Fundamentos de Projeto e Análise de Algoritmos
Trabalho Prático: Caixeiro Viajante

Alunos: Gabriel Lima de Souza, Nikolas Louret, Gabriel de Souza

Caixeiro Viajante

Para a primeira parte do trabalho, foram executados os mesmo 1000 grafos aleatórios com 12 vértices, para cada algoritmo estudado. O primeiro algoritmo analisado é o Força Bruta:

```
Número N-1: 12

Tempo total das iterações FB: 1497563ms
Tempo médio das iterações FB: 1497.563ms
```

Para essa abordagem, o tempo total das iterações foi de 1497563ms, com uma média de 1497.563ms por iteração, totalizando 24.96 minutos. Esses valores indicam que o algoritmo de Força Bruta é extremamente lento para resolver o problema do Caixeiro Viajante nesse contexto.

O próximo algoritmo analisado é o Guloso:

```
Tempo total das iterações G: 5ms
Tempo médio das iterações G: 0.005ms
```

Embora seja uma abordagem mais rápida do que a Força Bruta, o algoritmo não garante a solução ótima. Entretanto, os resultados mostram que o tempo total das iterações foi de apenas 5ms, com uma média de 0.005ms por iteração. Essa eficiência é bastante notável e indica que o algoritmo Guloso é uma boa escolha para o problema do Caixeiro Viajante nesse conjunto de grafos aleatórios.

O algoritmo a seguir é o Backtracking:

```
Tempo total das iterações Backtracking: 308ms
Tempo médio das iterações Backtracking: 0.308ms
```

O algoritmo de Backtracking é uma abordagem sistemática que explora todas as possíveis soluções, mas realiza podas quando identifica que a solução atual não pode levar à solução final. Embora mais eficiente do que a Força Bruta, ainda pode ser lento em problemas complexos. No caso dos 1000 grafos aleatórios com 12 vértices cada, o tempo total das iterações foi de 308ms, com uma média de 0.308ms por iteração. Essa melhoria em relação à Força Bruta é significativa, mas o algoritmo de Backtracking ainda não é tão rápido quanto o Guloso.

Por fim, os resultados obtidos para a Programação Dinâmica foram:

```
Tempo total das iterações PD: 1119ms
Tempo médio das iterações PD: 1.119ms
```

A Programação Dinâmica é uma técnica que resolve problemas dividindo-os em subproblemas menores e armazenando as soluções desses subproblemas para evitar cálculos repetidos. É uma abordagem eficiente para muitos problemas de otimização, incluindo o Caixeiro Viajante. Neste caso, o tempo total das iterações foi de 1119ms, com uma média de 1.119ms por iteração. Embora seja mais rápido que a Força Bruta, o tempo necessário ainda é considerável para o tamanho dos grafos aleatórios utilizados.

```
Quantidade de soluções iguais obtidas para FB e Guloso: 28
Quantidade de soluções iguais obtidas para BackTracking e PD: 276
```

A quantidade de soluções iguais encontradas pelos algoritmos de Força Bruta e Guloso foi de 28 casos. Isso significa que, em 28 dos 1000 grafos aleatórios com 12 vértices cada, ambos os algoritmos chegaram à mesma solução. Essa coincidência pode ser atribuída a características específicas desses grafos que permitiram que ambos os algoritmos chegassem ao mesmo resultado. No entanto, é importante notar que a taxa de convergência foi baixa devido às características de decisão do algoritmo Guloso. Por sempre escolher o caminho com o menor peso, ele muitas vezes não atinge a solução ótima alcançada pelo algoritmo de Força Bruta.

Por outro lado, houve 276 casos em que os algoritmos de Backtracking e Programação Dinâmica encontraram a mesma solução entre os 1000 grafos aleatórios. Isso sugere que a Programação Dinâmica tem uma tendência maior de convergir para a solução ótima encontrada pelo Backtracking em uma ampla variedade de grafos aleatórios com 12 vértices.

