UNIVERSIDADE FEDERAL DE OURO PRETO - UFOP
INSTITUTO DE CIÊNCIAS EXATAS E APLICADAS - ICEA

The Force Awakens

Projeto e Análise de Algoritmos - CSI546

GRUPO 01:
ALVARO BRAZ CUNHA - 21.1.8163
BERNARDO LUCAS DE ARAÚJO DIAS - 20.1.8011
BRENO ARTHUR ROTTE FERNANDES OLIVEIRA - 20.1.8124
DIEGO SANCHES NERE DOS SANTOS - 21.1.8003
JHONATAN FIGUEIREDO ALMEIDA - 20.1.8164
MELISSA MUNIZ MIRANDA DE SOUZA - 19.2.8112

PROFESSOR:
BRUNO CESAR COTA CONCEICAO

João Monlevade, MG
2024

1 Introdução

Dado a temática de análise de algoritmos e três paradigmas de programação: Força-Bruta, Algoritmos Gulosos e Programação Dinâmica, além do contexto de “Star Wars", este trabalho visa, por meio da implementação dos algoritmos citados, determinar, em uma rota específica, quais k pontos serão percorridos a fim de minimizar as sub-distâncias percorridas entre todos os n pontos. Em outras palavras, quais k “planetas"serão conquistados de forma a minimizar as sub-distâncias percorridas entre todos os n planetas, o início e o fim do caminho.

Além disso, não é possível repetir trechos em um rota, ou seja, o objetivo é ir de um ponto inicial I até um ponto final F passando uma única vez por cada trecho entre os planetas.

2 Paradigmas de Programação

Nesta seção, discorre-se sobre os paradigmas de programação utilizados no desenvolvimento dos algoritmos deste trabalho prático, assim como foi solicitado na descrição do mesmo.

2.1 Força-Bruta

Um algoritmo de Força-Bruta é um tipo de algoritmo que busca enumerar todas as possíveis soluções de um problema e verificar se cada uma dessas soluções satisfaz o problema. Normalmente, essa classe de algoritmo possui uma implementação simples e sempre encontrará uma solução se ela existir, ainda que em alguns casos seja extremamente ineficiente.

O objetivo do algoritmo desenvolvido é minimizar a sub-distância de maior custo na solução final. Dessa forma, ele seleciona todas as possíveis combinações de k planetas e calcula a sub-distância entre cada par em cada combinação, comparando a sub-distância encontrada com a maior sub-distância até o momento. Ao final, a sub-distância de maior custo é escolhida como solução final, garantindo que a solução encontrada minimize a sub-distância de maior custo.

Abaixo, segue o algoritmo de Força-Bruta implementado.

function FB($*distance, n, k$)
    $greaterSubdistance \leftarrow \infty$
    $currentDistance \leftarrow 0$
    $subdistance \leftarrow$ vetor de inteiros com tamanho $n + 1$
    $subdistance[0] \leftarrow 0$
    repeat
        if $subdistance[currentDistance] < n$ **then**
            $subdistance[currentDistance+1] \leftarrow subdistance[currentDistance] + 1$
            $currentDistance \leftarrow currentDistance + 1$
        else

```
            subdistance[currentDistance − 1] ← subdistance[currentDistance − 1] + 1
            currentDistance ← currentDistance − 1
        end if
        if currentDistance = k then
            subdistanceTemp ← vetor de inteiros com tamanho currentDistance + 1
            intermediateSubdistancia ← 0
            iter ← 0
            while iter < currentDistance + 1 do
                if iter ≠ currentDistance then
                    subdistanceTemp[iter] ← CALCULATESTHECOSTOFTHESUBDISTANCE(subdistance[iter], subdistance[iter + 1], distance)
                else
                    subdistanceTemp[iter] ← CALCULATESTHECOSTOFTHESUBDISTANCE(subdistance[iter], n + 1, distance)
                end if
                if subdistanceTemp[iter] > intermediateSubdistancia then
                    intermediateSubdistancia ← subdistanceTemp[iter]
                end if
                iter ← iter + 1
            end while
            if greaterSubdistance > intermediateSubdistancia then
                greaterSubdistance ← intermediateSubdistancia
            end if
            FREE(subdistanceTemp)
        end if
    until currentDistance = 0
    FREE(subdistance)
    return greaterSubdistance
end function
```

2.2 Algoritmo Guloso

Um Algoritmo Guloso é um tipo de algoritmo que, a cada passo, escolhe a melhor opção localmente para tentar alcançar a melhor solução global para um problema. Ele faz escolhas que parecem ser as melhores naquele momento, sem se preocupar com possíveis consequências futuras. Além disso, são voltados para resolução de problemas de otimização.

