Instituto Superior Técnico

Mestrado em Eng. Electrotécnica e de Computadores

Algoritmos e Estruturas de Dados

2011/2012 – 2º Ano, 1º Semestre

Relatório do Projecto

Network Slack

Grupo nº: 38

Manuel António Seixas Pinto

nº67827 e-mail: 1.bisc.8@gmail.com

Fábio Rafael Santos Lopes

nº 70436 e-mail: fabiosantoslopes@gmail.com

Docente: **Carlos Bispo**

INDICE

Descrição do problema

Em traços gerais de modo a resumir a informação do enunciado do projecto foi nos pedido que analisássemos um circuito lógico dando ênfase ao aspecto temporal que, dado um ficheiro com a descrição de um circuito lógico, conseguíssemos calcular as folgas desse circuito.

A folga de um nó define-se como a diferença entre o tempo de chegada requerido e o tempo de chegada calculado. O tempo de chegada requerido refere-se a restrições impostas ao circuito para que possua tempos de operação adequados. O tempo de chegada calculado refere-se a propagação calculada dos atrasos na propagação de informação ao longo do circuito.

As folgas são quantidades que importam calcular, por poderem influenciar o projecto dos circuitos. Se a folga num dado nó for maior que zero, tal implica que a informação chegará a esse nó antes de ser necessária. Se não é necessária, então talvez alguns dos canais do circuito poderão ser substituídos por ligações mais lentas (de menor capacidade), o que poderá conduzir a poupanças em custo. Por outro lado, se a folga de um nó é negativa significa que a informação chegará atrasada a esse nó. O projectista do circuito poderá corrigir esta situação utilizando ligações mais rápidas (maior capacidade) entre nos.

Num segundo modo de funcionamento era pedido que se calculasse o instante de chegada agora tendo em conta a configuração de portas lógicas do circuito que é um desafio um pouco mais complicado que o anterior e também igualmente útil.

Abordagem ao problema

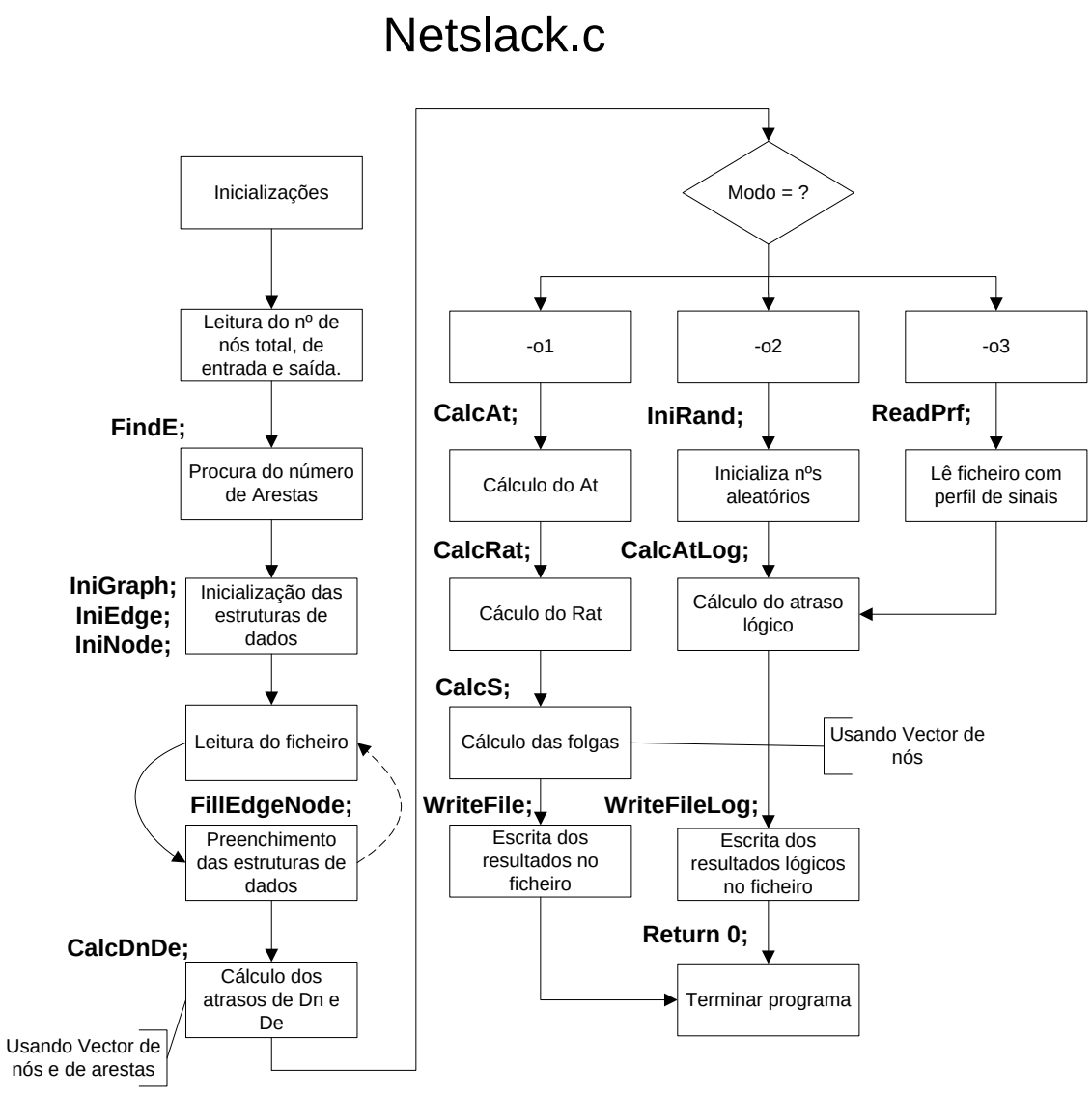
Começou-se por fazer o paralelismo entre o circuito lógico e a sua representação como um grafo direccionado. Numa primeira fase criámos apenas um grafo direccionado das entradas para as saídas representado por listas de adjacência. Para o percorrer decidimos utilizar um DFS simples para o cálculo do instante de chegada, e para o instante de chegada requerido foi utilizada uma recorrência que decorria os nós todos até chegar aos de saída e posteriormente fazia os cálculos quando ia regressando até às entradas. Embora o algoritmo funcionasse, era um processo demasiado complexo e demorado e altamente ineficiente por ter de visitar alguns caminhos mais do que uma vez.