É importante destacar que a Programação Dinâmica nem sempre alcança a solução ótima devido à falta de uma subestrutura ótima no problema do Caixeiro Viajante. Enquanto a Programação Dinâmica assume que o problema pode ser dividido em subproblemas menores e independentes, cujas soluções podem ser combinadas para obter a solução global ótima, o problema do Caixeiro Viajante não apresenta essa propriedade. Isso significa que não é possível simplesmente combinar soluções ótimas de subproblemas menores para obter uma solução global ótima.

Além disso, a Programação Dinâmica, por ser uma abordagem exata, visa encontrar a solução ótima para um problema. No entanto, em muitos casos práticos, encontrar a solução ótima pode ser computacionalmente inviável, devido à complexidade do problema.

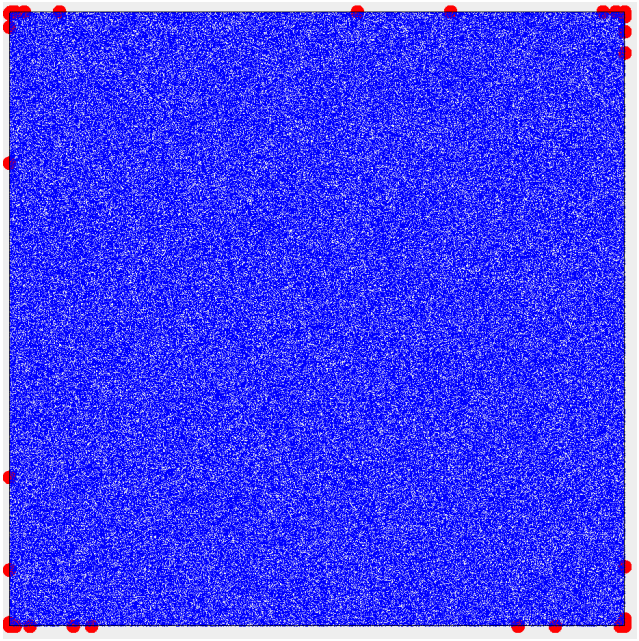
Em resumo, no conjunto de 1000 grafos aleatórios com 12 vértices cada, o algoritmo Guloso se destacou pela eficiência, sendo consideravelmente mais rápido do que os demais. Embora não garanta a solução ótima, ele fornece resultados aproximados com um tempo de execução notavelmente baixo. Os algoritmos de Backtracking e Programação Dinâmica também apresentaram melhorias em relação à Força Bruta, mas ainda requerem tempos consideráveis para resolver o problema do Caixeiro Viajante nesse conjunto específico de grafos aleatórios. A quantidade de soluções iguais encontradas entre os algoritmos sugere que certos grafos aleatórios podem ser resolvidos de forma eficiente pelos algoritmos analisados.

Envoltória Convexa

Para solução do problema da Envoltória Convexa, o algoritmo a seguir é uma implementação do algoritmo de divisão e conquista com e sem o uso de thread:

```
Tempo medio das iterações sem thread: 255.96
Tempo medio das iterações com thread: 143.52
```

Desenho do plano convexo utilizando 1 milhão de pontos:



A divisão e conquista é uma abordagem que consiste em dividir um problema sucessivamente em vários problemas menores. Após as divisões, as soluções são combinadas, o que gera como resultado final um algoritmo mais eficiente.

Na busca para encontrar o n-1 houve um problema devido a memória, ou seja, antes do algoritmo chegar em 10 segundos de iteração ele apresentava uma

exceção pelo consumo excessivo desse recurso, o que é um dos pontos mais delicados quando se trata da divisão e conquista.

A solução foi encontrada realizando o método da envoltória convexa em uma lista que possuía 1000000 de pontos com coordenadas aleatórias, solução essa que o método utilizando thread foi aproximadamente 1.7 vezes mais rápida que a sem thread.

O resultado obtido indica a natureza das aplicações envolvendo threads que consiste em fazer 2 ou mais partes do problema simultaneamente, otimizando o tempo da interação. O resultado, em um caso ótimo, tende a ser 2x mais rápido que o mesmo método sem o uso de thread, mas pelo tempo de instanciação e união das threads ele não alcança tal resultado.