Um Estudo do Algoritmo de Goldberg e Tarjan para o Problema do Fluxo Máximo

Este exemplar corresponde a redação final da tese devidamente corrigida e defendida pelo Sr. Gustavo Peixoto Silva e aprovada pela Comissão Julgadora.

Campinas, 24 de Fevereiro de 1992.

Prof. Dr. Clovis Perin Filho

Agradecimentos

- Ao Prof. Dr. Clovis Perin Filho pela orientação e amizade.
- Aos membros da banca pelo interesse demonstrado.
- Aos professores e alunos da área pela troca de idéias, informações e sugestões que auxiliaram muito neste trabalho.
- A todos os professores que direta ou indiretamente contribuiram na minha formação profissional.
- A todos os colegas e amigos pela convivência agradável neste período.
- A todos os funcionários que mantêm o IMECC em funcionamento.
- Às instituições CAPES e CNPq pelo suporte financeiro.

Conteúdo

Resumo

Este trabalho consiste no estudo e na implementação do algoritmo de Goldberg e Tarjan para o problema do fluxo máximo. Este algoritmo tem destacada importância por apresentar uma das complexidades mais baixas e também pelo fato de abordar o problema de maneira diferenciada.

Goldberg e Tarjan utilizam a estrutura de dados árvores dinâmicas para atingir a complexidade $O(nm\log(n^2/m))$ numa rede *n-nós*, *m-arcos*. Em redes densas ($m = O(n^2)$) a complexidade deste algoritmo é tão boa quanto qualquer outro algoritmo, tendo uma das melhores complexidades em redes de densidade moderada ($m = O(n^{3/2})$).

Este algoritmo apresenta duas versões, uma que não utiliza a estrutura de dados árvores dinâmicas e tem complexidade $O(n^3)$, e outra versão que incorpora ao algoritmo anterior as árvores dinâmicas, conseguindo a complexidade de $O(nm\log(n^2/m))$.

Foram realizados testes comparativos com as duas versões e os principais algoritmos conhecidos para o problema, tendo em vista o tempo de CPU em cada método. As redes utilizadas neste trabalho têm características particulares.

Capítulo 1

Introdução

1.1 Apresentação do Problema

Problemas de envio e movimentação de objetos de um determinado local para outro são comuns na vida real; por exemplo: a remessa de produtos de uma fábrica para um distribuidor, o movimento de pessoas das casas para os locais de trabalho, o envio de cartas dos remetentes para os destinatários, ou mesmo a alocação de tarefas em máquinas.

Este tipo de movimento é chamado de **fluxo**. Pontos onde os objetos são gerados para formar o fluxo são denominados **origens** ou **fontes** e pontos onde são consumidos os objetos são **destinos** ou **sorvedouros**. Os pontos de convêrgencia ou divergência de fluxo, isto é, as origens, os destinos e os pontos de passagem de fluxo são denominados **nós**. O fluxo de objetos está restrito às suas devidas vias de transporte que denominamos **arcos**. O conjunto dos nós (pontos) e arcos (vias de transporte) definem uma **rede** e o movimento de objetos através deste sistema define um **fluxo em rede**. É comum que haja um limitante superior para o fluxo que atravessa cada arco, o que define a **capacidade do arco**. Além disto, também é comum considerar o **custo** por objeto transportado em cada arco, de tal forma que surge o objetivo de minimizar o custo total de transporte dos objetos, satisfazendo à lei de demanda e consumo definida nos nós e satisfazendo a capacidade de transporte dos arcos.

Apesar da variedade de situações em que os problemas de fluxo em redes se apresentam, existem problemas clássicos tais como:

1. Problema do Caminho Mínimo - determinar o caminho de menor custo para enviar um dado número de objetos de uma origem para um destino através de uma rede.

2. Problema de PERT/CPM - determinar o caminho de maior custo da origem para um destino em uma rede acíclica.

3. Problema de Fluxo Ótimo (ou de Custo Mínimo ou mesmo de Transporte ou de Transbordo) - determinar um fluxo que satisfaça a demanda dos nós da rede com custo mínimo.

4. Problema de Designação - alocar operários em máquinas.

5. Problema de Fluxo Máximo - maximizar a quantidade de objetos transportados de uma origem para um destino.

Muitos problemas práticos podem ser vistos como um problema de fluxo máximo ou então como uma seqüência de problemas de fluxo máximo [21]. Podemos, por exemplo, encontrar um fluxo ótimo maximizando o fluxo através dos arcos de menor custo até satisfazer às demandas dos nós.

O problema do fluxo máximo também pode ser formulado como um caso especial de Programação Linear [23]. Considere um arco adicional ligando o destino à origem com custo -1; os outros arcos da rede tem custo nulo. O Programa Linear consiste em minimizar o custo do fluxo sujeito à lei de equilíbrio dos nós; isto é, a quantidade de fluxo que entra em cada nó deve ser igual à quantidade de fluxo que sai deste nó. Desta forma, o problema do fluxo máximo pode ser resolvido aplicando o método Simplex; no entanto, algoritmos específicos e mais eficientes têm sido desenvolvidos.

1.2 Histórico da Evolução do Algoritmo

Considere uma rede capacitada com n nós, m arcos, nó origem s e nó destino t. Determinar um fluxo em uma rede é determinar o fluxo em cada arco da rede. Um fluxo é máximo se o seu valor, a soma dos fluxos nos arcos que saem da origem (ou que chegam no destino) é máximo.

O problema do fluxo máximo foi formulado por Fulkerson e Dantzig [14] e primeiramente resolvido por Ford e Fulkerson [12]. Neste trabalho, Ford

e Fulkerson definiram que dado um fluxo em rede, um **caminho aumentante** é um caminho da origem ao destino que permite um aumento no valor do fluxo corrente. Além disso, demonstraram o seguinte teorema:

"Um fluxo corrente é máximo se e somente se não existe caminho aumentante para este fluxo."

A partir daí, um grande número de algoritmos têm sido desenvolvidos para resolver o problema, pois encontrar um caminho aumentante não requer um esforço computacional tão grande quanto resolver um sistema linear deste porte (um caminho aumentante pode ser encontrado em $O(m)$ operações no pior caso). Além disso, os algoritmos de fluxo em rede são geralmente simples, de fácil implementação e fortemente polinomiais (isto é envolvem apenas adição, subtração e comparação lógica).

O método de Ford e Fulkerson [12] parte de um fluxo inicial nulo e a cada iteração procura encontrar um caminho aumentante. Neste caso, uma ampliação do fluxo é efetuada e um novo fluxo corrente é obtido. O processo é iterativo e termina quando não há mais nenhum caminho aumentante; neste instante o fluxo corrente é máximo.

Edmonds e Karp [10] introduziram o conceito de **caminho aumentante mínimo** (menor número de arcos) e desenvolveram um algoritmo com complexidade $O(nm^2)$. Este algoritmo é semelhante ao anterior, onde cada caminho aumentante deve ser mínimo, desta forma o esforço para encontrar todos os caminhos aumentantes mínimos é $O(m^2)$. Este foi o primeiro algoritmo polinomial para o problema; isto é, um algoritmo em que o número de operações, no pior caso, pode ser limitado por uma função polinomial dos dados do problema. As complexidades são utilizadas teoricamente na comparação do desempenho entre algoritmos.

Dinic [9] propôs o processamento em paralelo de um conjunto maximal de caminhos aumentantes mínimos (de mesmo tamanho), de modo a tornar o processo mais eficiente. Assim, no estágio seguinte o comprimento dos caminhos aumentantes mínimos torna-se maior. O algoritmo está dividido em estágios. Em cada estágio é construído um **referente** a partir do fluxo corrente. Um referente (ou rede em camadas) possui os nós particionados em camadas V_0, V_1,..., V_k, onde $V_0 = s$, $V_k = t$, e cada um de seus arcos vai de uma camada para a camada seguinte. Além disso, se v é o i-ésimo nó de um caminho aumentante mínimo de s para t então $v \in V_{i-1}$. Dinic

determina um **fluxo bloqueante** no referente, o que implica em encontrar um conjunto maximal de caminhos aumentantes mínimos (de mesmo tamanho). A cada iteração, o algoritmo de Dinic estrutura todos os caminhos aumentantes mínimos, com a ajuda do referente, e determina um fluxo bloqueante em $O(nm)$. O número de estágios do algoritmo de Dinic é limitado por n visto que ao final de um estágio, o comprimento dos caminhos mínimos aumenta. Com esta estratégia, é obtido um algoritmo com complexidade $O(n^2m)$.

Algorítmos mais eficientes foram desenvolvidos a partir do trabalho de Dinic. Nenhum destes algoritmos reduziu o número de estágios mas alguns reduziram o esforço computacional realizado em cada estágio. Desta forma, o problema do fluxo máximo tem sido resolvido por uma seqüência de problemas de fluxo bloqueante.

Karzanov [22] introduziu o conceito de **prefluxo**: fluxo relaxado no sentido de que um nó qualquer diferente de s e de t pode receber uma quantidade maior de prefluxo do que envia. Este excesso de fluxo nos nós é gerado enquanto envia-se fluxo de camada em camada na direção de s para t. Posteriormente retorna-se o fluxo excedente para a camada anterior a fim de procurar um caminho alternativo em direção ao destino. O algoritmo de Karzanov leva vantagem teórica sobre o de Dinic pois embora o método pague um preço pelas tentativas malsucedidas, pois tem que retornar o fluxo excedente, mostra-se que este preço não é tão grande visto que sua complexidade é de $O(n^3)$.

O algoritmo de Cherkasky [6] é um refinamento do algoritmo de Karzanov. Cherkasky particionou as camadas em blocos de camadas consecutivas chamando-as de **supercamadas** e aplicou o algoritmo de Karzanov às supercamadas. No interior das supercamadas, Cherkasky utilizou o método de Dinic. Combinando os métodos de Dinic e Karzanov, Cherkasky obteve um algoritmo teoricamente melhor do que ambos. Ao aplicar o método de Karzanov às supercamadas, Cherkasky reduziu o preço mencionado acima no caso dos excessos retornados às camadas anteriores. Por outro lado, ele utilizou o método de Dinic no interior das supercamadas, o qual se mostra eficiente por percorrer uma distância relativamente curta (a espessura da supercamada) para detectar se pode haver um envio de fluxo através da supercamada. Este algoritmo tem complexidade $O(n^2\sqrt{m})$.

Galil [16] propôs uma implementação eficiente do algoritmo de Cherkasky. Para cada supercamada, ele adotou uma estrutura chamada de

floresta que armazena sob a forma de um arco um caminho na supercamada. Esta estrutura permite um ganho de tempo pois pode-se percorrer um caminho (possivelmente longo) em apenas um passo. Embora a idéia principal seja relativamente simples, os detalhes técnicos de como utilizar e manter as florestas são um tanto elaborados. Desta forma é obtido um algoritmo com complexidade $O(n^{5/3}m^{2/3})$.

Enquanto o algoritmo de Dinic perde algumas informações a respeito de caminhos aumentantes já utilizados, o algoritmo de Galil e Naamad [17] armazena estas informações; isto é, os caminhos remanescentes após a retirada dos arcos saturados. Estes caminhos são denominados **fragmentos de caminho**. A maior dificuldade neste algoritmo está na possibilidade de que durante a construção de um caminho, pode-se utilizar parte de um fragmento de caminho. Então é necessário quebrar aquele fragmento de caminho e concatenar a parte de interesse com o caminho em construção. Este trabalho desenvolvido por Galil e Naamad é mais uma implementação eficiente do algoritmo de Dinic e tem complexidade $O(nm\log(n)^2)$.

Malhotra et alli [25] apresentaram um algoritmo também com complexidade $O(n^3)$ porém mais simples do que o de Karzanov. Este algoritmo introduziu o conceito de **potencial de fluxo** para cada nó da rede. O potencial de fluxo é o máximo de fluxo extra que pode atravessar cada nó. Se r é um nó da rede com potencial mínimo em relação aos outros nós da rede, então r é chamado de **nó referente**. A cada iteração um nó referente é determinado e partindo dele um fluxo adicional igual ao potencial referente é empurrado em direção ao destino e também puxado da origem. Todos os arcos saturados devem ser retirados da rede assim como os nós que não possuem arcos de entrada ou arcos de saída (com potencial nulo).

Em redes densas ($m = O(n^2)$) os algoritmos de Karzanov, Cherkasky, Malhotra et alli, e Galil rodam em $O(n^3)$. Destes algoritmos, o de Malhotra é bem mais simples conceitualmente do que os demais.

Tarjan [34] desenvolveu uma versão do algoritmo de Karzanov de fácil implementação, denominado **algoritmo da onda**. Este algoritmo resolve o problema de encontrar um fluxo bloqueante alternando passagens para frente (ida) e para trás (volta) através da rede. Durante uma ida, o algoritmo envia o máximo de fluxo e o mais distante possível. Na volta, ele retorna fluxo que ficou bloqueado para a camada anterior para que na próxima ida, unidades de fluxo sejam enviadas por caminhos alternativos.

Este algoritmo proposto por Tarjan, além de ser fácil de implemen-

tar, necessita de espaço de memória menor do que os outros. Enquanto o método original de Karzanov utiliza um espaço de $O(n^2)$, o algoritmo da onda utiliza um espaço de $O(m)$. O algoritmo de Malhotra et alli requer dois números e dois apontadores por nó, o algoritmo da onda necessita de um bit, um número e um apontador por nó.

Goldberg e Tarjan [20] utilizando o conceito de prefluxo, introduzido por Karzanov, e uma estrutura de dados tipo **árvores dinâmicas** [32], desenvolveram um algoritmo com complexidade $O(nm \log(n^2/m))$. O algoritmo abandona a idéia de encontrar um fluxo em cada estágio e também abandona a idéia de um estágio global. Sua estratégia é manter um prefluxo na rede original e enviar excessos de fluxo local em direção ao destino pelo caminho mais curto na rede residual. Excessos de fluxo que não podem chegar ao destino são retornados à origem, também pelo caminho mais curto. Somente quando o algoritmo termina é que o prefluxo torna-se um fluxo máximo.

Martel [26] ao fazer testes comparativos entre o algoritmo de Goldberg e Tarjan [20] e algoritmos "tipo Dinic" escolheu o algoritmo da onda de Tarjan como representante desta classe.

Na tabela que se segue apresentamos os principais algoritmos com suas respectivas complexidades. O parâmetro U corresponde ao limitante superior para a capacidade de cada arco na rede, neste caso os arcos devem ter capacidade racional, enquanto os outros casos permitem capacidades reais.

ANO	COMPLEXIDADE	AUTOR
1956	—	Ford e Fulkerson
1969	$O(nm^2)$	Edmonds e Karp
1970	$O(n^2m)$	Dinic
1974	$O(n^3)$	Karzanov
1977	$O(n^2m^{1/2})$	Cherkasky
1978	$O(n^3)$	Malhotra et alli
1978	$O(n^{5/3}m^{2/3})$	Galil
1978	$O(nm(\log(n))^2)$	Galil e Naamad
1980	$O(nm\log(n))$	Sleator e Tarjan
1982	$O(n^3)$	Shiloach e Vishkin
1984	$O(nm\log(U))$	Gabow
1984	$O(n^3)$	Tarjan
1985	$O(n^3)$	Goldberg
1986	$O(nm\log(n^2/m))$	Goldberg e Tarjan
1986	$O(nm + n^2\log(U))$	Ahuja e Orlin
1989	$O(nm\log((n/m)((\log(U))^{1/2}+2))$	Ahuja, Orlin e Tarjan

Tabela 1.1: Algoritmos Polinomiais para o Problema do Fluxo Máximo

Acreditamos que o algoritmo de Goldberg e Tarjan [20] venha a ter um papel equivalente ao algoritmo de Dinic, o qual serviu de base para vários algoritmos mais eficientes. Explorando a idéia de Goldberg e Tarjan novos algoritmos tem sido desenvolvidos (Ahuja e Orlin [1], Ahuja, Orlin e Tarjan [2]), embora tais algoritmos não sejam fortemente polinomiais. A estrutura de dados utilizada por Goldberg e Tarjan tem papel importante no decréscimo da complexidade do algoritmo, embora sua manutenção seja um tanto cara em tempo computacional.

Com a utilização desta estrutura de dados pode-se resolver de forma eficiente, diversos problemas em teoria dos grafos tais como:

1. Encontrar o ancestral comum a dois nós contidos numa determinada árvore.

2. Construção de certos tipos de árvores geradoras mínimas.

3. Implementação do algoritmo Simplex, especializado para redes, a fim de encontrar o fluxo de custo mínimo.

Esta estrutura de dados pode ser utilizada onde há necessidade de armazenar e manipular as informações de um caminho, causando um decréscimo da ordem de $\log(n^2/m)$ na complexidade do algoritmo em questão.

Uma estrutura de dados de tipo similar às árvores dinâmicas é utilizada para armazenar filas de prioridades, chamadas de **pilhas binárias de Fibonacci** [13]. Esta estrutura é uma extensão do conceito de **filas binomiais** propostas por Vuillemin [35].

Utilizando esta estrutura, Fredman e Tarjan [13] melhoraram o desempenho do algoritmo de Dijkstra para o problema do caminho mínimo numa rede com arestas de comprimento não negativo. Como vários algoritmos de redes utilizam o algoritmo de Dijkstra como sub-rotina, para cada um destes, uma melhoria foi alcançada.

Sleator e Tarjan propoem uma estrutura de dados do tipo **árvores binárias auto ajustantes** [33] para se obter um algoritmo ainda mais eficiente na resolução do problema do fluxo máximo, embora tal idéia não tenha sido explorada por estes autores.

1.3 Objetivos do Trabalho

Com respeito aos objetivos deste trabalho de dissertação de mestrado podemos citar:

1. Realização do levantamento bibliográfico dos algoritmos para o problema do fluxo máximo.

2. Implementação do algoritmo de Goldberg e Tarjan com a devida estrutura de dados.

3. Implementação dos principais algoritmos para o problema do fluxo máximo tais como Edmonds e Karp [10], Malhotra et alli [25] e Tarjan [34].

4. Realização de testes comparativos entre os algoritmos implementados e outros algoritmos para o problema.

Capítulo 2

O Algoritmo de Goldberg e Tarjan

2.1 Notação e Definições

Um par de conjuntos $G = (V, E)$ é um **grafo** se $V = \{1, 2, ..., n\}$ é um conjunto finito não vazio de n **nós** e $E = \{e_1, e_2, ..., e_m\}$ é um conjunto finito não vazio de m **arcos** (pares ordenados de nós distintos). Se (i, j) é um arco então (i, j) **sai** de i, (i, j) **chega** em j, i é a **cauda** de (i, j) e j é a **cabeça** de (i, j).

Um grafo $G' = (V', E')$ é um **subgrafo** do grafo $G = (V, E)$ se $G' \subset G$, $V' \subset V$ e $E' \subset E$.

Uma seqüência alternada $P = (v_0, e_1, v_1, e_2, ..., e_k, v_k)$ de nós distintos $v_0, v_1, ..., v_k$ e de arcos $e_1, e_2, ..., e_k$, onde cada arco e_k é da forma $e_k = (v_{k-1}, v_k)$, é um **caminho** de v_0 para v_k e v_0, v_k são ditos **conectados** por P. O **comprimento** de um caminho é igual ao número de arcos que o constitui. Um **caminho mínimo** de s para t é um caminho de menor comprimento entre s e t.

Considere uma rede orientada $N = (V, E, s, t, c)$ definida por um grafo $G = (V, E)$ com dois nós distintos - um nó **origem s** e um nó **destino t** - e uma **função capacidade c** expressa por $c : V \times V \rightarrow R$ onde $c(v, w) = 0$ se $(v, w) \notin E$.