Para a problemática em questão, desenvolvemos o Algoritmo Guloso abaixo. A ideia deste algoritmo é escolher um planeta inicial que maximize a distância para o próximo planeta escolhido, e em seguida escolher o planeta que maximize a distância para o último planeta escolhido. Desta forma, para cada elemento da sequência, escolher o próximo elemento que maximize a soma entre o elemento atual e o próximo elemento.

```
function AG(dist, n, k)
    max_subdistance ← 0
    current_subdistance ← 0
    first ← 0
    last ← n − 1
    trav_distance ← array com n elementos inicializados com 0
    for i ← 0 to n − 1 do
        if dist[i] > dist[first] then
            first ← i
        end if
    end for
    trav_distance[first] ← 1
    for i ← 1 to k − 1 do
        best ← −1
        for j ← 0 to n − 1 do
            if not trav_distance[j] and (best = −1 or dist[j] − dist[last] > dist[best] − dist[last]) then
                best ← j
            end if
        end for
        trav_distance[best] ← 1
        last ← best
    end for
    for i ← 0 to n − 1 do
        if trav_distance[i] then
            current_subdistance ← current_subdistance + dist[i]
            if current_subdistance > max_subdistance then
                max_subdistance ← current_subdistance
            end if
            current_subdistance ← 0
        else
            current_subdistance ← current_subdistance + dist[i]
        end if
    end for
    return max_subdistance
end function
```

2.3 Programação Dinâmica

A Programação Dinâmica é uma técnica algorítmica utilizada para resolver problemas de otimização combinatória. Ademais, é aplicável a problemas em que é possível encontrar a solução ótima combinando soluções ótimas de subproblemas que foram armazenados com o fim de reutilizá-los, evitando o recálculo a cada iteração, reduzindo drasticamente o tempo de execução do algoritmo.

Dessa forma, no algoritmo de Programação Dinâmica desenvolvido, os subproblemas podem ser definidos sendo o conjunto de planetas visitado e a quantidade de planetas restantes para serem visitados. Mais especificamente, o valor da solução ótima para o subproblema que envolve visitar os planetas 1,2,...,i com j planetas restantes é armazenado na matriz memo[i][j]. Dessa forma, o algoritmo utiliza a propriedade de subestrutura ótima para resolver problemas menores (subproblemas) e construir soluções para problemas maiores (problema original).

Abaixo, segue o algoritmo de Programação Dinâmica implementado.

```
function PD(dist[], n, k)
    memo[0...n][0...k] ← −1
    for i ← 0 to n do
        for j ← 0 to k do
            memo[i][j] ← −1
        end for
    end for
    memo[0][0] ← 0
    for i ← 1 to n do
        memo[i][0] ← calculatesTheCostOfTheSubdistance(0, i, dist)
    end for
    trav_distance[0...n − 1] ← 0
    last ← n − 1
    first ← 0
    for i ← 1 to k − 1 do
        best ← −1
        for j ← 0 to n − 1 do
            if (trav_distance[j] == 0) and (best == −1 or dist[j] − dist[last] >
dist[best] − dist[last]) then
                best ← j
            end if
        end for
        trav_distance[best] ← 1
        last ← best
        if i == 1 then
            first ← best
        end if
    end for
    for j ← 1 to k do
        for i ← j to n do
            if j == 1 then
                memo[i][j] ← calculatesTheCostOfTheSubdistance(first, i, dist)
            else
                memo[i][j] ← INT_MAX
                for x ← j − 2 to i − 1 do
                    cost ← calculatesTheCostOfTheSubdistance(x, i, dist)
                    memo[i][j] ← menor(memo[i][j], maior(memo[x][j − 1], cost))
                end for
            end if
```

```
            end for
        end for
        return memo[n][k]
    end function
```

3 Análise de Complexidade

Nesta seção, são abordadas as análises de complexidade dos algoritmos implementados. Foram analisadas as complexidades de tempo dos algoritmos e as complexidades de espaço das principais estruturas de dados utilizadas.

