Relatório do Laboratório 3: Otimização com Métodos de Busca Local

Isabelle Ferreira de Oliveira
CT-213 - Engenharia da Computação 2020
Instituto Tecnológico de Aeronáutica (ITA)
São José dos Campos, Brasil
isabelle.ferreira3000@gmail.com

***Resumo*—Esse relatório documenta a implementação dos seguintes algoritmos de otimização baseados em busca local: Descida do Gradiente, Hill Climbing e Simulated Annealing. Esses métodos foram testados em uma regressão linear para obter parâmetros físicos relativos ao movimento de uma bola. Por fim, os resultados das três implementações foram comparados.**

***Index Terms*—Algoritmos de Otimização, busca local, Descida de Gradiente, Hill Climbing, Simulated Annealing, regressão linear**

I. Introdução

Planejamento de ações de agentes podem ser implementados por diversos métodos atualmente. Construindo-se um grafo que represente diferentes opções de estados para o agente, por exemplo, um planejamento de ação pode ser definido aplicando um algoritmo de busca em grafo, partindo da situação inicial do agente até o estado desejado, onde cada aresta representa uma possível ação, atrelada a um custo para executá-la.

Seguindo esse contexto, para decidir qual melhor planejamento a se escolher (considerando que essa escolha é baseada em menor custo de se sair de uma situação de início para um objetivo), pode-se aplicar algoritmos de busca de caminho mínimo, como Dijkstra, Greedy (ou Guloso) e A*.

Os pseudo-códigos desses algoritmos podem ser vistos nas subseções a seguir. Em seguida, será apresentado como esses algoritmos foram implementados no contexto do laboratório.

A. Algoritmo de Dijkstra

```
dijkstra(start):
  pq = PriorityQueue()
  start.cost = 0
  pq.insert_or_update(start.cost, start)
  while not pq.empty():
    node = pq.extract_min()
    for successor in node.successors():
      if successor.cost > node.cost +
          cost(node, successor):
        successor.cost = node.cost +
            cost(node, successor)
        successor.parent = node
        pq.insert_or_update(successor.cost,
            node)
```

B. Algoritmo Greedy

```
greedy_search(start, goal):
  pq = PriorityQueue()
  start.cost = h(start, goal)
  pq.insert_or_update(start.cost, start)
  while not pq.empty():
    node = pq.extract_min()
    for successor in node.successors():
      successor.parent = node
      if successor.content == goal:
        return successor
      successor.cost = h(successor, goal)
      pq.insert_or_update(successor.cost,
          successor)
```

C. Algoritmo A*

```
a_star(start, goal):
  pq = PriorityQueue()
  start.g = 0
  start.f = h(start, goal)
  pq.insert_or_update(start.f, start)
  while not pq.empty():
    node = pq.extract_min()
    if node.content == goal:
      return successor
    for successor in node.successors():
      if successor.f > node.g + cost(node,
          successor) + h(successor, goal):
        successor.g = node.g + cost(node,
            successor)
        successor.f = successor.g +
            h(successor, goal)
        successor.parent = node
        pq.insert_or_update(successor.f,
            successor)
```

II. Implementação dos algoritmos

Na parte relativa a implementação dos algoritmos de busca, era necessário preencher os códigos das funções *dijkstra*, *greedy* e *a_star* da classe *PathPlanner* do código base fornecido [1].

A análise de vários pontos dos algoritmos descritos acima terão uma breve descrição em alto nível da sua implementação a seguir.

Como os algoritmos foram implementados em Python, as filas de prioridade foram importadas da biblioteca *heapq*, sendo utilizado as funções *heappush* e *heappop* para adicionar e retirar um elemento da fila, respectivamente.

Já as funções heurísticas dos algoritmos Greedy e A* foram calculadas a partir da função *distance_to*, que calcula a distância euclidiana entre um nó específico e o nó objetivo. Essa função já foi fornecida pelo código básico.

Na etapa de percorrer os sucessores de um determinado nó para se calcular os custos relativos aquele caminho analisado, utilizou-se a função *get_successors* para se obter esses sucessores. Para cada nó analisado, então, seu nó sucessor foi adicionado à fila de prioridade, sendo o custo dependente se o movimento se deu pela diagonal ou pela lateral do nó do *grid*. Dessa forma, passos laterais tiveram custos menores que passos diagonais, na proporção de $1{:}\sqrt{2}$. Uma representação de um nó específico do grafo e seus 8 vizinhos foi apresentado na Figura 1.