Para contornar o problema da complexidade foi criado um grafo adicional direccionado das saídas para as entradas (representado também por listas de adjacência).

Para não estar continuamente a percorrer caminhos já visitados usou-se a estratégia de marcar nós já visitados. Uma descrição mais pormenorizada deste algoritmo e de como lá chegámos encontra-se na secção “Descrição dos algoritmos”.

Arquitectura do programa

Fluxograma do programa Netslack:



O Fluxograma anterior mostra a arquitectura do programa e condensa a informação sobre as principais funções e os seu objectivos.
As funções de CalcAt, CalcRat e CalcAtLog utilizam a estrutura de dados onde está guardado o grafo, um vector de arestas e o vector de nós.
Um estudo mais pormenorizado das estruturas de dados utilizadas e funcionamento dos algoritmos é feito mais à frente no relatório.

Descrição das estruturas de dados:

Optou-se por usar vectores de estruturas para guardar as informações dos nós e arestas de modo a se poder aceder directamente a um nó/aresta específico/a.

Vector de Arestas (struct edge)	Tamanho: E (número de arestas)
double nX;	Nó inicial;
double nY;	Nó final;
double dei;	Atraso intrínseco da aresta;
double den;	Coeficiente de carga característico da aresta;
double de;	Atraso da aresta;

Cada elemento do vector guarda a informação relativa a cada uma das arestas do grafo.

Vector de Nós (struct node)	Tamanho: V (número de nós)
char str[5];	Tipo de porta lógica;
double dno;	Atraso intrínseco do nó;
double dnc;	Atraso característico do nó;
double m;	Carga do nó destino;
int in;	Nó de Saída: in = -1;
double rt;	Instante de chegada requerido do nó;
double dn;	Atraso dos nós;
double at;	Instante de chegada real do nó;
double s;	Folga do nó;
int L;	Valor lógico do nó;

Cada elemento do vector guarda a informação relativa a cada um dos nós do grafo.

Grafo (struct graph)	Tamanho: 1
int V;	Número de nós do grafo;
int *in;	Vector de nós de entrada;
int *out;	Vector de nós de saída;
Link **adj;	Matriz de Adjacência do grafo direccionado das entradas para as saídas;
Link **adj2;	Matriz de Adjacência do grafo direccionado das saídas para as entradas;

A estrutura guarda toda a informação relativa ao grafo:

- O campo in foi criado para guardar o índice dos nós de entrada do circuito. Esta informação é relevante durante o algoritmo de pesquisa pois permite um fácil acesso aos índices das entradas do grafo.
- O campo out foi criado para guardar o índice dos nós de saída do circuito. Esta informação é relevante durante o algoritmo de pesquisa pois permite um fácil acesso às saídas do grafo.
- O campo adj guarda a informação relativa às ligações do grafo, usando uma lista de adjacências. Como se trata de um grafo direccionado, as ligações estão direccionadas das entradas para as saídas. Como no algoritmo de pesquisa vai-se estar interessado nas ligações a um nó e não a uma ligação específica, estas listas são uma forma eficiente de representar as ligações do grafo.

- O campo adj2 guarda a informação relativa às ligações do grafo, usando uma lista de adjacências. Neste caso, as ligações estão direccionadas das saídas para as entradas. Esta informação é relevante para melhorar a eficiência do algoritmo de procura que calcula o at e o at lógico.

Link (struct link)	V x (*Link) + E x (Link)
int nE;	Número da aresta ligante;
int nY;	Nó de destino;
Link *next;	Ponteiro para a próxima ligação.

A estrutura contém a informação de uma ligação do grafo:

- O campo nE é utilizado para aceder directamente à entrada nE do vector de arestas;
- O campo nY é utilizado para aceder directamente à entrada nY do vector de nós;
- O campo next é um apontador para a próxima ligação do nó;

Descrição dos algoritmos

Os principais algoritmos utilizados nos cálculos são os que se encontram nas funções **CalcDnDe**, **CalcAt**, **CalcRat**, **CalcAtLog**.

CalcDnDe

O cálculo do atraso dos nós (Dn) e das arestas (De) é feito percorrendo os respectivos vectores usando as parcelas aí incluídas. Apenas depende do tipo de grafo no cálculo do *'de*' pois neste cálculo é necessário calcular o número de arestas que saem do nó. Como é utilizada uma lista de adjacências para representação do grafo que é direccionado, o número de arestas que saem de um nó corresponde ao número de estruturas de ligação que são encontradas ao percorrer a lista correspondente ao nó.
Algoritmo usado:

Para cada aresta no vector de arestas:
Usar expressão "de = dei + m * den" para calcular o atraso da aresta
Para cada nó no vector de nós:
Inicializar o contador (f) que guarda o nº de arestas que saem do nó.
Para cada ligação na entrada do índice do nó na lista de adjacências:
Incrementar 'f'.
Usar a expressão "dn = dno + f * dnc" para calcula o atraso do nó.

