Rui Pedro Mendes Salgueiro

Um Algoritmo para o Problema do Fluxo Máximo

Aplicação à Partição de Redes

Departamento de Matemática
Faculdade de Ciências e Tecnologia
Universidade de Coimbra

1991

Relatório de Estágio
da
Licenciatura em Matemática
Ramo de Ciências da Computação

Elaborado no âmbito do Projecto 87/187 da JNICT:
"O Problema do Posicionamento no Contexto da Programação Matemática"

Nota Introdutória

Neste trabalho apresenta-se um novo algoritmo para o problema do fluxo máximo, que se baseia na determinação de cadeias incrementais. Deste algoritmo foram programadas várias versões deste algoritmo, e comparadas experimentalmente com o algoritmo simplex, o algoritmo de Goldberg e Tarjan e o algoritmo de Bertsekas e Tseng; este último resolve o problema do fluxo de valor dado e custo mínimo, em que o problema do fluxo máximo se pode transformar.

O algoritmo apresentado revelou-se, em geral, mais eficiente no que respeita ao tempo de execução, embora ocupe um pouco mais de memória do que o algoritmo simplex. Os resultados obtidos experimentalmente são apresentados graficamente.

Baseada no algoritmo, é também apresentada uma heurística, para determinar árvores de partição de redes. Esta heurística é comparada com a de Kerninghan e Lin e a de Almeida e Rodrigues, utilizando-se para isso diversas funções objectivo.

São apresentados, também graficamente, os resultados computacionais obtidos que mostram claramente que esta heurística não é competitiva.

Índice Geral

Capítulo 1

O Problema do Fluxo Máximo

1.1 Fluxo em Redes

1.1.1 Redes

Definição 1.1 *Um grafo orientado é um par* $(\mathcal{N}, \mathcal{A})$ *onde* $\mathcal{N} = \{v_1, \ldots, v_n\}$ *é chamado o conjunto dos vértices (ou nós) e* $\mathcal{A} = \{a_1, \ldots, a_m\} \subset \mathcal{N} \times \mathcal{N}$ *o conjunto dos arcos.*

Uma rede (orientada) é um grafo orientado a cujos arcos se associa um ou mais valores numéricos.

No que se segue, e sem perda de generalidade, supor-se-á que $a_i = (x, y)$, com $x \neq y$,$\forall a_i \in \mathcal{A}$.

Definição 1.2 *Sejam* x_1 *e* x_k *dois quaisquer dados vértices de* $(\mathcal{N}, \mathcal{A})$*. Chama-se cadeia de* x_1 *para* x_k*, e designa-se por* $C(x_1, x_k)$*, a qualquer sequência de vértices e arcos da forma* $[x_1, a_1, x_2, \ldots, a_{k-1}, x_k]$ *onde:*

1. $x_i \in \mathcal{N}$*, para* $i = 1, \ldots, k$*;*
2. $x_i \neq x_j$*, para* $i, j = 1, \ldots, k$*;*
3. $a_i = (x_i, x_{i+1})$ *ou* $a_i = (x_{i+1}, x_i)$*, para* $i = 1, \ldots, k-1$*.*

Quando $a_i = (x_i, x_{i+1})$ *para todo o* $i = 1, \ldots, k-1$*, a* $C(x_1, x_k)$ *chama-se caminho de* x_1 *para* x_k*.*

Usualmente denotar-se-á o caminho $C(x_1, x_k)$ pela sequência $x_1, x_2, \ldots, x_k$, dos seus nós.

Definição 1.3 *Seja $s \in \mathcal{N}$ um qualquer vértice de $(\mathcal{N}, \mathcal{A})$. Uma árvore com raiz em s é um grafo orientado $(\mathcal{N}', \mathcal{A}')$ em que $\mathcal{N}' \subseteq \mathcal{N}$, $\mathcal{A}' \subseteq \mathcal{A}$ e existe uma e uma só cadeia de s para qualquer nó.*

Definição 1.4 *Nível de um dado nó x de uma árvore de raiz s, é o número de arcos que existem na cadeia dessa árvore definida de s para x. Ao máximo dos níveis chama-se altura da árvore.*

Definição 1.5 *Sejam x e y dois nós de uma árvore tais que o arco (x, y) ou o arco (y, x) pertença à árvore. A x chamaremos pai de y e a y filho de x se o nível de y menos o nível de x for igual a 1. Dois nós chamar-se-ão irmãos se forem filhos do mesmo pai.*

Definição 1.6 *Seja x um qualquer vértice de $(\mathcal{N}, \mathcal{A})$. $S(x) = \{y \in \mathcal{N} \mid (x, y) \in \mathcal{A}\}$ é chamado o conjunto dos sucessores de x. Analogamente, $A(x) = \{y \in \mathcal{N} \mid (y, x) \in \mathcal{A}\}$ é o conjunto dos antecessores de x.*

1.1.2 Fluxos

A cada arco $(i, j) \in \mathcal{A}$ associe-se um valor $c(i, j) \in \mathbb{R}^+$ (denominado capacidade do arco (i, j)) e sejam s e t dois distintos e dados nós de $\mathcal{N}$.

No resto do capítulo, $(\mathcal{N}, \mathcal{A})$representará portanto uma rede (orientada).

Definição 1.7 *Um fluxo de s para t em $(\mathcal{N}, \mathcal{A})$ de valor v, é uma função $f : \mathcal{A} \longrightarrow \mathbb{R}_0^+$ que satisfaz as n equações lineares:*

$$\sum_{y \in S(x)} f(x, y) - \sum_{y \in A(x)} f(y, x) = \begin{cases} v & se\ x = s, \\ 0 & se\ x \neq s, t \\ -v & se\ x = t, \end{cases} \ , \forall x \in \mathcal{N}, \tag{1.1}$$

e as m inequações:

$$0 \leq f(x, y) \leq c(x, y),\ \forall (x, y) \in \mathcal{A}. \tag{1.2}$$

O vértice s será designado por fonte, t por destino e os restantes por vértices intermédios.

Note-se que, qualquer que seja a rede $(\mathcal{N}, \mathcal{A})$, existe sempre um fluxo; de facto, se $f(x, y) = 0$, $\forall (x, y) \in \mathcal{A}$ obtêm-se um fluxo (de valor $v = 0$) que verifica 1.1 e 1.2.

Um problema clássico de Optimização em Redes, é um programa linear que consiste na maximização de $v(f)$ sujeito às restrições 1.1 e 1.2. É o chamado **Problema do Fluxo Máximo**.

1.1.3 Notação

Sejam X e Y dois subconjuntos de $\mathcal{N}$; designa-se por (X, Y) o conjunto

$$(X, Y) = \{(x, y) \mid x \in X, y \in Y, (x, y) \in \mathcal{A}\}.$$

Qualquer que seja a função $g : \mathcal{A} \longrightarrow \mathrm{I\!R}$, denotar-se-á por $g(X, Y)$ a soma

$$g(X, Y) = \sum_{(x,y) \in (X,Y)} g(x, y).$$

Analogamente para qualquer função $h : \mathcal{N} \longrightarrow \mathrm{I\!R}$,

$$h(X) = \sum_{x \in X} h(x).$$

No caso de um conjunto ser singular, representar-se-á apenas pelo seu elemento.

Sejam X, Y, $Z \in \mathcal{N}$. Então

$$g(X, Y \cup Z) = g(X, Y) + g(X, Z) - g(X, Y \cap Z)$$

e

$$g(Y \cup Z, X) = g(Y, X) + g(Z, X) - g(Y \cap Z, X).$$

Se nas duas últimas igualdades, Y e Z são disjuntos, então:

$$g(X, Y \cup Z) = g(X, Y) + g(X, Z)$$

e

$$g(Y \cup Z, X) = g(Y, X) + g(Z, X).$$

Note-se que

$$(A(x), x) = (\mathcal{N}, x)$$

e

$$(x, S(x)) = (x, \mathcal{N}).$$

De igual modo,

$$g(\mathcal{N}, X) = \sum_{x \in X} g(\mathcal{N}, x) = \sum_{x \in X} g(A(x), x)$$

e

$$g(X, \mathcal{N}) = \sum_{x \in X} g(x, \mathcal{N}) = \sum_{x \in X} g(x, S(x)).$$

1.1.4 Cortes

Definição 1.8 *Sejam s e t dois quaisquer nós de $(\mathcal{N}, \mathcal{A})$. Um corte para s e t em $(\mathcal{N}, \mathcal{A})$ é um conjunto de arcos $(X, \bar{X})$ tais que $s \in X$ e $t \in \bar{X}$, onde $\bar{X}$ designa o complementar de X em $\mathcal{N}$.*

A capacidade do corte $(X, \bar{X})$ é $c(X, \bar{X})$, ou seja, é a soma das capacidades dos seus arcos.

Lema 1.9 *Qualquer caminho de s para t em $(\mathcal{N}, \mathcal{A})$ contém pelo menos um arco de qualquer corte $(X, \bar{X})$ para s e t.*

Demonstração: Seja $x_1, x_2, \ldots, x_n$ um caminho com $x_1 = s$, $x_n = t$. Como, pela definição de corte, $x_1 \in X$ e $x_n \in \bar{X}$, existe x_i $(1 \leq i < n)$ com $x_i \in X$, $x_{i+1} \in \bar{X}$. Portanto o arco (x_i, x_{i+1}) é um elemento do corte $(X, \bar{X})$. ∎

Lema 1.10 *Seja f um fluxo de s para t em $(\mathcal{N}, \mathcal{A})$ de valor v e seja $(X, \bar{X})$ um qualquer corte para s e t. Então:*

$$v = f(X, \bar{X}) - f(\bar{X}, X) \leq c(X, \bar{X}).$$

Demonstração: Como f é um fluxo, então f satisfaz as equações:

$$\begin{aligned} &f(s, \mathcal{N}) - f(\mathcal{N}, s) = v, \\ &f(x, \mathcal{N}) - f(\mathcal{N}, x) = 0 \qquad (x \neq s, t), \\ &f(t, \mathcal{N}) - f(\mathcal{N}, t) = -v. \end{aligned}$$

Dado que $(X, \bar{X})$ é um corte, então $x \in X$ e $t \in \bar{X}$. Por conseguinte, somando estas equações para todo o $x \in X$, obter-se-á:

$$\sum_{x \in X} (f(x, \mathcal{N}) - f(\mathcal{N}, x)) = v;$$

ou seja,

$$f(X, \mathcal{N}) - f(\mathcal{N}, X) = v.$$

E como $\mathcal{N} = X \cup \bar{X}$, então

$$f(X, X \cup \bar{X}) - f(X \cup \bar{X}, X) = v,$$

ou seja,

$$f(X, X) + f(X, \bar{X}) - f(X, X) - f(\bar{X}, X) = v,$$

donde se conclui que
$$f(X,\bar{X}) - f(\bar{X},X) = v,$$
verificando-se, portanto, a igualdade do enunciado.

Como $0 \leq f(x,y) \leq c(x,y)$ se verifica para todo o arco $(x,y) \in \mathcal{A}$, é imediato que $f(\bar{X},X) \geq 0$ e $f(X,\bar{X}) \leq c(X,\bar{X})$; logo $v = f(X,\bar{X}) - f(\bar{X},X) \leq c(X,\bar{X})$, como queríamos demonstrar. ∎

Definição 1.11 *Seja $x_1,\ldots,x_n$ uma qualquer cadeia de x_1 para x_n em $(\mathcal{N},\mathcal{A})$. Se $(x_i,x_{i+1}) \in \mathcal{A}$, então (x_i,x_{i+1}) diz-se que é percorrido no sentido directo; se $(x_{i+1},x_i) \in \mathcal{A}$, então (x_{i+1},x_i) é percorrido no sentido inverso.*

1.1.5 O teorema do fluxo máximo - corte mínimo

Um outro problema clássico de Optimização em Redes consiste na determinação de um corte de capacidade mínima no conjunto de todos os cortes para um dado par de nós s e t. O teorema que se segue, devido a Ford e Fulkerson [8], relaciona este problema com o Problema do Fluxo Máximo.

Teorema 1.12 *(Teorema do fluxo máximo - corte mínimo) Numa qualquer rede, o valor máximo do fluxo de s para t, é igual ao valor mínimo da capacidade de todos os cortes para s e t.*

Demonstração: Utilizando o lema anterior, basta mostrar a existência de um fluxo f e de um corte $(X,\bar{X})$, tal que o valor do fluxo seja igual à capacidade do corte. Para isso partimos de um fluxo máximo f (que obviamente existe) e definimos um corte $(X,\bar{X})$ tal que
$$f(X,\bar{X}) = c(X,\bar{X}) \text{ e } f(\bar{X},X) = 0.$$
Logo, dado que $v = f(X,\bar{X}) - f(\bar{X},X)$ (Lema 1.10), obtemos:
$$v = f(X,\bar{X}) - f(\bar{X},X) = c(X,\bar{X}).$$
Seja f um fluxo máximo. Definimos o conjunto X, recursivamente, da seguinte maneira:

1. $s \in X$,

2. se $x \in X$ e $f(x,y) < c(x,y)$, então $y \in X$,

3. se $x \in X$ e $f(y,x) > 0$, então $y \in X$.

Concluímos que $t \in \bar{X}$, pois caso contrário $t \in X$ e pela definição de X existe uma cadeia de s para t

$$s = x_1, x_2, \ldots, x_n = t$$

tal que, para todos os arcos (x_i, x_{i+1}) da cadeia percorridos no sentido directo se tem que

$$f(x_i, x_{i+1}) < c(x_i, x_{i+1})$$

e para todos os arcos (x_i, x_{i+1}) percorridos no sentido inverso

$$f(x_{i+1}, x_i) > 0.$$

Seja ε_1 o mínimo de $c(x,y) - f(x,y)$ para todos os arcos (x,y) da cadeia percorridos no sentido directo, ε_2 o mínimo de $f(x,y)$ para todos os arcos (x,y) da cadeia percorridos no sentido inverso, e seja $\varepsilon = min(\varepsilon_1, \varepsilon_2) > 0$. Se incrementarmos f de ε em todos os arcos percorridos no sentido directo e decrementarmos f de ε em todos os arcos percorridos no sentido inverso, obtemos um fluxo de s para t com valor $v + \varepsilon$. Portanto f não é máximo. Desta maneira supondo que $t \in X$ chegámos a uma contradição com a nossa hipótese de partida. Portanto $t \in \bar{X}$.

Demonstrámos que $x \in X$ e $t \in \bar{X}$. Logo $(X, \bar{X})$ é um corte para s e t. Da definição de X concluímos que

$$f(x, \bar{x}) = c(x, \bar{x}) \text{ para } (x, \bar{x}) \in (X, \bar{X}),$$

e

$$f(\bar{x}, x) = 0 \text{ para } (\bar{x}, x) \in (\bar{X}, X),$$

pois caso contrário $\bar{x}$ pertenceria a X. Portanto

$$f(X, \bar{X}) = c(X, \bar{X}),\ \ f(\bar{X}, X) = 0,$$

o que termina a demonstração. ■

Chamaremos *cadeia incremental* (de s para t) (em relação a um fluxo f), a uma cadeia de s para t em que $f(x,y) < c(x,y)$ nos arcos (x,y) percorridos no sentido directo, e $f(x,y) > 0$ nos arcos (x,y) percorridos no sentido inverso. O valor ε calculado atrás é a *capacidade* da cadeia incremental. É evidente, da definição de cadeia incremental, que a sua capacidade é sempre positiva.

Analogamente chamaremos *caminho incremental* a uma cadeia incremental em que todos os arcos são percorridos no sentido directo.

