INF1721 - Relatório do Trabalho 2

Algoritmos de Fluxo Máximo e sua Aplicação em Classificação de Dados

Carlos Mattoso - 1210553
Gabriel Barros - 1111061
Michelle Valente - 1312828

Sumário

Particionador *(Clusterizer.java)*

Nesse trabalho usamos algoritmos de fluxo para encontrar K partições. Com isso, foi necessário rodar o algoritmo para $\forall u \ \forall v \ (u \neq v)$, $u, \ v \in V$, $onde \ Source : u \ \ e \ Target \ : v.$ Assim executamos as soluções de fluxo $|V|^2$, buscado a solução ótima: menor fluxo máximo. Fez-se uso de paralelização para determinar-se a solução para cada par $(Source, \ Target)$, o que ajudou determinar mais rapidamente a solução.

```
public class Clusterizer {
   public List<PartitionOrder> findClusters(Graph g, int totalK, IMaximumFlow flowSolver) {
          List<PartitionOrder> partitioningOrders = new ArrayList<>();
                int k = 1; // last assigned cluster index - initially all together

                while (k < totalK) {
                      int[] clustersIds = new int[k];
                      for(int i = 1; i <= k; i++)
                            clustersIds[i-1] = i;
                      IMaximumFlow.Solution[] candidateSolutions = new IMaximumFlow.Solution[k+1];

                      // Each cluster elects its optimal partition.
                      Arrays.stream(clustersIds).parallel().forEach(clusterId -> {
                            candidateSolutions[clusterId] = new IMaximumFlow.Solution();

                            for (int source : g.partitions.get(clusterId)) {
                                  IMaximumFlow.Solution candidateOptimalSolution = null; // optimal solution
for given source

                                  // Execute the flow algorithm for each pair of vertices (s,t) for s,t in `clusterIdx`
                                  for (int target : g.partitions.get(clusterId)) {
                                        if (source == target)
                                              continue;

                                        candidateOptimalSolution = flowSolver.solve(g, source, target);

                                        // If the solution found now is better, replace it.
                                        if (candidateOptimalSolution.maximumFlow <
candidateSolutions[clusterId].maximumFlow)
                                              candidateSolutions[clusterId] = candidateOptimalSolution;
                                  }
                            }
                      });

                      // Find the first cluster for which a partition was possible.
                      int init_part = 1;
                      for (int i = 1; i <= k; i++) {
                            if (candidateSolutions[i].partition != null) {
                                  init_part = i;
                                  break;
                            }
                      }

                      // Select the optimal partition. Assume the flow by partitioning cluster `1` is the minimum.
                      Integer optimalPartitionIdx = init_part;
                      IMaximumFlow.Solution optimalPartition = candidateSolutions[init_part];

                      PartitionOrder order = new PartitionOrder(init_part, k+1,
                            optimalPartition.partition.get(init_part).size(),
                            optimalPartition.partition.get(init_part + 1).size(),
```

```
                                optimalPartition.maximumFlow);

                        // No cluster before init_part had a non-null partition.
                        for (int i = init_part + 1; i <= k; i++) {
                                if (candidateSolutions[i].maximumFlow < optimalPartition.maximumFlow) {
                                        optimalPartition = candidateSolutions[i];
                                        optimalPartitionIdx = i;

                                        order.original = i;
                                        order.originalSize = optimalPartition.partition.get(i).size();
                                        order.created = k+1;
                                        order.createdSize = optimalPartition.partition.get(i + 1).size();
                                        order.cutValue = optimalPartition.maximumFlow;
                                }
                        }
                        partitioningOrders.add(order);

                        // Repartition the graph.
                        for (Entry<Integer, Set<Integer>> partition : optimalPartition.partition.entrySet()) {
                                /* In splitting the vertices, keep a partition with the original index
                                 * and the new one with the index of `k+1`.
                                 */
                                int partitionIndex = partition.getKey();
                                if (partitionIndex != optimalPartitionIdx)
                                        partitionIndex = k+1;

                                g.partitions.put(partitionIndex, partition.getValue());

                                // Adjust the vertex to partition mappings.
                                for (int v : partition.getValue())
                                        g.vertex_partition[v] = partitionIndex;
                        };

                        k++;
                }

                return partitioningOrders;
        }
}
```

Edmonds-Karp

O algoritmo de Edmonds-Karp é uma implementação do método Ford-Fulkerson para computar o fluxo máximo de um grafo, com a diferença que esse algoritmo sempre escolhe o caminho de aumento *(augmenting path)* com o menor número de arestas.

Para iniciar a análise de sua complexidade, o algoritmo leva $O(m)$ para achar o caminho com o menor número de arestas, já que utiliza-se uma busca em largura. A complexidade da busca em largura é essa uma vez que assumimos que todos os vértices do grafo tem pelo menos uma aresta incidente, o que de fato verifica-se para as instâncias de entrada.

Em seguida, tanto para aumentar o fluxo usando o caminho encontrado quanto para atualizar o grafo leva-se tempo $O(n)$. Logo, o *loop* principal do algoritmo internamente executa tarefas que tem complexidade $O(m)$.

A quantidade de vezes máxima que esse *loop* será executado pode ser compreendida imaginando-se um grafo em que todas as capacidades sejam infinitas e onde todos os vértices, tirando o *source* estão conectados ao *target*. Além disso, imagine que os vértices podem ser divididos em um número arbitrário de componentes conexas, dentro das quais cada vértice está conectado com apenas um outro.

Dada esta estrutura, o primeiro caminho mais curto a ser encontrado será *{source, primeiro vértice de uma componente, target}*. A cada interação a distância de *source* ao *target* no grafo é aumentada de pelo menos 1 unidade, podendo ser aumentada em até $n - 1$ unidades, já que esta é a distância máxima possível entre *source* e *target*. Observe além disso que, no máximo, só podem ser feitas $O(m)$ operações de aumento em caminhos, antes que a distância de *source* pro *target* tenha que aumentar de 1 unidade. Ora, a escolha de um caminho corresponde a escolha de sua aresta de gargalo, e existem no máximo m arestas. Em razão disto, o *loop* no máximo executa $O(nm)$ vezes. Portanto, o algoritmo tem complexidade de $O(nm^2)$.