No cálculo dos instantes de chegada aos nós (At) e tendo em conta a configuração lógica do circuito (AtLog) e dos instantes requeridos (Rat) é utilizado um algoritmo recursivo para percorrer o circuito. A única diferença entre algoritmos são as condições de alteração dos valores a ser calculados e o sentido do percurso. O Objectivo destes algoritmos era o de

percorrer o circuito da forma mais eficiente possível, passando por todos os caminhos relevantes, visitando os nós apenas o mínimo de vezes.

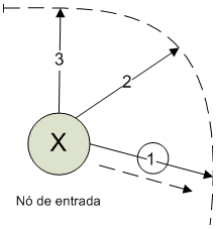
De seguida, mostra-se o exemplo do CalcAt de maneira pormenorizada, nos outros o raciocínio é semelhante e as diferenças serão mostradas.

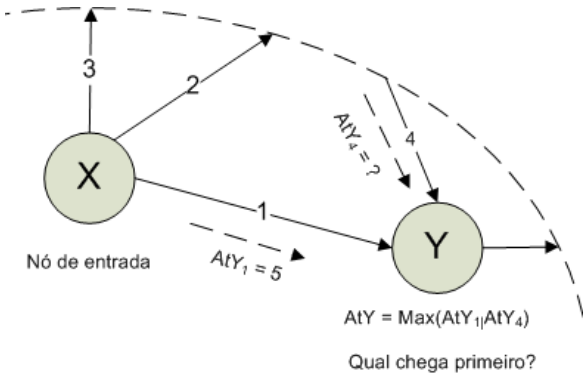
O Algoritmo teria de passar por todos os nós para calcular o seu atraso mas não estaríamos interessados em percorrer todos os caminhos do circuito pois em nós com mais do que uma aresta, o atraso a propagar para as saídas é o máximo dos instantes de chegada dos sinais que chegam das arestas. O Ideal seria que, ao chegar a um nó deste tipo, calculasse o valor máximo dos instantes de chegada para se ficar com o valor do atraso final nesse nó e assim avançar no circuito sem necessidade de voltar atrás. Com esta ideia no pensamento e adoptando uma estratégia de procura em profundidade, **DFS,** desenvolveu-se um algoritmo que, para o cálculo dos instantes de chegada, percorre o circuito em profundidade das saídas para as entradas.

Mas porquê percorrer um circuito em sentido inverso?

Imaginando que se está a percorrer um circuito lógico e pensando na sua representação em grafo direccionado.

Neste caso existem três possibilidades de iniciar o caminho, uma vez que apenas com esta informação é indiferente escolher qualquer uma das arestas escolha-se por exemplo a aresta 1.





Observa-se que a aresta 1 liga o nó X ao nó Y mas ao chegar ao nó Y observa-se que existe mais uma aresta com o nó Y como destino.

Sabe-se que o instante de chegada ao nó Y vai ser o máximo dos instantes de chegada do sinal nas duas arestas.

Como neste momento não se sabe qual é o instante de chegada do sinal na aresta 4 seria uma má decisão continuar a percorrer o circuito assumindo que o atraso no nó Y é o atraso final. Fazendo isto podia acontecer que depois ao retornar ao nó de entrada e escolhendo outra aresta para percorrer outro caminho se chegasse à conclusão que aquele não era o atraso máximo e todos os valores a partir desse nó teriam de ser alterados!

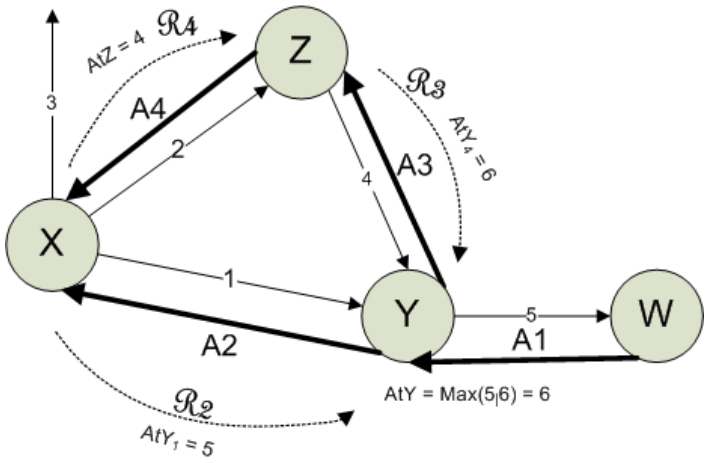
Ao chegar ao nó Y e tentando retornar ao nó de entrada para escolher outra aresta para iniciar um novo caminho na esperança que este vá terminar no nó Y resolvendo o problema, também não é uma boa política pois não há garantias que o caminho iniciado na aresta escolhida vá terminar no nó desejado.

Postos estes problemas, pensou-se num algoritmo que primeiramente percorre em profundidade o circuito, sem efectuar cálculos, das saídas para as entradas com auxílio de uma

lista de adjacências com as ligações em sentido invertido. Usando esta lógica invertida, ao chegar a um nó sem ligações lovas (inicialmente só o nó de entrada) marca-se no como visitado e a função retornaria ao nó anterior, calculava o atraso e substituía no atraso do nó, só que agora em vez retornar ao início (fim do circuito), a função vai procurar mais arestas ligadas a esse nó (que na realidade correspondem a ligações que terminam nesse nó) avançando novamente até um nó sem arestas.

