificar falhas no plano.

3.3 BUSCA

O processo de busca no espaço de planos pode ser feito em profundidade, largura ou guiado por heurísticas. Para garantir que a solução ótima seja encontrada foi adotado a busca em largura, onde o espaço de planos é explorado gradualmente dando prioridade para os planos com o menor número de ações. Em situações de empate, que são comuns, planos com o maior número de precondições satisfeitas tem prioridade.

Para representar domínios e instâncias de problemas pode-se utilizar alguma linguagem de descrição de ações, como as linguagens *ADL*, *STRIPS* ou *PDDL*. Dentre estas opções, *PDDL* é a mais recente, desenvolvida por McDermott para ser utilizada na competição internacional de planejamento automático, portanto é a melhor candidata dentre as opções para ser utilizada. *PDDL* é descrita em mais detalhes no apêndice A. No entanto, como uma maneira de simplificar o projeto, no momento apenas declarações diretas na linguagem hospedeira são possíveis. Esta representação é a mesma utilizada pelo planejador durante a busca, portanto baseada na representação de variáveis de estado.

A busca no espaço de planos pode ser realizada com ações totalmente ou parcialmente instanciadas. O benefício da busca com ações parcialmente instanciadas é a redução do espaço de busca, com o benefício de retardar o comprometimento com determinadas substituições. Esta redução no espaço de busca vem ao custo de uma maior complexidade de código, já que a escolha dos objetos para instanciar as ações passa a ser um problema de satisfação de restrições (*constraint satisfaction problem*).

O uso de variáveis de estado foi motivado pela representação implícita de efeitos negativos, o que não só simplifica a descrição de operadores como também a implementação do planejador. A simplificação na implementação ocorre porque apenas um tipo na linguagem de programação hospedeira é suficiente para representar as variáveis de estado, em contraste com as representações baseadas em predicados onde é necessário o uso de *tags* para diferenciar os predicados positivos dos negativos [4].

[4] *Tags* refere-se ou ao uso de diferentes tipos ou de uma variável booleana para diferenciar predicados positivos dos negativos.

A busca no *PNKE* evita a adição de ações conflitantes, portanto não é necessário durante a etapa de expansão tratar explicitamente a existência de ameaças ao plano. Isso reduz as falhas tratadas pelo algoritmo PSP aos objetivos pendentes, que é igual a lista de precondições não satisfeitas, indexadas durante a busca.

Os objetivos são satisfeitos de trás para frente, dando preferência para objetivos pendentes em ações mais tardias do plano, a intenção é acelerar a ocorrência de conflitos nas etapas iniciais da busca, reduzindo assim o espaço de busca.

Algorithm 10: Busca PNKE

```
   input  : Definição do problema
   output : Uma solução ou erro
 1 plano ← plano parcial(problema);
 2 agenda ← {};
 3 IncluirOrdenado(agenda, Heurística(plano), plano);
 4 while agenda não está vazia do
 5 |   plano ← ProximoOrdenado(agenda);
 6 |   if Satisfaz(plano, problema) then
 7 |   |   return plano;
 8 |   for pendente in precondições(plano) do
 9 |   |   for novo plano in substituição(plano) do
10 |   |   |   nota ← Heurística(novo plano);
11 |   |   |   IncluirOrdenado(agenda, nota, novo plano);
12 |   |   for novo plano in nova ação(plano, pendente) do
13 |   |   |   nota ← Heurística(novo plano);
14 |   |   |   IncluirOrdenado(agenda, nota, novo plano);
15 return erro;
```

A busca do *PNKE*, descrita no algoritmo 10, utiliza dois resolvedores para satisfazer as precondições pendentes. O primeiro resolvedor utiliza substituição para satisfazer precondições pendentes, onde uma ação relevante, já presente no plano parcial, é unificada com a ação pendente (linha 9). O segundo resolvedor insere uma nova ação parcialmente instanciada para satisfazer uma precondição do plano (linha 12).

Ambos resolvedores podem gerar zero ou mais novos planos, que são em seguida adicionados a agenda. A falha em gerar um novo

plano é equivalente a uma trajetória sem saída, onde o caminho da busca é abandonado. Planos na agenda são expandidos quando conveniente, onde conveniente é definido pela função heurística (linhas 10 e 13), a busca termina em sucesso quando todas ações têm suas precondições satisfeitas (linha 6), ou em falha quando não houverem mais planos para serem expandidos (linha 15).

3.3.1 Resolvedor nova ação

O resolvedor de nova ação, algoritmo 11, satisfaz apenas uma precondição por vez, com algum operador que satisfaça uma precondição pendente (linha 1). Cada operador relevante tem seu predicado [5] relevante igualado ao predicado pendente (linha 2), determinando algumas de suas variáveis de objeto, as outras variáveis objetivos pertinentes ao operador são instanciadas como novas variáveis, únicas para o novo plano, que serão determinadas posteriormente pelo resolvedor de unificação.

Tendo a nova ação instanciada é necessário verificar se há algum conflito entre os efeitos desta e o plano atual (linha 3), caso não existam conflitos a ação parcialmente instanciada é adicionada ao plano e os índices são atualizados. A atualização de índices inclui adicionar os efeitos da nova ação para serem utilizados pelo resolvedor de substituição, adicionado o novo pré-requisito na lista de precondições necessárias e adicionado todas os pré-requisitos da nova ação como insatisfeitos, plano parcial pode ser gerado, com a nova ação adicionada (linha 6).

[5] Tecnicamente o planejador utiliza variáveis de estado ao invés de predicados. O termo predicado neste capítulo é utilizado de maneira não formal.

Algorithm 11: Resolvedor por nova ação

```
input  : plano, ação pendente e precondição
output : Novos planos parciais satisfazendo precondição
1 for operador in operadores(plano, precondição) do
2     nova ← instanciar(operador, precondição);
3     if não conflita(nova, plano) then
4         novo plano ← adicionar(nova, plano);
5         atualizar índices(novo plano);
6         gerar novo plano;
```

A operação de adicionar ações precisa controlar três detalhes importantes para integridade do plano: A nova ação deve ser adicionada na camada que antecede a ação pendente; Ações não podem ser adicionadas na primeira camada do plano, já que nenhuma ação pode ser paralela a ação inicial do plano; Ações conflitantes não podem ser adicionadas na mesma camada, nestes casos uma nova camada deve ser criada apenas para a nova ação;

3.3.2 Resolvedor substituição

O resolvedor de substituição, algoritmo 12, é aplicado individualmente a cada ação com uma ou mais precondições pendentes. O resolvedor primeiro encontrada ações relevantes (linha 1), e tenta unificar a ação relevante com a ação pendente (linha 2). Caso as duas ações possam ser unificadas, verifica-se se a nova substituição de variáveis não produz nenhum conflito no plano atual (linha 3), caso não haja conflito um novo plano é construído. Além de unificar as variáveis com o plano (linha 4), os índices devem ser atualizados para remover a precondição recém satisfeita da lista de precondições pendentes e adicionar o novo efeito na lista de efeitos necessários.

É importante notar que após a unificação entre as ações pendente e relevante, a consistência de todo o plano é verificada. Isso é necessário porque a unificação de variáveis não é local para estas duas ações, já que uma mesma variável pode ser utilizada por outras ações levando a um efeito em cascata de atualização.

A unificação é necessária porque a busca feita pelo *PNKE* utiliza

Algorithm 12: Resolvedor por substituição

```
input  : plano, ação pendente e precondição não satisfeita
output : Novos planos parciais satisfazendo precondição
1 for relevante in ações relevantes(plano, precondição) do
2     σ ← unificar(relevante, pendente);
3     if σ ≠ ∅ não conflito(σ, plano) then
4         novo plano ← substituir (σ, plano);
5         atualizar indices (precondição, novo plano);
6         gerar novo plano;
```

ações parcialmente instanciadas. Para a busca com ações totalmente instanciadas, como todos os objetos são conhecidos, apenas um teste de igualdade estrutural entre os predicados é suficiente.

Como a busca é realizada com ações parcialmente instanciadas, os predicados de ambas ações, da ação relevante e da pendente, podem ter em seus valores variáveis ou objetos, o processo de unificação só será bem sucedido caso a unificação destes valores não conflitem.

Considerando o predicado $Sobre(o) = r$, onde o e r são objetos, e o predicado indica que objeto o está sobre o objeto r. O efeito e precondição conflitam quando: (a) Diferentes objetos forem utilizados para a mesma variável de objeto; (b) O mesmo objeto for utilizado como valor para mais de uma variável de objeto; (c) A mesma variável for utilizada como valor para mais de uma variável de objeto. Os três conflitos estão exemplificados a seguir:

$$\sigma(Sobre(caixa1) = \texttt{v1}, Sobre(caixa2) = \texttt{v2}) = \emptyset \quad \text{(a)}$$

$$\sigma(Sobre(caixa1) = \texttt{v1}, Sobre(\texttt{v2}) = caixa1) = \emptyset \quad \text{(b)}$$

$$\sigma(Sobre(\texttt{v1}) = \texttt{v2}, Sobre(\texttt{v3}) = \texttt{v1}) = \emptyset \quad \text{(c)}$$

No caso (a), a unificação não é valida porque existe um conflito com os objetos utilizados. O predicado a esquerda possui a substituição $o \leftarrow caixa1$, o predicado a direita a substituição $o \leftarrow caixa2$. Como as duas substituições utilizam objetos diferentes para a mesma variável de objeto, estas não podem ser unificadas.

No caso (b), a unificação não é valida porque o resultado

reutilizaria o mesmo objeto para duas variáveis de objeto. Como o predicado a esquerda já possui a substituição $o \leftarrow caixa1$, e o predicado a direita a substituição $r \leftarrow caixa1$, o resultado da unificação seria o predicado inválido $Sobre(caixa1) = caixa1$.

No caso (c), a unificação não é valida por uma razão semelhante ao caso (b), onde utiliza-se uma variável duas vezes, em contraste ao uso de um mesmo objeto. Como o predicado a esquerda utiliza a substituição $o \leftarrow v1$ e o predicado a direita $r \leftarrow v1$, o resultado da substituição seria o predicado inválido $Sobre(v1) = v1$, como qualquer objeto atribuído a variável $v1$ será inválido, esta substituição também é inválida.

Se não houver conflito, as novas substituições podem ser aplicadas e um novo plano refinado é gerado.

3.4 EXEMPLO

Os algoritmos e estrutura de dados descritos são suficientes para construir o planejador *PNKE*, aqui vamos exemplificar um pedaço da busca realizada pelo planejador e alguns dos planos encontrados durante ela.

Como exemplo será utilizado o mundo dos containers. Este problema possui apenas três ações:

1. A ação **move** permite que o robô movimente-se. O robô desloca-se de sua localização inicial para uma localização destino, tanto a precondição quanto o efeito são descritos pelo predicado *at*.

2. A ação **unload** permite que um robô carregado seja descarregado. A ação tem como precondição que o robô esteja carregado, descrito pelo predicado *loaded*. O container é descarregado na mesma localização em que o robô. Como efeito o predicado *at*, que descreve a localização do robô, é utilizado para setar o predicado *in*, que descreve a posição do container. Note que o robô pode carregar apenas um container por vez.

3. A ação **load** permite carregar um container no robô. O container pode ser carregado no robô desde que ambos estejam na mesma localização, descrito pelo predicado *at* para o robô e *in* para o container. O fato do robô estar carregado é descrito pelo predicado **loaded**, que é atribuído como valor o container está sobre o robô.

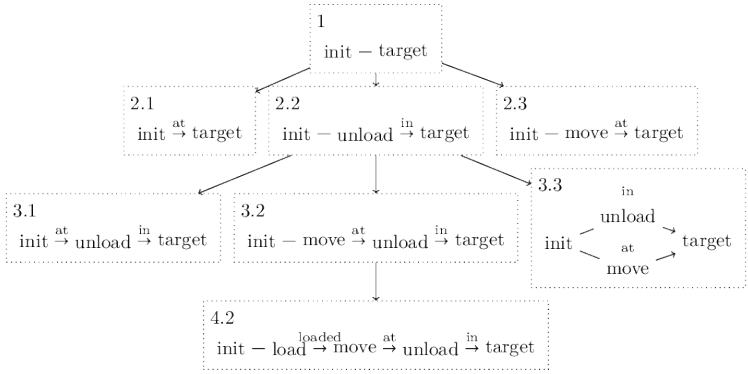

Figura 10 – Busca no espaço de planos para o mundo dos containers

O problema exemplo tem como objetivo posicionar tanto o container quanto o robô na localização **l2**, sendo que no estado inicial do mundo o container está na posição **l1** e o robô na localização **l2**. Alguns dos nós explorados pelo *PNKE* estão representados na figura 10.

A busca é iniciada por um plano parcial, formado apenas por duas ações que codificam o estado inicial e final como duas ações, representados pelo passo **1**.

A partir do estado inicial a busca prossegue aplicando as substituições possíveis, como é o caso dos planos **2.1** e **3.1**. Em **2.1** é feita uma substituição para satisfazer a precondição **at** utilizando a ação inicial, esta substituição é então garantida durante a busca pela adição de uma restrição aos índices de busca, impedindo que o predicado seja quebrado. A restrição no entanto impede o uso da ação **move** em qualquer subplano, como consequência o robô **r** é mantido na localização **l2** para todos os subplanos daquela trajetória de busca, impedindo que o robô possa primeiro mover-se para a localização **l1** para carregar o container **c2**, levando este trajeto da busca a um caminho sem saída.

A segunda estratégia adota pelo *PNKE* é a adição de novas ações, note que as ações são parcialmente instanciadas e adicionadas para satisfazer apenas uma precondição por vez. A ação **unload** foi adicionada ao plano **2.2** para satisfazer o requisito **in**, o planejador não atribui valor a nenhuma variável além das variáveis presentes no

predicado satisfeito, como resultado a ação possui apenas algumas de suas variáveis definidas, as variáveis que ficarem pendentes serão definidas por uma nova iteração da busca utilizando o resolvedor de substituição.

Ações podem ser adicionadas em uma mesma camada, permitindo a descrição de planos parciais, como é exemplificado pelo plano **3.3** com as ações **unload** e **move**.

Figura 11 – Solução para o problema exemplo

O plano **4.2** representa o caminho de busca que eventualmente levará a solução, representada pela figura 11. As ações serão adicionadas a este plano e gradualmente concretizadas pelos dois resolvedores, o plano solução é encontrado pela busca após utilizar estes resolvedores aproximadamente 81 mil vezes.

A busca *PNKE* apresentada é capaz de refinar planos e construir soluções a problemas de planejamento independente de domínio.

4 CONCLUSÃO

Os objetivos do trabalho são: O desenvolvimento de um planejador para o problema clássico do planejamento; Os planos elaborados pelo planejador devem ser de ordem parcial. O resultado deste trabalho foi o planejador *PNKE*.

Dois domínios de problema foram solucionados com *PNKE*: Problemas do mundo dos blocos, incluindo a anomalia de Sussman baseada no mundo dos containers; E uma instância de problema para o mundo de robôs. As declarações de ambos domínios e problemas estão no anexo A.

4.1 ANÁLISE DO MUNDO DOS BLOCOS

Para analisar a eficiência do planejador foi desenvolvido um gerador de problemas para o mundo dos blocos, o código do gerador está no anexo B. O gerador foi utilizado para elaborar problemas de maneira exaustiva, primeiro todas combinações possíveis com 3 blocos foram geradas e solucionadas, em seguida instâncias com 4 blocos. Os problemas de cada grupo possuem o mesmo número de objetos e predicados tanto para o estado inicial quanto o final, a diferença está contida na configuração destes estados.

Problemas que tiveram em sua agenda mais de 1 milhão de entradas foram abortados, o uso excessivo de memória para esses cenários levava a falha da busca. 1 milhão de elementos no computador de teste é aproximadamente 5GB de memória.

Todos os 576 problemas com 3 blocos puderam ser solucionados. É interessante notar a grande variação no número de planos refinados, independente do número de objetos e predicados para os estados inicial e final. O problema mais simples foi resolvido após o refinamento de 3 planos, enquanto o problema mais complexado refinou 88137 planos antes de encontrar uma solução. A distribuição no número de refinamentos necessários está representada pelo histograma a direita na imagem 13. O gráfico de dispersão a esquerda da imagem 13 demostra a evolução no tempo de execução em função do número de planos expandidos.

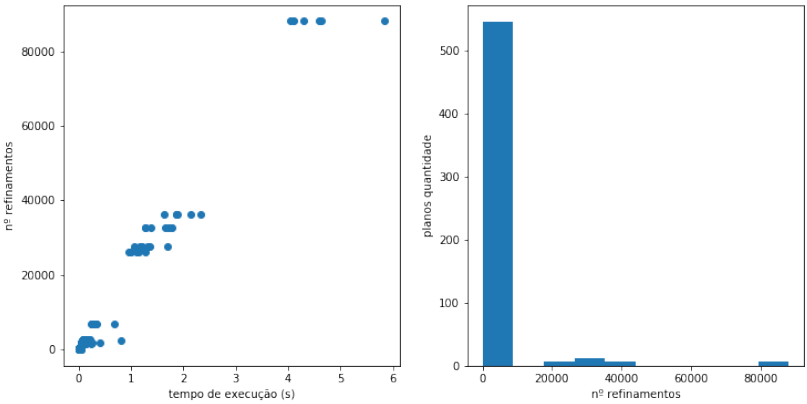

Figura 12 – Histograma 3-blocos

Após a remoção de 94 *outliers* superiores (8.16% dos cenários), que precisaram de mais de 210 ms para completar a busca, nota-se que a relação entre o número de planos expandidos e o tempo de execução não é linear. Indicando uma possível diferença de performance entre as duas estratégias de resolução.

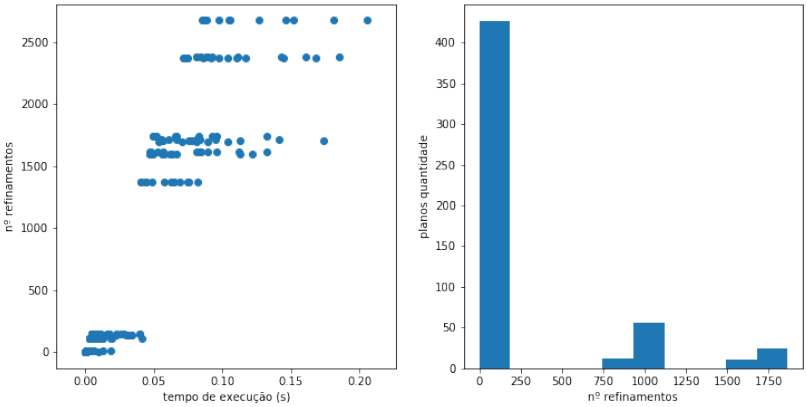

Figura 13 – Histograma 3-blocos sem *outliers*

Dos 36864 possíveis problemas com 4 blocos, 1348 foram experimentados durante 2 horas e meia de execução. Destes 1348 experimentos, 1233 foram solucionados e 115 foram abortados pelo critério de número

excessivo de planos na agenda, a imagem 14 sumariza os experimentos. Para os problemas com 4 blocos o menor plano foi encontrado após 4 expansões, enquanto para o grupo de experimentos realizados o maior teve 727013 expansões.