Corolário 1.13 *Um fluxo f é máximo se e só se não existe uma cadeia incremental em relação a f.*

Demonstração: Se f é máximo, é óbvio que não existe cadeia incremental. Vamos então supor que não existe cadeia incremental. Nesse caso o conjunto X definido a partir de f tal como na demonstração do teorema anterior não pode conter o nó destino, t. Logo, tal como na demonstração do teorema anterior, $(X, \bar{X})$ é um corte para s e t, de capacidade igual ao valor de f. Portanto f é máximo. ∎

1.1.6 Vários nós fonte e vários nós destino

Até agora temos assumido que a rede tem um só nó fonte e um só nó destino. É possível reduzir o caso mais geral, em que existem vários nós fonte e vários nós destino (com fluxo permitido de qualquer fonte para qualquer destino) ao caso de uma única fonte e um único destino.

Suponhamos então que $\mathcal{N} = \mathcal{S} \cup \mathcal{T} \cup \mathcal{R}$ tal que $\mathcal{S} \cap \mathcal{T} = \emptyset$, $\mathcal{S} \cap \mathcal{R} = \emptyset$, e $\mathcal{T} \cap \mathcal{R} = \emptyset$. A $\mathcal{S}$ chamar-se-á conjunto dos nós fonte, a $\mathcal{T}$ conjunto dos nós destino, sendo $\mathcal{R}$ conjunto dos nós de passagem. Admitindo que o fluxo pode ser "enviado" de um qualquer nó $s_i \in \mathcal{S}$ para um qualquer $t_j \in \mathcal{T}$, definimos um fluxo de $\mathcal{S}$ para $\mathcal{T}$ como sendo uma função real de domínio $\mathcal{A}$ que satisfaz as equações

$$\begin{array}{ll} f(x, \mathcal{N}) - f(\mathcal{N}, x) = 0 & \text{para } x \in \mathcal{R}, \\ 0 \leq f(x, y) \leq c(x, y) & \text{para } (x, y) \in \mathcal{A}, \end{array}$$

sendo o valor do fluxo

$$v = f(\mathcal{S}, \mathcal{N}) - f(\mathcal{N}, \mathcal{S}).$$

Expandimos a rede $(\mathcal{N}, \mathcal{A})$ para uma rede $(\mathcal{N}^*, \mathcal{A}^*)$ juntando dois nós s^*, t^* e os arcos $(s^*, \mathcal{S})$, $(\mathcal{T}, t^*)$. Associamos a cada arco da rede $(\mathcal{N}^*, \mathcal{A}^*)$ um valor $c^*(i, j) \in \mathbb{R}^+ \cup \{\infty\}$ definido por

$$\begin{array}{ll} c^*(s^*, x) = \infty & \text{para } x \in \mathcal{S}, \\ c^*(x, t^*) = \infty & \text{para } x \in \mathcal{T}, \\ c^*(x, y) = c(x, y) & \text{para } (x, y) \in \mathcal{A}. \end{array}$$

Se f^* é um fluxo de s^* para t^* em $(\mathcal{N}^*, \mathcal{A}^*)$ a sua restrição a $\mathcal{A}$ é um fluxo de $\mathcal{S}$ para $\mathcal{T}$ em $(\mathcal{N}, \mathcal{A})$. No sentido oposto podemos estender um fluxo f de $\mathcal{S}$ para $\mathcal{T}$ em $(\mathcal{N}, \mathcal{A})$ para um fluxo f^* de s^* para t^* em $(\mathcal{N}^*, \mathcal{A}^*)$ definindo

$$\begin{array}{ll} f^*(s^*, x) = f(x, \mathcal{N}) - f(\mathcal{N}, x), & x \in \mathcal{S}, \\ f^*(x, t^*) = f(\mathcal{N}, x) - f(x, \mathcal{N}), & x \in \mathcal{T}, \\ f^*(x, y) = f(x, y), & \text{nos outros casos.} \end{array}$$

Conclui-se pois que o problema do fluxo máximo com vários nós fonte e vários nós destino é equivalente a um problema do fluxo máximo com uma única fonte e um único destino na rede expandida.

Obviamente que no problema do Fluxo Máximo em causa, continua a pretender-se maximizar o valor do fluxo, no seu domínio de definição.

Um corte numa rede com vários nós fonte $\mathcal{S}$ e vários nós destino $\mathcal{T}$ é $(X, \bar{X})$ tal que $\mathcal{S} \subseteq X$ e $\mathcal{T} \subseteq \bar{X}$. Quer o teorema do fluxo máximo - corte mínimo quer o seu corolário continuam válidos, o que pode ser deduzido a partir do problema na rede expandida ou modificando ligeiramente as demonstrações.

1.2 Algoritmos de Rotulação

A demonstração do teorema do fluxo máximo - corte mínimo permite-nos definir uma classe de algoritmos que determinam, em simultâneo, um fluxo máximo e um corte mínimo numa rede. Estes algoritmos partem de um fluxo de valor zero e incrementam-no sucessivamente da seguinte maneira:

1. Constrói-se o conjunto X segundo a regra recursiva utilizada na demonstração do teorema.

2. Se durante essa construção o nó t é incluído no conjunto X, então existe uma cadeia incremental de s para t de capacidade ε. Seja (x, y) um arco da cadeia incremental:

   - se (x, y) é percorrido no sentido directo, adicione-se ε a $f(x, y)$,
   - se (x, y) é percorrido no sentido inverso, subtraia-se ε a $f(x, y)$.

   Deste modo incrementa-se de ε o valor do fluxo.

3. Se o conjunto X obtido não incluir t, então o fluxo f já é máximo, $(X, \bar{X})$ é um corte minímo e o algoritmo termina.

Para garantir que o algoritmo é finito pode-se impôr a restrição de que os valores das capacidades dos arcos têm valores inteiros. Esta restrição não é muito limitativa, pois se os valores das capacidades forem racionais, pode-se transformar o problema num equivalente multiplicando as capacidades dos arcos por um inteiro conveniente. Em termos computacionais não se põe o problema de capacidades irracionais pois num computador só é possível a representação de números racionais.

Esta restrição garante que o algoritmo termina porque:

- Cada cadeia incremental tem uma capacidade inteira positiva por ser o minímo de valores inteiros positivos.
- Como o fluxo começa por ser inteiro (zero) e, em cada iteração, é incrementado de um inteiro positivo, todos os fluxos que se definem na rede são inteiros.
- Cada incremento é pelo menos uma unidade, logo, como o valor do fluxo máximo é finito, o algoritmo é finito.

Note-se que, quando as capacidades dos arcos são inteiras, o número máximo de cadeias incrementais que é necessário determinar é o próprio valor do fluxo; no problema com capacidades racionais o número máximo de cadeias incrementais é proporcional à precisão com que são representadas as capacidades.

Alternativamente, como Edmonds e Karp [6] demonstraram, se se determinarem apenas cadeias incrementais com um número minímo de arcos, no máximo é necessário determinar $\frac{1}{4}(n^3 - n)$ cadeias, independentemente dos valores das capacidades dos arcos.

1.2.1 Avaliação de algoritmos

A maneira mais correcta, rigorosa e fiável de avaliar um algoritmo é calcular a complexidade para o caso médio. No entanto para o problema do fluxo máximo este cálculo é tão complicado que, o que geralmente se calcula, é a complexidade para o pior caso.

No entanto a complexidade para o pior caso não é uma boa indicação do comportamento no caso médio. Um exemplo bem conhecido é o algoritmo simplex para programas lineares, que tem uma complexidade factorial para o pior caso mas que na prática se comporta tão bem que continua a ser utilizado apesar da existência de algoritmos polinomiais para o mesmo problema.

Este facto levou a uma outra forma de avaliação dos algoritmos que consiste na realização de testes computacionais em problemas o mais representativos possível de casos reais. No nosso caso, para que os resultados obtidos fossem mais fiáveis, foram utilizados diversos geradores de redes. Com estes geradores obtêm-se:

- Redes sem estrutura (aleatórias), onde se impediu a existência de arcos do tipo (x, x). Nestas redes, dado um nó x, a probabilidade da existência de um arco (x, y) é igual à probabilidade da existência de um arco (x, z), quaisquer que sejam os nós y e z, com $x \neq y, z$.

  Na geração destas redes são apenas dados o número de nós e a densidade (quociente entre o número de arcos e o número de nós).

- Semelhante ao caso anterior mas gera redes com vários nós fonte e/ou vários nós destino. Os nós fonte e os nós destino são escolhidos ao acaso entre todos os possíveis, com igual probabilidade para todos os nós. Obviamente que a rede é gerada por forma a que os conjuntos dos nós fonte e dos nós destino sejam disjuntos.

  Para além do número de nós e da densidade, são também dados o número de nós fonte e de nós destino.

- Redes tipo grelha. São redes rectangulares onde o conjunto dos nós é disposto em linhas e colunas paralelas, e onde se x e y são dois nós da mesma linha e de colunas adjacentes, ou da mesma coluna e de linhas adjacentes, então existe o arco (x, y). Estas redes representam, por exemplo, as ruas de uma cidade do tipo de Manhattan. Neste tipo de redes, todos os nós da primeira coluna são nós fonte, e todos os nós da última coluna são nós destino.

  Na geração de grelhas são apenas dados o número de linhas e o número de colunas.

- Redes "difíceis". Nestas redes cada cadeia incremental tem capacidade igual a um, sendo o valor do fluxo máximo igual a $\frac{n^2}{4}$ (em que n é o número de nós da rede). Logo, os algoritmos que calculam cadeias incrementais têm de calcular $\frac{n^2}{4}$ cadeias.

Para gerar estas redes o único dado necessário é o número n de nós (que tem de ser par para as capacidades serem inteiras). Ao contrário dos outros três tipos de rede as capacidades dos arcos não são aleatórias, mas sim determinadas pela fórmula:

$$c(x,y) = \begin{cases} 1 & \text{se } v > u+1, \\ (u - \frac{\lfloor n \rfloor}{2})^2 & \text{se } v = u+1, \end{cases}$$

em que u e v são os números de ordem, respectivamente, dos nós x e y, não existem mais arcos, o nó fonte é o de ordem 1 e o nó destino é o de ordem n.

Estes tipos de rede foram sugeridos pelo trabalho de Glover et al [9].

1.2.2 O algoritmo de Ford e Fulkerson

Depois de começar com um fluxo de valor zero o algoritmo executa sucessivamente duas rotinas: *Rotular* e *Passar_fluxo*. A primeira constrói o conjunto X referido anteriormente. Se este conjunto incluir o vértice t, a segunda rotina é chamada para aumentar o fluxo; caso contrário, o algoritmo termina. A rotina *Rotular* atribui aos nós y que vão sendo incluídos em X um rótulo da forma $(x, \pm, \varepsilon)$, onde $x \in \mathcal{N}$ e ε é um inteiro positivo ou $+\infty$. Note-se que a inclusão de y em X significa que existe uma cadeia incremental de s para y; a capacidade dessa cadeia incremental é ε, x é o nó que antecede y nessa cadeia e será chamado *antecessor* de y. O símbolo $\pm$ especifica se estamos a utilizar o arco (x, y) no sentido directo ou o arco (y, x) no sentido inverso. Durante a execução desta rotina, um nó pode estar num de três estados: não rotulado, rotulado e não visitado, ou rotulado e visitado.

Rotular

1. Marcam-se todos os nós como não rotulados, excepto o nó fonte s o qual fica rotulado com $(—, —, +\infty)$ e na situação de não visitado.
2. **Repetir**

   Seleccione-se x, um qualquer nó rotulado e não visitado. Admita-se que o seu rótulo é $(z, \pm, \varepsilon(x))$:

(a) Para todos os nós y tais que $(x, y) \in \mathcal{A}$, $f(x, y) < c(x, y)$ e que não estão rotulados, atribua-se o rótulo $(x, +, \varepsilon(y))$, em que

$$\varepsilon(y) = min[\varepsilon(x), c(x, y) - f(x, y)].$$

(b) Para todos os nós y tais que $(y, x) \in \mathcal{A}$, $f(y, x) > 0$ e que não estão rotulados, atribua-se o rótulo $(x, -, \varepsilon(y))$, em que

$$\varepsilon(y) = min[\varepsilon(x), f(y, x)].$$

Estes nós y estão agora rotulados e não visitados e x passa à situação de rotulado e visitado.

Até que o nó destino t seja rotulado ou não existam nós rotulados e não visitados.

Passar_fluxo

1. No início desta rotina, t está rotulado com $(x, \pm, \varepsilon(t))$. Seja y uma variável que pode assumir qualquer valor no conjunto dos nós $\mathcal{N}$. Atribua-se t a y.
2. **Enquanto** o rótulo de y for diferente de $(—, —, +\infty)$, ou seja, enquanto não se atinge o nó fonte:

   executa-se
   - Se o rótulo de y é $(x, +, \varepsilon(y))$ incrementa-se $f(x, y)$ de $\varepsilon(t)$.
   - Se o rótulo de y é $(x, -, \varepsilon(y))$ decrementa-se $f(y, x)$ de $\varepsilon(t)$.
   - Atribui-se x a y.

Com estas duas rotinas pode descrever-se facilmente o algoritmo de Ford e Fulkerson:

Enquanto com a execução de *Rotular* se conseguir rotular t

executa-se a rotina *Passar_fluxo*.

Nesta descrição do algoritmo não é especificado qual dos nós rotulados e não visitados é escolhido para, posteriormente, passar à situação de visitado. Usualmente, de cada vez que um nó é rotulado, é colocado na cauda de uma lista e o nó que se selecciona para ser visitado é o que está na cabeça desta mesma lista. Desta maneira são determinadas as cadeias incrementais com menor número de arcos, o que, como vimos atrás, garante que o algoritmo é finito.

1.2.3 Novo algoritmo

Os algoritmos que neste trabalho se apresentam, são obtidos a partir do algoritmo de Ford e Fulkerson, por aplicação sucessiva de quatro melhoramentos. Sendo estes quatro melhoramentos compatíveis, são utilizados conjuntamente nas versões finais do algoritmo proposto.

1. A versão clássica do algoritmo de Ford e Fulkerson determina sucessivas cadeias incrementais. No entanto, experimentalmente concluiu-se que, na grande maioria dos casos, é possível determinar o fluxo máximo usando apenas caminhos incrementais.

   O algoritmo é então dividido em duas fases:

   - Na primeira fase a rotina *Rotular* executa apenas o passo 2a e não o 2b; isto é, só se determinam caminhos incrementais, o que é cerca de duas vezes mais rápido do que determinar cadeias. De facto, dado que todos os arcos serão percorridos no sentido directo, só é preciso percorrer o conjunto dos arcos que saiem de cada nó e não o conjunto dos que chegam a esse mesmo nó. Poupa-se ainda tempo, não especificando no rótulo de cada nó se o arco que o liga ao antecessor é percorrido no sentido directo ou inverso. Pela mesma razão, a rotina *Passar_fluxo* também fica mais simples e mais rápida.
   - A segunda fase é similar à versão clássica, sendo executada apenas quando na primeira fase não se consegue determinar um caminho incremental de s para t. No entanto, aproveita-se o trabalho já feito na primeira fase começando-se com o conjunto X nela determinado, isto é, com o conjunto dos nós para os quais existe caminho incremental a partir de s.

Como é evidente, este melhoramento só é aplicável em redes dirigidas.

2. Os algoritmos estudados até aqui geram uma árvore de raiz em s, cujo conjunto dos nós é X. É óbvio que é possível construir um algoritmo simétrico que gere uma árvore com raiz em t e cujo conjunto dos nós é $\bar{X}$.

   Admita-se que num dado momento a cadeia incremental com menor número de nós tem k arcos. Para a determinar, o algoritmo clássico necessita de rotular os nós de uma árvore com altura k.