Ao voltar a esse nó já pode ser calculado o máximo dos instantes de chegada e avançar no circuito (retornar na função). Desta maneira garante-se que a função só retorna quando os tempos de atraso desse nó para trás são os tempos finais. A estratégia de marcar os nós já visitados é vantajoso porque um nó já visitado significa um nó que não tem arestas novas por visitar logo o tempo de atraso é o tempo final fazendo com que se visite uma única vez cada aresta.

No exemplo de cima o algoritmo comportar-se-ia desta maneira, com atenção que se estão a usar as ligações inversas:



Inicializando no nó de Saída ‘W’:

A1 – A aresta 5 é a única aresta de saída (entrada no circuito real), avançar para o nó Y

Escolher uma aresta possível, aresta 1.

A2 – Avançar para o nó X.

Não existem arestas disponíveis

Marcar nó como visitado

Retornar

R2 – Retorno para o nó Y, cálculo do AtY_1

Existe uma aresta ainda não visitada, aresta 4

A3 – Avançar para o nó Z

Escolher aresta 2

A4 – Avançar para o nó X.

Nó X já visitado

Retornar

R4 – Retorno para o nó Z, cálculo do AtZ

Não existem novas arestas para visitar

Retornar

R3 – Retorno para o nó Y, cálculo do AtY_4

Valor de $AtY_4 > AtY_1$, substituir.

Não existem mais arestas

Retornar

Desta maneira o algoritmo consegue calcular os instantes de chegada de forma eficiente.

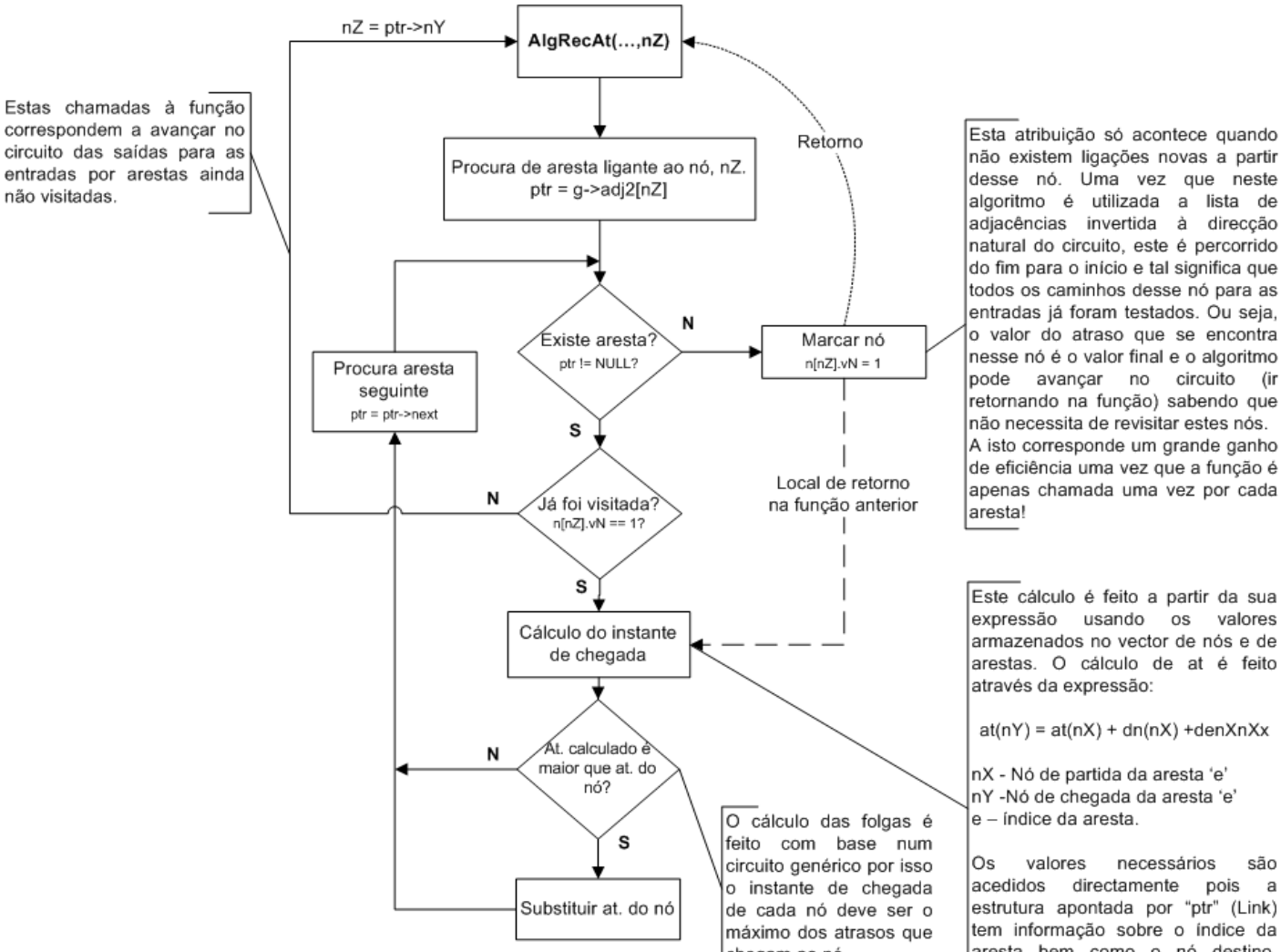
Funções implementadas com este algoritmo:

CalcAt

Para cada nó de saída

Chamar a função AlgRecAt(…,nZ) enviando, entre outros o índice do nó como nZ.