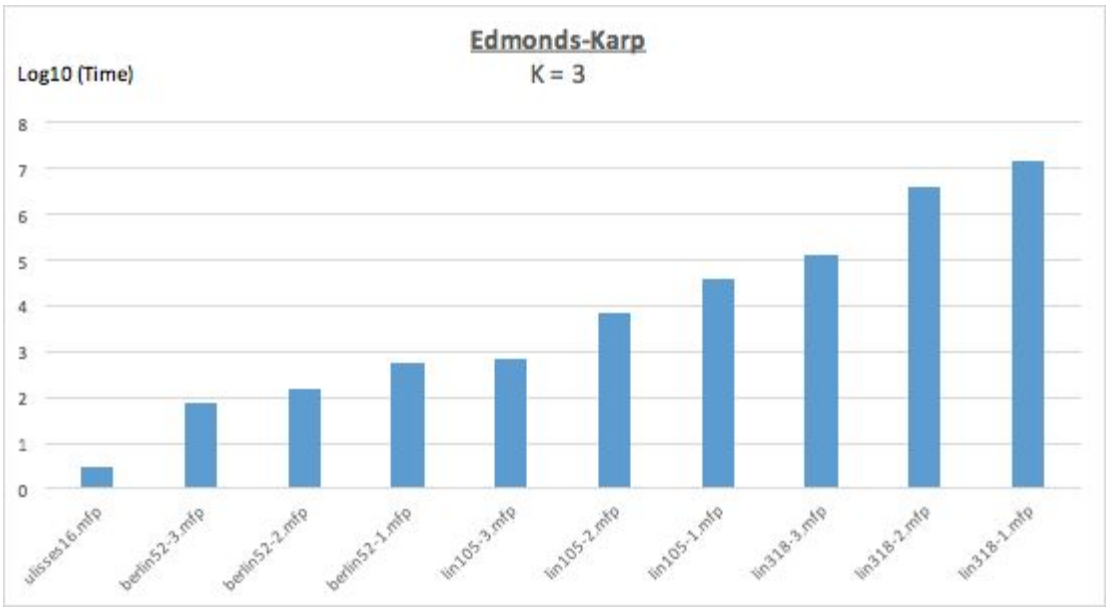
Análise da Implementação

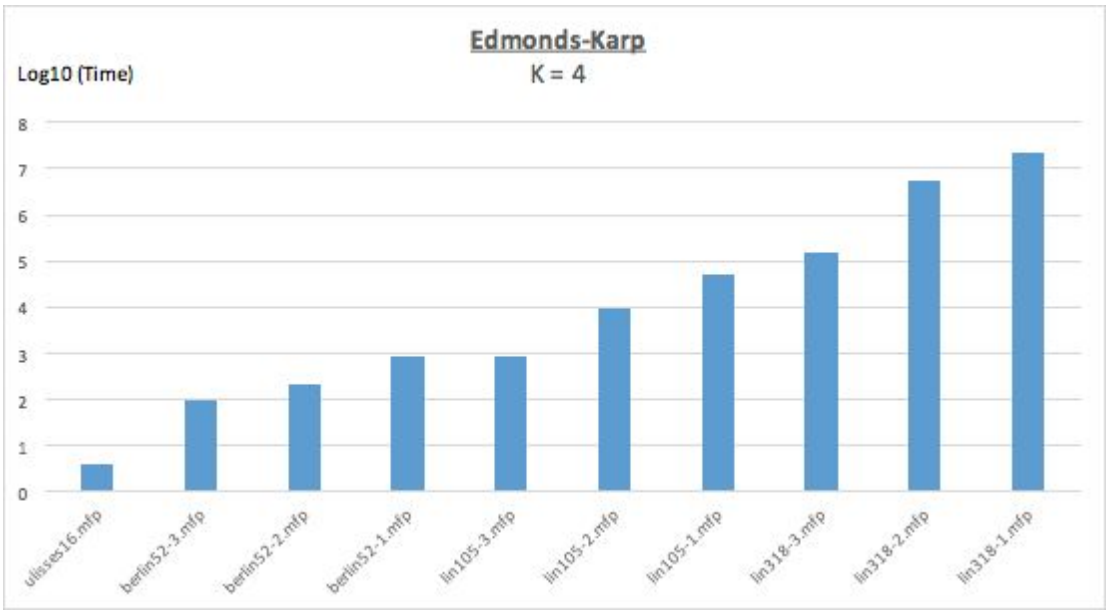
O *loop* principal do algoritmo implementado consiste na chamada à função que realiza a busca em largura e retorna o maior fluxo encontrado. A cada iteração o fluxo retornado é somado ao fluxo máximo, e quando o fluxo retornado da função for zero, o loop é interrompido. Após isso é chamada a função que gera as partições do grafo, e retornada a solução com elas e com o fluxo máximo. Levando-se em conta as chamadas feitas pelo *clusterizador,* a execução deste algoritmo tem complexidade na ordem de $O(n^3m^2)$.

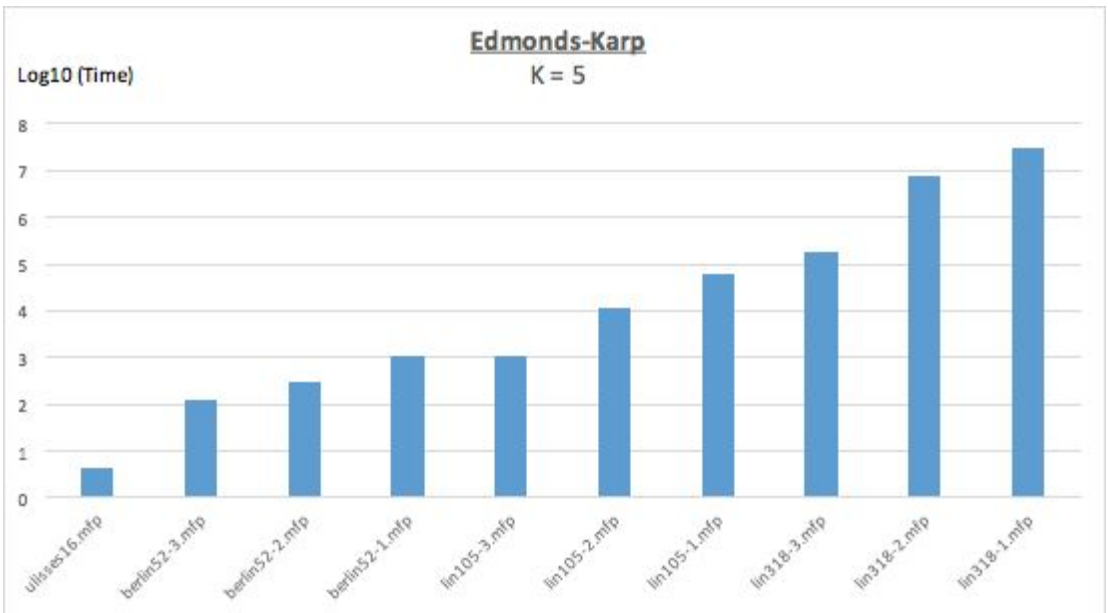
Abaixo apresenta-se o código da implementação da função de busca em largura.

```
public class EdmondsKarp extends IMaximumFlow {

	private int bfs(Graph g, int source, int target, int[][] f)
	{
		Queue<Integer> q = new LinkedList<>();
		int n = g.adjacencies.size();
		Integer[] previous = new Integer[n+1];
		int[] currentCapacity = new int[n+1];
		Arrays.fill(previous, null);
		Arrays.fill(currentCapacity, Integer.MAX_VALUE);

		previous[source] = source;
		q.add(source);

		while(!q.isEmpty())
		{
			int current = q.remove();

			for(Graph.Edge e : g.adjacencies.get(current))
			{
				int next = e.target;

				if(e.capacity - f[current][next] > 0 && previous[next] == null
					&& g.vertex_partition[current] == g.vertex_partition[next])
				{
					previous[next] = current;
					currentCapacity[next]  =  Math.min(currentCapacity[current],
e.capacity - f[current][next]);

					if(next != target)
					{
						q.add(next);
					}
					else // backtrack from `target` to `source`
					{
						int v = next; // v == target
						int df = currentCapacity[target];

						while(previous[v] != v)
						{
							int u = previous[v]; // u -> v

							f[u][v] += df;
							f[v][u] -= df;

							v = u;
						}

						return df;
					}
				}
			}
		}

		return 0;
	}
```