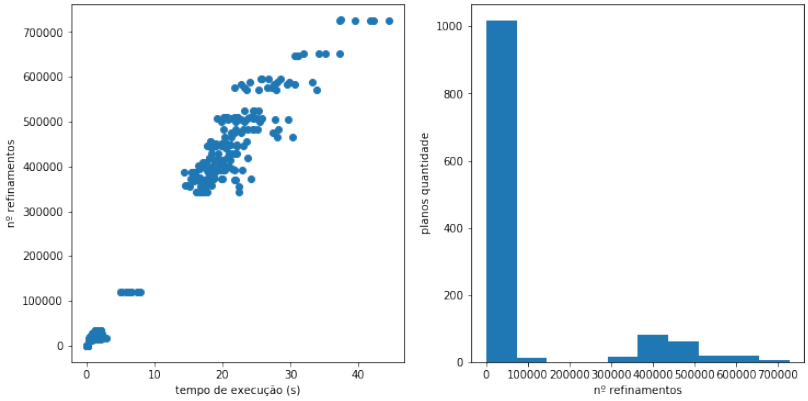

Figura 14 – Histograma 4-blocos

4.2 MELHORIAS

Estas são algumas das potenciais melhorias que podem ser aplicadas no planejador para melhorar a sua eficiência e diminuir o espaço de busca explorado.

4.2.1 Poda da busca

O processo de busca pode resultar no refinamento de um mesmo plano por mais de uma trajetória, como exemplo imagine uma ação que possui dois pré requisitos e estes podem ser sanados por duas ações, numa primeira fase de iterações o planejador irá gerar dois planos refinados, onde cada pré requisito é sanado individualmente, durante algum momento da segunda fase de iteração as mesmas ações serão adicionadas, resultando em planos idênticos, porém as ações foram refinadas na ordem reversa.

É aparente que ambos planos são iguais, e que explorar um destes deveria invalidar o outro, para que isso possa ser feito é necessário definir alguma noção de igualdade entre planos, permitindo identificar planos idênticos e podar a busca nas instâncias repetidas.

Na busca com ações parcialmente instanciadas o teste de igualdade se torna um pouco mais difícil, especialmente quando as ações parcialmente instanciadas possuem variáveis distintas. Para que a igualdade possa ser definida é necessário a normalização das ações e das variáveis. Um possível processo é a normalização de ações paralelas, mantendo-as ordenadas em suas respectivas camadas, como o uso de ordem topológica em função de seus nomes, e normalizar os nomes da variáveis do plano, utilizando nomes crescentes a partir de um dos extremos do plano.

4.2.2 Decidibilidade da busca com ações parcialmente instanciadas

Busca no espaço de estados possui uma grande vantagem em relação a busca no espaço de planos, durante cada iteração de uma busca no espaço de estados o planejador tem conhecimento sobre todo o estado do mundo, o valor de todos os predicados são conhecidos. Com esta informação torna-se trivial a detectar repetições durante a busca com o uso de conjuntos de fronteira e podar trajetórias repetidas de busca, tornando o processo de busca decidível.

Na busca no espaço de planos com ações parcialmente instanciadas não é tão trivial a detecção destas repetições, fazendo com que o processo de busca utilizado seja parcialmente decidível. Entre as alternativas estão a busca no espaço de plano com ações totalmente instanciadas, onde é possível deduzir o estado do mundo equivalente ao plano parcial durante cada iteração da busca, mesmo que o estado do mundo deduzido seja incompleto é possível detectar repetições e abortar a busca em trajetórias que já foram exploradas. Idealmente pode-se expandir a busca com ações parcialmente instanciadas para detectar repetições e evitar o aumento do *branch factor* da busca resultado do uso de ações instanciadas.

4.2.3 Estruturas persistentes

Como a busca refina gradualmente o plano parcial, fazendo pequenas alterações a cada iteração, é interessante reutilizar as estruturas alocadas anteriormente, evitando uma cópia completa do plano parcial que está sendo refinado. Isso pode ser feito com o uso de cópias rasas e alterações locais, apenas as alterações necessárias para o refinamento da busca.

Apesar do uso de cópia rasas ser o suficiente para evitar a alocação de todo um novo plano parcial, é possível reduzir a quantidade de elementos copiadas com o uso de estruturas persistentes eficientes, implementadas usando árvores, reduzindo o número de cópias de n para log_2 elementos, dependendo da estrutura utilizada.

4.2.4 Ações conflitantes

A busca no espaço de planos é necessariamente baseada na suposição de que existem ações independentes, caso contrário não seria possível a execução de ações em paralelo, no entanto essa suposição leva a busca a considerar ações que são conflitantes em uma mesma camada, especialmente problemático porque a busca com ações parcialmente instanciadas não pode identificar conflitos enquanto a ação não for completamente instanciada.

Identificar dependências entre ações como uma etapa de anterior a busca pode ser utilizado como uma heurística para guiar a busca, evitando a expansão de planos utilizando ações conflitantes.

APÊNDICE A – Planejamento clássico

O mundo real possui incertezas inerentes e agentes inteligentes autônomos precisam de mecanismos para lidar com elas, para o planejamento as incertezas de interesse envolvem a possibilidade de ações falharem durante a sua execução, por causa de interações inesperadas entre agentes e a possibilidade de algo fora do controle ocorrer, e.g. como mudanças nas condições do tempo. O tratamento de incertezas pode ser construído em um agente e um planejador de diversas formas, planos podem prever falhas e conter estratégias para mitigá-las, o agente e o planejador podem ter um *feedback loop* para que os planos sejam refinados e/ou corrigidos conforme falhas sejam detectadas, ou o agente pode replanejar a cada falha que ocorra no sistema em que ele está inserido (GHALLAB; NAU; TRAVERSO, 2004).

O planejamento onde há um *feedback loop* fechado entre o controlador do agente e o planejador é chamado de planejamento *online* (figura 15), como alternativa ao planejamento *online* existe o planejamento *offline*, onde ao invés do planejador ser informado sobre os eventos observados, o controlador utiliza-se do planejador para gerar novos planos conforme necessário (GHALLAB; NAU; TRAVERSO, 2004).

O planejamento *offline* (figura 16) é uma simplificação do modelo *online*, já que não é necessário que o planejador mantenha o presente estado de execução nem que haja a representação de incertezas. O planejamento clássico é uma das formulações possíveis para o planejamento *offline*, que também pode ser chamado de *STRIPS planning* em referência ao planejador histórico. Para o planejamento clássico, as seguintes características devem ser satisfeitas (RUSSELL; NORVIG, 1995):

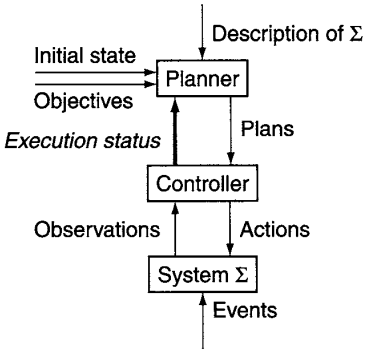

Figura 15 – Modelo conceitual de *feedback loop online*

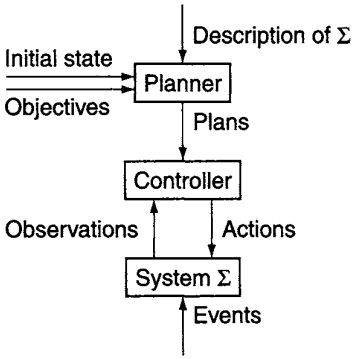

Figura 16 – Modelo conceitual de *feedback loop offline*

1. **Mundo finito**, assume-se que o mundo é composto por um número finito de estados.

2. **Observável**, assume-se que o mundo é completamente observável ao invés de parcialmente observável ou não observável.

3. **Determinístico**, assume-se que todas as ações executadas pelo controlador serão bem sucedidas, quando falhas ocorrerem pode-se criar um novo plano levando a situação atual do mundo em consideração.

4. **Estático**, a única força alterado do mundo é o agente.

5. **Definição simples de objetivo**, não há como definir preferências entre as ações que serão utilizadas em um plano ou otimização de planos.

6. **Planos sequenciais**, um plano é uma sequência linear de ações.

7. **Tempo implícito**, durante a execução do plano assume-se que todas as ações são executadas instantaneamente.

8. **Planejamento offline**, o planejador tem tempo o suficiente para gerar o plano.

APÊNDICE B – Decidibilidade e complexidade do planejamento

Planejamento clássico é um problema decidível, independente da representação utilizada, já que o espaço de planos é finito e uma solução pode ser encontrada pela enumeração de todos os planos.

Tanto a representação clássica quanto a representação baseada em variáveis de estados são semi-decidíveis quando estendidas para suportar funções, já que o espaço de planos passa a ser infinito. Ambas representações estendidas são decidíveis sobre a formulação do problema onde admite-se apenas planos com até $\boldsymbol{n}$ passos (EROL; NAU; SUBRAHMANIAN, 1992).

A complexidade do planejamento varia em função da liberdade para declaração das ações, problemas que restringem ações para que não haja precondições ou efeitos negativos pertencem a diferentes classes de complexidade. A complexidade pode ser sumarizadas utilizando os seguintes procedimentos de decisão (EROL; NAU; SUBRAHMANIAN, 1995):

1. **plano(D)**, Onde $\boldsymbol{D}$ é a definição de um problema de planejamento, este procedimento retorna **sim** caso haja uma solução para o domínio $\boldsymbol{D}$.

2. **plano-n-passos(D)**. $\boldsymbol{D}$ é novamente a definição de um problema de planejamento, este procedimento retorna **sim** caso haja uma solução para $\boldsymbol{D}$ com até $\boldsymbol{n}$ passos.

Considerando estes dois procedimentos de decisão, a complexidade do problema de planejamento está sumarizada na tabela 1. Repare que a tabela está dividida em seções, onde cada uma detalha a complexidade de um tipo de representação.

As colunas “Efeitos negativos” e “Condições negativas” referem-se a restrições na formulação do problema que podem alterar a sua complexidade. “Efeitos negativos” refere-se as formulações que permitem ou não ações com efeitos negativos. “Condições negativas” refere-se as formulações que permitem ou não que ações que dependem de predicados negativos.

Tabela 1 – Complexidade do planejamento

Representação baseada em conjuntos

Operadores definidos	Efeitos Negativos	Condições Negativas	Complexidade *plano*	Complexidade *plano-n-passos*
Durante execução	Sim	Sim/Não	PSPACE completo	PSPACE completo
	Não	Sim	NP completo	NP completo
	Não	Não	P	NP completo
	Não	Não[ac]	NLOGSPACE completo	NP completo
Pré processado	Sim/Não	Sim/Não	Linear	Linear

Representação clássica

Operadores definidos	Efeitos Negativos	Condições Negativas	Complexidade *plano*	Complexidade *plano-n-passos*
Durante execução	Sim	Sim/Não	EXPSPACE completo	NEXPTIME completo
	Não	Sim	NEXPTIME completo	NEXPTIME completo
	Não	Não	EXPTIME completo	NEXPTIME completo
	Não	Não[a]	PSPACE completo	PSPACE completo
Pré processado	Sim	Sim/Não	PSPACE[b]	PSPACE[b]
	Não	Sim	NP[b]	NP[b]
	Não	Não	P	NP[b]
	Não	Não[a]	NLOGSPACE	NP

Representação variáveis de estado

Operadores definidos	Efeitos Negativos	Condições Negativas	Complexidade *plano*	Complexidade *plano-n-passos*
Durante execução	Sim[d]	Sim/Não	EXPPSPACE completo	NEXPTIME completo
Pré processado	Sim[d]	Sim/Não	PSPACE[b]	PSPACE[b]

[a] Domínios onde as ações possuem no máximo 1 requisito

[b] Onde alguns conjuntos de operadores são PSPACE / NP completo

[c] Onde ações com mais de um requisito são compostas por outras ações

[d] A negação na representação de variáveis de estado é implícita e não há como restringí-la.

APÊNDICE C – Linguagens de descrição de ações

Planejamento genérico e independente de domínio possui por definição uma separação entre quem utiliza e quem desenvolve o planejador. Desta maneira é necessário que o usuário do planejador possa descrever os problemas para o planejador, sem que tenha conhecimento específico sobre o funcionamento de um planejador, tal como é necessário que o planejador não tenha suposições sobre o domínio em que este será utilizado. Para isso são necessárias linguagens de descrição de ações, como *STRIPS* e *PDDL*.

Para que um planejador possa raciocinar sobre o sistema em que o agente está inserido é necessária a descrição do ambiente em um formato compreensível pelo planejador. Esta descrição contém tanto os operadores, que são moldes para descrever as ações que o agente pode executar, quanto uma descrição suficiente do ambiente em que o agente está inserido, para que o planejador saiba quais ações podem ou não serem executadas. Idealmente a descrição do sistema contém informação suficiente para que o planejador possa gerar planos otimizados para critérios como tempo de execução ou gasto de recursos.

A descrição dos operadores que agentes podem executar, denomina-se domínio do problema. A descrição do ambiente do problema em que este agente está inserido e quais objetivos o agente deve satisfazer, denomina-se uma instância do problema. Abaixo há um exemplo de um esquema de ação (*action schema*) utilizando a linguagem de descrição *STRIPS*:

```
MOVE(x, z, y)
     PRE: CLEAR(x), ON(x, y), CLEAR(z)
     ADD: CLEAR(y), ON(x, z)
     DEL: CLEAR(z), ON(x, y)
```

Esta linguagem é composta pela descrição de *schemas* de operadores, no exemplo 'MOVE'. Este operador é parametrizado por três variáveis (x, y e z), todas sempre presentes no cabeçalho do operador. O corpo do *schema* é composto por três partes: primeiro as precondições para que o operador possa ser utilizada; segundo os seus efeitos positivos; terceiro os efeitos negativos. O uso de um operador denomina-se uma ação, e cada predicado que possui uma variável um denomina-se fluente (KAUTZ; MCALLESTER; SELMAN, 1996).

STRIPS é um exemplo de linguagem para descrição de problemas de planejamento, outras linguagens também foram criadas com

semânticas distintas como *ADL* e *Task Formalism* (SACERDOTI, 1975). Cada linguagem de descrição de ações permitem construtores com semânticas distintas, como átomos negativos, expansão de *schemas*, etc. É importante notar que historicamente houve uma sobreposição entre a sintaxe das linguagens de descrições e a sua semântica, essa realidade só foi alterada e uma sintaxe padronizada foi adotada no campo com a introdução de *PDDL* em 1998.

Planning Domain Definition Language (*PDDL*) foi desenvolvida por Drew McDermott em um esforço para padronizar as linguagens de descrição de ações, *PDDL* foi criada para permitir a realização da competição internacional de planejamento (*IPL*), de forma que uma grande gama de planejadores com diferentes requisitos possam ler uma mesma definição de domínio e instâncias de problemas, permitindo uma comparação mais direta de performance e resultado entre os planejadores (GHALLAB DREW MCDERMOTT, 1998).

Como *PDDL* pretende ser uma língua franca para definição de problemas ela precisa englobar uma variedade de requisitos que podem ocorrer em diversos problemas, em função da diversidade de problemas e estratégicas adotadas pelos planejadores a linguagem possui um recurso chamado de *:requirements*, que permite ao usuário definir quais os requisitos mínimos do planejador para que ele possa compreender a definição do problema, no total a versão 3.1 da linguagem possui 21 subconjuntos de funcionalidades que podem ser utilizadas para definição de problemas.

PDDL é uma linguagem para descrição de ações, ela é um super conjunto que pode representar problemas de outras linguagens de ação como ADL, e o formalismo presente nos planejadores SIPE-2, UMCP, Unpop e UCPOP. A linguagem possui o conceito de requisitos para ter a versalidade de poder definir ações para vários problemas de planejamento, de forma que a linguagem PDDL é suficiente como uma especificação sintática, de um linguagem para descrição de ações e estados, com ela é possível descrever o domínio do problema e instâncias do problema de planejamento.

Um exemplo de domínio de problema descrito de *PDDL* é o mundo dos blocos, exemplificado abaixo:

Código C.1 – Exemplo de domínio PDDL blocks world

```
(define
  (domain blocks-world-domain)
  (:requirements
        :strips
        :equality
        :conditional-effects)
  (:constants Table)
  (:predicates (on ?x ?y) (clear ?x) (block ?b))
  (:action puton
         :parameters (?X ?Y ?Z)
         :precondition
         (and
                 (on ?X ?Z) (clear ?X) (clear ?Y)
                 (not (= ?Y ?Z))
                 (not (= ?X ?Z))
                 (not (= ?X ?Y))
                 (not (= ?X Table)))
         :effect
         (and (on ?X ?Y) (not (on ?X ?Z))
           (when
               (not (= ?Z Table))
               (clear ?Z))
           (when
               (not (= ?Y Table))
               (not (clear ?Y))))))
```

Este domínio possui apenas uma ação, denominada *puton*, equivalente a ação *MOVE* descrição em *STIPS* anteriormente. Tendo a definição do domínio de problema, pode-se definir um problema, como a anomalia de Sussman descrita em *PDDL*:

Código C.2 – Exemplo de problema PDDL Sussman anomaly

```
(define
  (problem sussman-anomaly)
  (:domain blocks-world-domain)
  (:objects A B C)
  (:init
      (block A) (block B) (block C) (block Table)
      (on C A) (on A Table) (on B Table)
      (clear C) (clear B))
  (:goal (and (on B C) (on A B)))
  (:length (:serial 3) (:parallel 3)))
```

ANEXO A – Descrição dos problemas resolvidos pelo PNKE

Como descrito no capítulo de desenvolvimento, *PNKE* não utiliza nenhuma linguagem de descrição de ações. Neste anexo estão dispostos os códigos utilizados para declarar os domínios e instâncias de problemas utilizados para testar o planejador. As descrições são baseadas na linguagem do planejador, Python.

A função `operator_new` é utilizada para instanciar um novo operador, os argumentos correspondem em ordem: o nome do operador; o nome das variáveis utilizadas para parametrizar o operador; os tipos dos objetos que podem ser atribuídos a cada uma destas variáveis; uma lista de relações rígidas; uma lista de precondições; uma lista de efeitos.

A função `expression_new` é utilizada para instanciar expressões, uma estrutura de dados utilizada para representar variáveis de estados. Os argumentos correspondem em ordem: o nome da variável de estado; o tipo da expressão, podendo ser uma expressão de igualdade ou uma expressão nula; as variáveis utilizadas para parametrizar a variável de estado, é obrigatório que estas também estejam declaradas no operador; os tipos dos objetos que podem ser atribuídos a cada uma destas variáveis, é obrigatório que os tipos sejam iguais aos declarados no operador;

A função `action_new` é utilizada para instanciar uma ação, como o estado inicial da busca é um plano incompleto com duas ações que equivalem ao estado inicial e final, é necessário utilizar esta função para descrever a instância do problema. Os argumentos correspondem em ordem: O operador que descreve a ação; as substituições que estão instanciadas, i.e. o objeto que está atribuído a variável de objeto; todas as substituições, incluindo as as variáveis de objeto que o seu valor é uma variável.

A função `domain_new` é utilizada para instanciar a estrutura de dados que corresponde ao domínio do problema.

A função `pnke_plan` corresponde ao ponto de entrada principal do planejador, que realiza a busca.

