UNIVERSIDADE FEDERAL DO ESPIRITO SANTO

LUCAS MARTINUZZO BATISTA
PEDRO REISEN ZANOTTI

PROBLEMA DO CAIXEIRO VIAJANTE

Vitória

2015

LUCAS MARTINUZZO BATISTA
PEDRO REISEN ZANOTTI

PROBLEMA DO CAIXEIRO VIAJANTE

Trabalho de Estrutura de Dados II, realizado sob a orientação da professora Mariella Berger Andrade.

Vitória

2015

1 INTRODUÇÃO

O Problema do Caixeiro Viajante (PVC), ou, em inglês, Traveling Salesman Problem (TSP), é um problema clássico de matemática computacional e otimização combinatória que foi discutido pela primeira vez pelo matemático Sir Willian Rowan Hamilton por volta do ano 1800 e que ganhou notoriedade mundial em 1950. O problema se inspira na necessidade de um vendedor em realizar entregas em uma série de cidades, de forma que ele percorra o menor caminho possível, reduzindo o tempo de viagem e os possíveis custos com transporte e combustível. Durante esse percurso, cada cidade deve ser visitada uma única vez. O TSP é dito simétrico quando a distância de uma cidade qualquer para outra é igual à distância desta para a primeira; e assimétrico, ou ATSP, caso contrário.

Embora o conceito do TSP seja extremamente simples, uma análise mais cuidadosa leva à conclusão de que a quantidade de rotas possíveis de viagem, considerando um grupo de n cidades, é R(n) = (n-1)!, um valor que cresce absurdamente rápido conforme o número de cidades aumenta. A partir dessa informação, do ponto de vista computacional, é possível inferir que o tempo necessário para o cálculo da melhor rota aumenta proporcionalmente ao número de rotas possíveis. Portanto, o custo em tempo de execução de um programa elaborado resolvê-lo, a partir de um determinado número de cidades, torna-se tão grande a ponto do cálculo se tornar impraticável. Por exemplo, para 25 cidades, seriam gastos 470 milhões de anos.

Os métodos heurísticos surgiram como alternativas à resolução exata do TSP e consistem em encontrar uma solução viável e razoável para o problema, de formas relativamente intuitivas, sem que se garanta que ela é a melhor ou uma das melhores soluções possíveis. Neste trabalho, serão abordadas, além do método exato de resolução, as heurísticas do vizinho mais próximo, do envoltório convexo e a heurística de melhoramento 2-opt, as quais serão detalhadas mais à frente. Esses algoritmos foram implementados na linguagem C, com o objetivo de compreender-se a complexidade do problema e comparar-se a qualidade e a viabilidade de cada uma das soluções.

O programa foi implementado com o auxílio do IDE NetBeans e possui um arquivo-fonte principal, main.c, que utiliza o código dos módulos secundários, exato.c, nn2opt.c e convexHull.c, a partir da importação de seus respectivos cabeçalhos (.h). Por questões de reaproveitamento de código, optou-se pela importação do cabeçalho do arquivo exato.c nos módulo nn2opt.c e pela importação do cabeçalho deste em convexHull.c. A estrutura do programa foi desenvolvida de modo que ele possa ser testado através do comando ./trab1 n algoritmo, em que n é o número de cidades do

TSP e algoritmo refere-se ao nome do algoritmo que deseja-se utilizar, conforme a orientação fornecida. A função principal do arquivo main.c utiliza as demais funções contidas nele para executar o algoritmo selecionado, chamando-as de acordo com o mesmo.

2 IMPLEMENTAÇÃO

2.1 Solução ótima (ATSP)

A solução “ótima” consiste em obter a rota a ser percorrida que possui o menor custo possível num TSP. Nesta implementação, em **criarMatrizDistancias**, foi alocada uma matriz de números inteiros correspondentes às distâncias entre as cidades envolvidas no problema, de forma que um elemento da mesma na posição [i][j] corresponde a à distância da cidade i à cidade j. Essa matriz simula a matriz que o programa lê de um arquivo no momento em que é executado.