Resultados


Edmonds-Karp
K = 3
Log10 (Time)
8
7
6
5
4
3
2
1
0
ulisses16.mfp
berlin52-3.mfp
berlin52-2.mfp
berlin52-1.mfp
lin105-3.mfp
lin105-2.mfp
lin105-1.mfp
lin318-3.mfp
lin318-2.mfp
lin318-1.mfp


Edmonds-Karp
K = 4
Log10 (Time)
8
7
6
5
4
3
2
1
0
ulisses16.mfp
berlin52-3.mfp
berlin52-2.mfp
berlin52-1.mfp
lin105-3.mfp
lin105-2.mfp
lin105-1.mfp
lin318-3.mfp
lin318-2.mfp
lin318-1.mfp


Edmonds-Karp
K = 5
Log10 (Time)
8
7
6
5
4
3
2
1
0
ulisses16.mfp
berlin52-3.mfp
berlin52-2.mfp
berlin52-1.mfp
lin105-3.mfp
lin105-2.mfp
lin105-1.mfp
lin318-3.mfp
lin318-2.mfp
lin318-1.mfp

Preflow-Push

O algoritmo de *Preflow-Push* parte da ideia de se calcular um *pré-fluxo*, o qual é sempre maior ou igual ao fluxo que de fato sai de um vértice. A intuição é que calculando-se o pré-fluxo tem-se um bom ponto de partida que pode ser melhorado ao longo da execução do algoritmo, mas exigindo menos iterações. Além disso, em um pré-fluxo, não há necessidade de igualdade entra o fluxo de entrada e de saída do vértice.

Algumas definições inicias. Todo vértice tem um *excesso*, que indica quanto de fluxo está entrado a mais do que está saindo. Aqui quebra-se a definição tradicional de que tudo que entra é igual a tudo que sai. Contudo, este valor é zerado ao longo da execução, de modo que apenas o vértice alvo tenha excesso. Além disso, define-se uma *altura* para cada vértice, estrutura esta usada para a seleção de pares de vértices entre os quais fluxo pode passar.

Abaixo define-se um pseudocódigo geral para o algoritmo com definições de valores iniciais:

```
Preflow-Push(G(V, E), s, t):
      Preprocess:
            create a preflow f that saturates all out-edges of s
            let h[s] = |V|
            let h[v] = 0 ∀v ∈ V \ {s}
      while there is an applicable push or relabel operation
            Push/Relabel (ou Discharge)
```

Uma operação de *push* consiste em distribuir o máximo de fluxo de um vértice ativo (i.e. e[v] > 0) para um de vizinho seu admissível, isto é, um vizinho cuja altura seja igual ao incremento da altura do vértice ativo. Caso não haja operações de *push* possíveis, é necessária atualizar a altura do vértice, processo chamado de *relabel*. Este processo consiste em atualizar a altura do vértice em questão para o incremento da menor altura dentre seus vizinhos que ainda tenham capacidade residual positiva.

Ainda, utilizou-se a operação de *discharge*. Esta operação consiste em utilizar-se uma lista circular dos nós a serem processados. Percorrendo-se esta lista, zera-se o excesso de cada vértice ainda ativo, fazendo-se quantos *pushes* forem necessários. Caso neste processo seja preciso realizar um *relabel* no vértice, este é trazido de volta pro começo da lista e ela é percorrida do começo novamente. Utilizando-se esta estrutura o algoritmo tem complexidade *[PPT - Slide 53]* de $O(n^3)$.

Análise da Implementação

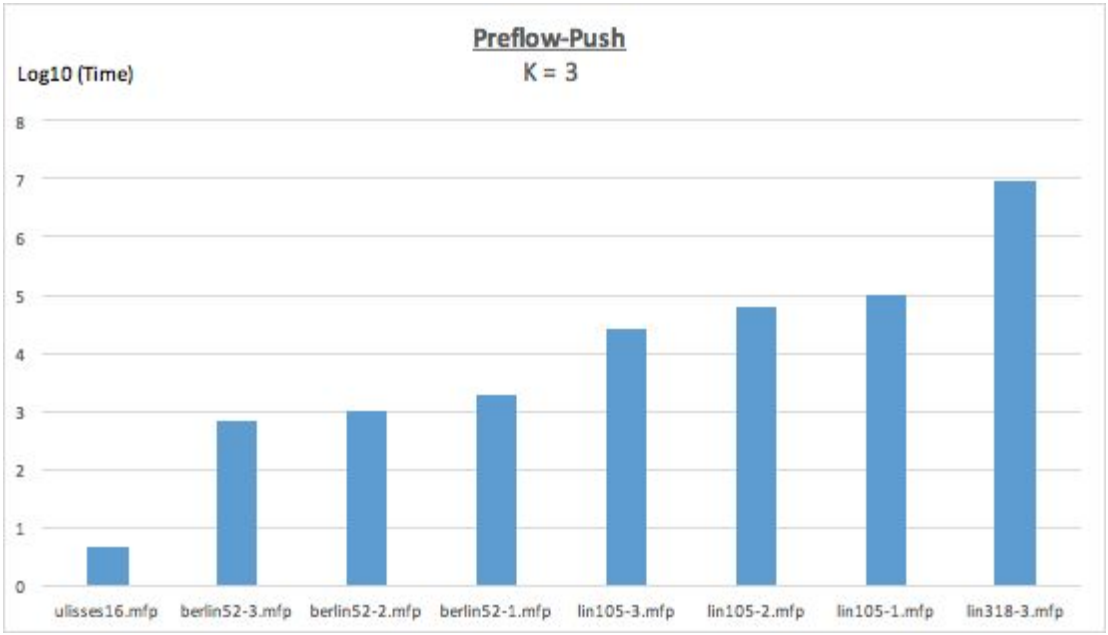
Abaixo apresenta-se o código da implementação de *PreflowPush*. Fizemos uma versão em *Java* do código em *Python* publicado na página da Wikipedia. Utiliza-se o procedimento de *discharge* descrito acima. Um ponto crítico desta implementação que explica seu desempenho abaixo do esperado é o que é feito após um *discharge* caso a altura do vértice sob análise tenha sido alterada. Observe que o vértice é movido para o começo da lista de nós sendo processados e o processamento desta é reinicializado.

