Busca Aplicada: Sudoku e Escalonamento de tarefas

Visão geral do projeto

Neste trabalho, aplicamos os algoritmos A estrela, busca em profundidade iterativa, subida de encosta e busca pelo menor custo nos problemas de escalonamento de tarefas em dois processadores e no "puzzle" do Sudoku (com grade simples). Os quatro algoritmos citados foram aplicados de maneira que eles são agnósticos para o problema. Isto é desde que recebam as funções, estado e espaço de busca apropriados, eles serão capazes de encontrar uma solução.

Dividimos esse projeto em pacotes python, separados por pastas na raiz desse projeto. Cada um dos algoritmos está implementado na pasta `algorithms`. Na pasta `sudoku` e `task_scheduling` você encontrará as representações de estado e funções pertinentes a cada um dos problemas em específico. Na pasta `tests` estão os testes automatizados, que usamos para ter um certo controle de qualidade e detectar problemas que afetem as aplicações dos algoritmos.

Existem algumas dependências que são necessárias para o uso dos algoritmos criados aqui. Elas estão presentes em `requirements.txt`. Para instalá-las, você pode usar `pip install -r requirements.txt`. Essas dependências existem porque por vezes usamos uma representação de grafo do `networkx` e o `matplotlib` para facilitar a visualização dos grafos e usar uma estrutura de dados para grafo mais robusta, porém sem abstrair com bibliotecas terceiras os algoritmos de busca implementados no trabalho. Executar o código abaixo no seu jupyter notebook também vai instalar tudo:

Requisitos

Os algoritmos de busca que devem ser implementados para resolver o primeiro problema: problema são:

- [x] Busca em Profundidade Iterativa;
- [x] Busca de Menor Custo (ou Busca de Custo Uniforme);
- [x] Busca A*;
- [x] Subida de Encosta (hill climbing) com movimentos laterais em caso de estagnação (ótimo local).

O segundo problema a ser implementado será o problema de escalonamento de tarefas em arquiteturas multiprocessadores. Os algoritmos de busca que devem ser implementados para resolver o segundo problema são:

- [x] Busca em Profundidade Iterativa;
- [x] Busca Gulosa;
- [x] Busca A*.

Especificações Gerais

- [x] **A)** A linguagem utilizada e as implementações devem ser as mesmas para os dois problemas. Por exemplo, o código do algoritmo A* que resolve o 1º problema deve ser o mesmo que resolve o 2º problema.

- [x] **B)** Para a utilização dos algoritmos de busca informada, será necessário estabelecer uma estimativa de custo final para cada problema.
- [x] **C)** Para os algoritmos de busca local também é necessário pensar em operações de vizinhança e uma função de avaliação.
- [x] **D)** Nos algoritmos de busca local, o programa também deve apresentar a sequência de passos desde a solução inicial até a obtenção da solução final.
- [x] **E)** Nos algoritmos de busca em árvore, deve-se evitar o processamento redundante na busca: Durante a busca, um mesmo estado pode ser visitado muitas vezes. Para melhorar a eficiência da busca é preciso evitar o processamento de nós repetidos. Se um nó já foi visitado então ele não deve ser incluído na lista de nós a serem visitados. Para tal, o código deve manter registro dos estados já visitados.
- [x] **F)** Para cada experimento, o programa deve imprimir algumas informações:
  - [x] **(a)** solução encontrada (sequência de estados desde o nó inicial até a meta);
  - [x] **(b)** número total de nós visitados (nós gerados que já foram testados e para os quais, se possível, foram gerados seus sucessores),
  - [x] **(c)** a profundidade em que a meta foi encontrada;
  - [x] **(d)** custo da solução.

Exemplo de saída:

```
Solução: BBXPP-BBPXP-BXPBP-BPXBP-XPBBP
Número de nós visitados: 20
Profundidade da meta: 4
Custo da solução: 6
```

Além dos códigos um relatório também deverá ser escrito contendo, no mínimo:

- [x] Introdução. Seção que descreve os problemas a serem resolvidos;
- [ ] Experimentos. Seção que descreve os experimentos realizados para cada um os algoritmos em cada um dos problemas investigados. Apresentar alguns exemplos de execução para cada problema (variando a condição inicial). Esta seção deve incluir um gráfico mostrando o tempo gasto pelo computador, para cada um dos algoritmos especificados;
- [x] Conclusão. Seção que conclui o relatório mostrando as dificuldades encontradas e observações pessoais a respeito do trabalho desenvolvido.

In [ ]:
```
%pip install -r ../requirements.txt
```

Um makefile também está disponível para rodar os testes com um comando simplificado, e limpar os arquivos de build.

Decidimos documentar esse projeto todo em inglês, usando a PEP 257 – Docstring Conventions

Experimentos

Separamos o relatório em mais de um documento, usando o Jupyter. Talvez a visualização desse projeto ao todo fique melhor no repositório do github, aqui. Daí basta clicar nos arquivos `.ipynb` que você quiser ver. Mas vamos exportar tudo em PDF, então a lógica abaixo é a mesma, só que mudando a extensão dos arquivos para `.pdf`.

O relatório referente ao problema do Sudoku está, em relação á raiz, em `relatorio/sudoku.ipynb`. Se estiver pelo github, ele pode ser acessado clicando aqui

Já o relatório, do problema de escalonamento de tarefas, está em `relatorio/task_scheduling.ipynb` e mais uma vez, se for pelo github, pode ser acessado aqui

Conclusão

Esse trabalho teve algumas dificuldades. A primeira delas foi criar os algoritmos de busca de uma maneira que eles pudessem funcionar com os dois problemas, que são bem diferentes. Isso exigiu que eles fossem representados sempre com abstrações em grafos. Esse requisito de manter os algoritmos agnósticos foi o que deu razão ao uso dos testes automatizados. Eles evitaram bastante dor de cabeça desnecessária, garantindo que uma mudança que conserta um problema no sudoku por exemplo, não criaria outro em um lugar diferente no escalonador. Da maneira que separamos os algoritmos em funções passadas como parâmetros foi possível testá-las unitariamente para ter certeza absoluta que estavam devolvendo os valores corretos.

Definir boas heurísticas também é complicado. Existiam algumas diferentes para o problema do escalonador especialmente: Caminho crítico, completude mais recente, média de recursos usados, tempo mínimo de execução restante.

A visualização dos problemas é um pouco complicada também, especialmente do problema do escalonador. O gráfico de Gantt ajuda bastante a entender o que está acontecendo nos processadores de fato. No sudoku é mais fácil porque ele é simplesmente uma matriz.

Apesar de não ser do escopo do trabalho, também criamos funções geradoras de problema. Uma delas gera grafos de escalonamento e a outra gera sudokus incompletos.

O escalonamento de tarefas, quando as dependências de tarefas são múltiplas, precisa de um certo tratamento especial para que seja gerado o gráfio de gantt adequado.