Busca de Rotas entre Cidades de Alagoas: Usando Dijkstra e A*

Luryan Delevati Dorneles

14 de abril de 2025

Resumo

Esse trabalho mostra como usei dois algoritmos de busca (Dijkstra e A*) pra achar o caminho mais curto entre cidades de Alagoas, usando as coordenadas delas. A ideia é que só existe "estrada"entre cidades se a distância em linha reta for menor que um valor 'r'. Comparei os dois algoritmos pra ver qual se sai melhor. Os resultados mostram como eles funcionam e como o 'r' influencia se a gente acha um caminho ou não.

1 Introdução

Achar o melhor caminho entre lugares é um problema clássico, importante pra logística, GPS e outras coisas. Aqui, a gente foca em achar rotas entre cidades de Alagoas usando as coordenadas (latitude e longitude) e dois algoritmos conhecidos: Dijkstra e A*.

O desafio é achar o caminho com a menor distância total entre duas cidades que o usuário escolher. Pra simplificar, a gente imagina que tem uma estrada direta entre duas cidades só se a distância em linha reta entre elas for menor ou igual a um raio 'r'. Isso transforma o mapa de cidades num grafo, onde as cidades são os nós e as "estradas"são as arestas com peso igual à distância.

2 Modelagem

Pra resolver isso como um problema de IA, a gente modelou assim:

2.1 Análise PEAS (Item 1)

Pensando num agente que resolve o problema:

- **Medida de Desempenho**: Achar a rota com a menor distância total.
- **Ambiente**: As 101 cidades de Alagoas com suas coordenadas. As conexões dependem do raio 'r'. A gente considera que sabe tudo sobre o ambiente e ele não muda durante a busca.
- **Atuadores**: A parte do algoritmo que decide qual a próxima cidade a explorar no caminho.

- **Sensores**: A capacidade de "ver"as coordenadas de todas as cidades e calcular a distância entre elas pra saber quem é vizinho de quem e qual o custo de ir de uma pra outra.

2.2 Espaço de Estados (Item 2)

Transformando o problema pra busca:

- **Estados**: Cada cidade é um estado.
- **Estado Inicial**: A cidade de onde a gente quer sair.
- **Estado(s) Objetivo**: A cidade aonde a gente quer chegar.
- **Ações**: Estando numa cidade C_i, as ações são ir pra qualquer cidade vizinha C_j (ou seja, $d(C_i, C_j) \leq r$).
- **Modelo de Transição**: Se estou em C_i e a ação é "ir pra C_j", o novo estado é C_j.
- **Custo da Ação**: O custo de ir de C_i pra C_j é a distância $d(C_i, C_j)$. O custo total do caminho é a soma das distâncias de cada passo.
- **Heurística (só pro A*)**: $h(n)$ é uma estimativa de quanto falta pra chegar no destino a partir da cidade n. Aqui, usei a distância em linha reta da cidade atual até o destino. Isso funciona bem porque nunca superestima a distância real (é admissível).

3 Implementação (Item 4)

Fiz o código em Python. Usei listas, dicionários e a biblioteca 'heapq' (que é boa pra filas de prioridade, essencial pros dois algoritmos).

3.1 Estruturas de Dados

Os dados das cidades de Alagoas foram organizados em um arquivo JSON, contendo informações como código do município, nome, latitude, longitude, distância em linha reta e distância por rodovia até Maceió, além do tempo estimado de viagem. Este arquivo foi gerado a partir de uma tabela original em formato texto, que foi processada e ordenada para facilitar o uso no projeto.

O arquivo JSON foi utilizado para construir o grafo de cidades, onde cada cidade é representada como um nó, e as conexões entre elas (arestas) são determinadas pela distância euclidiana entre suas coordenadas geográficas.

3.2 Estrutura do JSON

A estrutura do arquivo JSON é composta por uma lista de objetos, onde cada objeto representa uma cidade com os seguintes campos:

- **codigo**: Código do município (IBGE).
- **municipio**: Nome da cidade.

- **latitude**: Latitude da cidade.
- **longitude**: Longitude da cidade.
- **distanciaMaceio**: Distância em linha reta até Maceió.
- **distanciaRodoviaMaceio**: Distância por rodovia até Maceió.
- **tempoViagemMaceio**: Tempo estimado de viagem até Maceió.