   Se construirmos duas árvores, uma com raiz em s e outra com raiz em t, de modo que a diferença das suas alturas seja zero ou um, obteremos a cadeia incremental imediatamente após a sua intersecção. Nesta altura a soma das suas alturas é igual a k. Desta forma, para determinar a cadeia incremental, rotulamos apenas os nós destas árvores, cujo número, em geral, é bastante menor do que o número de nós da árvore única que resulta do algoritmo clássico [1].

3. No item anterior conclui-se que se rotulam muitos mais nós do que os que constituem a cadeia incremental. Tal facto levou-nos a retirar do rótulo de cada nó x o valor da capacidade da cadeia incremental de s para x (ou de x para t, quando estamos na árvore com raiz em t). Por este motivo a capacidade da cadeia incremental só pode ser calculada após a sua determinação e os arcos da cadeia incremental têm de ser percorridos duas vezes: uma para calcular a sua capacidade e outra para alterar o fluxo.

4. O último melhoramento aproveita parte da informação obtida no cálculo de uma cadeia incremental para acelerar a determinação da cadeia incremental seguinte, ou para concluir da sua não existência. Depois de encontrada uma cadeia incremental e de actualizado o fluxo, há pelo menos um arco da cadeia que fica saturado, isto é, cujo fluxo fica igual à capacidade se era percorrido no sentido directo, ou com fluxo a zero se era percorrido no sentido inverso. A um tal arco chamar-se-á bloqueador. É óbvio que numa cadeia incremental pode existir mais de

[1] Note-se que o número de nós de uma árvore aumenta, em geral, de modo exponencial com a sua altura.

que um arco bloqueador. No caso de existir mais do que um na mesma árvore interessar-nos-á determinar o que está mais próximo da raiz.

Para cada árvore que contém arco bloqueador, retira-se a sub-árvore com raiz no nó que está no extremo desse arco mais afastado da raiz, pois, depois da actualização do fluxo, deixámos de ter determinada uma cadeia incremental de s para esses nós ou/e de esses nós para t. É claro que para os nós não retirados, continuamos a conhecer uma cadeia incremental em relação ao fluxo actualizado, sendo esta informação utilizada como ponto de partida para a tentativa de determinação da cadeia incremental seguinte.

Repare-se que sem a utilização do melhoramento anterior, este procedimento estaria incorrecto. De facto, para quase todos os nós x da árvore que contém o arco bloqueador a capacidade da cadeia incremental de s para x ou/e de x para t é alterada quando o fluxo é alterado. Logo ter-se-ia de actualizar o ε dos rótulos com uma regra muito mais complexa e envolvendo um muito maior número de nós. Note-se também que o segundo melhoramento aumenta a eficiência deste ao reduzir a altura das árvores.

Estes dois últimos melhoramentos são também utilizados por Glover et al, [9].

1.2.4 Estruturas de dados

Na implementação do algoritmo foram usadas as seguintes estruturas de dados:

1. Rede (estrela directa):
   - Uma rede é representada por uma estrutura contendo um vector de nós [2] e dois inteiros n e m que representam respectivamente o número de nós e o número de arcos.
   - Cada nó x é representado por um estrutura contendo um vector de arcos e um inteiro que representa o número dos elementos do

[2] Cada nó é identificado, inequivocamente, pelo índice do vector. Para simplificar a notação referir-nos-emos muitas vezes a um nó pelo seu índice.

vector. Os elementos deste vector são todos os arcos que se iniciam em x.

- Um arco é representado por uma estrutura com dois campos: o primeiro indica o nó ao qual o arco chega (o nó de onde o arco sai está implícito do item anterior); o segundo contém as características do arco.
- As características do arco são dois reais [3] que correspondem ao seu fluxo e capacidade, ou seja, para o arco (x, y) são $f(x, y)$ e $c(x, y)$.

Esta estrutura [4] ocupa $16 + n \times 8 + m \times 12$ bytes.

2. Rede completa. Esta estrutura inclui a anterior, juntando-se-lhe um vector de nós inversos.

   - Um nó inverso é uma estrutura contendo um vector de arcos inversos e um inteiro que representa o número dos elementos do vector. Os elementos deste vector são todos os arcos que terminam nesse nó.
   - Um arco inverso é uma estrutura com dois campos: o primeiro indica o nó do qual o arco sai (o nó ao qual o arco chega está implícito do item anterior); o segundo contém um ponteiro para as características do arco.

     Como as características do arco são partilhadas pelas duas representações, daqui resulta que uma alteração, por exemplo, do fluxo, pode ser feita apenas numa delas, ficando acessível para ambas. Desta forma obtém-se uma maior eficiência quer em tempo, quer em espaço.

   Esta estrutura ocupa $24 + n \times 16 + m \times 20$ bytes.

3. Lista (simples). Ao longo do algoritmo é necessária uma ou mais listas de nós (consoante a versão do algoritmo). Esta lista é representada

[3] Note-se que no computador um "real" é um número racional ou um de três valores especiais que representam $+\infty$, $-\infty$ e indeterminado.

[4] Quando nos referirmos ao espaço ocupado por uma estrutura de dados, estamos a considerar a utilização de uma máquina de 32 bits.

por um vector em que o índice de cada elemento é um nó e o valor na posição i do vector é o nó que se segue a i na lista. Pode-se utilizar esta estrutura pois sabemos qual é o número máximo de elementos na lista (igual ao número de nós na rede) por não haver repetições de nós na lista. Em relação à hipótese alternativa da utilização de ponteiros têm-se as seguintes vantagens:

(a) Dado um nó, pode-se saber de imediato qual o nó que se lhe segue na lista sem necessidade de a pesquisar exaustivamente.

(b) Como toda a memória necessária é pedida de uma só vez, acrescentar elementos à lista é consideravelmente mais rápido por não ter o custo de chamar a função de atribuição de memória para cada elemento.

(c) Cada elemento na lista gasta metade do espaço. Em contrapartida é preciso reservar de início o espaço máximo que pode ser necessário.

4. Lista dupla. Esta é outra representação de uma lista em que se utiliza um vector como na estrutura anterior e um outro vector, que em vez de indicar o nó que se lhe segue na lista, indica o que o precede. A vantagem desta representação é a facilidade de retirar elementos.

5. Árvore. Dado que o antecessor de um nó numa cadeia incremental é o seu pai numa árvore, esta pode ser representada usando apenas os rótulos de cada nó. Portanto representamos uma árvore como um vector de rótulos cujos índices são nós. O rótulo do nó x é representado por uma estrutura contendo os campos:

   (a) Um inteiro que representa o pai de x, isto é, o nó que antecede x na cadeia incremental.

   (b) Um valor lógico que indica o sentido em que é percorrido o arco da cadeia incremental que liga x ao seu pai.

   (c) Um campo que pode assumir três valores que indicam se x está na árvore com raiz em s, na árvore com raiz em t ou ainda não foi rotulado.

   (d) Um ponteiro para as características do arco da árvore que liga x ao seu pai.

Esta estrutura ocupa $16 \times n$ bytes, mas poderia ocupar só $12 \times n$ se os campos (b) e (c) fossem compactados numa "word", à custa de uma ligeira penalização em tempo. Poderíamos igualmente retirar o campo (d), mas aqui a penalização em tempo já seria excessiva, pois seria necessário fazer frequentes pesquisas exaustivas no conjunto dos arcos que sai de um dado nó.

Note-se que nesta árvore só existe informação àcerca do pai de um nó e não dos seus filhos.

6. Árvore completa. Esta estrutura acrescenta à anterior um ponteiro para um dos filhos de cada nó e outro para um dos seus irmãos. Mais exactamente, todos os filhos de um dado nó estão numa lista ligada, sendo o primeiro da lista apontado pelo pai.

   Como é evidente esta estrutura resolve o principal problema da anterior. Tem a desvantagem de ser mais complicado fazer alterações, tornando-se estas mais lentas. Ocupa também mais espaço memória (exactamente mais $8 \times n$ bytes).

Note-se que todos os vectores são "alocados" dinamicamente, isto é, o espaço de memória que ocupam é pedido ao sistema na quantidade necessária e apenas quando é preciso. Esta foi uma das mais importantes vantagens da escolha da linguagem de programação *C*.

1.2.5 Implementação

No decurso deste trabalho foram programadas várias versões do algoritmo. Por experimentalmente se terem revelado as mais eficientes, descreveremos apenas três versões que utilizam todos os melhoramentos acima referidos. Estas versões, numeradas sete, oito e onze, variam entre si pelas estruturas de dados usadas para representar as árvores e, consequentemente, no modo de delas retirar nós. A rede foi sempre representada pela estrutura *rede completa*.

Sete Nesta versão são usadas duas árvores (não completas) e duas listas (simples). As listas ligadas são construídas (e usadas) pela rotina de rotulação. Esta versão utiliza, portanto, menos memória do que as outras, mas torna-se mais lenta na remoção de nós da árvore. De facto, para encontrar os filhos de um dado nó x na árvore é necessário percorrer sequencialmente a lista a partir de x à procura dos nós cujo pai é o próprio x. Por outro lado, para manter a lista ligada, quando se quer retirar um dado nó, é preciso conhecer o nó que o precede na lista. Para isso, ao percorrer a lista temos de conhecer sempre, em simultâneo, dois nós consecutivos: o que se examina (designado por *actual*), e o que o precede (designado por *anterior*), que será utilizado na remoção. Isto significa que para retirar a sub-árvore de raiz x é necessário começar a pesquisa com *anterior* a assumir o valor de um nó que preceda x na lista. Nesta implementação, *anterior* começa por assumir o valor de pai de x e *actual* o valor do nó que se segue, na lista, ao pai de x [5]. Percorre-se a lista até *actual* ser igual a x, sendo este, então, retirado da árvore e da lista. Depois percorre-se toda a restante lista procurando nós cujo pai já não esteja na árvore, nós esses que são retirados [6].

As estruturas usadas nesta implementação ocupam $n \times (4+16) = n \times 20$ bytes [7].

Oito Nesta versão a lista ligada é substituída por uma lista duplamente ligada. Isto facilita a remoção dos nós da árvore, o que por sua vez permite descobrir todos os nós a retirar dum modo mais eficiente. De facto, para retirar a sub-árvore com raiz em x, em vez de percorrer a lista de nós rotulados começando em x, o que corresponde a percorrer todos os nós da árvore cujo nível seja mais alto que o de x, vamos percorrer o conjunto dos sucessores de x. Para cada $y \in S(x)$ se o pai de y for x, y é retirado da árvore, sendo este processo aplicado recursivamente. Como esta recursão é simulada com uma lista (simples) a memória ocupada por esta versão é $n \times (4 + 8 + 16) = n \times 28$ bytes.

Onze A versão anterior ainda percorre desnecessáriamente muitos nós, embora normalmente menos que a sete. Para evitar isto, nesta nova versão

[5] Note-se que este nó pode ser o próprio x.

[6] De facto, se pai de x é retirado da árvore, o mesmo tem de suceder a todos os seus filhos, sendo isto válido para qualquer nó x.

[7] Um nó nunca está, simultâneamente, na árvore com raiz em s e na árvore com raiz em t, o mesmo acontecendo em relação às listas ligadas. Logo, a soma de elementos das duas árvores (listas) não ultrapassa n, pelo que se representam as duas árvores (listas) por um único vector de n elementos, o mesmo acontecendo em todas as versões programadas.

usamos a estrutura de dados árvore completa em vez da árvore simples. Assim em vez de percorrermos todos os nós que poderiam ser filhos de um dado nó, percorremos apenas os que são mesmo. Como inconvenientes, ocupa-se mais memória ($n \times (4 + 8 + 24) = n \times 36$ bytes) e a gestão da árvore é mais complexa.

1.2.6 Experiência computacional

As várias versões do algoritmo foram programadas em C, constituindo várias rotinas de um único programa. A escolha desta linguagem tem as seguintes vantagens:

1. Tal como já foi dito atrás, a gestão da memória é muito eficiente graças à existência de vectores dinâmicos (com tamanhos determinados durante a execução).

2. O programa ficou bem estruturado tanto em termos das estruturas de controlo usadas, como em termos de organização dos dados[8].

3. Conseguiu-se uma grande portabilidade, podendo-se executar o programa em qualquer máquina UNIX.

O algoritmo apresentado foi comparado principalmente com o algoritmo primal simplex, numa versão específica para o problema do fluxo máximo [9]. A comparação com este algoritmo deveu-se ao facto de este ser considerado o mais rápido na prática[9]. A implementação usada para o algoritmo simplex, foi a melhor das várias por nós testadas num trabalho anterior [15] [9]. Também foram feitos alguns testes com uma versão do algoritmo de Goldberg e Tarjan [10] e com o algoritmo de Bertsekas e Tseng [4] [5], este para o problema do fluxo de valor dado e custo minímo. Note-se que o problema do fluxo máximo pode ser formulado como um caso particular deste.

[8]Uma opção não muito correcta foi o ter-se amarzenado o valor do fluxo em cada arco na representação da rede. Apesar de ter resultado um programa mais simples e mais eficiente, esse valor é um *resultado* do problema e não um *dado* pelo que deveria estar separado. Aliás, no trabalho descrito do segundo capítulo, foi necessário modificar a estrutura (e portanto o programa) para fazer essa separação.

[9]Nesse trabalho não foram testadas todas as hipóteses referidas em [9], portanto é possível que esta implementação ainda possa ser melhorada.

	sete	oito	onze	simplex	GT	BT
100 fontes e 100 destinos	2.333	0.516	0.383	24.432	NC	151.490
grelhas (20×150)	31.198	33.498	33.282	24.449	NC	161.600
grelhas (150×20)	4.999	4.583	4.466	5.366	NC	51.380

Tempos de CPU, em segundos, obtidos num Sun Sparc Server 330 (32Mb RAM, 15.8 mips, 1.7 mflops)

GT: Algoritmo de Goldberg e Tarjan
BT: Algoritmo de Bertsekas e Tseng

NC: Não terminou ao fim de 20 minutos de CPU

Tabela 1.2.6

O algoritmo de Goldberg e Tarjan é o que tem a melhor complexidade para o pior caso ($\mathcal{O}(nm\ log(\frac{n^2}{m})))$. No entanto nos testes que efectuámos revelou-se bastante lento [10], excepto para redes "difíceis", em que foi o mais rápido. Note-se no entanto, e sem entrar em considerações sobre o algoritmo, que este tipo de redes é o seu melhor caso; pelo contrário, para qualquer algoritmo de rotulação constitui o pior caso.

Quanto ao algoritmo de Bertsekas e Tseng, que é extremamente rápido para o problema para o qual foi concebido, também se revelou mais lento, o que não é de admirar, dado resolver um problema mais geral. O código, em FORTRAN 77, utilizado para este algoritmo foi-nos oferecido pelos autores, a quem agradecemos.

Para ilustrar o comportamento destes dois algoritmos no caso geral, apresentam-se apenas alguns dos resultados obtidos, na tabela 1.2.6.

O algoritmo simplex tem a importante vantagem de só necessitar da representação da rede em estrela directa, o que tem duas consequências que o favorecem em relação ao algoritmo proposto:

1. A ocupação de memória é menor. Incluindo as representações da rede, das árvores e de uma lista ligada que só é precisa durante a fase de

[10] A implementação testada talvez não fosse a melhor possível; mas é tão lenta que não nos parece provável que se consiga melhorar a ponto de se tornar competitiva.

inicialização, o algoritmo simplex necessita de $n \times 8 + m \times 12 + n \times 32 = n \times 40 + m \times 12$ bytes [11]. Nas mesmas condições o novo algoritmo necessita de $n \times 16 + m \times 20 + n \times b_1 = n \times b_2 + m \times 20$ [12] bytes, onde $b_1 \in \{20, 28, 36\}$ e $b_2 \in \{36, 44, 52\}$. O valor dos b_i depende da versão utilizada. Ou se se exprimir o espaço ocupado em função do número de nós n e da densidade d teremos $n \times 40 + n \times d \times 12 = n \times (40 + d \times 12)$ para o algoritmo simplex e $n \times b_2 + n \times d \times 20 = n \times (b_2 + d \times 20)$ para as várias versões do novo algoritmo.