3.1 Complexidade de Tempo

Em relação à complexidade de tempo, para o algoritmo de Força-Bruta, tem-se $O(n^k)$, onde n é o número total de planetas e k é o número de planetas a serem conquistados. Isso porque o algoritmo de Força-Bruta percorre todas as combinações possíveis de subconjuntos de k planetas dentre os n possíveis.

Para o Algoritmo Guloso, tem-se a complexidade de $O(n^2)$, onde n é o número total de planetas. Isso por conta dos dois loops aninhados, ambos com complexidade de $O(n)$ cada.

Para o algoritmo de Programação Dinâmica, tem-se a complexidade de $O(n^2 * k)$, pelo fato de existir três loops aninhados. Isso ocorre porque, para cada j de 1 até k e para cada i de j até n, o algoritmo faz uma nova iteração de k-1 até i-1. Além disso, a função $calculatesTheCostOfTheSubdistance()$ é chamada dentro do loop interno, cuja complexidade de tempo é $O(n)$.

3.2 Complexidade de Espaço

Em relação à complexidade de espaço, para o algoritmo de Força-Bruta, tem-se $O(n + k)$, pois ele utiliza dois vetores alocados dinamicamente, *subdistance* e *subdistanceTemp*, de tamanhos $n + 1$ e $k + 1$, respectivamente.

Para o Algoritmo Guloso, tem-se a complexidade de $O(n)$, pois ele aloca apenas um vetor de tamanho n para armazenar as distâncias percorridas entre os planetas.

Para o algoritmo de Programação Dinâmica, tem-se a complexidade de $O(n*k)$, isso porque ele utiliza uma matriz de memoização de tamanho (n+1)x(k+1) para armazenar os valores calculados durante a execução do algoritmo. Além disso, ele aloca o vetor *trav_distance* de tamanho n.

4 Avaliação Experimental

Nesta seção, são explicitados os resultados dos experimentos que foram realizados com os algoritmos implementados. Dessa forma, foi avaliado o tempo de execução dos algoritmos em função da entrada passada como parâmetro para cada um.

Sendo assim, foi utilizado o valor de 10 para a quantidade k de planetas a serem conquistados e os valores de 25, 35, 45 e 55 para a quantidade n de planetas totais. Além disso, para criar o vetor de sub-distâncias, foi utilizada a função $rand()$ da biblioteca *time.h*. O tempo limite definido para os teste foi de 4 minutos (240.000 ms).

Abaixo, segue a tabela com os resultados obtidos. Note que foi utilizada a unidade de medida milissegundos para o tempo.

n	k	FB	AG	PD
25	10	760,753 ms	0,056 ms	0,111 ms
35	10	162489,277 ms	0,05 ms	1,171 ms
45	10	Extrapolou	0,06 ms	1,090 ms
55	10	Extrapolou	0,092 ms	2,897 ms

Desse modo, ao analisar os resultados, percebe-se que o algoritmo de Força-Bruta, ainda que seja o mais confiável, é o que demanda muito mais tempo. Por outro lado, o Algoritmo Guloso, se mostrou o mais eficiente em relação ao tempo de execução.

5 Conclusão

Portanto, ao comparar as saídas esperadas dos testes disponibilizados pelo professor com os nossos resultados, conclui-se que a implementação dos três paradigmas de programação com o intuito de solucionar o problema foi realizada de forma correta.

Além disso, no processo de desenvolvimento dos algoritmos, enfrentamos alguns problemas específicos para cada paradigma de programação. Por exemplo, para o Algoritmo Guloso, estávamos com dificuldade em decidir qual escolha gulosa utilizar, visto que várias não se encaixavam na problemática. Porém, depois de alguns testes e analisarmos mais profundamente, chegamos em uma escolha gulosa que resolveu nosso problema.