Esta é a descrição do domínio para o problema *blocksworld*. O mundo que foi utilizado para apresentar a anomalia de Sussman:

```
def blocksworld():
    moveop = operator_new(
        'move',
        ['?a', '?b'],
```

```
        ['block', 'block'],
        [],
        [
            expression_new(
                'on',
                NIL,
                ('?a', ),
                ('block', 'block'),
            ),
            expression_new(
                'on',
                NIL,
                ('?b', ),
                ('block', 'block'),
            ),
        ],
        [
            expression_new(
                'on',
                EQ,
                ('?a', '?b'),
                ('block', 'block'),
            ),
        ],
    )

    newpileop = operator_new(
        'newpile',
        ['?a', '?b'],
        ['block', 'block'],
        [],
        [
            expression_new(
                'on',
                EQ,
                ('?a', '?b'),
                ('block', 'block'),
            ),
            expression_new(
                'on',
                NIL,
```

```
                ('?b', ),
                ('block', 'block'),
            ),
        ],
        [
            expression_new(
                'on',
                NIL,
                ('?a', ),
                ('block', 'block'),
            ),
        ],
    )

    return domain_new([moveop, newpileop])
```

Esta é a descrição da anomalia de Sussman:

```
def sussman_anomaly():
    init_op = operator_new(
        'init',
        ['?a', '?b', '?c'],
        ['block', 'block', 'block'],
        [],
        [],
        [
            expression_new(
                'on',
                EQ,
                ('?a', '?c'),
                ('block', 'block'),
            ),
            expression_new(
                'on',
                NIL,
                ('?c', ),
                ('block', 'block'),
            ),
            expression_new(
                'on',
                NIL,
```

```
                ('?b', ),
                ('block', 'block'),
            ),
        ],
    )

    target_op = operator_new(
        'target',
        ['?a', '?b', '?c'],
        ['block', 'block', 'block'],
        [],
        [
            expression_new(
                'on',
                NIL,
                ('?a', ),
                ('block', 'block'),
            ),
            expression_new(
                'on',
                EQ,
                ('?b', '?a'),
                ('block', 'block'),
            ),
            expression_new(
                'on',
                EQ,
                ('?c', '?b'),
                ('block', 'block'),
            ),
        ],
        [],
    )

    init_action = action_new(
        init_op,
        {'?a': 'a', '?b': 'b', '?c': 'c'},
        {},
    )

    target_action = action_new(
```

```
        target_op,
        {'?a': 'a', '?b': 'b', '?c': 'c'},
        {},
    )

    return pnke_plan(
        blocksworld(),
        init_action,
        target_action,
    )
```

Esta é a descrição de domínio de problema para o mundo dos containers:

```
def dwn_simplified():
    moveop = operator_new(
        'move',
        ['?r', '?s', '?e'],
        ['robot', 'location', 'location'],
        [
            Relation(
                'adjacent',
                ('?s', '?e'),
                ('location', 'location'),
            ),
        ],
        [
            expression_new(
                'at',
                EQ,
                ('?r', '?s'),
                ('robot', 'location'),
            ),
        ],
        [
            expression_new(
                'at',
                EQ,
                ('?r', '?e'),
                ('robot', 'location'),
            ),
```

```
        ],
    )

    loadop = operator_new(
        'load',
        ['?c', '?r', '?l'],
        ['container', 'robot', 'location'],
        [],
        [
            expression_new(
                'at',
                EQ,
                ('?r', '?l'),
                ('robot', 'location'),
            ),
            expression_new(
                'in',
                EQ,
                ('?c', '?l'),
                ('container', 'location'),
            ),
            expression_new(
                'loaded',
                NIL,
                ('?r', ),
                ('robot', 'container'),
            ),
        ],
        [
            expression_new(
                'in',
                NIL,
                ('?c',),
                ('container', 'location'),
            ),
            expression_new(
                'loaded',
                EQ,
                ('?r', '?c'),
                ('robot', 'container'),
            ),
```

```
    ],
)

unloadop = operator_new(
    'unload',
    ['?c', '?r', '?l'],
    ['container', 'robot', 'location'],
    [],
    [
        expression_new(
            'at',
            EQ,
            ('?r', '?l'),
            ('robot', 'location'),
        ),
        expression_new(
            'loaded',
            EQ,
            ('?r', '?c'),
            ('robot', 'container'),
        ),
    ],
    [
        expression_new(
            'loaded',
            NIL,
            ('?r',),
            ('robot', 'container'),
        ),
        expression_new(
            'in',
            EQ,
            ('?c', '?l'),
            ('container', 'location'),
        ),
    ],
)

return domain_new([moveop, loadop, unloadop])
```

Uma instância de problema para o mundo dos containers:

```
    def dwn_problem1():
        init_op = operator_new(
            'init',
            ['?c', '?r', '?l', '?m'],
            ['container', 'robot', 'location', 'location'],
            [],
            [],
            [
                expression_new(
                    'in',
                    EQ,
                    ('?c', '?l'),
                    ('container', 'location'),
                ),
                expression_new(
                    'at',
                    EQ,
                    ('?r', '?m'),
                    ('robot', 'location'),
                ),
                expression_new(
                    'loaded',
                    NIL,
                    ('?r', ),
                    ('robot', 'container'),
                ),
            ],
        )

        target_op = operator_new(
            'target',
            ['?c', '?r', '?l'],
            ['container', 'robot', 'location'],
            [],
            [
                expression_new(
                    'in',
                    EQ,
                    ('?c', '?l'),
                    ('container', 'location'),
                ),
```

```
            expression_new(
                'at',
                EQ,
                ('?r', '?l'),
                ('robot', 'location'),
            ),
        ],
        [],
    )

    init_action = action_new(
        init_op,
        {'?c': 'c1', '?l': 'loc1', '?r': 'r1', '?m': 'loc2'},
        {},
    )

    target_action = action_new(
        target_op,
        {'?c': 'c1', '?l': 'loc2', '?r': 'r1'},
        {},
    )

    return pnke_plan(
        dwn_simplified(),
        init_action,
        target_action,
    )
```

Esta é a descrição do domínio de problema para o mundo dos robôs:

```
def robotworld():
    pickupop = operator_new(
        'pickup',
        ['?r', '?o', '?l'],
        ['robot', 'box', 'location'],
        [],
        [
            expression_new(
                'held_by',
                NIL,
```

```
                ('?o', ),
                ('box', 'robot'),
            ),
            expression_new(
                'robot_at',
                EQ,
                ('?r', '?l'),
                ('robot', 'location'),
            ),
            expression_new(
                'at',
                EQ,
                ('?o', '?l'),
                ('box', 'location'),
            ),
        ],
        [
            expression_new(
                'held_by',
                EQ,
                ('?o', '?r'),
                ('box', 'robot'),
            ),
            expression_new(
                'at',
                NIL,
                ('?o', ),
                ('box', 'location'),
            ),
        ],
    )

    dropop = operator_new(
        'drop',
        ['?r', '?o', '?l'],
        ['robot', 'box', 'location'],
        [],
        [
            expression_new(
                'held_by',
                EQ,
```

```
                ('?o', '?r'),
                ('box', 'robot'),
            ),
            expression_new(
                'robot_at',
                EQ,
                ('?r', '?l'),
                ('robot', 'location'),
            ),
            expression_new(
                'at',
                NIL,
                ('?o', ),
                ('box', 'location'),
            ),
        ],
        [
            expression_new(
                'held_by',
                NIL,
                ('?o', ),
                ('box', 'robot'),
            ),
            expression_new(
                'at',
                EQ,
                ('?o', '?l'),
                ('box', 'location'),
            ),
        ],
    )

    moveop = operator_new(
        'move',
        ['?r', '?s', '?e'],
        ['robot', 'location', 'location'],
        [
            Relation(
                'adjacent',
                ('?s', '?e'),
                ('location', 'location'),
```

```
            ),
        ],
        [
            expression_new(
                'robot_at',
                EQ,
                ('?r', '?s'),
                ('robot', 'location'),
            ),
        ],
        [
            expression_new(
                'robot_at',
                EQ,
                ('?r', '?e'),
                ('robot', 'location'),
            ),
        ],
    )

    return domain_new([pickupop, dropop, moveop])
```

Um exemplo de instância de problema para o mundo dos robôs:

```
def robotworld_problem1():
    init_op = operator_new(
        'init',
        ['?r', '?l1', '?b1', '?b2'],
        ['robot', 'location', 'box', 'box'],
        [],
        [],
        [
            expression_new(
                'robot_at',
                EQ,
                ('?r', '?l1'),
                ('robot', 'location'),
            ),
            expression_new(
                'at',
                EQ,
```

```
                ('?b1', '?l1'),
                ('box', 'location'),
            ),
            expression_new(
                'at',
                EQ,
                ('?b2', '?l1'),
                ('box', 'location'),
            ),
            expression_new(
                'held_by',
                NIL,
                ('?b1', ),
                ('box', 'robot'),
            ),
            expression_new(
                'held_by',
                NIL,
                ('?b2', ),
                ('box', 'robot'),
            ),
        ],
    )

    target_op = operator_new(
        'target',
        ['?b1', '?b2', '?l2'],
        ['box', 'box', 'location'],
        [],
        [
            expression_new(
                'at',
                EQ,
                ('?b1', '?l2'),
                ('box', 'location'),
            ),
            expression_new(
                'at',
                EQ,
                ('?b2', '?l2'),
                ('box', 'location'),
```

```
            ),
        ],
        []
    )

    init_action = action_new(
        init_op,
        {'?r': 'r', '?b1': 'b1', '?b2': 'b2', '?l1': 'l1'},
        {},
    )

    target_action = action_new(
        target_op,
        {'?b1': 'b1', '?b2': 'b2', '?l2': 'l2'},
        {},
    )

    return pnke_plan(
        robotworld(),
        init_action,
        target_action,
    )
```

ANEXO B – Gerador de problemas

O código a seguir foi utilizado para gerar problemas para o mundo dos blocos, como descrito pelo anexo A.

As funções `generate_piles` e `generate_piles_expressions` geram as expressões que definem a ordem em que os blocos devem ficar empilhados.

A função `generate_piles_expressions` produz todas as permutações possíveis de problema com um número n de blocos.

```
def generate_piles(length):
    if length == 0:
        yield []

    i = 1
    while length >= i:
        for next in generate_piles(length - i):
            next.append(i)
            assert sum(next) == length
            yield next

        i += 1


def generate_piles_expressions(block_names):
    from itertools import permutations

    length = len(block_names)
    for name_list in permutations(block_names, length):

        for pile_list in generate_piles(length):

            # builds each pile from the top block
            # to the bottom block
            name_pos = -1
            expressions = []
            for pile in pile_list:
                for curr in range(pile):
                    name_pos += 1
                    name = name_list[name_pos]
                    variable = '?' + name.lower()
```

```
                    if curr == 0:
                        expr = expression_new(
                            'on',
                            NIL,
                            (variable, ),
                            ('block', 'block'),
                        )
                    else:
                        expr = expression_new(
                            'on',
                            EQ,
                            (variable, upper_variable),
                            ('block', 'block'),
                        )

                    expressions.append(expr)
                    upper_variable = variable

            yield expressions


def generate_blocksworld(length=3):
    import string

    block_names = string.ascii_uppercase[:length]
    blocksworld_domain = blocksworld()

    for start_position in generate_piles_expressions(block_names):
        for end_position in generate_piles_expressions(block_names):
            variables = [
                '?' + name.lower()
                for name in block_names
            ]
            types = ['block'] * length
            assignments = dict(zip(variables, block_names))

            init_op = operator_new(
                'init',
                variables,
                types,
                [],
```

```
    [],
    start_position,
)

target_op = operator_new(
    'target',
    variables,
    types,
    [],
    end_position,
    [],
)

init_action = action_new(
    init_op,
    assignments,
    {},
)

target_action = action_new(
    target_op,
    assignments,
    {},
)

yield pnke_plan(
    blocksworld_domain,
    init_action,
    target_action,
)
```

