Estudos para a P2 de MAC0425

Graphplan

Motivação

A maioria dos algoritmos de planejamento são ineficientes porque o *fator de ramificação* fica muito alto, ou seja, são gerados muitos estados que não levam ao estado meta. Existem duas jeitos de contornar esse problema:

1. Criar um problema relaxado: removendo algumas restrições do problema original, teremos um problema mais fácil de ser resolvido.
2. Fazer uma versão modificada da busca original: incluindo somente as ações que ocorrem no problema relaxado.

Construção

O planejador *Graphplan* alterna entre duas fases:

1. Expansão do grafo de planejamento: cria um grafo e checa se há condições para um plano solução.
2. Extração da solução: Busca regressiva, considerando apenas as ações do grafo.

O grafo de planejamento é um esquema com os estados e as ações possíveis alinhadas em seus níveis. Os **níveis de estados** contém todos os estados possíveis no tempo i (pode incluir p e $\neg p$). Os **níveis de ações** contém todas as ações que tem as precondições satisteitas no tempo i.

Além das ações do problema, o *Graphplan* também leva em conta as ações de manutenção, que são para o caso de um literal permenecer inalterado.

Um exemplo ilustrativo é o exercício 10.9 da Lista 3.

Vale lembrar que o nível de estado s_0 do Graphplan contém todos os literais iniciais e as negações dos outros que não estão em s_0.

Exclusão Mútua (Mutex)

Duas ações no mesmo nível são mutex por:

- Efeitos inconsistentes: um efeito de uma ação nega um efeito de outra.
- Interferência: uma ação elimina uma pré-condição de outra.

- Necessidades que competem: ações com pré-condições que estão em mutex.

Caso contrário, as ações não interferem entre si, isto é, ambas podem fazer parte de um plano solução.

Dois literais num mesmo nível são mutex por:

- Suporte inconsistente: um é a negação do outro
- Efeitos que competem: todas as maneiras de satisfazê-los são mutex

Extração da Solução

Para acharmos a solução no nível de estado s_j, precisamos ver se os literais da meta estão nesse nível. Além disso, eles não podem estar em mutex entre si.

Então, fazemos uma escolha não-determinística de uma ação para ser usada no estado s_{j-1} para atingir uma das metas.

Se qualquer par possível de ações está em mutex então fazemos um backtracking.

Depois disso, usamos recursão para os outros níveis de estado.

Análise

O conjunto de literais e o de ações crescem monotonicamente. Isso ocorre porque se um literal/ação aparece num nível, ele aparecerá nas camadas subsequentes. Já o de mutex decresce monotonicamente, por causa da propagação recursiva de mutexes.

Com isso, chegaremos num ponto i que possui o mesmo número de estados que o ponto $i-1$. Isso é um ponto fixo que garante a convergência do algoritmo.

O Graphplan só olha as ações do grafo de planejamento na busca da solução, que é um espaço de busca menor que o POP. No entanto, ele pode criar muitos estados irrelevantes. Mesmo assim, ele é mais eficiente do que o POP.

Exercício

Action (Eat(Cake))

- Precond: Have(Cake)
- Eff: $\neg$Have(Cake) $\wedge$ Eaten(Cake)

Action (Bake(Cake))

- Precond: $\neg$Have(Cake)
- Eff: Have(Cake)

Init (Have(Cake))
Goal (Have(Cake) $\wedge$ Eaten(Cake))

Busca Heurística

Uma busca heurística trabalha com o espaço de estados de um problema P usando uma função heurística $h(s)$ a partir de um estado s. A ideia é que $h(s)$ seja extraída automaticamente do problema. Ela permite a solução de problemas gigantescos.

Vamos extrair heurísticas de problemas relaxados. A relaxação mais comum é a **P+**. Ela elimina todas os efeitos negativos das ações de P. Ela é intratável, mas é fácil aproximá-la e ela é uma heurística admissível.