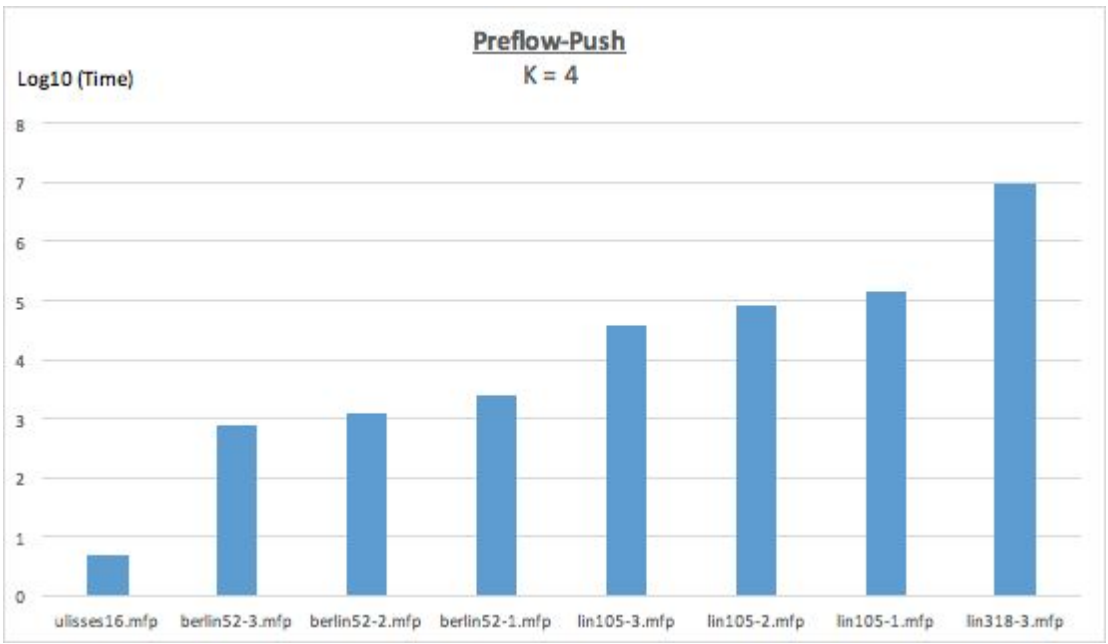
Note que, já que o algoritmo de *clusterização* tem complexidade de $O(n^2)$, utilizar este algoritmo acaba resultando em uma complexidade de $O(n^5)$, o que é pior do que a complexidade geral de $O(n^3m^2)$ utilizando-se o Edmonds Karp, já que para as instância de entrada é fato que $m << n^2$. De fato, os resultados precários decorrem de uma implementação não otimizada. Ainda assim, a diferença que se viu em tempo de execução foi de aproximadamente uma ordem de magnitude para as diferentes entradas, ressaltando que o projeto do *Preflow-Push* ajuda a mitigar os impactos negativos da implementação.

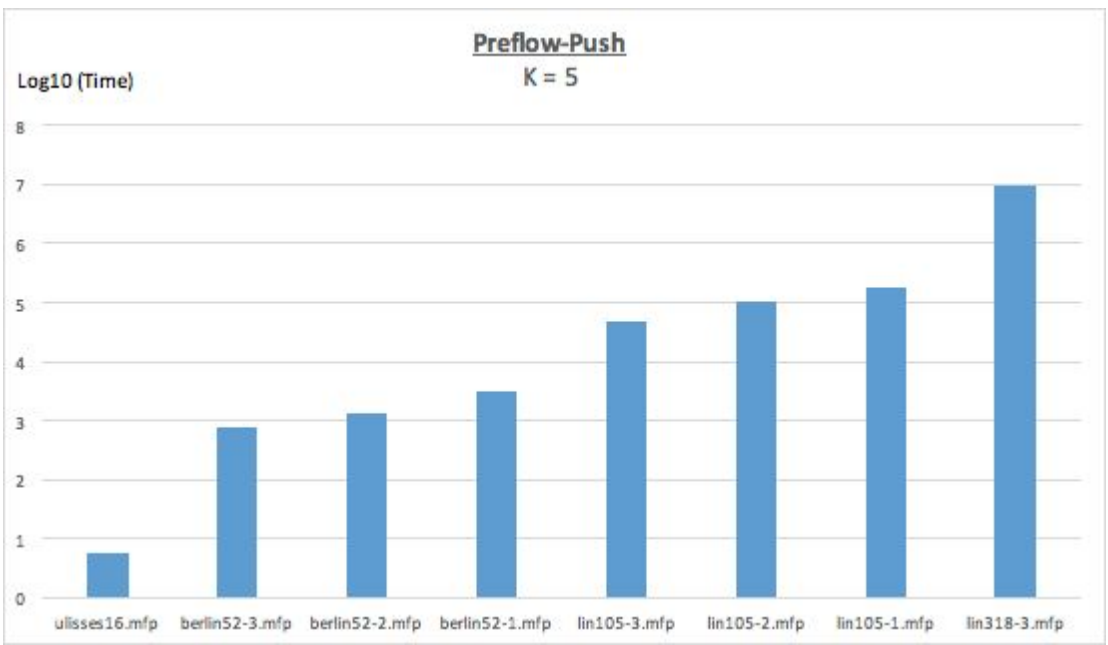
```
public class PreflowPush extends IMaximumFlow {
	private void push(Graph g, int u, int v, double capacity, int[] e, int[][] f) {...}
	private void relabel(Graph g, int u, int[] h, int[][] f) {...}

	private void discharge(Graph g, int u, int[] h, int[] seen, int[] e, int[][] f) {
		while (e[u] > 0) {
			int nNeighbors = g.adjacencies.get(u).size();

			/* Try to push all neighbors of `u` that belong to its partition,
			 * until all've been seen.
			 */
			if (seen[u] < nNeighbors) {
				Graph.Edge edge = g.adjacencies.get(u).get(seen[u]);
				int v = edge.target;

				if (edge.capacity - f[u][v] > 0 && h[u] > h[v]
					&& g.vertex_partition[u] == g.vertex_partition[v])
					push(g, u, v, edge.capacity, e, f);
				else
					seen[u]++;
			}
			else { // all neighbors of `u` have been checked
				relabel(g, u, h, f);
				seen[u] = 0; // `u` has been relabeled, so we can check its
neighbors again.
			}
		}
	}

    @Override
    public Solution solve(Graph g, Integer source, Integer target){
```

```
                int n = g.adjacencies.size(); // Number of vertices in partition `p`.
                int[] seen = new int[n+1];
                int[][] flows = new int[n+1][n+1];
                int[] height = new int[n+1]; // heights
                int[] excess = new int[n+1]; // excess flows

                // Create a preflow `f` that saturates all out-edges of `source`
                height[source] = n; // Longest path from source to sink is less than `n` long.
                excess[source] = Integer.MAX_VALUE; // Send as much flow as possible form source.
                for (Graph.Edge edge : g.adjacencies.get(source)) {
                        push(g, source, edge.target, edge.capacity, excess, flows);
                }

                /* Insert all vertices in partition `p` into the list of nodes to be seen
                 * (except for `source` and `target`.)
                 */
                Deque<Integer> nodeList = new ConcurrentLinkedDeque<Integer>();
                for (int i = 1; i <= n; i++) {
                        if (g.vertex_partition[i] == g.vertex_partition[source]
                                && i != source && i != target)
                                nodeList.add(i);
                }

                // Go over the list of nodes to be processed.
                Iterator<Integer> current = nodeList.iterator();
                while (current.hasNext()) {
                        int u = current.next();
                        int oldHeight = height[u];

                        // A discharge only happens if `u` is still an active node.
                        discharge(g, u, height, seen, excess, flows);

                        if (height[u] > oldHeight) {
                                nodeList.addFirst(u);
                                current.remove();
                                current = nodeList.iterator();
                        }
                }

                return repartition(g, source, excess[target], flows);
        }
}
```

