Planejamento em Inteligência Artificial

Entendimento do processo de construção de agentes inteligentes, com foco na análise, planejamento e projeto de algoritmos de planejamento. Estudo de problemas clássicos de satisfação com restrições. Aplicação de ferramentas de programação em lógica na resolução desses problemas.

Prof. Vinícius Gomes Pereira

1. Itens iniciais

Propósito

Entender como desenvolver e estruturar agentes inteligentes é fundamental para o profissional de Inteligência Artificial. Assim, iremos apresentar modelos teóricos para construção de um agente que atua racionalmente e mostrar como o agente pode estruturar planos para realizar suas ações. Para isso, apresentaremos dois tópicos voltados a problemas de satisfação com restrições, com uma abordagem mais analítica e outra mais prática.

Preparação

Para a execução dos códigos em Prolog, é necessário instalar um ambiente de desenvolvimento como o SWI Prolog ou utilizar uma IDE on-line como a SWISH – SWI-Prolog for SHaring.

Objetivos

- Reconhecer as etapas para o desenvolvimento de um agente inteligente.
- Aplicar os algoritmos apresentados na resolução de problemas de satisfação com restrições.
- Aplicar programação em lógica na resolução de problemas de satisfação com restrições.
- Aplicar métodos de planejamento na resolução de problemas.

Introdução

Planejamento em Inteligência Artificial é uma área da computação que estuda o processo deliberativo, de modo a construir agentes racionais que realizam ações com base em um determinado objetivo, antecipando o que pode acontecer antes de agir.

Não estamos interessados em técnicas para projetar sensores, motores, atuadores mecânicos, por exemplo, mas sim em como modelar, avaliar e entender os problemas relacionados à construção desses agentes inteligentes, bem como elaborar e projetar algoritmos para isso.

Iniciaremos nosso estudo de modo mais conceitual, introduzindo conhecimentos e conceitos necessários a respeito do planejamento de agentes em IA. A seguir, estudaremos técnicas mais práticas voltadas à resolução de problemas gerais. Ao final, retornaremos ao planejamento de agentes e introduziremos novos algoritmos específicos ao assunto.

Modelo conceitual de planejamento

Neste vídeo, exploraremos o modelo conceitual de planejamento em Inteligência Artificial (IA). Vamos analisar como os sistemas de IA tomam decisões estratégicas, desde a definição de objetivos até a geração de planos de ação.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Para criar um modelo conceitual de planejamento, é necessário ter embasamento e teoria necessários para representar e arquitetar agentes inteligentes que compreendem o ambiente e executam ações, com um determinado objetivo. Por isso, é necessário estudar como um agente artificial raciocina, escolhe e organiza suas ações, agindo deliberadamente para atingir um determinado objetivo, e, assim, representar matematicamente todos esses elementos.

Portanto, é crucial entender como nós podemos modelar conceitualmente o ambiente, as ações, os objetivos, os agentes, de forma que um agente artificial consiga com alto grau de autonomia executar metas especificadas.

O que é deliberação?

O que seria uma ação de um agente artificial? O que seria uma ação deliberativa?

Neste vídeo, exploraremos o papel dos agentes de IA e suas deliberações na tomada de decisões inteligentes. Vamos discutir como os agentes de IA são projetados para deliberar, avaliar alternativas, considerar objetivos e restrições, além de adaptar seu comportamento de acordo com o ambiente.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Uma ação é algo que um determinado agente faz (uma força, um movimento, uma percepção), que provoca mudanças no ambiente ou no seu próprio estado.

> Uma ação deliberativa é uma ação que é feita com um determinado objetivo.

Seres humanos agem deliberadamente. Uma pessoa, quando cozinha, compra antes os alimentos no supermercado, limpa, prepara e realiza diversas ações intermediárias para que alguns minutos depois consiga comer. Diversas ações são feitas para que objetivos intermediários sejam atingidos e, no fim, a meta final de comer é realizada.

Dessa forma, duas questões são importantes:

1. Se um agente performar uma ação, qual seria o resultado?

2. Como combinamos ações para produzir um determinado efeito?

Os mecanismos de agir com deliberação são tópicos de estudo de diversas áreas de pesquisa, como Psicologia, Biologia, Neurociência, Filosofia e Ciências Cognitivas, por exemplo. Em particular, o pensamento deliberativo avançado do ser humano é muito estudado, principalmente em bebês e crianças.

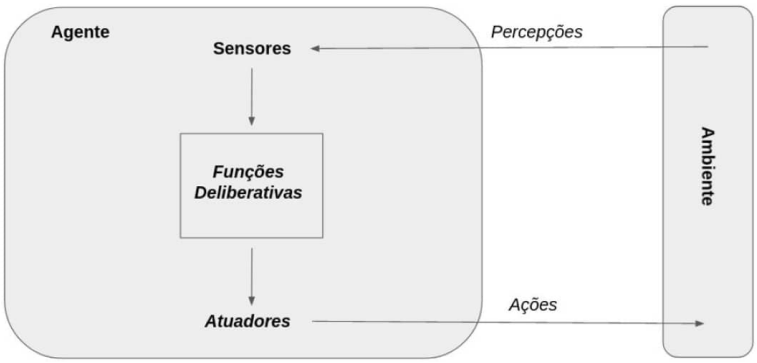
Comentário

O nosso foco de estudo será como um agente artificial performa ações deliberativas em um ambiente. Assim, queremos construir agentes racionais que elaboram um conjunto de ações corretas – ou que inferem como corretas baseados nas percepções de seus mundos – visando atingir seus objetivos. Por conseguinte, o escopo do nosso estudo é como arquitetar esses agentes inteligentes.

Um agente artificial desenvolvido age deliberadamente em um determinado grau de autonomia, diversidade das ações e ambientes. Quanto mais autonomia, menos interações humanas serão necessárias. E, quanto mais diferentes tipos de ações em diferentes cenários, com mais diversidade esse agente inteligente conseguirá atuar. Um robô autônomo que produz parafusos, por exemplo, não age com deliberação, por conseguir agir somente em um ambiente completamente especificado previamente. Contudo, um robô autônomo que foi projetado para agir em uma gama de tarefas e ambientes deverá ser planejado com algum nível de deliberação.

Modelo conceitual de um agente

Vamos definir **agente** como qualquer entidade que percebe o ambiente por meio de sensores e age por meio de atuadores. A imagem a seguir representa essa definição.



O agente interage com o ambiente por meio de atuadores, recebendo percepções por meio de sensores.

As funções deliberativas concretizam a ideia de deliberação que discutimos. Há um estado inicial e um final que devemos atingir por meio de várias ações.

> Uma ideia básica e inicial de como as funções deliberativas dos agentes podem ser projetadas é por meio de tabelas que mapeiam qualquer tipo de estado percebido pelo ambiente (inicial) a uma sequência de ações que o agente deve tomar para atingir um outro estado (final ou intermediário).

Para a maioria dos casos é infactível projetar esses tipos de agentes baseados em tabela, pois essas possibilidades de estados e sequências de ações podem ter um tamanho muito grande, inviabilizando a busca, armazenamento ou a construção da tabela.

Um agente precisa de uma medida de desempenho para saber se desempenhou bem ou não uma determinada ação e para, de alguma forma, buscar maximizar esse desempenho. Haverá, em nosso modelo, uma medida de desempenho para avaliar as sequências de estados do ambiente.

Assim, em qualquer momento, o agente deve saber:

a) A medida de desempenho que será um critério de sucesso;

b) O conhecimento prévio do agente sobre o ambiente;

c) As ações que o agente pode performar;

d) As percepções do agente pelo tempo.

No projeto de agentes racionais devemos especificar:

- A medida de desempenho (**Performance**).
- O ambiente (**Environment**).
- Os atuadores (**Actuators**).
- Os sensores (**Sensors**).

Essas especificações são chamadas de **PEAS** pelo acrônimo formado.

O quadro a seguir ilustra alguns exemplos de sistemas inteligentes e suas respectivas especificações PEAS.

Tipo de agente	Medida de desempenho	Ambiente	Atuadores	Sensores
Professor de inglês interativo	Pontuação do estudante em um teste	Conjunto de estudantes e sala de aula	Display de exercícios, aulas, correções etc.	Teclado de digitação
Robô de Limpeza	Tempo para limpeza, porcentagem dos cômodos limpos, nível de limpeza	Cômodos	Rodas, braço mecânico, sugadores etc.	Câmera, sensores de presença
Sistema de diagnóstico médico	Número de pacientes saudáveis, redução de custos	Hospital, paciente, médicos, enfermeiros etc.	Display de questões, testes, tratamentos, diagnósticos, redirecionar para um médico especialista	Teclado de digitação, respostas dos pacientes
Sistema de análise de imagens de satélite	Categorização correta da imagem	Conexão com o satélite em órbita	Display da categorização da cena	Pixels de cores da imagem

Quadro: Especificação PEAS de diversos tipos de agentes.
Vinícius Gomes Pereira.

Modelo conceitual de um ambiente

Há um elevado número de tipos de ambientes das mais variadas tarefas em IA. Entretanto, podemos categorizar um pequeno número de ambientes para o agente apropriado ser projetado. Assim, classificaremos o ambiente da seguinte forma:

Totalmente observável ou parcialmente observável

Se o sensor de um agente tem acesso ao estado completo do ambiente, em qualquer momento, dizemos que o ambiente é totalmente observável. Ambientes totalmente observáveis são interessantes por não ser necessário armazenar dados internos para saber o estado do ambiente. Caso haja algum tipo de informação do ambiente relevante que os sensores não conseguem captar, o ambiente é parcialmente observável.

Agente único ou multiagente

Se em um ambiente há um único agente atuando, isto é, somente um que pode mudar os estados do ambiente, dizemos que o ambiente é de agente único. Caso haja mais de um, dizemos que o ambiente é multiagente. O projeto de agentes em um ambiente multiagente é bem diferente de um ambiente com agente único.

Determinístico ou estocástico (não determinístico)

Se o próximo estado do ambiente é completamente determinado pelo estado atual e pelas ações executadas pelo agente, dizemos que o ambiente é determinístico. Caso contrário, é estocástico.

Estático ou dinâmico

Se o ambiente pode mudar enquanto o agente está em processo de deliberação, dizemos que o ambiente é dinâmico. Caso contrário, o ambiente é estático.

Discreto ou contínuo

Se há um número limitado de percepções e ações, dizemos que o estado do ambiente é discreto e, caso contrário, dizemos que o estado do ambiente é contínuo. Um robô autônomo que dirige um carro tem em cada estado do ambiente um número ilimitado de percepções e ações que podem ser feitas. Em um jogo de xadrez, há um número fixo de possíveis movimentos em cada jogada e um número fixo de posições de peças, o que torna o estado do ambiente discreto.

Episódico ou sequencial

Em um ambiente cuja ação do agente é episódica, a experiência do agente é dividida em episódios atômicos. Ou seja, a cada episódio, o agente percebe o ambiente e age sem depender das ações tomadas anteriormente. Em ambientes sequenciais, a decisão atual pode afetar todas as ações futuras, como, por exemplo, um robô autônomo que joga xadrez ou dirige um carro. Ambientes episódicos são muito mais simples que os sequenciais pelo fato de o agente não se preocupar com as ações tomadas futuramente.

No quadro a seguir, para os mesmos exemplos que fizemos a especificação PEAS, classificamos os ambientes de acordo com sua natureza.

Tipo de agente	Ambiente					
	Observável	Agentes	Determinístico	Episódico	Estático	Discreto
Professor de inglês interativo	Completamente observável	Multiagente	Estocástico	Sequencial	Dinâmico	Discreto
Robô de limpeza	Parcial	Único	Estocástico	Episódico	Dinâmico	Contínuo
Sistema de diagnóstico médico	Parcial	Único	Estocástico	Sequencial	Dinâmico	Contínuo

Tipo de agente	Ambiente					
	Observável	Agentes	Determinístico	Episódico	Estático	Discreto
Sistema de análise de imagens de satélite	Completamente observável	Único	Determinístico	Episódico	Semi	Contínuo

Quadro: Classificação do ambiente de acordo com sua natureza.
Vinícius Gomes Pereira

Modelo conceitual

Neste vídeo, veja a apresentação do conceito de sistemas de transição de estado e como realizar a representação por meio de grafos.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Até então, foram expostos os componentes de um agente e sua relação com o ambiente. É necessário desenvolver uma representação matemática para desempenhar as funções deliberativas.

> O modelo conceitual é um esquema teórico para descrever elementos de um problema.

Este modelo é útil para elucidar o problema que estamos resolvendo, assim como para levantar suposições e estruturar requisitos. O modelo conceitual que estudaremos são os **sistemas de transição de estado**.

Sistemas de transição de estado

Um sistema de transição de estados é uma 4-tupla $\Sigma = (S,A,E,\gamma)$, onde:

- $S = \{s_1, s_2, \ldots\}$ é finito ou recursivamente enumerável conjunto de estados.
- $A = \{a_1, a_2, \ldots\}$ é finito ou recursivamente enumerável conjunto de ações.
- $E = \{e_1, e_2, \ldots\}$ é finito ou recursivamente enumerável. É um conjunto de eventos exógenos, não controlados pelo agente.
- $\gamma = Sx(AUE) \rightarrow 2^S$ é uma função de transição de estados.