Heurísticas de plano serial

h_add

Ela ignora os efeitos negativos das ações e aproxima a distância para a meta g como a soma das distâncias das proposições presentes em g. Portanto, para atingirmos o átomo p a partir do estado s, teremos:

$$h(p,s) = \begin{cases} 0, & \text{se } p \in S \\ min_{a \in A}(1 + h(precond(a), s) | p \in eff^+(a)), & \text{caso contrário} \end{cases}$$

Assumimos independência. Seja C um conjunto de átomos:

$$h(C,s) = \sum_{p \in C} h(p,s)$$

Ela não é admissível, porém é rápida. Para problemas muitos difíceis, uma solução sub-ótima é suficiente.

h_max

Análoga à heurísitica anterior, mas para um conjunto C de átomos:

Assumimos independência. Seja C um conjunto de átomos:

$$h(C,s) = max_{p \in C}(h(p,s))$$

O código em Python dessas heurísticas está no arquivo **heuristics.py** do EP2.

Heurísticas de plano de ordem-parcial

Custo de nível

Essa heurística é usada no grafo de planejamento. Lembrando que se algum literal da meta não aparece no último nível do grafo, o problema não tem solução.

Podemos estimar o custo de alcançar qualquer literal como sendo o nível que ele aparece pela primeira vez.

Uma heurística admissível, porém não leva em conta o número de ações.

Custo de nível máximo

Leva em conta o custo de nível máximo entre as submetas de uma conjunção. Admissível porém pouco informativa.

Soma de custo de nível

Calcula a soma custo de nível entre as submetas de uma conjunção. Não é admissível mas funciona bem na prática.

Heurísticas do planejador FF

Ela ignora os efeitos negativos, resolvendo o problema relaxado em tempo polinomial. Usa o tamanho (número de níveis) do plano relaxado como valor de heurística.

Como não existem efeitos negativos no problema relaxado, não existem mutexes, o que faz com que tanto a expansão do grafo de planejamento quanto a extração da solução sejam polinomiais.

Usamos o algoritmo do *Graphplan* e usamos a heurística para fazer a busca regressiva ser mais rápida. Ela não é admissível, mas é informativa. A maioria dos planejadores atuais de sucesso usam variações dessa heurística.

Incerteza e Raciocínio Probabilístico

A lógica não serve para todos os aspectos do mundo. Em algumas vezes, temos que considerar incertezas, porque não conhecemos todos a "dinâmica" do mundo, não temos acesso a tudo, o modelo pode ser muito complexo, etc. Uma boa medida para incerteza é a *probabilidade*.

Faremos uma Base de Conhecimento baseada em um Espaço de Probabilidades. As conclusões virão por inferência. Usaremos os axiomas conhecidos da Teoria de Probabilidade (apenas variáveis discretas).

Lembrando que:

- Marginalização (*summing out*) é achar as probabilidades das marginais.
- Produto (*join*) é o produto matemático normal. Atente que:
  - $P(T) \cdot P(W) = P(T, W)$
  - $P(T) \cdot P(W|T) = P(T, W)$
- Condicionamento e Normalização: Achamos a marginal de uma variável e depois normatizamos.
- Regra de Bayes: $P(A|B) = \frac{P(B,A) \cdot P(A)}{P(B)}$

Redes Bayesianas

São uma representação compacta de distribuições conjuntas complexas através de distribuições condicionais locais. Para o esquema, são usados grafos direcionados acíclicos, onde os nós são as variáveis e as arestas são as dependências.

Um modelo probabilístico dinâmico pode ser visto como uma representação de um modelo de transição de estados juntamente com um modelo de observação

A propriedade de Markov aplicada em modelos dinâmicos implica que o futuro é independente do passado dado o estado presente

Modelos estacionários mantêm a mesma dinâmica em cada instante de tempo

Planejamento Probabilístico

Processos de Decisão Markovianos

No planejamento probabilístico, em vez de executarmos escolhermos ações de maneira determinística, escolhemos de maneira estocástica.

O sistema é estocástico, porém controlável, já que as funções de transição dependem da ação a ser tomada. A descrição da meta pode ser de alcançabilidade ou otimização (se estamos usando funções de custo ou de recompensa). Temos um conceito mais geral de planos, as *políticas*.

Portanto, um MDP é um modelo de transições baseado em probabilidade em que:

- as funções de transição e de custo são totalmente conhecidas
- o estado do sistema é totalmente observável.

Pela Suposição de Markov de 1ª ordem, podemos afirmar que a probabilidade do próximo estado só depende do atual.

