INSTITUTO FEDERAL DO ESPÍRITO SANTO
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO APLICADA – PPCOMP

GUSTAVO AMORA BASÍLIO

PROBLEMAS NP-COMPLETOS EM CLOJURE

Serra
2023

GUSTAVO AMORA BASÍLIO

PROBLEMAS NP-COMPLETOS EM CLOJURE

Trabalho apresentado ao Programa de Pós-Graduação em Computação Aplicada – PPCOMP do Instituto Federal do Espírito Santo, como requisito para aprovação da Disciplina de Teoria da Computação ministrada pelo Prof. Dr. Jefferson O. Andrade.

Serra
2023

LISTA DE TABELAS

SUMÁRIO

1 SELEÇÃO DO PROBLEMA

1.1 TSP

O problema escolhido para o trabalho foi o problema do caixeiro viajante, do inglês *Traveling Salesman Problem (TSP)*.

O problema do TSP é um problema de roteiro de viagem em que cada cidade deve ser visitada uma única vez e o objetivo é encontrar o caminho mais curto. Foi provado em 1972 que o problema é do tipo NP-difícil, e desde então diversos esforços foram empregados para desenvolver uma aproximação heurística efetiva (RUSSELL; NORVIG, 2009).

1.2 Implementação Inicial

Antes de iniciar a implementação dos algoritmos exato e heurístico, que veremos nas próximas seções, é necessário definiar algumas funções base para o problema. As implementações iniciais foram as seguintes:

I) Criação de uma lista chamada cidades, onde cada cidade é um mapa com um nome e suas coordenadas.

```
(def cidades
  [{:nome "A" :coordenadas [0 0]}
   ...
   ])
```

II) Função que calcula a distância euclidiana entre duas coordenadas e é usada para calcular a distância entre duas cidades.

```
(defn distancia-euclidiana [coord1 coord2]
.. )
```

III) Função que recebe um nome de cidade e retorna a cidade correspondente da lista 'cidades'.

```
(defn encontrar-cidade [nome]
.. )
```

IV) Função que utiliza 'encontrar-cidade' para obter as coordenadas de duas cidades e depois calcula a distância entre elas usando a função 'distancia-euclidiana'.

```
(defn distancia-entre-cidades [cidade1 cidade2]
.. )
```

V) Calcula o custo total de um caminho, somando as distâncias entre cada par de cidades consecutivas no caminho.

```
(defn calcular-custo [caminho]
.. )
```

2 ALGORITMO EXATO

2.1 Força Bruta

O algoritmo escolhido para encontarar o resultado exato foi o algoritmo 'força bruta'. O algoritmo força bruta não é um padrão para todos os problemas, mas em comum eles executam todas as possibilidades do problema proposto e retornam o melhor (ou pior) resultado.

2.2 Implementação

Para o problema do TSP, os seguintes passos foram feitos na implementação do algoritmo de força bruta:

I) O algoritmo começa gerando todas as permutações possíveis das cidades, excluindo a cidade inicial, e cada permutação representa um possível caminho.
II) A cidade inicial é adicionada ao início de cada permutação para formar um caminho completo que começa e termina na mesma cidade.
III) Para cada caminho gerado, o algoritmo calcula o custo total. O custo é a soma das distâncias entre cada par de cidades consecutivas no caminho.
IV) Após calcular o custo de todos os caminhos possíveis, o algoritmo seleciona o caminho com o menor custo total. Este caminho é considerado a solução ótima do problema.
V) O algoritmo registra o tempo de execução e o número total de iterações (permutações geradas).

```
(defn tsp-forca-bruta [cidade-inicial nomes-cidades]
  (let [cidades-sem-inicial (filter #(not= % cidade-inicial) nomes-cidades)
        inicio (System/currentTimeMillis)
        permutacoes (map #(cons cidade-inicial %) (permutations cidades-sem-inicial
        caminhos-custos (map #(vector % (calcular-custo %)) permutacoes)
        melhor-caminho (apply min-key second caminhos-custos)
        fim (System/currentTimeMillis)]
    {:caminho (first melhor-caminho)
     :custo (second melhor-caminho)
     :tempo (str (- fim inicio) " ms")
     :iteracoes (count permutacoes)}))
```

O algoritmo 'força bruta' garante a solução ótima, mas é computacionalmente intensivo e não escalável para um grande número de cidades (RUSSELL; NORVIG, 2009).

