CI209 - Inteligência Artificial

Prof. Aurora Pozo
DInf - UFPR
2020/2021

Trabalho prático 1

Neste trabalho você irá usar técnicas de inteligência artificial para resolver o Problema do Caixeiro Viajante (PCV), ou *Travelling Salesman Problem* (TSP) em inglês. Neste problema, o objetivo é encontrar o menor caminho para percorrer um conjunto de cidades (visitando uma única vez cada cidade) e retornar à cidade de origem.

Você deverá implementar (em python) o algoritmo A* e o algoritmo genético, que serão aplicados ao PCV. Deverá também elaborar um relatório de até 3 páginas comentando sobre os aspectos da implementação e os resultados observados durante os testes.

O arquivo compactado `codigo.tar.gz` contém os seguintes arquivos:

- `tsp.py`
- `a_star.py`
- `genetic_algorithm.py`
- `run.py`
- `instances/`

Neste trabalho, você precisará alterar somente os arquivos `a_star.py` e `genetic_algorithm.py`. Para realizar os testes com o A* e o algoritmo genético sobre as instâncias contidas no diretório `instances/`, execute o comando:

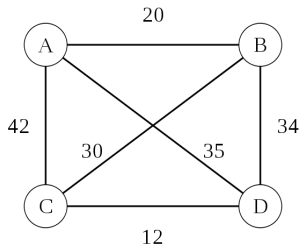
```
$ python3 run.py
```

O comando acima executa o A* uma vez sobre as 3 instâncias pequenas e o algoritmo genético 5 vezes sobre cada uma das 6 instâncias contidas no diretório. É necessário executar o genético mais de uma vez devido à sua natureza estocástica. O comando também irá salvar imagens com os gráficos da evolução de aptidão retornada pelo algoritmo genético. Para isso, é necessário instalar a biblioteca `matplotlib`:

```
$ pip install --user matplotlib
```

Problema do Caixeiro Viajante

O problema pode ser representado a partir de um grafo completo onde cada vértice é uma cidade e cada aresta é a distância entre as duas cidades.



Uma solução pode ser representada por uma permutação que indica a ordem (caminho hamiltoniano) que as cidades devem ser visitadas (Ex: a permutação (A, D, C, B) representa o caminho $A->D->C->B->A$). Dessa forma, o custo desse caminho se dá pela soma de cada distância percorrida entre duas cidades (Ex: 35+12+30+20). O PCV é um problema NP-difícil, portanto ainda não se conhece um algoritmo que o resolva em tempo polinomial.

O arquivo `tsp.py` implementa a classe `TSP` que é responsável por ler um arquivo de instância e fornecer os métodos necessários para a sua implementação dos algoritmos:

- *TSP.evaluate(solution)*: Calcula o custo/aptidão de uma permutação. Retorna um inteiro
- *TSP.random_path()*: Gera uma permutação aleatória. Retorna uma tupla com os identificadores (números inteiros) das cidades
- *TSP.get_start_state()*: Retorna o estado inicial para a busca do A*. O estado inicial é uma tupla com um único valor (cidade inicial)
- *TSP.is_goal_state(state)*: Verifica se o estado é o estado final para o A*. O estado final é quando o caminho está completo (a permutação inclui todas as cidades). Retorna um booleano
- *TSP.get_next_states(state)*: Retorna uma lista de tuplas: (próximo estado, próxima cidade, distância da cidade atual até a próxima)
- *TSP.get_heuristic(state)*: Calcula a heurística do estado para o A*. Retorna um inteiro

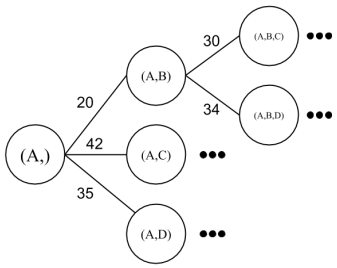
Divisão do trabalho

Parte 1 *A**

Você deverá alterar a função *a_star* no arquivo *a_star.py* e implementar o algoritmo A*, que deve retornar o caminho (permutação) encontrado. A função recebe como parâmetro um objeto `problem` da classe TSP. Você pode testar tua implementação em uma instância com o seguinte comando:

```
$ python3 a_star.py
```

Como o A* é um algoritmo para encontrar o caminho mínimo entre dois vértices e não uma rota, é necessário adaptar a formulação do problema para um problema de busca por caminho mínimo. Dessa forma, o A* realizará a busca sobre um grafo em que cada vértice representa uma permutação parcial. Sendo assim, busca-se o caminho mínimo entre uma solução com uma única cidade e uma solução completa (uma rota).