ANEXO C – Código fonte

Arquivo `factories.py`

```
# -*- coding: utf-8 -*-
from itertools import chain, count, permutations

from structures import (
    Action,
    ActionLayer,
    Domain,
    Effect,
    Expression,
    Operator,
    PartialPlan,
    EMPTY_DICT,
    EMPTY_LIST,
    EQ,
    NIL,
)

# pylint: disable=too-many-locals,too-many-branches


def gen_symbol():
    for n in count():
        yield '#' + str(n)


def is_substitution_consistent(first, second):
    """ False if `first` and `second` keys intersect and the values differ. """
    # loop over the smallest container
    if len(first) > len(second):
        first, second = second, first

    for k, v in first.items():
        if v.startswith('#'):
            continue

        v2 = second.get(k, v)

        if v2 is None or v2.startswith('#'):
            continue

        if v != v2:
            return False

    return True


def assert_expression_consistent(expression_list):
    for next_pos, precond in enumerate(expression_list, 1):
        for other in expression_list[next_pos:]:
            msg_conflict = 'conflicting precondition found "{}" and "{}"'.format(precond, other)

            if precond.expr_type == EQ:
                if precond.id == other.id:
                    assert precond.variables != other.variables, 'duplicated preconditions found'
                    assert precond.variables[:-1] != other.variables[:-1], msg_conflict

                elif precond.id == other.complement_id:
                    assert precond.variables[:-1] != other.variables, msg_conflict

            elif precond.expr_type == NIL:
                if precond.id == other.id:
                    assert precond.variables != other.variables, 'duplicated preconditions found'

                elif precond.id == other.complement_id:
                    assert precond.variables != other.variables[:-1], msg_conflict


def effect_normalize_assignments(effect, variable_assignments):
    effect_assignment = []
    for value in effect.assignment:
        while value in variable_assignments:
            value = variable_assignments[value]
        effect_assignment.append(value)
    return effect_assignment


def is_effects_consistent(variable_assignments, id_to_effects1, id_to_effects2):
    """ This assumes that id_to_effects1 and id_to_effects2 are consistent with
    themselves.
```

```
    """

    # use the smallest container on the outer loop
    if len(id_to_effects1) > len(id_to_effects2):
        id_to_effects1, id_to_effects2 = id_to_effects2, id_to_effects1

    for id_, effect_list in id_to_effects1.items():  # pylint: disable=too-many-nested-blocks
        expr_type = effect_list[0].expression.expr_type

        if expr_type == EQ:
            for effect1 in effect_list:

                eq_assignment1 = []
                for assignment in effect1.assignment:
                    while assignment in variable_assignments:
                        assignment = variable_assignments[assignment]
                    eq_assignment1.append(assignment)

                eq_params = eq_assignment1[:-1]

                for eq_effect2 in id_to_effects2.get(id_, EMPTY_LIST):
                    eq_assignment2 = []
                    for assignment in eq_effect2.assignment:
                        while assignment in variable_assignments:
                            assignment = variable_assignments[assignment]
                        eq_assignment2.append(assignment)

                    # p(a, b) = c
                    # p(a, b) = d
                    assignment_differ = (
                        eq_assignment2[:-1] == eq_params and
                        eq_assignment2[-1] != eq_assignment1[-1]
                    )

                    if assignment_differ:
                        return False

                for nil_effect2 in id_to_effects2.get(effect1.expression.complement_id, EMPTY_LIST):
                    nil_assignment2 = []
                    for assignment in nil_effect2.assignment:
                        while assignment in variable_assignments:
                            assignment = variable_assignments[assignment]
                        nil_assignment2.append(assignment)

                    # p(a, b) = c
                    # p(a, b) = nil
                    if nil_assignment2 == eq_params:
                        return False

        elif expr_type == NIL:
            for nil_effect1 in effect_list:
                nil_assignment1 = []
                for assignment in nil_effect1.assignment:
                    while assignment in variable_assignments:
                        assignment = variable_assignments[assignment]
                    nil_assignment1.append(assignment)

                for effect2 in id_to_effects2.get(nil_effect1.expression.complement_id, EMPTY_LIST):
                    assignment2 = []
                    for assignment in effect2.assignment:
                        while assignment in variable_assignments:
                            assignment = variable_assignments[assignment]
                        assignment2.append(assignment)

                    # p(a, b) = nil
                    # p(a, b) = c
                    if assignment2[:-1] == nil_assignment1:
                        return False

        else:
            raise RuntimeError()

    return True


GEN_SYMBOL = gen_symbol()


def action_effects_from_substitutions(operator, substitutions):
    id_to_effects = dict()

    for id_, expr_list in operator.id_to_expressions.items():
```

```
        effects = []
        id_to_effects[id_] = effects

        for expr in expr_list:
            assignment = tuple(
                substitutions[v]
                for v in expr.variables
            )
            effects.append(Effect(assignment, expr))

    return id_to_effects


def action_new(operator, object_substitutions, substitutions):
    if __debug__:
        for k, v in object_substitutions.items():
            assert substitutions.get(k, v) == v

    id_to_pendingpreconditions = {}

    # variables that must be generated
    missing_variables_set = set(operator.variables)
    substitutions.update(object_substitutions)
    missing_variables_set.difference_update(substitutions)

    # "fully" instantiated action, instead of using a real object for the
    # substitution a hole is used, this can be used to check for conflicts
    # given that proper care is taken to unify it.
    if missing_variables_set:
        substitutions.update({
            var: next(GEN_SYMBOL)
            for var in missing_variables_set
        })

    # an action starts with all the preconditions pending
    for precond in operator.preconditions_expr:
        id_to_pendingpreconditions.setdefault(precond.id, list()).append(precond)

    id_to_effects = action_effects_from_substitutions(
        operator,
        substitutions,
    )

    return Action(
        id_to_effects,
        id_to_pendingpreconditions,
        object_substitutions,
        operator,
        substitutions,
    )


def action_remove_pending(pending_action, id_, precond_pos):
    id_to_pendingpreconditions = dict(pending_action.id_to_pendingpreconditions)
    pending_preconditions = list(id_to_pendingpreconditions[id_])
    pending_preconditions.pop(precond_pos)
    if not pending_preconditions:
        del id_to_pendingpreconditions[id_]
    else:
        id_to_pendingpreconditions[id_] = pending_preconditions

    return Action(
        pending_action.id_to_effects,
        id_to_pendingpreconditions,
        pending_action.object_substitutions,
        pending_action.operator,
        pending_action.substitutions,
    )


def unify_effect_precondition(  # pylint: disable=too-many-statements,too-many-nested-blocks
        variable_assignments,
        id_,
        effect_action,
        pending_action):
    """ Satisfy a pending precondition with an existing effect.

    Args:
        variable_assignments: Global assingments, used to lazily update the variables.
        id_: The id_ of the effect being satisfied.
        effect_action: The action used to satisfy the precondition.
        pending_action: The action with an unsatisfied precondition.
```

```
    """
    pending_objects = set(pending_action.object_substitutions.values())
    effect_objects = set(effect_action.object_substitutions.values())

    for pending_pos, pending_expr in enumerate(pending_action.id_to_pendingpreconditions[id_]):
        for effect_expr in effect_action.id_to_effects[id_]:
            # Set to False when:
            # - the objects differ and cannot be unified (unification)
            # - the same object is assigned to two variables of the same effect (instantiation)
            is_valid_substitution = True

            # ties up the current value for the effect expression with the
            # corresponding variable name for the pending expression
            var_value = zip(
                pending_expr.variables,
                effect_expr.assignment,
            )

            # unification
            variable_updates = {}
            for pending_variable, effect_value in var_value:
                pending_value = pending_action.substitutions[pending_variable]

                while effect_value in variable_assignments:
                    effect_value = variable_assignments[effect_value]

                while pending_value in variable_assignments:
                    pending_value = variable_assignments[pending_value]

                effect_is_variable = effect_value.startswith('#')
                pending_is_variable = pending_value.startswith('#')

                # if both expressions have an object assigned, these must match
                if not effect_is_variable and not pending_is_variable:
                    if effect_value != pending_value:
                        is_valid_substitution = False
                        break

                # indirectly update the other structures variable assignments
                elif effect_is_variable and not pending_is_variable:
                    current_value = variable_updates.get(effect_value)
                    if current_value:
                        if current_value.startswith('#'):
                            raise RuntimeError()

                        is_valid_substitution = False
                        break

                    variable_updates[effect_value] = pending_value

                # indirectly update the other structures variable assignments
                # update the pending action
                elif not effect_is_variable and pending_is_variable:
                    current_value = variable_updates.get(pending_value)
                    if current_value:
                        if current_value.startswith('#'):
                            raise RuntimeError()

                        is_valid_substitution = False
                        break

                    variable_updates[pending_value] = effect_value

                # unify variables if different (do not set #0 == #0 as this creates
                # an loop for variable unification)
                elif effect_value != pending_value:
                    # Trick edge case:
                    #
                    #   effect: Effect(on, ('#0', '#1'))
                    #   precond: Effect(on, ('a', '#0'))
                    #
                    # The first iteration sets #0=a, without checking the
                    # second iteration overwrites it with #0=#1.

                    if pending_value not in variable_updates:
                        variable_updates[pending_value] = effect_value

                    elif effect_value not in variable_updates:
                        variable_updates[effect_value] = pending_value

                    else:
                        raise RuntimeError()
```

```
if not is_valid_substitution:
    continue

# Tricky edge case:
#
#   effect: Effect(on, ('#9', '#0'))
#   precond: Effect(on, ('#0', 'a'))
#
# Unifying the effect with the precond sets #0 on the effect, which
# in turn sets the #0 in the precondition, this makes the
# substitution invalid
#
# TODO: This is only cheking the validity of relevant/pending
# actions, the variable assignment will also cascade to other
# actions that may become invalid with this variable assignments.
# The proper fix is to only store variables in the effects and have
# a constrained global variable assignment.

# instantiation
new_substitutions = dict(pending_action.substitutions)
new_object_substitutions = dict(pending_action.object_substitutions)

for variable, current_value in pending_action.substitutions.items():
    if current_value.startswith('#'):
        while current_value in variable_assignments:
            current_value = variable_assignments[current_value]

        while current_value in variable_updates:
            current_value = variable_updates[current_value]

        if current_value.startswith('#'):
            new_substitutions[variable] = current_value
        else:
            new_substitutions[variable] = current_value
            new_object_substitutions[variable] = current_value

        # invalid assignment, the same object cannot be used twice
        if current_value in pending_objects:
            is_valid_substitution = False
            break
    else:
        new_object_substitutions[variable] = current_value

if not is_valid_substitution:
    continue

for variable, current_value in effect_action.substitutions.items():
    if current_value.startswith('#'):
        while current_value in variable_assignments:
            current_value = variable_assignments[current_value]

        while current_value in variable_updates:
            current_value = variable_updates[current_value]

        # invalid assignment, the same object cannot be used twice
        if current_value in effect_objects:
            is_valid_substitution = False
            break

if not is_valid_substitution:
    continue

# remove the now satisfied precondition
new_id_to_pendingpreconditions = dict(pending_action.id_to_pendingpreconditions)
pending_preconditions = list(new_id_to_pendingpreconditions[id_])
pending_preconditions.pop(pending_pos)
if not pending_preconditions:
    del new_id_to_pendingpreconditions[id_]
else:
    new_id_to_pendingpreconditions[id_] = pending_preconditions

new_id_to_effects = action_effects_from_substitutions(
    pending_action.operator,
    new_substitutions,
)
action = Action(
    new_id_to_effects,
    new_id_to_pendingpreconditions,
    new_object_substitutions,
    pending_action.operator,
    new_substitutions,
```

```
            )

            effect_values = tuple(
                new_substitutions[pending_var]
                for pending_var in pending_expr.variables
            )
            required_effect = Effect(effect_values, pending_expr)

            yield action, required_effect, variable_updates


def action_layer_new(action_list, id_to_requireeffects, acc_id_to_requireeffects):
    return ActionLayer(
        action_list,
        id_to_requireeffects,
        acc_id_to_requireeffects,
    )


def expression_new(name, expr_type, variables, variables_types):
    variables_set = set(variables)
    assert len(variables) == len(variables_set), 'variable names must be unique'

    # these expr do /not/ have equal ids:
    #   expr(a,b) = c
    #   expr(a,b) = nul
    # the unique ids simplify action lookup
    id_ = hash((name, expr_type, variables_types))

    if expr_type == EQ:
        assert len(variables) == len(variables_types), 'number of variables and types must match'
        complement_id = hash((name, NIL, variables_types))
        expression_types = variables_types
    elif expr_type == NIL:
        # the last type is required to properly compute the complement_id
        assert len(variables) + 1 == len(variables_types)
        complement_id = hash((name, EQ, variables_types))
        expression_types = variables_types[:-1]
    else:
        raise RuntimeError()

    return Expression(
        name,
        expr_type,
        id_,
        complement_id,
        variables,
        expression_types,
    )


def operator_new(
        name,
        variables,
        variables_types,
        required_relations,
        preconditions_expr,
        effects_expr):

    variables_set = set(variables)
    assert len(variables) == len(variables_set), 'variable names must be unique'
    assert len(variables) == len(variables_types), 'number of variables and types must match'

    var_to_type = dict(zip(variables, variables_types))

    # check relation variables types match
    for relation in required_relations:
        for var, type_ in zip(relation.variables, relation.variables_types):
            assert var in var_to_type, 'relation variable must be declared in the operator'
            assert var_to_type[var] == type_, 'relation type must match the one in the operator'

    # check precondition variables types match
    for precond in preconditions_expr:
        for var, type_ in zip(precond.variables, precond.variables_types):
            assert var in var_to_type, 'precondition variable must be declared in the operator'
            assert var_to_type[var] == type_, 'precondition type must match the one in the operator'

    # check effect variables types match
    for eff in effects_expr:
        for var, type_ in zip(eff.variables, eff.variables_types):
            assert var in var_to_type, 'effect variable must be declared in the operator'
            assert var_to_type[var] == type_, 'effect type must match the one in the operator'
```

```
    # check for conflicting preconditions/effects
    assert_expression_consistent(preconditions_expr)
    assert_expression_consistent(effects_expr)

    effs_variables = set(chain.from_iterable(
        eff.variables for eff in effects_expr
    ))

    msg = 'all effect variables must be listed in the operator schema or nil'
    assert effs_variables.issubset(variables_set), msg

    id_to_expressions = {}
    for eff in effects_expr:
        id_to_expressions.setdefault(eff.id, []).append(eff)

    return Operator(
        name,
        variables,
        variables_types,
        required_relations,
        preconditions_expr,
        id_to_expressions,
    )


def domain_new(operators):
    id_to_intantiable_operators = {}
    name_to_expressions = {}
    for op in operators:
        for expressions in op.id_to_expressions.values():
            for expr in expressions:
                id_to_intantiable_operators.setdefault(expr.id, []).append(op)
                name_to_expressions.setdefault(expr.name, []).append((expr, op))

    # TODO:
    # - validate the effects, the same effect must always have the same
    #   structure (number of arguments / types)

    for expressions in name_to_expressions.values():
        it = iter(expressions)
        base_expr, base_op = next(it)

        for expr, op in it:
            msg = 'expressions with the same name but different ids found. op1: {} op2: {}'.format(
                base_op,
                op
            )
            assert expr.id == base_expr.id or expr.id == base_expr.complement_id, msg

    for op in operators:
        for precond in op.preconditions_expr:
            if precond.id not in id_to_intantiable_operators:
                msg = (
                    "The operator '{op}' has an unsatisfiable precondition '{precond}', "
                    "because there are no operators with it as an effects. Please double "
                    "check the preconditions / effects and their types."
                ).format(
                    precond=precond.name,
                    op=op.name,
                )

                raise ValueError(msg)

    return Domain(id_to_intantiable_operators)


def action_in(action, other_list, variable_assignments):
    id_to_pendingpreconditions = action.id_to_pendingpreconditions

    action_substitutions = {}
    for key, value in action.substitutions.items():
        while value in variable_assignments:
            value = variable_assignments[value]
        action_substitutions[key] = value

    for other in other_list:
        if action.operator == other.operator:
            other_substitutions = {}

            for key, value in other.substitutions.items():
                while value in variable_assignments:
```

```
                    value = variable_assignments[value]
                other_substitutions[key] = value

            is_equal = (
                action_substitutions == other_substitutions and
                id_to_pendingpreconditions == other.id_to_pendingpreconditions
            )

            if is_equal:
                return True

    return False

def effect_in(effect, other_effect_list, variable_assignments):
    effect_assignments = effect_normalize_assignments(effect, variable_assignments)

    for other_effect in other_effect_list:
        if effect.expression.id != other_effect.expression.id:
            continue

        other_assignments = effect_normalize_assignments(other_effect, variable_assignments)
        if effect_assignments == other_assignments:
            return True

    return False


def sanity_check(
        domain,
        variable_assignments,
        type_to_objects,
        action_layers,
        id_to_action_by_layerpos,
        id_to_pendingaction_by_layerpos):
    # pylint: disable=too-many-nested-blocks,too-many-statements

    assert not EMPTY_LIST, 'EMPTY_LIST contains an element'
    assert not EMPTY_DICT, 'EMPTY_DICT contains an element'
    assert isinstance(domain, Domain)
    assert not action_layers[0].acc_id_to_requireeffects

    # assignments must not loop
    for k, v in variable_assignments.items():
        assert k != v

    # object_substitutions must contain objects only
    for layer in action_layers:
        for action in layer.action_list:
            for value in action.object_substitutions.values():
                assert not value.startswith('#')

    # all values in substitutions must match the values in object_substitutions
    for layer in action_layers:
        for action in layer.action_list:
            for key, value in action.substitutions.items():
                assert action.object_substitutions.get(key, value)

    # check the action doesnt conflict with itself
    for layer in action_layers:
        for action in layer.action_list:
            for effect_list in action.id_to_effects.values():
                for eff in effect_list:
                    need_check = (
                        eff.expression.expr_type == EQ and
                        eff.expression.complement_id in action.id_to_effects
                    )

                    if need_check:
                        for other_effect in action.id_to_effects[eff.expression.complement_id]:
                            effect_normalized = effect_normalize_assignments(
                                eff,
                                variable_assignments,
                            )

                            other_normalized = effect_normalize_assignments(
                                other_effect,
                                variable_assignments,
                            )
                            assert effect_normalized[:-1] != other_normalized

    # All linearizations must form a valid plan. Check the parallel actions
    # in the same layer don't conflict.
```

```
for layer in action_layers:
    for first, second in permutations(layer.action_list, 2):
        for precond in second.operator.preconditions_expr:
            precond_assignment = []
            for var in precond.variables:
                value = second.substitutions[var]
                while value in variable_assignments:
                    value = variable_assignments[value]
                precond_assignment.append(value)
            precond_assignment = precond_assignment

            if precond.expr_type == EQ:
                for effect in first.id_to_effects.get(precond.id, EMPTY_LIST):
                    effect_normalized = effect_normalize_assignments(
                        effect,
                        variable_assignments,
                    )

                    if effect_normalized[:-1] == precond_assignment[:-1]:
                        assert effect_normalized[-1] == precond_assignment[-1]

                for effect in first.id_to_effects.get(precond.complement_id, EMPTY_LIST):
                    effect_normalized = effect_normalize_assignments(
                        effect,
                        variable_assignments,
                    )

                    assert effect_normalized != precond_assignment[-1]

            elif precond.expr_type == NIL:
                for effect in first.id_to_effects.get(precond.complement_id, EMPTY_LIST):
                    effect_normalized = effect_normalize_assignments(
                        effect,
                        variable_assignments,
                    )

                    assert effect_normalized[:-1] != precond_assignment

# TODO: assignments must not use the same object
# for layer in action_layers:
#     for action in layer.action_list:
#         value_set = set()
#         for value in action.substitutions.values():
#             while value in variable_assignments:
#                 value = variable_assignments[value]
#             value_set.add(value)
#         assert len(value_set) == len(action.substitutions)

# all pending actions are tracked
for layer_pos, layer in enumerate(action_layers):
    for action in layer.action_list:
        for id_, _ in action.id_to_pendingpreconditions.items():
            assert action_in(
                action,
                id_to_pendingaction_by_layerpos[id_][layer_pos],
                variable_assignments,
            )

# all action's effects are registered
for layer_pos, layer in enumerate(action_layers):
    for action in layer.action_list:
        for id_, _ in action.id_to_effects.items():
            assert action_in(
                action,
                id_to_action_by_layerpos[id_][layer_pos],
                variable_assignments,
            )

# everything that is tracked is a valid action
for id_, pending_layer_list in id_to_pendingaction_by_layerpos.items():
    for layer_pos, action_list in enumerate(pending_layer_list):
        for action in action_list:
            assert action_in(
                action,
                action_layers[layer_pos].action_list,
                variable_assignments,
            )

# all registered effects are from valid actions
for id_, effect_layer_list in id_to_action_by_layerpos.items():
    for layer_pos, action_list in enumerate(effect_layer_list):
        for action in action_list:
```

```
            assert action_in(
                action,
                action_layers[layer_pos].action_list,
                variable_assignments,
            )

    # check target action values are not overwriten
    for action in action_layers[-1].action_list:
        for v in action.substitutions.values():
            assert not v.startswith('#')

    # check the substitutions are of the correct type
    for layer in action_layers:
        for action in layer.action_list:
            for var, value in action.substitutions.items():
                if not value.startswith('#'):
                    pos = action.operator.variables.index(var)
                    type_ = action.operator.variables_types[pos]
                    assert value in type_to_objects[type_]

    # required effects must not conflict
    for layer in action_layers:
        for effect_list in layer.acc_id_to_requireeffects.values():
            if effect_list[0].expression.expr_type == EQ:
                for effect_pos, effect in enumerate(effect_list, 1):
                    effect_assignments = effect_normalize_assignments(
                        effect,
                        variable_assignments,
                    )

                    # ignore partially instantiated effects
                    if any(v.startswith('#') for v in effect_assignments):
                        continue

                    for other in effect_list[effect_pos:]:
                        other_assignments = effect_normalize_assignments(
                            other,
                            variable_assignments,
                        )

                        if any(v.startswith('#') for v in other_assignments):
                            continue

                        is_conflict = (
                            effect_assignments[:-1] == other_assignments[:-1] and
                            effect_assignments[-1] != other_assignments[-1]
                        )
                        assert not is_conflict

                    next_nil_effects = layer.acc_id_to_requireeffects.get(
                        effect.expression.complement_id,
                        EMPTY_LIST,
                    )
                    for other in next_nil_effects:
                        other_assignments = effect_normalize_assignments(
                            other,
                            variable_assignments,
                        )

                        is_conflict = (
                            effect_assignments[:-1] == other_assignments
                        )
                        assert not is_conflict

    # the pending effects are accumulated from the last layer to the first
    accumulated = {}
    allow_removal = []
    for layer in action_layers[::-1]:
        for action in layer.action_list:
            satisfied_preconditions = list(action.operator.preconditions_expr)
            for _, pending_preconds in action.id_to_pendingpreconditions.items():
                for pending in pending_preconds:
                    satisfied_preconditions.remove(pending)

            required_effects = []
            for precond in satisfied_preconditions:
                assignment = []
                for k in precond.variables:
                    value = action.substitutions[k]
                    while value in variable_assignments:
                        value = variable_assignments[value]
                    assignment.append(value)
```

```
                eff = Effect(tuple(assignment), precond)
                required_effects.append(eff)

            for effect in required_effects:
                assert effect.expression.id in layer.acc_id_to_requireeffects
                assert effect_in(
                    effect,
                    layer.acc_id_to_requireeffects[effect.expression.id],
                    variable_assignments,
                )

                accumulated.setdefault(effect.expression.id, []).append(effect)

        # actions in different layers may satisfy a required effect, the
        # planner is free to choose whichever action it wants, so we cant
        # assume that the effect is removed on the first occurence
        allow_removal = []
        for id_, required_effects in accumulated.items():
            for action in layer.action_list:
                for eff in action.id_to_effects.get(id_, EMPTY_LIST):
                    if effect_in(eff, required_effects, variable_assignments):
                        allow_removal.append(eff)

        remove_id = []
        for id_, effects in accumulated.items():
            remove_eff = []
            for eff in effects:
                can_remove = effect_in(
                    eff,
                    allow_removal,
                    variable_assignments,
                )

                required = effect_in(
                    eff,
                    layer.acc_id_to_requireeffects.get(id_, EMPTY_LIST),
                    variable_assignments,
                )

                if can_remove and not required:
                    remove_eff.append(eff)

            for eff in remove_eff:
                effects.remove(eff)

            if not effects:
                remove_id.append(id_)
            else:
                for eff in effects:
                    assert effect_in(
                        eff,
                        layer.acc_id_to_requireeffects[id_],
                        variable_assignments,
                    )

        for id_ in remove_id:
            del accumulated[id_]

        assert sorted(accumulated.keys()) == sorted(layer.acc_id_to_requireeffects.keys())


def partial_plan_new(
        domain,
        variable_assignments,
        type_to_objects,
        action_layers,
        id_to_action_by_layerpos,
        id_to_pendingaction_by_layerpos):

    if __debug__:
        sanity_check(
            domain,
            variable_assignments,
            type_to_objects,
            action_layers,
            id_to_action_by_layerpos,
            id_to_pendingaction_by_layerpos,
        )

    return PartialPlan(
        domain,
```

```
        variable_assignments,
        type_to_objects,
        action_layers,
        id_to_action_by_layerpos,
        id_to_pendingaction_by_layerpos,
    )
```