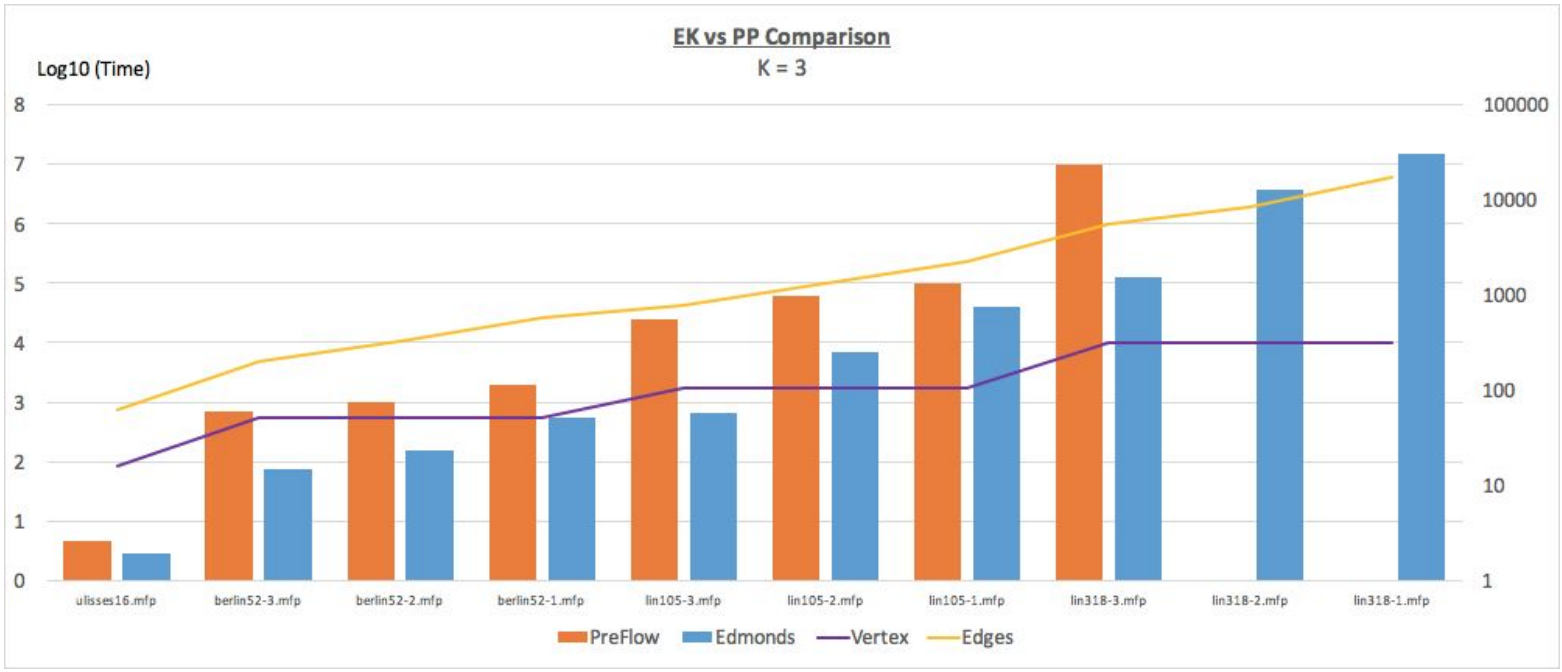
Resultados

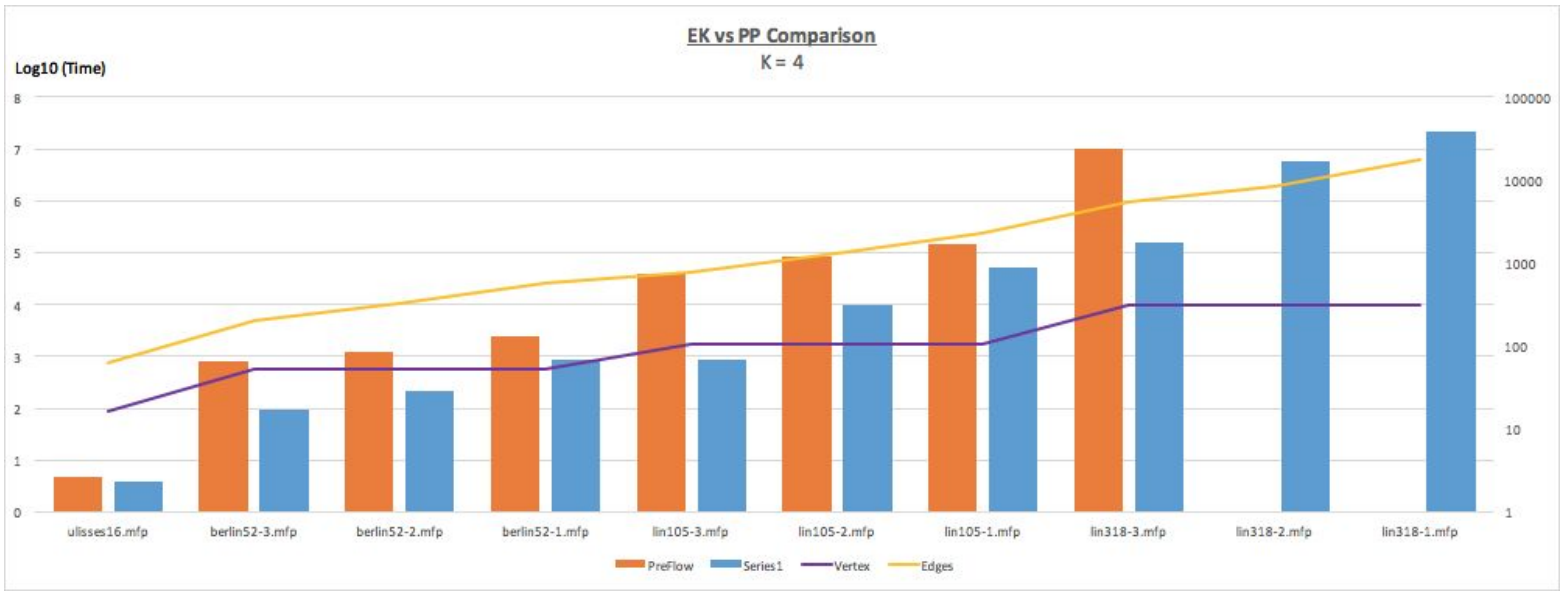

Preflow-Push
K = 3
Log10 (Time)
8
7
6
5
4
3
2
1
0
ulisses16.mfp
berlin52-3.mfp
berlin52-2.mfp
berlin52-1.mfp
lin105-3.mfp
lin105-2.mfp
lin105-1.mfp
lin318-3.mfp