AlgRecAt (Fluxograma detalhado)

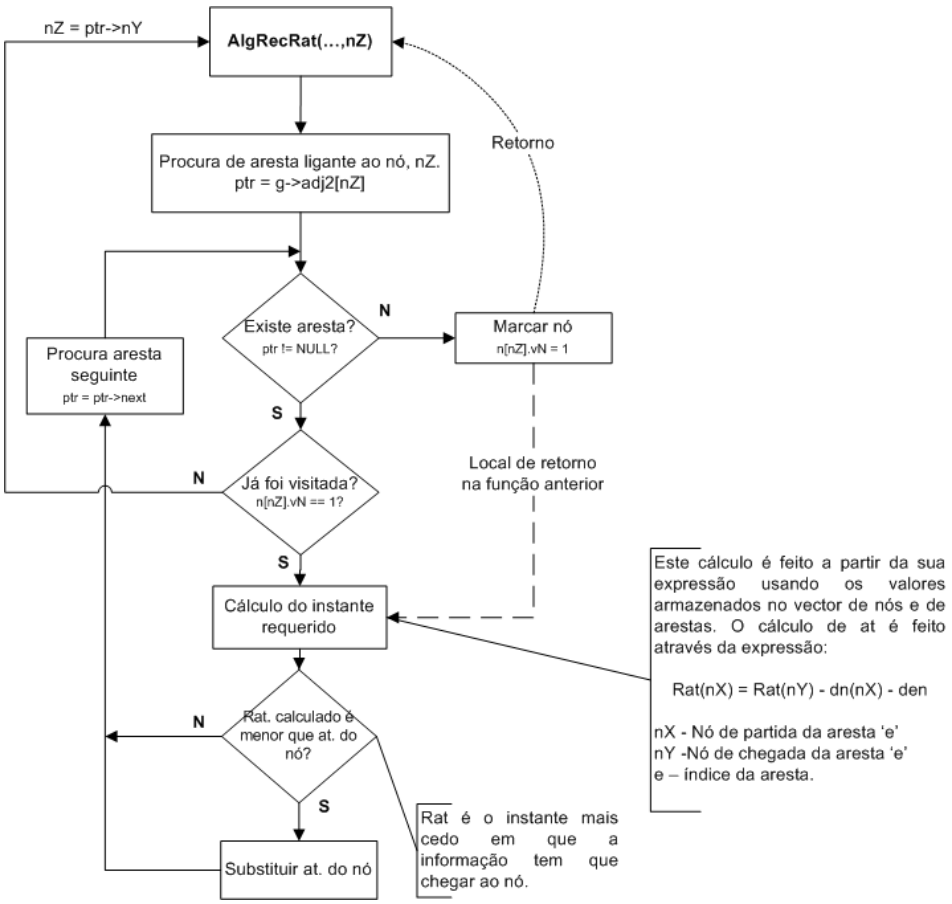


CalcRat

Para cada nó de entrada

Chamar a função AlgRecRat(…,nZ) enviando, entre outros o índice do nó como nZ.

AlgRecRat



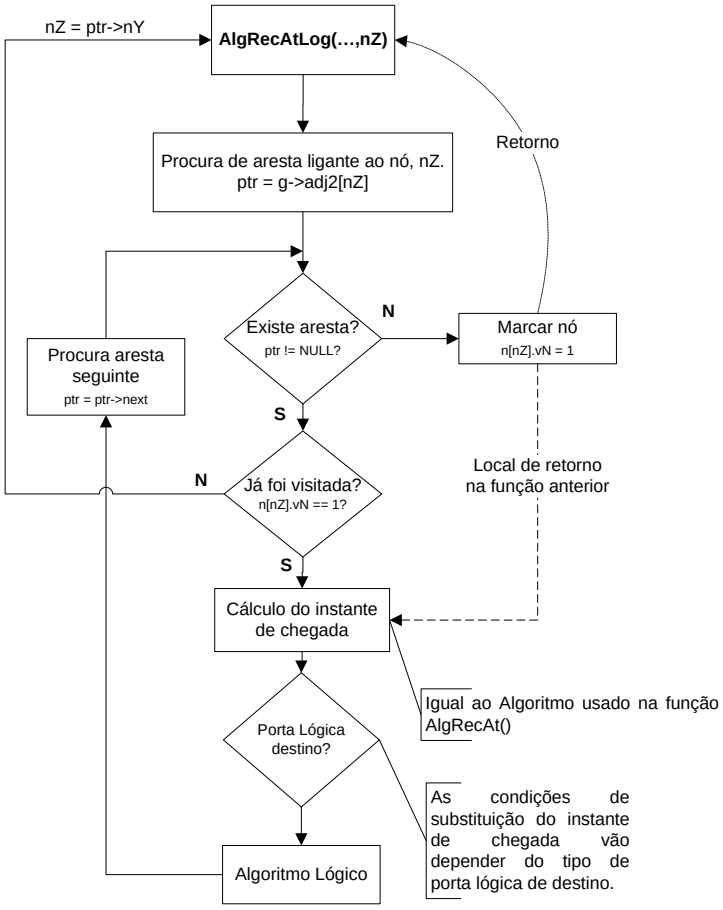
Como neste cálculo, queremos observar a propagação do Rat desde as saídas, usa-se a lista de adjacências de sentido normal e no algoritmo o circuito é percorrido das entradas para as saídas.

CalcAtLog

Para cada nó de saída

Chamar a função AlgRecAtLog(…,nZ) enviando, entre outros o índice do nó como nZ.