Comentário

A função de transição de estados mapeia os estados do sistema, com uma ação ou um evento aplicável a esse estado, a um novo conjunto possível de estados. É um mapeamento de como eventos e ações acontecendo em um determinado momento podem mudar o ambiente.

Se a é uma ação do conjunto de ações A, e essa ação pode ser aplicada em um determinado estado s, do conjunto de estados S, e, ao aplicar essa ação a, o sistema irá para o estado s_2, dizemos que a é aplicável em s. Matematicamente:

Se $a \in A, e, \gamma(s,a) \neq \varnothing$, então a é aplicável em s. Além disso, $s_2 \in \gamma(s,a)$.

Representação do sistema de transição de estados por grafos

Um sistema $\Sigma = (S, A, E, \gamma)$ de transição de estado pode ser representado por meio de um grafo direcionado $G = (N_g, E_g)$, onde:

- Os nós representam os estados de S. Ou seja $N_g = S$.
- Existe um arco que $s \rightarrow s'$ onde $s \in S, s' \in S$ e $s \rightarrow s' \in E_g$ com label $u \in (AUE)$, se somente se, $s' \in \gamma(u, s)$.

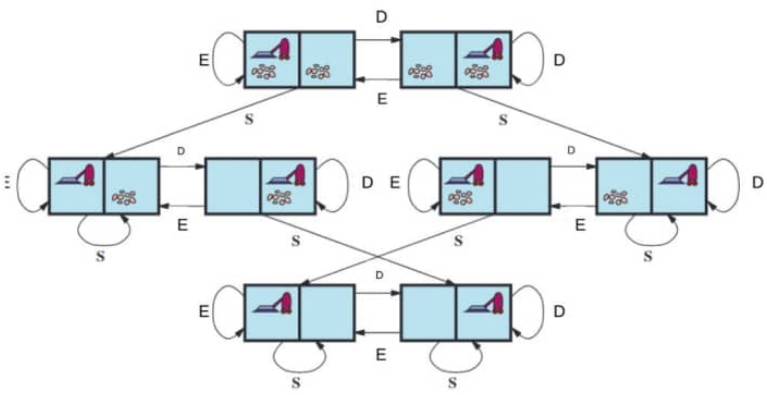
Comentário

Desse modo, o grafo do sistema de transição de estados consegue ilustrar como esse sistema de transição de estados se comporta e pode evoluir.

Na imagem a seguir, ilustramos um grafo de transição de estados para o problema do robô limpador.

Há dois cômodos em que o robô deve limpar e há três operações a serem feitas:

1. **E** representando que o robô deve ir para o cômodo da Esquerda e **D**, para o da Direita;
2. **S** representando que ele deve sugar a sujeira;
3. Cada nó do grafo é uma imagem que representa um determinado estado. **D**, **E** e **S** representam as ações a serem tomadas.



O grafo de transição de estados para o robô limpador. Há exatamente 8 estados e 3 ações a serem tomadas: D = Direita, E = Esquerda e S = Sugar.

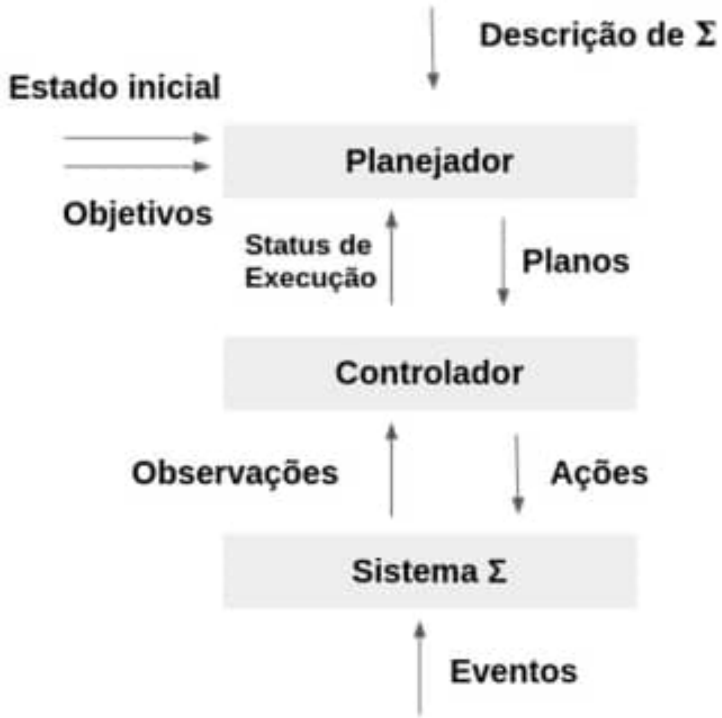
Assim, um sistema de transição de estados descreve todas as maneiras pelas quais o sistema pode evoluir. Se S e A são conjuntos pequenos, pode ser factível construir um agente tabela, em que há o mapeamento entre ação e estado, por meio de tabelas, que representam de forma tabular a função $\gamma(s,a)$.

Modelo conceitual para planejamento

Um plano é uma estrutura que fornece ações apropriadas visando atingir algum objetivo a partir de um estado inicial. Esse objetivo pode ser único, representado por $s_g \in S$, ou pode ser um conjunto $S_g \subset S$.. Um objetivo é atingido por uma sequência de transição de estados até o estado final.

O controlador, dado um input do estado atual *s* do sistema, fornece como saída a ação de acordo com o plano.

O planejador, dada a descrição do sistema, uma situação inicial e algum objetivo, sintetiza um plano para o controlador. Dessa forma, o diagrama da imagem a seguir sintetiza essas relações em um modelo conceitual para o planejamento:



Modelo conceitual para planejamento.

Suposições restritivas e planejamento clássico

Sobre o sistema $\sum$ $, podemos assumir algumas suposições restritivas, de modo que conseguimos melhorar a modelagem sobre o problema e elaborar planejadores mais específicos. O quadro a seguir contém oito suposições restritivas sobre o sistema:

Suposição restritiva	Descrição
A0	Sistema Σ tem um número finito de estados.
A1	Sistema Σ é completamente observável.

Suposição restritiva	Descrição
A2	Sistema Σ é determinístico.
A3	Sistema Σ é estático.
A4	O planejador só lida com metas restritas que são explícitas e conhecidas. Não há estados intermediários a serem evitados.
A5	Um plano é uma sequência linear finita de ações.
A6	Ações e eventos não têm duração. A transição de estado é instantânea.
A7	O planejador não se preocupa em qualquer mudança que pode acontecer no ambiente, enquanto estiver planejando (off-line planning).

Quadro: Suposições restritivas e a respectiva descrição.

Vinícius Gomes Pereira.

Planejamento clássico requer todas essas oito suposições: conhecimento completo sobre um sistema determinístico, estático, estado-finito com satisfação de metas e tempo implícito.

Dessa forma, o planejamento clássico se reduz ao seguinte problema:

Dado um problema de planejamento $P = (\Sigma, s_i, S_g)$, onde

- $\Sigma = (S, A, \gamma)$ é um sistema de transição de estados.
- $s_i \in S$ é o estado inicial.
- $S_g \subset S$ é o conjunto de metas.

Achar uma sequência de ações $(a_1, a_2, \ldots a_k)$, que corresponde a uma sequência de transições de estado $(S_1, S_2, \ldots S_k)$, tal que:

$$s_1 = \gamma(s_i, a_1), s_2 = \gamma(s_1, a_2), \ldots s_k = \gamma(s_{k-1}, a_k), \text{ sendo } s_k \in S_g$$

Em um problema de planejamento clássico, muitas vezes conseguimos estruturar como uma busca em um grafo do sistema de transição de estados. Não é um problema trivial pela quantidade de estados, eventos e ações que podem ser tomadas.

O planejamento clássico restringe muito a modelagem do sistema. Assim, dadas as oito restrições, modelos menos restritos são criados, de modo que alguma restrição não seja cumprida, o que denominamos como relaxação. Relaxar uma restrição é, portanto, não a cumprir. Planejadores específicos para diferentes combinações das relaxações das restrições são elaborados. Descrevemos em que situações uma relaxação da restrição pode ser considerada:

Suposição Restritiva	Relaxar a restrição
A0	Quando queremos modelar estados não discretos, mas sim contínuos. Quando as ações constroem ou trazem novos objetos para o ambiente.

Suposição Restritiva	Relaxar a restrição
A1	Quando o ambiente não é completamente conhecido.
A2	Quando a ação pode ter múltiplos estados de saída.
A3	Quando eventos exógenos podem acontecer.
A4	Quando não há um objetivo definido, mas uma função utilidade, por exemplo. Quando há restrições para um determinado estado ser atingido.
A5	Quando o sistema é dinâmico.
A6	Quando as ações têm uma determinada duração, ou há concorrência de ações e possíveis deadlines.
A7	Quando pode haver mudanças no ambiente enquanto há o planejamento.

Quadro: Suposições restritivas e respectivas situações exemplo para relaxação.
Vinícius Gomes Pereira.

Verificando o aprendizado

Questão 1

Considere um agente inteligente desenvolvido para explorar a superfície de Titan, uma lua de Saturno. Marque a alternativa verdadeira a respeito do tipo de ambiente.

A

O ambiente é totalmente observável.

B

O ambiente é estático.

C

O ambiente é discreto.

D

O ambiente é estocástico.

E

O ambiente é determinístico.

A alternativa D está correta.

O ambiente estocástico é aquele em que o estado do ambiente não é totalmente determinado pelas ações do agente! Desse modo, há diversos eventos exógenos que podem impactar o estado do ambiente.

Questão 2

Assinale a afirmativa verdadeira a respeito do planejamento de agentes inteligentes em Inteligência Artificial.

A

Para o problema de um robô que foi desenvolvido para explorar a superfície da lua, um agente baseado em tabela seria adequado, considerando que o ambiente é contínuo.

B

Desenvolver melhores sensores que percebem fielmente o ambiente, tornando-os completamente observáveis, resulta em planejadores mais simples.

C

O planejamento clássico atende todos os tipos de problemas no desenvolvimento de um agente inteligente.

D

O planejamento clássico assume o conhecimento completo sobre um sistema determinístico, estático, estado-finito com satisfação de metas e tempo implícito.

E

Um agente baseado em tabela é o mais adequado quando o número de estados é finito.

A alternativa D está correta.

É a definição exata de planejamento clássico. No planejamento clássico, assumimos as oito restrições do modelo restritivo.

Problemas de satisfação com restrições

Agora vamos estudar algoritmos e problemas de satisfação com restrições, que são muito comuns no planejamento de agentes inteligentes. Trataremos de forma genérica, não necessariamente aplicada ao planejamento de Inteligência Artificial. Vamos nos concentrar em **problemas de satisfação com restrições**, nosso objetivo principal. Dessa forma, conseguimos aplicar o conceito em qualquer tipo de problema, independentemente do escopo da área.

Neste vídeo, mergulharemos nos algoritmos e problemas de satisfação com restrições em Inteligência Artificial (IA) e sua importância nas deliberações inteligentes. Exploraremos conceitos como modelagem de problemas, restrições, variáveis e domínios, além de algoritmos de busca e otimização para resolver problemas.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Revisão de lógica proposicional

Iremos rever lógica booleana proposicional para introduzir o problema base de satisfação booleana (**SAT**). Esta seção também servirá como suporte para outros módulos seguintes.

Um literal é tanto uma variável proposicional ou uma negação de uma variável proposicional. As variáveis **x** e **y** são chamadas de literais positivas, enquanto **¬x** e **¬y** são literais negativas. As variáveis x e y podem assumir os valores de verdadeiro ou falso, e as suas respectivas literais negativas, os valores contrários.

1. $x \wedge y$ se x e y são verdadeiros. 0 símbolo $\wedge$ significa e. Algumas vezes, escrevemos e ou usamos o símbolo $\wedge$;
2. $x \vee y$ é verdadeiro se algum dos valores de x ou y for verdadeiro, inclusive os dois valores sendo verdadeiros juntos. 0 símbolo $\vee$ significa ou. Algumas vezes, escrevemos ou, outras usamos o símbolo V ;
3. $\neg x$ é verdadeiro se x for falso. Se x for verdadeiro, $\neg x$ é falso. Lemos $\neg x$ como $n\tilde{a}o x$
4. $x \Rightarrow y$ é verdadeiro se x for falso ou se y for verdadeiro. $x \Rightarrow y$ é lido como x implica y. Por definição $x \Rightarrow y$ é $(\neg x) \vee y$.
5. $x \Leftrightarrow y$ significa $(x \Rightarrow y) \wedge (y \Rightarrow x)$. Dizemos que x e y são equivalentes lógicos.

Uma cláusula é uma disjunção (operador ou, representado por $\vee$) de um ou mais literais. Como exemplo, temos:

1

$x \vee y \vee z$ é uma cláusula que se lê x ou y ou z.

2

$x \vee \neg y \vee z$ é uma outra cláusula, porém usando operador de negação $\neg$.

