UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
INE5633 – SISTEMAS INTELIGENTES

LUCAS AUGUSTO ZANICOSKI SERGIO
MARIANY FERREIRA DA SILVA
MÉRCIA DE SOUZA MAGUERROSKI CASTILHO

RELATÓRIO ATIVIDADE PRÁTICA 1

FLORIANÓPOLIS
2022

Lucas Augusto Zanicoski Sergio
Mariany Ferreira Da Silva
Mércia De Souza Maguerroski Castilho

RELATÓRIO ATIVIDADE PRÁTICA 1

Trabalho submetido à matéria de Sistemas Inteligentes da Universidade Federal de Santa Catarina, como critério de avaliação parcial na disciplina. Orientador: Prof. Dr.: Elder Rizzon Santos.

FLORIANÓPOLIS
2022

O código contém comentários detalhados, mas aqui acrescentamos as respostas requisitadas pelo professor.

1. **Qual a representação (estrutura de dados) do estado**

   board_input: É uma lista com a estrutura inicial escolhida pelo usuário. Este array é preenchido conforme a lista de possibilidade de escolhas, armazenadas no array *choices* inicializado com [1, 2, 3, 4, 5, 6, 7, 8, None]. A cada escolha feita pelo usuário o array choices é atualizado, de forma que restem no array apenas opções ainda não escolhidas. O *board_input* é o mesmo para todos os algoritmos implementados.

   current_node: é o Nodo que está sendo analisado pelo algoritmo

   number_of_nodes: é o número total de nodos criados pelo algoritmo

   visited: é a lista de nodos visitados pelo algoritmo

   possibilities: é a lista de nodos que podem ser visitados pelo algoritmo

2. **Qual a estrutura de dados para a fronteira e nodos fechados**

   Ambos (visited, possibilities) são uma lista de nodos, cada nodo é composto pela seguinte estrutura:

   ```
   node = {
         'number': number,
         'board': list of numbers,
         'path': list of nodes
         'cost': number
     }
   ```

   number: O número do nodo (sequencial, usado para identificar o nodo)

   board: Representa o tabuleiro, lista dos números que contém a sequência da configuração do tabuleiro.

   path: O caminho percorrido para chegar ao nodo, é uma lista de nodos

   cost: é o custo calculado pela heurística de custo

3. **Descrição das heurísticas e algumas simulações dos seus valores (pior caso, melhor caso, caso médio); breve descrição sobre suas implementações**

   Heurística simples: Para a heurística simples, incrementamos o custo sempre que o número está fora do lugar, então com um array [1,2,3,4,5,6,None,7,8] teria um total de 3 números fora do lugar esperado, sendo eles os números 7, 8 e o *None*.

   Como o exemplo citado acima, temos o seguinte caso:

```
THE INITIAL STATE OF YOUR BOARD IS:
1 2 3
4 5 6
None 7 8
```

Em nossa solução, no arquivo engine.py, possuímos a *def calculate_simple_cost()* que recebe o *board.* Sua função é calcular o custo apenas somando o número de itens do tabuleiro fora do local definido como solução.

Na *def add_children()*, utilizamos a mesma arquitetura dos algoritmos anteriores porém como agora possuímos um custo de heurística, é necessário adicionar um novo parâmetro ‘cost’. Além disso, é necessário ordenar essa nova lista ‘cost’ por custo.

Como resultado do algoritmo para o board comentado anteriormente, encontramos:

```
NODE NUMBER:  2
HEURISTIC COST:  2
BOARD:
1 2 3
4 5 6
7 None 8


NODE NUMBER:  4
HEURISTIC COST:  0
BOARD:
1 2 3
4 5 6
7 8 None


HEURISTIC COST:  2
COST:  2
```

Heurística complexa: Para a heurística complexa pensamos na seguinte questão - Quantos passos cada peça está da posição correta (meta)? Para isso, precisamos contar quantos passos o algoritmo precisaria dar para cada que elemento esteja na posição desejada.