3 ALGORITMO HEURÍSTICO

3.1 Heurística Gulosa

A heurística escolhida para encontarar o resultado exato foi o algoritmo 'guloso'. O algoritmo gulosa tenta expandir para o caminho que está mais próximo de seu local atual, com o fundamento de que isso pode conduzir a uma solução rapidamente. Assim, ela avalia as cidades do problema apenas em função da distãncia mais próxima. (RUSSELL; NORVIG, 2009)

3.2 Implementação

Os seguintes passos foram feitos na implementação do algoritmo guloso:

I) A partir da cidade inicial, o algoritmo entra em um loop onde, a cada iteração, escolhe a próxima cidade a ser visitada.
II) Em cada iteração, o algoritmo escolhe a cidade mais próxima que ainda não foi visitada. Esta escolha é baseada na distância mais curta da cidade atual a uma cidade não visitada.
III) Após escolher a próxima cidade, o algoritmo atualiza o caminho incluindo esta cidade e remove a cidade da lista de cidades não visitadas.
IV) Este processo se repete até que todas as cidades tenham sido visitadas e retorna à cidade inicial.
V) O algoritmo também registra o tempo de execução e o número de iterações, que neste caso é igual ao número de cidades.

```
(defn tsp-guloso [cidade-inicial nomes-cidades]
  (let [inicio (System/currentTimeMillis)
        [caminho custo] (loop [atual cidade-inicial
                               nao-visitadas (disj (set nomes-cidades) cidade-inici
                               caminho [cidade-inicial]
                               contador 0]
                          (if (empty? nao-visitadas)
                            [caminho (calcular-custo caminho)]
                            (let [proxima (apply min-key (fn [c] (distancia-entre-
                              (recur proxima (disj nao-visitadas proxima) (conj ca
        fim (System/currentTimeMillis)]
    {:caminho caminho
     :custo custo
     :tempo (str (- fim inicio) " ms")
     :iteracoes (count caminho)}))
```

O algoritmo 'guloso' é eficiente em termos de tempo de execução, mas pode não encontrar a solução ótima. É mais adequado para um número maior de cidades onde a solução ótima não é prática de ser calculada.

4 COMPLEXIDADE

4.1 Complexidade: Força Bruta

O algoritmo examina todas as permutações possíveis de cidades para encontrar o caminho mais curto. Assim, se houver n cidades há $(n-1)!$ permutações a considerar (já que a cidade inicial é fixa). Para cada permutação, o algoritmo calcula a distância total do percurso, o que envolve somar $n1$ distâncias. Portanto, a complexidade de tempo é $O((n-1)!(n-1))$, que é aproximadamente $O(n!)$ (SIPSER, 2006).

Essa complexidade de tempo é fatorial e, portanto, o algoritmo de força bruta torna-se impraticável mesmo para um número moderadamente grande de cidades.

4.2 Complexidade: Guloso

Começando de uma cidade inicial, o algoritmo seleciona a cidade mais próxima ainda não visitada em cada passo. Há $n1$ escolhas a serem feitas (uma para cada cidade, excluindo a cidade inicial). Em cada escolha, o algoritmo compara as distâncias a todas as cidades restantes não visitadas. Na primeira escolha, compara $n1$ cidades, na segunda $n2$, e assim por diante, até 1. Assim, o número total de comparações é $(n1) + (n2) + ... + 1$, que é a soma dos primeiros $n1$ números inteiros, resultando em $n(n1)/2$. Portanto, a complexidade de tempo do algoritmo guloso é $O(n^2)$ (SIPSER, 2006).