Um **fluxo f** é uma função $f : V \times V \rightarrow R$ satisfazendo:

1. $-c(w,v) \leq f(v,w) \leq c(v,w)$

2. $f(v,w) = -f(w,v)$

3. $\sum_w f(v,w) = \sum_{w \in V} f(v,w) = 0,\ v \neq s,t$

Observe que $f(w,w) = 0\ \forall\ w$ e $\sum_w f(v,w) = -\sum_w f(w,v)\ \forall\ v$.

O **valor** $|\ f\ |$ de um fluxo f é a quantidade de fluxo que sai da origem (menos a quantidade de fluxo que chega) ou o que chega no destino (menos o que sai).

$$|\ f\ | = \sum_w f(s,w) = \sum_w f(w,t)$$

Um **fluxo máximo** é um fluxo de valor máximo.

Um **corte** $(S,\ \bar{S})$ é uma partição do conjunto de nós ($S \cup \bar{S} = V,\ S \cap \bar{S} = \emptyset$) com $s \in S, t \in \bar{S}$.

A **capacidade** de um corte é dado por :

$$c(S,\bar{S}) = \sum [c(v,w)|\ v \in S, w \in \bar{S}]$$

E o **valor do corte** é dado por:

$$f(S,\bar{S}) = \sum [f(v,w)|\ v \in S, w \in \bar{S}] = -\sum [f(w,v)|\ v \in S,\ w \in \bar{S}]$$

Um **corte mínimo** é um corte de capacidade mínima.

Lema 2.1: Qualquer fluxo f de s para t numa rede $N = (V,E,s,t,c)$ tem valor menor ou igual à capacidade de qualquer corte da rede [11].

Prova: Seja f um fluxo e $(S,\bar{S})$ um corte da rede. Como f é um fluxo, temos que

$$\sum_w f(v,w) = |\ f\ |,\ v = s$$

$$\sum_w f(v,w) = 0,\ \forall\ v \neq s,t$$

$$\sum_w f(v,w) = -\ |\ f\ |,\ v = t$$

Somando estas equações sobre $v \in S$, como $s \in S$ e $t \in \bar{S}$, o resultado é

$$|\ f\ | = \sum_{v \in S} f(v,V) = f(S,V)$$

Escrevendo $V = S \cup \bar{S}$ temos:

$$| f | = f(S, S \cup \bar{S})$$

$$| f | = f(S, S) + f(S, \bar{S})$$

$$| f | = f(S, \bar{S}) \quad \text{pois} \quad f(S, S) = 0.$$

Pela definição de fluxo temos que $f(S, \bar{S}) \leq c(S, \bar{S})$ logo

$$| f | = f(S, \bar{S}) \leq c(S, \bar{S}) \quad \square$$

Teorema 2.2 (Fluxo Máximo-Corte Mínimo) Para qualquer rede o valor do fluxo máximo de s para t é igual à capacidade do corte mínimo [11].

Prova: Pelo Lema 2.1 é suficiente mostrar a existência de um fluxo f e um corte $(S, \bar{S})$ para o qual a igualdade entre o valor do fluxo e a capacidade do corte se verifique. Seja f um fluxo máximo (claramente ele existe) pois todos os arcos possuem capacidade finita. Usando f definimos o conjunto S recursivamente da seguinte maneira

1. $s \in S$,
2. se $v \in S$ e $f(v, w) < c(v, w)$ então $w \in S$,
3. se $v \in S$ e $f(w, v) > 0$ então $w \in S$.

Afirmação: $t \in \bar{S}$. Suponha por contradição que $t \in S$. Então da definição de S existe um caminho de s para t, digamos

$$s = v_1, e_1, v_2, e_2, ..., e_{n-1}, v_n = t$$

com a propriedade de que para cada arco no sentido direto (v_i, v_{i+1}) do caminho, temos que

$$f(v_i, v_{i+1}) < c(v_i, v_{i+1})$$

enquanto para cada arco no sentido inverso (v_{i+1}, v_i) do caminho,

$$f(v_i, v_{i+1}) > 0$$

Seja ϵ_1 o mínimo de $c - f$ tomado sobre todos os arcos no sentido direto do caminho e ϵ_2 o mínimo de f sobre todos os arcos no sentido inverso do caminho, e seja $\epsilon = \min\{\epsilon_1, \epsilon_2\} > 0$. Incrementando o fluxo em todos os arcos do caminho no sentido direto em ϵ unidades e decrementando o fluxo em todos os arcos do caminho no sentido inverso também em ϵ unidades conseguimos aumentar o fluxo em ϵ unidades o que é uma contradição, portanto $t \in \bar{S}$. Conseqüentemente $(S, \bar{S})$ é um corte. Da definição temos que:

$$f(v, \bar{v}) = c(v, \bar{v}) \quad \text{para } (v, \bar{v}) \in (S, \bar{S})$$

$$f(\bar{v}, v) = 0 \quad \text{para } (\bar{v}, v) \in (\bar{S}, S)$$

Assim temos que $f(S, \bar{S}) = c(\bar{S}, S)$, $c(S, \bar{S}) = 0$ e portanto

$$| f | = f(S, \bar{S}) - f(\bar{S}, S) = c(S, \bar{S}) \quad \square$$

Um **prefluxo g** é uma função $g : V \times V \rightarrow R$ satisfazendo:

1. $-c(w, v) \leq g(v, w) \leq c(v, w)$
2. $g(v, w) = -g(w, v)$
3. $\sum_w g(v, w) \geq 0, \ v \neq s, \ t.$

Em um prefluxo temos que $\sum_w g(w, v) \geq 0$ e $g(v, v) = 0$.

Define-se o **excesso de fluxo** no nó v por:

$$e(v) = \begin{cases} \infty & \text{se } v = s \\ \sum_w g(w, v) & \text{caso contrário} \end{cases}$$

Com esta definição, um fluxo é um prefluxo com $e(v) = 0 \ \forall \ v \neq s, t$.

Dado um prefluxo g, a **rede residual** $\bar{N} = (V, \bar{E}, s, t, \bar{c})$ da rede N é definida pela função **capacidade residual** $\bar{c}$ dada por:

$$\bar{c} : V \times V \rightarrow R \quad \text{onde } \bar{c}(v, w) = c(v, w) - g(v, w)$$

e pelo conjunto de **arcos residuais** $\bar{E}$ expresso por:

$$\bar{E} = \{(v,w) \in V \times V \mid \bar{c}(v,w) > 0\}$$

Um **rotulamento válido d** para um prefluxo g é uma função

$$d : V \rightarrow N \text{ tal que}$$

1. $d(t) = 0$,
2. $d(v) > 0 \; \forall \; v \neq t$,
3. $d(w) \geq d(v) - 1 \; \forall \; (v,w) \in \bar{N}$.

Um rotulamento válido $d(v)$ nada mais é do que um limitante inferior para a distância entre v e t em N; isto é, todo caminho de s para t contém pelo menos $d(v)$ arcos.

Um nó v é **ativo** se:

1. $v \neq s$ e t,
2. $d(v) < \infty$ e $e(v) > 0$.

2.2 Estrutura Básica do Algoritmo Seqüencial de Goldberg e Tarjan

Este algoritmo resolve o problema de fluxo máximo mantendo um prefluxo g e um rotulamento válido d para g. Grosseiramente falando, o algoritmo examina os nós com excesso de fluxo positivo, enviando fluxo deste nó para nós considerados mais próximos de t na rede residual. Como estimativa da distância, o algoritmo usa o rotulamento d, o qual é periódicamente atualizado para melhor refletir o grafo residual.

O algoritmo consiste de dois estágios. Durante o primeiro estágio, unidades de fluxo são enviadas em direção ao destino. Esta rotina é chamada de Push . O primeiro estágio termina com a determinação de um corte mínimo e portanto o valor do fluxo máximo. No segundo estágio, o excesso de fluxo retido nos nós intermediários, se houver, é retornado à origem

transformando o prefluxo (máximo) em um fluxo máximo. Esta rotina é denominada Reduce e a rotina que atualiza periódicamente o rotulamento é denominada Relabel.

O algoritmo do fluxo máximo começa com o prefluxo que é igual à capacidade em cada arco que sai da origem e é zero nos outros arcos. Um rotulamento válido inicial dado por $d(v) = 1 \ \forall \ v \neq \{s,t\}, d(t) = 0$ e $d(s) = n$ permite que o algoritmo trabalhe em um único estágio. Neste rotulamento, se $d(v) < n$ então $d(v)$ é um limitante inferior para a distância atual de v até t na rede residual, e se $d(v) \geq n$, então $d(v) - n$ é um limitante inferior da distância de v até s na rede residual. No segundo caso t não é alcançável a partir de v em $\bar{N}$.

A idéia do algoritmo é executar repetidamente as operações Push e Relabel, em qualquer ordem, até que não mais existam nós ativos.

As operações Push e Relabel são denominadas **operações básicas** por serem utilizadas nas demais operações e são implementadas da seguinte maneira:

Push(v,w)
Aplicabilidade: v é um vértice ativo, $\bar{c}(v,w) > 0$ e $d(v) = d(w) + 1$.
Ação: Envie $\delta = \min \{ e(v), \bar{c}(v,w) \}$ unidades de fluxo de v para w da seguinte forma:
$g(v,w) \leftarrow g(v,w) + \delta$; $g(w,v) \leftarrow g(w,v) - \delta$;
$e(v) \leftarrow e(v) - \delta$; $e(w) \leftarrow e(w) + \delta$.

Figura 2.1: Operação básica de envio de fluxo.

Obs: A operação Push faz com que $g(v,w)$ e $e(w)$ aumentem em δ unidades e $g(w,v)$ e $e(v)$ diminuam em δ unidades.

Relabel(v)
Aplicabilidade: v é um nó ativo e $\forall \ w \in V$, $c(v,w) > 0$ e $d(v) < d(w)$.
Ação: $d(v) \leftarrow \min\{d(w) + 1 \mid (v,w) \in \bar{E}\}$. Se este mínimo for sobre um conjunto vazio então $d(v) \leftarrow \infty$.

Figura 2.2: Operação básica que atualiza o rotulamento.

2.3 O Algoritmo Seqüencial de Goldberg e Tarjan.

Nós agora discutiremos o algoritmo de Goldberg e Tarjan com complexidade de $O(n^3)$. Este algoritmo, que chamaremos de **algoritmo seqüencial** necessita de algumas estruturas de dados que representem a rede de trabalho e o prefluxo. A cada arco (v,w) associamos dois valores, $c(v,w)$ e $g(v,w)$ $(= -g(w,v))$. É associado a cada nó v uma lista dos arcos incidentes a ele, colocados numa ordem arbitrária mas fixa. Todo nó v tem um **arco corrente** (v,w) que é o candidato do momento através do qual tenta-se aplicar uma operação Push a partir de v. Inicialmente o arco corrente de v é o primeiro arco da lista de v.

A iteração principal do algoritmo seqüencial consiste na repetição da operação Push/Relabel descrita na figura 2.3, até que não exista qualquer nó ativo. A operação Push/Relabel é aplicada a um nó ativo v e tenta enviar excesso de fluxo através do arco corrente (v,w) de v. A operação Push/Relabel pode não ser aplicável ao arco corrente (v,w), neste caso, a operação troca o arco corrente pelo próximo arco da lista de v. No caso de (v,w) ser o último arco da lista, o arco corrente volta a ser o primeiro arco da lista e aplica-se uma operação Relabel ao nó v.

Push/Relabel(v)
Aplicabilidade: v é um nó ativo.
Ação: Seja (v,w) o arco corrente de v.
If *Push(v,w)* é aplicável **Then** *Push(v,w)*
Else If (v,w) não é o último arco da lista de v
Then troque (v,w), o arco corrente de v pelo próximo arco incidente da lista de v.
Else faça do primeiro arco da lista de v o arco corrente e aplique *Relabel(v)*.

Figura 2.3: Operação composta pelas operações básicas.

O algoritmo seqüencial necessita ainda de uma estrutura de dados adicional, um conjunto Q contendo todos os nós ativos. Inicialmente $Q = \{v \neq s,t \mid c(s,v) > 0\}$. A manutenção de Q gasta um tempo de $O(1)$

Discharge(v)
Aplicabilidade: $Q \neq \emptyset$.
Ação: Remove o nó v da frente de Q (v deve ser ativo).
Repeat
    Push/Relabel(v);
    If w torna-se ativo durante este Push/Relabel
    Then inclua w no final de Q;
Until $e(v) = 0$ **or** $d(v)$ aumente.
If v permanecer ativo **Then** inclua v no final de Q

Figura 2.4: Operação utilizada no algoritmo seqüencial.

por operação Push/Relabel. Esta operação aplicada a um arco (v, w) pode adicionar w a Q e/ou retirar v de Q.

Para reduzir o número de operações Push não saturantes (operações Push onde $\delta < \bar{c}$), explora-se a liberdade existente ao selecionar o vértice ativo no qual será aplicado a operação Push/Relabel; neste caso é usada a estratégia First-in First-out. Isto significa que Q é mantida como uma **fila**.

A operação Discharge(v) descrita na figura 2.4, consiste em aplicar operações Push/Relabel a este nó v pelo menos até que o excesso deste nó torne-se zero ou o seu rotulamento aumente. Caso algum nó (adjacente a v) fique ativo durante estas operações, adiciona-se este nó ao final de Q. Se v permanecer ativo após a iteração, ele será incluífinal de Q.

O algoritmo seqüencial consiste em aplicar operações Discharge até que a fila fique vazia. A estrutura principal do algoritmo está na figura 2.5.

Goldberg e Tarjan [20] mostraram que durante a execução deste algoritmo, o rotulamento permanece finito e válido para cada nó da rede. Este resultado é assegurado encontrando-se um limitante de $2n(n-1)$ para o número de Push saturantes, $2n$ para o número de operações Relabel e $4n^3$ operações Push não saturantes, com isto eles mostraram que o algoritmo seqüencial tem complexidade $O(n^3)$.

Neste algoritmo pode-se transformar a fila Q de nós ativos em uma fila de prioridades segundo o excesso de fluxo existente em cada nó, ou então priorizando-se os nós com maior rótulo. Estas são duas variações do algoritmo seqüencial.

Em várias aplicações [28] nas quais surge o problema do fluxo máximo, o que se procura é o valor do fluxo máximo ou o corte mínimo, e não o

$<<$*Programa Seqüencial*$>>$
$<$*Inicialização do Prefluxo*$>$
$\forall\ (v,w) \in\ (V - \{s\}) \times (V - \{s\})$ **Do** $g(v,w) \leftarrow 0$;
$\forall v \in V$ **Do Begin** $g(s,v) \leftarrow c(s,v)$;
$e(v) \leftarrow c(s,v)$
Inclua v à lista Q **End**;
$<$*Inicialização do Rotulamento e Excesso*$>$
$d(s) \leftarrow n;\ d(t) \leftarrow 0$;
$\forall v \in V - \{s,t\}$ **Do Begin** $d(v) \leftarrow 1;\ e(v) \leftarrow 0$ **End**;
$<$*Iteração Principal*$>$
Begin
While $Q \neq \emptyset$ **Do Begin**
Retire o primeiro nó de Q, suponha v;
Aplique *Discharge*(v);
End;
End.

Figura 2.5: Esquema do algoritmo seqüencial de Goldberg e Tarjan.

fluxo máximo em si. O algoritmo [20] pode ser adaptado para tal situação calculando apenas um corte mínimo e o valor do fluxo máximo. Para tanto basta redefinir um nó ativo da seguinte forma: $v \in V - \{s,t\}$ tal que $e(v) > 0$ e $d(v) < n$. Com esta modificação, quando o algoritmo termina, o excesso do destino $e(t)$ é o valor do fluxo máximo e o corte mínimo é dado por $(S, \bar{S})$ onde $\bar{S}$ contém exatamente os nós a partir dos quais t é alcançável em $\bar{N}$.

Muitos autores se referem apenas à implementação seqüencial ao estudarem o algoritmo de Goldberg e Tarjan. Este algoritmo também é utilizado na implementação com processamento em paralelo. Anderson e Setubal [3] estão atualmente trabalhando em uma implementação deste tipo.

2.4 O Algoritmo de Goldberg e Tarjan com a Estrutura de Dados Árvores Dinâmicas

Para reduzir a complexidade $O(n^3)$, Goldberg e Tarjan reduziram ainda

mais o número de operações Push não saturantes. Para alcançar tal resultado utilizaram a estrutura de dados Tipo Árvores Dinâmicas proposta por Sleator e Tarjan [32].

Faremos inicialmente uma descrição da estrutura de dados Árvores Dinâmicas e de seu funcionamento. Completamos a apresentação introduzindo a estrutura de dados ao algoritmo descrito na seção anterior.

2.4.1 A Estrutura de Dados Tipo Árvore Dinâmica

Esta estrutura nos permite manter as informações de um conjunto de árvores enraizadas nós-disjuntas na qual cada nó v tem um valor real associado $h(v)$, submetido a uma seqüência qualquer de operações de árvore, que são utilizadas pelo algoritmo. Adota-se a convenção de que todo nó é um ancestral e um descendente de si mesmo. As operações de árvores utilizadas são:

find-root(v): Encontra e retorna a raiz da árvore contendo o nó v.

find-size(v): Encontra e retorna o número de nós na árvore contendo o nó v.

find-value(v): Calcula e retorna o valor de $h(v)$.

find-min(v): Encontra e retorna o ancestral de v com valor de $h(v)$ mínimo. Em caso de empate retorna o nó mais próximo da raiz.

change-value(v, x): Soma o valor real x a $h(w)$ para cada ancestral w de v. Convencionamos que $\infty - \infty = 0$.

link(v, w, x): Liga as duas árvores contendo nós v e w tornando w o pai de v, através do arco (v, w) com capacidade x. Esta operação nada faz se v e w estão na mesma árvore ou se v não é um nó raiz.

cut(v): Quebra a árvore contendo v em duas árvores retirando o arco de v para seu pai. Esta operação não faz nada se v é raiz de alguma árvore.

As operações *find-root*, *find-size*, *find-value* e *find-min* apenas extraem informações das árvores sem alterar suas estruturas. A operação *change-value* altera apenas a capacidade em alguns arcos, mantendo a estrutura da árvore. As operações *link* e *cut* é que alteram a estrutura da árvore.

A figura 2.6 apresenta exemplos da execução destas operações de árvores dinâmicas.