Uma cláusula fórmula (chamada simplesmente de fórmula) é uma conjunção (operador e, representado por $\wedge$) de uma ou mais cláusulas. Como exemplo, temos:

> $(x \vee y \vee z) \wedge (x \vee \neg y \vee z)$ é uma fórmula, porque é a conjunção ($\wedge$) da cláusula ($x \vee \neg y \vee z$) com a cláusula $(x \vee \neg y \vee z)$.

Uma fórmula ϕ é satisfazível se existe uma atribuição de valores a suas variáveis que faz com que a fórmula seja verdadeira.

Considere a fórmula:

$$\varphi = (x \vee y \vee z) \wedge (x \vee \neg y \vee z)$$

Resumindo

Ela é satisfazível para os valores de x = Verdadeiro, y = Falso e z = Falso. Assim como ela é satisfazível para x = Verdadeiro, y = Verdadeiro e z = Verdadeiro.

O problema de achar atribuições de valores booleanos às variáveis de modo a tornar verdadeira uma fórmula é chamado de **SAT**. Se não houver valores para as variáveis, de modo que a fórmula nunca se torna verdadeira, dizemos que o problema é não satisfazível.

Comentário

SAT é um problema dificílimo. SAT é o primeiro problema a ser provado como NP-Completo. Ainda não há algoritmo que resolva SAT de forma eficiente e talvez não exista. Resolver o problema SAT de forma eficiente (em tempo polinomial) implicaria em uma revolução no mundo da computação (e no mundo de forma geral).

Diversos problemas podem ser reduzidos a SAT. Se um problema é redutível a SAT, esse problema também se torna **NP-Completo**. SAT é o problema de satisfação base e há diversos algoritmos que o resolvem (não de forma eficiente em relação à complexidade), aplicando diversas heurísticas.

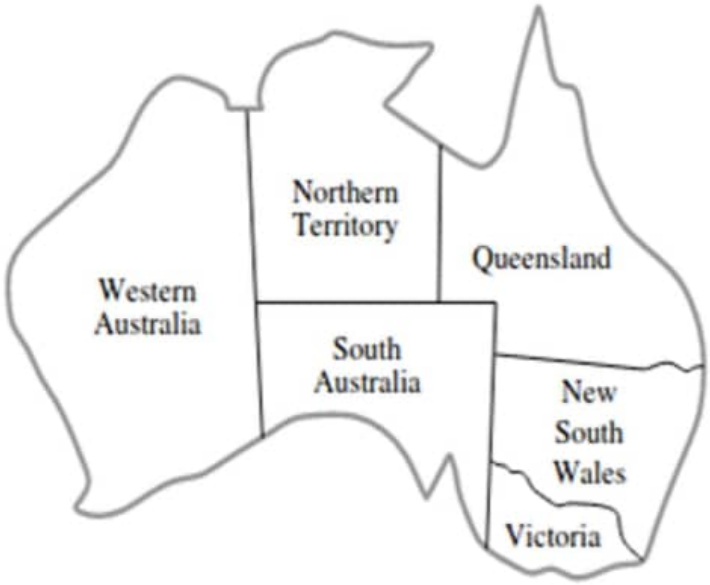
Problema de satisfação com restrições

Quando impomos condições explícitas para um problema de satisfação, o classificamos como **Problema de Satisfação com Restrições (PSR)**. É uma classe mais genérica de problemas. Mas muitos problemas podem ser reduzidos a resolver um problema SAT somente.

Problema de coloração de mapas

Para ilustrar, considere o seguinte problema:

Queremos colorir o mapa da imagem a seguir com três cores (azul - **B**, vermelho - **R** e verde - **G**) de modo que regiões adjacentes tenham cores distintas. O mapa da Austrália (o primeiro) foi transformado na imagem que vem após a dele.



Coloração de mapa com três cores e regiões adjacentes com cores distintas.

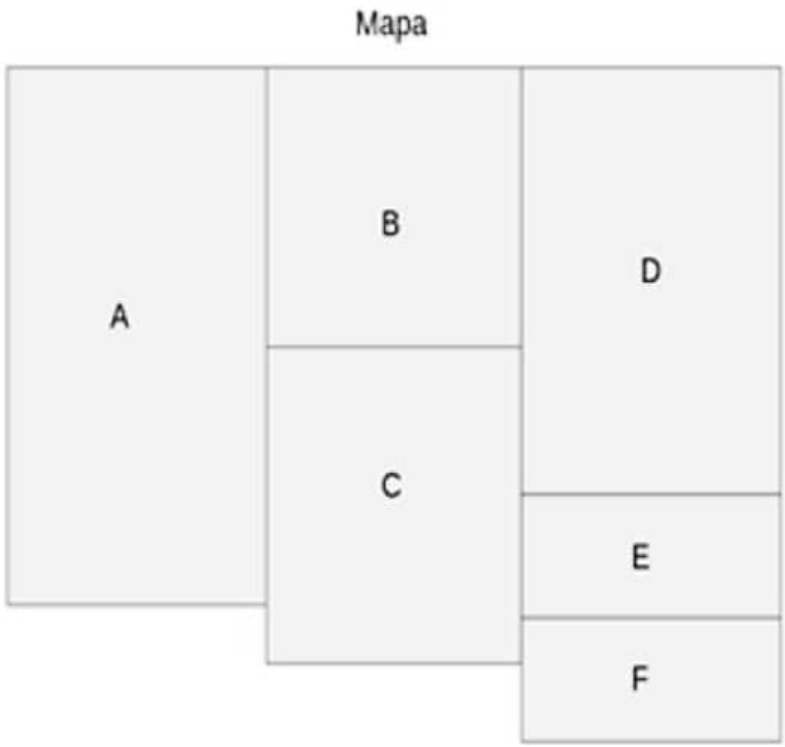


Imagem resultante da transformação do mapa da Austrália.

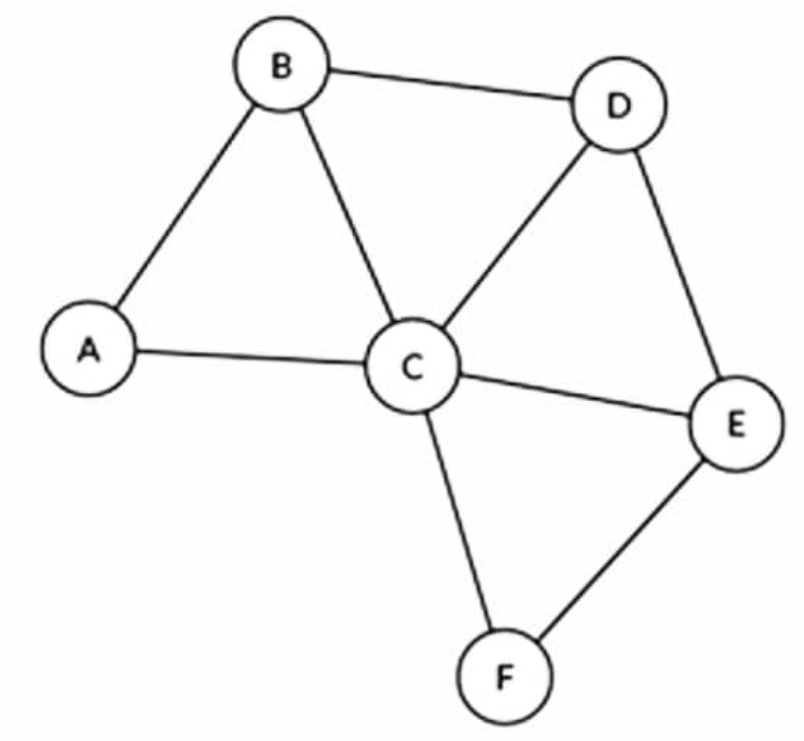
Escrevemos o problema de coloração da seguinte forma:

Variáveis: A, B, C, D, E, F

Domínios: $D_A = D_B = D_C = D_D = D_E = D_F = \{R; G; B\}$

Restrições: $A \neq B, A \neq C, B \neq C, B \neq D, C \neq D, C \neq E, C \neq F, D \neq E, E \neq F$

Esse problema pode ser representado por um grafo não direcionado de restrições, como na imagem a seguir.



Problema de coloração de regiões representada por um grafo de restrições. Cada aresta do grafo liga duas cidades que possuem cores diferentes.

O problema de achar o número mínimo de cores para colorir um grafo de modo que vértices conectados tenham cores diferentes é NP completo.

É fácil provar que um grafo é k-colorível (se conseguimos colorir com k cores). Basta exibir o grafo colorido! Mas como provar que o grafo não é k-colorível?

Coloração de grafos pode ser redutível a resolver um problema SAT. Reduzir a SAT significa elaborar esse problema como um **problema de satisfação booleano**.

Comentário

Há diversos outros problemas interessantes de satisfação com restrições, como o problema das oito rainhas (queremos dispor oito rainhas no tabuleiro de xadrez, sem que uma ataque a outra), o famoso jogo Sudoku, entre outros jogos.

Assim, de forma geral, um problema de satisfação com restrições é definido matematicamente da seguinte forma:

- X é um conjunto de k variáveis: $X = \{x_1, x_2, \ldots, x_k\}$
- Cada variável tem um domínio. Seja D o conjunto de domínios: $D = \{D_1, D_2, \ldots, D_k\}$, onde $x_k \in D_k$
- Há um conjunto de C com m restrições: $C = \{c_1, c_2, \ldots, c_m\}$
- Queremos achar valores para X: Valores que satisfaçam o conjunto de restrições e ao domínio.

As variáveis podem assumir valores inteiros, não inteiros ou booleanos, como também valores reais, por exemplo. O caso mais simples de problema de satisfação com restrições é quando o conjunto de variáveis é discreto e envolve domínios finitos, como o problema exposto de coloração de mapas. As restrições podem ser de diferentes naturezas:

1
Unárias
Se cada restrição envolve somente uma variável. $x_1 \neq 1, x_2 gt; 3$, por exemplo, por exemplo

2
Binárias
Se há restrições que envolvem até duas variáveis. $x_1 + x_2 gt; 5, x_3 \neq x_4$, por exemplo.

3
Ordem superior
Se há restrições que envolvem mais de duas variáveis.

Abordaremos PSR, quando os domínios são discretos e finitos. Assim, introduziremos diversos algoritmos que resolvem o problema por meio de buscas.

Resolvendo problemas por meio de buscas

Neste vídeo, vamos mergulhar no algoritmo de busca Backtracking e sua aplicação na resolução de problemas complexos em Inteligência Artificial. Vamos explorar como o Backtracking Search funciona, seu processo de exploração de soluções, técnicas de otimização e estratégias para lidar com restrições.

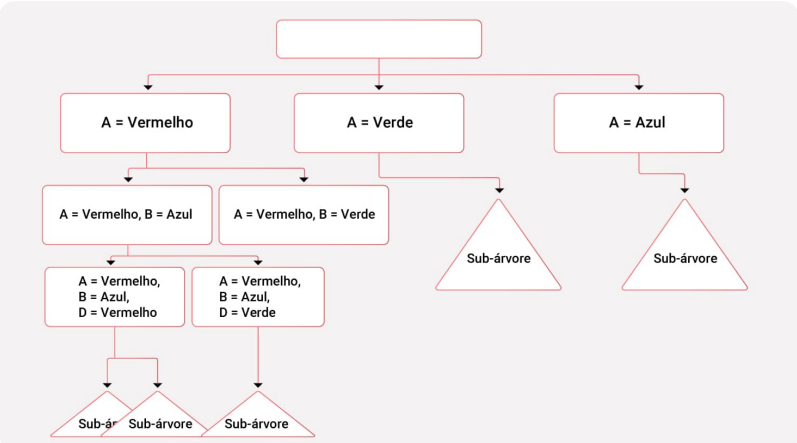
Conteúdo interativo
Acesse a versão digital para assistir ao vídeo.

O algoritmo base para a solução desse problema que introduziremos é a busca retroativa (backtracking search). A busca retroativa genérica é estruturada como uma busca em profundidade.

Uma árvore de possibilidades é formada, de modo que, em cada nível, uma atribuição de valor a uma variável é realizada. Se não houver como atribuir valores a uma variável, devido ao conjunto de restrições, retornamos para um nível acima da árvore de busca. Se chegarmos em um nível em que todas as variáveis apresentarem atribuições válidas, considerando as restrições, uma solução foi encontrada e a busca é cessada. O pseudocódigo a seguir implementa a busca retroativa para esse problema.

```
plain-text

function BacktrackingSearch(level):
 # Se todas as variáveis já foram atribuídas
 if all variables are assigned
 return Value
 #Escolhemos una variável que não tem atribuição de valor a ela
 V = PickUnassignedVariable()
 #Para a variável V, o nível level da busca será com essa variável V
 Variable[level] = V
 # E marcaremos V com label true, indicando que será feita a atribuição a ela.
 Assigned[V] = True
 #Olharemos todo domínio de V (os valores que V pode assumir)
 for d in Domain(V):
 #Esse valor for consistente com todas as atribuições já feitas
 if d is a consistent assignment according to CONSTRAINTS
     Value[V] = d
     result = BacktrackingSearch(level+1)
     # Se essa nova atribuição näo retornar falha
     if result != failure
         return result
     #Se retornar falha:
     else
         Value[V] =  None
 # Se chegou até aqui, é porque não retornou nenhum resultado.
 # Demonstrando que nao há valores possiveis a serem assinalados a V
 return failure
```

A imagem a seguir mostra parte da árvore gerada, utilizando o algoritmo de busca retroativa, no problema de coloração anteriormente exposto.