A principal função deste módulo chama-se **metodoExato** e baseia-se na ideia de gerar recursivamente todas as permutações possíveis de um conjunto de cidades representadas por números armazenados no vetor percursoAtual. Esse vetor é inicializado com os números das cidades em ordem crescente graças à função **criarPercursoInicial**, assim como o vetor menorPercurso, o qual receberá a resposta do problema. O argumento menorDistancia, inicializado com o custo do percurso inicial, da mesma forma, receberá a distância total da trajetória contida em menorPercurso. As verificações necessárias para identificar o percurso menos custoso estão incluídas em **compararMenorPercurso**, função que é invocada através da função metodoExato. Esta também utiliza uma função básica auxiliar, **trocar**, que realiza a troca dos valores de duas variáveis inteiras, as quais, neste caso, representam duas cidades distintas do vetor percursoAtual. A função **calcularDistancia**, cujo nome é bastante intuitivo, calcula o custo total de um percurso por meio do acesso da matriz de distâncias entre as cidades. Finalmente, **imprimirResultado** recebe as variáveis de interesse menorDistancia e menorPercurso, e imprime os seus respectivos conteúdos na tela de acordo com a especificação do enunciado do trabalho.

É interessante esclarecer que grande parte dos parâmetros utilizados foram passados por referência, visto que seus conteúdos deveriam ser modificados nas diversas funções citadas acima, ao longo do programa. Convencionou-se, por questões de lógica e coesão, que o custo de uma sequência que possui uma única cidade é zero, o que é válido para todos os algoritmos implementados, assim como o fato da cidade inicial (cidade zero) ser fixada como o início e o fim de qualquer percurso. A estrutura condicional na função main estabelece um limite para o número de cidades para o qual o TSP deve ser resolvido. A partir de quinze cidades, o programa leva mais de uma hora para computar uma solução.

2.2 Heurística do vizinho mais próximo (ATSP)

Este método de solução baseia-se na ideia de sempre visitar-se a cidade mais próxima daquela onde se está, desconsiderando as cidades que já foram visitadas e até que se retorne à cidade inicial. Trata-se de uma heurística um tanto primitiva, a qual provavelmente não apresentará um bom resultado. Nesta implementação, as funções **criarMatrizDistancias**, **calcularDistancia**, **imprimirResultado** e **trocar** são análogas às suas funções correspondentes na implementação do método se solução exato. Desta vez, as respostas finais são armazenadas nas variáveis distancia e percurso, e, inicialmente, todas as posições deste vetor são inicializadas com o valor negativo -1.

O grande diferencial da implementação deste algoritmo em relação à implementação do algoritmo anterior está na função **vizinhoMaisProximo**, a qual percorre a matriz de distâncias entre as cidades por meio de dois iteradores com o objetivo de identificar a cidade mais próxima da cidade em questão e, à medida que conclui esse processo, atualiza o vetor percurso, construindo a trajetória final. Essa função chama a função auxiliar **cidadeVisitada**, a qual é utilizada apenas para determinar se a cidade em análise, representada por cidadeAtual, já está contida no vetor percurso, ou seja, se ela já foi visitada.

2.3 Heurística de melhoramento 2-opt (ATSP)

O algoritmo de melhoramento 2-opt utiliza como base a solução do TSP proveniente da heurística do vizinho mais próximo. Ele consiste em eliminar duas conexões não-adjacentes entre as cidades e, em seguida, reestabelecer essas conexões de uma maneira distinta, utilizando outras duas conexões. Esse processo é realizado para todos os pares de conexões e, em cada caso, verifica-se a ocorrência de melhoria no custo da solução.

