TRABALHO PRÁTICO 3:

Alocação de Tarefas entre Agentes Robóticos usando Técnicas de Coloração de Grafos

Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)

viniciusoliveira@dcc.ufmg.br

Resumo: *Esse relatório descreve um problema de coloração de grafos solucionado por diferentes abordagens. As abordagens são relacionadas a três maneiras de fornecer soluções para um problema NP-Completo: força bruta (sequencial e paralela) e heurística. Para cada implementação, foi analisado se esta poderia fornecer uma solução ótima e os resultados obtidos foram comparados.*

1. Introdução

Em diversos contextos, é comum encontrarmos situações nas quais queiramos otimizar o uso de recursos disponíveis para realizar uma determinada tarefa, de forma que não exista trabalho repetido ou sub-aproveitamento dos recursos. Muitas vezes, esse tipo de situação pode ser resolvida através da modelagem do problema como um grafo a ser colorido de forma que vértices adjacentes não possuam a mesma cor.

Neste trabalho, o objetivo é utilizar o problema de coloração mínima de grafos para representar a uma situação de alocação de tarefas entre agentes robóticos. O desafio é dado um número máximo de agentes e um conjunto de pontos a serem visitados por eles, determinar quais pontos devem ser visitados por cada agente. Essa alocação deve ser feita de forma que todos os pontos sejam visitados e que um agente não tenha que cobrir um ponto que esteja fora da sua área circular de atuação. A modelagem do problema com grafos é feita da seguinte forma:

Os pontos são representadas por um vértice, cada agente disponível é representado por uma cor e dois vértices u e v são adjacentes se, e somente se, a distância entre u e v é maior que o raio da área de atuação dos agentes.

Assim, se colorirmos o grafo de forma que dois vértices adjacentes tenham necessariamente cores diferentes, e utilizarmos o menor número de cores possível, teremos a solução ótima do problema.

Considerando essa cenário, foi desenvolvido um trabalho com o objetivo de determinar qual é a coloração com o menor número de cores possível de um grafo especificado. Dada a complexidade exponencial do problema, a solução é feita de três formas: Testar todas as possibilidades sequencialmente, testar todas as possibilidades utilizando múltiplas threads e finalmente, uma heurística, que resolve o problema com complexidade polinomial, mas não garante solução ótima.

Para a elaboração desse trabalho, foram implementados tipos abstratos de dados (TAD's) para representar os vértices de um grafo, o grafo propriamente dito, além de estruturas de dados auxiliares. Espera-se com isso, praticar técnicas de solução de problemas classificados como NP-completos.

2. Referências Relacionadas

Pode-se dividir as referências associadas ao problema estudado e às soluções propostas dentre os seguintes tópicos:

• **Algoritmos de Força Bruta:** Algoritmos de força bruta, também conhecidos como buscas exaustivas consistem em enumerar todas as possíveis soluções de um problema e avaliá-las, uma a uma, em busca de uma solução satisfatória. Em geral, possuem implementação muito simples e sempre encontram a solução ótima se ela existir. No entanto, o custo computacional costuma ser bastante elevado, pois é proporcional ao tamanho do espaço de soluções do problema.

• **Algoritmos Paralelos:** Algoritmos paralelos são desenvolvidos de forma a aproveitar o fato de um computador possuir mais de um núcleo de processamento. Em geral, a estratégia utilizada é dividir o algoritmo sequencial de forma que uma amostra do espaço de solução não interfira nas demais e, assim, utilizar os diversos núcleos do processador para buscar a solução ótima ao mesmo tempo.

• **Heurísticas:** Podemos dizer que heurísticas são aproximações que muitas vezes são utilizados em tentativas de fornecer soluções para problemas cujo algoritmo de solução exata é exponencial. As heurísticas em geral, não fornecem nenhum tipo de garantia, podendo gerar soluções arbitrariamente ruins dependendo da entrada fornecida. A vantagem de se utilizar heurísticas está no fato de que as mesmas possuem em geral complexidade polinomial, o que permite que a solução seja gerada em um tempo viável.