Parte da árvore percorrida no algoritmo de busca retroativa na coloração do mapa.

O algoritmo pode ser resumido nos seguintes passos:

1. Escolhemos uma variável ainda sem atribuição de valor. No caso do mapa, seria algum dos países A, B, C,...F;

2. Percorremos os valores que essa variável escolhida pode assumir (verde, vermelho ou azul), de acordo com o domínio;

3. Se esse valor for consistente com o conjunto de restrições, nós descemos um nível da árvore, e voltamos ao passo 1;

4. Caso já tivermos todas as variáveis com valores atribuídos e com atribuições consistentes com as restrições, o algoritmo termina;

5. Caso tivermos variáveis ainda sem atribuição, porém sem valores a serem atribuídos devido a restrições, nós fazemos o retrocesso, ou seja, subimos a árvore de forma a desfazer a última atribuição de valor feita, escolhendo outro valor a ser atribuído à variável escolhida no nível de cima.

Esse algoritmo resolve o problema e, no pior dos casos, ele testa todas as combinações de cores possíveis no caso do problema dos mapas. Precisamos assumir algumas hipóteses (heurísticas) que acelerem o processo de encontrar uma solução válida.

Heurísticas

Variável com mais restrições

A função PickUnassignedVariable() seleciona uma variável cujo valor ainda não tenha sido atribuído a ela. A escolha dessa variável pode alterar o desempenho da busca em relação ao tempo.

Na imagem anterior, quando escolhemos A=Vermelho e B=Azul, só há uma possibilidade na coloração da cidade C (verde), e duas possibilidades para cidade D (vermelho e verde). A variável C é a mais restrita, porém o ramo mais à esquerda da árvore preferiu escolher a cidade D como variável ainda não atribuída. Faz mais sentido ter preferido escolher primeiro a cidade C como variável ainda sem atribuição de valor.

Essa estratégia de escolher a variável com mais restrições de valores (ou seja, com menos possibilidade de escolha) é uma heurística aplicada a esse problema e, em muitos casos, melhora o desempenho para encontrar uma solução. Ela detecta mais rapidamente falhas, porque, se não houver valores a serem atribuídos a uma variável, podemos retroceder na busca imediatamente. Por isso, essa heurística é comumente chamada de valores mínimos restantes (minimum remaining values – MRV), ou variável mais restrita (most constrained variable), ou heurística “fail-first”.

Exemplo

A cidade C faz fronteira com cinco cidades, e diremos que seu grau é 5. É a cidade que possui maior grau. Escolhê-la como cidade para começar o algoritmo é uma estratégia interessante, pois, pelo fato de C ter mais restrições, a escolha das cores de seus vizinhos interfere na escolha de sua cor, podendo levar a casos de falha, gerando retrocessos na busca.Essa heurística é muitas vezes utilizada como desempate, quando utilizamos junto com a MRV, no caso em que variáveis possuem o mesmo número de restrições de valores.

Valor que restringe menos

Ambas as heurísticas apresentadas são utilizadas na escolha da cidade (variável) que ainda não possui atribuição de valor, sem considerar a cor (valor) que pode assumir. Na atribuição de valores da variável escolhida, podemos ordenar os valores seguindo uma determinada estratégia.

Suponha que no mapa anterior começamos a escolha com A=Azul e B=Vermelho, e queremos escolher a cor de D. Se escolhermos D=verde, C fica sem escolha de cor. Assim, iremos priorizar a escolha de D=Azul.

> A estratégia proposta é baseada no fato de que a melhor cor para uma cidade é a que restringe menos seus vizinhos. Ela é muito aplicada na escolha dos valores das variáveis ainda sem atribuição.

Dessa forma, no código da imagem anterior, o loop que percorre o domínio da variável a ser atribuída (for d in Domain(V)) pode ser modificado de modo que Domain(V) esteja ordenado pelos valores que menos restringem seus vizinhos.

Verificação prévia – Forward Checking

No algoritmo de busca retroativa genérico proposto, escolhemos uma variável ainda não atribuída, selecionamos valores possíveis de seu domínio e testamos o valor escolhido em relação às restrições. Se for possível escolher aquele valor, nós descemos um nível na árvore.

Podemos fazer um melhor uso das restrições e, por isso, introduziremos o conceito de Forward Checking.

Comentário

Quando um valor é atribuído a X, nós iremos verificar todas as variáveis relacionadas com X. No caso dos mapas, verificaremos todos os vizinhos da variável escolhida.

Para todas essas variáveis relacionadas a X, nós excluiremos dos domínios delas todos os valores que são inconsistentes com uma determinada atribuição.

No caso dos mapas, se escolhemos **Vermelho** para a cidade **X**, excluiremos do domínio dos vizinhos a possibilidade de escolher vermelho. Dessa forma, iremos antever se a escolha dessa atribuição impactará seus vizinhos. Assim, se eles não tiverem opção de valor a ser escolhido, nós não faremos essa escolha, evitando que desçamos na árvore e, consequentemente, prevendo o retrocesso na busca. Além disso, se, com essa escolha de vermelho para a variável X, houver um vizinho com somente uma possibilidade de escolha, já é possível selecioná-lo como próximo valor a ser atribuído e ver se essa escolha obrigatória também impactará seus vizinhos.

> Com a verificação prévia, eliminamos algumas escolhas errôneas que iremos introduzir com uma determinada escolha de valor.

Esse processo de antever se uma atribuição de valor vai gerar uma inconsistência é utilizado junto com a busca genérica proposta e com as heurísticas apresentadas. Essa combinação acelera o encontro de soluções para o problema em alguns casos.

Na tabela a seguir, comparamos o algoritmo proposto inicialmente, usando as heurísticas propostas e forward checking, no problema de colorir o mapa dos EUA usando somente quatro cores. Listado em cada célula está o número médio de verificações de consistência (em cinco execuções) necessárias para resolver o problema. Os valores que estão em parênteses são de execuções que não conseguiram achar solução na janela de tempo avaliada.

Estratégia de busca	Colorir mapa dos 50 estados dos EUA, usando quatro cores
Busca com Retrocesso (BackTracking) - BT	(>; 1,000K)
BT+MRV	(>; 1,000K)
BT+Forward Checking (FC)	2K
BT+MRV+FC	60

Tabela: Comparação dos algoritmos de Busca com Retrocesso com diversas estratégias de implementação. Extraída de: Russell, S. J., Norvig, P., & Davis, E., 2010, p.143.

Dessa forma, a busca retroativa com a heurística MRV e Forward Checking conseguiu solucionar o problema de coloração com tempo menor que outras implementações, sendo que a busca retroativa genérica simples não conseguiu achar solução em uma janela de tempo hábil.

Comentário

As aplicações de heurísticas dependem da natureza do problema. Empregar diferentes estratégias pode acelerar consideravelmente a busca de soluções.

Resolvendo problemas por meio de buscas

Neste vídeo, será explicado como buscar solução para problemas de satisfação com restrições, utilizando exemplos práticos para auxiliar o entendimento.

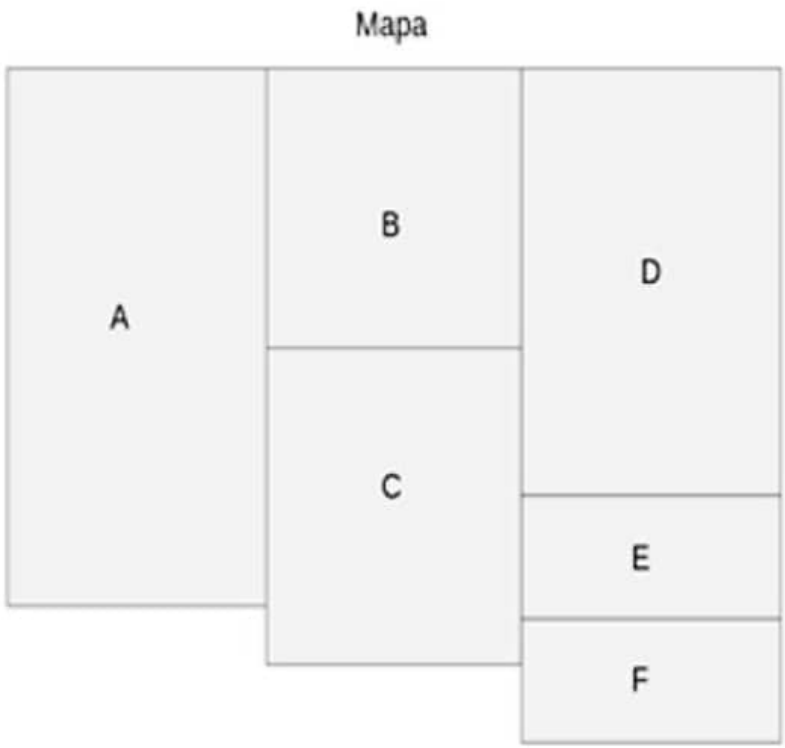
Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Verificando o aprendizado

Questão 1

Considere o mapa a seguir. Sobre o problema de colorir um mapa utilizando k-cores, com a restrição de que países vizinhos não podem possuir a mesma cor, assinale a alternativa verdadeira:



A

É um problema com restrições unárias.

B

A variável com maior número de restrições com seus vizinhos é a D.

C

É um problema cuja solução é de ordem polinomial.

D

Assumir heurísticas torna o problema de ordem polinomial.

E

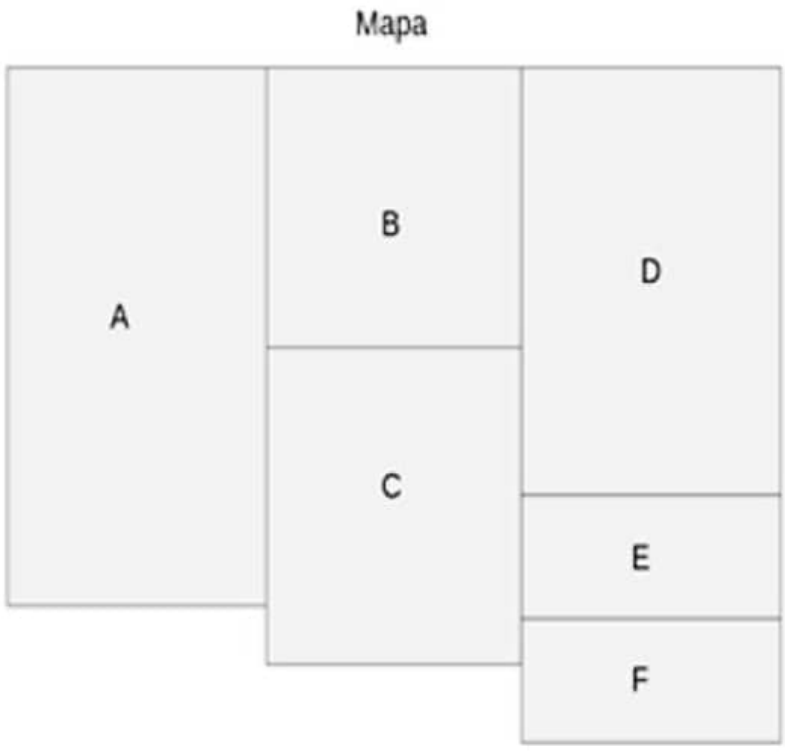
Esse problema pode ser redutível a um problema SAT.

A alternativa E está correta.

De fato, esse problema pode ser redutível ao problema SAT.

Questão 2

Considere o mapa a seguir. Utilizando a heurística MRV, ou seja, selecionaremos primeiro as variáveis que possuem a menor quantidade possível de valores para atribuir, e, em caso de empate, usaremos outra heurística de escolher primeiro a variável com o maior número de restrições com outras variáveis (a variável de maior grau de restrições), qual seria o primeiro país a começar a ser preenchido?



A

A

B

B

C

C

D

D

E

E

A alternativa C está correta.

Como todas os países possuem disponíveis três valores a serem preenchidos, usaremos como desempate a segunda heurística. Logo, o país com o maior número de restrições é o C.

O que é programação em lógica?

Linguagens de programação em lógica (**PL**) também chamadas de **linguagens declarativas** diferem substancialmente de outras linguagens de programação procedurais comumente usadas como Python, C, Java, Javascript etc. Programas lógicos não especificam explicitamente, na maioria dos casos, sua sequência de execução. Por causa disso, programação em lógica se torna, muitas vezes, bem difícil para iniciantes.

Neste vídeo, vamos explorar os fundamentos da Programação em Lógica, uma abordagem declarativa para resolver problemas computacionais. Vamos mergulhar na lógica de predicados, cláusulas de Horn e unificação, revelando como a Programação em Lógica difere da Programação Imperativa.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

> PL é baseada em um subconjunto de lógica de primeira ordem, materializando-a em um paradigma de programação declarativo. A dedução lógica dessas linguagens determina se uma cláusula lógica é uma consequência do programa.