5 INTERFACE

5.1 Implementação

Para criar a interface com o usuário foi criada a função obter-entrada-usuario. Ela é responsável por ler a entrada do usuário, pedindo para digitar o número de cidades (limitado de 7 a 12) e o método de resolução (força bruta ou guloso). As escolhas do usuário são então usadas para determinar o subconjunto de cidades e o algoritmo a ser utilizado. O código pressupõe que o usuário fornecerá entradas válidas e no formato esperado.

```
(defn obter-entrada-usuario []
  (println "Digite o número de cidades (entre 7 e 12):")
  (let [num-cidades (Integer/parseInt (read-line))]
    (println "Escolha o algoritmo ('forca-bruta' ou 'guloso'):")
    (let [algoritmo (read-line)]
      [num-cidades algoritmo])))
```

6 AVALIAÇÃO E EXPERIMENTAÇÃO

6.1 Testes

Os códigos teste foram executados em uma máquina com as especificações conforme a Tabela 1, sem aplicações em segundo plano e com acesso a internet.

Tabela 1 – Configurações da máquina

Configuração
SO Ubuntu 22.04
CPU AMD Ryzen 5 5600G, 3901 Mhz, 6 núcleos, 12 Processadores lógicos
GPU AMD Vega 7, 2GB
RAM 16 GB
ARMAZENAMENTO SSD 512 GB

Para avaliar e experimentar os algoritmos o código foi rodado 12 vezes, sendo agrupados de 1 a 6. Em cada grupo o número de cidades é o mesmo e o algoritmo de resolução é variado a fim de comparação dos resultados.

Os resultados de cada algoritmo com relação ao caminho final podem ser observados na Tabela 2.

Tabela 2 – Resultado do caminho final por teste

Teste	Algoritmo	Caminho
1	Força Bruta	A E C G D F B
1	Guloso	A E C G D F B
2	Força Bruta	A E C G D H F B
2	Guloso	A E C G D H F B
3	Força Bruta	A E C G D H F I B
3	Guloso	A E C G D H F I B
4	Força Bruta	A E C G D H F I J B
4	Guloso	A E C G D H F I J B
5	Força Bruta	A E C G D H F I J K B
5	Guloso	A E C G D H F I J K B
6	Força Bruta	OutOfMemoryError
6	Força Bruta	A E C G D H F I J L K B

Surpreendentemente os caminhos do algoritmo guloso foram iguais aos da solução exata, exceto para o teste 6, onde ocorreu um erro de memória e o algoritmo exato força bruta não conseguiu resolvê-lo. Provavelmente esta solução exata do algoritmo guloso se deu pelo baixo número de cidades do problema, bem como as distâncias das coordenadas disponibilizadas e do baixo número de testes. Este comportamento não era esperado e não é garantia de que este algoritmo encontrará a melhor solução para os demais testes.

Também foi monitorado do tempo de execução de cada algoritmo e o número de iterações de cada um deles. Os resultos podem ser verificados na Tabela 3.

Tabela 3 – Resultado do tempo de execução e custo por teste

Teste	Algoritmo	N. Cidades	Tempo (ms)	Custo	Iterações
1	Força Bruta	7	30	11.89	720
1	Guloso	7	1	11.89	7
2	Força Bruta	8	155	12.06	5040
2	Guloso	8	1	12.06	8
3	Força Bruta	9	1100	14.57	40320
3	Guloso	9	1	14.57	9
4	Força Bruta	10	11143	15.47	362880
4	Guloso	10	1	15.47	10
5	Força Bruta	11	119822	15.45	3628800
5	Guloso	11	2	15.45	11
6	Força Bruta	12	OutOfMemoryError	OutOfMemoryError	
6	Força Bruta	12	2	15.47	12

Os resultados de tempo de execução dos algoritmos e do número de iterações foram conforme o esperado. Detalhe para a diferença de velocidade de execução do algoritmo guloso para a força bruta conforme o número de cidades cresce. Isso indica claramente que para problemas de maior ordem é preciso a utilização de heurísticas para o problema do TSP ser solucinado em tempo 'razoavelmente' rápido.

REFERÊNCIAS

RUSSELL, Stuart J.; NORVIG, Peter. *Artificial Intelligence: a modern approach*. 3. ed. [S.l.]: Pearson, 2009.

SIPSER, Michael. *Introduction to the Theory of Computation*. Second. [S.l.]: Course Technology, 2006. ISBN 7111173279 9787111173274.