2. Tanto o gerador como as rotinas de entrada-saída utilizadas, usam a estrutura rede (estrela directa). A rotina que converte esta estrutura na estrutura rede dupla pode consumir um tempo significativo [13].

Como é evidente isto só é relevante se o programa onde este algoritmo for usado não necessitar da representação rede completa.

O motivo desta preocupação com o espaço de memória ocupado é devido ao facto de problemas que ocupam quase toda a memória de um computador, ainda se conseguirem resolver em alguns segundos (caso dos problemas com redes aleatórias). Portanto com um algoritmo que necessite de menos memória é possível resolver problemas de maiores dimensões, pois o factor limitativo é o espaço ocupado e não o tempo de execução.

1.2.7 Análise dos resultados

Dos muitos problemas resolvidos, apresentam-se alguns dos resultados mais significativos obtidos para o tempo de execução, sob a forma de gráfico.

[11] Não se contabiliza o espaço ocupado pelas variáveis e campos de estruturas quando o seu tamanho não depende de n ou m, pois para valores elevados de n ou m, este é desprezável.

[12] Em ambos os casos pode-se poupar $n \times 4$ bytes compactando dois campos numa única "word". Ao contrário das implementações do novo algoritmo, isso foi feito na implementação do algoritmo simplex. Por outro lado, parece possível dispensar a lista ligada usada durante a inicialização, pois a informação que ela contém está também disponível na árvore.

[13] Consoante o tipo de rede usada para testar os algoritmos, observámos que esta rotina consumia menos de 1% do tempo de execução de cada um dos algoritmos (caso das redes "difíceis" e das grelhas) ou mais tempo que algumas das versões (caso das redes aleatórias com um único nó fonte e um único nó destino).

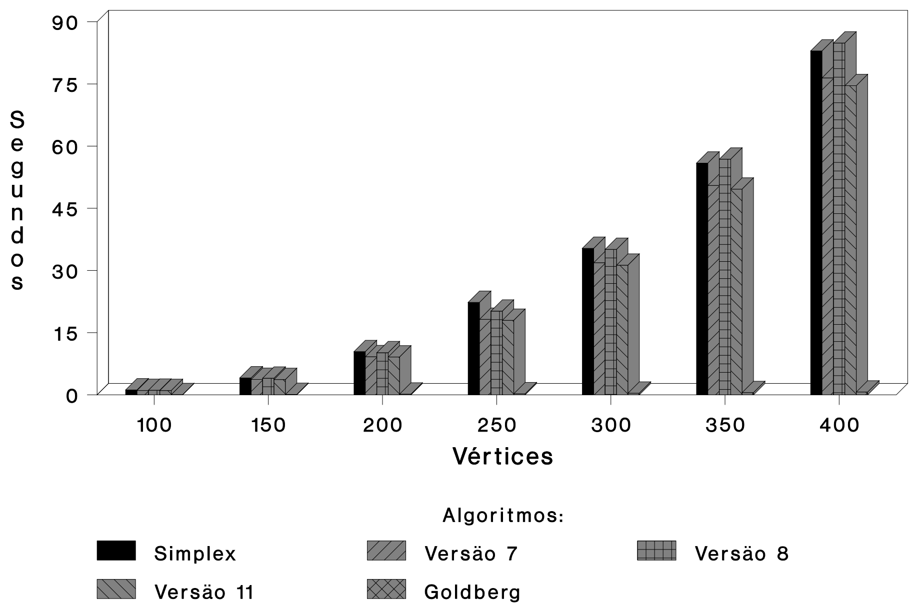


Figura 1 - Redes difíceis

Os tempos são a média para trinta problemas diferentes mas da mesma dimensão[14]. Estes tempos foram obtidos num computador DEC System 3100 com 16 Mb de RAM, excepto os testes para redes "difíceis", que foram executados num SUN SPARC Server 330 com 32Mb de RAM[15].

Dos resultados apresentados na figura 1, verifica-se que, para redes "difíceis", o algoritmo de Goldberg e Tarjan é muito mais rápido em relação quer ao algoritmo simplex, quer ao algoritmo proposto. Das três versões apresentadas, a oito é a mais lenta de todas, sendo mesmo batida pelo algoritmo simplex à medida que o número de nós aumenta. As versões sete e onze têm comportamentos idênticos, com uma ligeira vantagem para a versão onze.

Para este caso, que é o pior possível para os algoritmos de rotulação, calcularam-se empiricamente as complexidades dos vários algoritmos, tendo-

[14]Exceptuam-se os testes com redes "difíceis", por, para um dado número de nós, serem todas iguais. Neste caso cada problema foi executado dez vezes.

[15]Ambos os sistemas pertencem ao Laboratório de Cálculo do Departamento de Matemático da FCTUC. Estas duas máquinas têm aproximadamente 15 MIPS.

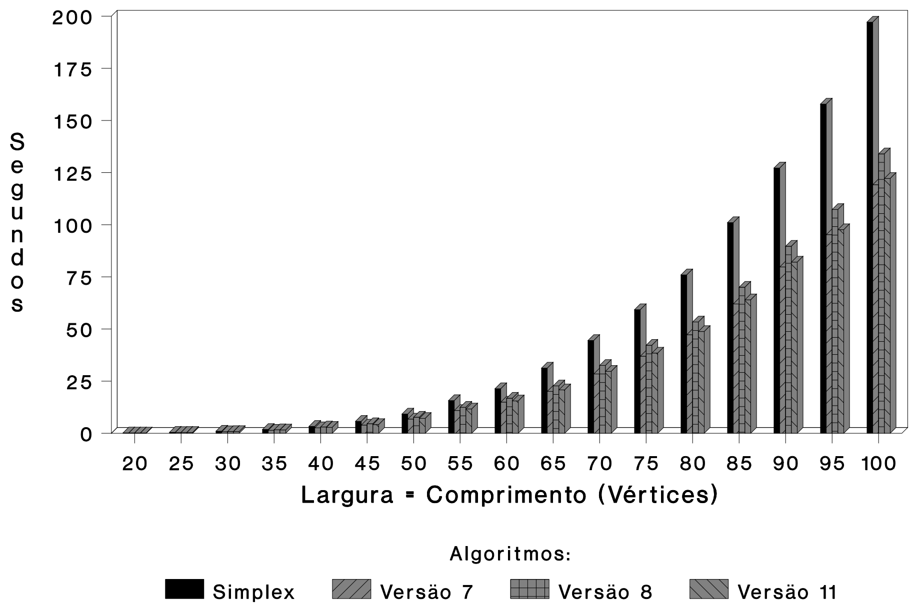


Figura 2 - Grelhas quadradas

se obtido n^3 para o simplex e para as três versões do algoritmo apresentado e n^2 para o algoritmo de Goldberg e Tarjan. Esta análise foi feita calculando por regressão linear os parâmetros a e b' da equação $\ln t = a \times \ln n + b'$, ou seja, $t = b \times n^a$, onde $b = e^{b'}$.

Concretizando, as expressões obtidas e respectivo coeficiente de correlação foram:

$$t = 8.39799 \times 10^{-7} \times n^{3.07891},\ r^2 = 0.999082,$$

$$t = 6.99920 \times 10^{-7} \times n^{3.09075},\ r^2 = 0.999946,$$

$$t = 6.28336 \times 10^{-7} \times n^{3.12785},\ r^2 = 0.999887,$$

$$t = 7.37717 \times 10^{-7} \times n^{3.07843},\ r^2 = 0.999926,$$

$$t = 2.93358 \times 10^{-6} \times n^{2.06872},\ r^2 = 0.999570,$$

respectivamente, para o algoritmo simplex, para as versões sete, oito e onze do algoritmo apresentado e para o algoritmo de Goldberg e Tarjan.

Nas figuras 2, 3 e 4 apresentam-se os resultados obtidos com redes do tipo grelha. Na figura 2 verifica-se o comportamento dos algoritmos relativamente a grelhas quadradas. É evidente a supremacia de qualquer das versões relativamente ao algoritmo simplex, supremacia essa que aumenta com o número de nós. Relativamente às três versões, os resultados são análogos aos anteriores, mas desta vez a versão onze tem uma ligeira vantagem em relação à sete.

Na figura 3 as redes têm comprimento fixo (25 nós) e varia-se a largura entre 10 e 200 nós. Na figura 4 é exactamente o contrário, ou seja, fixamos a largura (também em 25 nós) e variamos o comprimento. Conclui-se que, para redes largas, o algoritmo mais lento é o simplex, seguindo-se por ordem as versões sete, oito e onze. Pelo contrário, para redes compridas, as posições relativas quase se invertem, sendo agora o algoritmo simplex o mais rápido, seguindo-se as versões sete, onze e oito. Entre os testes efectuados, este é o único caso em que o algoritmo simplex é melhor que qualquer uma das versões do novo algoritmo.

Na figura 5 mostram-se os resultados para redes com 10 nós fonte e 10 nós destino, densidade 5 e número variável de nós. Devido à diferença de velocidade entre o algoritmo simplex e as três versões, na figura 6 são apresentados apenas os resultados paras estas versões. Para este tipo de problemas o algoritmo simplex é batido por um factor superior a 10. Entre as três versões a sete é claramente a mais lenta. Geralmente a onze é superior à oito, embora, em alguns problemas, esta consiga ser ligeiramente melhor. Este gráfico parece sugerir que a versão oito se torna melhor para problemas maiores. Para testar esta hipótese foram executados alguns testes com redes ainda maiores (até 250000 nós) que não a confirmaram. Pelo contrário verificou-se que a versão sete ultrapassou as outras duas para redes *muito* grandes. Isto é certamente devido às sua menores necessidades de memória, pois estas redes já não cabiam totalmente em RAM obrigando ao uso de memória virtual.

As figuras 7 e 8 são análogas às 6 e 7, mas desta vez o número de nós é fixo (25000) e a densidade é variável. As conclusões também são análogas às anteriores, sendo a principal diferença o facto da versão oito nunca conseguir ser melhor que a onze.

As figuras 9 e 10 mostram os tempos para uma rede de tamanho fixo (10000 nós e densidade 5) variando o número de nós fonte e destino (que são iguais). Também neste caso o algoritmo simplex é mais lento que o novo por um grande factor. A versão sete é bastante pior que as outras duas, e

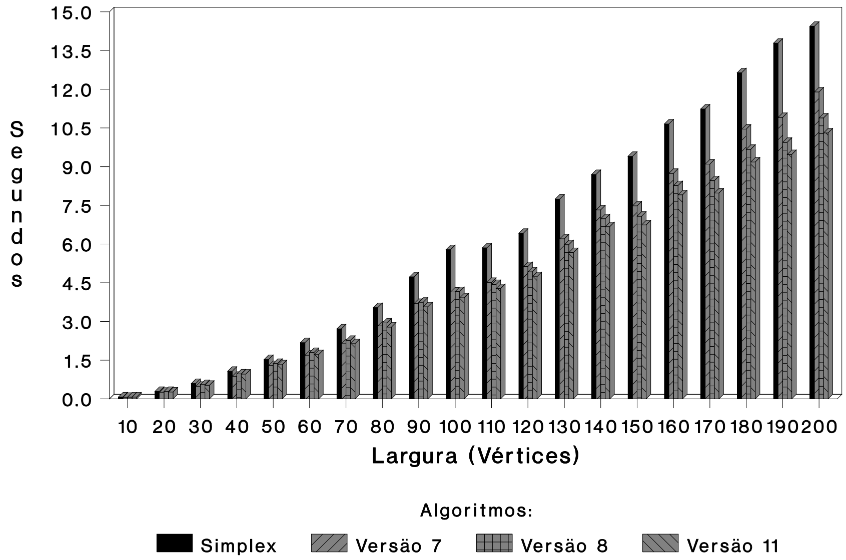


Figura 3 - Grelhas (Comprimento=25 nós)

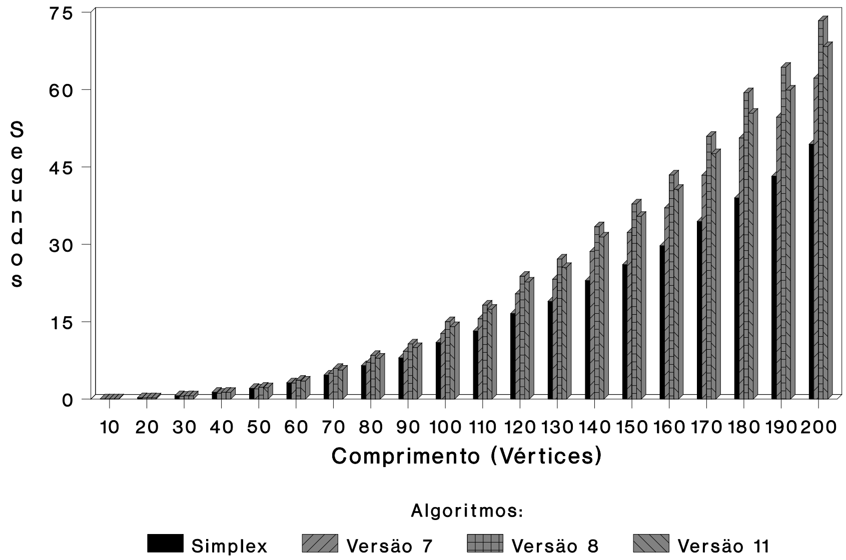


Figura 4 - Grelhas (Largura=25 nós)