Sendo assim, definiremos elementos básicos da lógica de predicados (ou lógica de primeira ordem), para entendermos como construir um programa lógico.

Revisão de lógica dos predicados

Na lógica proposicional, só conseguimos representar fatos que podem ser verdadeiros ou falsos, o que não é suficiente para representar sentenças e relações complexas, tendo um poder expressivo limitado.

As sentenças a seguir são exemplos de relações que não conseguimos representar pela lógica proposicional:

- Alguns humanos são inteligentes.
- Humanos são mortais.
- Sócrates é humano.
- Logo, Sócrates é mortal.

A quarta sentença é conclusão da segunda e terceira. Por meio da lógica proposicional não conseguimos deduzir que Sócrates é humano, por exemplo. Além disso, não conseguimos também representar eficientemente a primeira sentença. A lógica dos predicados é uma extensão da lógica proposicional! Além dos símbolos $f' v_y x^2, \frac{z}{z} \rightarrow \frac{z}{z}$, teremos também objetos, variáveis, predicados e quantificadores.

Objetos

Pessoas, números, cores etc. Convencionalmente são escritos com letras minúsculas.

Predicados

Denotam relações entre um ou mais objetivos. Essas relações podem ser verdadeiras ou falsas.

Exemplo 1: João é irmão de Maria, pode ser representado por irmao(joao,maria). O predicado irmão estabelece uma relação entre dois objetos (João e Maria). Os predicados convencionalmente são representados em letra minúscula. Se João realmente for irmão de Maria, ele assume o valor verdadeiro.

Exemplo 2: red(cadeira) pode estabelecer que o objeto cadeira é vermelho. Esse predicado é unário, só estabelece uma relação com um objetivo, ao contrário do exemplo anterior.

Variáveis

São usadas para não precisarmos nomear explicitamente objetivos. O irmao(A,B) representa uma relação entre A e B. Quando A é instanciado pelo objetivo a e B por b, a é irmão de b. Representamos variáveis com letra inicial maiúscula.

Quantificadores

São usados para especificar quantidades dos objetos em um determinado contexto.

- O quantificador universal $(\forall)$ estabelece fatos a respeito de todos os objetos de um contexto.
- O quantificador existencial $(\exists)$ estabelece a existência de um objeto.

As sentenças anteriormente exemplificadas podem ser representadas como:

Humanos são mortais

$\forall X[$ humano (X) $\rightarrow$ mortal (X)

Sócrates é humano

Humano(socrates).

Logo, Sócrates é mortal

$$\begin{aligned} &\forall X[\text{ humano (X) } \rightarrow \text{ mortal (X)}], \\ &\text{ humano(socrates)} \\ &\text{ mortal(socrates)} \end{aligned}$$

A conclusão de que Sócrates é mortal a princípio é bem simples, mas, dependendo da complexidade das sentenças e predicados, pode ser humanamente impossível tirar conclusões diretas ou, até mesmo, quais são todas as conclusões possíveis de um determinado problema, ou se uma determinada sentença pode ser concluída como verdadeira ou falsa.

Programas em lógica podem ser utilizados para implementar raciocínios dedutivos. Vamos mostrar como eles podem ser aplicados na resolução de problemas úteis do mundo real, e como podem se tornar ferramentas poderosas de Inteligência Artificial.

Prolog

As principais famílias de linguagens de programação lógica incluem Prolog, ASP e Datalog. Nessas linguagens, as regras são escritas na forma de cláusulas: $H :- B_1, \ldots, B_n$

Essa cláusula significa que H é verdade se B_1 é verdade se B_2 é verdade se B_3 é verdade, assim por diante.

Prolog (PROgramming LOGic) (1972) foi a primeira linguagem de lógica formal amplamente difundida. De longe, a linguagem de programação lógica dominante é o Prolog. Então, essa é a linguagem que introduziremos. A computação ocorre tentando inferir respostas às consultas do usuário por meio de fatos e regras fornecidos. O usuário fornece os fatos, regras e consultas (expressos no cálculo de predicados), enquanto o computador executa a resolução para tentar inferir as respostas.

Instalação

Sugerimos instalar SWI Prolog como ambiente de desenvolvimento da linguagem Prolog. O manual de instalação está disponível na página **Download e Instalação**, para diversos sistemas operacionais.

Introdução à linguagem Prolog

Prolog

Neste vídeo, veja a apresentação da linguagem Prolog.

> Conteúdo interativo
>
> Acesse a versão digital para assistir ao vídeo.

Os elementos básicos da linguagem Prolog são os fatos, regras e consultas. Fatos e regras são cláusulas da lógica de predicados. Em Prolog, uma regra é descrita como:

Head :- Body

Head é verdade se **Body** for verdade. Body é um conjunto de predicados. Avaliar Head é equivalente a avaliar Body. Fato é uma regra que é sempre verdade, ou seja, Body é sempre verdade. Por exemplo:

“Sócrates é mortal” na lógica de predicados é um fato, sendo representado em Prolog pelo predicado:

mortal(socrates).

Isso é equivalente a dizer em Prolog:

mortal(socrates) :- true.

As sentenças “Sócrates é humano” e “Todo humano é mortal” são codificadas como:

humano(socrates).

mortal(X) :- humano(X).

humano(socrates) é um fato, mortal(X):-humano(X) é uma regra. A regra pode ser lida: mortal(X) é verdade se humano(X) é verdade. As cláusulas sempre terminam com ponto final.

Saiba mais

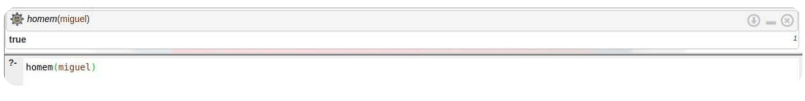
Perceba que socrates é um objeto, humano é um predicado e X é uma variável. Em Prolog, os objetos e os predicados são representados por letras minúsculas e as variáveis pela letra inicial maiúscula.

Uma consulta é basicamente uma pergunta que utilizamos por meio do prompt de comando para verificar se existe relacionamento entre objetos. A linguagem buscará quais variáveis tornam a consulta verdadeira, baseado nos fatos e nas regras. Se não tiver como inferir que a consulta é verdadeira, o retorno será **false**.

Considere o seguinte exemplo de árvore genealógica.

```
prolog


 /* Miguel é homem, etc*/
 homem(miguel).
 homem(arthur).
 homem(henrique).
 homem(pedro).
 homem(joao).
 homem(jose).
 /* Maria é mulher, etc*/
 mulher(maria).
 mulher(valentina).
 mulher(laura).
 mulher(larissa).

 /* Miguel é um dos pais de Laura*/
 pais_de(miguel,laura).
 pais_de(miguel,larissa).
 /* Maria é um dos pais de Laura*/
 pais_de(maria, laura).
 pais_de(maria, larissa).
 pais_de(arthur,pedro).
 pais_de(valentina, pedro).
 pais_de(laura, joao).
 pais_de(henrique, joao).
 pais_de(larissa, jose).
 pais_de(pedro, jose).
```

Realizando a consulta **homem(miguel)** retorna **true**, porque a linguagem irá verificar se existe algum fato ou regra que torne a consulta verdadeira. Poderá retornar falso caso não consiga provar logicamente que a consulta seja verdadeira.



Podemos ainda fazer consultas com variáveis, como **homem(X)**. Haverá a verificação dentre todas as possibilidades de regras e fatos de quais variáveis tornam a consulta verdadeira.

Podemos ainda definir diversos tipos de regras e relacionamentos. A vírgula faz a conjunção (operador Λ) dos predicados, o ponto e vírgula faz a disjunção (operador V).

Outro trecho do código representando essas regras está ilustrado a seguir:

```
prolog


                        /* Regras */
 /* X é pai de Y, se X é homem e X é um dos pais de Y.
  * Para isso, homem(X) deve ser verdade e pais_de(X,Y) deve ser verdade*/

 pai_de(X,Y):- homem(X),
     pais_de(X,Y).

 /* X é mãe de Y, se X é mulher e X é um dos pais de Y.
  * Para isso, mulher(X) deve ser verdade e pais_de(X,Y) deve ser verdade*/
 mae_de(X,Y):- mulher(X),
     pais_de(X,Y).

 /* X é avô de Y, se X é homem e X é um dos pais de Z e Z é um dos pais de Y!
  * Para isso, homem(X) deve ser verdade e pais_de(X,Z) deve ser verdade e pais_de(Z,Y)
deve ser verdade*/
 avô_de(X,Y):- homem(X),
     pais_de(X,Z),
     pais_de(Z,Y).

  /* X é avó de Y, se X é mulher e X é um dos pais de Z e Z é um dos pais de Y!
  * Para isso, mulher(X) deve ser verdade e pais_de(X,Z) deve ser verdade e pais_de(Z,Y)
deve ser verdade*/
 avó_de(X,Y):- mulher(X),
     pais_de(X,Z),
     pais_de(Z,Y).

  /* X é irmã de Y, se X é mulher e P é pai de Y e P é pai de X e X é diferente de Y!
  * Para isso, mulher(X) deve ser verdade e pai_de(P,Y) deve ser verdade e pai_de(P,Y)
deve ser verdade*/
 irma_de(X,Y):-
     mulher(X),
     pai_de(P, Y), pai_de(P,X),X \= Y.
```

As seguintes consultas retornam true ou false?

```
prolog

mae_de(laura,maria)
```

Você obterá como resultado falso para os dois, porque não há uma regra ou fato que torne verdadeira a consulta.

A primeira consulta é aplicação da regra mae_de(X,Y) quando os objetos são laura e maria. A linguagem irá avaliar mulher(laura) e pais_de(laura,maria), porque mae_de é uma regra que depende de mulher(X) e pais_de(X,Y). Para avaliar mulher(laura), haverá a verificação dos fatos e regras. Nesse caso, há um fato que

torna verdade mulher(laura). Haverá a avaliação de pais_de(laura,maria). Não há um fato que torne esse predicado verdadeiro, e, por isso, é retornado false.

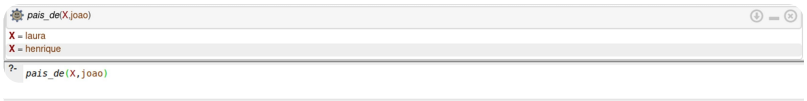
Que tal agora você executar as seguintes consultas e avaliar a resposta?

```
prolog

irmao_de(miguel,joao)
 ancestral_de(pedro,joao)
```

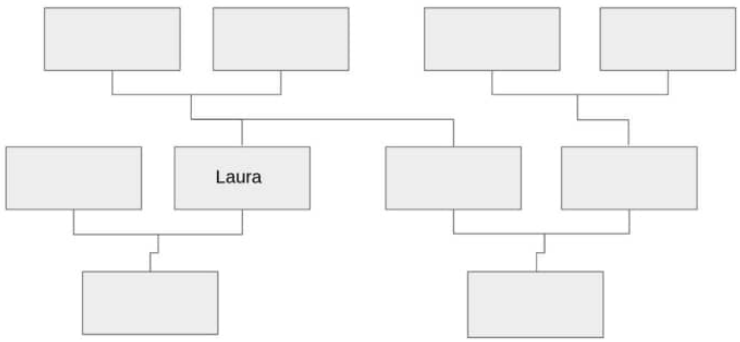
Como dito, Prolog busca nos fatos e regras valores que tornam a consulta verdadeira. Por isso, podemos fazer consultas utilizando variáveis, e a linguagem retornará todos os valores possíveis da variável consultada. As consultas a seguir retornam lista de valores para as variáveis inseridas:

```
prolog

mae_de(X, laura)
 pais_de(X,joao)
 irma_de(X,larissa)
 ancestral_de(X,larissa)
```

A consulta **pais_de(X,joao)**, por exemplo, retorna **X = laura** ou **X=joao**, pois os únicos valores possíveis que tornam pais_de(X,joao) verdadeiros são esses, como mostrado no trecho de código:



Agora você pode utilizar o código da árvore genealógica, realizar consultas e preencher a imagem da árvore genealógica a seguir:



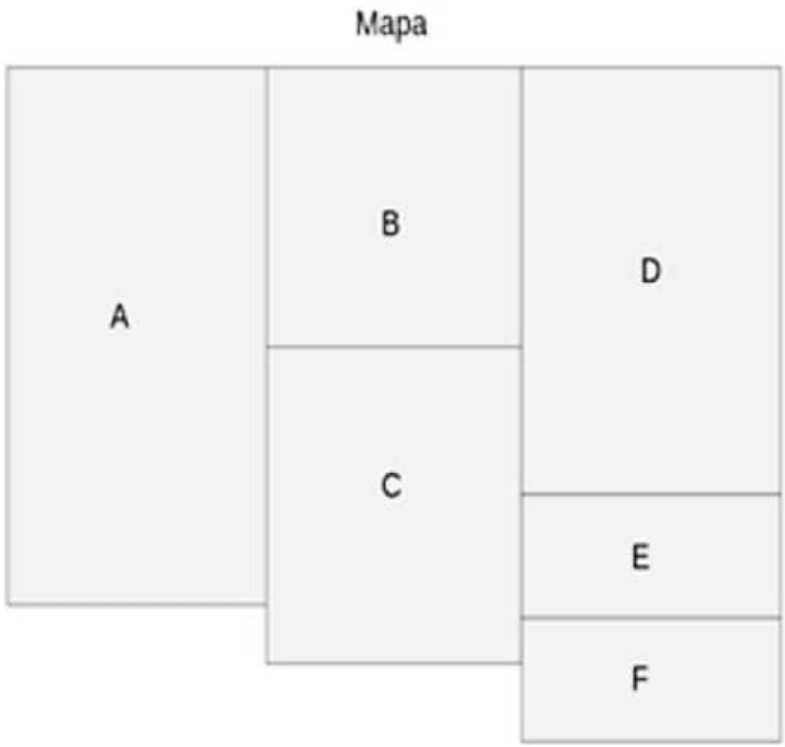
Programação em lógica com restrições

A habilidade de uma linguagem lógica como Prolog de deduzir relacionamentos por meio dos fatos e regras faz com que ela seja extremamente útil na inteligência artificial. Os problemas de satisfação com restrições são exemplos de aplicabilidade de Prolog. Iremos, portanto, elucidar como essa linguagem pode ser utilizada para resolver problemas de satisfação com restrições.