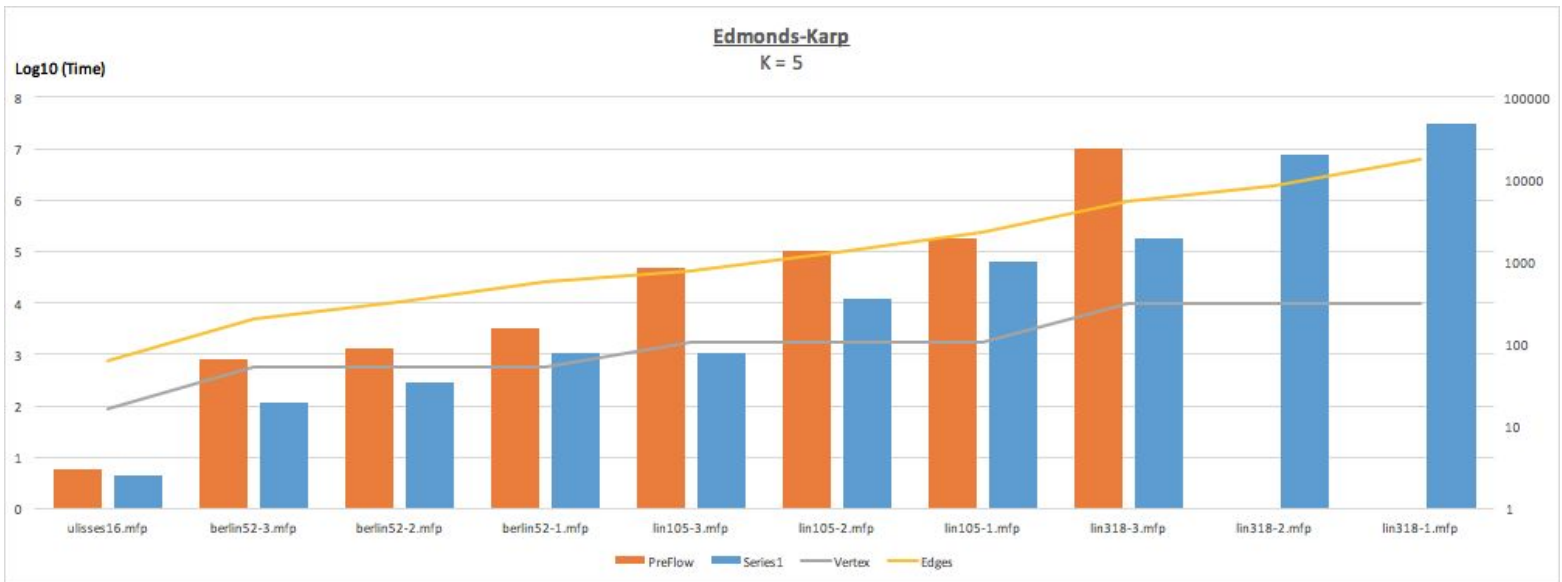

Preflow-Push
K = 4
Log10 (Time)
8
7
6
5
4
3
2
1
0
ulisses16.mfp
berlin52-3.mfp
berlin52-2.mfp
berlin52-1.mfp
lin105-3.mfp
lin105-2.mfp
lin105-1.mfp
lin318-3.mfp


Preflow-Push
K = 5
Log10 (Time)
8
7
6
5
4
3
2
1
0
ulisses16.mfp
berlin52-3.mfp
berlin52-2.mfp
berlin52-1.mfp
lin105-3.mfp
lin105-2.mfp
lin105-1.mfp
lin318-3.mfp

Gráficos Consolidados







Anexo

Resultados do Edmonds Karp

ulisses16.mfp // k: 3 // avg runtime 2.883619ms [# runs: 1734]
Partitioned 1 [16] into {1 [15], 2 [1]} with cut value 0.
Partitioned 1 [15] into {1 [13], 3 [2]} with cut value 1.

ulisses16.mfp // k: 4 // avg runtime 3.976983ms [# runs: 1258]
Partitioned 1 [16] into {1 [15], 2 [1]} with cut value 0.
Partitioned 1 [15] into {1 [13], 3 [2]} with cut value 1.
Partitioned 1 [13] into {1 [4], 4 [9]} with cut value 1.

ulisses16.mfp // k: 5 // avg runtime 4.335083ms [# runs: 1154]
Partitioned 1 [16] into {1 [15], 2 [1]} with cut value 0.
Partitioned 1 [15] into {1 [13], 3 [2]} with cut value 1.
Partitioned 1 [13] into {1 [4], 4 [9]} with cut value 1.
Partitioned 4 [9] into {4 [8], 5 [1]} with cut value 1.

berlin52-3.mfp // k: 3 // avg runtime 76.444622ms [# runs: 66]
Partitioned 1 [52] into {1 [27], 2 [25]} with cut value 0.
Partitioned 2 [25] into {2 [3], 3 [22]} with cut value 0.

berlin52-3.mfp // k: 4 // avg runtime 95.896676ms [# runs: 53]
Partitioned 1 [52] into {1 [27], 2 [25]} with cut value 0.
Partitioned 2 [25] into {2 [3], 3 [22]} with cut value 0.
Partitioned 3 [22] into {3 [1], 4 [21]} with cut value 0.

berlin52-3.mfp // k: 5 // avg runtime 118.276061ms [# runs: 43]
Partitioned 1 [52] into {1 [27], 2 [25]} with cut value 0.
Partitioned 2 [25] into {2 [3], 3 [22]} with cut value 0.
Partitioned 3 [22] into {3 [1], 4 [21]} with cut value 0.
Partitioned 4 [21] into {4 [4], 5 [17]} with cut value 0.

berlin52-2.mfp // k: 3 // avg runtime 156.702746ms [# runs: 32]
Partitioned 1 [52] into {1 [35], 2 [17]} with cut value 0.
Partitioned 2 [17] into {2 [1], 3 [16]} with cut value 0.

berlin52-2.mfp // k: 4 // avg runtime 215.642000ms [# runs: 24]
Partitioned 1 [52] into {1 [35], 2 [17]} with cut value 0.
Partitioned 2 [17] into {2 [1], 3 [16]} with cut value 0.
Partitioned 3 [16] into {3 [3], 4 [13]} with cut value 0.

berlin52-2.mfp // k: 5 // avg runtime 288.725645ms [# runs: 18]
Partitioned 1 [52] into {1 [35], 2 [17]} with cut value 0.
Partitioned 2 [17] into {2 [1], 3 [16]} with cut value 0.
Partitioned 3 [16] into {3 [3], 4 [13]} with cut value 0.
Partitioned 4 [13] into {4 [1], 5 [12]} with cut value 0.

berlin52-1.mfp // k: 3 // avg runtime 538.130960ms [# runs: 10]
Partitioned 1 [52] into {1 [47], 2 [5]} with cut value 0.
Partitioned 2 [5] into {2 [1], 3 [4]} with cut value 0.

berlin52-1.mfp // k: 4 // avg runtime 837.288677ms [# runs: 6]
Partitioned 1 [52] into {1 [47], 2 [5]} with cut value 0.
Partitioned 2 [5] into {2 [1], 3 [4]} with cut value 0.
Partitioned 3 [4] into {3 [3], 4 [1]} with cut value 0.

berlin52-1.mfp // k: 5 // avg runtime 1056.446167ms [# runs: 5]

Partitioned 1 [52] into {1 [47], 2 [5]} with cut value 0.
Partitioned 2 [5] into {2 [1], 3 [4]} with cut value 0.
Partitioned 3 [4] into {3 [3], 4 [1]} with cut value 0.
Partitioned 1 [47] into {1 [46], 5 [1]} with cut value 4.