MDP é uma tupla $< S, D, A, T, C >$:

- S: um conjunto (finito) de estados
- D: uma sequência de passos de tempo discreto (1,2,3,...,L), L pode ser finito ou infinito (estágios de decisão)
- A: um conjunto (finito) de ações
- $T : S \times A \times S \to [0, 1]$, uma função de transições probabilísticas estacionária
- $C : S \times A \times S \to R$, uma função de custo estacionária

A função de transição representa a probabilidade condicional $P(s_t|a, s_{t-1})$.

Para cada estágio queremos encontrar uma política π, que será a solução ($\pi : S \times D \to A$). Executar uma política leva a uma sequência de custos ou recompensas ao passar do tempo. Queremos políticas de maior recompensa/menor custo.

Para avaliar a solução de um MDP, definimos uma função valor:

$$V^{\pi}(s,t) = u^{\pi}(C_1, C_2, ...) = E(C_1 + \gamma C_2 + \gamma^2 C_3 + ...), \text{ onde } \gamma \text{ é o fator de desconto.}$$

Podemos enxergar que:

- $0 < \gamma < 1$: o agente dá mais importância aos custos passados.
- $\gamma = 1$: a importância para todos os custos é a mesma.
- $\gamma > 1$: o agente dá mais importância aos custos recentes.

π^* é uma política (solução) ótima se $V^*(s,t) \geq V^{\pi}(s,t)$ para todo π, s, t.

Existem três tipos de MDPs:

1. MDPs de horizonte finito.
2. MDPs de horizonte infinito e recompensa descontada.
3. Stochastic Shortest-Path MDPs (SSP MDPs): o que mais usamos no curso, pois é o mais apropriado para planejamento. Ele é um conjunto que contém os outros dois.

SSP MDPs

MDP é uma tupla $< S, (D), A, T, C, s_0, G >$:

- S: um conjunto (finito) de estados
- (D: uma sequência de passos de tempo discreto (1,2,3,...,L))
- A: um conjunto (finito) de ações
- $T : S \times A \times S \to [0, 1]$, uma função de transições probabilísticas estacionária
- $C : S \times A \times S \to R$, uma função de custo estacionária
- s_0: o estado inicial.
- G: é o conjunto de estados metas absorventes.

Fazemos duas suposições:

1. Existe uma política própria (alcança um estado meta com $P = 1$)
2. Todas as políticas impróprias implicam em um custo infinito.

Avaliação de Política

Para avaliarmos a política, resolvemos de um sistema de equações lineares.

$$V^{\pi}(s) = \begin{cases} 0, & \text{se } s \in G \\ \sum_{s' \in S} T(s, \pi(s), s')(C(s, \pi(s), s') + V^{\pi}(s')), & \text{caso contrário} \end{cases}$$

Que é um algoritmo cúbico, muito ineficiente. Para accelerar, usamos um refinamento iterativo:

$$V^{\pi}(s) = \sum_{s' \in S} T(s, \pi(s), s')(C(s, \pi(s), s') + V^{\pi}_{n-1}(s'))$$

.
É garantido que V^{π}_n converge para V^{π}.

MDP de horizonte infinito

Em geral, um MDP não envolve estados iniciais e meta.

- Estados Iniciais: em extensões de MDPs, é mais comum ser dada uma distribuição de probabilidades sobre um conjunto de estados iniciais
- Estados Meta: num MDP clássico, a meta pode ser modelada através da função custo (ou recompensa)

Portanto, o objetivo aqui é encontrar uma política ótima que maximize a recompensa aditiva descontada esperada, usando aquela função valor.

Seja $V^*(s)$ o valor descontado esperado máximo alcançável no estado s.

Seja $Q^*(a, s)$ o valor de um estado se forçamos o agente a escolher a ação $a_i \in A$, mas agir otimamente no futuro, por exemplo:

$$Q^*(a_i, s) = R(s) + \gamma \sum_{s' \in S} T(s, a_i, s') \cdot V^*(s')$$

Logo, $V^*(s)$ deve atender à equação de Bellman:

$$V^*(s) = max_{a \in A}(Q^*(a, s))$$

Algoritmo de Iteração de Valor

Usaremos a fórmula do refinamento iterativo (Bellman Backup), e vamos iterar até V_{t+1} se aproximar de V^*, a menos de um erro ϵ. (Erro de Bellman). Devolveremos o valor máximo de $Q(s, a)$ e a ação que o maximiza (argmax).