Arquivo `state_variable.py`

```
# -*- coding: utf-8 -*-
from itertools import chain, count, permutations

from structures import (
    Action,
    ActionLayer,
    Domain,
    Effect,
    Expression,
    Operator,
    PartialPlan,
    EMPTY_DICT,
    EMPTY_LIST,
    EQ,
    NIL,
)

# pylint: disable=too-many-locals,too-many-branches


def gen_symbol():
    for n in count():
        yield '#' + str(n)


def is_substitution_consistent(first, second):
    """ False if `first` and `second` keys intersect and the values differ. """
    # loop over the smallest container
    if len(first) > len(second):
        first, second = second, first

    for k, v in first.items():
        if v.startswith('#'):
            continue

        v2 = second.get(k, v)

        if v2 is None or v2.startswith('#'):
            continue

        if v != v2:
            return False

    return True


def assert_expression_consistent(expression_list):
    for next_pos, precond in enumerate(expression_list, 1):
        for other in expression_list[next_pos:]:
            msg_conflict = 'conflicting precondition found "{}" and "{}"'.format(precond, other)

            if precond.expr_type == EQ:
                if precond.id == other.id:
                    assert precond.variables != other.variables, 'duplicated preconditions found'
                    assert precond.variables[:-1] != other.variables[:-1], msg_conflict

                elif precond.id == other.complement_id:
                    assert precond.variables[:-1] != other.variables, msg_conflict

            elif precond.expr_type == NIL:
                if precond.id == other.id:
                    assert precond.variables != other.variables, 'duplicated preconditions found'

                elif precond.id == other.complement_id:
                    assert precond.variables != other.variables[:-1], msg_conflict


def effect_normalize_assignments(effect, variable_assignments):
    effect_assignment = []
```

```
    for value in effect.assignment:
        while value in variable_assignments:
            value = variable_assignments[value]
        effect_assignment.append(value)
    return effect_assignment


def is_effects_consistent(variable_assignments, id_to_effects1, id_to_effects2):
    """ This assumes that id_to_effects1 and id_to_effects2 are consistent with
    themselves.
    """

    # use the smallest container on the outer loop
    if len(id_to_effects1) > len(id_to_effects2):
        id_to_effects1, id_to_effects2 = id_to_effects2, id_to_effects1

    for id_, effect_list in id_to_effects1.items():  # pylint: disable=too-many-nested-blocks
        expr_type = effect_list[0].expression.expr_type

        if expr_type == EQ:
            for effect1 in effect_list:

                eq_assignment1 = []
                for assignment in effect1.assignment:
                    while assignment in variable_assignments:
                        assignment = variable_assignments[assignment]
                    eq_assignment1.append(assignment)

                eq_params = eq_assignment1[:-1]

                for eq_effect2 in id_to_effects2.get(id_, EMPTY_LIST):
                    eq_assignment2 = []
                    for assignment in eq_effect2.assignment:
                        while assignment in variable_assignments:
                            assignment = variable_assignments[assignment]
                        eq_assignment2.append(assignment)

                    # p(a, b) = c
                    # p(a, b) = d
                    assignment_differ = (
                        eq_assignment2[:-1] == eq_params and
                        eq_assignment2[-1] != eq_assignment1[-1]
                    )

                    if assignment_differ:
                        return False

                for nil_effect2 in id_to_effects2.get(effect1.expression.complement_id, EMPTY_LIST):
                    nil_assignment2 = []
                    for assignment in nil_effect2.assignment:
                        while assignment in variable_assignments:
                            assignment = variable_assignments[assignment]
                        nil_assignment2.append(assignment)

                    # p(a, b) = c
                    # p(a, b) = nil
                    if nil_assignment2 == eq_params:
                        return False

        elif expr_type == NIL:
            for nil_effect1 in effect_list:
                nil_assignment1 = []
                for assignment in nil_effect1.assignment:
                    while assignment in variable_assignments:
                        assignment = variable_assignments[assignment]
                    nil_assignment1.append(assignment)

                for effect2 in id_to_effects2.get(nil_effect1.expression.complement_id, EMPTY_LIST):
                    assignment2 = []
                    for assignment in effect2.assignment:
                        while assignment in variable_assignments:
                            assignment = variable_assignments[assignment]
                        assignment2.append(assignment)

                    # p(a, b) = nil
                    # p(a, b) = c
                    if assignment2[:-1] == nil_assignment1:
                        return False

        else:
            raise RuntimeError()
```

```
    return True


GEN_SYMBOL = gen_symbol()


def action_effects_from_substitutions(operator, substitutions):
    id_to_effects = dict()

    for id_, expr_list in operator.id_to_expressions.items():
        effects = []
        id_to_effects[id_] = effects

        for expr in expr_list:
            assignment = tuple(
                substitutions[v]
                for v in expr.variables
            )
            effects.append(Effect(assignment, expr))

    return id_to_effects


def action_new(operator, object_substitutions, substitutions):
    if __debug__:
        for k, v in object_substitutions.items():
            assert substitutions.get(k, v) == v

    id_to_pendingpreconditions = {}

    # variables that must be generated
    missing_variables_set = set(operator.variables)
    substitutions.update(object_substitutions)
    missing_variables_set.difference_update(substitutions)

    # "fully" instantiated action, instead of using a real object for the
    # substitution a hole is used, this can be used to check for conflicts
    # given that proper care is taken to unify it.
    if missing_variables_set:
        substitutions.update({
            var: next(GEN_SYMBOL)
            for var in missing_variables_set
        })

    # an action starts with all the preconditions pending
    for precond in operator.preconditions_expr:
        id_to_pendingpreconditions.setdefault(precond.id, list()).append(precond)

    id_to_effects = action_effects_from_substitutions(
        operator,
        substitutions,
    )

    return Action(
        id_to_effects,
        id_to_pendingpreconditions,
        object_substitutions,
        operator,
        substitutions,
    )


def action_remove_pending(pending_action, id_, precond_pos):
    id_to_pendingpreconditions = dict(pending_action.id_to_pendingpreconditions)
    pending_preconditions = list(id_to_pendingpreconditions[id_])
    pending_preconditions.pop(precond_pos)
    if not pending_preconditions:
        del id_to_pendingpreconditions[id_]
    else:
        id_to_pendingpreconditions[id_] = pending_preconditions

    return Action(
        pending_action.id_to_effects,
        id_to_pendingpreconditions,
        pending_action.object_substitutions,
        pending_action.operator,
        pending_action.substitutions,
    )


def unify_effect_precondition(  # pylint: disable=too-many-statements,too-many-nested-blocks
        variable_assignments,
```

```
    id_,
    effect_action,
    pending_action):
""" Satisfy a pending precondition with an existing effect.

Args:
    variable_assignments: Global assingments, used to lazily update the variables.
    id_: The id_ of the effect being satisfied.
    effect_action: The action used to satisfy the precondition.
    pending_action: The action with an unsatisfied precondition.
"""
pending_objects = set(pending_action.object_substitutions.values())
effect_objects = set(effect_action.object_substitutions.values())

for pending_pos, pending_expr in enumerate(pending_action.id_to_pendingpreconditions[id_]):
    for effect_expr in effect_action.id_to_effects[id_]:
        # Set to False when:
        # - the objects differ and cannot be unified (unification)
        # - the same object is assigned to two variables of the same effect (instantiation)
        is_valid_substitution = True

        # ties up the current value for the effect expression with the
        # corresponding variable name for the pending expression
        var_value = zip(
            pending_expr.variables,
            effect_expr.assignment,
        )

        # unification
        variable_updates = {}
        for pending_variable, effect_value in var_value:
            pending_value = pending_action.substitutions[pending_variable]

            while effect_value in variable_assignments:
                effect_value = variable_assignments[effect_value]

            while pending_value in variable_assignments:
                pending_value = variable_assignments[pending_value]

            effect_is_variable = effect_value.startswith('#')
            pending_is_variable = pending_value.startswith('#')

            # if both expressions have an object assigned, these must match
            if not effect_is_variable and not pending_is_variable:
                if effect_value != pending_value:
                    is_valid_substitution = False
                    break

            # indirectly update the other structures variable assignments
            elif effect_is_variable and not pending_is_variable:
                current_value = variable_updates.get(effect_value)
                if current_value:
                    if current_value.startswith('#'):
                        raise RuntimeError()

                    is_valid_substitution = False
                    break

                variable_updates[effect_value] = pending_value

            # indirectly update the other structures variable assignments
            # update the pending action
            elif not effect_is_variable and pending_is_variable:
                current_value = variable_updates.get(pending_value)
                if current_value:
                    if current_value.startswith('#'):
                        raise RuntimeError()

                    is_valid_substitution = False
                    break

                variable_updates[pending_value] = effect_value

            # unify variables if different (do not set #0 == #0 as this creates
            # an loop for variable unification)
            elif effect_value != pending_value:
                # Trick edge case:
                #
                #   effect: Effect(on, ('#0', '#1'))
                #   precond: Effect(on, ('a', '#0'))
                #
                # The first iteration sets #0=a, without checking the
```

```
            # second iteration overwrites it with #0=#1.

            if pending_value not in variable_updates:
                variable_updates[pending_value] = effect_value

            elif effect_value not in variable_updates:
                variable_updates[effect_value] = pending_value

            else:
                raise RuntimeError()

    if not is_valid_substitution:
        continue

    # Tricky edge case:
    #
    #   effect: Effect(on, ('#9', '#0'))
    #   precond: Effect(on, ('#0', 'a'))
    #
    # Unifying the effect with the precond sets #0 on the effect, which
    # in turn sets the #0 in the precondition, this makes the
    # substitution invalid
    #
    # TODO: This is only cheking the validity of relevant/pending
    # actions, the variable assignment will also cascade to other
    # actions that may become invalid with this variable assignments.
    # The proper fix is to only store variables in the effects and have
    # a constrained global variable assignment.

    # instantiation
    new_substitutions = dict(pending_action.substitutions)
    new_object_substitutions = dict(pending_action.object_substitutions)

    for variable, current_value in pending_action.substitutions.items():
        if current_value.startswith('#'):
            while current_value in variable_assignments:
                current_value = variable_assignments[current_value]

            while current_value in variable_updates:
                current_value = variable_updates[current_value]

            if current_value.startswith('#'):
                new_substitutions[variable] = current_value
            else:
                new_substitutions[variable] = current_value
                new_object_substitutions[variable] = current_value

            # invalid assignment, the same object cannot be used twice
            if current_value in pending_objects:
                is_valid_substitution = False
                break
        else:
            new_object_substitutions[variable] = current_value

    if not is_valid_substitution:
        continue

    for variable, current_value in effect_action.substitutions.items():
        if current_value.startswith('#'):
            while current_value in variable_assignments:
                current_value = variable_assignments[current_value]

            while current_value in variable_updates:
                current_value = variable_updates[current_value]

            # invalid assignment, the same object cannot be used twice
            if current_value in effect_objects:
                is_valid_substitution = False
                break

    if not is_valid_substitution:
        continue

    # remove the now satisfied precondition
    new_id_to_pendingpreconditions = dict(pending_action.id_to_pendingpreconditions)
    pending_preconditions = list(new_id_to_pendingpreconditions[id_])
    pending_preconditions.pop(pending_pos)
    if not pending_preconditions:
        del new_id_to_pendingpreconditions[id_]
    else:
        new_id_to_pendingpreconditions[id_] = pending_preconditions
```

```
            new_id_to_effects = action_effects_from_substitutions(
                pending_action.operator,
                new_substitutions,
            )
            action = Action(
                new_id_to_effects,
                new_id_to_pendingpreconditions,
                new_object_substitutions,
                pending_action.operator,
                new_substitutions,
            )

            effect_values = tuple(
                new_substitutions[pending_var]
                for pending_var in pending_expr.variables
            )
            required_effect = Effect(effect_values, pending_expr)

            yield action, required_effect, variable_updates


def action_layer_new(action_list, id_to_requireeffects, acc_id_to_requireeffects):
    return ActionLayer(
        action_list,
        id_to_requireeffects,
        acc_id_to_requireeffects,
    )


def expression_new(name, expr_type, variables, variables_types):
    variables_set = set(variables)
    assert len(variables) == len(variables_set), 'variable names must be unique'

    # these expr do /not/ have equal ids:
    #   expr(a,b) = c
    #   expr(a,b) = nul
    # the unique ids simplify action lookup
    id_ = hash((name, expr_type, variables_types))

    if expr_type == EQ:
        assert len(variables) == len(variables_types), 'number of variables and types must match'
        complement_id = hash((name, NIL, variables_types))
        expression_types = variables_types
    elif expr_type == NIL:
        # the last type is required to properly compute the complement_id
        assert len(variables) + 1 == len(variables_types)
        complement_id = hash((name, EQ, variables_types))
        expression_types = variables_types[:-1]
    else:
        raise RuntimeError()

    return Expression(
        name,
        expr_type,
        id_,
        complement_id,
        variables,
        expression_types,
    )


def operator_new(
        name,
        variables,
        variables_types,
        required_relations,
        preconditions_expr,
        effects_expr):

    variables_set = set(variables)
    assert len(variables) == len(variables_set), 'variable names must be unique'
    assert len(variables) == len(variables_types), 'number of variables and types must match'

    var_to_type = dict(zip(variables, variables_types))

    # check relation variables types match
    for relation in required_relations:
        for var, type_ in zip(relation.variables, relation.variables_types):
            assert var in var_to_type, 'relation variable must be declared in the operator'
            assert var_to_type[var] == type_, 'relation type must match the one in the operator'

    # check precondition variables types match
```

```
    for precond in preconditions_expr:
        for var, type_ in zip(precond.variables, precond.variables_types):
            assert var in var_to_type, 'precondition variable must be declared in the operator'
            assert var_to_type[var] == type_, 'precondition type must match the one in the operator'

    # check effect variables types match
    for eff in effects_expr:
        for var, type_ in zip(eff.variables, eff.variables_types):
            assert var in var_to_type, 'effect variable must be declared in the operator'
            assert var_to_type[var] == type_, 'effect type must match the one in the operator'

    # check for conflicting preconditions/effects
    assert_expression_consistent(preconditions_expr)
    assert_expression_consistent(effects_expr)

    effs_variables = set(chain.from_iterable(
        eff.variables for eff in effects_expr
    ))

    msg = 'all effect variables must be listed in the operator schema or nil'
    assert effs_variables.issubset(variables_set), msg

    id_to_expressions = {}
    for eff in effects_expr:
        id_to_expressions.setdefault(eff.id, []).append(eff)

    return Operator(
        name,
        variables,
        variables_types,
        required_relations,
        preconditions_expr,
        id_to_expressions,
    )


def domain_new(operators):
    id_to_intantiable_operators = {}
    name_to_expressions = {}
    for op in operators:
        for expressions in op.id_to_expressions.values():
            for expr in expressions:
                id_to_intantiable_operators.setdefault(expr.id, []).append(op)
                name_to_expressions.setdefault(expr.name, []).append((expr, op))

    # TODO:
    # - validate the effects, the same effect must always have the same
    #   structure (number of arguments / types)

    for expressions in name_to_expressions.values():
        it = iter(expressions)
        base_expr, base_op = next(it)

        for expr, op in it:
            msg = 'expressions with the same name but different ids found. op1: {} op2: {}'.format(
                base_op,
                op
            )
            assert expr.id == base_expr.id or expr.id == base_expr.complement_id, msg

    for op in operators:
        for precond in op.preconditions_expr:
            if precond.id not in id_to_intantiable_operators:
                msg = (
                    "The operator '{op}' has an unsatisfiable precondition '{precond}', "
                    "because there are no operators with it as an effects. Please double "
                    "check the preconditions / effects and their types."
                ).format(
                    precond=precond.name,
                    op=op.name,
                )

                raise ValueError(msg)

    return Domain(id_to_intantiable_operators)


def action_in(action, other_list, variable_assignments):
    id_to_pendingpreconditions = action.id_to_pendingpreconditions

    action_substitutions = {}
    for key, value in action.substitutions.items():
```

```
            while value in variable_assignments:
                value = variable_assignments[value]
            action_substitutions[key] = value

        for other in other_list:
            if action.operator == other.operator:
                other_substitutions = {}

                for key, value in other.substitutions.items():
                    while value in variable_assignments:
                        value = variable_assignments[value]
                    other_substitutions[key] = value

                is_equal = (
                    action_substitutions == other_substitutions and
                    id_to_pendingpreconditions == other.id_to_pendingpreconditions
                )

                if is_equal:
                    return True

        return False

    def effect_in(effect, other_effect_list, variable_assignments):
        effect_assignments = effect_normalize_assignments(effect, variable_assignments)

        for other_effect in other_effect_list:
            if effect.expression.id != other_effect.expression.id:
                continue

            other_assignments = effect_normalize_assignments(other_effect, variable_assignments)
            if effect_assignments == other_assignments:
                return True

        return False


    def sanity_check(
            domain,
            variable_assignments,
            type_to_objects,
            action_layers,
            id_to_action_by_layerpos,
            id_to_pendingaction_by_layerpos):
        # pylint: disable=too-many-nested-blocks,too-many-statements

        assert not EMPTY_LIST, 'EMPTY_LIST contains an element'
        assert not EMPTY_DICT, 'EMPTY_DICT contains an element'
        assert isinstance(domain, Domain)
        assert not action_layers[0].acc_id_to_requireeffects

        # assignments must not loop
        for k, v in variable_assignments.items():
            assert k != v

        # object_substitutions must contain objects only
        for layer in action_layers:
            for action in layer.action_list:
                for value in action.object_substitutions.values():
                    assert not value.startswith('#')

        # all values in substitutions must match the values in object_substitutions
        for layer in action_layers:
            for action in layer.action_list:
                for key, value in action.substitutions.items():
                    assert action.object_substitutions.get(key, value)

        # check the action doesnt conflict with itself
        for layer in action_layers:
            for action in layer.action_list:
                for effect_list in action.id_to_effects.values():
                    for eff in effect_list:
                        need_check = (
                            eff.expression.expr_type == EQ and
                            eff.expression.complement_id in action.id_to_effects
                        )

                        if need_check:
                            for other_effect in action.id_to_effects[eff.expression.complement_id]:
                                effect_normalized = effect_normalize_assignments(
                                    eff,
                                    variable_assignments,
```

```
                        )

                        other_normalized = effect_normalize_assignments(
                            other_effect,
                            variable_assignments,
                        )
                        assert effect_normalized[:-1] != other_normalized

    # All linearizations must form a valid plan. Check the parallel actions
    # in the same layer don't conflict.
    for layer in action_layers:
        for first, second in permutations(layer.action_list, 2):
            for precond in second.operator.preconditions_expr:
                precond_assignment = []
                for var in precond.variables:
                    value = second.substitutions[var]
                    while value in variable_assignments:
                        value = variable_assignments[value]
                    precond_assignment.append(value)
                precond_assignment = precond_assignment

                if precond.expr_type == EQ:
                    for effect in first.id_to_effects.get(precond.id, EMPTY_LIST):
                        effect_normalized = effect_normalize_assignments(
                            effect,
                            variable_assignments,
                        )

                        if effect_normalized[:-1] == precond_assignment[:-1]:
                            assert effect_normalized[-1] == precond_assignment[-1]

                    for effect in first.id_to_effects.get(precond.complement_id, EMPTY_LIST):
                        effect_normalized = effect_normalize_assignments(
                            effect,
                            variable_assignments,
                        )

                        assert effect_normalized != precond_assignment[-1]

                elif precond.expr_type == NIL:
                    for effect in first.id_to_effects.get(precond.complement_id, EMPTY_LIST):
                        effect_normalized = effect_normalize_assignments(
                            effect,
                            variable_assignments,
                        )

                        assert effect_normalized[:-1] != precond_assignment

    # TODO: assignments must not use the same object
    # for layer in action_layers:
    #     for action in layer.action_list:
    #         value_set = set()
    #         for value in action.substitutions.values():
    #             while value in variable_assignments:
    #                 value = variable_assignments[value]
    #             value_set.add(value)
    #         assert len(value_set) == len(action.substitutions)

    # all pending actions are tracked
    for layer_pos, layer in enumerate(action_layers):
        for action in layer.action_list:
            for id_, _ in action.id_to_pendingpreconditions.items():
                assert action_in(
                    action,
                    id_to_pendingaction_by_layerpos[id_][layer_pos],
                    variable_assignments,
                )

    # all action's effects are registered
    for layer_pos, layer in enumerate(action_layers):
        for action in layer.action_list:
            for id_, _ in action.id_to_effects.items():
                assert action_in(
                    action,
                    id_to_action_by_layerpos[id_][layer_pos],
                    variable_assignments,
                )

    # everything that is tracked is a valid action
    for id_, pending_layer_list in id_to_pendingaction_by_layerpos.items():
        for layer_pos, action_list in enumerate(pending_layer_list):
            for action in action_list:
```

```
            assert action_in(
                action,
                action_layers[layer_pos].action_list,
                variable_assignments,
            )

# all registered effects are from valid actions
for id_, effect_layer_list in id_to_action_by_layerpos.items():
    for layer_pos, action_list in enumerate(effect_layer_list):
        for action in action_list:
            assert action_in(
                action,
                action_layers[layer_pos].action_list,
                variable_assignments,
            )

# check target action values are not overwriten
for action in action_layers[-1].action_list:
    for v in action.substitutions.values():
        assert not v.startswith('#')

# check the substitutions are of the correct type
for layer in action_layers:
    for action in layer.action_list:
        for var, value in action.substitutions.items():
            if not value.startswith('#'):
                pos = action.operator.variables.index(var)
                type_ = action.operator.variables_types[pos]
                assert value in type_to_objects[type_]

# required effects must not conflict
for layer in action_layers:
    for effect_list in layer.acc_id_to_requireeffects.values():
        if effect_list[0].expression.expr_type == EQ:
            for effect_pos, effect in enumerate(effect_list, 1):
                effect_assignments = effect_normalize_assignments(
                    effect,
                    variable_assignments,
                )

                # ignore partially instantiated effects
                if any(v.startswith('#') for v in effect_assignments):
                    continue

                for other in effect_list[effect_pos:]:
                    other_assignments = effect_normalize_assignments(
                        other,
                        variable_assignments,
                    )

                    if any(v.startswith('#') for v in other_assignments):
                        continue

                    is_conflict = (
                        effect_assignments[:-1] == other_assignments[:-1] and
                        effect_assignments[-1] != other_assignments[-1]
                    )
                    assert not is_conflict

                next_nil_effects = layer.acc_id_to_requireeffects.get(
                    effect.expression.complement_id,
                    EMPTY_LIST,
                )
                for other in next_nil_effects:
                    other_assignments = effect_normalize_assignments(
                        other,
                        variable_assignments,
                    )

                    is_conflict = (
                        effect_assignments[:-1] == other_assignments
                    )
                    assert not is_conflict

# the pending effects are accumulated from the last layer to the first
accumulated = {}
allow_removal = []
for layer in action_layers[::-1]:
    for action in layer.action_list:
        satisfied_preconditions = list(action.operator.preconditions_expr)
        for _, pending_preconds in action.id_to_pendingpreconditions.items():
            for pending in pending_preconds:
```

```
                    satisfied_preconditions.remove(pending)

            required_effects = []
            for precond in satisfied_preconditions:
                assignment = []
                for k in precond.variables:
                    value = action.substitutions[k]
                    while value in variable_assignments:
                        value = variable_assignments[value]
                    assignment.append(value)

                eff = Effect(tuple(assignment), precond)
                required_effects.append(eff)

            for effect in required_effects:
                assert effect.expression.id in layer.acc_id_to_requireeffects
                assert effect_in(
                    effect,
                    layer.acc_id_to_requireeffects[effect.expression.id],
                    variable_assignments,
                )

                accumulated.setdefault(effect.expression.id, []).append(effect)

        # actions in different layers may satisfy a required effect, the
        # planner is free to choose whichever action it wants, so we cant
        # assume that the effect is removed on the first occurence
        allow_removal = []
        for id_, required_effects in accumulated.items():
            for action in layer.action_list:
                for eff in action.id_to_effects.get(id_, EMPTY_LIST):
                    if effect_in(eff, required_effects, variable_assignments):
                        allow_removal.append(eff)

        remove_id = []
        for id_, effects in accumulated.items():
            remove_eff = []
            for eff in effects:
                can_remove = effect_in(
                    eff,
                    allow_removal,
                    variable_assignments,
                )

                required = effect_in(
                    eff,
                    layer.acc_id_to_requireeffects.get(id_, EMPTY_LIST),
                    variable_assignments,
                )

                if can_remove and not required:
                    remove_eff.append(eff)

            for eff in remove_eff:
                effects.remove(eff)

            if not effects:
                remove_id.append(id_)
            else:
                for eff in effects:
                    assert effect_in(
                        eff,
                        layer.acc_id_to_requireeffects[id_],
                        variable_assignments,
                    )

        for id_ in remove_id:
            del accumulated[id_]

        assert sorted(accumulated.keys()) == sorted(layer.acc_id_to_requireeffects.keys())


def partial_plan_new(
        domain,
        variable_assignments,
        type_to_objects,
        action_layers,
        id_to_action_by_layerpos,
        id_to_pendingaction_by_layerpos):

    if __debug__:
        sanity_check(
```

```
            domain,
            variable_assignments,
            type_to_objects,
            action_layers,
            id_to_action_by_layerpos,
            id_to_pendingaction_by_layerpos,
        )

    return PartialPlan(
        domain,
        variable_assignments,
        type_to_objects,
        action_layers,
        id_to_action_by_layerpos,
        id_to_pendingaction_by_layerpos,
    )
```