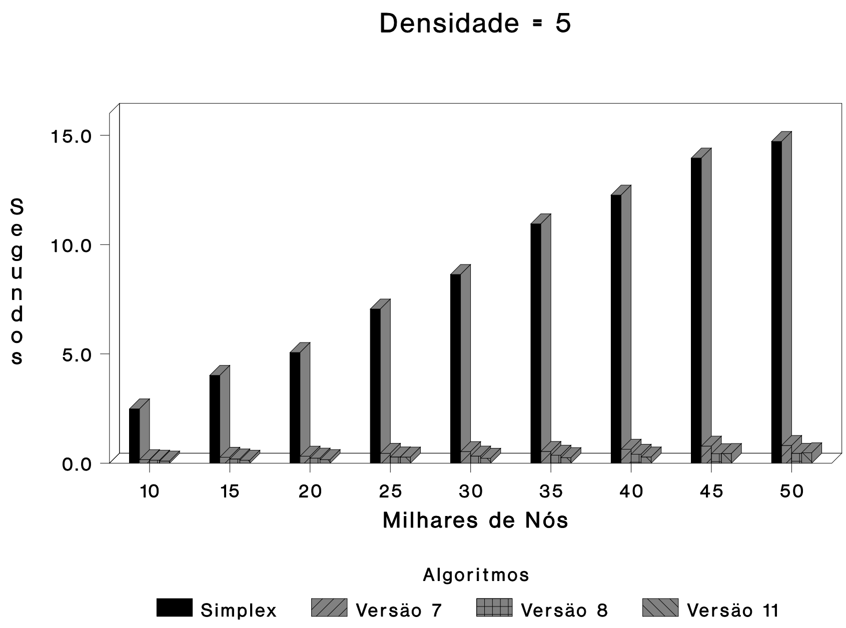


Figura 5 - 10 nós fonte e 10 nós destino

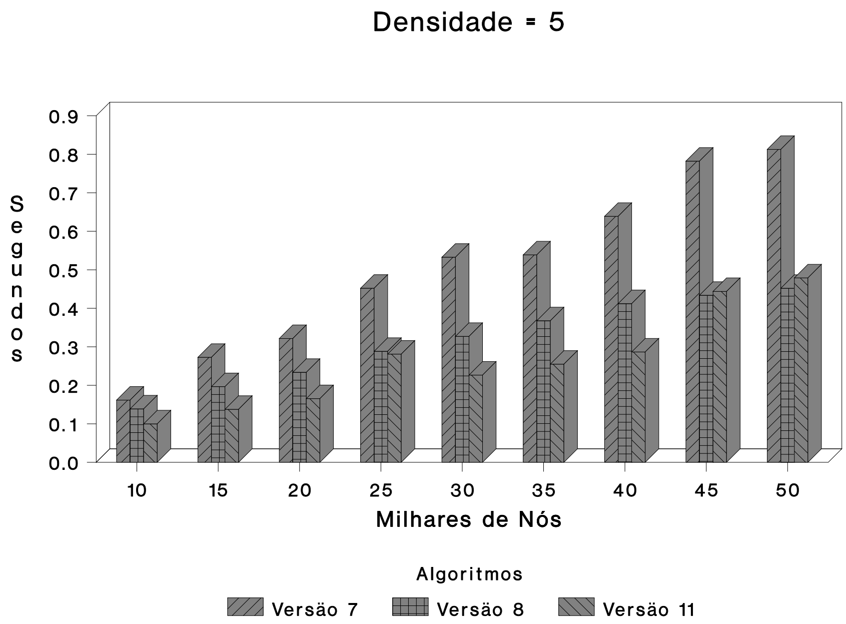


Figura 6 - 10 nós fonte e 10 nós destino

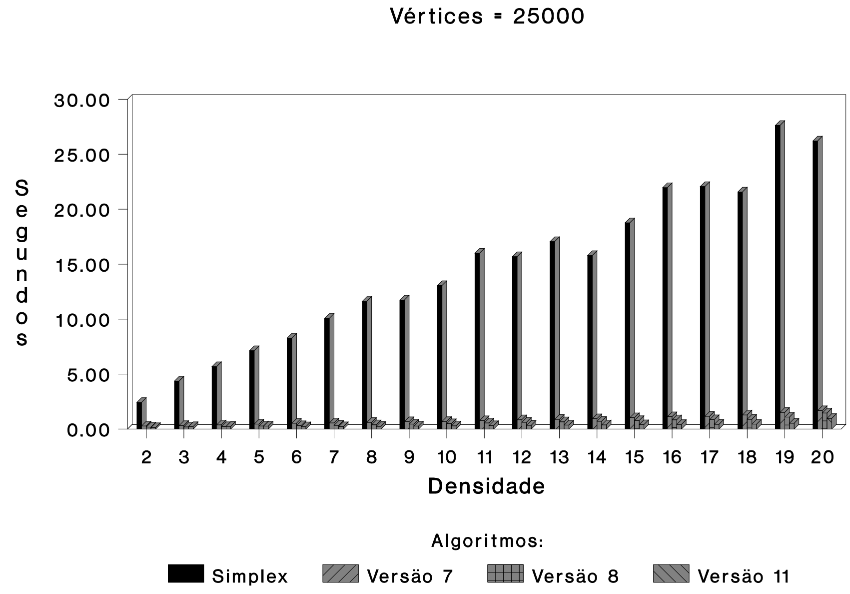


Figura 7 - 10 nós fonte e 10 nós destino

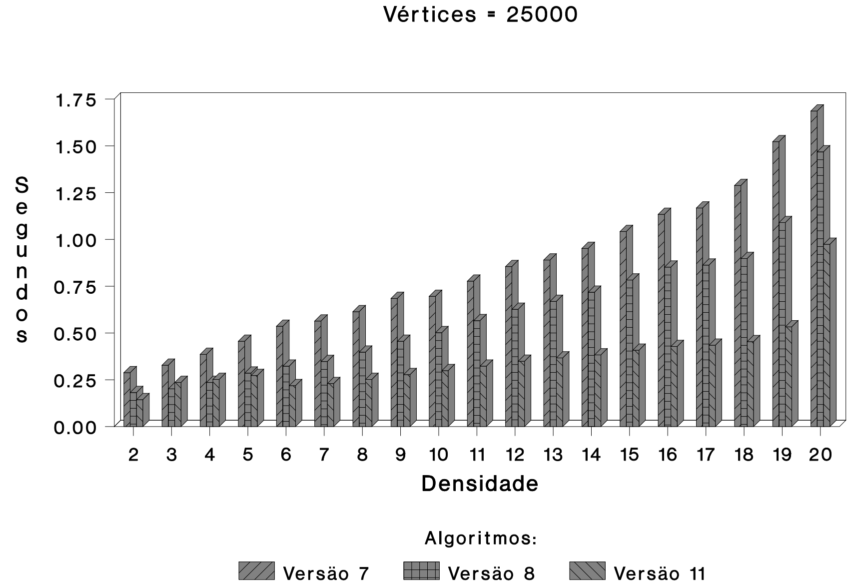


Figura 8 - 10 nós fonte e 10 nós destino

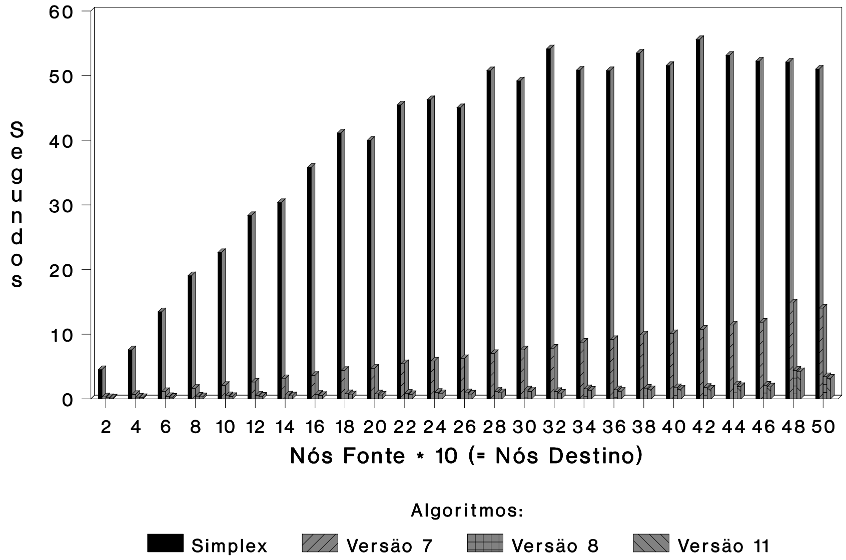


Figura 9 - 10000 nós e densidade 5

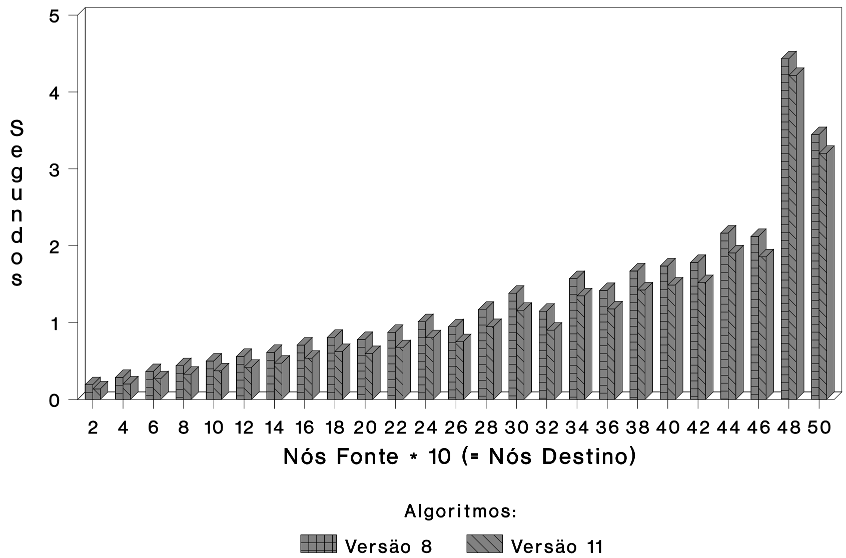


Figura 10 - 10000 nós e densidade 5

portanto na fig 10 só se apresentam os resultados para estas. Observa-se que a versão onze é sempre melhor do que a oito.

Nas figuras 11 e 12 temos os tempos obtidos para redes de tamanho fixo (novamente 25000 nós e densidade 5) variando o número de nós destino (o número de nós fonte está fixo em 10). Repare-se que, devido à simetria dos algoritmos considerados, não há uma diferença significativa entre estes problemas e os que seriam gerados para um número de nós destino fixo e número de nós fonte variável.

Mais uma vez o simplex é muito pior que o novo algoritmo, e, entre as três versões, a melhor é a onze seguida pela oito e finalmente pela sete. Repare-se que, se a princípio os tempos aumentam com o número de nós destino, quando este ultrapassa o número de nós fonte, os tempos deixam de aumentar, chegando mesmo a diminuir. Isto parece dever-se a que, enquanto há mais nós fonte que nós destino, o número de cadeias que o algoritmo necessita de encontrar aumenta com o número de nós destino. A partir da altura em que o número de nós destino ultrapassa o número de nós fonte, o número de cadeias necessárias deixa de aumentar[16] e passam a ser mais fáceis de determinar, por a árvore de raiz em t ser maior e depois da determinação de cada cadeia incremental, lá restarem mais nós.

1.2.8 Conclusões

Da análise dos resultados, verificamos que o algoritmo simplex só conseguiu ser mais rápido para grelhas "compridas", ou seja, aquelas nas quais o número de colunas é bastante maior que o número de linhas. Em todos os outros casos o novo algoritmo foi mais rápido. Este comportamento poderá ser devido ao facto da determinação de uma cadeia incremental com maior número de arcos ser mais demorada.

Das três versões apresentadas, a mais rápida é sem dúvida a onze. Apesar de ser ultrapassada pela versão sete nas grelhas "compridas" e pela oito em dois casos (em redes aleatórias), tem o melhor comportamento médio, comportando-se bastante bem em todo o tipo de problemas. Notou-se também a grande estabilidade das três versões em relação ao algoritmo simplex.

O problema da maior quantidade de memória necessária ao novo algoritmo só é relevante nos problemas com redes aleatórias (com um ou vários

[16] O valor do fluxo, que era limitado pela capacidade dos arcos que chegam aos nós destino, passa a ser limitado pela capacidade dos arcos que saiem dos nós fonte.

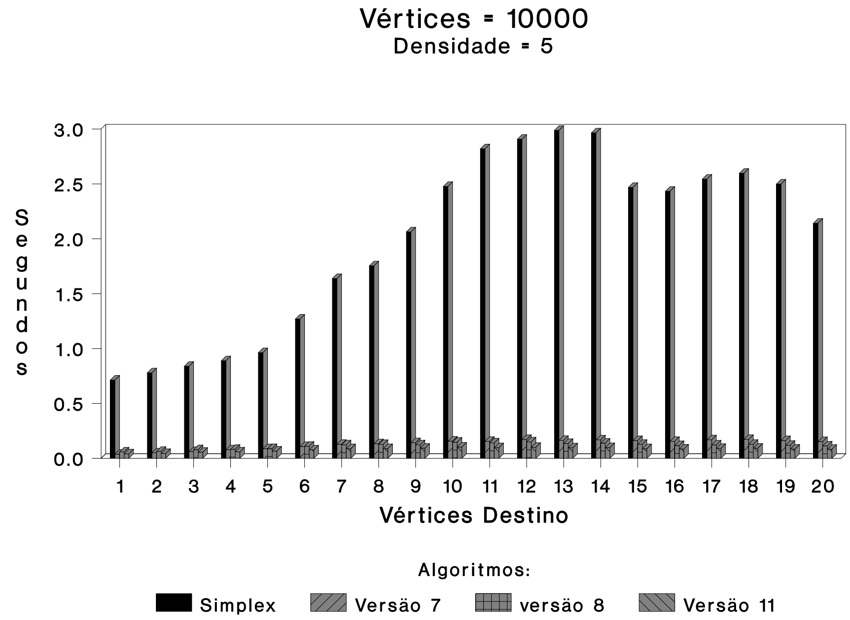


Figura 11 - 10 nós fonte

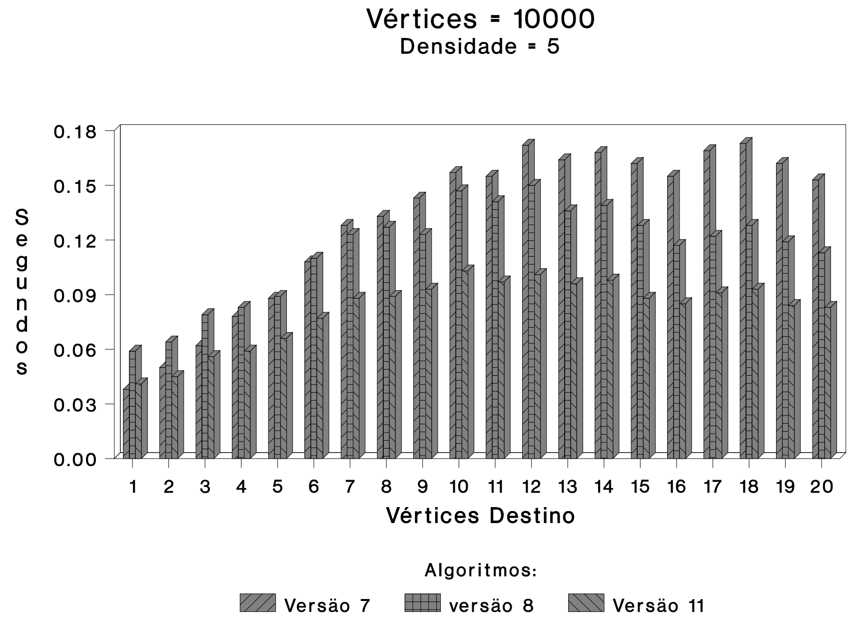


Figura 12 - 10 nós fonte

nós fonte e um ou vários nós destino), pois para os outros tipos de redes, o tempo de resolução já é considerável, mesmo em redes não muito grandes. Por exemplo, um problema para uma rede aleatória que ocupa mais de 32 Mb de RAM (250000 nós, densidade 5) é resolvido em cerca de 7, 13 e 9 segundos, respectivamente pelas versões sete, oito e onze. Estes são tempos de CPU, já que o tempo real é muito superior, da ordem dos 5 minutos. Pelo contrário um problema com grelhas que ainda não ocupa totalmente a mesma quantidade de memória (grelha quadrada de 500 por 500 nós) demora mais de 10 *horas* de tempo de CPU. No entanto é justamente para redes aleatórias que o novo algoritmo é mais vantajoso em relação ao algoritmo simplex (por um factor que pode ser superior a 10).

1.2.9 Futuras direcções

Na avaliação de algoritmos com base em testes empíricos as comparações devem ser justas. Para isso todos os códigos testados devem ser da autoria do mesmo programador, codificados na mesma linguagem e corridos no mesmo sistema em idênticas condições de utilização. Isto implica que não podemos tirar conclusões muito seguras a partir de resultados de vários autores. Portanto, para obtermos conclusões mais rigorosas, falta a comparação com as melhores versões apresentadas em [9], e ainda com a versão do algoritmo de Goldberg e Tarjan, programada com a utilização de árvores AVL[17].

O gerador de problemas também parece poder ser melhorado. As redes aleatórias, apesar de serem usadas por muitos autores, não parecem muito realistas. O ideal seria obter redes usadas em problemas reais e utilizá-las para testar os algoritmos ou desenvolver um gerador de redes semelhantes.

[17] De referir que esta versão do algoritmo é a que apresenta a melhor ordem de complexidade para o pior caso

Capítulo 2

Aplicação ao Problema da Partição de Redes

2.1 Introdução

A partição de redes é um problema que surge em projecto automático de circuitos electrónicos.