3. Soluções Propostas

Nessa seção, as três abordagens propostas para o problema serão detalhadas de forma a explicar de maneira clara os algoritmos desenvolvidos:

3.1 Algoritmo de Força Bruta

Devido à característica dos algoritmos de força bruta de percorrer todo o espaço de soluções de um problema, eles sempre encontram a solução ótima caso ela exista. Portanto, uma maneira simples de apresentar uma solução ótima para o problema proposto é testar todas as possibilidades de coloração armazenando aquela que apresenta o menor número de cores.
A solução por força bruta desenvolvida para este trabalho testa cada uma das $(\text{numero\_de\_cores})^{\text{numero\_de\_pontos}}$ combinações possíveis (complexidade exponencial) e apresenta aquela cujo número de cores utilizado é o mínimo possível. Para se fazer isso, utilizamos como auxiliar, uma função que converte um número decimal para um número em uma base numérica qualquer. A utilidade dessa função é detalhada abaixo:
Suponha que o grafo a ser colorido tenha 4 vértices e dispomos de n=3 cores para colori-lo. Existem portanto 3^4 combinações possíveis. Se enunciarmos apenas as primeiras, já é possível perceber a utilidade uma função de conversão de base 10 para base n:

Índice	Combinações			
	Cor vértice 1	Cor vértice 2	Cor vértice 3	Cor vértice 4
0	0	0	0	0
1	0	0	0	1
2	0	0	0	2
3	0	0	1	0
4	0	0	1	1
5	0	0	1	2
. . .	. . .	. . .	. . .	. . .

Podemos notar que cada combinação é igual ao seu respectivo índice convertido em base n=3 (base ternária), com numero_de_vértices dígitos. Por exemplo, 410 = 00113.
Assumindo a existência de uma função (referenciada como 'converter') capaz de fazer essa conversão, com o número correto de dígitos, podemos escrever o seguinte algoritmo:

```
Algoritmo ForçaBruta{

   Inteiro indiceSolucao=0;
   Inteiro numeroCores=Infinito;
   Vetor de Inteiros solucaoGerada;
   Vetor de Inteiros melhorSolucao;

   Enquanto (indiceSolucao < numero_de_cores ^ numero_de_vertices) faça{
      solucaoGerada = converter(indiceSolucao);

      Se solucaoGerada é uma coloração válida{
         Se número de cores únicas de solucaoGerada < numeroCores{
            numeroCores = número de cores únicas de solucaoGerada;
            melhorSolucao = solucaoGerada;
          }
      }
      incremente indiceSolucao;
   }
   Retorne melhorSolucao;

Fim_Algoritmo
```

Com relação a complexidade espacial, podemos dizer que o algoritmo tem complexidade quadrática quando o grafo é completo, uma vez que cada vértice terá todos os outros em sua lista de adjacência, que é representada no programa como uma lista encadeada de inteiros.

3.2 Algoritmo Paralelo

O algoritmo paralelo desenvolvido tem basicamente a mesma ideia (e complexidade) do algoritmo força bruta, com a diferença de que ao invés de testar todas as possibilidades sequencialmente, o algoritmo divide as possibilidades a serem testadas entre as diferentes threads executadas, de forma que vários testes possam ser feitos ao mesmo tempo. O objetivo do algoritmo paralelo neste caso é simplesmente fazer com que o algoritmo Força Bruta execute mais rapidamente.
É importante salientar que o algoritmo foi desenvolvido de forma a exigir o mínimo de sincronização possível entre as threads, visando sacrificar o mínimo possível de memória. Um pseudocódigo do algoritmo paralelo (executado por cada thread) é apresentado abaixo:

```
Algoritmo Paralelo{

   Inteiro indiceSolucao=inicioEspacoThread;

   Inteiro minimoCoresLocal=Infinito;
   Vetor de Inteiros solucaoGerada;
   Vetor de Inteiros melhorSolucaoLocal;

   Enquanto (indiceSolucao < fimEspacoThread) faça{
      solucaoGerada = converter(indiceSolucao);

      Se solucaoGerada é uma coloração válida{
         Se número de cores únicas de solucaoGerada < minimoCoresLocal{
            minimoCoresLocal = número de cores únicas de solucaoGerada;
            melhorSolucaoLocal = solucaoGerada;
          }
      }
      incremente indiceSolucao;
   }


   Inicio Secao Critica (Exclusoa Mutua){
     Se (numeroCoresLocal < numero de cores da solucao global){
       numero de cores da solucao Global = minimoCoresLocal;
       solucao global = melhorSolucaoLocal; //escreve em memoria compartilhada
     }
   Fim da Secao Critica


Fim_Algoritmo
```