Arquivo `state_variable.py`

```
# -*- coding: utf-8 -*-
from __future__ import print_function, division

import time
from heapq import heappush, heappop

from structures import (
    ChoicePoint,
    Effect,
    Rank,
    Result,
    EMPTY_DICT,
    EMPTY_LIST,
)
from factories import (
    action_new,
    action_layer_new,
    action_remove_pending,
    unify_effect_precondition,
    partial_plan_new,

    is_effects_consistent,
)

# pylint: disable=line-too-long,too-many-locals,too-many-branches,too-many-lines,too-many-nested-blocks


def merge_inplace(to_id_to_list, id_to_list):
    for id_, item_list in id_to_list.items():
        if id_ not in to_id_to_list:
            # copy it because this will be updated in place
            to_id_to_list[id_] = list(item_list)

        else:
            current_list = to_id_to_list[id_]

            for item in item_list:
                if item not in current_list:
                    current_list.append(item)


def genplan_precond_substitutions(partial_plan):  # pylint: disable=too-many-statements
    """ Adds causal links among existing actions without reordering. """
    action_layers = partial_plan.action_layers
    total_layers = len(action_layers)

    id_to_action_by_layerpos = partial_plan.id_to_action_by_layerpos
    id_to_pendingaction_by_layerpos = partial_plan.id_to_pendingaction_by_layerpos

    # For each action with an unsatisfied precondition:
    #   find an action with an effect that can satisfy the precondition
    #   unify the effect with the precondition
    #
    #   If the unified action is consistent:
    #     update the partial plan
    #     update all the tracking structures

    for id_, pendingactions_by_layerpos in id_to_pendingaction_by_layerpos.items():
        layer_with_effect = id_to_action_by_layerpos.get(id_)
```

```
        if layer_with_effect is None:
            continue

        pending_layer_pos = len(pendingactions_by_layerpos)

        for pending_action_list in reversed(pendingactions_by_layerpos):
            pending_layer_pos -= 1
            idx_pending_action_pos = -1
            pending_layer = action_layers[pending_layer_pos]
            action_list = pending_layer.action_list

            for pending_action in pending_action_list:
                idx_pending_action_pos += 1
                relevant_layer_list = layer_with_effect[:pending_layer_pos]

                relevant_layer_pos = -1
                for idx_relevant_layer in relevant_layer_list:
                    relevant_layer_pos += 1

                    for effect_action in idx_relevant_layer:

                        # At this point is not yet possible to check
                        # consistency. It's first required to known the effect
                        # that will be unified, and then check consistency
                        # against that effect.

                        new_pending_actions = unify_effect_precondition(
                            partial_plan.variable_assignments,
                            id_,
                            effect_action,
                            pending_action,
                        )

                        # the action position in the pending index and
                        # the action layer is not always the same
                        pending_action_pos = action_list.index(pending_action)

                        for new_pending_action, required_effect, variable_updates in new_pending_actions:

                            if variable_updates:
                                variable_assignments = dict(partial_plan.variable_assignments)
                                variable_assignments.update(variable_updates)
                            else:
                                variable_assignments = partial_plan.variable_assignments

                            is_consistent = True
                            id_to_required_effect = {required_effect.expression.id: [required_effect]}

                            is_consistent = is_effects_consistent(
                                variable_assignments,
                                id_to_required_effect,
                                action_layers[relevant_layer_pos + 1].acc_id_to_requireeffects,
                            )

                            if not is_consistent:
                                continue

                            # check if the pending action breaked a required
                            # effect from the next layer after unification
                            if len(action_layers) >= pending_layer_pos + 2:
                                is_consistent = is_effects_consistent(
                                    variable_assignments,
                                    new_pending_action.id_to_effects,
                                    action_layers[pending_layer_pos + 1].acc_id_to_requireeffects,
                                )

                                if not is_consistent:
                                    continue

                            new_action_list = (
                                *action_list[:pending_action_pos],
                                new_pending_action,
                                *action_list[pending_action_pos + 1:],
                            )

                            new_acc_id_to_requireeffects = dict(pending_layer.acc_id_to_requireeffects)
                            index_add_required_effect(
                                new_acc_id_to_requireeffects,
                                required_effect,
                            )
```

```
new_pending_layer = action_layer_new(
    new_action_list,
    pending_layer.id_to_preconditions,
    new_acc_id_to_requireeffects,
)

# Note: do not include the effect of the action in
# the layer it belongs to, this simplifies code for
# adding a new layer
start_effect_pos = relevant_layer_pos + 1
end_effect_pos = pending_layer_pos

# The updated acc_id_to_requireeffects is required
# for the effect conflict check bellow
effect_layers = index_add_required_effect_for_layers(
    action_layers,
    start_effect_pos,
    end_effect_pos,
    required_effect,
)
new_plan = (
    *action_layers[:start_effect_pos],
    *effect_layers,
    new_pending_layer,
    *action_layers[pending_layer_pos + 1:],
)

# check if the new substitutions created an
# inconsistency.
# TODO: Add a index to skip the unecessary effects.
# Because the variable substitution may ground a
# partially instantiated effect, all the
# required_effect must be checked.
layer_effect = (
    (layer, effect)
    for layer in new_plan
    for effect_list in layer.acc_id_to_requireeffects.values()
    for effect in effect_list
)
for layer, eff in layer_effect:
    id_to_required_effect = {eff.expression.id: [eff]}
    is_consistent = is_effects_consistent(
        variable_assignments,
        id_to_required_effect,
        layer.acc_id_to_requireeffects,
    )

    if not is_consistent:
        break

if not is_consistent:
    continue

# Don't allow two equal actions on the same layer.
# Note that actions with the same operator but
# different substitutions are allowed.
#
# TODO: This may happen on other layers because of
# the variable assignment, the todo index for
# conflicting effects can be used to track this
# here too. Additionally the pending_layer actions
# should be orderd to speed up comparison.
if len(pending_layer) > 1:
    for pos, other_action in enumerate(action_list):
        if pos != pending_action_pos and new_pending_action == other_action:
            is_consistent = False
            break

    if not is_consistent:
        continue


new_id_to_pendingaction = dict(id_to_pendingaction_by_layerpos)
index_remove_pending(
    new_id_to_pendingaction,
    id_,
    pending_layer_pos,
    idx_pending_action_pos,
    new_pending_action,
)

index_update_preconditions(
```

```
                                    new_id_to_pendingaction,
                                    pending_layer_pos,
                                    pending_action,
                                    new_pending_action,
                                )

                                new_id_to_action_by_layerpos = dict(id_to_action_by_layerpos)
                                index_update_effects(
                                    new_id_to_action_by_layerpos,
                                    pending_layer_pos,
                                    pending_action,
                                    new_pending_action,
                                    total_layers,
                                )

                                yield choice_point_new(partial_plan_new(
                                    partial_plan.domain,
                                    variable_assignments,
                                    partial_plan.type_to_objects,
                                    new_plan,
                                    new_id_to_action_by_layerpos,
                                    new_id_to_pendingaction,
                                ))


def index_update_preconditions(new_id_to_pendingaction, layer_pos, old_pending_action, new_pending_action):
    """ Update the action in the other indexes. """
    for pending_id in new_pending_action.id_to_pendingpreconditions.keys():
        new_pending = list(new_id_to_pendingaction[pending_id])
        new_pending_layer = list(new_pending[layer_pos])

        index = new_pending_layer.index(old_pending_action)

        new_pending_layer[index] = new_pending_action
        new_pending[layer_pos] = new_pending_layer
        new_id_to_pendingaction[pending_id] = new_pending


def index_add_required_effect_for_layers(new_plan, start, end, required_effect):
    """ Set the required effect in all layers. """
    layer_update = start - 1

    new_layers = []
    for layer in new_plan[start:end]:
        layer_update += 1

        new_acc_id_to_requireeffects = dict(layer.acc_id_to_requireeffects)
        index_add_required_effect(
            new_acc_id_to_requireeffects,
            required_effect,
        )

        new_layer = action_layer_new(
            layer.action_list,
            layer.id_to_preconditions,
            new_acc_id_to_requireeffects,
        )
        new_layers.append(new_layer)

    return new_layers


def index_add_required_effect(acc_id_to_requireeffects, required_effect):
    """ Track the required effect to satisfy a precondition.

    This tracks all the required preconditions aggregated by layer. It is used
    to check consistency when a new new action is added to the plan, to avoid
    destructing a required effect for another action.
    """
    effect_id = required_effect.expression.id
    required_effect_list = acc_id_to_requireeffects.get(effect_id)

    if required_effect_list is None:
        required_effect_list = [required_effect]

    elif required_effect not in required_effect_list:
        required_effect_list = list(required_effect_list)
        required_effect_list.append(required_effect)

    acc_id_to_requireeffects[effect_id] = required_effect_list
```

```
def index_add_pending_preconditions_new_action(
        id_to_pendingaction_by_layerpos,
        action,
        action_layer_pos,
        total_layers):
    """ Track pending preconditions of a new action.

    This adds the *pending preconditions* of the new action to the indexing
    structure.
    """
    if not action.id_to_pendingpreconditions:
        return

    for id_ in action.id_to_pendingpreconditions:
        layerpos_to_actions = id_to_pendingaction_by_layerpos.get(id_)

        if layerpos_to_actions is None:
            layerpos_to_actions = [EMPTY_LIST] * total_layers
        else:
            layerpos_to_actions = list(layerpos_to_actions)

        actions = list(layerpos_to_actions[action_layer_pos])
        layerpos_to_actions[action_layer_pos] = actions

        # assume the action is always appended
        actions.append(action)

        id_to_pendingaction_by_layerpos[id_] = layerpos_to_actions


def index_remove_pending(
        id_to_pendingaction_by_layerpos,
        id_,
        pending_layer_pos,
        pending_action_pos,
        new_pending_action):
    """ Update the pending index.

    If the action has all of its preconditions satisfied, remove it from the
    index, otherwise update it for the next resolution to work.
    """

    pending_layers = id_to_pendingaction_by_layerpos[id_]
    pending_actions = pending_layers[pending_layer_pos]

    # actions can have the same precondition multiple times, remove from the
    # index only if all of these were satisfied
    if id_ not in new_pending_action.id_to_pendingpreconditions:
        new_pending_actions = list(pending_actions)
        new_pending_actions.pop(pending_action_pos)

        new_pending_layers = list(pending_layers)
        new_pending_layers[pending_layer_pos] = new_pending_actions

        if not any(new_pending_layers):
            del id_to_pendingaction_by_layerpos[id_]
        else:
            id_to_pendingaction_by_layerpos[id_] = new_pending_layers

    # else: the function index_update_preconditions is used to update the index
    # with the new pending action if the precondition was not completely
    # satisfied


def index_add_effects_by_layerpos(id_to_action_by_layerpos, action, action_layer, total_layers):
    """ Track actions effects.

    Preconditions may be satified by using the effects of available actions,
    this function add an action with its *effects* to the tracking structure to
    speed up the search of an available effect.
    """
    for id_ in action.id_to_effects:
        actions_by_pos = id_to_action_by_layerpos.get(id_)

        if actions_by_pos is None:
            actions_by_pos = [EMPTY_LIST] * total_layers
        else:
            actions_by_pos = list(actions_by_pos)

        id_to_action_by_layerpos[id_] = actions_by_pos

        actions = list(actions_by_pos[action_layer])
```

```
        actions_by_pos[action_layer] = actions

        actions.append(action)


def index_update_effects(
        id_to_action_by_layerpos,
        action_layer_pos,
        old_action,
        new_action_,
        total_layers):
    """ Replace the old action with the new instance. """

    for id_ in new_action_.id_to_effects:
        id_to_action = id_to_action_by_layerpos.get(id_)

        if id_to_action is None:
            new_id_to_action = [EMPTY_LIST] * total_layers
        else:
            new_id_to_action = list(id_to_action)

        new_layer = list(new_id_to_action[action_layer_pos])

        if id_ in old_action.id_to_effects:
            new_layer[new_layer.index(old_action)] = new_action_
        else:
            new_layer.append(new_action_)

        new_id_to_action[action_layer_pos] = new_layer
        id_to_action_by_layerpos[id_] = new_id_to_action


def add_action_to_layer(  # pylint: disable=too-many-statements
        domain,
        partial_plan,
        satisfing_effect,
        effect_action,
        old_pending_action,
        new_pending_action,
        id_to_pendingaction_by_layerpos,
        pending_layer_pos):

    action_layers = partial_plan.action_layers

    id_to_action_by_layerpos = partial_plan.id_to_action_by_layerpos
    current_layer = action_layers[pending_layer_pos - 1]

    pending_layer = action_layers[pending_layer_pos]
    new_acc_id_to_requireeffects = dict(pending_layer.acc_id_to_requireeffects)
    index_add_required_effect(
        new_acc_id_to_requireeffects,
        satisfing_effect,
    )

    action_list = pending_layer.action_list
    pending_action_pos = action_list.index(old_pending_action)
    new_action_list = (
        *action_list[:pending_action_pos],
        new_pending_action,
        *action_list[pending_action_pos + 1:],
    )

    new_satisfied_layer = action_layer_new(
        new_action_list,
        pending_layer.id_to_preconditions,
        new_acc_id_to_requireeffects,
    )

    # don't add an action to the init layer
    if pending_layer_pos == 1:
        insert_into_layer_pos = pending_layer_pos

        id_to_preconditions = {}
        for precond in effect_action.operator.preconditions_expr:
            required_precondition = Effect(
                tuple(effect_action.substitutions[v] for v in precond.variables),
                precond,
            )

            if precond.id in id_to_preconditions:
                new_preconditions = id_to_preconditions[precond.id]
                new_preconditions.append(required_precondition)
```

```
            else:
                id_to_preconditions[precond.id] = [required_precondition]

        # preserve substitution effects
        acc_id_to_requireeffects = pending_layer.acc_id_to_requireeffects
        new_layer = action_layer_new(
            (effect_action, ),
            id_to_preconditions,
            acc_id_to_requireeffects,
        )
        new_plan = (
            *action_layers[:pending_layer_pos],
            new_layer,
            new_satisfied_layer,
            *action_layers[pending_layer_pos+1:],
        )

        # add the new layer to all indexes
        new_id_to_action_by_layerpos = {}
        for id_, actions_by_pos in id_to_action_by_layerpos.items():
            new_actions_by_pos = [actions_by_pos[0], EMPTY_LIST]
            new_actions_by_pos.extend(actions_by_pos[1:])
            new_id_to_action_by_layerpos[id_] = new_actions_by_pos

        new_id_to_pendinglayers = {}
        for id_, actions_by_pos in id_to_pendingaction_by_layerpos.items():
            new_pending_by_pos = [actions_by_pos[0], EMPTY_LIST]
            new_pending_by_pos.extend(actions_by_pos[1:])
            new_id_to_pendinglayers[id_] = new_pending_by_pos

    else:
        # Check the new action can be scheduled before the current layer's
        # actions. IOW, the new action's effects must not break a precondition
        # of an existing action.
        #
        # This is exemplified by the sussman anomaly, were the target
        # precondition needs two move actions (move b to c, and move a to b),
        # since the second invalidates the first, these must be added into
        # different layers.
        all_orderings_are_consistent = is_effects_consistent(
            partial_plan.variable_assignments,
            effect_action.id_to_effects,
            current_layer.id_to_preconditions,
        )

        # Check the new action can be scheduled after the current layer's
        # actions. IOW, the existing actions must not break a precondition of
        # the new action.
        if all_orderings_are_consistent:
            new_id_to_preconditions = dict(current_layer.id_to_preconditions)
            for precond in effect_action.operator.preconditions_expr:
                required_precondition = Effect(
                    tuple(effect_action.substitutions[v] for v in precond.variables),
                    precond,
                )

                if precond.id in new_id_to_preconditions:
                    new_preconditions = list(new_id_to_preconditions[precond.id])
                    new_preconditions.append(required_precondition)
                    new_id_to_preconditions[precond.id] = new_preconditions
                else:
                    new_id_to_preconditions[precond.id] = [required_precondition]

            for action in current_layer.action_list:
                all_orderings_are_consistent = is_effects_consistent(
                    partial_plan.variable_assignments,
                    action.id_to_effects,
                    new_id_to_preconditions,
                )
                if not all_orderings_are_consistent:
                    break

        if all_orderings_are_consistent:
            # All orderings are valid, create a new layer containing the new
            # action.
            action_list = (*current_layer.action_list, effect_action)
            new_layer = action_layer_new(
                action_list,
                new_id_to_preconditions,
                current_layer.acc_id_to_requireeffects,
            )
            insert_into_layer_pos = pending_layer_pos - 1
```

```
            new_plan = (
                *action_layers[:insert_into_layer_pos],
                new_layer,
                new_satisfied_layer,
                *action_layers[pending_layer_pos+1:],
            )

            new_id_to_action_by_layerpos = dict(id_to_action_by_layerpos)
            new_id_to_pendinglayers = dict(id_to_pendingaction_by_layerpos)

        else:
            # If the action cannot be added to the current layer, create a new one.
            #
            # This is required for actions with conflicting preconditions.
            # IOW, a new layer is created only if the previous is the init
            # layers or there is a conflict.
            insert_into_layer_pos = pending_layer_pos

            new_id_to_action_by_layerpos = {}
            for id_, actions_by_pos in id_to_action_by_layerpos.items():
                new_actions_by_pos = actions_by_pos[0:insert_into_layer_pos]
                new_actions_by_pos.append(EMPTY_LIST)
                new_actions_by_pos.extend(actions_by_pos[insert_into_layer_pos:])
                new_id_to_action_by_layerpos[id_] = new_actions_by_pos

            new_id_to_pendinglayers = {}
            for id_, actions_by_pos in id_to_pendingaction_by_layerpos.items():
                new_pending_by_pos = actions_by_pos[0:insert_into_layer_pos]
                new_pending_by_pos.append(EMPTY_LIST)
                new_pending_by_pos.extend(actions_by_pos[insert_into_layer_pos:])
                new_id_to_pendinglayers[id_] = new_pending_by_pos

            id_to_preconditions = {}
            for precond in effect_action.operator.preconditions_expr:
                required_precondition = Effect(
                    tuple(effect_action.substitutions[v] for v in precond.variables),
                    precond,
                )

                if precond.id in id_to_preconditions:
                    new_preconditions = id_to_preconditions[precond.id]
                    new_preconditions.append(required_precondition)
                else:
                    id_to_preconditions[precond.id] = [required_precondition]

            new_layer = action_layer_new(
                (effect_action, ),
                id_to_preconditions,
                pending_layer.acc_id_to_requireeffects,
            )

            new_plan = (
                *action_layers[:pending_layer_pos],
                new_layer,
                new_satisfied_layer,
                *action_layers[pending_layer_pos+1:],
            )

    total_layers = len(new_plan)

    index_add_effects_by_layerpos(
        new_id_to_action_by_layerpos,
        effect_action,
        insert_into_layer_pos,
        total_layers,
    )

    index_add_pending_preconditions_new_action(
        new_id_to_pendinglayers,
        effect_action,
        insert_into_layer_pos,
        total_layers,
    )

    index_update_effects(
        new_id_to_action_by_layerpos,
        insert_into_layer_pos + 1,
        old_pending_action,
        new_pending_action,
        total_layers,
    )
```

```
    return partial_plan_new(
        domain,
        partial_plan.variable_assignments,
        partial_plan.type_to_objects,
        new_plan,
        new_id_to_action_by_layerpos,
        new_id_to_pendinglayers,
    )


def genplan_new_action(partial_plan):
    """ Add a new action to satisfy a pending precondition. """
    domain = partial_plan.domain
    action_layers = partial_plan.action_layers

    # For each action with an unsatisfied precondition:
    #   find an operator with an effect that can satisfy the precondition
    #   partially instantiate the operator into an action
    #
    #   If the partial action is consistent:
    #     add the partial action to a new partial plan
    #     update all the tracking structures

    for id_, pending_layers in partial_plan.id_to_pendingaction_by_layerpos.items():
        pending_layer_pos = len(pending_layers)

        for pending_action_list in reversed(pending_layers):
            pending_layer_pos -= 1
            idx_pending_action_pos = -1
            pending_layer = action_layers[pending_layer_pos]

            for pending_action in pending_action_list:
                idx_pending_action_pos += 1
                precond_pos = -1

                for pending_precond in pending_action.id_to_pendingpreconditions[id_]:
                    precond_pos += 1

                    # new_pending_action and id_to_pendingaction_by_layerpos are shared
                    # among all possible partial plans with different operator
                    # choices

                    # update the action removing the now satisfied precond
                    new_pending_action = action_remove_pending(
                        pending_action,
                        id_,
                        precond_pos,
                    )

                    new_id_to_pendingaction = dict(
                        partial_plan.id_to_pendingaction_by_layerpos
                    )

                    index_remove_pending(
                        new_id_to_pendingaction,
                        id_,
                        pending_layer_pos,
                        idx_pending_action_pos,
                        new_pending_action,
                    )

                    index_update_preconditions(
                        new_id_to_pendingaction,
                        pending_layer_pos,
                        pending_action,
                        new_pending_action,
                    )

                    # some expressions may be set only on the init action only
                    # that cannot be instantiated
                    instantiable_operators = domain.id_to_intantiable_operators.get(id_, EMPTY_LIST)

                    for op in instantiable_operators:
                        for op_expr in op.id_to_expressions[id_]:

                            required_effect = Effect(
                                tuple(
                                    pending_action.substitutions[k]
                                    for k in pending_precond.variables
                                ),
                                pending_precond,
                            )
```

```
                substitutions = {
                    k: v
                    for k, v in zip(op_expr.variables, required_effect.assignment)
                }
                object_substitutions = {
                    k: v
                    for k, v in substitutions.items()
                    if not v.startswith('#')
                }

                action = action_new(
                    op,
                    object_substitutions,
                    substitutions,
                )

                # Even though the action was chosen to satisfy a
                # single effect, a single substitution may ground
                # multiple effects, so the consistency check must
                # use an action instance
                is_consistent = is_effects_consistent(
                    partial_plan.variable_assignments,
                    action.id_to_effects,
                    pending_layer.acc_id_to_requireeffects,
                )
                if not is_consistent:
                    continue

                new_plan = add_action_to_layer(
                    domain,
                    partial_plan,
                    required_effect,
                    action,
                    pending_action,
                    new_pending_action,
                    new_id_to_pendingaction,
                    pending_layer_pos,
                )

                yield choice_point_new(new_plan)


def choice_point_new(partial_plan):
    # the generated plans will commit to the order of the actions, and only
    # make them strictier, different orderings are reached through different
    # choice points.
    gen_with_new_substitution = genplan_precond_substitutions(partial_plan)
    gen_with_new_action = genplan_new_action(partial_plan)

    return ChoicePoint(
        partial_plan,
        gen_with_new_substitution,
        gen_with_new_action,
    )


def expand_choice_point(choice_point):
    try:
        # - for every pending precondition find an action in the plan that has
        #   an effect
        # - check if the two actions can be unified
        # - if they can be unified add the new required effect to the indeces
        partial_plan = next(choice_point.gen_with_new_substitution)
    except StopIteration:
        pass
    else:
        return [partial_plan]

    try:
        # - satify one of the pending preconditions and remove it for the
        #   pending list
        # - add the action that satifies the precondition into the partial plan
        # - add the new action's preconditions into the the pending list
        partial_plan = next(choice_point.gen_with_new_action)
    except StopIteration:
        pass
    else:
        return [partial_plan]

    return
```

```
def rate(partial_plan):
    total_actions = 0
    total_pending = []

    for layer in partial_plan.action_layers:
        total_actions += len(layer)

        total_pending.insert(
            0,
            sum(
                len(action.id_to_pendingpreconditions)
                for action in layer.action_list
            )
        )

    # large numbers are deeper in the heap
    return (
        total_actions,
        total_pending,
    )


def initial_choice_point(domain, init, target):
    assert not target.id_to_effects, 'target action can not have effects'
    msg = 'target action must be grounded'
    assert set(target.substitutions.keys()) == set(target.operator.variables), msg

    initial_partial_plan = (
        action_layer_new((init,), EMPTY_DICT, EMPTY_DICT),
        action_layer_new((target, ), EMPTY_DICT, EMPTY_DICT),
    )

    id_to_action_by_layerpos = {
        id_: [[init], EMPTY_LIST]
        for id_ in init.id_to_effects
    }

    id_to_pendingaction_by_layerpos = {
        id_: [EMPTY_LIST, [target]]
        for id_ in target.id_to_pendingpreconditions
    }

    type_to_objects = {}
    for var, value in init.substitutions.items():
        pos = init.operator.variables.index(var)
        type_ = init.operator.variables_types[pos]
        type_to_objects.setdefault(type_, []).append(value)

    variable_assignments = {}
    choice = choice_point_new(partial_plan_new(
        domain,
        variable_assignments,
        type_to_objects,
        initial_partial_plan,
        id_to_action_by_layerpos,
        id_to_pendingaction_by_layerpos,
    ))

    return choice


# max_queue_size=2500000
def pnke_plan(domain, init, target, heuristic=rate, max_queue_size=1000000):
    initial_choice = initial_choice_point(domain, init, target)

    ranked_choice_points = PrioritySet()
    ranked_choice_points.add(
        heuristic(initial_choice.partial_plan),
        initial_choice,
    )

    number_of_totally_expanded_plans = 0
    number_of_generated_plans = 0
    start = time.time()

    if __debug__:
        print(initial_choice.partial_plan)

        while ranked_choice_points:
            new_choice_points = expand_choice_point(ranked_choice_points.peek())
```

```
            if new_choice_points is None:
                number_of_totally_expanded_plans += 1
                ranked_choice_points.get()
                print('.')
                continue

            for choice_point in new_choice_points:
                number_of_generated_plans += 1
                partial_plan = choice_point[0]
                print(partial_plan)
                if not partial_plan.id_to_pendingaction_by_layerpos:
                    elapsed_time = time.time() - start

                    return Result(
                        partial_plan.action_layers,
                        elapsed_time,
                        number_of_totally_expanded_plans,
                        number_of_generated_plans,
                        len(ranked_choice_points),
                    )

                ranked_choice_points.add(
                    heuristic(partial_plan),
                    choice_point,
                )

            if len(ranked_choice_points) > max_queue_size:
                return

    else:
        while ranked_choice_points:
            new_choice_points = expand_choice_point(ranked_choice_points.peek())

            if new_choice_points is None:
                number_of_totally_expanded_plans += 1
                ranked_choice_points.get()
                continue

            for choice_point in new_choice_points:
                number_of_generated_plans += 1
                partial_plan = choice_point[0]
                if not partial_plan.id_to_pendingaction_by_layerpos:
                    elapsed_time = time.time() - start

                    return Result(
                        partial_plan.action_layers,
                        elapsed_time,
                        number_of_totally_expanded_plans,
                        number_of_generated_plans,
                        len(ranked_choice_points),
                    )

                ranked_choice_points.add(
                    heuristic(partial_plan),
                    choice_point,
                )

            if len(ranked_choice_points) > max_queue_size:
                return


class PrioritySet(object):
    def __init__(self):
        self.heap = []
        self.set = set()

    def add(self, rank, item):
        heappush(self.heap, Rank(rank, item))
        if not item in self.set:
            heappush(self.heap, Rank(rank, item))
            self.set.add(item)

    def get(self):
        return heappop(self.heap).choice_point

    def peek(self):
        return self.heap[0].choice_point

    def __len__(self):
        return len(self.heap)
```