Na função **opt2**, a não adjacência sugerida anteriormente é garantida pelas inicializações e condições de parada estabelecidas dentro das estruturas for. O parâmetro percurso, vetor resultante da função do algoritmo do vizinho mais próximo, tem as suas cidades manipuladas conforme a ideia citada no parágrafo anterior. Após uma dessas manipulações, realizada através da função de troca já utilizada em implementações anteriores, verifica-se o custo da nova trajetória. Caso este seja menor, ele e seu respectivo percurso serão armazenados em distancia e menorPercurso. Caso não ocorram melhoras, as alterações em percurso são desfeitas. A variável distanciaOriginal funciona apenas como uma forma de acessar o custo original do vetor percurso, de modo que ela seja utilizada na verificação necessária para atualizá-lo, ao final da execução.

A função **inverter** não seria necessária caso a implementação sugerida seguisse o modelo da heurística de melhoramento 2-opt apresentado em sala de aula, em que as conexões entre as cidades são bidirecionais. Porém, como esta é uma implementação de um ATSP, em que as conexões são unidirecionais, no momento em que estas são quebradas e reestabelecidas de uma maneira distinta, o caminho entre as cidades que trocaram de posição fica na ordem contrária à ordem na qual deveria estar segundo a heurística. Portanto, realiza-se uma inversão do mesmo, de forma que ele seja corrigido.

2.4 Heurística do envoltório convexo (TSP)

Esta implementação difere-se das demais no que diz respeito à categoria do TSP com a qual se está lidando. Neste caso, o TSP é simétrico, ou seja, a distância de uma cidade qualquer à outra é idêntica à distância desta à primeira. Os arquivos lidos por este módulo também possuem uma formatação distinta dos anteriores: eles contêm coordenadas cartesianas de cada uma das cidades, as quais serão armazenadas em um vetor gerado pela função criarVetorCoordenadas. Para facilitar esse processo, definiu-se o tipo estruturado **Coordenadas**, o qual contém um campo para o número correspondente à cidade em questão, um campo para a sua coordenada x e um campo para a sua coordenada y. A partir deste ponto, as posteriores citações do termo "cidade(s)" estarão se referindo ao tipo Coordenadas.

Desta vez, a distância entre duas as cidades não está explícita, mas pode ser facilmente calculada utilizando a fórmula da distância euclidiana entre pontos coordenados que as representam, encargo da função **distanciaEuclidiana**. A função **calcularDistanciaEuc** é apenas uma versão adaptada das funções das outras implementações com nomes similares, ou seja, trata-se da função que calcula o custo total de um percurso.

A ideia geral deste algoritmo é gerar um envoltório convexo que engloba todo o grupo de cidades sob análise e, então, incluir cada uma das cidades não contidas no tour desse envoltório no mesmo, de forma que cada inserção acarrete o menor custo possível. Para gerar o envoltório, utiliza-se a função **gerarEnvoltorio**, a qual percorre o vetor cidades e seleciona a sequência das cidades adequadas para formá-lo e as armazena no parâmetro envoltorio. Essa mesma função ainda atualiza o valor do parâmetro tamEnvoltorio para a quantidade de cidades que compõem o envoltório. Para fazê-lo, ela utiliza a função auxiliar **examinarRotacao**, a qual calcula, dadas três cidades de coordenadas (x1, y1), (x2, y2) e (x3, y3), o produto vetorial dos dois vetores definidos pelos pontos (x1, y1), (x2, y2) e (x2, y2), (x3, y3). Caso o resultado seja zero, as três cidades constituem uma única reta; caso seja positivo, as cidades estão localizadas de modo que formam uma "curva para a esquerda"; e caso contrário, as

cidades formam uma “curva para a direita”. O algoritmo implementado para gerar o envoltório convexo é conhecido como Algoritmo de Andrew, uma versão adaptada do Exame de Graham em que o vetor cidades é previamente ordenado segundo a coordenada x de cada cidade, o que é garantido pela função **ordenacao**.