Os seguintes problemas serão abordados: **problema de coloração de mapas, problema de satisfação com restrições com domínios finitos e com domínios reais**.

Problema de coloração de mapas

Como já introduzido no módulo anterior, considere o seguinte mapa, em que poderemos utilizar **três cores** (vermelho, azul e verde), codificadas como (**R,G,B**) para colorir cada país, de modo que países adjacentes tenham cores diferentes.



Variáveis: A,B,C,D,E,F

Domínio: $D_A = D_B = D_C = D_D = D_E = D_F = \{R, G, B\}$

Restrições: $A \neq B, A \neq C, B \neq C, B \neq D, C \neq D, C \neq E, C \neq F, D \neq E, E \neq F$

O problema pode ser codificado facilmente em Prolog. Para colorir o mapa, utilizamos o predicado coloring (A,B,C,D,E,F), impondo as condições de serem diferentes por meio do predicado different, e listando todas as restrições no corpo da regra. Os fatos são as cores que devem ser diferentes entre si.

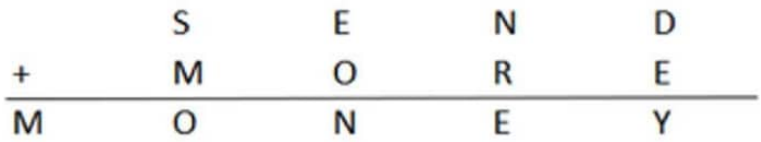
```
prolog

/*
  * Variáveis: A,B,C,D,E,F
     Domínio: {vermelho,verde,azul}
 Restrições: A!=B, A!=C, B!=C, B!=D, C!=D, C!=E, C!=F, D!=E, E!=F
 */

 /*
  * Impomos as restrições: different(A,B) siginifica que a cor de A deve ser diferente da
de B.
  * */

 coloring(A,B,C,D,E,F) :-
 different(A,B),
 different(A,C),
 different(B,C),
 different(B,D),
 different(C,D),
 different(C,E),
 different(C,F),
 different(D,E),
 different(E,F).

 /*
  * Os fatos: vermelho é diferente de azul, que é diferente de verde etc
  * */
 different(vermelho,azul).
 different(azul,vermelho).
 different(vermelho,verde).
 different(verde,vermelho).
 different(verde,azul).
 different(azul,verde).
```

A consulta **coloring** (A,B,C,D,E,F) irá buscar todas as variáveis que tornam **coloring** (A,B,C,D,E,F) verdadeiro. E pela regra do predicado coloring, a linguagem verificará quando cada um dos predicados **different** (X,Y) são verdadeiros.

Os fatos são verificados e conseguimos listar todas as possibilidades que tornam **coloring** (A,B,C,D,E,F) verdade.

Problema de restrição de domínio finito

O primeiro problema com restrições de domínio finito que será abordado é o seguinte:



As letras S, E, N, D, M, O, R, Y representam **algarismos de 0 a 9**. Para solucionar esse problema, iremos utilizar a biblioteca **clpfd** (constraint logic programming for finite domains).

Saiba mais

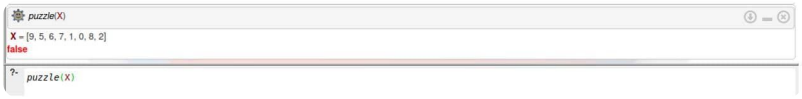
No espaço de busca para avaliar um predicado, Prolog só consegue deduzir o que vem de fatos e regras. Por isso, se usarmos o operador “is” e fizermos a consulta “3 is Y+2”, a linguagem não consegue deduzir o valor de Y.

A biblioteca clpfd consegue resolver esse tipo de problema. Usando operadores e predicados de restrições definidos em clpfd, conseguimos delimitar quais valores cada variável pode assumir. Assim, ao fazermos a consulta “3 #= Y+2”, usando operador de restrição “#=” de clpfd, a linguagem consegue inferir que Y=2. Mais informações sobre a biblioteca podem ser encontradas em **Biblioteca Clpfd**.

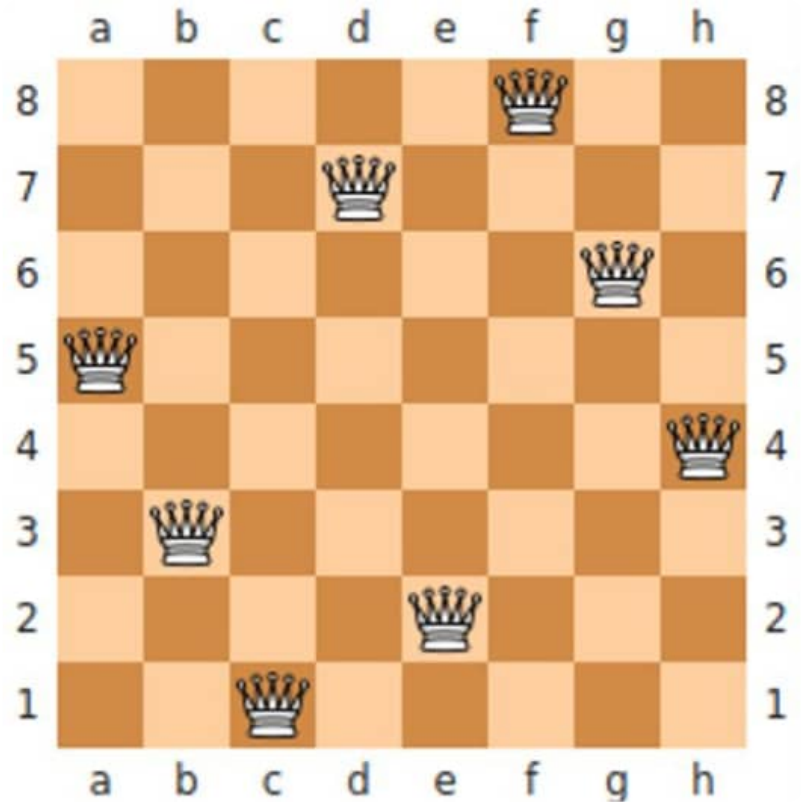
No próximo trecho de código conseguimos resolver facilmente o problema de restrição de domínios finitos com poucas linhas de código.

```
prolog

:- use_module(library(clpfd)).
 puzzle([S,E,N,D,M,O,R,Y]) :-
      Variables = [S,E,N,D,M,O,R,Y],
      Variables ins 0..9, %Todos os dígitos de 0 a 9. Usamos a restrição ins de clpfd
      all_different(Variables), %Todos os dígitos diferentes. Usamos a restrição
all_different de clpfd
      (1000*S + 100*E + 10*N + D) +  %S E N D
      (1000*M + 100*O + 10*R + E) #= %M O R E . Usamos a restrição #= de clpfd
      (10000*M  + 1000*O + 100*N + 10*E + Y), %M O N E Y
      S #\= 0, M #\=0, %Dìgitos iniciais diferentes de 0. Usamos a restrição #\= de clpfd
      label(Variables). %Atribuição de Valores. Usamos label de clpfd que faz atribuição
de valores
```

Ao realizar a consulta puzzle(X), todas as soluções são exibidas:



O segundo problema de restrição que apresentaremos é o **problema das rainhas**. Queremos dispor **n rainhas** em um tabuleiro de xadrez **n por n**, de modo que as rainhas não se ataquem entre si, sabendo que elas podem andar em toda diagonal, horizontal e vertical. A imagem a seguir mostra uma solução possível, no caso em que **n=8**.



Solução para o caso de um tabuleiro 8×8.

A solução foi implementada também utilizando a biblioteca clpfd, porém, antes de mostrá-la, é necessário entender como manipular listas em Prolog.

> Uma lista é uma coleção de itens.

Em Prolog, a lista vazia é representada por []. Uma lista não vazia pode ser representada por [X|Y] . X (cabeça da lista) representa o primeiro elemento e Y (cauda) representa os demais elementos.

Dessa forma, as seguintes consultas retornam os seguintes resultados:

```
prolog

[X|Y] = [a,b,c].
 %retorna X = a e Y = [b,c]
 [X|Y] = [a].
 %retorna X = a e Y = []
 [X|Y] = [].
 %retorna falso, pois X é não vazio
```

No próximo trecho de código, implementamos três predicados. O primeiro exibe todos os elementos da lista. O segundo verifica se um elemento X é membro da lista. O terceiro verifica se uma lista é o anexo de duas outras listas.

```prolog
% mostrar(L) : printa os elementos da lista L.
 mostrar([]).
 mostrar([X|Y]) :- write(X), write('',''), mostrar(Y).


 %membro(X,L) : o item X é membro da lista L. Vamos verificar como deduzir se X é membro da lista L

 membro(X,[X|_]). %se o item X é membro da cabeça, retornamos verdade! (Fato)
 membro(X,[_|Y]) :- membro(X,Y). %vou verificar se o item X é membro da cauda e não da cabeça.

 % attach(A,B,C): A anexado com B é a lista C. Vamos verificar como deduzir se C é anexo de A e B!

 attach([], B, B). %Anexar uma lista vazia a outra lista B dá B. Então é verdade! (Fato)
 attach([X|A], B, [X|C]) :- attach(A, B, C). %Anexar uma lista que tem cabeca X, vai resultar uma lista com cabeca X. Dessa forma, precisamos,

%verificar o anexo de A com B dando C.
```

Sugiro realizar as seguintes consultas para entender um pouco mais sobre o código:

- **mostrar([])**
- **mostrar([1,2])**
- **membro(1,[1,2])**
- **membro(3,[1,2])**
- **membro(X,[1,2])**

Deverá retornar todos os membros de [1,2], pois buscará dentre os fatos quais tornam membro(X,[1,2]) verdade.

- **attach([1,2],[3,4],[1,2,3,4])**
- **attach([1,2],[3,4],[1,2,3])**
- **attach([1,2],[3,4,5],C)**

Deverá retornar C = [1,2,3,4,5]

- **attach([1,2],B,[1,2,3])**

Deverá retornar B = [3]

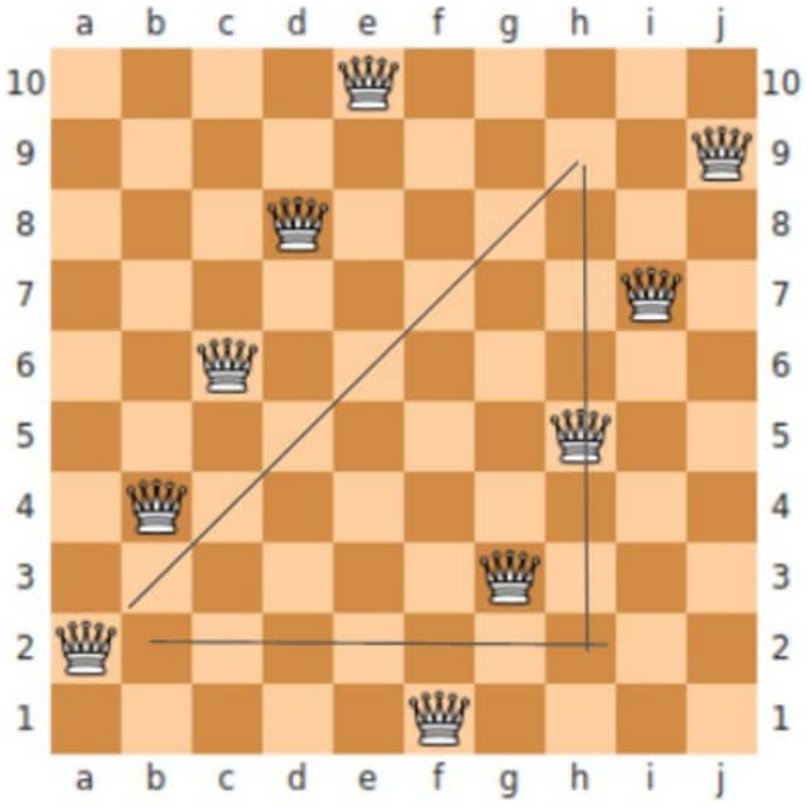
Já que aprendemos um pouco sobre manipulação com listas, podemos voltar ao problema das n rainhas.