Para este capítulo vamos necessitar de generalizar a noção de árvore usada no capítulo anterior. Para isso será necessário introduzir a noção de grafo não orientado.

Definição 2.1 *Grafo não orientado é um par $(\mathcal{N}, \mathcal{A})$, onde $\mathcal{N} = \{v_1, \ldots, v_n\}$ é chamado o conjunto dos vértices (ou nós), $\mathcal{A} = \{a_1, \ldots, a_m\}$ é o conjunto dos arcos e cada arco a_i é um par não ordenado de nós.*

Note-se que num grafo não orientado, a noção de caminho coincide com a noção de cadeia.

Definição 2.2 *Uma árvore é um grafo não orientado onde existe uma e uma só cadeia entre qualquer par de vértices.*

Numa árvore existe um nó especial denominado raiz. As noções de pai, filho, irmão e nível de um nó, definem-se de um modo análogo ao primeiro capítulo.

Definição 2.3 *Uma árvore binária é uma árvore em que cada nó tem no máximo dois filhos. Uma árvore de partição é uma árvore binária onde só existem nós sem filhos (denominados folhas) ou nós com dois filhos (denominados nós intermédios).*

Para modelar um circuito utiliza-se uma rede em que os nós correspondem a componentes do circuito e os arcos representam as ligações entre duas componentes. As capacidades dos arcos entre dois nós representam o número de ligações [1] entre as duas componentes correspondentes.

Para representar as posições relativas das várias componentes usa-se uma árvore de partição. Nesta árvore cada folha representa uma componente. Os nós intermédios representam conjuntos de componentes, conjuntos esses que são o resultado da união dos conjuntos representados pelos seus filhos. Como é evidente, a raiz da árvore representa todas as componentes do circuito.

Em termos de posição, cada nó da árvore representa um rectângulo, que é resultado da justaposição dos dois rectângulos correspondentes aos filhos desse nó. As folhas representam o mais pequeno rectângulo em que cabe a componente correspondente. A raiz representa o rectângulo ocupado por todo o circuito.

Uma descrição mais detalhada deste modelo de circuitos pode ver-se, por exemplo, em [14] ou [1].

2.2 Descrição da Heurística

A árvore de partição de um circuito, pode ser contruída particionando sucessivamente a rede que representa o circuito. Dado que um dos principais objectivos a que esta árvore deve satisfazer é agrupar as componentes (ou conjuntos de componentes representados por sub-árvores) mais fortemente ligadas, cada uma dessas partições deveria ser a solução do seguinte problema:

$$\text{minimizar } c(X, \bar{X}),$$

no conjunto de todos os cortes $(X, \bar{X})$ da sub-rede respectiva e sujeito a:

$$|\#X - \#\bar{X}| \leq k, \; k \in \mathbb{N}.$$

[1] Pode-se dar mais peso a certas ligações adicionando um valor a esta capacidade.

Este problema é habitualmente designado *Problema do Corte Mínimo*. Note-se no entanto que este corte não é idêntico ao definido na subsecção 1.1.4, pois nesse especificava-se um nó ou um grupo de nós que tinha de pertencer a X (os nós fonte) e um nó ou um grupo de nós que tinha de pertencer a $\bar{X}$ (os nós destino). Pelo contrário, agora não são especificados os nós ou conjuntos de nós que têm de ser separados pelo corte.

Note-se também que o problema de encontrar um corte para s e t de capacidade mínima tem complexidade polinomial. Pelo contrário, este problema é NP-completo, o que torna necessário o recurso a heurísticas.

Neste trabalho tentou-se construir uma heurística baseada no algoritmo apresentado para o problema do fluxo máximo, que, como foi visto atrás, encontra ao mesmo tempo um corte (para s e t) de capacidade miníma.

O algoritmo teve de ser adaptado para este caso, pois os grafos utilizados são não orientados (o que obrigou a retirar o primeiro melhoramento).

Dado que o algoritmo precisa de um nó fonte e de um nó destino como dados, é necessário definir um critério para os escolher. Verificou-se experimentalmente que o corte obtido tem tendência para isolar o nó fonte ou o nó destino, ou seja, obtêm-se frequentemente cortes do tipo $(s, \mathcal{N} - s)$ ou $(\mathcal{N} - t, t)$. Para tentar obstar a este efeito escolheu-se para nó fonte o que está mais ligado aos outros, ou seja, o que maximiza $c(s, \mathcal{N})$, no conjunto dos nós da sub-rede em causa. Uma vez escolhido s segundo este critério, para nó destino usou-se o nó mais afastado de s, e se houver mais do que um nessas condições, o mais ligado de entre estes. Como mesmo assim o algoritmo tende a isolar um nó, se a distância entre s e t for maior do que dois, ou seja, se os conjuntos dos sucessores de s e t forem disjuntos, calculamos o corte minímo para os conjuntos $S(s)$ e $A(t)$ [2].

Por se suspeitar que as redes aleatórias não são muito realistas, usaram-se redes geométricas tal como definidas em [11]. Para construir estas redes faz-se corresponder a cada nó um ponto. As coordenadas destes pontos vão estar uniformemente distribuídos no intervalo $[0, 1]$. Dois nós estão ligados por um arco se e só se a distância (Euclideana) entre os pontos correspondentes for menor ou igual a um dado parâmetro d. O grau médio dos nós não demasiado perto dos extremos do quadrado unitário será aproximadamente $n\pi d^2$. Esta propriedade é utilizada para que, ao gerar as redes, se possa utilizar um "interface" idêntico aos outros geradores.

[2] Como o grafo é não orientado, o conjunto dos antecessores de um nó coincide com o conjunto dos seus sucessores.

2.3 Avaliação da Heurística

Esta heurística foi comparada com a de Kernighan e Lin [7] e a heurística de Almeida e Rodrigues [3]. Utilizou-se também um método misto em que a heurística de Almeida e Rodrigues é utilizada como solução inicial da de Kernighan e Lin [3].

Para avaliar a "qualidade" das árvores obtidas foram usadas as funções objectivo O_1 e O_2, descritas por Almeida em [1]. A árvore óptima é a solução da minimização simultânea de O_1 e $|O_2|$, portanto é um problema multi-critério. O O_1 é uma medida de "clustering", ou seja, mede a distância entre componentes fortemente ligadas. O O_2 mede a uniformidade da distribuição de ligações entre conjuntos de componentes.

O circuito obtido a partir das árvores foi avaliado em termos dos objectivos clássicos: minimização de área e do semi-perímetro. A qualidade do posicionamento (em termos de facilitar o "routing") foi avaliada por D_1F e D_2F que são definidas do seguinte modo:

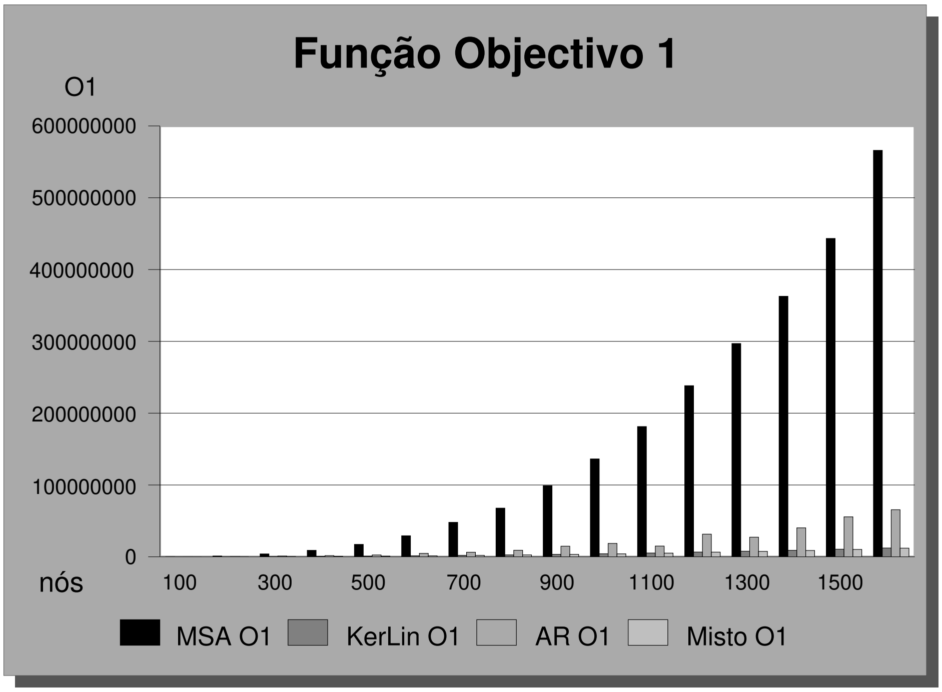
$$D_iF = \sum_{v_1, v_2 \in \mathcal{N}} d_i(v_1, v_2) \times c(v_1, v_2), i = 1, 2$$

em que $d_1(v_1, v_2)$ é a distância de Manhattan e $d_2(v_1, v_2)$ é o quadrado da distância euclideana entre as duas componentes correspondentes ao nós v_1 e v_2. Como é evidente o objectivo é minimizar estas medidas.

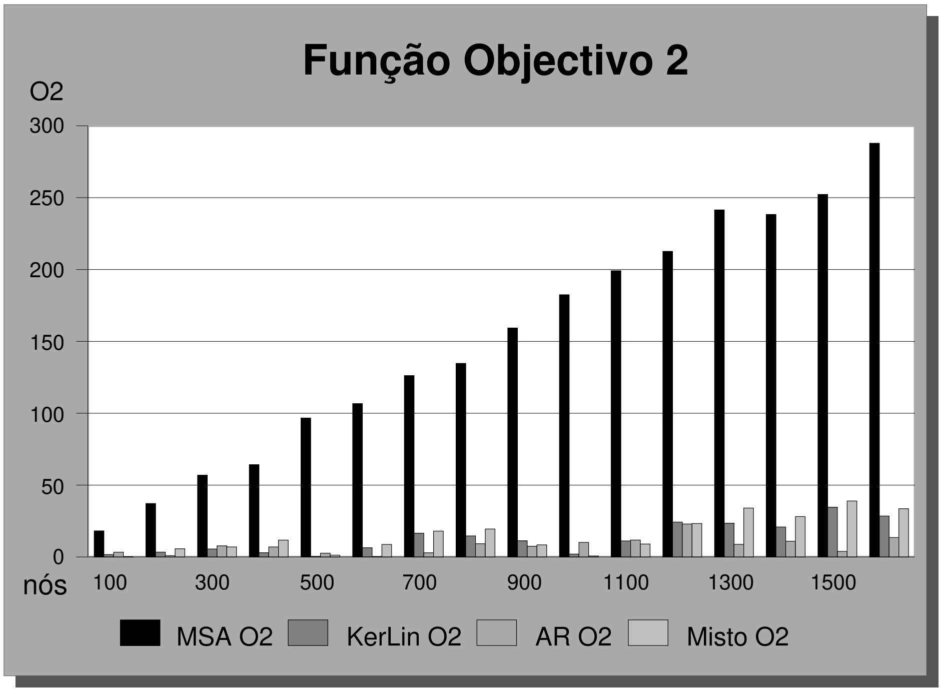
Teve-se igualmente em atenção o tempo necessário a cada heurística para a construção da árvore e para colocar as componentes no circuito. Verificou-se que o tempo necessário para esta colocação depende da "qualidade" da árvore obtida. Repare-se que quando comparamos *algoritmos*, estamos interessados na quantidade de recursos (tempo e memória) necessários para atingir a solução, enquanto que ao compararmos *heurísticas*, o factor essencial é a qualidade da solução obtida. O tempo só é importante para eliminar métodos que não obtêm uma boa solução em tempo razoável.

2.4 Análise de resultados e Conclusões

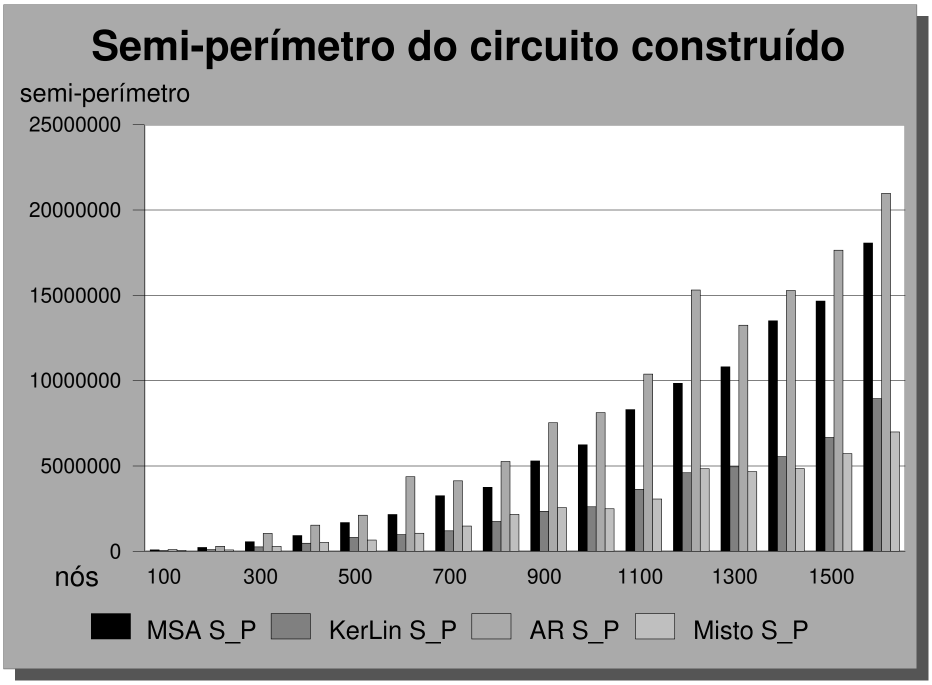
Como se pode ver dos gráficos juntos a heurística apresentada é pior que as outras em quase todos os aspectos. Somente para o semi-perímetro do circuito construído consegue ser melhor que a pior das outras heurísticas consideradas, sendo no entanto muito pior que as melhores.