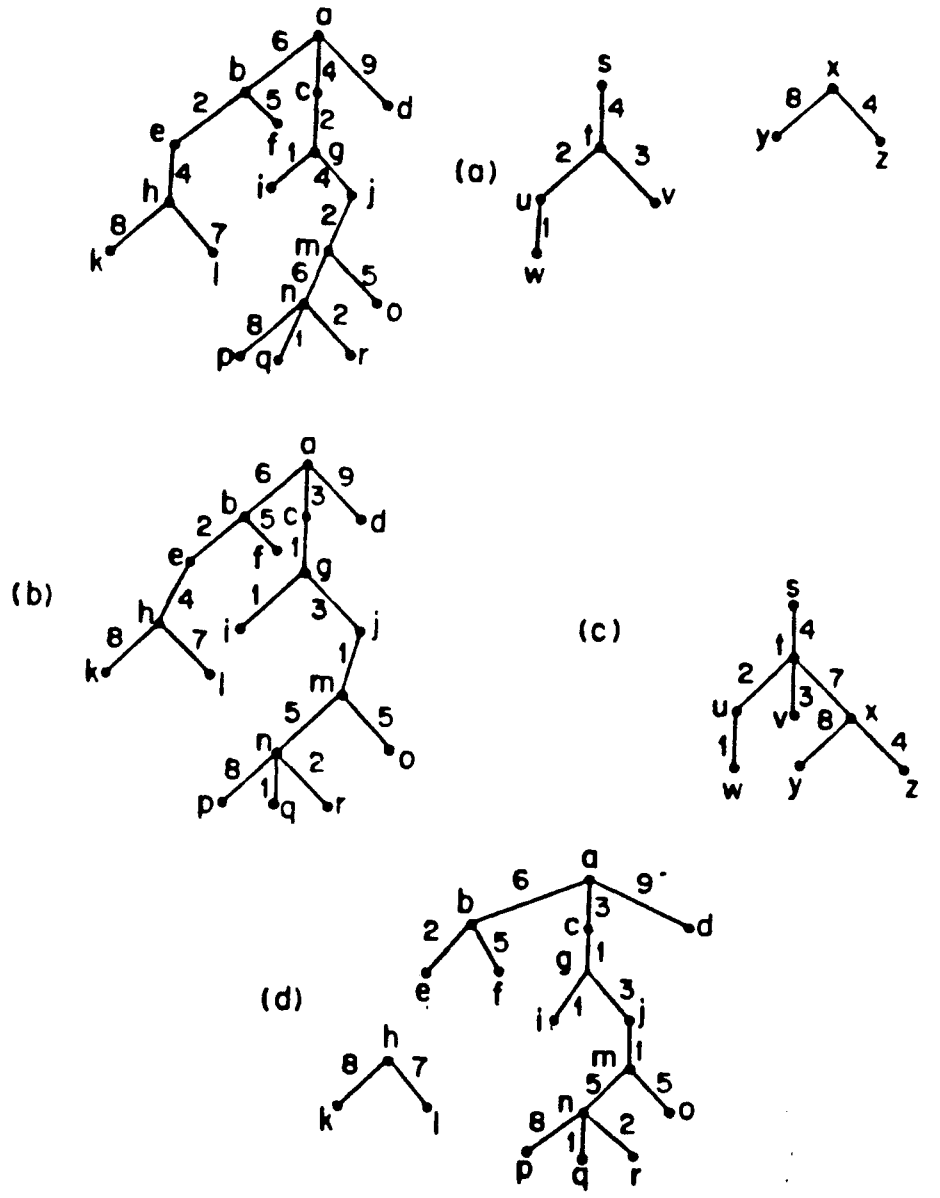

Figura 2.6: Operações de árvores dinâmicas. (a) Três árvores. Operação $parent(n)$ retorna m, $root(n)$ retorna a, $find-value(n)$ retorna 6, $find-min(n)$ retorna g. (b) Árvore contendo n após $change-value(n,-1)$. (c) Árvore formada após $link(x,t,7)$. (d) Árvores formadas após $cut(h)$ aplicada na árvore do item (b). Retorna o valor 4.

A implementação destas operações de árvore é feita através de uma combinação de **operações de caminho**. Os caminhos são gerados ao particionar-mos cada árvore da floresta. Sleator e Tarjan [32] sugerem dois tipos diferentes de partição: Partição Simples e Partição por Tamanho.

Nesta implementação foi utilizada a partição simples pois sua descrição é mais completa e seus resultados teóricos são suficientes para se garantir a complexidade do algoritmo. A partição simples tem complexidade média de $O(\log n)$ por operação sobre uma árvore com n nós. As operações de caminho utilizadas são:

path(v): Retorna o caminho contendo o nó v.

head(p): Retorna o nó cabeça do caminho p.

tail(p): Retorna o nó cauda do caminho p.

before(v): Retorna o nó antes de v em $path(v)$. Se v é a cabeça do caminho retorna nulo.

after(v): Retorna o nó depois de v em $path(v)$. Se v é a cauda do caminho retorna nulo.

pcap(v): Retorna a capacidade do arco $(v, after(v))$. Esta operação assume que v não é a cauda de $path(v)$.

pmincost(p): Retorna o nó v mais próximo de $tail(p)$ tal que o arco $(v, after(v))$ tem capacidade mínima dentre os arcos de p.

pupdate($p; x$): Adiciona o valor x à capacidade de todos os arcos em p.

concatenate($p, q; x$): Combina os caminhos p e q adicionando o arco $(tail(p), head(q))$ com capacidade x. Retorna o caminho combinado.

split(v): Divide o caminho $path(v)$ em até três partes deletando os arcos incidentes a v. Retorna a lista $|p,\ q,\ x,\ y|$, onde p é o sub-caminho consistido pelos nós de $head(path(v))$ até $before(v)$, q é o sub-caminho de todos os nós desde $after(v)$ até $tail(path(v))$, x é a capacidade do arco deletado $(before(v), v)$ e y é a capacidade de $(v, after(v))$. Se v for a cabeça de $path(v)$, p é nulo e x é indefinido; se v for a cauda de $path(v)$, q é nulo e y é indefinido.

Na partição simples, os caminhos são determinados não pela estrutura da árvore mas pela seqüência das operações de árvore executadas. Nesta partição os arcos de cada árvore são dos tipos **sólidos** e **tracejados**, com

a seguinte propriedade: no máximo um arco sólido chega em qualquer nó. Assim os arcos sólidos definem uma coleção de **caminhos sólidos** que particionam os nós. Um nó sem arco sólido incidente é um caminho sólido um-nó. A cabeça do caminho é o nó mais "baixo" e a cauda é o nó mais "alto" na árvore.

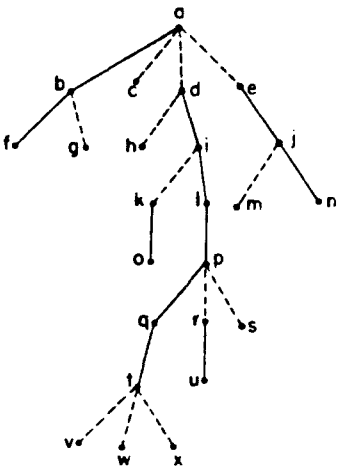

Figura 2.7: Árvore particionada em caminhos sólidos. O caminho $[t, q, p, l, i, d]$ tem cabeça t e cauda d.

As informações dos arcos tracejados são armazenadas da seguinte maneira: com cada nó v que é a cauda de um caminho sólido, nós armazenamos $dparent(v)$, o pai de v via arco tracejado e $dcost(v)$, a capacidade do arco $(v, parent(v))$. Se v é a raiz de uma árvore então $dparent(v) =$ nulo e $dcost(v)$ é indefinido. Manipulam-se os caminhos sólidos utilizando as operações de caminho definidas anteriormente. Adicionalmente são necessárias duas operações que são composições de operação de caminho. Estas operações são:

1. $Splice(p)$: Transforma o arco que sai da cauda de p em direção à raiz da árvore, caso exista, em um arco sólido.

2. $Expose(v)$: Transforma em arcos sólidos aqueles que compoêm o caminho que vai do nó v até à raiz da árvore que o contém.

Estas operações são implementadas da seguinte forma:

Function Splice(p)
nó v; caminho p, q; *real* x;
1. $[q, r, x, y]$:= $split(v)$;
   If $q \neq$ null **Then Begin** v := $dparent(tail(q))$;
   x := $dcost(tail(q))$; **End**;
2. p := $concatenate(p, path(v), dcost(tail(p)))$;
3. **If** r − null **Then** *splice* := p **Else** *splice* := $concatenate(p, q, y)$;

Figura 2.8: Implementação da operação composta Splice.

Obs. A linha 1 do *Splice* transforma em tracejado os arcos sólidos incidentes a $v = parent(tail(p))$ se houver tal arco. A linha 2 transforma em sólido o arco tracejado que sai de $tail(p)$. A linha 3 retorna o caminho anterior que acrescido de um arco sólido à cauda pode aumentar o caminho sólido em mais de um arco.

Function Expose(v);
caminho p, q; real x, y;
1. $[q, r, x, y]$:= $split(v)$;
   If q <> null
   Then Begin v := $dparent(tail(q))$;
   x := $dcost(tail(q))$; **End**
2. **If** r = null **Then** p := $path(v)$ **Else** p := $concatenate(path(v), r, x)$;
3. **While** $dparent(tail(p))$ <> null
   Do p := $splice(p)$;
   expose := p;

Figura 2.9: Implementação da operação composta Expose.

Obs. A linha 1 transforma em tracejado os arcos sólidos incidentes a v, se houver. Na linha 2 retorna a sólido o arco que sai de v se ele acabou de ser transformado em tracejado. A linha 3 é a iteração principal que extende o caminho contendo v, através de sucessivas operações Splice até que a cauda do caminho seja a raiz da árvore. A seguir, na figura 2.10, ilustramos o efeito das operações Splice e Expose sobre uma árvore particionada qualquer.

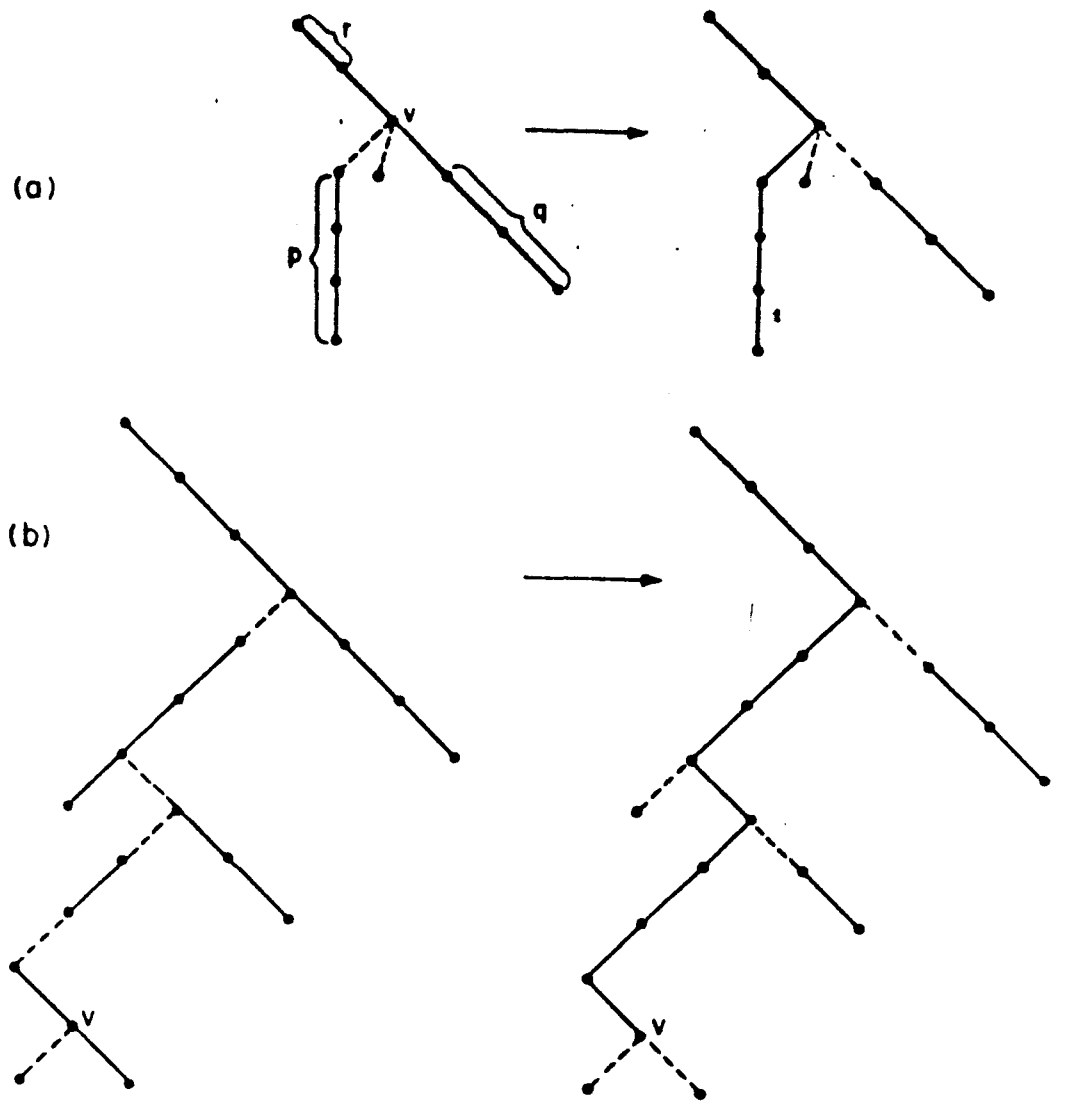

Figura 2.10: Operações Splice e Expose. (a) Efeito de splice(p). As letras q, r, e v referem-se às variáveis do programa. (b) Efeito de expose(v).

Considerando já implementadas as operações de árvore, efetua-se as operações de caminho da seguinte maneira:

Function $Parent(v)$;
If $v = tail(path(v))$ **Then** $parent := dparent(v)$
Else $parent := after(v)$;
Return $parent$;

Function $Find - root(v)$;
$root := tail(expose(v))$;
Return $root$;

Function $Size(v)$;
$size := size(find - root(v))$;
Return $size$;

Function $Find - value(v)$;
If $v = tail(parent(v))$ **Then** $cost := dcost(v)$
Else $cost := pcost(v)$
Return $cost$;

Function $Find - min(v)$;
$mincost := pmincost(expose(v))$;
Return $mincost$;

Procedure $Change - value(v, x)$;
$pupdate(expose(v))$;

Procedure $Link(v, w; x)$;
$concatenate(path(v), expose(w), x))$;

Function $Cut(v)$;
$caminho$: p, q; **real** x, y;
$expose(v)$;
$[p, q, x, y] := split(v)$;
$dparent(v)$:= null;
$cut := y$;
Return cut;

Para completar a descrição das árvores dinâmicas passamos a detalhar a estrutura de dados que representa estes caminhos dinâmicos e também como implementar as operações de caminho citadas anteriormente.

Cada caminho é representado por uma **árvore binária** cujos elementos são denominados **vértices** e **arestas**. Neta estrutura os **vértices externos** (folhas) correspondem aos **nós do caminho**. A orientação da esquerda para a direita na árvore binária corresponde à orientação da cabeça para a cauda e os **vértices internos** correspondem aos **arcos** do caminho da seguinte forma: o arco interno correspondente ao arco (v_1, v_2) é o primeiro ancestral comum a v_1 e v_2. Todo vértice da estrutura de dados corresponde a um subcaminho compreendido entre os seus descendentes extremos, conforme a orientação. Com isto a raiz da árvore binária corresponde a um caminho. Sempre que nos referimos a um caminho durante a descrição da implementação estamos falando da raiz de sua árvore binária.

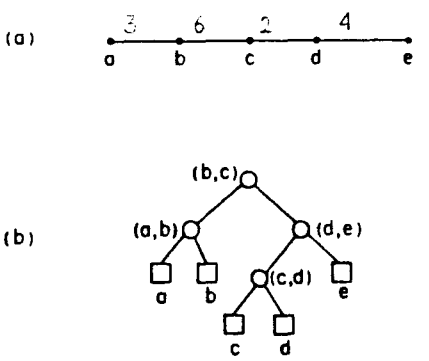

Figura 2.11: Representação de um caminho através de uma árvore binária.

Várias informações a respeito do caminho são armazenadas em cada vértice da árvore binária. Um vértice desta árvore pode ser externo (folha) ou interno (galho). Quando um vértice não tem filho este vértice é externo, caso contrário este vértice será interno. Todo vértice v tem um apontador $pai(v)$ indicando o seu pai na árvore binária; se v for a raiz da árvore binária então $pai(v) = \text{null}$.

Cada vértice interno v contém os seguintes apontadores: $esq(v)$ e $dir(v)$

que indicam os **filhos à direita** e **filho à esquerda** de v, $cab(v)$ e $cau(v)$ indicando a **cabeça** e a **cauda** do subcaminho correspondente ao vértice v (o descendente externo de v mais à esquerda e mais à direita). Além destes apontadores, os vértices internos contém informações sobre a capacidade do arco correspondente no caminho. Foi armazenada em $grc(v)$ a **capacidade residual do arco** e em $grm(v)$ a **capacidade mínima no subcaminho** correspondente a v.

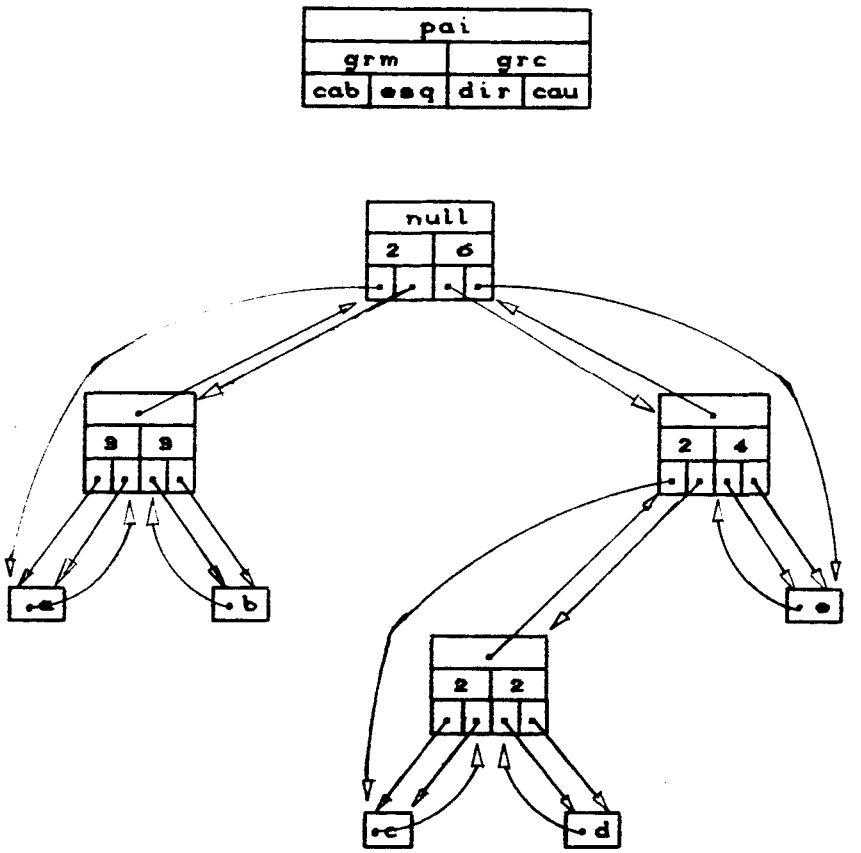

Figura 2.12: Detalhes da representação do caminho na figura 2.11 em árvore binária.

Representando os caminhos através destas árvores binárias podemos implementar as operações de caminho da seguinte forma:

path(v): Seguir o apontador $bparent(v)$ até encontrar um nó w com $bparent(w)$ = null. Retorna o nó w.

head(p): Retorna $cab(p)$.

tail(p): Retorna $cau(p)$.

before(v): Se o nó v for o filho à direita, retorna a cauda do filho à esquerda de $parent(v)$, caso contrário, seguir o apontador $bparent(v)$ até encontrar um nó w tal que $w = dir(parent(w))$, então retorna $cau(esq(parent(w)))$.

after(v): Simétrico à operação $before(v)$.

pcap(v): Se o nó v for o filho à direita, retorna $grm(parent(v))$, caso contrário, seguir o apontador $bparent(v)$ até um nó w tal que $w = dir(parent(w))$, então retorna $grm(parent(w))$.

pmincost(p): Partindo de p, que é a raiz da árvore binária, descer em direção ao filho w tal que $grm(w) = grm(p)$. Em caso de empate escolher o filho à direita e quando $grm(filho\ da\ direita)$ e $grm(filho\ da\ esquerda)$ forem diferentes de $grm(p)$, retorna $cab(filho\ da\ direita)$ do nó w onde estiver.

pupdate(p, x): Percorre os vértices internos da árvore binária somando x a grm e grc de cada vértice visitado.