lin105-3.mfp // k: 3 // avg runtime 661.203018ms [# runs: 8]
Partitioned 1 [105] into {1 [29], 2 [76]} with cut value 0.
Partitioned 2 [76] into {2 [1], 3 [75]} with cut value 0.

lin105-3.mfp // k: 4 // avg runtime 863.691331ms [# runs: 6]
Partitioned 1 [105] into {1 [29], 2 [76]} with cut value 0.
Partitioned 2 [76] into {2 [1], 3 [75]} with cut value 0.
Partitioned 3 [75] into {3 [4], 4 [71]} with cut value 0.

lin105-3.mfp // k: 5 // avg runtime 1057.015391ms [# runs: 5]
Partitioned 1 [105] into {1 [29], 2 [76]} with cut value 0.
Partitioned 2 [76] into {2 [1], 3 [75]} with cut value 0.
Partitioned 3 [75] into {3 [4], 4 [71]} with cut value 0.
Partitioned 4 [71] into {4 [2], 5 [69]} with cut value 0.

lin105-2.mfp // k: 3 // avg runtime 7080.605445ms [# runs: 1]
Partitioned 1 [105] into {1 [88], 2 [17]} with cut value 0.
Partitioned 2 [17] into {2 [9], 3 [8]} with cut value 0.

lin105-2.mfp // k: 4 // avg runtime 9801.079454ms [# runs: 1]
Partitioned 1 [105] into {1 [88], 2 [17]} with cut value 0.
Partitioned 2 [17] into {2 [9], 3 [8]} with cut value 0.
Partitioned 2 [9] into {2 [4], 4 [5]} with cut value 6.

lin105-2.mfp // k: 5 // avg runtime 11911.578589ms [# runs: 1]
Partitioned 1 [105] into {1 [88], 2 [17]} with cut value 0.
Partitioned 2 [17] into {2 [9], 3 [8]} with cut value 0.
Partitioned 2 [9] into {2 [4], 4 [5]} with cut value 6.
Partitioned 3 [8] into {3 [4], 5 [4]} with cut value 6.

lin105-1.mfp // k: 3 // avg runtime 38996.670540ms [# runs: 1]
Partitioned 1 [105] into {1 [97], 2 [8]} with cut value 639.
Partitioned 1 [97] into {1 [88], 3 [9]} with cut value 664.

lin105-1.mfp // k: 4 // avg runtime 52505.507682ms [# runs: 1]
Partitioned 1 [105] into {1 [97], 2 [8]} with cut value 639.
Partitioned 1 [97] into {1 [88], 3 [9]} with cut value 664.
Partitioned 2 [8] into {2 [4], 4 [4]} with cut value 689.

lin105-1.mfp // k: 5 // avg runtime 61487.246018ms [# runs: 1]
Partitioned 1 [105] into {1 [97], 2 [8]} with cut value 639.
Partitioned 1 [97] into {1 [88], 3 [9]} with cut value 664.
Partitioned 2 [8] into {2 [4], 4 [4]} with cut value 689.
Partitioned 3 [9] into {3 [4], 5 [5]} with cut value 689.

lin318-3.mfp // k: 3 // avg runtime 128248.499041ms [# runs: 1]
Partitioned 1 [318] into {1 [90], 2 [228]} with cut value 0.
Partitioned 2 [228] into {2 [105], 3 [123]} with cut value 0.

lin318-3.mfp // k: 4 // avg runtime 154418.532942ms [# runs: 1]
Partitioned 1 [318] into {1 [90], 2 [228]} with cut value 0.
Partitioned 2 [228] into {2 [105], 3 [123]} with cut value 0.
Partitioned 3 [123] into {3 [105], 4 [18]} with cut value 0.

lin318-3.mfp // k: 5 // avg runtime 176693.590527ms [# runs: 1]
Partitioned 1 [318] into {1 [90], 2 [228]} with cut value 0.
Partitioned 2 [228] into {2 [105], 3 [123]} with cut value 0.
Partitioned 3 [123] into {3 [105], 4 [18]} with cut value 0.
Partitioned 4 [18] into {4 [8], 5 [10]} with cut value 0.

lin318-2.mfp // k: 3 // avg runtime 3766547.937447ms [# runs: 1]
Partitioned 1 [318] into {1 [317], 2 [1]} with cut value 1182.
Partitioned 1 [317] into {1 [316], 3 [1]} with cut value 797.

lin318-2.mfp // k: 4 // avg runtime 5564086.517839ms [# runs: 1]
Partitioned 1 [318] into {1 [317], 2 [1]} with cut value 1182.
Partitioned 1 [317] into {1 [316], 3 [1]} with cut value 797.
Partitioned 1 [316] into {1 [315], 4 [1]} with cut value 1015.

lin318-2.mfp // k: 5 // avg runtime 7398078.095261ms [# runs: 1]
Partitioned 1 [318] into {1 [317], 2 [1]} with cut value 1182.
Partitioned 1 [317] into {1 [316], 3 [1]} with cut value 797.
Partitioned 1 [316] into {1 [315], 4 [1]} with cut value 1015.
Partitioned 1 [315] into {1 [314], 5 [1]} with cut value 772.

lin318-1.mfp // k: 3 // avg runtime 14936918.275507ms [# runs: 1]
Partitioned 1 [318] into {1 [317], 2 [1]} with cut value 3554.
Partitioned 1 [317] into {1 [316], 3 [1]} with cut value 3843.

lin318-1.mfp // k: 4 // avg runtime 22262596.116197ms [# runs: 1]
Partitioned 1 [318] into {1 [317], 2 [1]} with cut value 3554.
Partitioned 1 [317] into {1 [316], 3 [1]} with cut value 3843.
Partitioned 1 [316] into {1 [315], 4 [1]} with cut value 3886.

lin318-1.mfp // k: 5 // avg runtime 29473945.306945ms [# runs: 1]
Partitioned 1 [318] into {1 [317], 2 [1]} with cut value 3554.
Partitioned 1 [317] into {1 [316], 3 [1]} with cut value 3843.
Partitioned 1 [316] into {1 [315], 4 [1]} with cut value 3886.
Partitioned 1 [315] into {1 [314], 5 [1]} with cut value 4248.