AlgRecAtLog



Nesta última função torna-se especialmente complicado calcular os atrasos porque, ao contrário do cálculo do At para um circuito lógico genérico com um perfil de ligações, neste caso é preciso ter em conta os tipos de portas lógicas. Como tal num nó com várias arestas incidentes o sinal de saída pode ser estabilizado antes do tempo máximo de chegada de todos os sinais.

Para resolver este problema é necessário analisar as características de cada porta lógica individualmente. O pensamento usado foi, para um sinal que chega a um nó verificar a influência que esse valor lógico tem no valor lógico do nó destino bem como o atraso tendo em conta as funções lógicas.

Como tal fez-se uma tabela para cada porta lógica com todas as combinações de bits possíveis e a influência no valor lógico e no instante de estabilização. Nas tabelas só são feitas as combinações do sinal que chega com o sinal que se encontra no nó pois nas funções lógicas Booleanas pode-se aplicar a propriedade associativa. Quer isto dizer que o resultado de uma função lógica com várias parcelas é igual a aplicar a função a duas parcelas e usar o resultado nas restantes parcelas.

Para cada função, é enviado o instante de chegada que designaremos por *aux*. Todos os nós são inicializados com os valores lógicos '-1' (desconhecido). LY é o valor lógico que se encontra no nó destino e LX o valor lógico que chega ao nó.

Em todas as portas lógicas, excepto NOT, o primeiro sinal a chegar é guardado, bem como o seu instante de chegada não sendo aplicada nenhuma operação.

LY	LX	L	At
-1	0	0	aux
-1	1	1	aux

AND	LY	LX	L	AT
	0	0	0	MIN
	0	1	0	-
	1	0	0	AUX
	1	1	1	MAX
NAND				
	0	0	1	MIN
	0	1	1	-
	1	0	1	AUX
	1	1	0	MAX
XOR				
	0	0	0	MAX
	0	1	1	MAX
	1	0	1	MAX
	1	1	0	MAX

OR	LY	LX	L	AT
	0	0	0	MAX
	0	1	1	AUX
	1	0	1	-
	1	1	1	MIN
NOR				
	0	0	1	MAX
	0	1	1	AUX
	1	0	1	-
	1	1	0	MIN
NOT				
	-1	0	1	AUX
	-1	1	0	AUX
BUF				
	-1	0	0	AUX
	-1	1	0	AUX

Descrição dos subsistemas:

O programa por nós implementado tem o subsistema **Graph*,*** contituido por ***Graph.c*** e ***Graph.h*** onde estão guardadas as funções que estão directamente ligadas à criação e manipulação de dados nas estruturas de dados. As estruturas de dados estão definidas no ficheiro ***Graph.h***. Existe outro subsistema **File** constituído por ***File.c*** e ***File.h*** que contém as funções directamente relacionadas com a leitura ou manipulação de dados em ficheiros de texto. Existe ainda o ficheiro ***Defs.h*** que contém a informação do valor definido para o tamanho máximo de uma linha na leitura de um ficheiro.

Graph.c

Funções de inicialização:

Graph ***IniGraph** (int V, int in, int out);

Inicializa os vários parâmetros contidos na estrutura de representação do grafo.

Node ***IniNode** (int V);

Inicializa o vector de nós com V número de elementos.

Inicializa o atraso de cada nó com o valor 0.

Edge ***IniEdge**(int E);

Inicializa o vector de arestas com E número de elementos.

Função de criação de nova ligação:

void **NewLink**(Graph *g, int nX, int nY, int nE);

Cria uma estrutura de ligação para colocar na lista de adjacências.

Esta estrutura é inserida no início da lista.

É também feito o mesmo para a lista de adjacências inversa, trocando os valores.

Funções de inicialização de valores:

void **IniRat** (Node *n,int V, double rmax);

Inicializa os nós que não sejam de saída com o rmax.

void **IniLogic**(Node *n, int V);

Inicializa os nós com o valor lógico indefinido '-1'.

void **IniRand**(Graph *g,Node *n,int V,int in);

Cria um perfil de valores lógicos de entrada e aplica-os ao nó.

Caso a porta de entrada seja uma porta NOT o valor lógico a atribuir nessa porta é o complementar.

Funções de cálculo:

void **CalcDnDe** (Graph *g, Edge *e, int E, Node *n, int V);

Calcula o atraso dos nós e das arestas.

void **CalcAt** (Graph *g,Edge *e, int E, Node *n, int V, int out);

Inicializa nós como não visitados .

Chama função AlgRecAt para cada nó de saída.

int **AlgRecAt** (Graph *g, Edge *e, int E, Node *n, int V, int nZ);

Função recursiva que percorre o circuito das saídas para as entradas numa estratégia de marcar os nós já visitados e percorrer as arestas uma única vez.

Calcula o atraso máximo de cada nó.

void **CalcRat** (Graph *g,Edge *e, int E, Node *n, int V, int in);

Inicializa nós não visitados

Chama função AlgRecRat para cada nó de entrada

int **AlgRecRat** (Graph *g, Edge *e, int E, Node *n, int V, int nZ);

Função recursiva que percorre o circuito das entradas para as saídas numa estratégia de marcar os nós já visitados e percorrer as arestas uma única vez.

Calcula o atraso requerido mínimo de cada nó.

void **CalcAtLog** (Graph *g, Edge *e, int E, Node *n, int V, int out);

Inicializa nós não visitados.

Chama função AlgRecLog para cada nó de saída.

int **AlgRecAtLog** (Graph *g, Edge *e, int E, Node *n, int V, int nZ);

Função recursiva que percorre o circuito das saídas para as entradas numa estratégia de marcar os nós já visitados e percorrer as arestas uma única vez.