Para implementar as operações *concatenate* e *split*, são utilizadas as seguintes operações adicionais:

construct(v, w; x): Dadas as raízes v, w e o real x, combina as duas árvores simples criando um novo nó raiz com filho à esquerda v, filho à direita w e grc x.

destroy(u): Dada a raiz u de uma árvore binária, divide esta árvore em duas, uma cuja raiz é o filho à esquerda, digamos v e a outra cuja raiz é o filho à direita, digamos w. Retorna a lista $[v, w, x]$, onde x é a capacidade do arco retirado u.

raise(v): Dado um vértice v interno da árvore binária, "levanta" este vértice e o transforma na raiz da árvore binária.

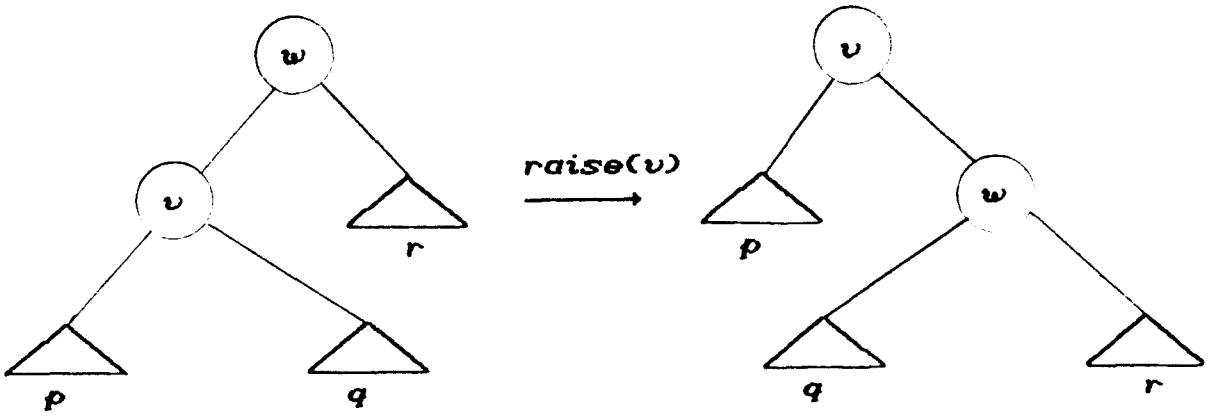

Figura 2.13: Efeito da operação *raise*(v).

Com o auxílio destas operações adicionais, implementamos as operações *concatenate* e *split* restantes, concluindo a descrição da estrutura de dados árvore dinâmica.

Esta estrutura que acabamos de descrever armazena as informações de uma coleção de árvores nó-disjuntas e Sleator e Tarjan [32] mostram que a execução de uma operação árvore dinâmica tem complexidade $O(\log n)$ quando aplicada a uma árvore com n nós.

2.4.2 O Uso das Árvore Dinâmica no Algoritmo de Goldberg e Tarjan

Nesta aplicação, os arcos das árvores dinâmicas formam um subconjunto dos arcos correntes dos nós. O arco corrente (v, w) de um nó $v \in V - \{s, t\}$ é **elegível** para ser um arco árvore dinâmica (com $p(v) = w$) se $d(v) = d(w) + 1$ e $\bar{c}(v, w) > 0$, mas nem todos os arcos elegíveis são arcos árvore dinâmica. O valor $h(v)$ de um nó v na sua árvore dinâmica representa o valor de $\bar{c}(v, w)$ se v tiver pai w e ∞ se v for raiz de uma árvore. Limita-se o comprimento máximo das árvores em k, onde este parâmetro será escolhido posteriormente.

Utilizando apropriadamente as operações de árvore, pode-se enviar fluxo ao longo de um caminho inteiro em uma árvore, causando um Push satu-

rante ou então movendo excesso de fluxo de algum nó na árvore até sua raiz. Combinando esta idéia com uma análise cuidadosa mostra-se que o número de vezes que um nó é adicionado à lista Q de nós ativos é $O(nm + n^3/k)$. Com um custo de $O(\log k)$ para cada operação de árvore, o tempo total de execução do algoritmo é de $O((nm + n/k)\log k)$, o qual é minimizado para um fator constante de $O(nm\log(n^2/m))$ ao escolher $k = n^2/m$. Isto implica que cada árvore na rede residual tem seu tamanho limitado por k nós.

Os detalhes deste algoritmo, o qual chamaremos de algoritmo Árvore Dinâmica são os seguintes. A parte central do algoritmo é o procedimento Send(v), definido por:

$Send(v)$
Aplicabilidade: v é ativo.
Ação: **While** $find - root(v) <> v$ **And** $e(v) > 0$
Do Begin envie $\delta = \min\{e(v), find - value(find - min(v))\}$ unidades de fluxo ao longo da árvore caminho de v executando $change - value(v, -\delta)$;
While $find - value(find - min(v)) = 0$
Do Begin $u \leftarrow find - min(v)$;
execute $cut(u)$;
$change - value(u, \infty)$; **End;**
End.

Figura 2.14: Operação que envia fluxo através de uma árvore dinâmica.

Este procedimento envia excesso de fluxo do nó v não raiz para a raiz de sua árvore, corta os arcos saturados pelo envio, e repete este passo até que $e(v) = 0$ ou v seja raiz de uma árvore.

Superficialmente, o algoritmo Árvore Dinâmica é exatamente o mesmo que o algoritmo seqüencial apresentado em seção anterior. Mantem-se uma fila Q de nós ativos e repetidamente executamos operações Discharge até que Q fique vazia. A diferença está na operação Push/Relabel que será substituida pela operação Tree-Push/Relabel

$Tree - Push/Relabel(v)$.

Aplicabilidade: v é um nó raiz ativo.

Ação: Seja (v, w) o arco corrente de v.

(1) **If** $d(v) = d(w) + 1$ e $rg(v, w) > 0$ **Then Begin**

(1a) **If** $find - size(v) + find - size(w) < k$

Then Begin faça w pai de v executando $change - value(v, -\infty)$; $change - value(v, rg(v))$; e $link(v, w)$; envie excesso de fluxo de v para w executando $send(v)$.

End

(1b) **Else** $\{find - size(v) + find - size(w) > k\}$

Begin aplique uma operação $Push(v)$ para enviar excesso de fluxo de v para w; execute $Send(w)$.

End

End

(2) **Else** $\{d(v) < d(w)$ ou $rg(v, w) = 0\}$

If (v, w) não é o último arco da lista de v

Then troque o arco corrente (v, w) pelo próximo arco da lista de v.

Else (v, w) é o último arco da lista de v

Begin faça o primeiro arco da lista o corrente;

execute $cut(u)$ e $change - value(u, \infty)$ para todo filho u de v;

aplique a operação $Relabel(v)$.

End.

Figura 2.15: Procedimento que monitora as operações do algoritmo.

Uma operação Tree-Push/Relabel aplica-se a um nó ativo que é raiz de uma árvore dinâmica. Existem essencialmente dois casos. O primeiro caso ocorre se o arco corrente (v, w) de v é elegível para execução de envio de fluxo. Se as árvores contendo v e w juntas tiver no máximo k nós, a operação Tree-Push/Relabel liga estas árvores tornando w o pai de v e então executa uma operação Send(v). O segundo caso ocorre se o arco (v, w) não é elegível para um envio de fluxo. Neste caso a operação Tree-Push/Relabel atualiza o arco corrente de v e aplica uma operação Relabel(v) se necessário. Caso a operação Relabel seja aplicável a v, a operação Tree-Push/Relabel corta todos os arcos de árvore que chegam em v, mantendo invariante o

fato de que todos os arcos árvore dinâmica são elegíveis para uma operação de envio de fluxo.

É importante salientar que este algoritmo armazena os valores do pre-fluxo de duas formas diferentes. Se (v, w) é um arco que pertence a qualquer árvore dinâmica então $g(v, w)$ é armazenado explicitamente com (v, w). Se (v, w) for arco árvore dinâmica, então $h(v) = c(v, w) - g(v, w)$ é armazenado implicitamente na estrutura de dados árvore dinâmica. Por isto, sempre que um arco (v, w) é cortado, $h(v)$ deve ser computado e $g(v, w)$ atualizado com seu valor corrente. Além disso, quando o algoritmo termina, o valor do fluxo deve ser computado para todos os arcos remanescentes na árvore dinâmica.

Duas observações implicam que o algoritmo é correto. Primeiro, qualquer arco (v, w) que estiver em alguma árvore dinâmica tem $d(v) = d(w)+1$. Portanto, no caso (1a) da operação Tree-Push/Relabel, os nós v e w estão em árvores diferentes, e o algoritmo nunca tenta ligar uma árvore dinâmica a si mesma. Segundo, um nó v que não é uma raiz de árvore pode ter excesso positivo apenas no meio do caso (1) da operação Tree-Push/Relabel. Para ver isto, basta notar que apenas neste caso é que o algoritmo cria um nó ativo que não é uma raiz de árvore, e este fato é seguido de uma operação Send que leva o excesso deste nó não raiz para um ou mais nós raizes.

O algoritmo árvore dinâmica é executado num tempo de $O(nm \log(k))$ mais o tempo de $O(\log(k))$ por adição de um nó a Q. Como o número de vezes que um nó é adicionado a Q é de $O(nm + n^3/k)$, tem-se o seguinte resultado:

Teorema 2.2 O algoritmo árvore dinâmica tem complexidade $O(nm \log(n^2/m))$ onde $k = n^2/m$ é o limitante para o número de nós em cada árvore.

Prova: Em Goldberg e Tarjan [20], pag 934.

Capítulo 3

Resultados Obtidos

3.1 Algoritmos Utilizados nos Testes Comparativos

Foram implementados os principais algoritmos desenvolvidos para o problema do fluxo máximo. São eles:

1. **GT** - Algoritmo seqüencial de Goldberg e Tarjan [20].
2. **GTDin** - Algoritmo com árvores dinâmicas de Goldberg e Tarjan[20].
3. **EK** - Algoritmo de Edmonds e Karp [10].
4. **EKSca** - Algoritmo de Edmonds e Karp com "scaling" [10].
5. **DMa** - Algoritmo tipo Dinic desenvolvido por Malhotra et alli [22].
6. **DTa** - Algoritmo tipo Dinic desenvolvido por Tarjan [34].
7. **SIMPLEX** - Método Simplex Primal para redes [7].

Os testes foram realizados com tais algoritmos pois estes representam as mais significantes abordagens para o problema. Com exceção do método Simplex, os demais algoritmos têm complexidades polinomiais, são simples e intuitivos tanto em seus conceitos quanto nas suas implementações.

A seguir apresentamos uma breve descrição dos algoritmos.

- ALGORITMO DE GOLDBERG E TARJAN $O(n^3)$ - 1986

  Este algoritmo mantém um prefluxo na rede original e procura enviar o excesso de fluxo local em direção ao destino através do menor caminho estimado. Como estimativa de distância, o algoritmo utiliza o conceito de rotulamento válido que é um limitante inferior da distância de cada nó até o destino.

- ALGORITMO DE GOLDBERG E TARJAN COM ÁRVORES DINÂMICAS $O(nm\log(n^2/m))$ - 1986

  Este algoritmo incorpora a estrutura de dados tipo Árvore Dinâmica ao algoritmo anterior.

- ALGORITMO DE EDMONDS E KARP $O(nm^2)$ - 1969

  Este algoritmo determina um único caminho aumentante mínimo por iteração. Se tal caminho é encontrado, uma ampliação do fluxo corrente é realizada, caso contrário o fluxo corrente é máximo e o processo é interrompido.

- ALGORITMO DE EDMONDS E KARP COM "SCALING"

  É uma segunda versão do algoritmo anterior que utiliza a técnica de *scaling* no envio de fluxo através dos caminhos aumentantes.

- ALGORITMOS TIPO DINIC

  Estes algoritmos determinam um fluxo bloqueante no referente proposto por Dinic, trabalhando portanto com vários caminhos aumentantes em paralelo em cada iteração. O que difere cada um destes algoritmos é a forma de determinar um fluxo bloqueante.

  - ALGORITMO DE MALHOTRA ET ALLI $O(n^3)$ - 1978

    Este algoritmo trabalha com o potencial referente de cada nó da rede que é dado pelo mínimo entre a soma das capacidades residuais dos arcos que chegam no nó (da camada anterior do referente) e a mesma soma em relação aos arcos que saem do nó (para a camada seguinte do referente). Encontrando o menor potencial referente sobre todos os nós da rede; caminhos aumentantes podem ser determinados a partir deste nó. A iteração é interrompida quando a origem e o destino ficam desconectados.

- ALGORITMO DA ONDA DE TARJAN $O(n^3)$ - 1984

  Este algoritmo é considerado um dos melhores representantes desta classe de algoritmos. Tarjan utiliza o conceito de prefluxo para encontrar o fluxo bloqueante, assim o algoritmo envia o máximo de fluxo até o destino sem que a lei de equilíbrio dos nós seja necessariamente satisfeita. Ao alcançar o destino, o algoritmo retorna em direção à origem movendo o excesso de fluxo nos nós intermediários para a camada anterior. Desta forma, na próxima ida em direção ao destino, tenta-se enviar estes excessos por caminhos alternativos.

- SIMPLEX $O(e^n)$ - 1947

  Foi utilizada a implementação do Método Primal Simplex para programação em redes. Trata-se de um algoritmo fortemente factível [7] e que foi extensivamente testado anteriormente [4].

O espaço de memória requisitado por cada algoritmo é de:

GTDIN	GT	DTa	DMa	EKSca	EK	SIMPLEX
$11n + 11m$	$6n + 6m$	$6n + 7m$	$7n + 6m$	$4n + 6m$	$5n + 6m$	$6n + 4m$

Tabela 3.1: Nesta tabela n e m correspondem a vetores inteiros n e m-dimensionais. Vetores booleanos também foram considerados inteiros.

3.2 Redes Testadas

As redes geradas são armazenadas em um arquivo de dados a ser lido por cada algoritmo implementado. Este arquivo de entrada contém todas as informações da rede e tem a seguinte configuração:

- O primeiro registro contém os seguintes dados:

1. n = número de nós.
2. m = número de arcos.
3. s = nó origem (ou fonte) do fluxo máximo.
4. t = nó destino (ou sorvedouro) do fluxo máximo.

- Os demais registros se referem aos arcos da rede, os quais são armazenados numa ordem arbitrária porém fixa. Cada registro contém os seguintes dados:

  1. $t[j]$ = nó cauda do arco j (nó de onde sai o arco).
  2. $h[j]$ = nó cabeça do arco j (nó onde chega o arco).
  3. $c[j]$ = capacidade do arco j.

A saída é composta por:

- $f[j]$ = vetor de fluxo.
- $p[i]$ = vetor partição da rede.
- vf = valor do fluxo máximo.
- nt = número de iterações.
- cpu = tempo de cpu.

Três tipos diferentes de rede foram geradas a fim de se efetuar os testes comparativos. São eles:

Tipo 1: rede em camadas com a origem ligada a todos os nós da primeira camada e com todos os nós da última camada ligados ao destino. Cada um dos outros arcos liga um nó de uma camada a um nó da camada seguinte.

Um arco que liga duas camadas é gerado com probabilidade pré-definida (dado de entrada) e sua capacidade gerada com distribuição uniforme em um intervalo $[c_{min}, c_{max}]$ pré-definido. Os arcos que saem da origem e também aqueles que chegam no destino tem capacidades c_{max}.

Para gerar este tipo de rede são fornecidos os seguintes dados:

- s = origem,

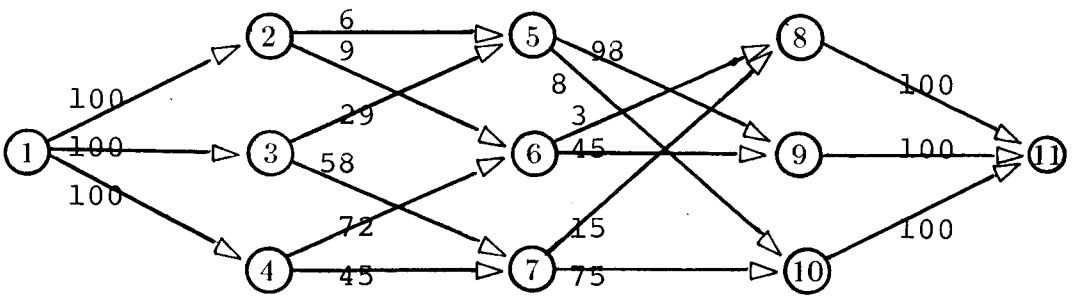

Figura 3.1: Rede Tipo 1 com $s = 1$, $t = 11$, $l = 3$, $k = 3$, $p = 2/3$.

- t = destino,
- l = número de nós em cada camada (ou tamanho da camada),
- k = número de camadas,
- p = probabilidade de existência de cada arco.
- seed = semente geradora.

O número de nós desta rede é $n = lk + 2$ e o número de arcos esperado para cada rede é $E[m] = l^2(k-1)p + 2l$.

Este tipo de rede foi inspirado no trabalho de Martel [26]. Um exemplo destas redes é dado na figura 3.1.

Tipo 2: é uma extensão do primeiro tipo onde podem existir arcos com probabilidade p saindo de uma camada e chegando em uma camada posterior (não necessariamente a seguinte). Este tipo de rede permite a existência de caminhos aumentantes de qualquer tamanho. Os arcos desta rede também têm suas capacidades geradas com distribuição uniforme em um intervalo $[c_{min}, c_{max}]$ pré-definido.

Para gerar este tipo de rede são fornecidos os seguintes dados:

- s = origem,
- t = destino,
- l = número de nós em cada camada (ou tamanho da camada),
- k = número de camadas,

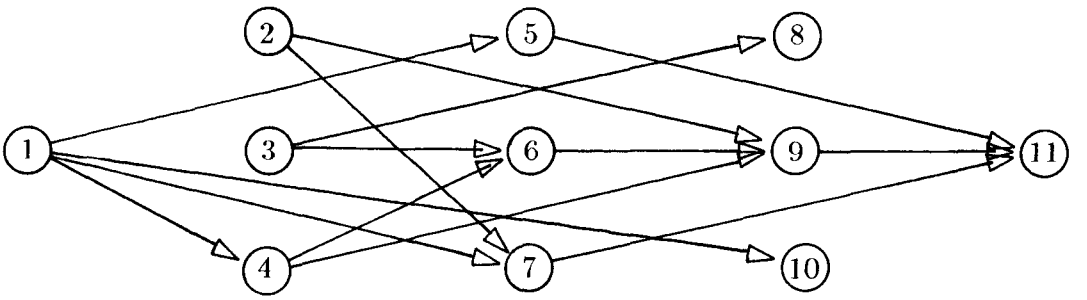

Figura 3.2: Rede Tipo 2 com $s = 1$, $t = 11$, $l = 3$, $k = 3$, $p = 1/3$, $c_j \in [1,100]$.

- p = probabilidade de existência de cada arco.
- seed = semente geradora.