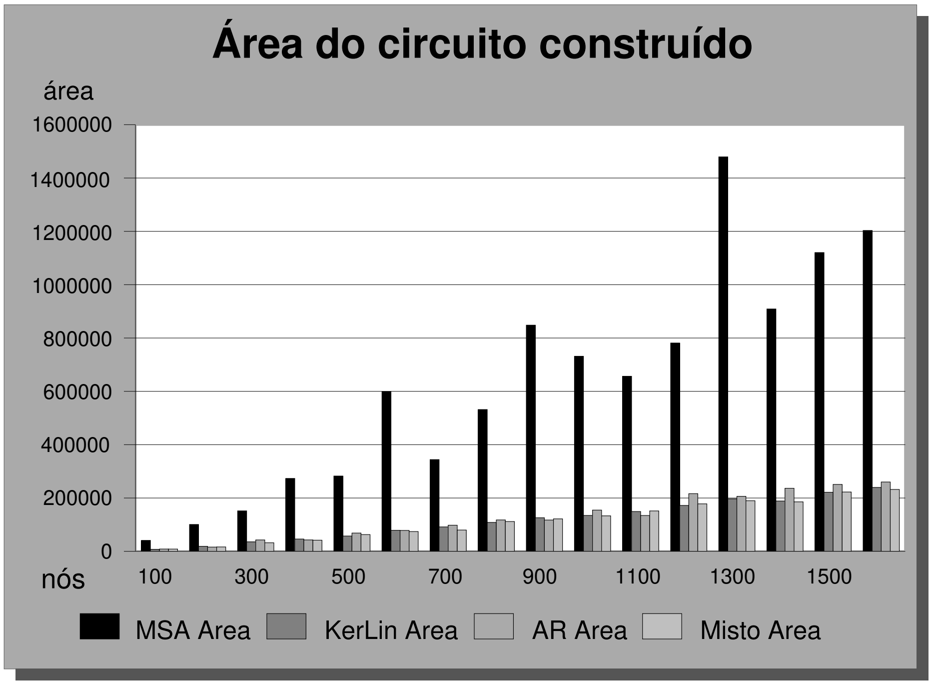


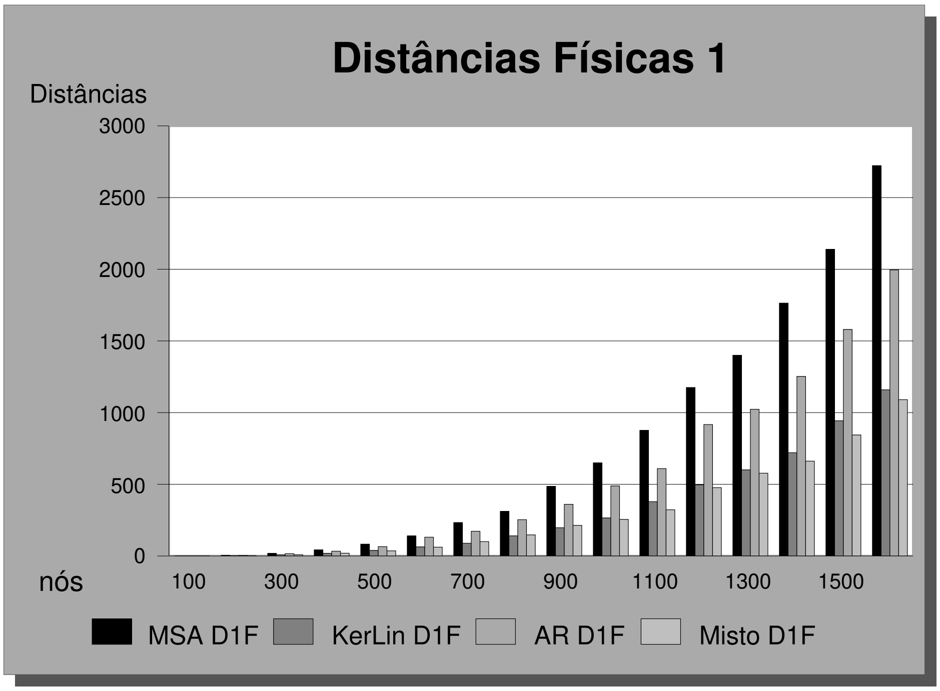
Função Objectivo 1
O1
600000000
500000000
400000000
300000000
200000000
100000000
0
nós
100
300
500
700
900
1100
1300
1500
MSA O1
KerLin O1
AR O1
Misto O1

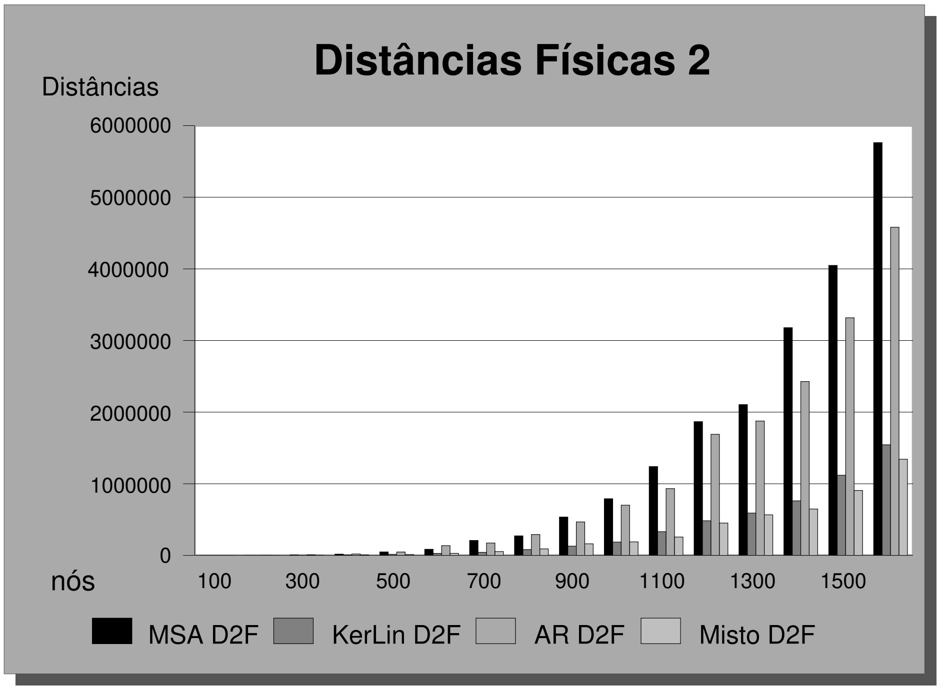


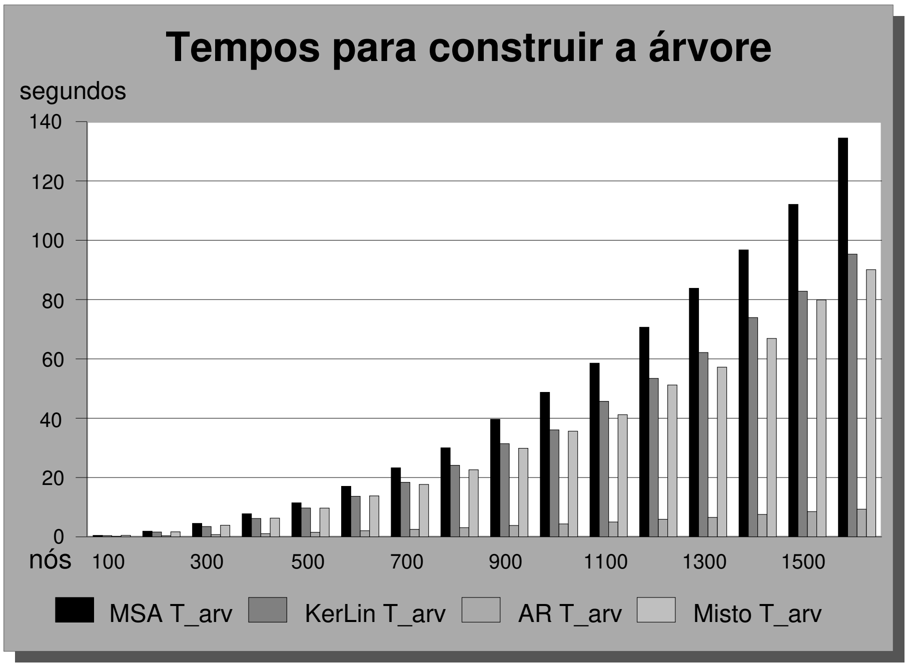
Função Objectivo 2
O2
300
250
200
150
100
50
0
nós
100
300
500
700
900
1100
1300
1500
MSA O2
KerLin O2
AR O2
Misto O2

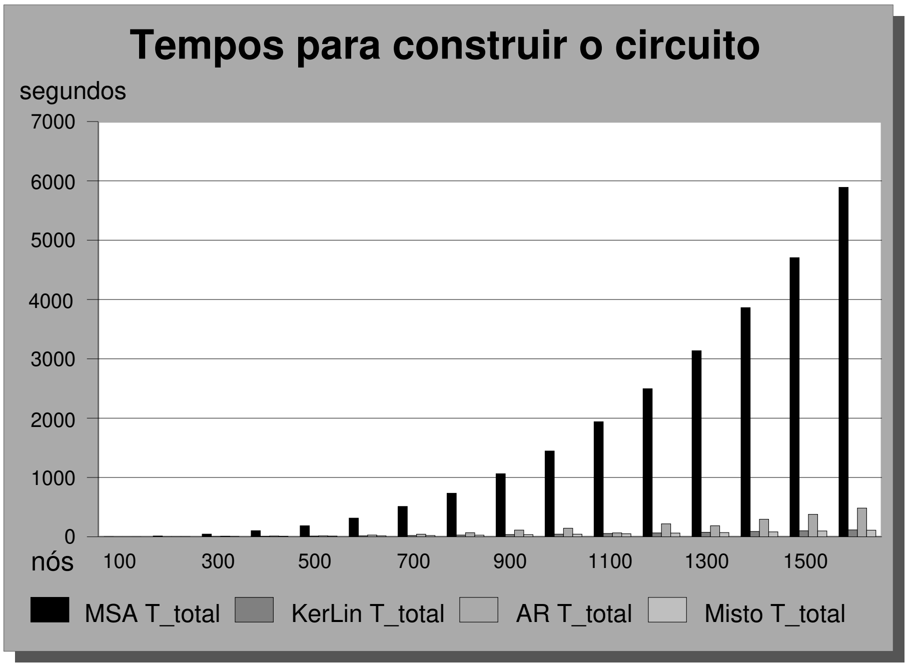

Semi-perímetro do circuito construído
semi-perímetro
25000000
20000000
15000000
10000000
5000000
0
nós
100
300
500
700
900
1100
1300
1500
MSA S_P
KerLin S_P
AR S_P
Misto S_P

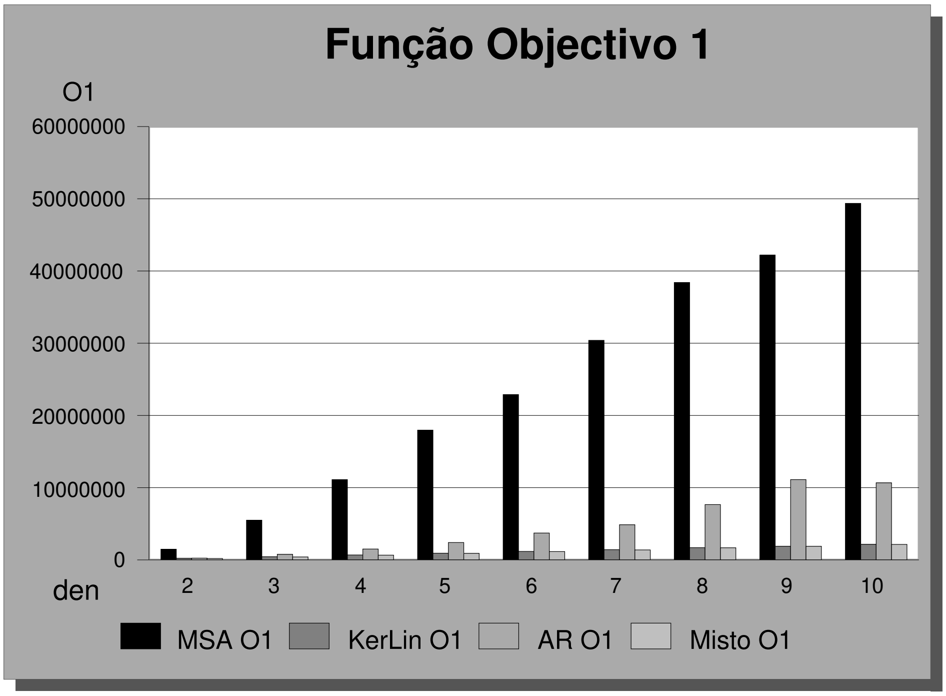

Área do circuito construído
área
1600000
1400000
1200000
1000000
800000
600000
400000
200000
0
nós
100
300
500
700
900
1100
1300
1500
MSA Area
KerLin Area
AR Area
Misto Area

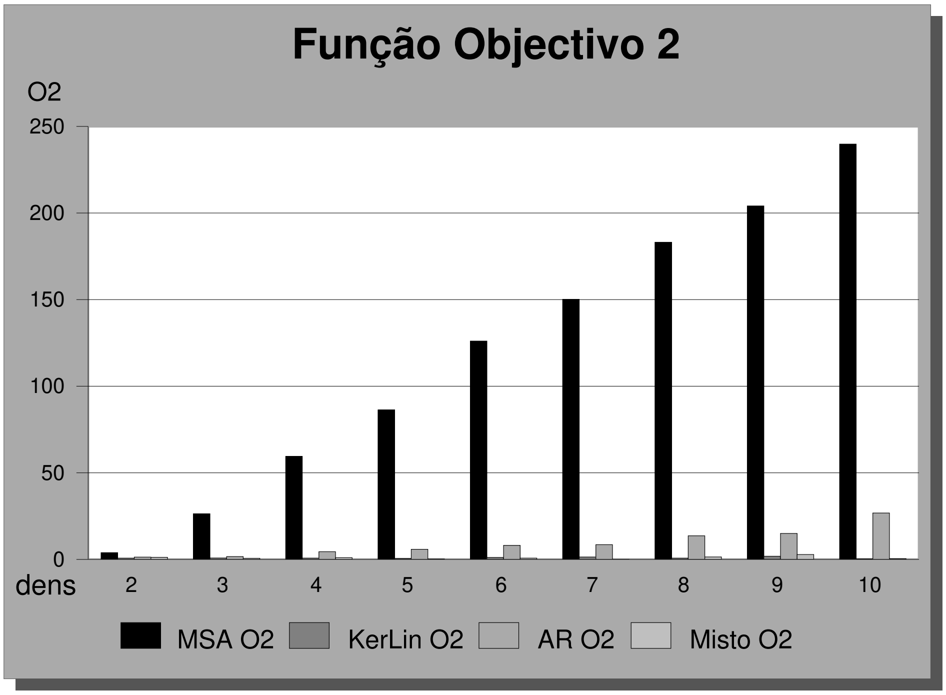

Distâncias Físicas 1
Distâncias
3000
2500
2000
1500
1000
500
0
nós
100
300
500
700
900
1100
1300
1500
MSA D1F
KerLin D1F
AR D1F
Misto D1F

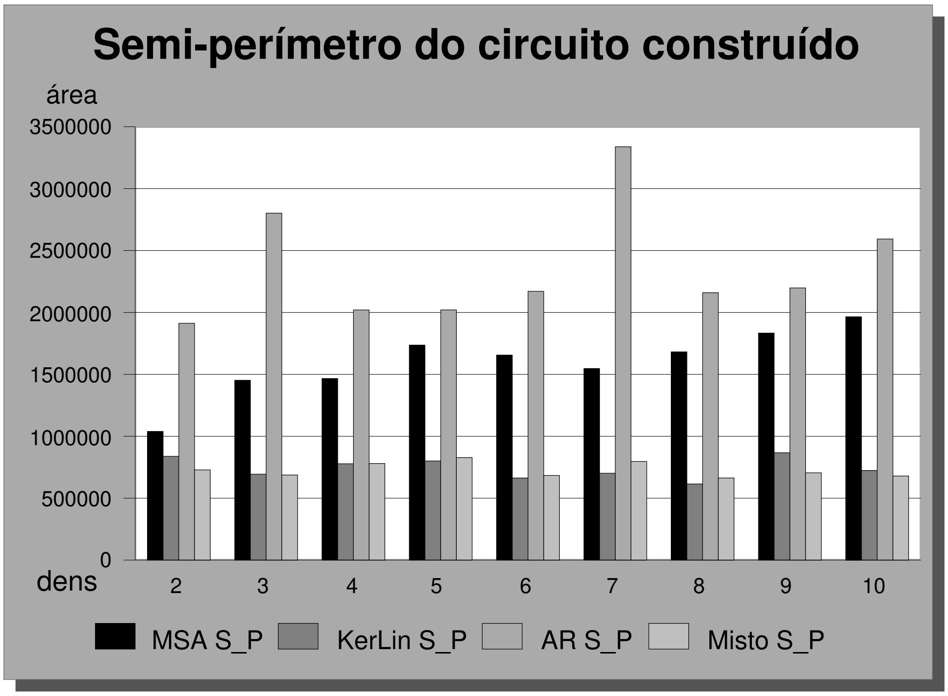

Distâncias Físicas 2
Distâncias
6000000
5000000
4000000
3000000
2000000
1000000
0
nós
100
300
500
700
900
1100
1300
1500
MSA D2F
KerLin D2F
AR D2F
Misto D2F