Arquivo `structures.py`

```
# -*- coding: utf-8 -*-
from collections import namedtuple
from functools import total_ordering

import six


EQ = 'eq'
NE = 'ne'
OBJECT = 'object'
NIL = 'nil'
EMPTY_LIST = []
EMPTY_DICT = {}

ChoicePoint = namedtuple(
    'ChoicePoint',
    (
        'partial_plan',
        'gen_with_new_substitution',
        'gen_with_new_action',
    )
)

# loc(a) = b
# Expression('loc', 'eq', 'a b', ['object', 'object'], 'a b')

Relation = namedtuple(
    'Relation',
    (
        'name',
        'variables',
        'variables_types',
    )
)

# Operator description, contains the data that don't change
Operator = namedtuple(
    'Operator',
    (
        'name',
        'variables',
        'variables_types',
        'required_relations',
        'preconditions_expr',
        'id_to_expressions',
    ),
)

Domain = namedtuple(
    'Domain',
    (
        'id_to_intantiable_operators',
    )
)

Result = namedtuple(
    'Result',
    (
        'plan',
        'elapsed_time',
        'number_of_totally_expanded_plans',
        'number_of_generated_plans',
        'number_of_waiting_plans',
    )
)



class Action(object):
    """ Represents an action. """
    __slots__ = (
        'id_to_effects',
        'id_to_pendingpreconditions',
        'object_substitutions',
        'operator',
        'substitutions',
    )

    def __init__(
```

```
            self,
            id_to_effects,
            id_to_pendingpreconditions,
            object_substitutions,
            operator,
            substitutions):

        self.id_to_effects = id_to_effects
        self.id_to_pendingpreconditions = id_to_pendingpreconditions
        self.object_substitutions = object_substitutions
        self.operator = operator
        self.substitutions = substitutions

    def __eq__(self, other):
        if isinstance(other, Action):
            # cannot compare using object_substitutions otherwise two different
            # partially instantiated actions will be considered equal.
            # self.object_substitutions == other.object_substitutions
            return (
                self.operator == other.operator and
                self.substitutions == other.substitutions
            )
        return False

    def __ne__(self, other):
        return not self.__eq__(other)

    def __hash__(self):
        value = (self.operator, self.object_substitutions)
        return hash(value)

    def __repr__(self):
        all_values = []
        for k in sorted(self.substitutions):
            pair = '{}={}'.format(
                k,
                self.substitutions[k]
            )
            all_values.append(pair)

        substitutions = ','.join(all_values)

        pending = ','.join(
            ('~' if precond.expr_type == NIL else '') + precond.name
            for precond_list in six.itervalues(self.id_to_pendingpreconditions)
            for precond in precond_list
        )
        if pending:
            pending = '[' + pending + ']'

        return '{name}{{{substitutions}}}{pending}'.format(
            name=self.operator.name,
            substitutions=substitutions,
            pending=pending,
        )


class ActionLayer(object):
    __slots__ = (
        'action_list',
        'id_to_preconditions',
        'acc_id_to_requireeffects',
    )

    def __init__(self, action_list, id_to_preconditions, acc_id_to_requireeffects):

        # the parallel actions that compose this layer
        self.action_list = action_list

        self.id_to_preconditions = id_to_preconditions

        # the effects that must not be negated for this and the next layers
        self.acc_id_to_requireeffects = acc_id_to_requireeffects

    def __len__(self):
        return len(self.action_list)

    def __repr__(self):
        return '[{actions}]'.format(
            actions=','.join(repr(action) for action in self.action_list),
        )
```

```
class Effect(object):
    __slots__ = (
        'assignment',
        'expression',
    )

    def __init__(self, assignment, expression):
        self.assignment = assignment
        self.expression = expression

    def __hash__(self):
        values = (self.assignment, self.expression.id)
        return hash(values)

    def __eq__(self, other):
        if isinstance(other, Effect):
            return (
                self.assignment == other.assignment and
                self.expression.id == other.expression.id
            )

        return False

    def __ne__(self, other):
        return not self.__eq__(other)

    def __repr__(self):
        return 'Effect({}, {})'.format(
            self.expression.name,
            self.assignment,
        )


class Expression(object):
    __slots__ = (
        'name',
        'expr_type',
        'id',
        'complement_id',
        'variables',
        'variables_types',
    )

    def __init__(
            self,
            name,
            expr_type,
            id_,
            complement_id,
            variables,
            variables_types):

        self.name = name
        self.expr_type = expr_type
        self.id = id_
        self.complement_id = complement_id
        self.variables = variables
        self.variables_types = variables_types

    def __repr__(self):
        if self.expr_type == EQ:
            arguments = ', '.join(self.variables[:-1])
            assignment = self.variables[-1]

            text = 'Expression<{}({})={}>'.format(
                self.name,
                arguments,
                assignment,
            )
        else:
            arguments = ', '.join(self.variables)
            text = 'Expression<{}({})=nil>'.format(
                self.name,
                arguments,
            )

        return text


@total_ordering
class Rank(object):
```

```
    __slots__ = (
        'rate',
        'choice_point',
    )

    def __init__(self, rate_, choice_point):
        self.rate = rate_
        self.choice_point = choice_point

    def __le__(self, other):
        return self.rate <= other.rate

    def __eq__(self, other):
        return (
            self.rate == other.rate and
            self.choice_point == other.choice_point
        )

    def __ne__(self, other):
        return not self.__eq__(other)


def normalize(variable_assignments):
    for k, v in variable_assignments.items():
        while v in variable_assignments:
            v = variable_assignments[v]
        variable_assignments[k] = v


class PartialPlan(object):
    __slots__ = (
        'domain',
        'variable_assignments',
        'type_to_objects',
        'action_layers',
        'id_to_action_by_layerpos',
        'id_to_pendingaction_by_layerpos',
        'assignment_items',
    )

    def __init__(
            self,
            domain,
            variable_assignments,
            type_to_objects,
            action_layers,
            id_to_action_by_layerpos,
            id_to_pendingaction_by_layerpos):

        normalize(variable_assignments)
        self.variable_assignments = variable_assignments
        self.assignment_items = tuple(variable_assignments.items())

        self.domain = domain
        self.type_to_objects = type_to_objects
        self.action_layers = action_layers
        self.id_to_action_by_layerpos = id_to_action_by_layerpos
        self.id_to_pendingaction_by_layerpos = id_to_pendingaction_by_layerpos

    def __repr__(self):
        return repr(self.action_layers)

    def __eq__(self, other):
        if isinstance(other, PartialPlan):
            # type_to_objects follows from the domain equality
            assert self.domain == other.domain

            # assume the indexes are consistent, they must be equal if the
            # action_layers are equal
            return (
                self.assignment_items == other.assignment_items and
                self.action_layers == other.action_layers
            )
        return False

    def __hash__(self):
        return hash((self.assignment_items, self.action_layers))
```

REFERÊNCIAS

ARISTÓTELES. **De Anima**. [S.l.: s.n.], –350.