O número de nós desta rede é $n = lk + 2$ e o número de arcos esperado para cada rede é $E[m] = pl^2k(k-1)/2 + 2plk + p$. Um exemplo destas redes aparece na figura 3.2.

Tipo 3: rede gerada com base num grafo completo onde a existência de cada arco depende da probabilidade fornecida. Este tipo tem por finalidade representar as redes sem estruturas, embora elas guardem características do grafo completo. Para gerar tal tipo de rede basta fornecer o número de nós e a probabilidade de existência de cada arco. O número de nós da rede é n e o número esperado de arcos é $E[m] = pn(n-1)$ onde n e p são pré-fixados. Um exemplo destas redes é dado na figura 3.3.

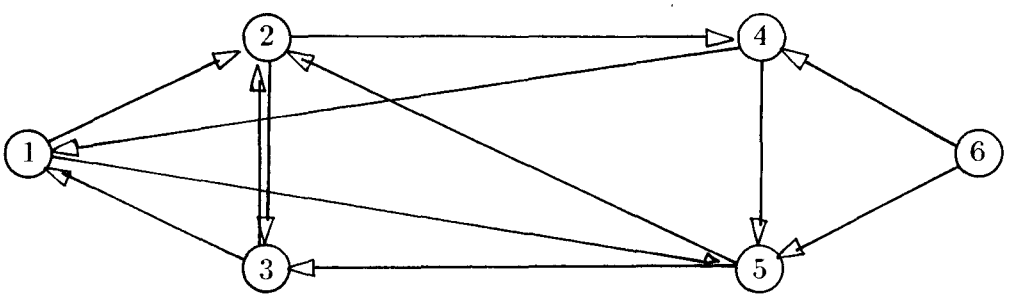

Figura 3.3: Rede Tipo 3 com $s = 1$, $t = 6$, $p = 1/3$, $c_j \in [1,100]$.

3.3 Resultados Obtidos nos Testes Comparativos

Utilizamos a linguagem "Pascal" na implementação dos algoritmos e os testes foram realizados em uma Estação de Trabalho "SPARCstation 1+", com chip Risc, memória principal de 20 megabytes, rodando o sistema operacional Sun OS 4.1.

Cada problema é definido fixando-se os parâmetros $l, k, p.$ (nas redes tipo 1 e 2), n e p (na rede tipo 3). Foram geradas entre 5 e 10 redes com os mesmos parâmetros e diferentes sementes, tendo $[c_{min}, c_{max}] = [1, 100]$.

3.3.1 Testes Realizados com Redes do Tipo 1

Inicialmente fixamos o número de camadas k e o número de nós por camada l, e variando a probabilidade p obtivemos redes de diferentes densidades.

Para $l = 50$, $k = 50$ e $p \in \{0,1; 0,2; ...; 0,8\}$ ou seja, variando a densidade de uma rede "quadrada" de 50×50 temos $n = 2500$, $12400 \leq m \leq 98000$. Os resultados apresentados abaixo são as médias dos tempos de CPU em segundos. Estes tempos não incluem as operações de entrada e saída.

l	k	p	GTDIN	GT	DTa	DMa	EKSca	EK	SIMPLEX
50	50	0,1	15,9	3,8	1,2	16,8	25,8	500,1	269,0
		0,2	5,7	1,3	1,1	10,8	32,0	1029,0	383,2
		0,3	5,3	1,2	1,2	12,9	47,6	1499,0	515,2
		0,4	5,1	1,3	1,2	16,3	63,2	1971,8	675,7
		0,5	5,2	1,4	1,5	18,6	71,8	2365,2	837,7
		0,6	5,3	1,5	1,6	21,5	91,0	2921,2	1034,3
		0,7	5,2	1,6	1,7	24,7	105,5	3380,3	1214,2
		0,8	5,5	1,7	1,9	27,3	113,6	3797,6	1397,2

Tabela 3.2: Segundos de CPU para Redes do Tipo 1.

Nestes testes temos: n constante, $E[m] \propto p$ e $E[m]/n \propto p$.

É importante salientar que os valores na tabela acima são as médias dos tempos de solução de 5 problemas de mesmo porte com o desvio padrão da ordem de 2 % das médias.

Pode-se observar que os métodos Simplex EK e EKSca são os únicos que apresentam o tempo de CPU sempre crescente com relação ao número de arcos esperados (ocasionado pela variação de p). Os método DTa e DMa que determinam o referente a cada iteração, apresentam o menor tempo de CPU para as redes do tipo 1 com $p \approx 0,2$. Os métodos GT e GTDin apresentam os menores tempos de CPU para $p \approx 0,3$ e $p \approx 0,4$ respectivamente. É fácil verificar que os melhores resultados foram obtidos com os métodos DTa ($p \leq 0,4$) e GT ($p \geq 0,5$).

Observa-se também que o método Simplex apresenta resultados melhores do que EK.

A técnica "*scaling*" é eficiente nestas redes testadas conforme pode-se verificar comparando os tempos de EK (sem scaling) com EKSca (com scaling). O uso de referentes é significativo conforme demonstrado pelos tempos de DTa, DMa contra EK e EKSca.

Nestas redes, para $p \geq 0,2$ o corte mínimo é constituido apenas pelos arcos que saem da origem s ou pelos arcos que chegam no destino t. Isto não ocorre para $p < 0,2$.

A seguir apresentamos os gráficos relacionados com os resultados acima.

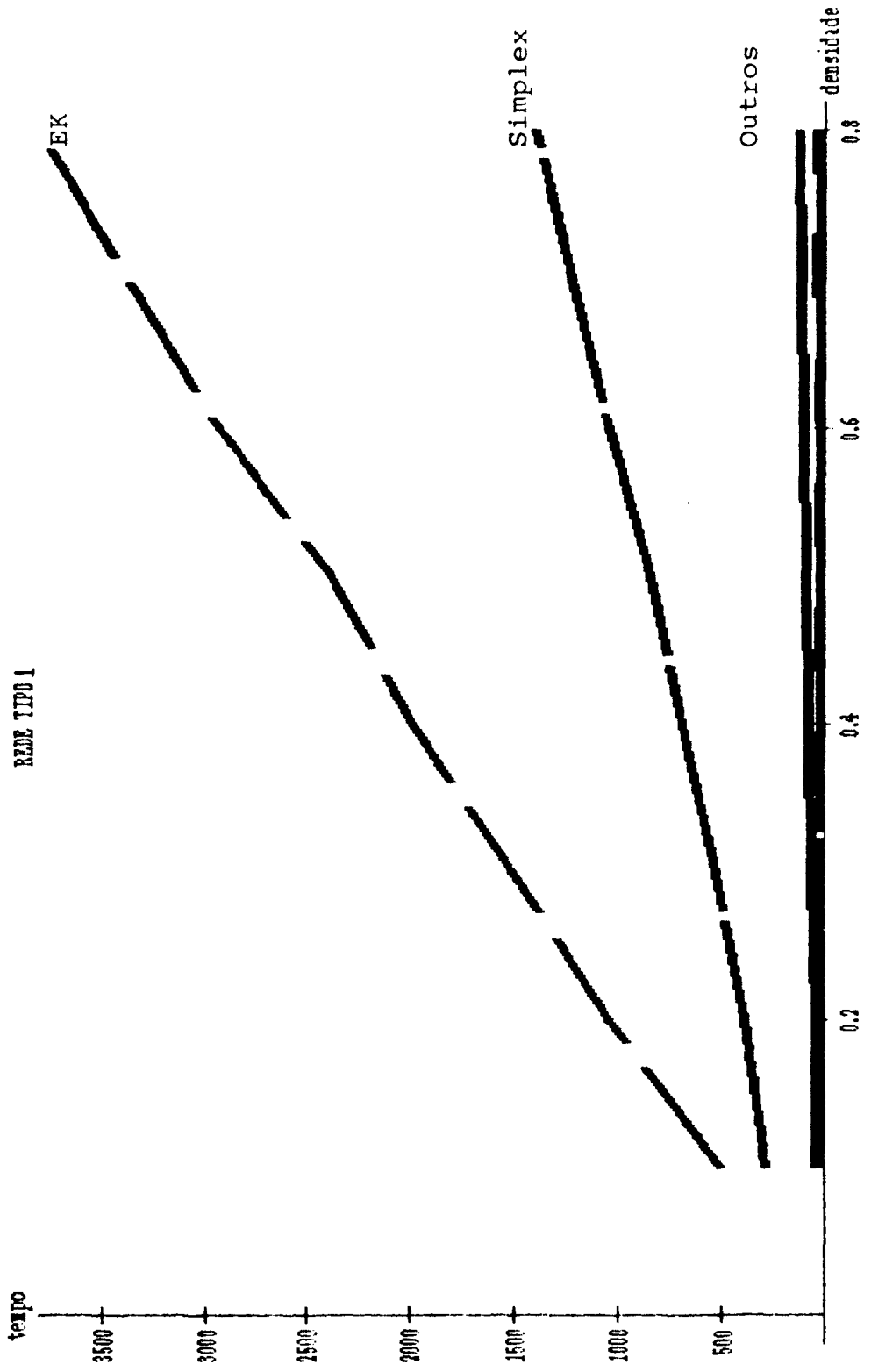

Gráfico 1: Contendo todos os métodos, onde o tempo é dado em segundos.

Como estes métodos apresentam tempos muito diferentes, não é possível visualizar o desempenho de cada algoritmo.

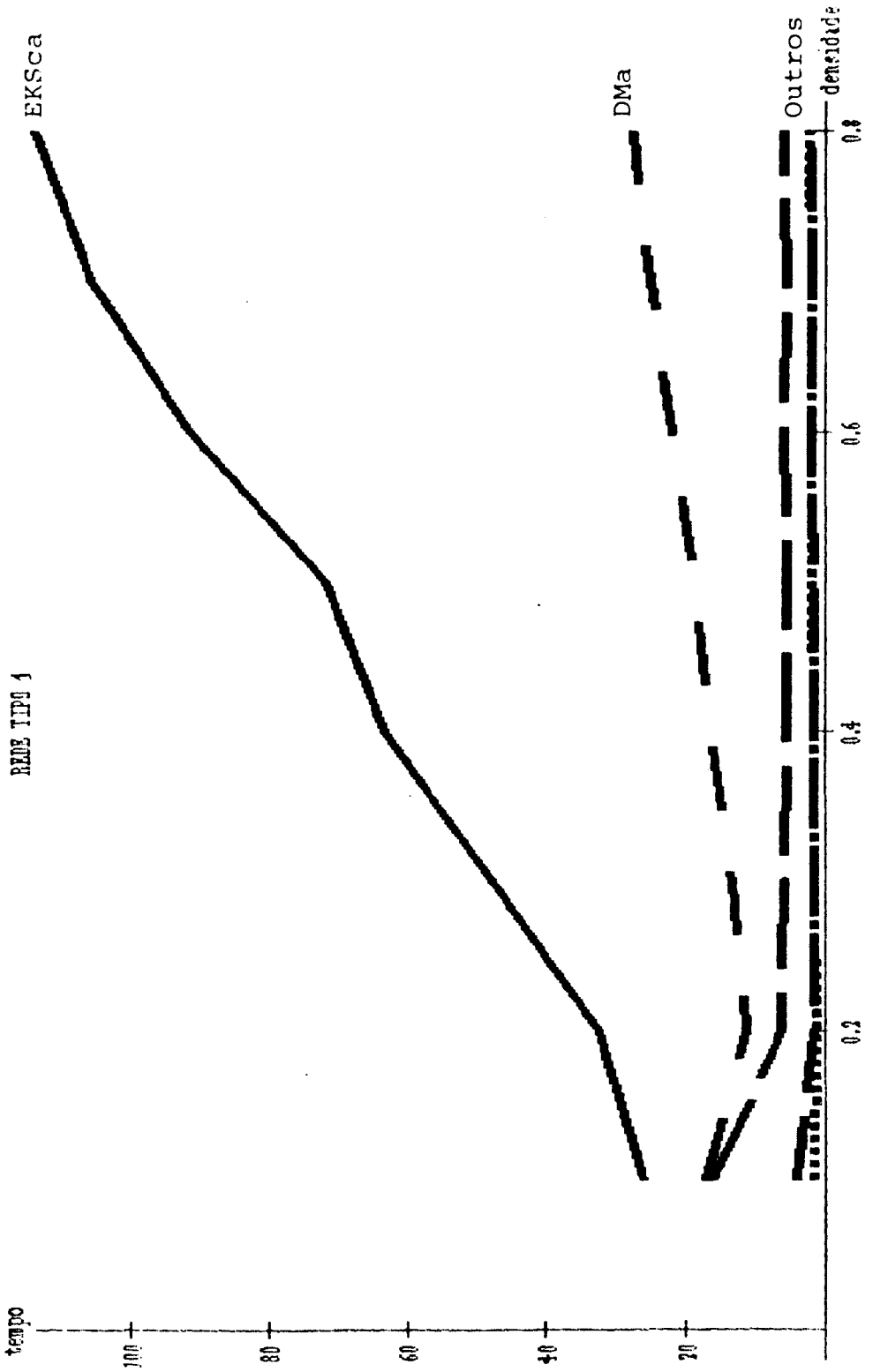

Gráfico 2: Retirando do gráfico anterior os dois métodos com maior tempo de CPU: EK e SIMPLEX.

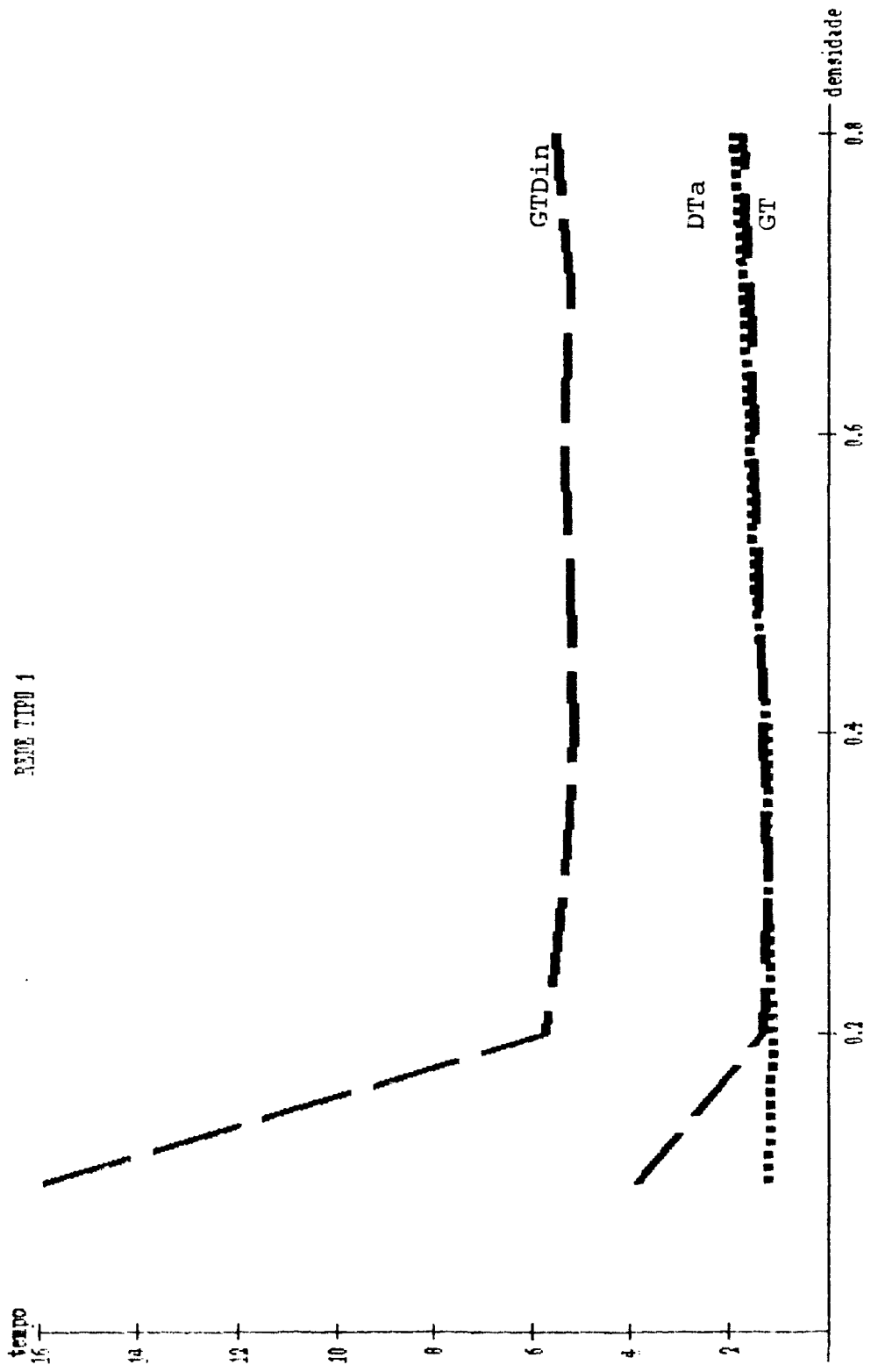

Gráfico 3: Comparação entre os algoritmos GT, GTDin e DTa (a melhor versão do algoritmo tipo Dinic).

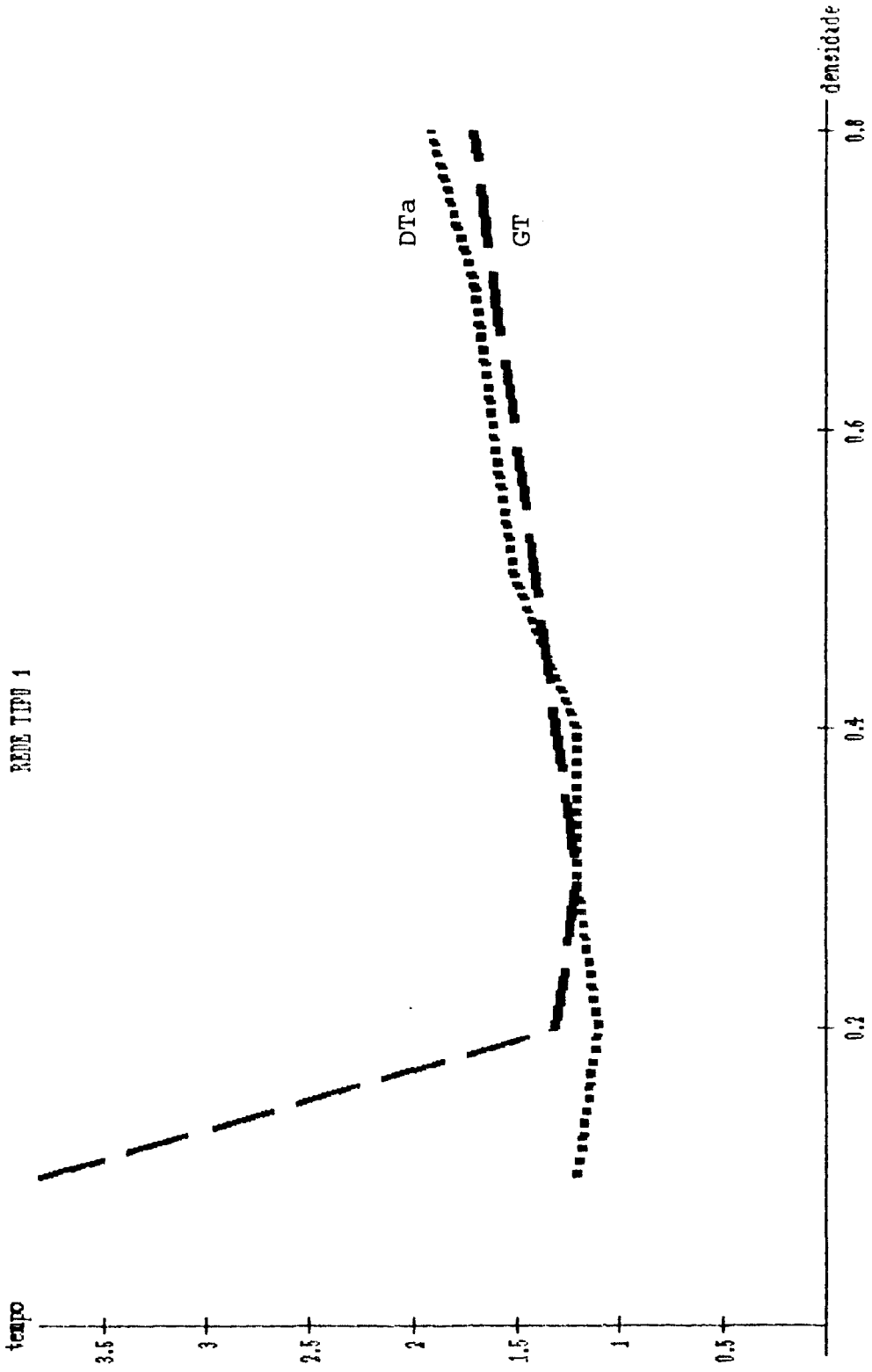

Gráfico 4: Comparação entre os dois melhores métodos: GT e DTa.