- Cada rainha ocupará uma coluna porque somente uma rainha ocupa uma horizontal e uma vertical.
- Então, assumiremos que a rainha 1 ocupa a coluna 1, a rainha 2, a coluna 2; e assim por diante.
- Pelo fato de já sabermos a coluna da rainha 1, por exemplo, basta representar somente a posição da linha que ela ocupa. Essa representação impõe que as rainhas ocupem colunas diferentes.

- Por isso, nos preocuparemos com somente as restrições delas ocuparem linhas e diagonais diferentes.

Sejam **a** e **b** as colunas ocupadas por duas rainhas. Sejam **c** e **d** as linhas ocupadas por elas. A segunda rainha está **b-a** colunas após a rainha **a**. Para que elas não ocupem a mesma diagonal, |**b-a**| deve ser diferente de |**c-d**|.

É fácil perceber olhando na imagem a seguir, que se as rainhas estão na mesma diagonal, então um triângulo retângulo e isósceles é formado, com lados |**b-a**| e |**c-d**|. Por ser isósceles, |**b-a**| = |**c-d**|. Portanto, para não estarem na mesma diagonal, vamos impor a restrição que |**b-a**| seja diferente de |**c-d**|.



A diagonal da rainha da coluna 2. Se uma rainha estiver nessa diagonal, |b-a| vai ser igual a |c-d|.

O código é estruturado seguindo esse raciocínio. O código é pequeno, mas está bastante comentado. Não é fácil entender de primeira, e nem é um problema para iniciantes, mas, após entendido, conseguimos perceber como Prolog é uma ferramenta versátil e poderosa de Inteligência Artificial.

Problema de restrição de domínio real

O próximo problema é bem mais simples de ser entendido. Considere o seguinte problema:

$$\begin{aligned} 2X + Y &\leq 16 \\ X + 2Y &\leq 11 \\ X + Y &\leq 15 \quad Maximizar 30X + 50Y \end{aligned}$$

Para domínios reais, utilizamos a biblioteca clpr. As restrições aos problemas são inseridas entre chaves, conforme o trecho de código a seguir.

```
prolog

:- use_module(library(clpr)).
 sistema(X,Y,Z) :- {2*X+Y =< 16, X+2*Y =< 11,X+3*Y =< 15, Z = 30*X+50*Y}, maximize(Z).
```

A consulta do predicado sistema(X,Y,Z) retorna as variáveis que satisfazem a regra codificada.

Verificando o aprendizado

Questão 1

Modifique o programa **Restrições reais** para resolver o seguinte problema de restrição de domínio real:

$$Ygt;=7$$

$$X+Y+Zgt;=16$$

$$X=2Y$$

$$MinimizarX+Y$$

O valor mínimo é:

A

18

B

14

C

7

D

15

E

21

A alternativa E está correta.

Vejamos a resolução:

```
:- use_module(library(clpr)).


sistema(X,Y,Z) :- {Y >= 7, X+Y+Z >= 16,X = 2*Y, Z = X+Y}, minimize(Z).

```

Questão 2

Modifique o programa Restrições Domínios Finitos para resolver o seguinte problema de restrição de domínios finitos, em que cada letra representa um algarismo de 0 a 9.

A solução tem valores tais que o valor de T é:

A

4

B

5

C

7

D

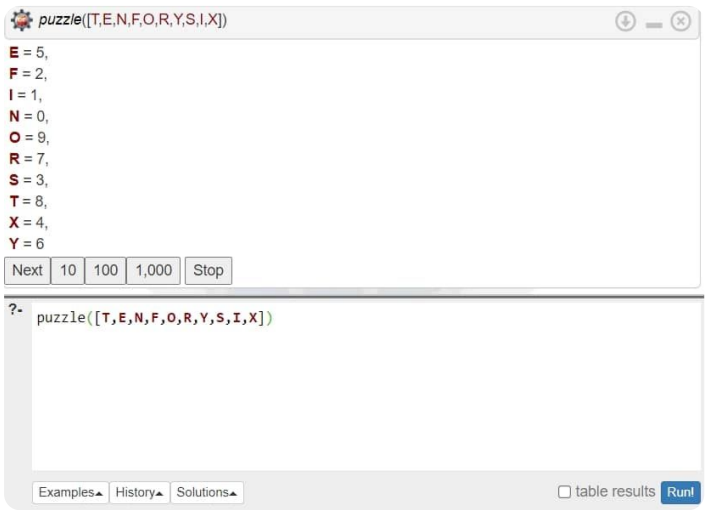
8

E

2

A alternativa D está correta.

Vejamos a resolução:



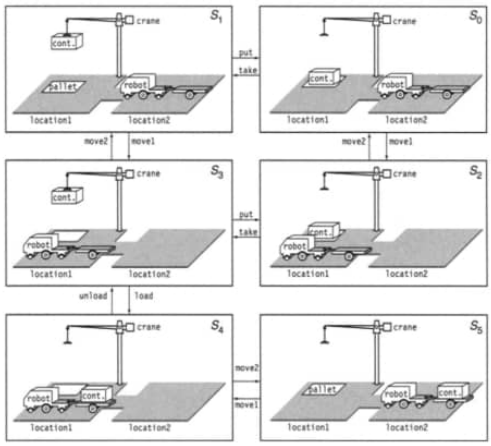
Métodos de planejamento

Neste vídeo, vamos explorar os desafios enfrentados pelos Dock Worker Robots (DWR), robôs que atuam em operações de carga e descarga em portos e armazéns. Vamos discutir como a Inteligência Artificial pode ser aplicada para melhorar a eficiência e a segurança dessas operações. Abordaremos técnicas como planejamento de trajetórias, alocação de recursos e colaboração entre robôs.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

O caso base de estudo neste módulo é o problema dos robôs autônomos do porto, doravante **DWR** (Dock Worker Robots). Robôs conseguem carregar e descarregar contêineres, que estão em cima de pallets e se movem por meio de guindastes. A imagem a seguir ilustra o problema DWR.



Representação dos estados do problema dos robôs do porto (DWR) no caso de 2 localizações, 1 robô e uma pilha.

Exemplo

Nesse problema, para 1 contêiner, 1 robô, uma pilha (pallet) e 2 localizações possíveis (location1 e location2) há oito estados possíveis. Para cinco localizações, três pilhas, três robôs e 100 contêineres, há 10277 estados possíveis! Esse número é muito maior que o número de átomos do universo observável!

Para o problema DWR, urge a necessidade de ter uma representação para cada estado, sem precisarmos enumerar todos os estados possíveis, pois, muitas vezes, a enumeração é não factível.

Uma representação de cada estado é necessária de modo que possamos entender o ambiente em que o sistema se encontra. A representação dos estados permitirá que, ao aplicarmos ações, o que do estado muda. Para isso, apresentaremos a representação clássica.

Representação clássica

A ideia é representar cada estado por um conjunto de características e definir um conjunto de operadores que serão usados para computar a transição entre outros estados. Veja as etapas que compõem essa representação.

1

Os estados serão representados como um conjunto de átomos lógicos (que poderão ser verdadeiros ou falsos).

2

As ações serão representadas por operadores que mudam os valores desses átomos.

3

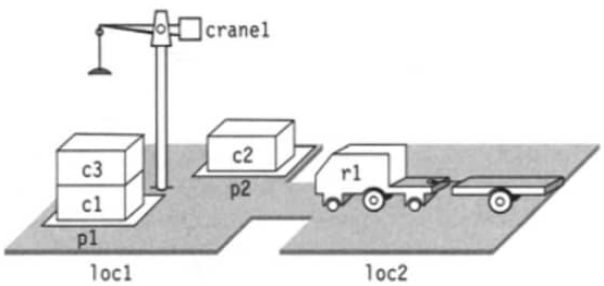
Usaremos uma linguagem de primeira ordem Q, na qual haverá símbolos de predicados e constantes, mas não símbolos de funções.

4

Forneceremos somente o estado inicial s_0 e utilizaremos operadores para gerar outros estados.

Para o problema DWR, no caso da imagem a seguir, no domínio do planejamento, podemos definir as seguintes constantes:

```
plain-text

Localizações: {loc_1,loc_2\}
 Pallets: {p1,p2}
 Robôs: {r1}
 Guindastes: {crane1}
 Containers: {c1,c2,c3}
```



Exemplo de um estado do problema DWR.

Utilizaremos diversos predicados para definir esse estado específico do sistema:

```
plain-text

state = {attached(p1,loc1), attached(p2,loc1), in(c1,p1), in(c3,p1), top(c3,p1),
on(c3,c1), on(c1,pallet),  in(c2,p2),  top(c2,p2), on(c2,pallet),  belong(crane1,loc1),
 empty(crane1),adjacent(loc1,loc2), adjacent(loc2, loc1), at(r1,loc2), occupied(loc2),
 unloaded(r1)}
```

Os predicados representam relações, como:

1. attached(p1,loc1)
   Significa que o pallet **p1** está anexado à localização **loc1**.

2. in(c1,p1)
   Significa que o container **c1** está no pallet **p1**.

3. top(c3,p1)
   Significa que o container **c3** está no topo do pallet **p1**.

4. on(c3,c1)
   Significa que o container **c3** está em cima do container **c1**.

5. belong(crane1,loc1)
   Significa que o guindaste **crane1** está na localização **loc1**.

6. empty(crane1)
   Significa que o guindaste **crane1** está vazio.

7. adjacent(loc1,loc2)
   Significa que as localizações **loc1** e **loc2** são adjacentes.

8. at(r1,loc2)
   Significa que o robô **r1** está na localização **loc2**.

9. occupied(loc2)
   Significa que a localização **loc2** está totalmente ocupada.

10. unloaded(r1)
    Significa que o robô não tem cargas.

Perceba que, para esse sistema, **o predicado adjacent é invariante**. Porque as localizações de loc1 e loc2 não mudam. Portanto, dizemos que o predicado adjacent é uma relação rígida.

Por outro lado, os predicados que representam relações que podem mudar dependendo do estado são chamados de predicados fluentes ou relações flexíveis, como **at**, que é um predicado fluente, porque o robô muda de localização entre os estados.

> A função de transição é especificada genericamente por um conjunto de operadores de planejamento, instanciados nas ações.

No planejamento clássico, o operador de planejamento é uma tripla ***o** = (name(o), precond(o), effects(o))*, onde:

Name(o)

É uma expressão sintática da forma n(x_1,...,x_k) , onde o n é um símbolo único do operador e x_1, ...,x_k são todas as variáveis que aparecem em **o**.

Precond(o)

São as pré-condições (literais) para que o operador seja executado.

Effects(o)

São os efeitos do operador. Literais que sofrerão mudanças (se tornarão verdadeiro ou falso) após o operador ser executado.

As ações serão representadas por operadores. A imagem a seguir apresenta o pseudocódigo da função load, que implementa a ação de carregar um container em cima do robô.

```
plain-text

Load(k,l,c,r)
      ;;guindaste k, na localização l, carrega o conteiner c em cima do robô r
 precond: belong(k,l), holding(k,c), at(r,l), unloaded(r)
 effects: empty(k),¬holding(k,c), loaded(r,c),¬unloaded(r)
```

Para o **guindaste k**, na **localização l** carregar o **container c** em cima do **robô r**, algumas precondições devem ser estabelecidas:

- O guindaste k deve estar na localização l: **belong(k,l)**.
- O guindaste k deve estar segurando o container c: **holding(k,c)**.
- O robô r deve estar na localização l : **at(r,l)**.
  O robô r deve estar vazio: **unloaded(r)**.

Quais serão os efeitos?

- O guindaste k será esvaziado: **empty(k)**.
- O guindaste k não estará segurando o container c: **¬holding(k, c)**.
- O robô r estará com o container c carregado: **loaded(r,c)**.
  O robô r não estará vazio: **¬unloaded(r)**.

Seja **a** um operador ou ação. Nos efeitos e precondições, selecionamos os átomos que são positivos e negativos em dois conjuntos:

Precond

Precond $^{+}(a) =$
{ átomos positivos de precond(a) }

Precond $^{-}(a) =$
{ átomos negativos de precond(a)}

Efeitos

Efeitos $^{+}(a) =$
{ efeitos positivos de effects(a) }

Efeitos $^{-}(a) =$
$\{$ efeitos negativos de effects (a)}

Se **a** é aplicável a s, o resultado da execução de a será:

$$\gamma(s, a) = \left(s - \text{ efeitos }^{-}(a)\right) \cup \text{ efeitos }^{+}(a) \$$$

Comentário

Removemos os efeitos negativos e adicionamos os positivos. Portanto, conseguimos fazer transição entre estados conhecendo os efeitos e precondições de cada ação.

Uma ação a é relevante para uma meta g, se ela for aplicável no estado corrente s e:

$$\text{g} \cap \text{efeitos }^{+} = \emptyset \text{ e } g \cap \text{ efeitos }^{-}(a) = \emptyset \$$$

Resolvendo problemas de planejamento

Neste vídeo, veja quais são os algoritmos de busca que podem ser utilizados para resolver os problemas de planejamento.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

As técnicas e ferramentas aprendidas no módulo de problemas de satisfação com restrições (PSR) e de programação em lógica com restrições podem ser aplicadas no problema de planejamento ao estruturar o problema de planejamento como um PSR. A abordagem mais utilizada e frequente é usar técnicas de resolução de PSR com uma abordagem específica para o planejamento.