RTDP e LRTDP

O RTDP é semelhante ao Iteração de valor mas usa os mínimos ao invés dos máximos.
Já o LRTDP supõe alcançabilidade e é dirigido por uma heurística (política gulosa).

Aprendizado por Reforço

O aprendizado modifica o mecanismo de decisão do agente para melhorar seu desempenho. É essencial em ambientes desconhecidos e é útil como um método de construção de um sistema. Nós inserimos o agente diretamente no ambiente para explorá-lo e aprender a partir dele.

As experiências podem ser avaliadas com algum critério. O projeto de um agente depende do retorno disponível.

- Aprendizado supervisionado: são oferecidas avaliações corretas para cada experiência.
- Aprendizado não supervisionado: não é oferecido retorno.
- Aprendizado por reforço: são oferecidas avaliações graduais, à medida em que os eventos ocorrem.

No aprendizado por reforço, supomos que o ambiente se comporta como um MDP, mas queremos que o agente aprenda a agir otimamente, sem conhecer as funções de transição e de probabilidade no início.

Temos 2 opções para o Aprendizado por Reforço:

- Opção A: O agente aprende o modelo do mundo (calculando média de valores tanto para a função de transição quanto para a de probabilidade) e resolve o problema usando algoritmos conhecidos. No entanto, estimar a cada iteração deixa o algoritmo muito lento e estimar a cada n iterações pode deixá-lo muito desatualizado.
- Opção B: Estimar a função Valor diretamente. Essa abordagem é correta e utiliza o algoritmo Q-Learning.

Existem duas motivações que levam o agente a escolher uma ação:

1. Exploração: O agente acredita que a ação lhe trará mais informações sobre o ambiente.
2. Exploitação: Ele acredita que a ação lhe trará bons resultados no ambiente.

No exemplo do *k-armed Bandit*, podemos entender que a melhor forma de determinarmos uma ação é por combinação: sendo $0 \leq \epsilon < 1$, com probabilidade ϵ fazemos uma escolha aleatória e com probabilidade $1 - \epsilon$, escolhemos a melhor ação. Estamos explorando com probabilidade ϵ (tentando as máquinas para conhecer suas probabilidades) e exploitando com probabilidade $1 - \epsilon$. Isso é a estratégia ϵ-greedy.

Q-Learning

Do MDP usaremos a função $Q(s, a)$ para os cálculos. No entanto, aqui não conhecemos as funções T e P. Portanto, usaremos uma "média" durante a aprendizagem.

Recebemos uma experiência (s, a, s', r) que sugere:

$$Q(s, a) = r + \gamma max_{a' \in A} Q(s', a').$$

Então, temos que calcular a média móvel, já que queremos relacionar com as experiências anteriores:

$$Q(s, a) = (1 - \alpha)Q(s, a) + \alpha(r + \gamma max_{a' \in A} Q(s', a')).$$

Ou seja, o agente aprende a função $Q^{\pi}(s, a)$ e consequentemente $\pi^*(s)$, calculando o argmax.

O α é a taxa de aprendizado, que é o quanto a nova experiência influi no valor antigo. Com um α pequeno, o aprendizado converge lentamente, mas a estimativa não flutua muito. O ideal é começar com um α grande e ir diminuindo.

A convergência para uma solução ótima é garantida se o ambiente for um MDP, se a taxa de aprendizagem for adequada, e se a exploração do ambiente for tal que nenhuma ação seja completamente ignorada. No entanto, ela é lenta.

O planejamento em Aprendizado por Reforço intercala ciclos de aprendizado baseado em experiência no ambiente e conhecimento adquirido com experiências passadas.

O algoritmo Q-Learning depende de um espaço S e A finitos e pequenos o suficiente para armazenar a função Q na memória.

A exploração pode ficar um pouco melhor com Funções de Exploração.

Approximate Q-Learning

Vamos explorar áreas que não sabemos quão ruim são, e eventualmente, parar de explorar. Nossas funções vão ter uma estimativa de valor u e um contador de visitas n, e devolver uma utilidade para otimização, como $f(u, n) = u + \frac{k}{n}$.