A complexidade espacial do algoritmo paralelo é a mesma do algoritmo força bruta, já que os dois são bastante similares, alterando apenas a ordem de verificação das combinações geradas.

3.3 Heurística:

Diferentemente dos dois algoritmos já apresentados, a heurística desenvolvida na implementação deste trabalho não garante que a solução cujo número de cores é mínima seja encontrada. A heurística sempre encontra uma coloração válida, mas não fornece garantia alguma sobre o número de cores utilizadas. A única restrição que faz com que a heurística determine que não é possível colorir o grafo é que o número de cores da solução encontrada seja maior do que o número de cores disponíveis.
Um pseudocódigo que descreve o funcionamento da heurística está apresentado abaixo:

```
Heurística{

   Inteiro i=0;
   Inteiro j=0;
   Inteiro corAtual=0;

   Faça com que todos os vértices do Grafo estejam em branco;

   Enquanto houver vértices em branco faça{
      Se o i-ésimo vértice estiver em branco & não houver vizinhos pintados de corAtual{
        Colora o i-ésimo vértice de corAtual;
      }
      incremente i;
      Se i for igual ao numero total de vértices no grafo faça{
        Incremente corAtual;
        Incremente j;
        Atribua j a i;
      }
   }

Fim_Heurística
```

Analisando o código apresentado, vemos que sua complexidade temporal é quadrática no pior caso (grafo completo), uma vez que devem ser percorridos todos os vértices em branco (que inicialmente são todos) e para cada um, sua lista de adjacência deve ser percorrida com o objetivo de verificar se algum vértice vizinho já possui a cor com a qual desejamos colorir o vértice atual.
Assim como nos demais algoritmos, a complexidade espacial da heurística também é quadrática, já que nesse caso, também é necessário que se armazene n listas de adjacência contendo n vértices, onde n é o número total de vértices do grafo no pior caso.

4. Implementação

4.1 Código

O código foi implementado de maneira modularizada, e desta forma, temos os seguintes arquivos:

4.1.1. Arquivos .c

IOHandler.c: Implementação das funções de entrada e saída de dados.
AdjList.c: Implementação do funcionamento de uma lista encadeada de inteiros.
Graph.c: Implementação do funcionamento de um grafo através de listas de Adjacências.
PaintUtils.c: Implementação das técnicas de coloração já apresentadas, juntamente com funções auxiliares importantes para o correto funcionamento dos algoritmos principais.
TP3.c: Implementação da função main do programa, que basicamente chama as demais funções implementadas.

4.1.2. Arquivos .h

IOHandler.h: Declaração das funções de escrita e leitura dados.
AdjList.h: Declaração das funções enunciadas no tipo abstrato de dados lista.
Graph.h: Declaração das funções enunciadas no tipo abstrato de dados Grafo.
PaintUtils.h: Declaração das funções enunciadas no tipo abstrato de dados PaintUtils, tais como chamadas ao algoritmo força bruta, paralelo e heurística.
ThreadArg.h: Esse arquivo se faz necessário pois define um tipo de dados que servirá como parâmetro para a execução das threads que serão executadas no algoritmo paralelo. É necessário encapsular todos os dados que as threads precisam usar pois a biblioteca pthreads define que apenas um parâmetro pode ser passado para as funções que as threads executam.
Spot.h: Cabeçalho que define o que representará um ponto no contexto do programa, tais como seu ID, e suas coordenadas cartesianas.