> Entretanto, a maneira mais simples de resolver um problema de planejamento clássico (com as oito restrições do modelo restritivo impostas) é utilizar os algoritmos de buscas no espaço dos estados.

Abordaremos o problema de planejamento como um problema de busca, e não como PSR, portanto. Cada nó corresponde a um estado possível e cada arco corresponde a uma transição de estado, e o plano corrente corresponde ao caminho percorrido pelo espaço de busca. Essa abordagem é chamada de planejamento no espaço de estados.

Há diversos algoritmos de buscas para grafos, como:

- A busca em **profundidade**, conhecida como **DFS** (Depth First Search).
- Busca em **largura**, conhecida como **BFS** (Breadth-first search).

Esse tipo de busca é conhecido como **busca não informada**, porque não há nenhuma informação adicional sobre os estados (nós do grafo), se o determinado estado é melhor que outro por exemplo, seguindo algum tipo de critério.

A **busca informada**, também conhecida como **busca heurística**, apresenta informação adicional sobre o estado, de modo que conseguimos saber o quão promissor aquele estado é.

Exemplo

A* e Best-First Search (busca gulosa) são exemplos de busca informada.

Introduziremos alguns algoritmos de busca aplicados diretamente ao problema de planejamento.

Planejamento progressivo

0 algoritmo de busca progressiva busca no espaço de estados, partindo de um estado inicial S_0 , com objetivo de chegar a um estado final g , ou a um conjunto de estados finais g. Como input, ele recebe a declaração de um problema de planejamento $P = (O, s_0, g)$, ou seja, a coleção de operadores, o estado inicial e a meta. A imagem a seguir exibe o pseudocódigo de como a busca progressiva pode ser implementada.

```
plain-text


                        BuscaProgressiva(0,s0,g)
    S <- s0
    #π será o conjunto de ações – nosso plano
    π <- plano vazio
    while
       if s satisfizer meta g
          return π
          #Todas as ações que podem ser aplicadas em
          #s tem pré condições verdadeiras.
          Acoes_aplicaveis <- {selecionar ações
                                      aplicáveis}
               #se não tiver ação aplicável
            if acoes_aplicaveis == ∅
               Return failure
            else
               #Devemos escolher uma ação aplicável
               #de maneira não determinística para
               #prosseguir a busca
                   a <- escolher uma ação aplicável do
                         conjunto de ações_aplicaveis           #Fazendo o mapeamento
para o novo estado
               s <- γ(s,a)
                # Concatenamos ao plano π a ação a
               Π <- π.a
```

Há um loop cujo critério de parada é se o estado corrente s pertencer ao conjunto de metas g, nós retornamos o plano π . Caso s não satisfaça essa condição, nós selecionamos todas as ações possíveis aplicadas em s.

> Basicamente, as ações são os operadores, e as aplicáveis são aquelas cujas precondições são satisfeitas para o estado s.

Para selecionar essas ações aplicadas, devemos testar as precondições de todos os operadores e selecionar aquelas que satisfazem essas condições.

Se não houver ações aplicáveis (um conjunto vazio), há o retorno de falha. Caso contrário, nós selecionamos uma ação do conjunto e a aplicamos ao estado **s**.

Ao aplicar a função de transição gama, o sistema é levado a um novo estado corrente. Nós salvamos essa ação ao plano e continuamos no loop até s pertencer ao conjunto de metas **g**. No fim, o plano terá o conjunto de ações a serem tomadas, cujo estado final será a meta **g**.

Planejamento regressivo

No planejamento regressivo, nós começamos a busca pela meta g, e o objetivo é chegar no estado inicial. Para isso, precisamos saber como evoluir de um estado posterior a um estado anterior. Perceba que a função de transição $\gamma(s, a)$ leva progressivamente de um estado anterior s a um posterior s'. Assim, precisamos utilizar a inversa da função de transição $\gamma(s, a)$. Essa função faz a transição da meta g a um estado anterior s, conforme explicado.

Portanto, a busca evoluirá de modo que faremos a transição para um estado anterior **s**.

A imagem a seguir exibe o pseudocódigo de como a busca regressiva pode ser implementada.

```
plain-text

BuscaRegressiva(O,s0,g)
    S<- s0
    #π será o conjunto de ações – nosso plano
    Π <- plano vazio
    While
        if s satisfizer meta g
          return  π
       #selecionamos todas as ações relevantes a s
       #naquele estado
       acoes_relevantes <-{selecionar ações relevantes
                                                em s}
       #se não tiver ações relevante
 s
       if acoes_relevante == ∅
           return failure
       else
             #devemos escolher uma ação relevante de
            #maneira não determinística para prosseguir
            #a busca
            a <- escolher uma ação relevante do
                        conjunto de acoes_relevante
            #Concatenamos ao plano π a ação a
            #fazemos o mapeamento inverso para o novo
            #estado g
            g <- σ(g,a)
```

Também há um loop cujo critério de parada é se o estado inicial S_0 satisfizer a meta g, nós retornamos o plano π. Ao contrário da busca progressiva, a meta g é móvel, como se a meta estivesse cada vez mais perto de S_0. A meta g começa com o valor do objetivo final.

Caso s não satisfaça a meta **g**, nós selecionamos todas as ações relevantes em **g**. Basicamente, as ações são os operadores, e as ações relevantes são aquelas definidas anteriormente.

Atenção

Se não houver ações relevantes, há o retorno de falha. Caso contrário, nós selecionamos uma ação do conjunto e a aplicamos ao estado g. Ao aplicar a função de transição inversa, o sistema é levado a uma nova meta g corrente. Nós salvamos essa ação ao plano e continuamos no loop.

O plano terá o conjunto de ações a serem tomadas, cujo estado final será a meta g. Observe que o plano final está na ordem inversa, pois começamos o algoritmo da meta g até o estado inicial S_0. No fim, teremos que aplicar esse plano ao contrário para atingir a meta desejada.

O algoritmo Strips

O planejamento progressivo lida com o fato de que o número de ações a serem aplicadas em um determinado estado tende a ser **alto**. Nesse algoritmo, nós selecionamos ações aplicáveis ao estado corrente sem nenhum tipo de heurística. Por isso, a convergência do algoritmo demora para atingir o estado final. De maneira semelhante se comporta o algoritmo da busca regressiva. Em ambos os algoritmos, a solução sempre é encontrada, se houver alguma.

> Strips é similar à busca regressiva, ou seja, partimos da meta para chegar ao estado inicial, porém utiliza uma estrutura recursiva.

A imagem a seguir ilustra como o algoritmo de Strips pode ser implementado.

```
plain-text

GroundStrips(0,s0,g)
    S<- s0
    #π será o conjunto de ações – nosso plano
    π <- plano vazio
    While
        if s satisfizer meta g
          return  π
       #selecionamos todas as ações relevantes a s
       #naquele estado
       acoes_relevantes <-{selecionar ações relevantes
                                              em s}
       #se não tiver ações relevante
 s
       if acoes_relevante == ∅
           return failure
       else
           #devemos escolher uma ação relevante de
           #maneira não determinística para prosseguir
           #a busca
           a <- escolher uma ação relevante do
                        conjunto de acoes_relevante
           # A ação relevante tem um conjunto de pré
           #condiçoes
           #Usaremos recursão para acharmos um conjunto
           #de ações a serem tomadas a partir do
           #estado s que nos leva a um estado cujas as
           #pre condições de a sejam satisfeitas

           #o retorno será o conjunto π_2 - o plano
           π_2 = GroundStrips(0,s,precond(a))
           #se o plano for vazio
           if π_2 == ∅
               return failure
           #Ao aplicar esse plano, as pré condições de
           #a serão satisfeitas. Qual será o estado ao
           #aplicar este plano
           # basta aplicar a função de mapeamento
           s <- γ(s,π_2)
           #como as pre condições de a foram satisfeitas,
           #basta aplicar a nos levando a outro estado
           s <- γ(s,a)
           #concatenamos ao plano π: π_2 e a ação a
           π <- π.π_2.a
```

Há um loop cujo critério de parada é, se o estado inicial S_0 satisfizer a meta g, nós retornamos o plano π caso sim. De todas as ações relevantes que podem ser aplicadas à meta móvel e corrente g, escolhemos não deterministicamente uma ação.

Os estados que satisfizerem as precondições dessa ação são as próximas metas correntes, e utilizamos recursão para repetir o algoritmo novamente, com as mesmas especificações, porém com as metas atualizadas. Não há retrocesso (backtracking) caso haja uma falha. O algoritmo retorna o plano se o estado inicial pertence à meta corrente.

Comentário

Os três algoritmos apresentados são genéricos. Com base neles, podemos desenvolver algoritmos para domínios específicos. Heurísticas podem ser elaboradas de modo a melhorar o desempenho em relação ao tempo. O algoritmo Strips nem sempre acha solução, mesmo se tiver, e pode não achar a solução ótima para alguns casos, o que chamamos de anomalia de Sussman.

Verificando o aprendizado

Questão 1

Assinale a alternativa verdadeira a respeito da resolução do problema de planejamento por meio de buscas.

A

Strips sempre acha a melhor solução.

B

Strips implementa um tipo de busca progressiva.

C

Planejamento progressivo seleciona todas as ações relevantes do estado e seleciona uma para aplicar ao estado corrente.

D

Planejamento regressivo seleciona todas as ações aplicáveis do estado e seleciona uma para aplicar ao estado corrente.

E

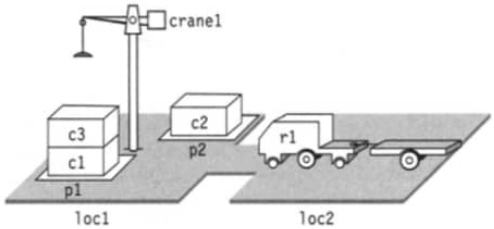
Planejamento progressivo retorna falha se não tiver ações aplicáveis a um determinado estado.

A alternativa E está correta.

De fato, o planejamento progressivo seleciona um conjunto de ações aplicáveis, e retorna falha caso não consiga encontrar! Atentar que para Strips e planejamento regressivo, as ações selecionadas são as relevantes, e não aplicáveis.

Questão 2

Considere o problema dos robôs autônomos do porto. Considere um dos estados do problema definido na imagem a seguir:



state = {attached(p1,loc1), attached(p2,loc1), in(c1,p1), in(c3,p1), top(c3,p1), on(c3,c1), on(c1,pallet), in(c2,p2), top(c2,p2), on(c2,pallet), belong(crane1,loc1),empty(crane1),adjacent(loc1,loc2), adjacent(loc2, loc1), at(r1,loc2), occupied(loc2), unloaded(r1)}

Qual predicado é invariante?

A

adjacent

B

at

C

occupied

D

in

E

top

A alternativa A está correta.

O predicado adjacent é invariante porque as plataformas (loc1 e loc2) estão fixas. Por isso, seu valor nunca é mudado. Predicado invariante é aquele que não muda, que é uma relação fixa.

Considerações finais

Iniciamos nosso estudo entendendo o problema de criar agentes inteligentes que agem deliberadamente com o objetivo de atingir um objetivo. Foram introduzidas diversas terminologias e conceitos a respeito dos agentes e ambientes, e fizemos uma formulação matemática do problema. Expusemos todos os tipos de restrições que podem ser feitas ao modelo apresentado.

A seguir, analisamos problemas de satisfação com restrições, estudamos algoritmos de resolução e heurísticas que podem ser implementadas nessa difícil tarefa. No fim, poderemos aplicar os algoritmos apresentados em problemas diversos.

No passo seguinte, nós introduzimos uma linguagem de programação lógica. O foco foi na resolução de problemas de restrições com um paradigma declarativo de computação.

Por fim, apresentamos como representar estados de um problema de projetar agentes inteligentes e conseguimos propor métodos para o planejamento clássico.

Podcast

Neste podcast, ouça uma entrevista sobre Planejamento em Inteligência Artificial.

Conteúdo interativo

Acesse a versão digital para ouvir o áudio.

Explore +

Pesquise sobre outros problemas interessantes de satisfação com restrições, como o problema das oito rainhas (queremos dispor oito rainhas no tabuleiro de xadrez, sem que uma ataque a outra), o famoso jogo Sudoku, entre outros jogos.

Para aprender mais sobre Prolog, faça a trilha do tutorial completo de Prolog Learn Prolog Now!

Referências

NAU, D.; GHALLAB, M.; TRAVERSO, P. **Automated Planning**: Theory & Practice. San Francisco, CA: Morgan Kaufmann Publishers Inc., 2004.

RUSSELL, S. J.; NORVIG, P.; DAVIS, E. **Artificial intelligence**: a modern approach. 3. ed. Upper Saddle River, NJ: Prentice Hall, 2010.

STERLING, L.; SHAPIRO, E. **The art of Prolog**: advanced programming techniques. 2. ed. Cambridge, MA: MIT Press, 1994.