Por exemplo, considerando um tabuleiro assim:

```
UP_MOVEMENT = [
    1, 2, 3,
    None, 5, 6,
    4, 7, 8
]
```

Nós temos um custo zero para todas as posições que já estão no lugar certo Já para os elementos que estão na posição errada precisaremos calcular os passos necessários para que o elemento esteja na posição certa. Como sabemos a resposta final do tabuleiro, conseguimos preparar a lista de custos para cada posição incorreta de cada item (arquivo cost_map.py).

```
COST_MAP = {
    # goal_index == 0
    0: {
        # actual_index == ?: cost to 0
        0: 0, 1: 1, 2: 2, 3: 1, 4: 2, 5: 3, 6: 2, 7: 3, 8: 4
    },
    # goal_index == 1
    1: {
        # actual_index == ?: cost to 1
        0: 1, 1: 0, 2: 1, 3: 1, 4: 2, 5: 3, 6: 2, 7: 3, 8: 4
    },
    # goal_index == 2
    2: {
        # actual_index == ?: cost to 2
        0: 2, 1: 1, 2: 0, 3: 3, 4: 2, 5: 1, 6: 4, 7: 3, 8: 2
    },
    # goal_index == 3
```

O número 4 está na posição errada, se movermos ele uma vez para cima ele estará na sua posição correta, por isso o valor de custo para mover o elemento 4 é 1.

O número 7 também está na posição errada, se movermos ele uma vez para a esquerda ele estará na sua posição correta, por isso o valor de custo para mover o elemento 7 é 1.

O número 8 também está na posição errada, se movermos ele uma vez para a esquerda ele estará na sua posição correta, por isso o valor de custo para mover o elemento 8 é 1.

None está na posição errada, mas não vamos calcular o custo de mover esse elemento no tabuleiro. Sendo assim o custo total calculado pela heurística seria de 3.

4. **Como foi gerenciada a fronteira, verificações, quais etapas foram feitas ao adicionar um estado na fronteira (explicação das estratégias, respectivos métodos e possibilidades além do que foi implementado);**

A fronteira é uma lista que vai recebendo as novas possibilidades (possibilities) de nodos a se visitar, dependendo do algoritmo escolhido pelo usuário a lista de possibilidades deve ser ordenada, do menor custo para o maior custo.

is_not_know_board checa para cada nodo nas possibilidades e nos nodos visitados que foram passados como parâmetro se o *board* daquele nodo é igual ao board atual, se for, retorna falso se não retorna verdadeiro. Depois utilizamos da amplitude_search ou da depth_search no *array* da fronteira para retornar o primeiro ou o último nodo respectivamente.

5. **Quais os métodos principais e breve descrição do fluxo do algoritmo;**

1- Usuário digita a opção do algoritmo que deseja rodar:

1 - Custo uniforme em amplitude,

2 - Custo uniforme em profundidade

3 - Heurística simples

4 - Heurística complexa

2 - Usuário coloca o *board* do jogo a ser resolvido (considera-se que o usuário coloque somente casos possíveis de resolver).

3 - chama search_solution que tem como parâmetros o número do algoritmo e o *board* inicial e retorna o número de nodos, os nodos visitados, as possibilidades e nodo atual.

4 - search_solution() manda chamar a função correspondente ao número do algoritmo.

search(search_function, initial_board, cost_function = None): é a função genérica da base de busca para custo uniforme e busca com heurística. Dentro dela, setamos as variáveis de number_of_nodes, visited e possibilities, o current_node que é instanciado da maneira explicada na questão 2 e o cost_function. Caso o algorismo utilizado seja custo uniforme, o cost_function será zero. Quando selecionado com heurística, poderá ser a heurística simples ou complexa.