Tempos para construir a árvore
segundos
140
120
100
80
60
40
20
0
nós
100
300
500
700
900
1100
1300
1500
MSA T_arv
KerLin T_arv
AR T_arv
Misto T_arv

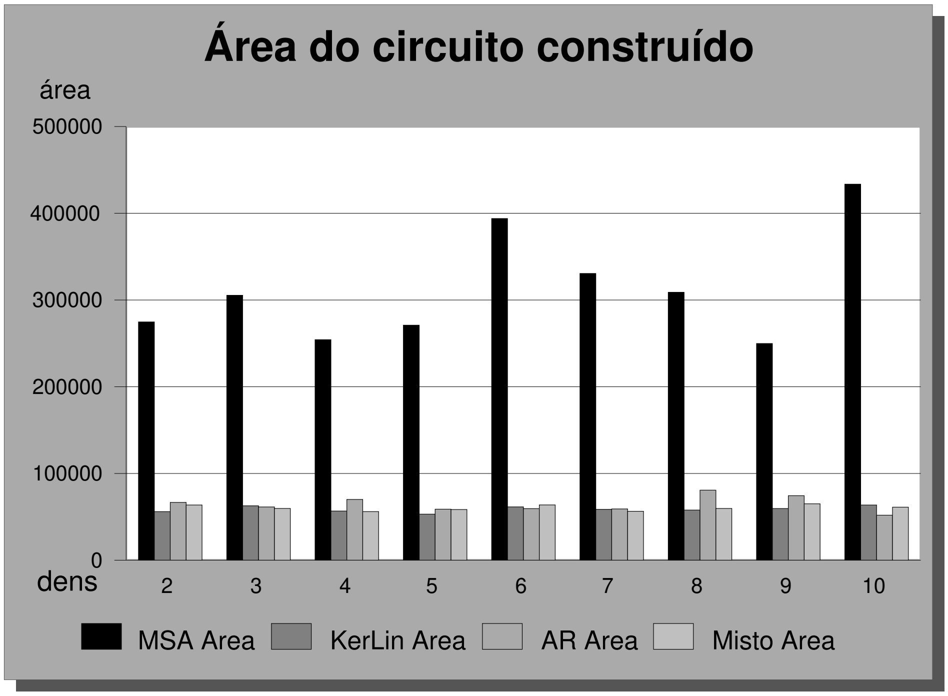

Tempos para construir o circuito
segundos
7000
6000
5000
4000
3000
2000
1000
0
nós
100
300
500
700
900
1100
1300
1500
MSA T_total
KerLin T_total
AR T_total
Misto T_total


Função Objectivo 1
O1
60000000
50000000
40000000
30000000
20000000
10000000
0
den
2
3
4
5
6
7
8
9
10
MSA O1
KerLin O1
AR O1
Misto O1

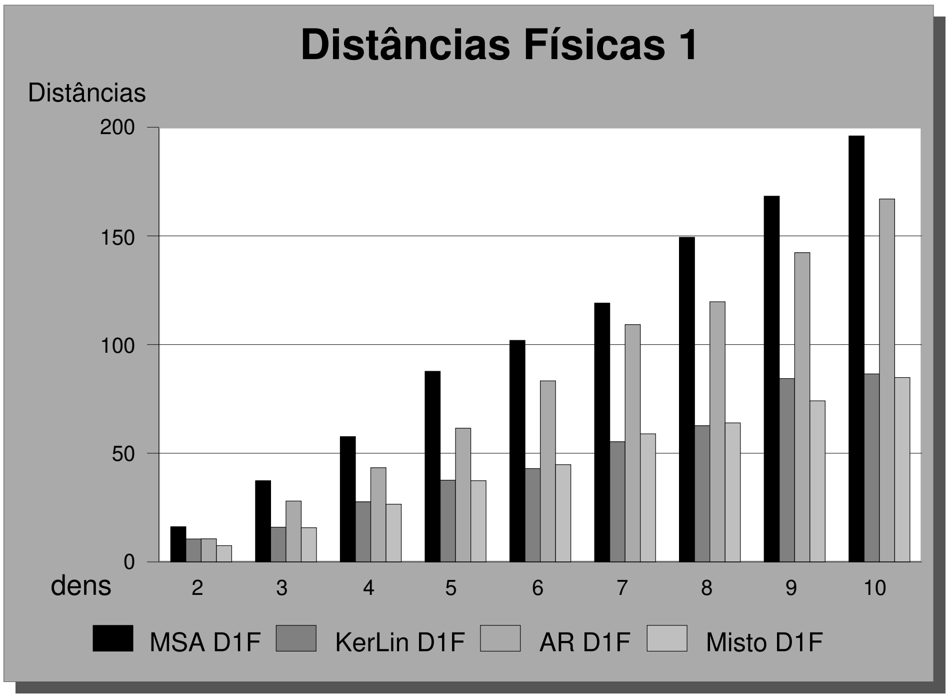

Função Objectivo 2
O2
250
200
150
100
50
0
dens
2
3
4
5
6
7
8
9
10
MSA O2
KerLin O2
AR O2
Misto O2

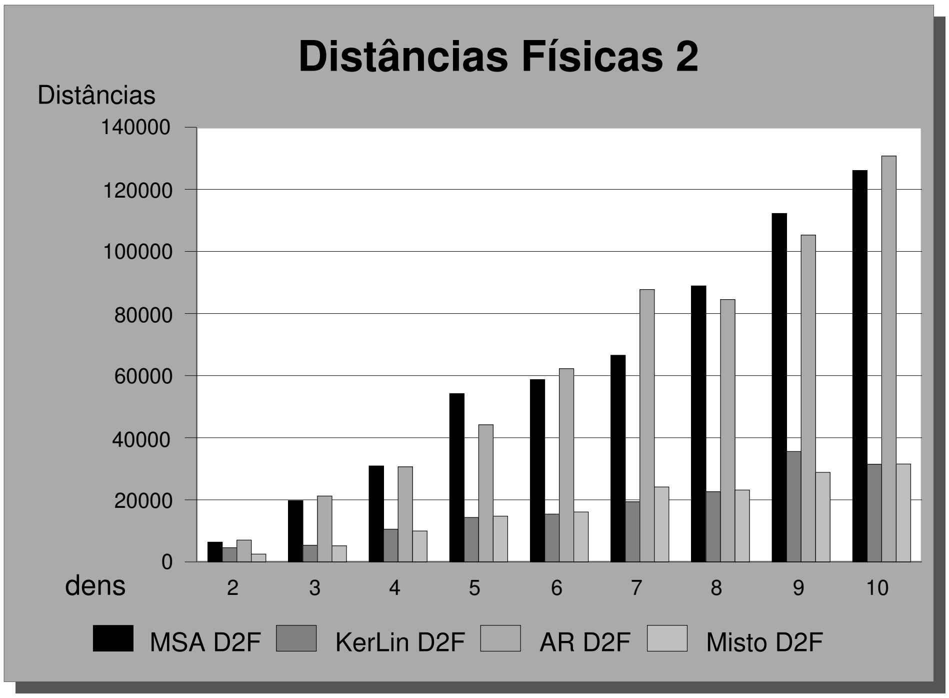


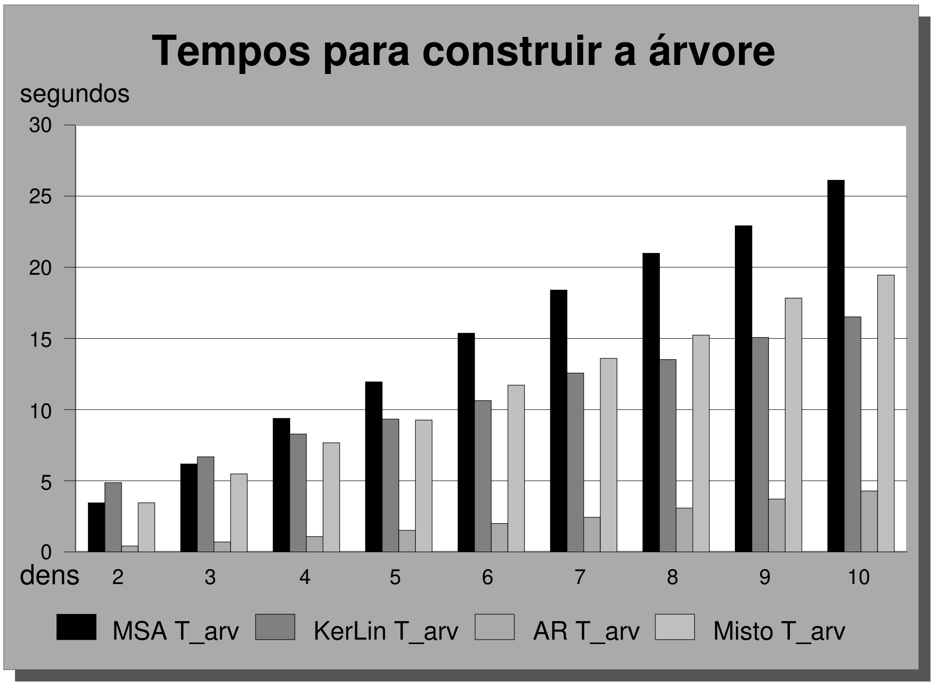
Semi-perímetro do circuito construído
área
3500000
3000000
2500000
2000000
1500000
1000000
500000
0
dens
2
3
4
5
6
7
8
9
10
MSA S_P
KerLin S_P
AR S_P
Misto S_P

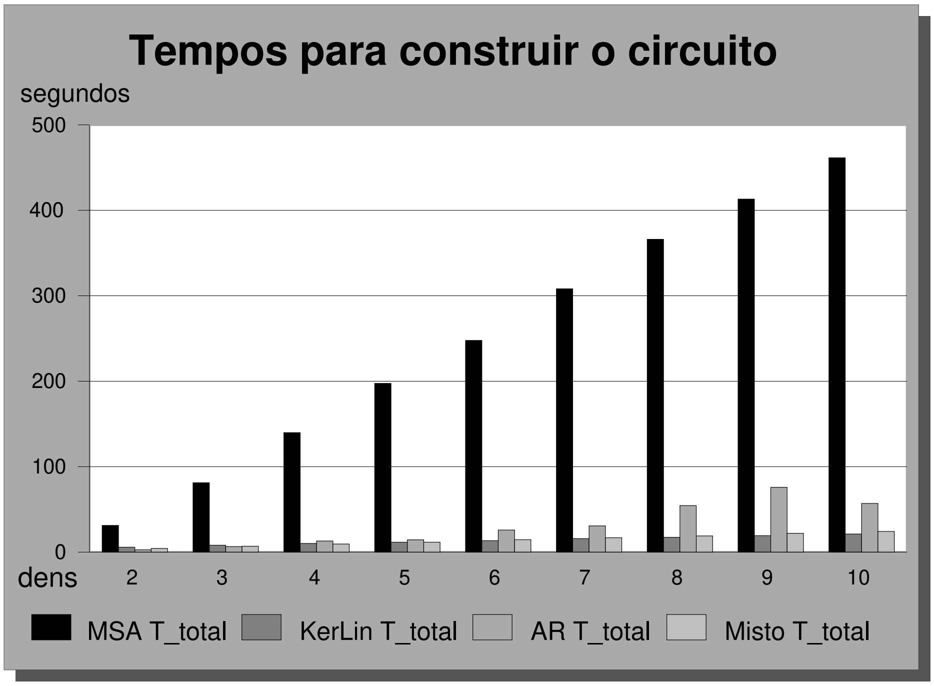


Área do circuito construído
área
500000
400000
300000
200000
100000
0
dens
2
3
4
5
6
7
8
9
10
MSA Area
KerLin Area
AR Area
Misto Area


Distâncias Físicas 1
Distâncias
200
150
100
50
0
dens
2
3
4
5
6
7
8
9
10
MSA D1F
KerLin D1F
AR D1F
Misto D1F


Distâncias Físicas 2
Distâncias
140000
120000
100000
80000
60000
40000
20000
0
dens
2
3
4
5
6
7
8
9
10
MSA D2F
KerLin D2F
AR D2F
Misto D2F


Tempos para construir a árvore
segundos
30
25
20
15
10
5
0
dens
2
3
4
5
6
7
8
9
10
MSA T_arv
KerLin T_arv
AR T_arv
Misto T_arv


Tempos para construir o circuito
segundos
500
400
300
200
100
0
dens
2
3
4
5
6
7
8
9
10
MSA T_total
KerLin T_total
AR T_total
Misto T_total

Bibliografia

[1] Almeida, A. M. C. *Definição de Objectivos para Algoritmos de Partição de Grafos-Aplicações ao Projecto Automático de Circuitos*, Relatório de Estágio da Licenciatura em Matemática, Ramo de Ciências da Computação, Departamento de Matemática da Faculdade de Ciências e Tecnologia da Universidade de Coimbra, 1990.

[2] Almeida, A. M. C. e Rodrigues M. R. D. *Objectivos para Avaliação de Algoritmos Geradores de Árvores de Partição*, Actas das XV Jornadas Luso-Espanholas de Matemática, Évora, 1990.

[3] Almeida, A. M. C. e Rodrigues M. R. D. *A New Clustering Approach to Hierarchical Layout*, Proc. Euro XI - 11th European Congress on Operational Research, Aachen, 1991.

[4] Bertsekas, D. P. e Tseng, P. *Relaxation Methods for Minimum Cost Ordinary and Generalized Network Flow Problems*, Operations Research, 36, (1988), 93-114.

[5] Bertsekas, D. P. e Tseng, P. *RELAXT-III: A new and Improved Version of the RELAX Code*, Research Report, M.I.T., Cambridge, 1990.

[6] Edmonds, J. e Karp, R. M. *Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems*, Journal of the Association for Computing Machinery, 19, (1972), 248-264.

[7] Kernighan, B.W. e Lin, S. *An Efficient Heuristic Procedure for Partitioning Graphs*, Bell Systems Technical Journal, 49, (1970), 291-307.

[8] Ford, L. R., e Fulkerson, D. R. *Flows in Networks*, Princeton University Press, Princeton, N.J., 1962.

[9] Glover, F.; Klingman, D.; Mote, J. e Whitman, D. *A Primal Simplex Variant for the Maximum Flow Problem*, Center for Cybernetic Studies, The University of Texas, Austin, Texas, 1979.

[10] Goldberg, A. V. e Tarjan, R. E. *A New Aproach to the Maximum-Flow Problem*, Journal of the Association for Computing Machinery, 35, (1988), 921-940.

[11] Johnson, D. S.; Aragon C. R.; McGeoch, L. A. e Schevon, C. *Optimization by Simulated Annealing: An Experimental Evaluation; Part I, Graph Partioning*, Operations Research, 35, (1989), 865-892.

[12] Martins, E. Q. V. e Salgueiro, R. P. M. *Um Algoritmo para o Problema do Fluxo Máximo*, Actas da 1ª Conferência em Estatística e Optimização, Tróia, 1990.

[13] Martins, E. Q. V. e Salgueiro, R. P. M. *An Incremental Chain Algorithm for the Maximal Flow Problem*, Proc. Workshop On Large-Scale Otimization, Coimbra, 1991.

[14] Rodrigues, M. R. D. *A Tree-Based Algorithm For Component Placement*, Ph. D. Thesis, University of Manchester, 1986.

[15] Salgueiro, R. P. M. *O Problema do Fluxo Máximo*, Relatório do segundo projecto da cadeira de Optimização em Redes da Licenciatura em Matemática (Ramo de Ciências da Computação), Departamento de Matemática da Faculdade de Ciências e Tecnologia da Universidade de Coimbra, 1990.