3.3 Algoritmo de Dijkstra

É o algoritmo clássico que vai expandindo a busca a partir da origem, sempre olhando primeiro pros caminhos mais curtos já encontrados. Garante achar o melhor caminho se as distâncias não forem negativas. O Algoritmo 1 mostra a ideia geral.

Algorithm 1 Ideia do Dijkstra

```
1: procedure DijkstraSearch(start, end, cities, radius)
2:     Inicializar distances (∞, 0 p/ start) e previous (null)
3:     pq ← Fila Prioritária com (0, start)
4:     visited ← ∅
5:     while pq não vazia do
6:         (dist, current) ← pq.pegar_menor()
7:         if current já foi visited then continue
8:         end if
9:         if current = end then break                          ▷ Achou!
10:        end if
11:        Marcar current como visited
12:        neighbors ← AcharVizinhos(current, cities, radius)
13:        for cada neighbor em neighbors do
14:            if neighbor não foi visited then
15:                edge_dist ← Distancia(current, neighbor)
16:                new_dist ← distances[current] + edge_dist
17:                if new_dist < distances[neighbor] then      ▷ Achou caminho melhor
18:                    distances[neighbor] ← new_dist
19:                    previous[neighbor] ← current
20:                    pq.add((new_dist, neighbor))
21:                end if
22:            end if
23:        end for
24:    end while
25:    path ← MontarCaminho(previous, start, end)
26:    return path, distances[end]
27: end procedure
```

3.4 Algoritmo A*

Parecido com o Dijkstra, mas usa a heurística (distância em linha reta até o fim) pra tentar ir mais direto ao ponto. Ele prioriza caminhos que já são curtos E parecem promissores

pela heurística. A ideia tá no Algoritmo 2.

Análise da Solução:

- **Estados Inicial/Final:** Água Branca / Piaçabuçu.
- **Resultado:** Não achou caminho. Com r=0.10, o grafo fica "quebrado", e não tem como ir de uma cidade pra outra seguindo as conexões permitidas. Os algoritmos exploraram o que dava a partir de Água Branca, mas não chegaram em Piaçabuçu.

3.5 Comparação dos Algoritmos (Item 6)

Comparando Dijkstra e A*:

- **Geral:** Dijkstra explora tudo ao redor, A* tenta ser mais esperto usando a heurística pra ir na direção certa. Ambos acham o melhor caminho (se a heurística do A* for boa).
- **Implementação:** Não muda muito a dificuldade, A* só precisa da função heurística a mais.
- **Entendimento:** Dijkstra talvez seja mais fácil de sacar no começo. A* tem a "mágica"da heurística, mas a ideia é lógica.
- **Tempo:**** Pelos testes (coloque seus tempos aqui!), A* geralmente foi mais rápido. A heurística ajuda a não perder tempo explorando caminhos ruins. A diferença deve ser maior em rotas longas.
- **Memória:**/** Os dois precisam guardar informações sobre os nós. A* pode precisar de mais memória pra guardar os nós "promissores"na fila, mas como ele geralmente explora menos nós no total, o uso de memória pode acabar sendo parecido ou até menor. (Se conseguir medir nós visitados, ajuda a comparar).

(Complete essa parte com seus números e sua experiência.)

3.6 Escalabilidade e Limitações (Item 7)

Escalabilidade: A forma como fiz tem problema pra crescer:

- **Achar Vizinhos Lento:** Pra cada cidade, o código compara com TODAS as outras pra ver quem tá perto (complexidade O(N)). Fazer isso toda hora deixa tudo lento se tiver muitas cidades.
- **Muitas Cidades:** Com milhares de cidades, esse jeito de achar vizinhos na hora não rola.

Limitações: O modelo é bem simples e tem vários problemas:

- **Distância vs Realidade:** Distância em linha reta ignora estradas, rios, montanhas. Uma rota curta no mapa pode ser impossível ou longa na vida real.
- **Raio 'r' Fixo:** Conectar só quem tá perto com 'r' fixo é artificial. Tem estrada longa e às vezes não tem estrada entre cidades vizinhas.