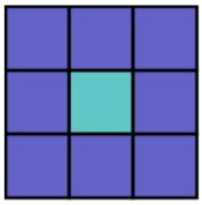

Figura 1. Representação de 8-conectado. Cada nó apresenta 8 vizinhos, sendo 4 laterais e 8 diagonais.

Quando um determinado nó era, por fim, retirado da fila de prioridade, o custo mínimo do início do trajeto até aquele ponto já estaria definido. Nesse instante, então, o atributo *closed* daquele nó foi setado como *True*. Ao nó com custo definido de caminho do início até ele for o nó objetivo, a busca encerrou-se. No algoritmo Greedy, excepcionalmente, o custo já estava definido desde a primeira vez que o nó em específico era visitado.

III. Resultados e Conclusões

Os resultados obtidos após 1 execução das implementações dos algoritmos descritos acima foram apresentados nas Figuras 2, 3 e 4 para Dijkstra, Greedy e A*, respectivamente.

Foi executado também 100 vezes cada um dos códigos, para se calcular as médias e as variâncias dos custos de tempo computacional e de caminho para os três algoritmos, e os resultados foram apresentados na Tabela I.

Tabela I
Comparação dos custos de tempo computacional e custos de caminho para os algoritmos do laboratório.

Algoritmo	Tempo Computacional (s)		Custo do caminho	
	Média	Desvio Padrão	Média	Desvio Padrão
Dijkstra	0.2796	0.1616	79.8291	38.5709
Greedy Search	0.03101	0.0864	175.9340	91.9683
A*	0.3112	0.2100	79.8291	38.5709

Isso era esperado, uma vez que, os custos de caminho dos algoritmos de Dijkstra e A* são equivalentes (ambos ótimos), e o do Greedy é significantemente maior. Além disso, os tempos computacionais são crescentes na ordem dos algoritmos Greedy, A* e Dijkstra.

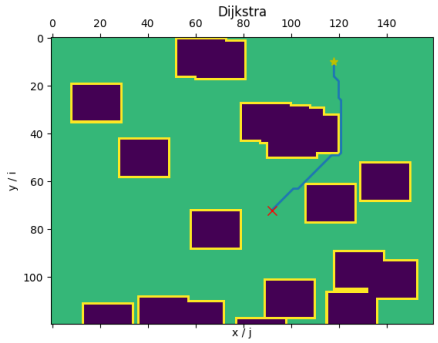

Figura 2. Execução do algoritmo de Dijkstra para busca do caminho mínimo no grafo em grid do laboratório.

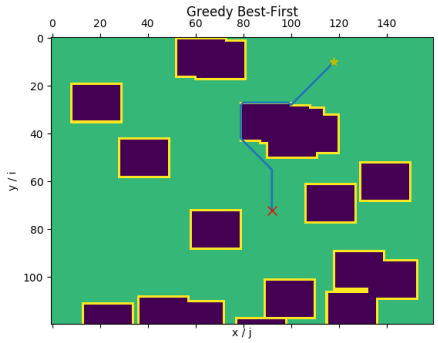

Figura 3. Execução do algoritmo Greedy para busca do caminho mínimo no grafo em grid do laboratório.

Referências

[1] M. Maximo, "Roteiro: Laboratório 2 - Busca Informada". Instituto Tecnológico de Aeronáutica, Departamento de Computação. CT-213, 2019.

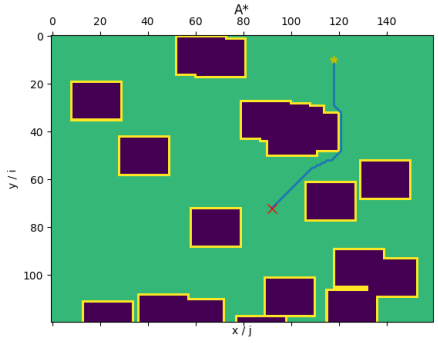

Figura 4. Execução do algoritmo A* para busca do caminho mínimo no grafo em grid do laboratório.