O Atraso e os valores lógicos são calculados com auxilio de uma função para cada tipo de porta lógica.

void **CalcS** (Node *n, int V);

Função que percorre o vector de nós e aplica o algoritmo de cálculo das folgas para cada nó.

Funções com os algoritmos de portas lógicas:

void **NOT** (Node *n, double aux, int nX, int nY);

Calcula o valor lógico com base no valor lógico que se encontra no nó aplicando o algoritmo da porta NOT

Como uma porta NOT tem apenas uma entrada o atraso acumulado vai ser o atraso final do nó.

void **BUF** (Node *n, double aux, int nX, int nY);

Calcula o valor lógico com base no valor lógico que se encontra no nó aplicando o algoritmo da porta BUF

Como uma porta BUF tem apenas uma entrada o atraso acumulado vai ser o atraso final do nó.

void **NAND** (Node *n, double aux, int nX, int nY);

Calcula o valor lógico com base no valor lógico que se encontra no nó aplicando o algoritmo da porta NAND.

Usando o algoritmo da porta NAND o atraso é apenas guardado se se verificarem as condições de valor lógico.

void **NOR** (Node *n, double aux, int nX, int nY);

Calcula o valor lógico com base no valor lógico que se encontra no nó aplicando o algoritmo da porta NOR.

Usando o algoritmo da porta NOR o atraso é apenas guardado se se verificarem as condições de valor lógico.

void **AND**(Node * n, double aux, int nX, int nY);

Calcula o valor lógico com base no valor lógico que se encontra no nó aplicando o algoritmo da porta AND.

Usando o algoritmo da porta AND o atraso é apenas guardado se se verificarem as condições de valor lógico.

void **OR**(Node * n, double aux, int nX, int nY);

Calcula o valor lógico com base no valor lógico que se encontra no nó aplicando o algoritmo da porta OR.

Usando o algoritmo da porta OR o atraso é apenas guardado se se verificarem as condições de valor lógico.

void **XOR** (Node *n, double aux, int nX, int nY);

Calcula o valor lógico com base no valor lógico que se encontra no nó aplicando o algoritmo da porta XOR.

Usando o algoritmo da porta XOR o atraso é apenas guardado se se verificarem as condições de valor lógico.

Funções de libertação de memória

void **FreeAdjList**(Graph *g, int V);

Função que liberta a memória alocada para as duas listas de adjacências.

void **FreeMem**(Graph *g,Node *n, Edge *e, int V, int E, int in);

Função que liberta o resto da memória alocada.

File.c

Funções de abertura de ficheiros

FILE ***OpenFile**(char *fname, char *mode);

Abre o ficheiro no modo desejado.

FILE ***OpenOutFile**(char *inFile);

Cria o nome do ficheiro de saída alterando a extensão .in para .out ao ficheiro de entrada.

Abre o ficheiro de saída preparando-o para escrita.

Funções de leitura de ficheiros

int **FindE**(FILE *fp);

Calcula o número total de arestas através da análise do ficheiro.

void **ReadPrf**(Node *n, int V, char *fname);

Abre o ficheiro com o perfil de sinais de entrada do circuito.

Atribui os valores lógicos às entradas dos circuitos.

Inverte o valor lógico na entrada caso se trate de uma porta NOT.

void **FillEdgeNode**(Graph *g, Edge *e, Node *n, FILE *fp, int V);

Lê o ficheiro com a configuração do grafo e guarda os valores nas estruturas de dados para reapresentação do grafo.

Calcula o instante de chegada requerido máximo das saídas.

Funções de escrita no ficheiro

void **WriteFile**(Node *n, int V, FILE *fp2);

Escreve no ficheiro de saída os valores calculados

void **WriteFileLog**(Node *n, int V, FILE *fp2);

Escreve no ficheiro de saída os valores calculados para o modo lógico

Análise dos requisitos computacionais:

As várias escolhas para estruturas de dados do problema foram tidas em conta face à maneira como iriam ser acedidas de modo a minimizar a complexidade.

Vector de nós e de arestas:

Continuamente por todo o programa era necessário aceder a valores sobre determinado nó e determinada aresta, como tal escolheu-se vectores para representar estes dados, assim por cada vez que fosse necessário aceder a um nó/aresta específico esse acesso era feito directamente com complexidade **O(1)**. Neste caso a memória armazenada é proporcional ao número de nós e ao número de arestas já que a memória alocada para cada nó ou para cada aresta é a mesma. São estes vectores que consomem a maior parte da memória necessária para correr o programa.

Vector de nós de entrada e de saída:

Estes vectores contêm o índice dos nós de entrada e de saída. Quando os elementos destes vectores são necessários, são necessários sequencialmente pelo que a complexidade quando é acedido um elemento é de **O(1).**

A memória armazenada é proporcional ao número de nós de saída e de entrada.

Por cada elemento do vector guardar apenas um inteiro e os nós de entrada e de saída serem uma pequena parte do total dos nós, a memória alocada para estes vectores é pequena comparando com a memória total.

Grafo:

Na estrutura onde são armazenadas as informações do grafo são criadas, duas listas de adjacências, um inteiro um vector de nós de saída e um vector de nós de entrada.

Lista de adjacências:

Na lista de adjacências é alocada memória para cada ligação. Durante as várias etapas de processamento é apenas necessário aceder ao primeiro elemento da lista e seguidamente aos próximos. Ou seja nunca é preciso aceder a um elemento específico pelo que a complexidade quando se quer aceder a um dos termos desta lista, seja o primeiro ou o próximo será **O(1).** Como esta lista guarda ligações de cada nó a memória alocada é proporcional ao número de nós e ao número de arestas, mas como por cada nó é apenas alocado um ponteiro, a influência do número de arestas na memória alocada é maior.