Testamos os algoritmos agora variando o número de nós por camada l, mantendo os outros parâmetros do gerador constantes. Neste caso temos: $n \propto l$, $E[m] \propto l^2$, mantendo a relação $E[m]/n^2$ constante. Os resultados são apresentados na tabela 3.3.

l	k	p	GTDIN	GT	DTa	DMa	EKSca	EK	SIMPLEX
25	50	0,5	2,5	0,6	0,5	4,3	10,3	311,5	127,7
50	50	0,5	5,2	1,4	1,5	18,6	71,8	2365,2	837,7
75	50	0,5	8,5	2,7	3,2	46,3	265,8	8505,0	3012,5

Tabela 3.3: Resultados de Testes com a Rede Tipo 1 variando-se l.

O aumento no número de nós por camada provoca um aumento no tempo de execução em todos os métodos. Da tabela pode-se observar que a técnica de "*scaling*" é eficiente comparando os tempos de EKSca contra EK. Da mesma forma, o uso de referentes é importante pois os tempos de DTa e DMa são melhores do que EKSca e EK.

Também fizemos variar o número de camadas k da rede, mantendo os outros parâmetros fixos. Para este caso temos as seguintes relações: $n \propto k$ e $E[m] \propto k$, mantendo a relação $E[m]/n$ constante.

l	k	p	GTDIN	GT	DTa	DMa	EKSca	EK	SIMPLEX
50	25	0,5	2,6	0,7	0,7	8,3	37,3	723,8	193,7
50	50	0,5	5,2	1,4	1,5	18,6	71,8	2365,2	837,7
50	75	0,5	7,7	2,0	2,0	29,2	115,9	4645,5	1937,5

Tabela 3.4: Resultados de Testes com a Rede Tipo 1 variando-se o k.

Observa-se que o tempo de execução de todos os métodos aumentam com o aumento do número de camadas. Os mesmas observações quanto ao uso da técnica de "*scaling*" e do referente podem ser feitas também neste caso. O gráfico 6 se refere à tabela 3.4.

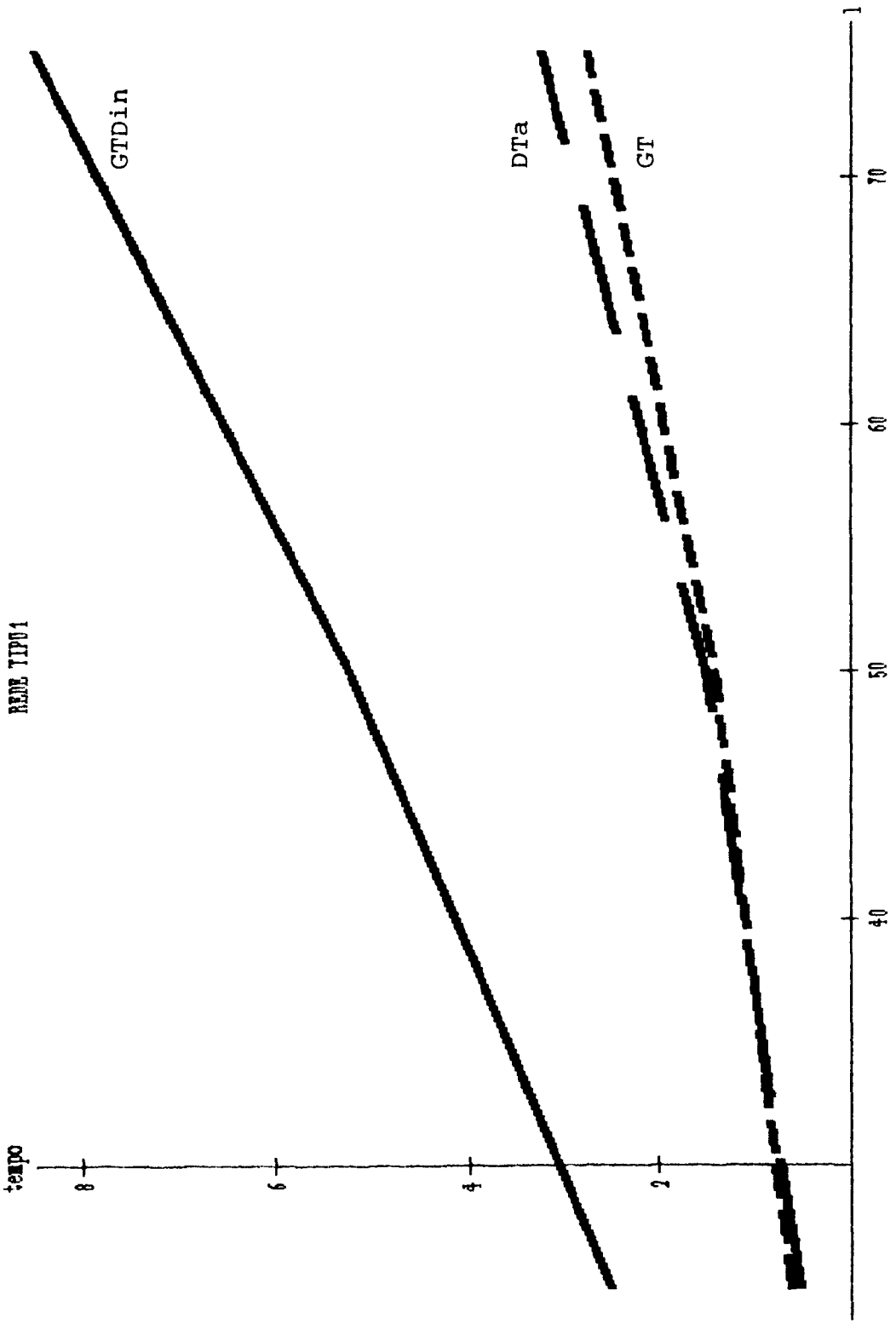

Gráfico 5: Comparação entre os três melhores métodos na tabela 3.3.

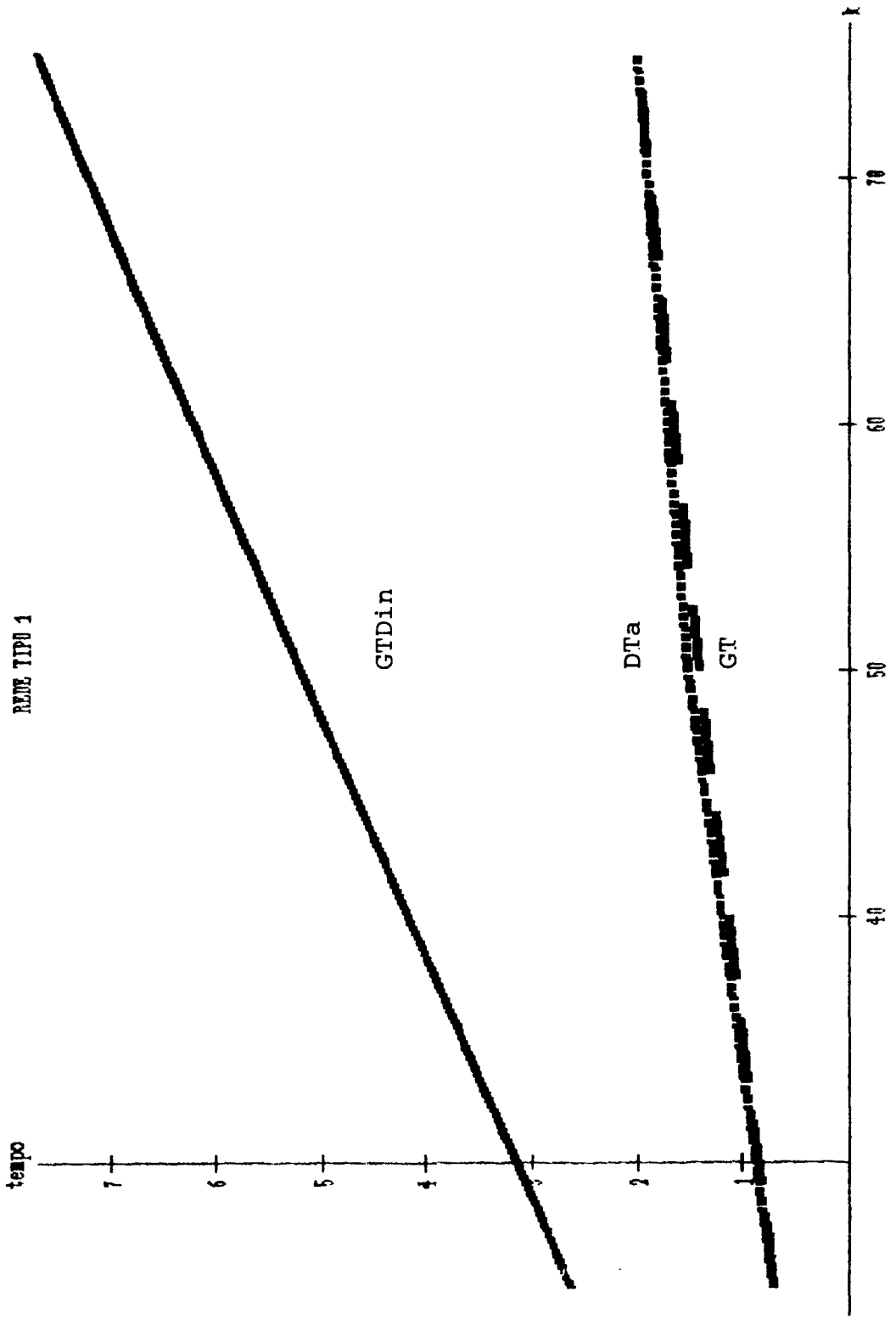

Gráfico 6: Comparação entre os três melhores métodos na tabela 3.4.

3.3.2 Testes Realizados com Redes do Tipo 2

Foi utilizada a mesma estratégia das redes tipo 1 ou seja, fixamos o número de camadas k e o número de nós por camada l, e variando a probabilidade p obtivemos redes de diferentes densidades. O número de nós é $n = 920$ e para $p \in \{0,01; 0,05; ...0,5\}$ temos $3000 \leq m \leq 196000$. Os resultados apresentados abaixo são as médias dos tempos de CPU em segundos para grupos de 5 a 10 problemas do mesmo tipo.

l	k	p	GTDIN	GT	DTa	DMa	EKSca	EK	SIMPLEX
30	30	0,01	106,3	18,4	0,2	0,1	0,2	0,1	0,9
		0,05	467,3	223,3	10,2	4,2	6,3	16,1	7,6
		0,10	422,9	258,9	19,5	8,4	20,3	50,6	13,4
		0,15	740,6	500,0	27,5	14,2	42,6	82,1	14,8
		0,20	807,7	577,4	39,4	27,1	53,7	126,1	25,1
		0,25	821,3	596,1	14,2	15,7	179,8	253,5	15,7
		0,30	663,4	428,9	88,0	25,4	258,2	403,8	31,2
		0,40	1266,5	1054,9	138,3	45,4	538,3	803,1	57,8
		0,50	2118,0	1631,7	165,0	46,1	908,1	1181,9	37,3

Tabela 3.5: Segundos de CPU para Rede do Tipo 2.

Nesta tabela temos: n constante e $E[m] \propto p$.

O uso da técnica de "*scaling*" é aconselhável ao observarmos os tempos de EK, EKSca. Da mesma forma, o uso de referente produz bons resultados se compararmos os métodos DTa, DMa contra EKSca.

Os métodos GTDin e GT são os que tem os piores desempenhos para este tipo de rede sendo que GT é sempre melhor que GTDin.

Observa-se que os melhores tempos foram obtidos por DMa e Simplex.

A seguir apresentamos os gráficos relacionados com os resultados acima.

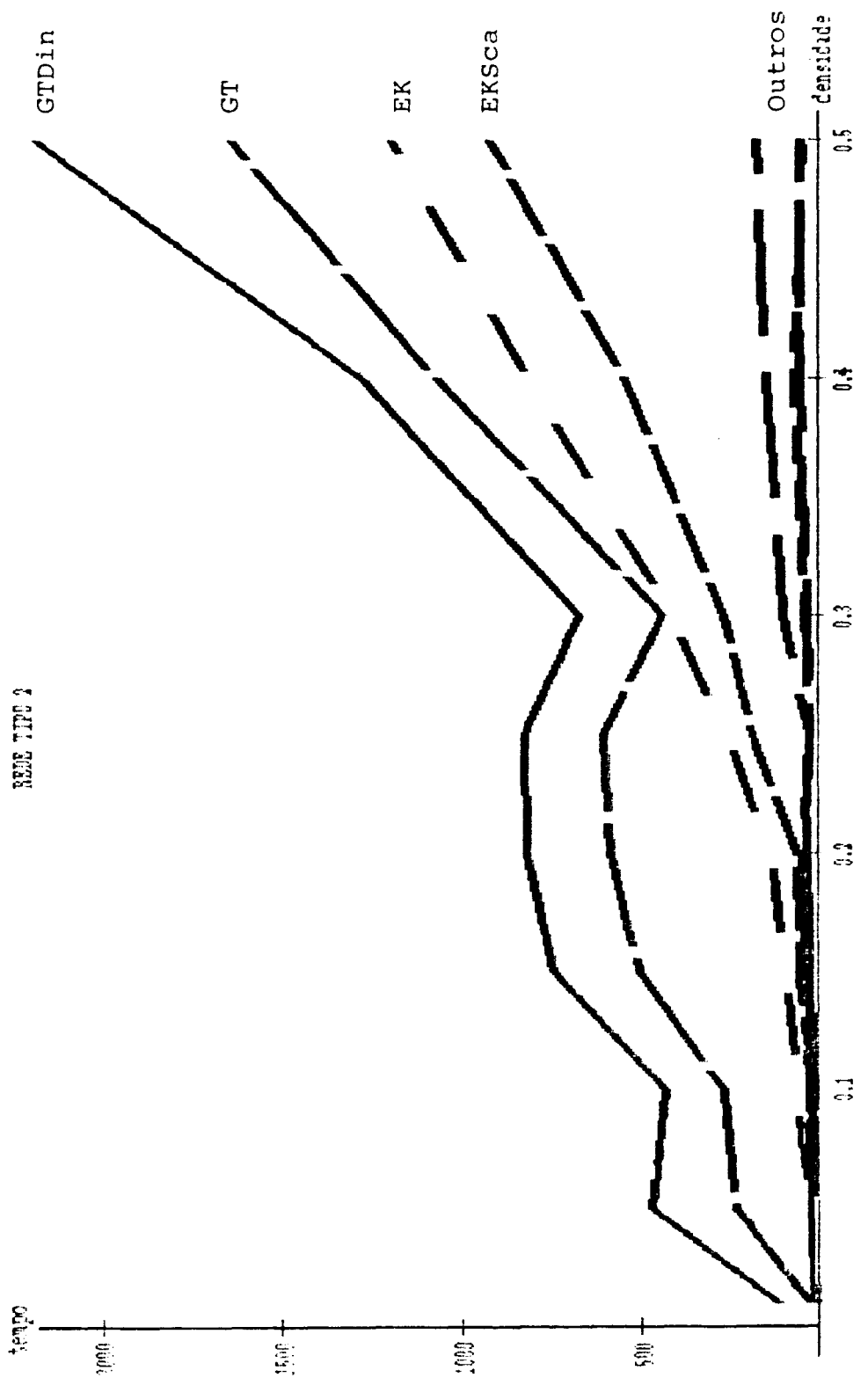

Gráfico 7: Contendo todos os métodos, onde o tempo é dado em segundos.

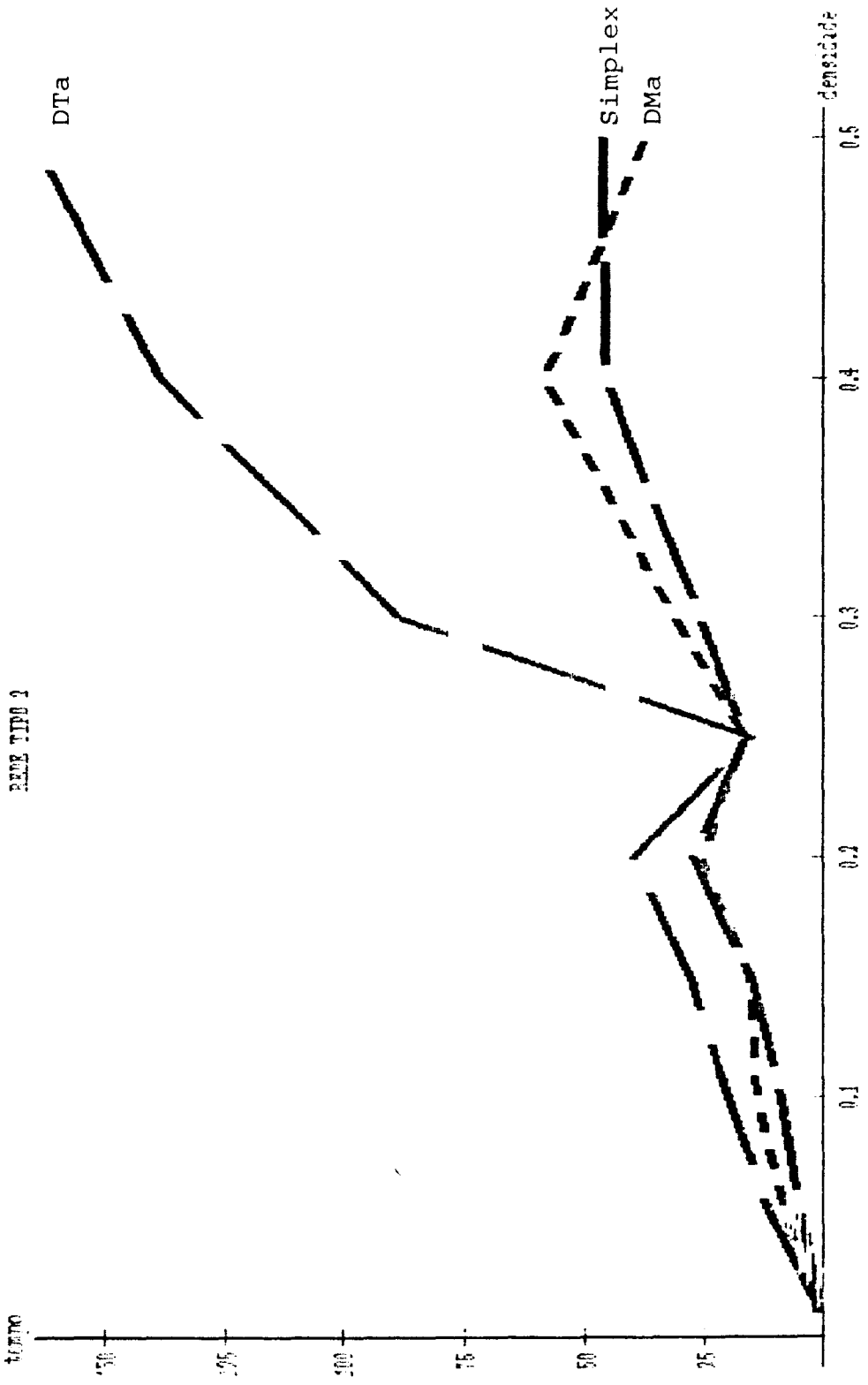

Gráfico 8: Contendo apenas os melhores métodos: DTa. DMa e Simplex.

