Planejamento Utilizando Busca Heurística

Joaquim Siqueira Lucena
Pontifícia Universidade Católica do Rio Grande do Sul - Escola Politécnica
Av. Ipiranga, 6681 - Partenon
Porto Alegre - RS

Introdução

O planejamento no campo de Inteligência Artificial é uma técnica utilizada para encontrarmos um conjunto de ações que — dado um domínio, um problema alvo e um objetivo — transforma o estado inicial do nosso problema em um estado objetivo. Existem vários formalismos e linguagens de planejamento, como STRIPS e PDDL, que definem sintaxe e semântica para representarmos problemas e planos descritivamente. Esses conjuntos problema-domínio, podem ser representados por grafos de estados de espaço. Nesses grafos os nós representam os estados do nosso problema, enquanto as conexões representam as ações válidas que transformam o estado de um para o outro. Os estados são conjuntos de condições que definem como está disposto o problema naquele instante, enquanto as ações impõem efeitos no estado após serem aplicadas, que podem ser positivos (adição de condições) ou negativos (deleção de condições) e algumas ações possuem pré-condições, isto é, só podem ser executadas se todas as suas pré-condições estiverem atendidas no estado atual. Algoritmos como Backwards Search, que começa no objetivo e tenta retornar ao estado inicial, ou como o GraphPlan, que se basea no relaxamento do problema, por ser mais fácil de resolver que o problema original, são exemplos de algoritmos utilizados dentro de planejadores.

A representação por grafos, somado com uma técnica chamada de relaxamento de grafo (também utilizada no GraphPlan) e de uma função heurística, nos dá a possibilidade de também utilizar algoritmos de busca, como DFS ou BFS para agilizarmos a execução. Neste trabalho, foi utilizada a linguagem PDDL. Foram fornecidos pré-prontos todos os arquivos e parsers necessários para ler e entender arquivos PDDL e transformá-los em código e estruturas de dados utilizáveis em Python, bem como diversos domínios e problemas descritos em PDDL, que nós utilizamos para experimentação. Neste paper entrarei em detalhes sobre a minha implementação em Python de um planejador automatizado utilizando o algoritmo de busca A* e discutirei sobre a performance do planejador baseada em diversos testes.

Função Heurística

A função heurística foi implementada com base no código de Relaxed Planning Graph que estava disponível. Este algoritmo monta um grafo relaxado que utiliza como entrada todas as ações possíveis, o nosso estado inicial e o nosso objetivo e retorna o grafo se o objetivo foi alcançado, ou retorna inatingível caso não tenha sido possível alcançar o objetivo. Para que esse algoritmo nos trouxesse como resposta a heurística máxima, foi necessário alterá-lo para que em vez de retornarmos o grafo, retornássemos o nível de profundidade do grafo (que representa a heuristica máxima), ou retornássemos infinito, caso o objetivo não fosse alcançado.

Validador de Plano

O validador de plano tem como função analisar um plano sobre um domínio e um problema qualquer e dizer se esse plano é válido ou não. Um plano somente é válido se todas as ações que estão incluídas nele são aplicáveis e se ao final dele chegamos ao nosso destino, caso contrário ele é inválido. Meu validador recebe na entrada a lista de todas as ações, com seus efeitos e pré-condições, o estado inicial do problema, o objetivo do problema e o plano a ser validado. A validação é feita percorrendo o plano e aplicando as ações nele em sequência. Assim, para cada ação do plano, procuramos seus efeitos e pré-condições na lista de todas as ações possíveis (dado que o plano não carrega essas informações) e vemos se ela é aplicável no estado atual, se for, aplicamos ela e avançamos para a próxima ação. Caso após a última ação o objetivo desejado tenha sido alcançado, o plano é válido. Caso não o objetivo não tenha sido alcançado ou se alguma ação que tentamos aplicar durante o caminho é inválida, o nosso plano é inválido.

Busca A*

A busca A* é o solucionador do problema, tendo como entradas a lista de todas as ações possíveis, o estado inicial do problema e os objetivos, ela retorna um plano ótimo que resolve o problema escolhido. O dicionário closed representa os estados já visitados no grafo. A lista frontier guarda tuplas com os estados do grafo que ainda temos que visitar, A tupla tem o formato de (estado a visitar,heuristica do estado a visitar,tupla de frontier do estado anterior,ação anterior). A ação anterior é a ação que foi aplicada para transformar

o estado anterior no estado a visitar. Enquanto a tupla do estado anterior possui todas essas informações, só que pertinentes ao estado anterior. Essas duas informações são utilizadas no final da execução para percorrermos o caminho construído em sentido contrário e montarmos o plano a partir das informações obtidas através dessa regressão. Para iniciarmos a execução a primeira tupla de Frontier possui o estado inicial, heuristica zero, tupla do estado anterior e ação anterior como None, após isso entramos no while que construirá o caminho do plano.