ARISTÓTELES. **Organon**. [S.l.: s.n.], –350.

BLOSS, A.; HUDAK, P.; YOUNG, J. Code optimizations for lazy evaluation. **LISP and Symbolic Computation**, v. 1, n. 2, p. 147–164, Sep 1988. ISSN 1573-0557. Disponível em: <https://doi.org/10.1007/BF01806169>.

BOOLE, G. **The Laws of Thought**. [S.l.: s.n.], 1854.

BRACHMAN, R.; LEVESQUE, H. **Knowledge Representation and Reasoning**. [S.l.]: Morgan Kaufmann, 2004. (Morgan Kaufmann). ISBN 9781558609327.

DREPPER, U. **What Every Programmer Should Know About Memory**. 2007. Disponível em: <https://www.akkadia.org/drepper/cpumemory.pdf>.

EROL, K.; NAU, D. S.; SUBRAHMANIAN, V. S. When is planning decidable. In: **In Proc. First Internat. Conf. AI Planning Systems**. [S.l.: s.n.], 1992. p. 222–227.

EROL, K.; NAU, D. S.; SUBRAHMANIAN, V. S. Complexity, decidability and undecidability results for domain-independent planning. **Artificial Intelligence**, v. 76, p. 75–88, 1995.

FIKES, R. E.; NILSSON, N. J. **STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving**. 333 Ravenswood Ave, Menlo Park, CA 94025, 5 1971. SRI Project 8259. Disponível em: <http://www.ai.sri.com/pubs/files/tn043r-fikes71.pdf>.

GARDNER, H. **Nova Ciência da Mente, A — Uma História da Revolução Cognitiva Vol. 09**. [S.l.]: EDUSP, 1996. ISBN 9788531401459.

GHALLAB DREW MCDERMOTT, e. a. M. PDDL - The Planning Domain Definition Language. 10 1998.

GHALLAB, M.; NAU, D.; TRAVERSO, P. **Automated planning: theory & practice**. [S.l.]: Elsevier, 2004.

GOMES, L. **Hidden Obstacles for Googles Self-Driving Cars**. 8 2014. Disponível em: <http://www.technologyreview.com/news/530276/hidden-obstacles-for-googles-self-driving-cars/>.

GREEN, C. Application of theorem proving to problem solving. In: . Morgan Kaufmann, 1969. p. 219–239. Disponível em: <http://ijcai.org/Past%20Proceedings/IJCAI-69/PDF/023.pdf>.

KARPATHY, A.; JOULIN, A.; FEI-FEI, L. Deep fragment embeddings for bidirectional image sentence mapping. **CoRR**, abs/1406.5679, 2014. Disponível em: <http://arxiv.org/abs/1406.5679>.

KAUTZ, H.; MCALLESTER, D.; SELMAN, B. Encoding plans in propositional logic. In: . Morgan Kaufmann, 1996. p. 374–384. Disponível em: <http://www.cs.cornell.edu/selman/papers/pdf/96.kr.plan.pdf>.

KNUTH, D. E. **The Art of Computer Programming: Combinatorial Algorithms, Part 1**. 1st. ed. [S.l.]: Addison-Wesley Professional, 2011. ISBN 0201038048, 9780201038040.

KOREN, Y. **1 The BellKor Solution to the Netflix Grand Prize**. 2009.

LAIRD, J. E.; ROSENBLOOM, P. S. In pursuit of mind… the research of allen newell. **AI Mag.**, American Association for Artificial Intelligence, Menlo Park, CA, USA, v. 13, n. 4, p. 17–ff., dez. 1992. ISSN 0738-4602. Disponível em: <https://www.aaai.org/ojs/index.php/aimagazine/article/view/1019>.

LARRANAGA, P. et al. Genetic algorithms for the travelling salesman problem: A review of representations and operators. **Artificial Intelligence Review**, Kluwer Academic Publishers, v. 13, n. 2, p. 129–170, 1999. ISSN 0269–2821. Disponível em: <http://dx.doi.org/10.1023/A%3A1006529012972>.

MCCARTHY, J. Programs with common sense. In: **Semantic Information Processing**. [S.l.]: MIT Press, 1968. p. 403–418.

MCCARTHY, J.; HAYES, P. J. Some philosophical problems from the standpoint of artificial intelligence. **Readings in artificial intelligence**, Los Altos, CA: Morgan Kaufmann, p. 431–450.

MCCARTHY, J.; HAYES, P. J. Some philosophical problems from the standpoint of artificial intelligence. In: **Machine Intelligence**. [S.l.]: Edinburgh University Press, 1969. p. 463–502.

MCCARTHY, J.; LABORATORY, S. A. I. **Situations, actions, and causal laws**. [S.l.], 1963. (Memo (Stanford Artificial Intelligence Project)). Disponível em: <http://oai.dtic.mil/oai/oai?verb=getRecord&metadataPrefix=html&identifier=

MCCARTHY, J. et al. A Proposal for the Dartmouth Summer Research Project on Artificial Intelligence. **AI Magazine**, v. 27, n. 4, p. 12, 8 1955.

MCDERMOTT, J. H. D. Planning: What it is, what it could be, an introduction to the special issue on planning and scheduling. **Artificial Intelligence**, 1995.

MICHALEK, G. V. **Mind Models Artificial Intelligence Discovery At Carnegie Mellon**. 2001. Disponível em: <http://shelf1.library.cmu.edu/IMLS/MindModels/>.

MINSKY, M. Steps toward artificial intelligence. In: **Computers and Thought**. [S.l.]: McGraw-Hill, 1961. p. 406–450.

NEWELL, A.; SHAW, J. C.; SIMON, H. A. Report on a general problem-solving program. In: **IFIP Congress**. [S.l.: s.n.], 1959. p. 256–264.

NEWELL, A.; SIMON, H. A. The logic theory machine—a complex information processing system. **Information Theory, IRE Transactions on**, IEEE, v. 2, n. 3, p. 61–79, 1956. Disponível em: <http://shelf1.library.cmu.edu/IMLS/MindModels/logictheorymachine.pdf>.

PELLETIER, F. J. **A History of Natural Deduction and Elementary Logic Textbooks**. 2000. Disponível em: <http://www.sfu.ca/ jeffpell/papers/pelletierNDtexts.pdf>.

RUSSELL, S. J.; NORVIG, P. **Artificial intelligence: a modern approach**. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1995. ISBN 0–13–103805–2.

SACERDOTI, E. D. **A structure for plans and behavior**. [S.l.], 1975. Disponível em: <http://oai.dtic.mil/oai/oai?verb=getRecord&metadataPrefix=html&identifier=

SHANNON, C. E. **A Symbolic Analysis of Relay and Switching Circuits**. [s.n.], 1938. Disponível em: <http://hdl.handle.net/1721.1/11173>.

SUSSMAN, G. J. A computational model of skill acquisition. 1973. Disponível em: <http://hdl.handle.net/1721.1/6894>.

TATE, A. Interacting goals in problem solving. In: **AISB Newsletter**. [s.n.], 1974. v. 10, p. 31–38. Disponível em: <http://www.aiai.ed.ac.uk/project/oplan/documents/1990-PRE/1974-aisb-tate-interacting-goals.pdf>.

TURING. Computing machinery and intelligence. **Mind**, Oxford University Press, LIX, n. 236, p. 433–460, 1950. Disponível em: <http://dx.doi.org/10.1093/mind/LIX.236.433>.

WADLER, P. How to replace failure by a list of successes. In: **Proc. Of a Conference on Functional Programming Languages and Computer Architecture**. New York, NY, USA: Springer-Verlag New York, Inc., 1985. p. 113–128. ISBN 3-387-15975-4. Disponível em: <https://rkrishnan.org/files/wadler-1985.pdf>.

WANG, H. Toward mechanical mathematics. **IBM Journal of research and development**, IBM, v. 4, n. 1, p. 2–22, 1960.

WELD, D. S. An introduction to least commitment planning. **AI Magazine**, v. 15, p. 27–61, 1994.

WONG, G. **Baidus Andrew Ng on Deep Learning and Innovation in Silicon Valley**. 11 2014. Disponível em: <http://blogs.wsj.com/digits/2014/11/21/baidus-andrew-ng-on-deep-learning-and-innovation-in-silicon-valley/>.

PNKE: Uma estratégia para representação eficiente de planos de ordem parcial

Augusto Hack

[1]Sistemas de Informação – Universidade Federal de Santa Catarina (UFSC)

hack.augusto@grad.ufsc.br

Abstract. *Planning is the act of elaborating plans that achieve abstract goals. Automated planning is an A.I. research area with the goal of simulating rational thought on machines. Plans are computed by planners using search techniques, based on problem and domain descriptions. Planning is used to support human decision making or to create automated agents. A plan is a complete description of how to achive a goal, with the required actions and their dependecies.*

Resumo. *Planejamento automático é um área de estudo da inteligência artificial, com o objetivo de simular raciocínio em máquinas, permitindo que estas atinjam objetivos de maneira abstrata. Planejadores são capazes de elaborar planos a partir de descrições de domínios e problemas, onde quais ações e a ordem de execução necessária para que o objetivo seja atingido é descoberta pela máquina. Planejadores são construídos para auxiliar humanos, automatizar tarefas abstratas e como uma forma de estudo do raciocínio. Para que planos possam ser elaborados diversas técnicas foram desenvolvidas, todas elas baseadas em busca. A busca pode ser realizada em diferentes espaços e representações, é objetivo desse trabalho a construção de um planejador que faz a busca no espaço de planos.*

1. Introdução

Planejar é um processo natural aos humanos, um processo abstrato e deliberado onde ações são escolhidas a partir do seu resultado esperado. Planejamento, o ato de planejar, é essencial para resolver problemas complexos, onde para atingir um objetivo é necessário considerar as possíveis ações e suas interações, onde a ordem de execução é importante para que o resultado estipulado seja atingido.

Planejamento automático (*Automated Planning*) é uma área de pesquisa, onde criam-se algoritmos capazes de sintetizar planos de maneira independente de domínio. Planos são compostos por um conjunto de ações, que devem ser executadas por um ou mais agentes, em paralelo ou em sequência, que estima-se, atingirá o objetivo estipulado. Planejamento automático é um subcampo da Inteligência Artificial, que auxilia humanos na tomada de decisão em problemas complexos, ou automatiza a criação e execução de planos para agentes autônomos [Drew McDermott 1995].

Organização deste artigo:

- Introdução: Esta seção.

- Planejamento: Breve apresentação do problema de planejamento.
- Representação: Estrutura de dados proposta para representação de planos parciais.
- Conclusão: Notas finais.

2. Planejamento

Um modelo de transição é uma tupla de quatro elementos $\Sigma = \langle E, A, O, \gamma \rangle$, neste modelo O representa eventos não determinísticos que podem ocorrer no ambiente. Eventos são úteis para modelar sistemas dinâmicos, desnecessário para problemas clássicos onde assumem-se um sistema estático. Portanto para este trabalho será adotado uma versão restrito do modelo de transição definido como:

Definição 1. *Modelo de transição restrito Σ é a tripla*

$$\Sigma = \langle E, A, \gamma \rangle$$

Onde E representa o conjunto finito de estados possíveis e A é conjunto finito de ações do sistema, por fim γ é função de transição de estados.

Definição 2. *A função de transição mapeia o estado resultado após a aplicação de uma ação*

$$\gamma : E \times A \mapsto E$$

As ações possuem condições e efeitos, as condições representam em quais circunstâncias a ação pode ser executada, os seus efeitos representam o que será alterado no mundo após a sua aplicação.

Definição 3. *Efeito de uma ação é igual ao estado resultado com os efeitos negativos removidos e efeitos positivos adicionados*

$$\gamma(e, a) = (e \cap efeito^{-}(a)) \cup efeito^{+}(a)$$

A função de transição pode ser invertida, onde ao invés de definir o estado obtido pela aplicação de uma ação tem-se o estado onde a ação pode ser aplicada.

Definição 4. *O inverso da função transição é*

$$\gamma^{-1}(e, a) = (e \setminus efeito^{+}(a) \cup efeito^{-}(a))$$

O modelo de transição restrito é suficiente para descrever o domínio de um problema de planejamento, para um que o plano possa ser gerado, além dos detalhes do sistema de transição é necessário a descrição de uma instância deste domínio.

Definição 5. *Instância de um problema de planejamento é uma tripla*

$$\mathcal{P} = \langle \Sigma, e_0, e_o \rangle$$

Onde Σ é o modelo de transição restrito, e_0 é o estado inicial dessa instância e e_o o estado objetivo que deve ser atingido pelo o plano.

As ações podem ser relevantes ou aplicáveis, onde as ações relevantes satisfazem algum objetivo pendente do plano atual e ações aplicáveis possuem todos as suas precondições satisfeitas.

Ações relevantes permitem a elaboração de planos de trás para frente, a partir dos objetivos de um plano.

Definição 6. *Ações relevantes possuem um ou mais efeitos que contribuem de maneira positiva para que o objetivo seja atingido.*

Enquanto ações aplicáveis permitem a elaboração da frente para trás, a partir do estado atual de um plano.

Definição 7. *Ações aplicáveis possuem alguma transição válida a partir do estado considerado.*

A função Γ descreve o conjunto de possíveis estados sucessores.

Definição 8. *O conjunto de estados sucessores é obtido pela aplicação de todas as ações aplicáveis a um determinado estado*

$$\Gamma(e) = \{\gamma(e, a) | a \in \textit{ações aplicáveis em } e\}$$

Estados sucessores são computados pela aplicação sucessiva de ações válidas, $\Gamma^2 = \Gamma(\Gamma(e))$, $\Gamma^3 = \Gamma(\Gamma(\Gamma(e)))$, a união dos estados sucessores formam o conjunto transitivo contendo todos os estados que podem ser atingidos pela aplicação de ações.

Definição 9. *O conjunto de estados acessíveis é o conjunto transitivo de estados sucessores*

$$\hat{\Gamma} = \Gamma(e) \cup \Gamma^1(e) \cup \cdots \cup \Gamma^n(e)$$

Como o problema clássico de planejamento possui número finito de estados o conjunto $\hat{\Gamma}$ possui ponto fixo. Um problema de planejamento pode ser resolvido quando o conjunto de estados acessíveis contém o estado objetivo.

Definição 10. *Um problema é solucionável quando o estado objetivo está contido em* $\hat{\Gamma}$.

Esta definição formal é suficiente para descrever o planejamento clássico, a seguir serão introduzidas representações para computar soluções.

3. Representação

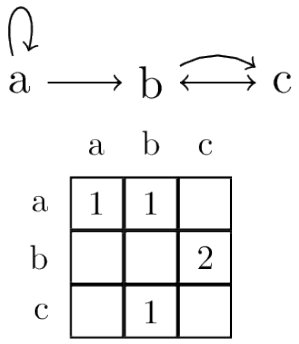

Figura 1. Representação matricial de um grafo

Grafos podem ser representados utilizando matrizes, onde estas são dispostas de forma contígua em memória. Nesta representação cada nó do grafo possui uma linha **e** uma coluna, i.e. cada linha e coluna da matriz é etiquetada pelos nós do grafo. As arestas partem do elemento representado pela linha para o elemento representado pela coluna, o valor de cada célula representa o número de arestas que conecta estes nós, como demonstrado pela figura 1.

Planos são grafos direcionados acíclicos, onde arestas não seguem na direção inversa e nós não podem ter auto referências, portanto apenas uma matriz triangular é suficiente para representar planos. Como as arestas utilizadas pelos planos representam ordem entre as ações, não há a necessidade para mais de uma aresta, dessa forma cada célula só precisa ter um bit de tamanho, setado para verdadeiro caso as ações estejam conectadas [Knuth 2011].

A representação baseada em matriz é uma melhoria em relação ao uso de objetos nós, no que diz respeito a localidade dos dados em memória, no entanto uma matriz triangular representa todas as possíveis arestas em um grafo, o que não é ótimo para grafos esparsos, resultando em uso excessivo memória. Como planos são grafos esparsos, para reduzir o desperdício de memória uma representação baseada em matriz esparsa foi utilizada.

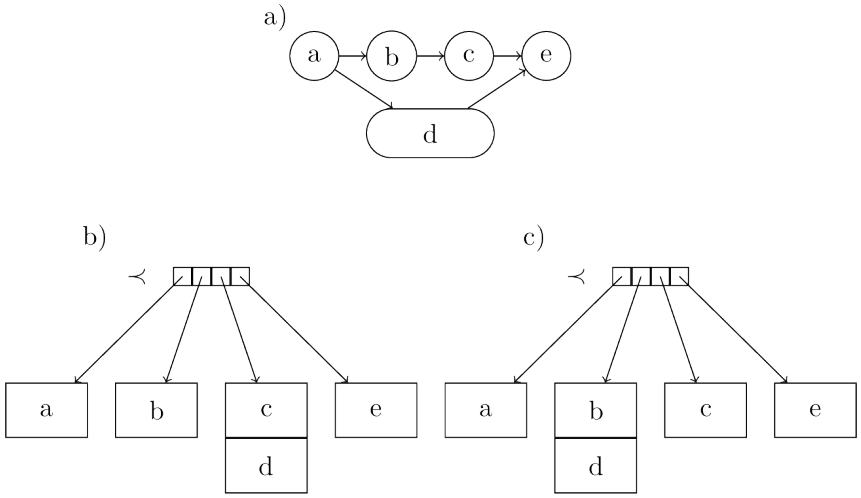

Figura 2. Representação esparsa de um plano parcial

Na representação esparsa adotada, representada pela figura 2, nós são agrupados em camadas. Em contraste com uma matriz triangular que mantém a ordem entre todos os nós do grafo, esta representação mantém ordem apenas entre **camadas**. A ordem é representada por uma lista, que mantém relação "ocorre antes de", onde os elementos na posição p ocorrem antes de alguma das camadas sucessoras.

Para exemplificar a invariante da estrutura repare na figura 2 o nó **d**, este nó não está ordenado em relação aos nós **b** e **c**, e portanto poderia ser adicionado tanto a camada que contém **b** ou **c**, preservando em ambos os casos as relações de ordem presentes no grafo original.

Essa representação não é fiel ao grafo original, já que nós com maior liberdade são forçados em camadas que com relações mais restritivas, mas esta estrutura é suficiente para representar a relação de dependências entre ações, necessárias para manter a consistência de um plano durante a busca.

4. Conclusão

Este artigo apresentou uma nova estrutura de dados que pode ser utilizada para representar planos parciais de forma eficiente, que também é adequada para

a indexição de dados, eliminando a necessidade de percorrer todo o grafo de ações para encontrar falhas.

A eficiência em termos de espaço desta estrutura dependerá da linguagem hospedeira e das técnicas utilizadas para codificar as estruturas, onde é importante tomar cuidado especial para o alinhamento correto da estrutura de dados.

Referências

Drew McDermott, J. H. (1995). Planning: What it is, what it could be, an introduction to the special issue on planning and scheduling. *Artificial Intelligence*.

Knuth, D. E. (2011). *The Art of Computer Programming: Combinatorial Algorithms, Part 1*. Addison-Wesley Professional, 1st edition.