Para obter os próximos estados a partir do atual, utilize o método `problem.get_next_states(state)`, que retornará uma lista de tuplas, cada uma contendo o próximo estado, a próxima cidade a ser visitada, e a distância da cidade atual até a próxima. Atenção: o estado é representado pelas cidades já visitadas até aquele estado, porém em ordem embaralhada. Ou seja, o estado **NÃO** representa o caminho encontrado, que deverá ser obtido através de *backtracking*. Para isso, você deverá de alguma forma ser capaz de identificar o caminho que levou até determinado estado (pode usar a classe `Node()` fornecida no código para isso)¿

A heurística de um estado (`problem.get_heuristic(state)`) é calculada de acordo com as cidades ainda não visitadas da seguinte forma: menor distância entre uma cidade não visitada e a atual + custo da Árvore Geradora Mínima

das cidades não visitadas + menor distância entre uma cidade não visitada e a cidade inicial

Dicas

- Utilize a classe Node() fornecida no arquivo `a_star.py` para representar os nodos da busca e suas respectivas informações necessárias para o algoritmo. Essa classe faz sobrecarga dos operadores >, >=, < e <= para poder comparar dois nodos em função de seus custos.
- Você pode gerenciar a fronteira de estados com uma estrutura de Fila de Prioridades. Para isso, utilize a classe `PriorityQueue()` que está importada no arquivo. Adicione novos nodos com o método `put(<objeto>)` e obtenha o nodo com a maior prioridade (menor custo) usando método `get()` da classe.
- Testar a implementação com as instâncias maiores poderá consumir toda a memória RAM do computador.

Parte 2 *Algoritmo Genético*

Você deverá alterar a função *genetic_algorithm* no arquivo *genetic_algorithm.py* e implementar o algoritmo genético, que deve retornar a melhor solução (permutação) encontrada e uma lista com o melhor valor de aptidão de cada geração. A função recebe como parâmetro um objeto `problem` da classe TSP, o tamanho da população (`pop_size`) e o número máximo de gerações (`max_gen`). Você pode testar tua implementação em uma instância com o seguinte comando:

```
$ python3 genetic_algorithm.py
```

A arquivo também fornece a classe `Individual` para armazenar uma solução e sua aptidão (com sobre dos operadores de comparação), e o método `random_population()` que retorna uma lista (do tamanho da população) de objetos do tipo `Individual` com permutações aleatórias.

Você implementará os seguintes componentes do algoritmo genético (no mínimo um operador de cada):

- Seleção de pais (Ex: roleta, torneio, etc)
- Cruzamento (Ex: *order crossover*, *order based, position based*, *partially mapped*, *cycle*, etc)
- Mutação (Ex: inserção, troca, etc)

- Seleção de indivíduos para a próxima geração (geracional, uniforme, competição, elitismo, etc)

Os testes serão executados utilizando uma população de tamanho 50 por 2000 gerações. Os demais parâmetros (taxa de cruzamento, taxa de mutação, etc) serão definidos por você. Todos os detalhes da implementação e metodologias (inclusive as que forem além das especificadas neste enunciado) deverão ser descritos no relatório. Os 5 trabalhos que obtiverem o melhor desempenho receberão pontuação bônus.

Dicas

- Implemente cada componente em um método separado para depois construir o algoritmo genético.
- Para gerar um valor aleatório entre [0, 1) utilize o método `random.random()`

Parte 3 *Relatório*

Crie um relatório de até 3 páginas detalhado tua implementação e explicando as suas conclusões em relação às diferenças observadas entre os algoritmos. Compare as qualidades e desvantagens do A* e Algoritmo Genético para os cenários e explique os possíveis motivos para os resultados obtidos (tempo e qualidade da solução).

Se preferir, utilize figuras, tabelas entre outros recursos. Recomenda-se o uso do LaTeXpara construir o seu relatório.

Entrega

Você deverá entregar um arquivo compactado com o nome *login.tar.gz*, onde *login* é o nome do seu usuário no sistema do Departamento de Informática, que contenha os seguintes arquivos:

- *a_star.py*
- *genetic_algorithm.py*
- *login.pdf*

Trabalhos atrasados serão aceitos com um atraso máximo de 1 semana com desconto de nota. Após o prazo máximo o trabalho não será mais aceito.

Observações

Você deverá desenvolver e compreender todas implementações feitas no seu trabalho. Não serão admitidos quaisquer tipos de plágio.

Para referência, a tabela a seguir mostra os custos do caminho ótimo de cada instância:

Instância	Custo ótimo
berlin10	4889
berlin15	4967
berlin16	5027
dj38	6656
eli51	426
berlini52	7542

Dúvidas

Dúvidas poderão ser retiradas por e-mail:

bhmeyer@inf.ufpr.br
aldantas@inf.ufpr.br