4.2 Compilação:

Para se compilar o programa, basta usar o comando *make* do Unix quando o terminal está executando no diretório que contém o arquivo *makefile* incluso no .zip do trabalho.

4.3 Execução:

Para executar o programa, basta utilizar o comando abaixo, onde *algoritmo* pode ser 1, 2 ou 3 (Força Bruta, Paralelo e Heurística, respectivamente).

```
./TP3 <algoritmo> <arquivo_de_entrada.txt>
```

4.3.1 Formato de entrada

A entrada do programa vem de um arquivo que deve estar formatado da seguinte maneira:

<numero_de_agentes_disponiveis_na_1ª_instancia> <raio_do_sensor_na_1ª_instancia>
<ID_do_1º_ponto_na_1ª_instancia> <coord_x_do_1º_ponto > <coord_y_do_1º_ponto >
<ID_do_2º_ponto_na_1ª_instancia> <coord_x_do_1º_ponto > <coord_y_do_1º_ponto >.
.
.
.
<ID_do_i-ésimo_ponto_na_1ª_instancia> <coord_x_do_ i-ésimo _ponto> <coord_y_do_ i-ésimo _ponto>
<linha_em_branco_para_separar_as_instancias>
<numero_de_agentes_disponiveis_na_2ª_instancia> <raio_do_sensor_na_2ª_instancia>
<ID_do_1º_ponto_na_2ª_instancia> <coord_x_do_1º_ponto > <coord_y_do_1º_ponto >
<ID_do_2º_ponto_na_2ª_instancia> <coord_x_do_1º_ponto > <coord_y_do_1º_ponto >.
.
.
.
<ID_do_i-ésimo_ponto_na_2ª_instancia> <coord_x_do_ i-ésimo _ponto> <coord_y_do_ i-ésimo _ponto>
<linha_em_branco_para_separar_as_instancias>
.
.
.

4.3.2 Formato de saída

A saída do programa é escrita no arquivo de output da seguinte forma:

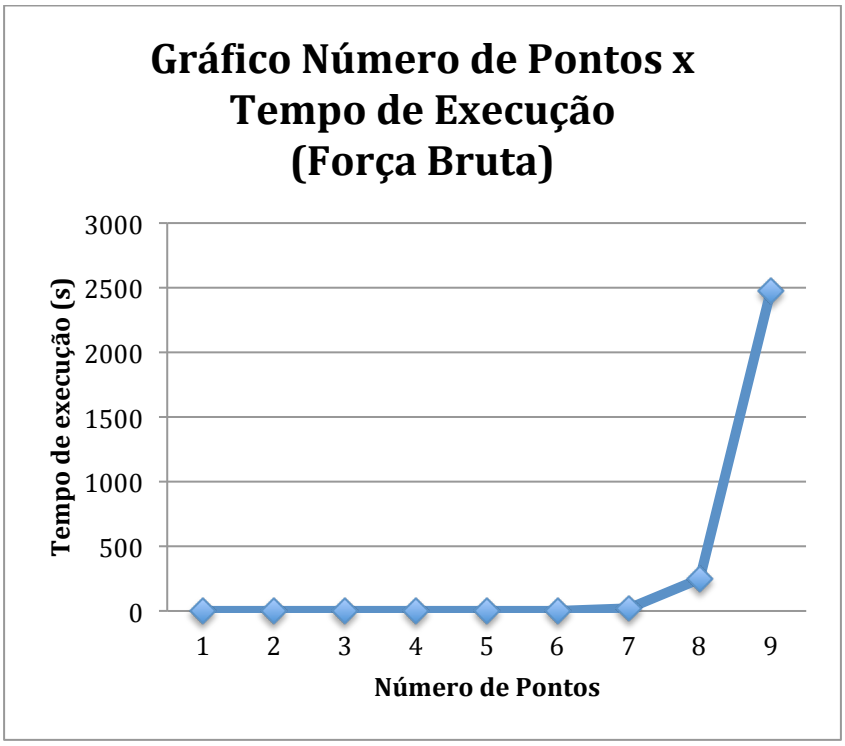
<ID_do_1º_vértice_com_a_cor_0> <ID_do_2º_vértice_com_a_cor_0> ... <ID_do_i-ésimo_vértice_com_a_cor_0>
<ID_do_1º_vértice_com_a_cor_1> <ID_do_2º_vértice_com_a_cor_1> ... <ID_do_i-ésimo_vértice_com_a_cor_1>
.
.
.
<ID_do_1º_vértice_ com_a_cor_k> <ID_do_2º_vértice_com_a_cor_k> ... <ID_do_i-ésimo_vértice_com_a_cor_k>