Nossa fórmula do Q-Learning fica:

$$Q(s, a) = (1 - \alpha)Q(s, a) + \alpha(r + \gamma f(Q(s', a'), N(s', a')).$$

O algoritmo original mantém informações de todos os estados. No mundo real não fazemos isso. Queremos generalizar, então aprendemos por algumas tentativas e usamos a aprendizagem para novas experiências.

Para implementar f, vamos usar um vetor de *features*, com alguns pesos. Nossa função Q ficará:

$$Q(s, a) = \sum_{i=1}^{k} w_i \cdot f_i(s, a).$$

Portanto, em nosso algoritmo, quando temos uma experiência (s, a, s', r), guardamos a diferença $(\delta = r + \gamma max_{a' \in A} Q(s', a') - Q(s, a))$, calculamos $Q(s, a) = Q(s, a) + \alpha \cdot \delta$ e atualizamos os pesos $(w_i = w_i + \alpha \cdot \delta f_i(s, a))$.

A otimização é garantida pelos mínimos quadrados.

Lembrando de:

- Fitting: Quando a função aproximada acerta valores o suficiente (errando alguns, mas poucos).
- Underfitting: Quando a função erra muitos valores.
- Overfitting: Quando ela parece acertar muitos, mas pode errar muitos outros (está exagerada).

Aprendizado de Máquina

Aprendizado Não-Supervisionado

Ele recebe entradas, mas não exemplos de saída. O agente deve inferir se uma saída é boa.

Aprendizado Supervisionado

O sistema aprende a partir de exemplos fornecidos e vai modificando seus parâmetros para a saída ir chegando no desejado. Na utilização, ele generaliza as entradas e gera as saídas.

Ele recebe um vetor de entradas $D = ((x_1, t_1), (x_2, t_2), ...)$ e tenta mapear x para t a partir de uma função.

É usado para classificação (de rostos, de fala) e regressão (para valores como o da bolsa).

A função deve ter *fitting*. Queremos generalização e não memorização. São exemplos de aprendizado supervisionado Redes Neurais e Árvores de Decisão.

Redes Neurais

Aqui, a ideia é tentar construir modelos matemáticos que se assemelhem com os neurônios, inspirados na arquitetura paralela do cérebro. A inteligência vem do paralelismo massivo de unidades simples.

A metodologia consite em:

1. Modelar um neurônio
2. Construir uma rede de neurônios interconectados
3. Propor "algoritmos de aprendizagem"

O **Perceptron** é usado como classificador linear e funciona assim: recebe duas entradas, dois pesos e uma entrada de viés. Ele devolve se $w_0 + w_1 \cdot x + w_2 \cdot y$ é maior, menor ou igual a algum valor (função de ativação). Um neurônio desses consegue separar duas classes, e portanto, n neurônios separam 2^n classes.

Um bom exemplo de aplicação são as funções booleanas E e OU. No entanto, o Perceptron não consegue fazer a XOR.

O **MLP(MultiLayer Perceptron)** consegue a partir de conjuntos de Perceptrons alinhados em diversas camadas. Eles podem montar áreas mais complexas de classificação. Possuem pelo menos uma camada oculta (que não é nem a entrada nem a saída). As funções de ativação continuam iguais para cada Perceptron, mas podem ser lineares ou senoides.

Já o algoritmo **Backpropagation** aprende os pesos para uma rede multicamadas, dada uma rede com um número fixo de unidades e interconexões.Ele emprega a descida do gradiente para minimizar o erro quadrático entre a saída da rede e os valores alvos para estas saídas. O treinamento para quando o erro no conjunto de validação começa a aumentar.

Aprendizado por indução

Usamos algoritmos de extrapolação (tipo em Métodos Numéricos) mas não com funções, e sim com conjuntos de exemplos. Criamos uma aproximação h para f, que é a hipótese de indução. Um viés é a preferência por qualquer hipótese. Qualquer conhecimento criado pela generalização de fatos específicos não pode ser provado verdadeiro, mas só pode ser provado falso.

O elemento de aprendizado é um conjunto de treinamento contendo exemplos positivos e negativos sobre um conceito (classe ou categoria), encontrar uma maneira que pode classificar futuros exemplos como positivo e negativo.

Já o elemento de execução é um conjunto de exemplos de algum conceito, e o agente deve determinar se um novo exemplo dado é uma instância do conceito ou não.