Enquanto frontier não estiver vazio procuramos nela qual é o nó com menor heurística, utilizando a função search_lowest_h(), que busca o índice de frontier com menor heurística, e com este índice, removemos ele da lista, atualizamos o estado sendo visitado atualmente com o valor removido, e criamos uma chave no dicionário closed para representar que ele já foi visitado. Após essa operação, nós testamos todas as ações, quais delas podem ser aplicadas, e para cada uma delas, geramos um estado novo depois de aplicá-las, calculamos a heurística deste novo estado, e verificamos se o estado novo já foi visitado. Caso não tenha sido, nós montamos a tupla da frontier, com (estado novo;heuristica nova;toda a tupla do estado atual, não apenas o estado atual;ação aplicada agora) e inserimos na lista. Após fazermos isso com todas as ações, tudo se repete, até que frontier fique completamente vazio, que nesse caso não existe um plano que resolve o problema, ou até que o estado atual seja igual ao objetivo. Neste caso a função get_path() monta o plano com a regressão pelo caminho encontrado.

A função get_path() recebe como entrada a tupla do estado em que alcançamos o objetivo, e cria uma lista plan vazia. Enquanto a tupla anterior e a ação forem diferentes de None, ela guarda a ação utilizada da tupla sendo examinada na cabeça de Plan, e itera para a tupla anterior, para poder examiná-la. Desse jeito, quando alcançarmos None, retornamos com o plano pronto.

Resultados

Inicialmente foi testado cada parte da implementação em separado (heurística, validador e busca) e os resultados obtidos foram comparados com o gabarito fornecido no notebook. Após todos os resultados obtidos fossem iguais aos gabaritos, passamos para a fase de testes de performance. Para esses testes de performance, foram utilizados todos os domínios e problemas pré-disponibilizados. Estes testes consistem, em uma primeira parte, da geração do plano utilizando a busca A* para um problema e um domínio escolhidos. Após gerarmos o plano, utilizamos o nosso próprio validador para dizer se o plano é aplicável ou não. Depois que todos os planos gerados forem validados com sucesso, partimos para a segunda fase, onde repetimos o processo de solução, mas agora medindo o tempo total de execução do algoritmo. Foram testados todos os 4 domínios e todos os problemas de cada domínio. Para problema 1 do domínio DWR, foi obtido um tempo de execução entre 16-18 segundos. Uma implementação eficiente teria um tempo de execução de cerca de 20 segundos dependendo da máquina. Os problemas dos outros domínios eram mais simples de se resolver, então os tempos de execução ficaram em torno de alguns milisegundos(com exceção do problema 6 do domínio blocksworld que levou cerca de 7 segundos para encontrar a resposta).

A última parte do notebook consiste em um último teste de performance, onde além de medirmos apenas o tempo de execução, é medido também a quantidade de nós visitados até encontrarmos a solução e quantas vezes é chamada a função heurística (nocaso do A*). Além disso também seria feita uma comparação da performance da minha implementação com a performance de uma implementação do BFS. Infelizmente, quando eu tentei rodar este último teste, o notebook acusou um erro que está faltando um arquivo chamado pddl.bfs_planner, que pelo nome eu diria que é o módulo em python que construiria a versão BFS do plano. Não ficou claro se era necessário que a implementação ficasse por nossa conta, ou se era para ela estar juntos com os arquivos fornecidos. Foi feita uma tentativa de implementação, mas surgiram diversos erros que eu não consegui resolver, relacionados a dependências de arquivos e de questões relacionadas com Python, além de eu acabar quebrando meu código implementado antes. Por causa desses problemas, as mudanças não foram colocadas no repositório do GitHub e não foi possível realizar esta etapa dos testes de performance.

Conclusão

Este trabalho foi importante para que ficasse mais claro o conteúdo visto em aula, e para que trouxesse um exemplo prático do conteúdo, pois realmente com apenas as aulas teóricas é um conteúdo difícil de abraçar. Creio que agora eu seja capaz de implementar outras formas de resolução de problemas e domínio, sem ser a resolução por A* por exemplo. Sobre o planejador e a performance dele, é possível deixá-lo mais otimizado. Algumas partes do código eu fiz pensando em clareza e facilidade de uso para que eu conseguisse implementar com menos dificuldades. Mas por exemplo, trocarmos a lista Frontier por um dicionário certamente deixaria a execução mais rápida, já que a busca e a adição de novos elementos em um dicionário é muito mais rápida. Outro ponto que possivelmente deixaria o código melhor é em vez de iterarmos sobre toda a lista para procurarmos o próximo estado que queremos visitar, poderia ser feita uma inserção ordenada já na lista, então nesse caso não seria necessário fazer essa busca, já que o nó com menor heurística estaria sempre na cabeça da lista. Outro ponto de otimização de código certamente seria com relação a estruturação do código em Python, que é muito diferente das linguagens que eu estou acostumado(C, C++) e possui estruturas de dados e/ou formas de iterar e descrever loops que possam ser mais apropriadas para o caso, mas que eu não tenho experiência ou conheço o uso. Sobre limitações durante o trabalho eu senti um pouco a falta de uma ferramenta mais poderosa de debugging, mas não sei dizer se é pela minha falta de experiência utilizando o jupyter notebook e por isso não conhecer as capacidades da ferramenta ou se realmente é uma questão pertinente quando se utiliza o notebook.