Uma das justificativas apresentadas por Goldberg e Tarjan para o uso de árvores dinâmicas é o fato de reduzir o número de operações Push saturantes executadas. Estas operações são contabilizadas na complexidade dos algoritmos GT e GTDin. O gráfico 9 apresenta o número destas operações para os problemas da rede tipo 2.

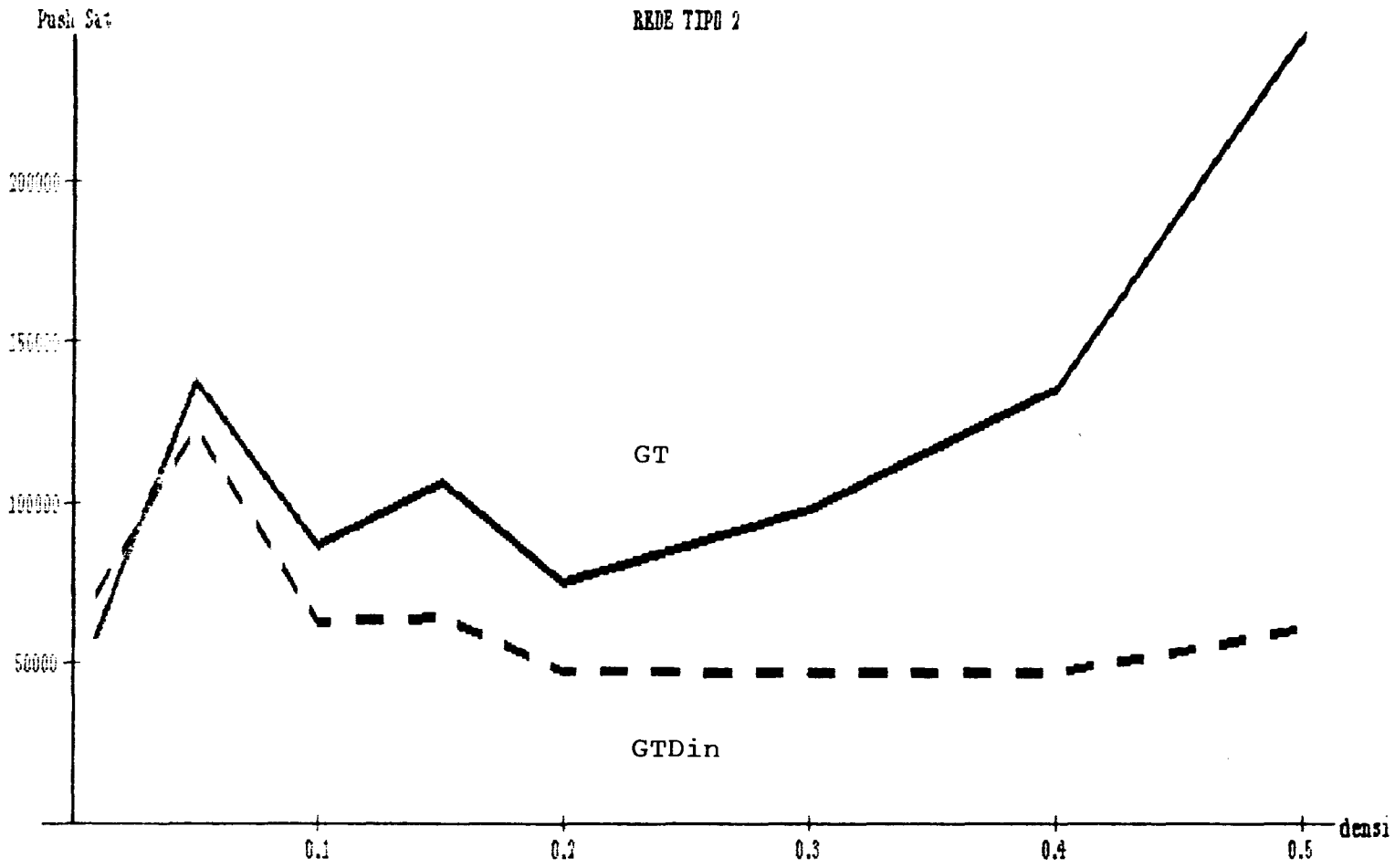

Gráfico 9: Número de operações Push saturantes de GTDin e GT.

3.3.3 Testes Realizados com Redes do Tipo 3

Foi utilizada a mesma estratégia dos casos anteriores ou seja, fixando neste caso o número de nós n, variamos a probabilidade p, obtendo redes do tipo 3 com diferentes densidades.

Para $n = 2000$ e $p \in \{0,001; 0,005; 0,01; 0,015; 0,02; 0,025; 0,03\}$ temos $4000 < m < 120000$. Os resultados apresentados abaixo são as médias dos tempos de CPU em segundos oara grupos de 4 a 10 problemas do mesmo tipo.

As colunas GTDIN* e GT* são médias que não envolvem todos os dados conhecidos. A última coluna indica a porcentagem dos dados utilizada na obtenção destas médias.

p	GTDIN	GT	DTa	DMa	EKSca	EK	SIMPLEX	GTDIN*	GT*	%
0,001	689,7	128,3	1,5	0,2	0,2	0,1	6,7	2,2	0,3	56
0,005	1207,6	279,5	6,0	1,9	2,4	5,5	19,0	2,7	1,0	70
0,010	3356,9	1564,5	5,9	2,7	9,7	20,7	28,1	2,8	1,3	40
0,015	3677,6	1971,6	6,8	3,7	16,1	27,8	35,0	4,0	2,2	50
0,020	6991,5	3911,4	6,6	4,9	36,2	58,1	38,8	5,1	3,0	25
0,025	8214,7	4865,7	8,5	7,2	52,9	84,2	44,4	5,3	3,2	25
0,030	3697,7	2031,7	8,6	9,0	60,6	111,5	50,0	5,9	3,3	67

Tabela 3.6: Segundos de CPU para Rede do Tipo 3, com $n = 2000$.

Nesta tabela temos n constante e $E[m] \propto p$.

Com exceção dos métodos GTDin e GT, todos demais apresentam valores de tempo crescentes com relação a p.

Observando as colunas de EK e EKSca podemos concluir que a técnica de "*scaling*" é significante. O uso de referente é interessante para valores de $p \geq 0,005$ ao analisarmos os métodos DTa e DMa contra EKSca.

Foi observado que os métodos GT e GTDin apresentam tempos bastante diferentes dentro da mesma classe de problemas ocasionando valores de desvio padrão altos com respeito à média, tais como $\sigma = 1840s$, $\sigma = 557s$, para $p = 0,005$, nos casos extremos para GTDin, GT respectivamente. Neste caso ($p = 0,005$), considerando os sete melhores dentre os 10 tempos

obtidos, resultam as médias $2,7s$ e $1,0s$ para GTDin e GT respectivamente. Os outros 3 problemas produziram médias de $4019s$ e $1393s$ para GTDin e GT respectivamente.

Os demais métodos apresentam os seguintes coeficientes de variação (i. e., proporções de desvio padrão com relação à média) para p = 0,005: DMa 11 %, Simplex 13 %, DTa 30%, EK 54% e EKSca 67%.

No conjunto de redes geradas, foi verificado que os cortes mínimos podem ser classificados em dois tipos. O primeiro tipo corresponde à partição $(S, \bar{S})$ onde S = $\{s\}$, $\bar{S} = V - \{s\}$. Neste caso, a quantidade de fluxo que sai da origem durante a inicialização dos algoritmos GTDin e GT já é o valor do fluxo máximo e nenhum excesso deve ser retornado à origem após encontrar o corte mínimo. O segundo tipo corresponde à partição $(S, \bar{S})$ com S = V - $\{t\}$, $\bar{S} = \{t\}$. Neste caso o menor corte fora o corte mínimo é o da origem, o que implica que o prefluxo que deixa a origem é empurrado até o corte mínimo e o excesso tem que ser retornado para a origem. O comportamento do algoritmo neste tipo de rede indica que se trata de exemplos de situações extremas de mau desempenho (corte no destino) e bom desempenho (corte na origem).

O fato de terem sido obtidos apenas estes dois tipos de corte mínimo pode ser explicado da seguinte forma. Considere uma partição $(S, \bar{S})$ onde $k = \min\{|S|, |\bar{S}|\}$. O valor esperado do corte associado a esta partição é $E[c(S, \bar{S})] = k(n-k)p(c_{max} - c_{min})/2$ onde p é a probabilidade de geração de cada arco e (c_{min}, c_{max}) é o intervalo de geração das capacidades dos arcos. Mantendo n, p, c_{min}, c_{max} constantes , observa-se que o mínimo de $E[c(S, \bar{S})]$ é obtido para $k = 1$ ou $k = n$ - 1 o que implica nos dois tipos de cortes discutidos acima.

A seguir apresentamos os graficos relacionados com os resultados acima.

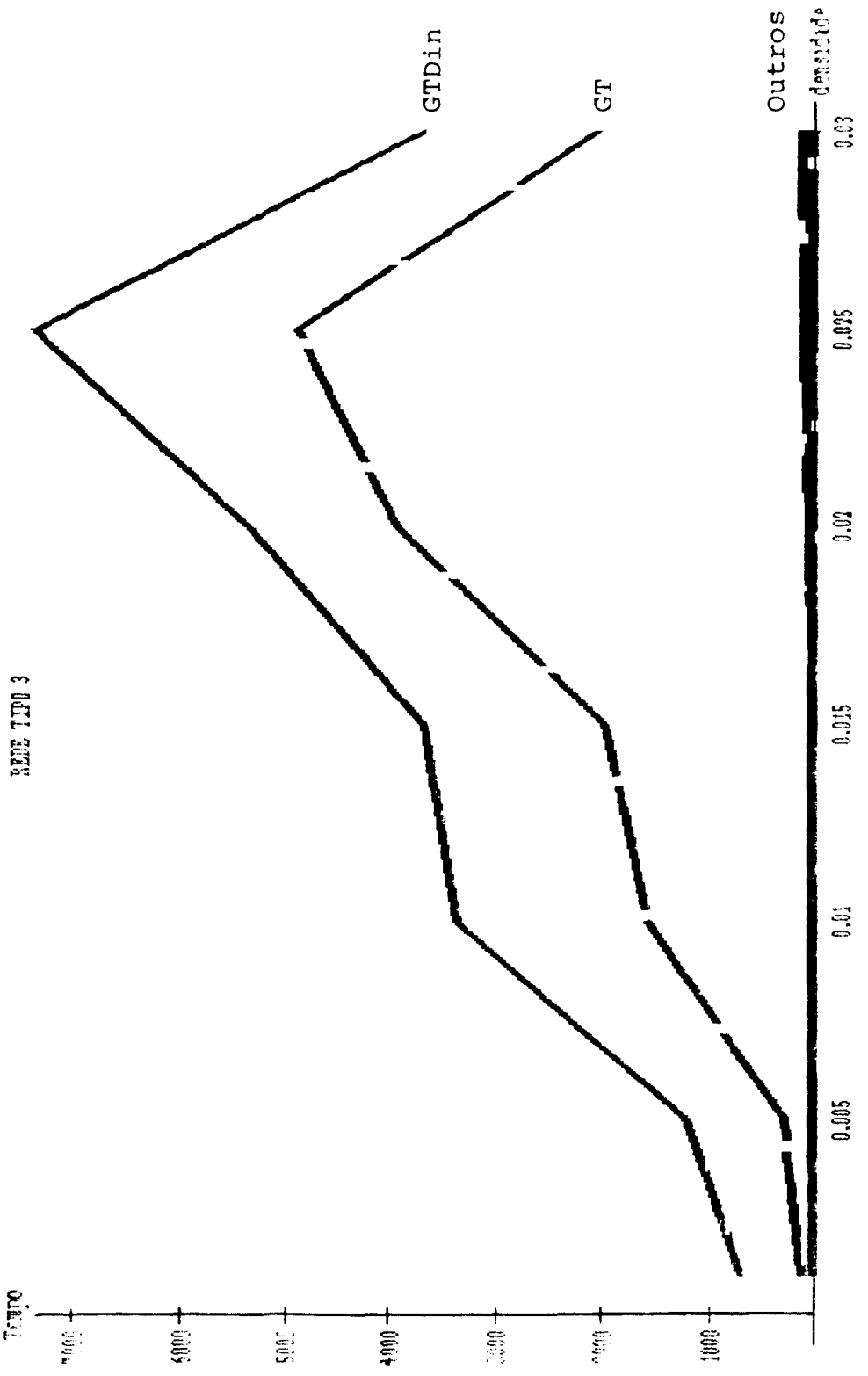

Gráfico 10: Contendo todos os métodos, onde o tempo é dado em segundos.

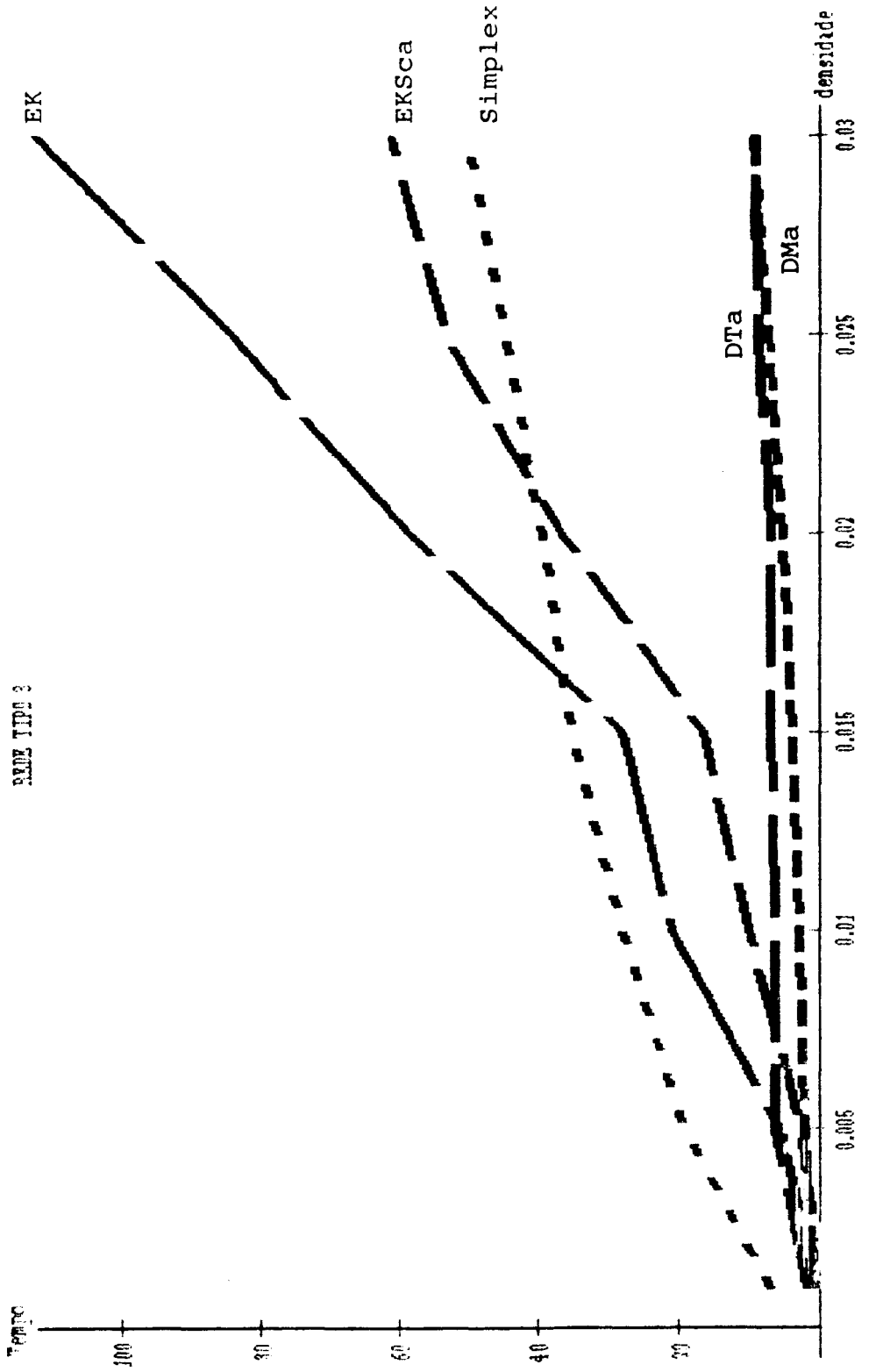

Gráfico 11: Retirando do gráfico anterior os dois piores métodos.