A função **definirPercurso** é a responsável pela execução da segunda parte da heurística do envoltório convexo. Ela percorre o vetor cidades e insere cada uma das cidades que não se encontram no vetor envoltorio no mesmo, a partir de uma posição calculada obedecendo o critério de minimização de custo. A função auxiliar que identifica se uma cidade encontra-se no vetor do envoltório ou não é a **verificarCidade**. As funções que calculam a posição exata do vetor das cidades do envoltório convexo onde uma cidade qualquer ainda não contida nele deve ser inserida são **definirPosicaoCidade** e sua auxiliar, **definirPosicaoAuxiliar**. A primeira computa, a princípio, qual é a cidade-vértice que forma o envoltório mais próxima da cidade a ser inserida, entretanto essa informação não basta para que a inserção possa ser realizada pela função **inserirCidade**. Nesta etapa, há duas possibilidades possíveis: inserir essa cidade antes da cidade-vértice encontrada ou depois dela. A função definirPosicaoAuxiliar é a delegada para decidir a opção mais viável, ou seja, a menos custosa. Por fim, a função **reordenar** recebe o percurso final completo proveniente da função definirPercurso e o reordena de forma que a cidade zero seja a cidade inicial do vetor, pois, como já foi dito, ela deveria ser fixada e todo tour deveria ser realizado partindo-se dela.

3 ANÁLISE

Os algoritmos implementados foram testados com base nos arquivos fornecidos pela TSLIB, a qual também foi utilizada como referência para a análise da qualidade dos resultados obtidos, visto que o site contém as melhores soluções possíveis já encontradas para cada grupo de cidades. Conforme esperado, o algoritmo da solução "ótima" encontra com exatidão a melhor solução possível para o problema, porém esse processo torna-se cada vez mais lento conforme o número de cidades aumenta. Para 14 cidades, o programa leva cerca de 26 minutos e 28 segundos para computar a solução exata. Aumentado esse número em apenas uma única cidade, esse tempo eleva-se para mais de uma hora de execução, o que certamente é um grande inconveniente. Isso ocorre devido ao fato de que o número de permutações das cidades a serem analisadas aumenta segundo uma expressão fatorial, como foi descrito na introdução. Para que se tenha uma noção da proporção desse aumento, é interessante notar que o programa levou 0,005 segundos para calcular uma solução para uma série de 6 cidades; e 9,095 segundos para uma série de 12.

Comparando os tempos de execução das heurísticas vizinho mais próximo e 2-opt, para números de cidades abaixo das 14 citadas acima, observam-se valores muito próximos, oscilando e torno de 0,004 e 0,005 segundos. O algoritmo 2-opt obteve uma melhora significativa no custo da trajetória calculada pelo algoritmo do vizinho mais próximo, de até 66, 37%. Entretanto, considerando um número mais elevado de cidades, na ordem de centenas de cidades, esse valor é reduzido até 0,6%, o pior caso analisado (desconsiderando os casos em que não houve melhora). Para essas mesmas centenas de cidades, também observa-se que o tempo do algoritmo 2-opt é ligeiramente superior ao algoritmo da heurística do vizinho mais próximo. Por exemplo, para 443 cidades, a execução deste leva cerca de 0,167 segundos para ser finalizada, enquanto a execução do 2-opt gasta 1,015 segundos. Este é um resultado óbvio, afinal a heurística 2-opt é de melhoramento, ou seja, ela precisa de que outro algoritmo seja utilizado previamente para que possa aprimorar o seu resultado. Neste caso, o algoritmo previamente executado é o do vizinho mais próximo. O algoritmo do envoltório convexo mostrou desempenho superior, em relação ao tempo de execução, aos demais algoritmos, para esse número de cidades.

Para milhares de cidades, o algoritmo do envoltório convexo, que havia demonstrado eficiência para um número reduzido de cidades, passa a levar um tempo relativamente alto para que compute a solução do TSP. Para 13509 cidades, ele leva 11, 911 segundos para fazê-lo. Embora esse tempo seja elevado, ele ainda é pequeno

em comparação com o tempo que os algoritmos vizinho mais próximo e 2-opt gastam durante o mesmo processo, o qual pode levar a horas de execução. A partir do confronto das soluções encontradas pelo programa implementado com as melhores soluções já encontradas, é possível dizer que essas foram aceitáveis, apesar de serem piores. À medida que o número de cidades aumenta, a distância entre essas soluções computadas também aumenta.