Resultados do Preflow-Push

ulisses16.mfp // k: 3 // avg runtime 4.690939ms [# runs: 1066]
Partitioned 1 [16] into {1 [15], 2 [1]} with cut value 0.
Partitioned 1 [15] into {1 [13], 3 [2]} with cut value 1.

ulisses16.mfp // k: 4 // avg runtime 4.903682ms [# runs: 1020]
Partitioned 1 [16] into {1 [15], 2 [1]} with cut value 0.
Partitioned 1 [15] into {1 [13], 3 [2]} with cut value 1.
Partitioned 1 [13] into {1 [4], 4 [9]} with cut value 1.

ulisses16.mfp // k: 5 // avg runtime 5.838709ms [# runs: 857]
Partitioned 1 [16] into {1 [15], 2 [1]} with cut value 0.
Partitioned 1 [15] into {1 [13], 3 [2]} with cut value 1.
Partitioned 1 [13] into {1 [4], 4 [9]} with cut value 1.
Partitioned 4 [9] into {4 [8], 5 [1]} with cut value 1.

berlin52-3.mfp // k: 3 // avg runtime 693.447416ms [# runs: 8]
Partitioned 1 [52] into {1 [27], 2 [25]} with cut value 0.
Partitioned 2 [25] into {2 [3], 3 [22]} with cut value 0.

berlin52-3.mfp // k: 4 // avg runtime 787.014665ms [# runs: 7]
Partitioned 1 [52] into {1 [27], 2 [25]} with cut value 0.
Partitioned 2 [25] into {2 [3], 3 [22]} with cut value 0.
Partitioned 3 [22] into {3 [1], 4 [21]} with cut value 0.

berlin52-3.mfp // k: 5 // avg runtime 797.150684ms [# runs: 7]
Partitioned 1 [52] into {1 [27], 2 [25]} with cut value 0.
Partitioned 1 [27] into {1 [1], 3 [26]} with cut value 0.
Partitioned 2 [25] into {2 [3], 4 [22]} with cut value 0.
Partitioned 4 [22] into {4 [1], 5 [21]} with cut value 0.

berlin52-2.mfp // k: 3 // avg runtime 995.170155ms [# runs: 6]
Partitioned 1 [52] into {1 [35], 2 [17]} with cut value 0.
Partitioned 2 [17] into {2 [1], 3 [16]} with cut value 0.

berlin52-2.mfp // k: 4 // avg runtime 1193.811031ms [# runs: 5]
Partitioned 1 [52] into {1 [35], 2 [17]} with cut value 0.
Partitioned 2 [17] into {2 [1], 3 [16]} with cut value 0.
Partitioned 3 [16] into {3 [3], 4 [13]} with cut value 0.

berlin52-2.mfp // k: 5 // avg runtime 1319.908670ms [# runs: 4]
Partitioned 1 [52] into {1 [35], 2 [17]} with cut value 0.
Partitioned 2 [17] into {2 [1], 3 [16]} with cut value 0.
Partitioned 3 [16] into {3 [3], 4 [13]} with cut value 0.
Partitioned 4 [13] into {4 [1], 5 [12]} with cut value 0.

berlin52-1.mfp // k: 3 // avg runtime 1949.387770ms [# runs: 3]
Partitioned 1 [52] into {1 [47], 2 [5]} with cut value 0.
Partitioned 2 [5] into {2 [1], 3 [4]} with cut value 0.

berlin52-1.mfp // k: 4 // avg runtime 2466.909340ms [# runs: 3]
Partitioned 1 [52] into {1 [47], 2 [5]} with cut value 0.
Partitioned 2 [5] into {2 [1], 3 [4]} with cut value 0.
Partitioned 3 [4] into {3 [3], 4 [1]} with cut value 0.

berlin52-1.mfp // k: 5 // avg runtime 3229.585697ms [# runs: 2]
Partitioned 1 [52] into {1 [47], 2 [5]} with cut value 0.
Partitioned 2 [5] into {2 [1], 3 [4]} with cut value 0.
Partitioned 3 [4] into {3 [3], 4 [1]} with cut value 0.
Partitioned 1 [47] into {1 [46], 5 [1]} with cut value 4.

lin105-3.mfp // k: 3 // avg runtime 25282.236872ms [# runs: 1]
Partitioned 1 [105] into {1 [29], 2 [76]} with cut value 0.
Partitioned 2 [76] into {2 [1], 3 [75]} with cut value 0.

lin105-3.mfp // k: 4 // avg runtime 38616.514053ms [# runs: 1]
Partitioned 1 [105] into {1 [29], 2 [76]} with cut value 0.
Partitioned 2 [76] into {2 [1], 3 [75]} with cut value 0.
Partitioned 3 [75] into {3 [4], 4 [71]} with cut value 0.

lin105-3.mfp // k: 5 // avg runtime 48244.775674ms [# runs: 1]
Partitioned 1 [105] into {1 [29], 2 [76]} with cut value 0.
Partitioned 2 [76] into {2 [1], 3 [75]} with cut value 0.
Partitioned 3 [75] into {3 [4], 4 [71]} with cut value 0.
Partitioned 4 [71] into {4 [2], 5 [69]} with cut value 0.

lin105-2.mfp // k: 3 // avg runtime 60388.261078ms [# runs: 1]
Partitioned 1 [105] into {1 [88], 2 [17]} with cut value 0.
Partitioned 2 [17] into {2 [9], 3 [8]} with cut value 0.

lin105-2.mfp // k: 4 // avg runtime 82482.154848ms [# runs: 1]
Partitioned 1 [105] into {1 [88], 2 [17]} with cut value 0.
Partitioned 2 [17] into {2 [9], 3 [8]} with cut value 0.
Partitioned 2 [9] into {2 [4], 4 [5]} with cut value 6.

lin105-2.mfp // k: 5 // avg runtime 104986.526541ms [# runs: 1]
Partitioned 1 [105] into {1 [88], 2 [17]} with cut value 0.
Partitioned 2 [17] into {2 [9], 3 [8]} with cut value 0.
Partitioned 2 [9] into {2 [4], 4 [5]} with cut value 6.
Partitioned 3 [8] into {3 [4], 5 [4]} with cut value 6.

lin105-1.mfp // k: 3 // avg runtime 97740.093805ms [# runs: 1]
Partitioned 1 [105] into {1 [97], 2 [8]} with cut value 639.
Partitioned 1 [97] into {1 [88], 3 [9]} with cut value 664.

lin105-1.mfp // k: 4 // avg runtime 142710.135654ms [# runs: 1]
Partitioned 1 [105] into {1 [97], 2 [8]} with cut value 639.
Partitioned 1 [97] into {1 [88], 3 [9]} with cut value 664.
Partitioned 2 [8] into {2 [4], 4 [4]} with cut value 689.

lin105-1.mfp // k: 5 // avg runtime 180184.823484ms [# runs: 1]
Partitioned 1 [105] into {1 [97], 2 [8]} with cut value 639.
Partitioned 1 [97] into {1 [88], 3 [9]} with cut value 664.
Partitioned 2 [8] into {2 [4], 4 [4]} with cut value 689.
Partitioned 3 [9] into {3 [4], 5 [5]} with cut value 689.

lin318-3.mfp // k: 3 // avg runtime 9527824.209668ms [# runs: 1]
Partitioned 1 [318] into {1 [90], 2 [228]} with cut value 0.
Partitioned 2 [228] into {2 [105], 3 [123]} with cut value 0.

lin318-3.mfp // k: 4 // avg runtime 9822437.347690ms [# runs: 1]
Partitioned 1 [318] into {1 [90], 2 [228]} with cut value 0.
Partitioned 2 [228] into {2 [105], 3 [123]} with cut value 0.
Partitioned 3 [123] into {3 [105], 4 [18]} with cut value 0.

lin318-3.mfp // k: 5 // avg runtime 9912608.598590ms [# runs: 1]
Partitioned 1 [318] into {1 [90], 2 [228]} with cut value 0.
Partitioned 2 [228] into {2 [105], 3 [123]} with cut value 0.
Partitioned 3 [123] into {3 [105], 4 [18]} with cut value 0.
Partitioned 4 [18] into {4 [8], 5 [10]} with cut value 0.