5. Avaliação Experimental

Foram realizados diversos testes com objetivo de avaliar e comparar o comportamento dos algoritmos implementados. Os testes foram executados em uma máquina equipada com um processador Intel Core 2 Duo @2.4GHz, com 4GB de RAM, executando o sistema operacional Mac OSX 10.9.5 (Mavericks).

Foram gerados gráficos com o intuito de avaliar o desempenho dos algoritmos a medida que o número de vértices do grafo cresce e, no caso do algoritmo paralelo, foi gerado um gráfico para medir o seu desempenho a medida que o número de threads cresce. Os resultados estão apresentados abaixo:

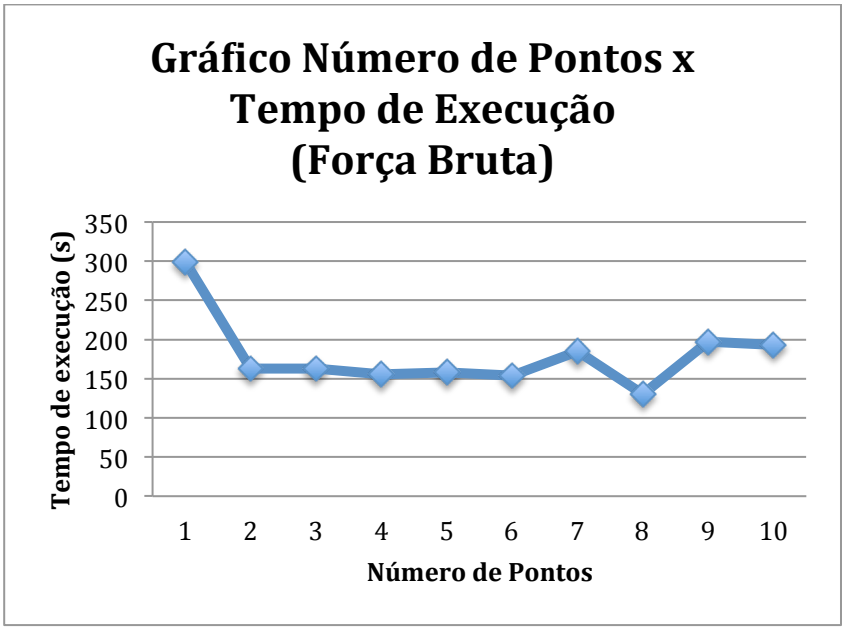
Algoritmo Força Bruta:
(número de cores disponíveis: 10)



Número de pontos	Tempo de Execução
1	0.007s
2	0.009s
3	0.007s
4	0.022s
5	0.180s
6	2s
7	22,4s
8	247,557s
9	2.477s

É interessante notar que quando o número de pontos aumenta em 1, o tempo de execução é multiplicado por um fator de aproximadamente 10 (número de cores) devido a complexidade exponencial do problema.