Uma vez com o current_node criado, entramos em um laço para verificar se já temos a solução no board atual, se tivermos, retornamos as variáveis number_of_nodes, visited, possibilities, current_node, cost_function; caso contrário, colocamos o

current_node no *array* de visitados e setamos o numer_of_nodes chamando add_children(number_of_nodes, visited, possibilities, current_node, cost_function = None), o current_node entao recebe o nodo atual e assim que achar a solução, retorna o number_of_nodes, possibilities, visited, o current_node e o cost_function.

add_children(number_of_nodes, visited, possibilities, current_node, cost_function = None):
Uma vez chamada, setamos a variável none_position com a posicao do None dentro do *board* e setamos a variável choices com os movimentos possíveis para a none_position. Feito isso, entramos em um laço para cada um dos itens de choice, dentro desse laço, setamos o *board* chamando create_node_board(current_node, none_position, choice) que retorna o *board* com a posição do None atualizada. Depois, checamos se esse *board* já é conhecido chamando is_not_know_board(visited, possibilities, board) que faz dois laços e verifica se os nodos presentes em visited e possibilities são iguais ao *board* atual, se for retorna falso, caso contrário retorna verdadeiro. Caso is_not_know_board retorne falso, retornados o número de nodos e finalizamos a execução de add_children, caso seja verdadeiro, incrementamos a variável de number_of_nodes e criamos um novo nodo com a estrutura da resposta 2, mas criando um path chamando create_node_path(current_node) que adiciona a uma *deepcopy* do path do current_node, o próprio, depois adicionamos esse new_node ao *array* de possibilities e retornamos o number_of_nodes.

create_node_board(current_node, none_position, choice): criamos uma *deepcopy* com o *board* do current_node chamada *board*, criamos uma variável position que chama calcule_next_position(choice, none_position) que pega o valor de choice e a de current_position e calcula onde é a próxima posição com base no dicionário de NEXT_POSITION. Entao, removemos o None da variável *board* e colocamos o None na posição calculada anteriormente por calcule_next_position. Retornamos esse board.

calculate_simple_cost(board): quando selecionarmos o tipo 3 de algoritmo para solução, é criado uma lista enumerada de SOLUTION, quando comparado com a lista do *board* atual, é verificado se o *index* e o número no *board* estão nos locais

corretos, caso não esteja no local correto é calculado a heurística do custo e retornado o valor de cost atualizado.

calculate_complex_cost(board): quando selecionarmos o tipo != 1, 2 e 3 consideramos que queremos o algoritmo do tipo heurística complexa. Como neste caso possuímos um mapeamento para o custo em cada local de cada peça fora da solução, pegamos o board.index(number) no arquivo cost_map.py. Calculamos a nova heurística e retornamos o cost atualizado.

6. **Caso algum dos objetivos não tenha sido alcançado explique o que você faria VS o que foi feito e exatamente qual o(s) problema(s) encontrado(s), bem como limitações da implementação.**

Espera-se que o usuário faça uma entrada de um tabuleiro que pode ser solucionado, porém, se o usuário adicionar um tabuleiro que não há solução, será retornado um erro, o ideal seria ter uma checagem antes de iniciarmos o código.

Para resolver o custo de heurística do caso complexo criamos um mapeamento dos custos baseados em onde o elemento está e onde ele deveria estar. Sabemos que o ideal seria ter uma função um pouco mais elaborada que calculasse esse valor pois da forma como fizemos limitamos nossa solução apenas para esse Puzzle (tamanho de board).

7. **Referências**

WIKIPÉDIA. **City block.** Disponível em: <https://en.wikipedia.org/wiki/City_block#:~:text=Oblong%20blocks%20range%20considerably%20in,adopted%20by%20other%20US%20cities.>

Sistemas Inteligentes - Aula 1.1 - Busca: Profundidade e Amplitude, último acesso 22/05/2022
https://www.youtube.com/watch?v=LQ_47j5O1MY

Sistemas Inteligentes - Aula 2 - Busca: Custo Uniforme, último acesso 22/05/2022
https://www.youtube.com/watch?v=Ai0qiD_iuXk

Sistemas Inteligentes - Aula 3 - Busca Heurística, último acesso 22/05/2022

https://moodle.ufsc.br/mod/resource/view.php?id=3929325