4 CONCLUSÃO

A partir da implementação dos algoritmos de resolução do Problema do Caixeiro Viajante e da análise realizada anteriormente, foi possível comparar e verificar a complexidade do mesmo e compreender algumas formas de solucioná-lo, as quais podem apresentar um bom desempenho ou não dependendo da situação em que são aplicadas. Observou-se, com base nos dados coletados, que resolver o TSP de modo exato é um processo extremamente custoso para um computador, e que esse processo pode levar desde segundos até milhares de anos em tempo de execução.

Explorando as heurísticas tratadas neste trabalho, pode-se concluir um pouco a respeito de cada uma delas. O algoritmo do vizinho mais próximo, de modo geral, é pouco eficiente e pode gerar percursos que, apesar de serem certamente válidos, talvez não poderiam ser utilizados na prática, pois obviamente não estão entre as melhores opções. O algoritmo de melhoramento 2-opt realmente funciona e obtém uma solução igual ou melhor ao resultado obtido pelo algoritmo do vizinho mais próximo. Apesar disso, essa melhoria não chega a ser tão significativa quando o número de cidades envolvidas é alto. A respeito do algoritmo do envoltório convexo, é possível inferir que, apesar de basear-se numa ideia mais elaborada, ele apresenta resultados razoavelmente satisfatórios. Seguramente esta implementação seria mais rápida caso ela fosse realizada utilizando-se listas encadeadas, e não vetores, como foi feito. Dessa forma, não haveria necessidade de deslocar o conteúdo armazenado no vetor quando um novo item fosse inserido. Apesar desse fato ter sido notado, diversos problemas surgiram durante a implementação das listas, então optou-se por permanecer utilizando vetores.

A principal dificuldade encontrada durante a implementação do programa foi compreender o funcionamento em código do algoritmo 2-opt. Neste programa, ele deveria ser utilizado na resolução de um ATSP, em que as conexões entre as cidades não são bidirecionais como na sua representação visual apresentada nas aulas. Entretanto, após algumas pesquisas, essa dúvida foi esclarecida. Algumas das instruções passadas no enunciado do trabalho, apesar de estarem apresentadas de forma clara e objetiva, também não foram inicialmente compreendidas pela dupla, devido à inexperiência da mesma em utilizar o sistema operacional Linux e o LaTeX.

5 REFERÊNCIAS

TSP - WIkipedia, the free encyclopedia. Disponível em: <https://en.wikipedia.org/wiki/TSP>. Acesso em: 2 de Setembro de 2015.

TSP History Home. Disponível em: <http://www.math.uwaterloo.ca/tsp/history/>. Acesso em: 2 de Setembro de 2015.

Write a C program to print all permutations of a given string - GeeksforGeeks. Disponível em: <http://www.geeksforgeeks.org/write-a-c-program-to-print-all-permutations-of-a-given-string/>. Acesso em: 21 de Agosto de 2015.

Segundo Trabalho Prático de PAA: O Problema do Caixeiro Viajante 1. Disponível em: <http://homepages.dcc.ufmg.br/ nivio/cursos/pa03/tp2/tp22/tp22.html>. Acesso em: 24 de Agosto de 2015.

BURTSCHER, Martin. **A High-Speed 2-Opt TSP Solver for Large Problem Sizes.** Disponível em: <http://on-demand.gputechconf.com/gtc/2014/presentations/S4534-high-speed-2-opt-tsp-solver.pdf>. Acesso em: 24 de Agosto de 2015.

Andrew's Monotone Chain Algorithm. Disponível em: <http://www.codecodex.com/wiki/Andrew%27s_Monotone_Chain_Algorithm>. Acesso em: 29 de Agosto de 2015.

Convex Hull |Set 1 (Jarvis's Algorithm or Wrapping) - Geeks for Geeks. Disponível em: <http://www.geeksforgeeks.org/convex-hull-set-1-jarviss-algorithm-or-wrapping/>. Acesso em: 29 de Agosto de 2015.