Algorithm 2 Ideia do A*

```
1: function HEURISTIC(city, end)
2:     return Distancia(city, end)                          ▷ Distância em linha reta
3: end function
4: procedure ASTARSEARCH(start, end, cities, radius)
5:     Inicializar g_score (∞, 0 p/ start)
6:     Inicializar f_score (∞)
7:     f_score[start] ← Heuristic(start, end)
8:     open_set ← Fila Prioritária com (f_score[start], start)
9:     came_from ← ∅
10:    closed_set ← ∅
11:    while open_set não vazia do
12:        (f, current) ← open_set.pegar_menor()            ▷ Nó mais promissor
13:        if current = end then break                      ▷ Achou!
14:        end if
15:        if current ∈ closed_set then continue
16:        end if
17:        Adicionar current a closed_set
18:        neighbors ← AcharVizinhos(current, cities, radius)
19:        for cada neighbor em neighbors do
20:            if neighbor ∈ closed_set then continue
21:            end if
22:            edge_dist ← Distancia(current, neighbor)
23:            tentative_g_score ← g_score[current] + edge_dist
24:            if tentative_g_score < g_score[neighbor] then   ▷ Achou caminho melhor
25:                came_from[neighbor] ← current
26:                g_score[neighbor] ← tentative_g_score
27:                f_score[neighbor] ← tentative_g_score + Heuristic(neighbor, end)
28:                if neighbor não está em open_set then
29:                    open_set.add((f_score[neighbor], neighbor))
30:                else
31:                    open_set.atualizar_prioridade((f_score[neighbor], neighbor))
32:                end if
33:            end if
34:        end for
35:    end while
36:    path ← MontarCaminho(came_from, start, end)
37:    return path, g_score[end]
38: end procedure
```

- **Só Distância Importa?:** O custo é só a distância. Tempo de viagem, pedágio, tipo de estrada, nada disso entra na conta.
- **Desempate por População:**** A atividade pedia pra desempatar usando população (preferir cidade menos populosa). O arquivo 'cities-al.json' que usei não tem população, então não implementei isso. Pra fazer, teria que ter o dado de população e mudar a fila de prioridade pra considerar '(custo, populacao, cidade)'. (Confirme se isso se aplica ao seu caso).

Como Melhorar: Pra deixar mais realista ou desafiador:

- **Mapa de Estradas Real:** Usar dados tipo OpenStreetMap, que já tem as estradas e distâncias/tempos reais.
- **Mais Custos:** Considerar tempo, dinheiro, etc.
- **Trânsito e Afins:** Incluir coisas que mudam, como tráfego.
- **Algoritmos Melhores pra Mapas Grandes:**
  - **Pré-calcular:** Técnicas como Contraction Hierarchies (CH) preparam o mapa pra responder rápido depois.
  - **Busca mais esperta:** Heurísticas melhores (ALT) ou buscar dos dois lados ao mesmo tempo (Bidirecional).
  - **Achar Vizinhos Rápido (se usar raio):** Usar k-d trees ou Quadtrees pra achar vizinhos por raio mais rápido (tipo O(log N)), mas ainda assim o modelo de raio é limitado.

4 Conclusão

Consegui implementar Dijkstra e A* pra achar rotas em Alagoas usando o modelo simplificado de raio e distância. Modelei o problema (PEAS, estados) e testei em cenários diferentes. Deu pra ver que os dois funcionam (quando tem caminho) e que A* costuma ser mais rápido por causa da heurística.

Discuti as limitações do modelo (linha reta, raio fixo) e o problema de escalabilidade (achar vizinhos é lento). Pra um sistema real, teria que usar dados de estradas de verdade e talvez algoritmos mais avançados. Mas valeu pra entender e comparar Dijkstra e A* num problema geográfico.

5 Referências

Referências

[1] Russell, S. J., & Norvig, P. (2010). *Artificial Intelligence: A Modern Approach* (3rd ed.). Prentice Hall.

[2] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.

[3] Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1), 269–271.

[4] Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2), 100–107.

[5] Python Software Foundation. (2023). *Python Language Reference, version 3.x*. Recuperado de https://docs.python.org/3/reference/