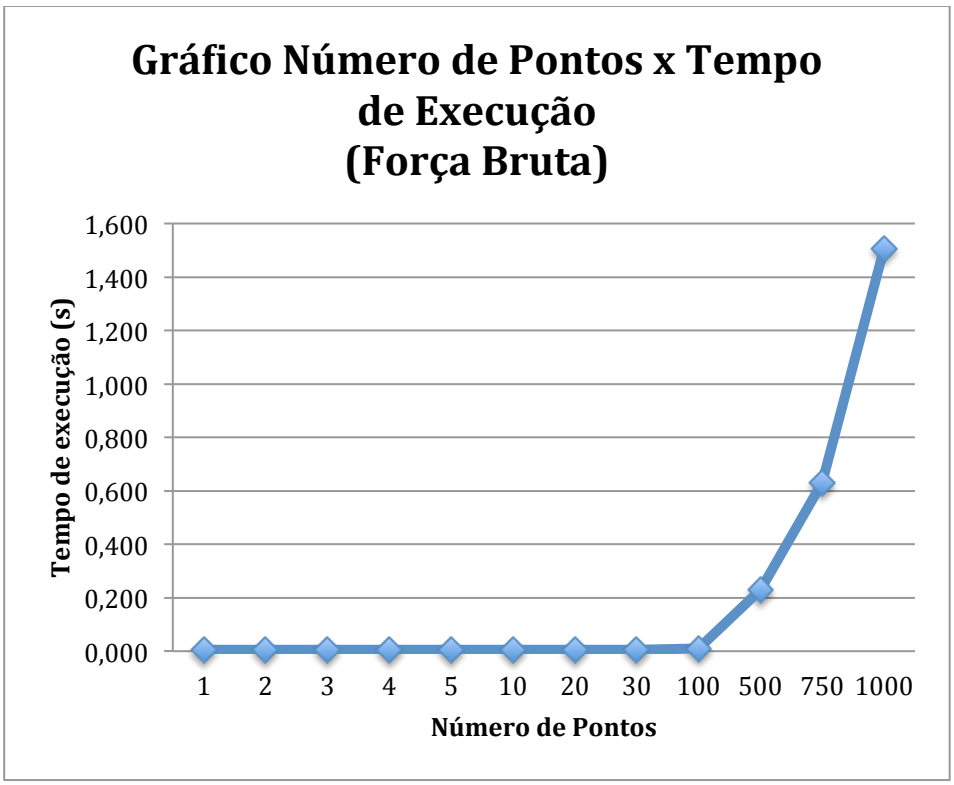
Algoritmo Paralelo:
(número de cores disponíveis: 10 / número de pontos no grafo: 8)



Número de Threads	Tempo de Execução
1	298,96
2	163,089
3	155,552
4	157,871
5	154
6	185
7	130
8	197,54
9	193

Observamos que só houve ganho considerável de performance quando passamos de 1 para 2 threads, um resultado esperado, já que o processador utilizado é dual core.

Heurística:
(número de cores disponíveis: 10)



Número de Pontos	Tempo de Execução
1	0,005s
2	0,005s
3	0,005s
4	0,005s
5	0,005s
10	0,005s
20	0,005s
30	0,005s
100	0,010s
500	0,230s
750	0,630s
1000	1,506s

Observamos que como a heurística tem complexidade polinomial, o crescimento do tempo de execução é bastante linear.

6. Conclusão

O trabalho mostrou de forma bastante clara a importância de se escolher uma estratégia adequada para resolver determinados problemas, já que as consequências dessa escolha não são apenas perceptíveis, mas também fundamentais para a viabilidade ou não da implementação. Através do desenvolvimento trabalho, foi possível observar na prática, os efeitos de muitos dos fenômenos que ocorrem no campo da ciência da computação, tais como o rápido crescimento do tempo de execução de problemas exponenciais em função do crescimento da entrada, a imprecisão dos resultados apresentados por heurísticas, etc.
O trabalho também permitiu que tenhamos uma visão mais ampla sobre o funcionamento e os efeitos do paralelismo, possibilitando o desenvolvimento de algoritmos que façam uso eficiente dos recursos de máquinas multi-core.

7. Referências

Ziviani, N., Projeto de Algoritmos com Implementações em Pascal e C, 3a Edição, Cengage Learning.