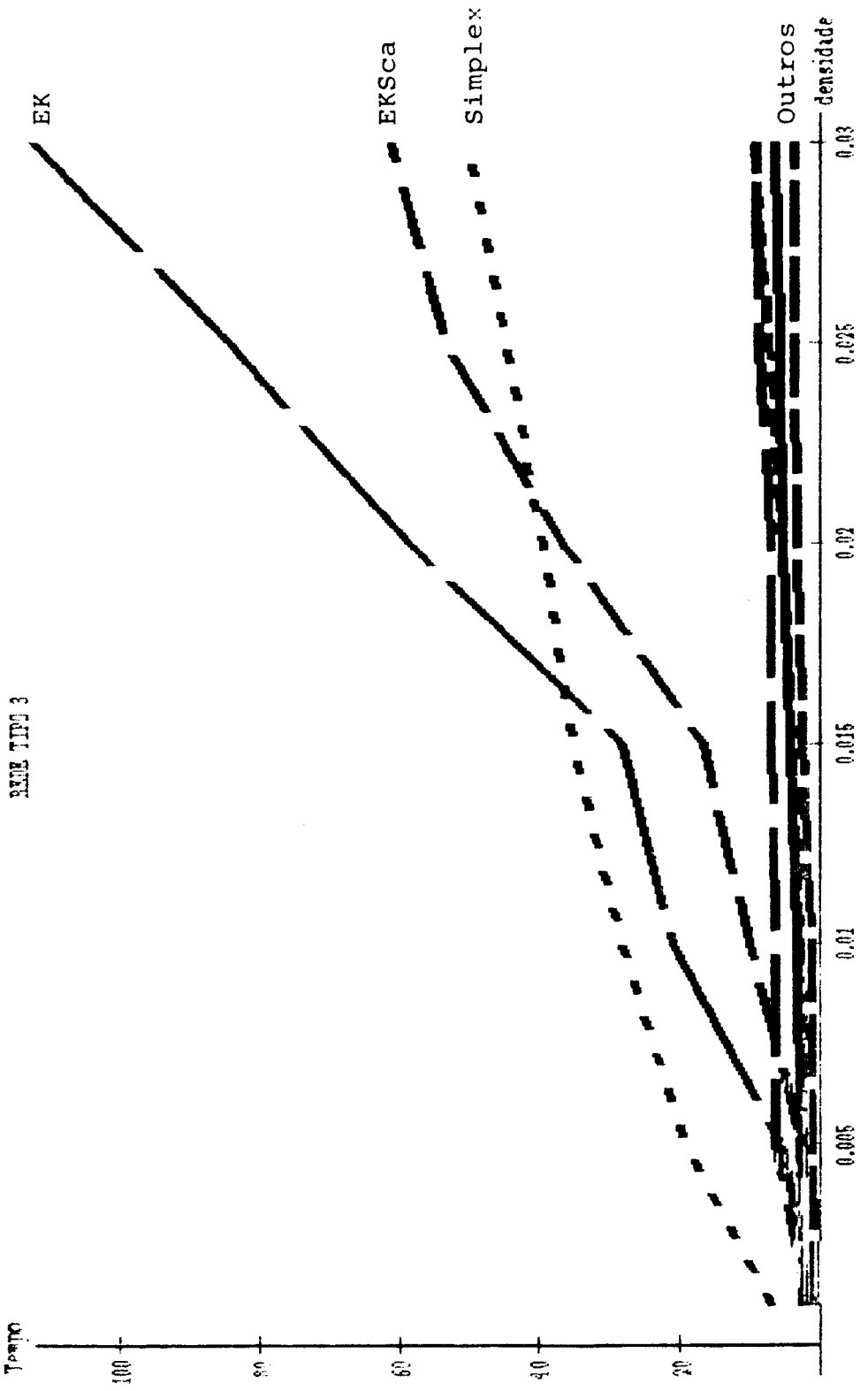

Gráfico 12: Considerando agora os tempos nas colunas GTDin* e GT*.

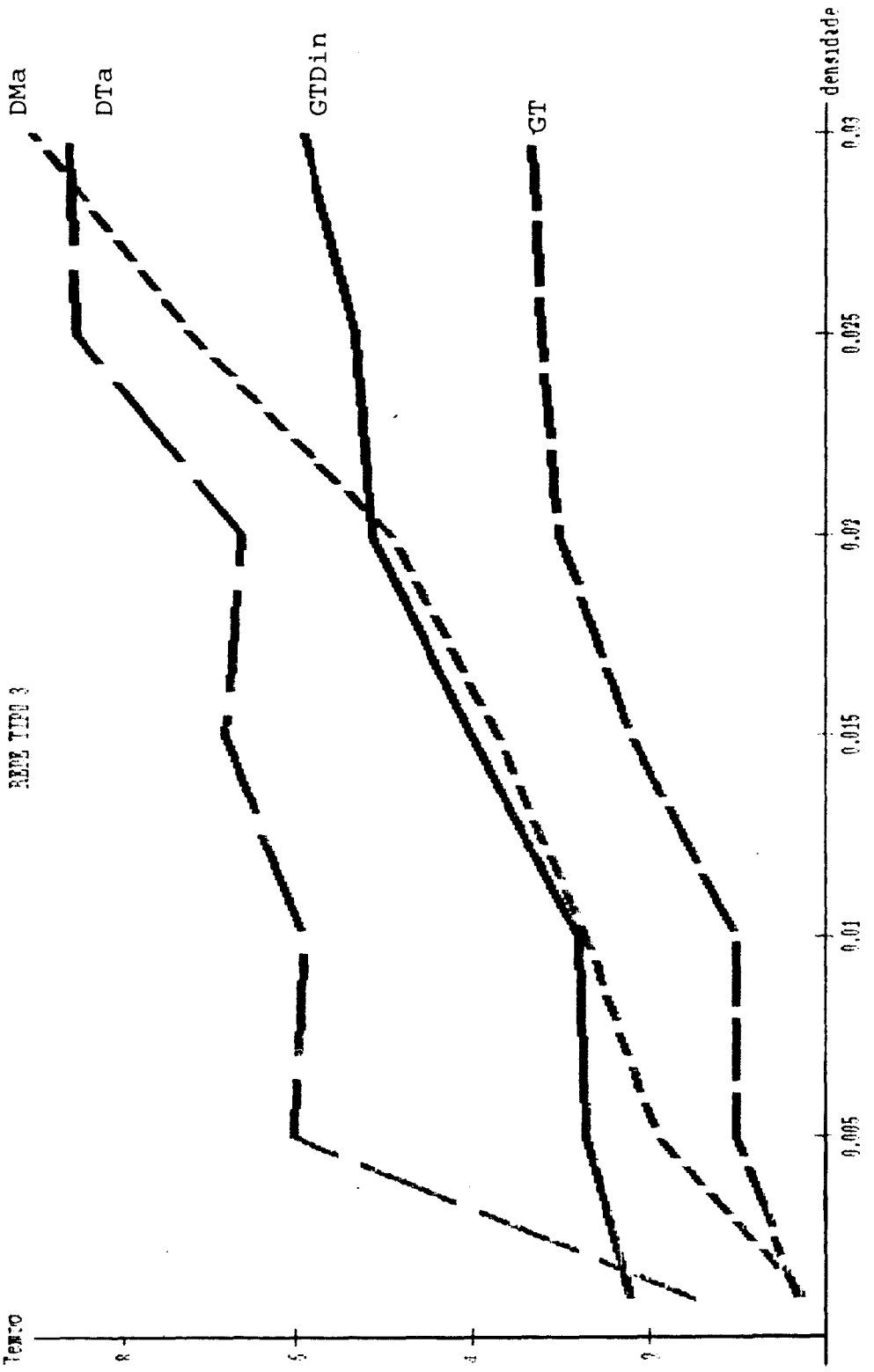

Gráfico 13: Comparação entre os melhores métodos: GTDin*, GT*, DMa e DTa.

Capítulo 4

Conclusões

4.1 Sobre as Redes Tipo 1

Os gráficos mostram que os melhores resultados são dos algoritmos GT e DTa, seguidos pelo algoritmo GTDin. Os demais algoritmos apresentam tempos de CPU muito maiores do que estes, colocando-os em um segundo plano.

Entre GT e DTa, observa-se que o segundo algoritmo tem um melhor desempenho quando a rede possui poucos arcos ou seja, quando se trata de uma rede esparsa. Este fato contraria a teoria e pode ser explicado devido ao rotulamento utilizado no método GT. Neste algoritmo o rotulamento da origem é inicializado com n, logo são necessárias pelo menos n operações Relabel para que o algoritmo termine, independente do número de arcos na rede.

Por outro lado o algoritmo DTa, por ser um representante da classe de Dinic, trabalha encontrando um fluxo bloqueante o que é facilitado conforme a rede fica esparsa.

Em contra-partida, à medida que a rede torna-se mais densa o algoritmo GT tem o melhor desempenho entre todos os algoritmos.

Entre as duas implementações do algoritmo de Goldberg e Tarjan, observa-se que os melhores resultados de tempo são obtidos pela primeira versão, a qual não faz uso da estrutura de dados Árvores Dinâmicas. A partir de $p = 0.2$ até o seu valor máximo, os desempenhos mantêm uma diferença constante. A estrutura de dados neste caso tornou o algoritmo mais demorado,

comprometendo seu desempenho.

4.2 Sobre as Redes Tipo 2

Neste caso os algoritmos GTDin e GT apresentam seu piores resultados, sendo que o uso da estrutura de dados, embora comprometa ainda mais o tempo de execução do algoritmo GTDin, o número de operações Push Saturantes diminui em relação ao algoritmo GT.

Os desempenhos destes métodos podem ser melhorados através de atualizações periódicas do rotulamento. Tais alterações reduziram o tempo de processamento para 1% do tempo gasto anteriormente nos problemas apresentados em [3].

Este tipo de rede é favorável aos métodos DMa e Simplex sendo as redes onde o Simplex tem seu melhor desempenho.

4.3 Sobre as Redes Tipo 3

Um fato curioso é detectado neste tipo de rede pois para alguns problemas os métodos GTDin e GT são os melhores, passando aos piores desempenhos quando mudamos a semente geradora da rede.

Se consideramos apenas os melhores resultados, tais métodos apresentam o mesmo comportamento detectado nas redes do tipo 1, ou seja; para $p \leq 0,001$ seus desempenhos não são muito bons, passando aos melhores tempos quando $p \geq 0,005$.

Este fato é explicado pelas posições dos cortes mínimos, que levam os algoritmos GTDin e GT a situações extremas de desempenho.

4.4 Conclusão Geral

Nas redes tipo 1 e 3 os métodos GTDin e GT são os melhores, sendo que a estrutura de dados utilizada em GTDin compromete seu desempenho. Isto se deve à complexidade da estrutura de dados que torna sua

manutenção demorada. Esta estrutura foi introduzida ao algoritmo inicial com a finalidade de se conseguir uma complexidade melhor do que a anterior. Nestas circunstâncias, espera-se que o algoritmo com a estrutura de dados Árvores Dinâmicas tenha um desempenho superior aos demais algoritmos em casos com redes muito grandes.

Na tentativa de alcançar tal situação foi utilizada uma "workstation" visto que tal equipamento suporta redes de grande porte, no entanto os resultados não foram satisfatórios.

Com base nos testes realizados, e nos tipos de redes, concluímos que para um algoritmo ter um bom desempenho em tempo real é necessário que este possua uma baixa complexidade aliada a uma implementação simples. Os algoritmos com baixas complexidades, quando fazem uso de uma estrutura de dados complexa ou então são de difícil implementação, na prática poderão ter seus desempenhos comprometidos. Neste trabalho temos um exemplo típico onde a primeira versão do algoritmo de Goldberg e Tarjan, com complexidade $O(n^3)$ tem um desempenho médio melhor do que a versão final, embora esta última apresente complexidade de $O(nm \log(n^2/m))$.

Também detectamos que as técnica de "*scaling*" aplicada ao algoritmo EK produz resultados positivos. Esta técnica foi utilizada no algoritmo de Goldberg e Tarjan [2] e embora este novo algoritmo não tenha sido testado neste trabalho, deve apresentar resultados ainda melhores que o algoritmo original [20].

A utilização do referente é outra estratégia que realmente melhora o tempo de processamento, isto foi verificado nos algoritmos DTa e DMa que tiveram resultados melhores que os algoritmos EK e ESca. Estes algoritmos são os que apresentam os melhores resultados, sendo em alguns casos os únicos a competir com GTDin e GT.

4.5 Proposta de Continuação do Trabalho

Neste trabalho foi realizada a implementação das versões seqüenciais do algoritmo de Goldberg e Tarjan. Como continuação nesta mesma linha consideramos a possibilidade de implementar a versão do algoritmo de processamento em paralelo dos nós ativos. Neste sentido alguns trabalhos

já estão sendo realizados como por exemplo Anderson e Setubal [3] que implementaram o algoritmo sem a utilização das árvores dinâmicas. Poucos trabalhos utilizam esta estrutura de dados devido às dificuldades de implementação e também porque o algoritmo sem esta estrutura tem um desempenho muito bom.

Visto que contamos com a estrutura de dados já implementada e extensivamente testada durante este trabalho, ficamos tentados a utilizá-la na versão de processamento paralelo. Tal implementação no entanto requer um equipamento com um grande número de processadores em paralelo pois Anderson e Setubal [3] mostram resultados onde a implementação em paralelo não é tão satisfatória quanto a versão seqüencial. Isto se deve em parte pelo fato destes autores utilizarem poucos processadores; no caso foram utilizados apenas oito processadores em paralelo.

Seguindo outra linha, com a realização deste trabalho verificamos quais são os principais pontos de estrangulamento do algoritmo e a partir daí estamos em condições de discutir maneiras de melhor o algoritmo. Isto pode se dar com a otimização do algoritmo propriamente dito, ou então indiretamente, conseguindo uma estrutura de dados que atenda às necessidades do algoritmo de maneira mais eficiente do que as árvores dinâmicas.

O algoritmo de Goldberg e Tarjan [20] tem certas flexibilidades que podem ser exploradas. Além disto, algumas alterações podem ser efetuadas para se obter uma implementação mais eficiente. O que pode ser feito nesta implementação é:

- Considerar um rotulamento válido inicial de cada nó v como sendo o comprimento do caminho mínimo de v até t. Este rotulamento é denominado "rotulamento exato" pois trabalha com a distância mínima de cada nó até o destino t.

- Durante a execução do algoritmo manter este rotulamento atualizado. Isto pode ser conseguido das seguintes maneiras:

    1. Realizar uma pesquisa em amplitude a partir do nó destino, calculando o tamanho do caminho mínimo via arcos residuais até o destino. Uma segunda pesquisa do mesmo tipo deve ser efetuada agora partindo do nó origem para encontrar os nós que estão disconectados do destino.

2. Ao executar a operação Relabel no nó v, propagar este rotulamento na rede residual, isto é; se existe um nó u com $d(u) = d(v) + 1$ e v é o único sucessor de u com rotulamento $d(v)$ então aplique Relabel ao nó u, e assim por diante.

- A fila Q pode ser uma fila de prioridades. Neste caso pode-se priorizar:
  1. Nó com maior excesso de fluxo.
  2. Nó com maior rotulamento.
- Utilizar uma **pilha** no lugar da fila Q.

Uma outra possibilidade de continuação do trabalho é explorar esta estrutura de dados em outros algoritmos ou então explorar a utilização deste algoritmo já implementado na resolução de outros problemas que utilizam o algoritmo do fluxo máximo como sub-rotina.

Bibliografia

[1] AHUJA, R. K. & ORLIN, J. B. A fast algorithm for the maximum flow problem. Tech. Rep. 1905-87, Sloan School of Management, Massachusetts Institute of Technology, Cambridge, Mass.,1987.

[2] AHUJA, R. K., ORLIN, J. B. & TARJAN, R. E. Improved time bounds for the maximum flow problem. *SIAM J. COMPUT. 18, 5 (1989), 939-954.*

[3] ANDERSON, R. J. & SETUBAL, J. C. Parallel and sequential implementation of maximum - flow algorithms. *Comunicação Privada.*

[4] CAMPANHOLI, S. M. Refinamentos no Método Simplex para Redes. Dissertação de mestrado, Departamento de Matemática Aplicada, IMECC, Unicamp, 1989.

[5] CHERIYAN, J. & MAHESHWARI, S. N. Analysis of preflow push algorithms for maximum network flow. *SIAM J. COMPUT. 18, 6 (1989), 1057-1086.*

[6] CHERKASKY, E. A. An algorithm for constructing maximal flows in networks with complexity of $O(V^2E^{1/2})$ operations. *Math. Methods Solution Econ. Probl. 7 (1977), 112-125. (In Russian.)*

[7] CUNNINGHAM W. H. Theoretical Properties of the Network Simplex Method. *Mathematics of Operations Research, 4, 2 (1979), 196-209.*

[8] DERIGS, U. & MEIER, W. Implementing Goldberg's max - flow - algorithm - A computational investigation. *ZOR - Methods and Models of Operations Research, 33 (1989), 383-403.*

[9] DINIC, E. A. Algorithm for solution of a problem of maximum flow in networks with power estimation. *Sov. Math. Dokl. 11 (1970), 1277-1280.*

[10] EDMONDS, J. & KARP, R. M. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM 19, 2 (1972), 248-264.*

[11] FORD, L. R., Jr. & FULKERSON, D. R. *Flows in Networks.* Princeton University Press, Princeton, N.J., 1962.

[12] FORD, L. R., Jr. & FULKERSON, D. R. Maximal flow through a network. *Can. J. Math. 8 (1956), 399-404.*

[13] FREDMAN, M. L. & TARJAN, R. E. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM 34, 3 (1987), 596-615.*

[14] FULKERSON, D. R. & DANTZIG, G. B. Computation of maximum flow in networks. *Naval Res. Log. Quart. 2, (1955), 277-283.*

[15] GABOW, H. N. Scaling algorithms for network problems. *J. Comput. Syst. Sci. 31 (1985), 148-168.*

[16] GALIL, Z. An $O(V^{5/3}E^{2/3})$ algorithm for the maximal flow problem. *Acta Inf. 14 (1980), 221-242.*

[17] GALIL, Z. & NAAMAD, A. An $O(EV \log^2 V)$ algorithm for the maximal flow problem. *J. Comput. Syst. Sci. 21 (1980), 203-217.*

[18] GOLDBERG, A. V. A new max-flow algorithm. Tech. Rep. MIT/LCS/TM-291, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Mass., 1985.

[19] GOLDBERG, A. V., GRIGORIADIS, M. D. & TARJAN, R. E. Use of dynamic trees in a network Simplex algorithm for the maximum flow problem. *Mathematical Programming 50 (1991) 277-290.*

[20] GOLDBERG, A. V. & TARJAN, R. E. A new approach to the maximum flow problem. *J. ACM 35, 4 (1988), 921-940*

[21] GOLDBERG, A. V. & TARJAN, R. E. Finding minimun cost circulation by sucessive approximation. *Math. of O.R., 15, 3 (1990), 430-466*

[22] KARZANOV, A. V. Determining the maximal flow in a network by the method of preflows. *Sov. Math. Dokl. 15 (1974), 434-437.*

[23] KENNINGTON, J. L. & HELGASON, R. V. *Algorithms for Network Programming.* John Wiley and Sons, New York, 1980.

[24] LAWLER, E. L.*Combinatorial Optimization: Networks and Matroids.* Holt, Rinehart, and Winston, New York, 1976.

[25] MALHOTRA, V. M., PRAMODH,K. M. & MAHESHWARI, S. N. An $O(V^3)$ algorithm for finding maximum flows in networks. *Inf. Process Lett. 7 (1978), 277-278.*

[26] MARTEL, C. A Comparison of Phase and Nonphase Network Flow Algorithms. *Networks 19 (1989), 691-705.*

[27] PAPADIMITRIOU, C. H. & STEIGLITZ, K. *Combinatorial Optimization: Algorithms and Complexity.* Prentice-Hall, Englewood Cliffs, N.J., 1982.

[28] PICARD, J. C. & RATLIFF, H. D. Minimum cuts and related problems. *Networks 5 (1975), 357-370.*

[29] SHILOACH, Y. An $O(nI \log^2 I)$ maximum-flow algorithm. Tech. Rep. STAN-CS-78-802, Computer Science Dept., Stanford. Calif., 1978.

[30] SHILOACH, Y. & VISHKIN, U. An $O(n^2 \log n)$ parallel max-flow algorithm. *J. Algorithms 3 (1982), 128-146.*

[31] SLEATOR, D. D. An $O(nm \log n)$ algorithm for maximum network flow. Tech. Report. STAN-CS-80-831, Computer Science Dept., Stanford Univ., Stanford, Calif., 1980.

[32] SLEATOR, D. D. & TARJAN, R. E. A data structure for dynamic trees. *J. Comput. Syst. Sci. 26 (1983), 362-391.*

[33] SLEATOR, D. D. & TARJAN, R. E. Self-adjusting binary trees. *J. ACM 32, 3 (1985), 652-686.*

[34] TARJAN, R. E. A simple version of Karzanov's blocking flow algorithm. *Oper. Res. Lett. 2 (1984), 265-268.*

[35] VUILLEMIN, J. A data struture for manipulating priority queues. *Commun. ACM 21, 4 (1978), 309-315.*