Árvore de Decisão

É um dos métodos mais usados e mais simples de serem representados.

A entrada é uma situação descrita por um conjunto de atributos (características ou features) e seus valores. A saída é uma decisão V/F. A função aprendida é uma função booleana representada por uma árvore de decisão (ou conjunto de regras se-então). Cada nó na árvore corresponde a um teste do valor de um atributo (multi-valorado). Cada ramo a partir de um nó é rotulado com os possíveis valores do teste. Os valores nas folhas da árvore representam as possíveis decisões finais (V/F).

Qualquer função booleana pode ser representada como uma árvore de decisão. Cada linha na tabela verdade é um caminho na árvore. Logo, o crescimento é exponencial.

O exemplo positivo é uma entrada com saída verdade. O negativo possui saída falsa.

A maneira eficiente de fazer os testes é fazendo os testes de atributos. Vamos do melhor atributo até os piores. Um atributo é o melhor quando ele melhor separa os dados da

entrada. A medida usada aqui é a entropia, que é dada por:

$$\text{Entropia}(X) = (p+) \cdot log_2(p+) - (p-) \cdot log_2(p-), \text{ onde:}$$

- $p+$ é a proporção de efeitos positivos em X.
- $p-$ é a proporção de efeitos negativos em X.

Com essa função, podemos calcular o Ganho que teremos escolhendo um atributo para separar os dados.

$$\text{Ganho}(X, A) = \text{Entropia}(X) - \sum_{val \in valores(A)}(|X_{val}| \ |X|) \cdot \text{Entropia}(X_val), \text{ onde:}$$

- valores(A): todos possíveis valores do atributo A
- X_{val}: subconjunto de X no qual A tem valor val ($X_{val} = (x \in X | A(x) = val)$)

O algoritmo ID3 analisa todos os exemplos para decidir o melhor atributo classificador.

A melhor representação será aquela que é mais geral e concorda com os exemplos e escreve de forma simples e concisa o maior número de exemplos possível. Queremos evitar a memorização (uma boa forma de fazer isso é identificar padrões).

O cálculo de entropia pode ser generalizado. É só somar a proporção vezes o logaritmo da proporção para todas os valores possíveis.

Algoritmos de aprendizagem: são bons se fazem boas hipóteses sobre a classificação de exemplos não utilizados no treinamento.

Avaliação de desempenho é feita da seguinte maneira:

1. colecione um grande número de exemplos
2. divida os exemplos em: training set e test set
3. use o algoritmo de aprendizado no training set para gerar as Hipóteses de classificação
4. analise a porcentagem de exemplos com o test set que são classificados corretamente com a Hipótese
5. repita passos 1 a 5 para diferentes conjuntos selecionados randomicamente

O algoritmo ID3 funciona assim:

1. incluir (testar) na árvore os atributos mais importantes primeiro
2. depois do primeiro teste dividir os exemplos sendo que cada saída é uma nova árvore de decisão
3. se existem exemplos positivos e negativos, escolha o atributo que melhor os divida (ex.: método do ganho de entropia)

4. se todos os exemplos forem positivos (ou negativos) → fim

5. se não restam exemplos significa que não foram fornecidos exemplos suficientes para aquele caminho → valor default calculado para a maioria dos casos

6. Se não restarem atributos mas sobraram exemplos positivos e negativos: problema (exemplos com mesma descrição mas classificações diferentes!)

Esse algoritmo prefere árvores com altura pequena, pois coloca mais perto da raiz atributos com maior ganho. Existem alguns problemas como ruído nos dados, atributos ineficientes e pouco treinamento. O pior, no entanto, é a ambiguidade, que é quando a mesma entrada no conjunto de treinamento oferece dois resultados diferentes. Podemos contornar esse com probabilidade.

Reforço Profundo

Deep Learning é Representation Learning. Ele tem obtido grande sucesso nos últimos anos em tarefas de percepção, processamento de linguagem natural e aprendizado por reforço. São diversas arquiteturas/modelos/aplicações, porém sempre baseados nas ideias básicas de machine learning: álgebra linear, regressão, otimização, gradiente descendente, back-propagation, entre outros. Basicamente, são muitos dados combinados com computação de alto desempenho. Um exemplo para falar disso são gatos (em visão computacional).