Existem duas listas de adjacências com o mesmo número de elementos mas ligadas de maneira diferente pelo que a memória total gasta com a representação do grafo é o dobro. À primeira vista esta segunda lista pode parecer supérflua, já que se vai utilizar mais memória para representar as mesmas ligações de maneira diferente, no entanto apesar de um maior consumo de memória esta segunda lista leva a um grande diminuição do número de operações nos algoritmos de que percorrem o circuito aumentando a *performance* do programa. Como o aumento da memória não é muito significativo, achamos que analisando as vantagens e desvantagens descritas é uma boa opção.

Principais algoritmos:

CalcDnDe();

Este algoritmo percorre o vector de arestas realizando o cálculo do atraso nas mesmas, logo esta parte é proporcional ao número de arestas. Na segunda parte é percorrido o vector de nós mas é necessário calcular o número de arestas que sai do nó pelo que a complexidade vai ser proporcional a V + E.

AlgRecAt(), **AlgRecRat**(), **AlgRecLog**();

Nestes algoritmos como foi visto anteriormente, cada aresta é apenas percorrida um única vez e são executadas aproximadamente o mesmo número de instruções cada vez que esta função é chamada, como tal pode afirmar-se que a complexidade dos algoritmos é proporcional ao número de arestas, **O(E).**

As funções AlgRecAt() e AlgRecRat() têm complexidade muito semelhante pois na verificação do valor a substituir no nó, uma realiza a operação de máximo e a outra a de mínimo. Na função AlgRecAtLog() apesar de ser chamada o mesmo número de vezes e de ser proporcional a E, o algoritmo de substituição é mais complexo e exige mais verificações por isso esta função é mais exigente computacionalmente.

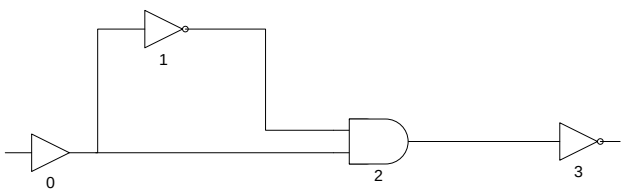
Funcionamento do programa

Inicialmente o programa passava em apenas em 9 dos 13 testes fornecidos pelo corpo docente, mas depois de identificado o problema e feito um pequeno ajuste para um tipo de circuito que não tinha sido pensado inicialmente, o programa funcionou correctamente passando nos 13 testes.

Uma vez que todas as decisões de escolha de estruturas de dados, algoritmos e estratégia foram feitas tendo em conta o seu fim de modo a ser o mais eficiente possível conciliando memória e complexidade pensamos que o projecto foi bem conseguido e os pensamentos descritos neste relatório foram bem implementados.

Exemplo:

Para este circuito lógico simples:



E para o perfil de entrada:

```
4 1 1
i n0
o n3
n n0 BUF 1 0.5 1
n n1 NOT 0.5 1 1
n n2 AND 1 1 0.5
n n3 NOT 0 0 1
e n0 n1 1 1
e n0 n2 1 1
e n1 n2 1 1
e n2 n3 1 1
a n0 0
r n3 12
```

1 – Ler número de nós, número de nós de entrada e número de nós de saída

V = 4, ni = 1, out = 1

2 – Depois de inicializar as estruturas de dados, guardar a informação do ficheiro nos vectores de nós e arestas através da função FillEdgeNode;

Listas:

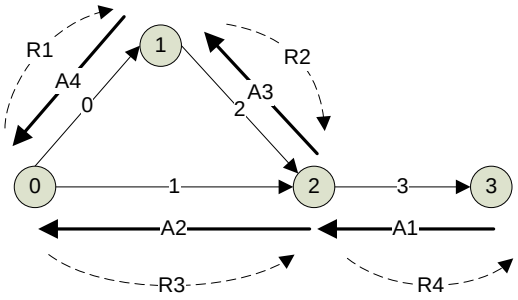
Lista adj

[0]->(2|1)->(1|0)
[1]->(2|2)
[2]->(3|3)
[3]

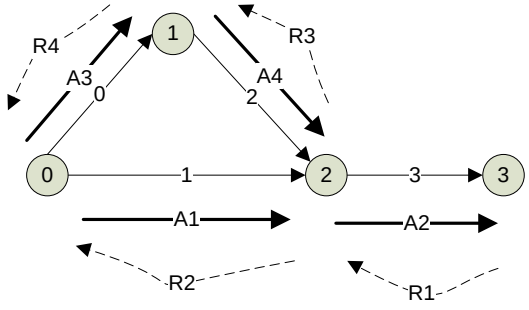
Lista adj2

[0]
[1]->(0|0)
[2]->(1|2)->(0|1)
[3]->(2|3)

Seguidamente é calculado o Dn e o De com a função CalcDnDe;
Depois para o cálculo do At é usada a função CalcAt que segue este caminho:



Os índices indicam a ordem.
Depois é calculado o Rat com este caminho:



E por fim são calculadas as folgas.

O cálculo do atraso lógico sege um caminho igual ao do ao do At fazendo umas verificações diferentes.
O Output final no ficheiro é:
n0 1.00 0.00 1.00
n1 5.00 4.00 1.00
n2 8.00 7.00 1.00
n3 12.00 11.00 1.00

Se atribuíssemos um valor lógico 0 à entrada o output seria:
n 0 0 0.0
n1 1 4.0
n2 0 3.5
n3 1 7.5