

DOMINÓS

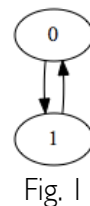
Descrição do Problema

Dado um determinado conjunto de dominós, e a sua disposição, qual o número mínimo de dominós que são necessários derrubar manualmente para que todos os dominós sejam derrubados? No input é fornecida a quantidade de dominós existentes, e uma listagem de pares (a,b), indicando que o dominó 'a' derruba o dominó 'b'.

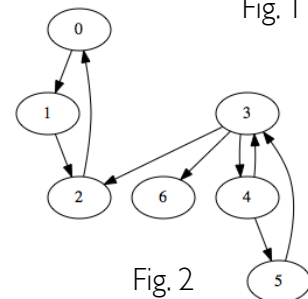
Abordagem ao Problema

Desde início que concluí que a melhor representação para este problema seria um grafo dirigido, em que cada dominó seria um vértice, e cada "derrube" seria representado por um arco. Um arco entre os vértices u e v representaria que o dominó u derrubaria o dominó v (caso fosse derrubado).

Numa primeira fase, pensei ser possível descobrir a solução apenas contabilizando a quantidade de dominós não presentes na segunda coluna do input (dominós não derrubáveis por outros). Isto levantou-me alguns problemas no que respeitava a "ciclos" (Fig. 1) - quando todos os dominós eram derrubados por outros - a resposta nunca poderia ser inferior a um, contudo o meu 'algoritmo' iria indicar que seria zero.



Tentei dar a volta a esse problema de detecção de "ciclos" com alguns métodos bastante simples (e eficientes), que funcionavam para os casos mais simples. Contudo, chegou a um ponto que não me foi possível apenas identificar e compensar na solução a existência destes "ciclos" - formulei um caso de teste (Fig. 2) que não me permitiu continuar esta linha de pensamento.



Após tentar ver o problema de outras formas diferentes (sem sucesso), concluí que a detecção destes tais "ciclos" seria obrigatória, e que caso contrário nunca conseguiria um algoritmo que calculasse de forma fiável a solução para o problema. O nome correcto destes tais "ciclos" em grafos, após investigar, descobri ser **Componentes Fortemente Ligadas** - CFLs (ou SCCs - Strongly Connected Components, em inglês) [1].

Solução

A primeira versão de uma solução (que soubesse funcionar) para este problema foi inspirada numa das aulas práticas da cadeira: começaria por identificar as componentes fortemente ligadas e criar o **grafo de componentes**, ou **grafo condensado** (ComponentGraph/CondensationGraph - a cada vértice desse grafo corresponderia um SCC no grafo original, e a cada arco entre cada par de vértices u e v no grafo original

corresponderia a, no máximo, um arco entre o SCC de u e o SCC de v no grafo de componentes. Arcos duplicados entre cada SCC, ou arcos com mesmo SCC de origem e destino seriam eliminados por não serem relevantes para o problema).

Após a construção deste grafo acíclico, poderia finalmente aplicar o meu rudimentar algoritmo formulado logo no início do meu estudo deste problema (explicado acima).

Após uma breve pesquisa, descobri que o método mais eficiente conhecido para a detecção de SCCs é o **algoritmo de Tarjan [2]**. Este, dado um grafo $G=(V,E)$, devolve uma lista de conjuntos de vértices (cada conjunto representa um SCC, e cada vértice dentro desse conjunto é um dos vértices que compõe esse SCC), em tempo e memória linear (apenas caso a procura na pilha de nós seja efectuada em $O(1)$, o qual é possível facilmente utilizando um vector de “presenças” dos vértices que estão dentro da pilha - cada vez que é feito PUSH de um vértice, marca-se a posição respectiva do vértice a “true”, e cada vez que é feito POP, marca-se a posição respectiva do vértice a “false”. Para verificar se um vértice está ou não na pilha, apenas é necessário fazer a leitura de uma posição do vector, ao invés de percorrer o vector à sua procura, que corre em tempo linear).

Para evitar cálculos desnecessários, determinei que não é necessário construir novo grafo - apenas necessito de contar os SCCs que não são atingíveis a partir de nenhum outro SCC (i.e. não há nenhum caminho/arco a partir de outro SCC para estes).

Implementação

A solução foi implementada na linguagem C, standard de 1999 (devido à utilização das bibliotecas `stdbool.h` e `stdint.h`).

Segue-se uma breve listagem das estruturas de dados relevantes utilizadas - o que representam e que operações são possíveis sobre as mesmas. Segue-se a notação `nome_estrutura(tipo1 nome_campo1, tipo2 nome_campo2, ...)` para as estruturas e `nome_função(tipo1 nome_argumento1, tipo2 nome_argumento2, ...): tipo_retorno` para as funções sobre essa mesma estrutura. “uint” representa um tipo de dados inteiro positivo. “int” representa um tipo de dados inteiro com sinal. A utilização de uma pilha deve-se ao facto de suportar todas as operações necessárias em tempo constante.

- Pilha

Estrutura **`stack_item_t(uint value, stack_item_t* next)`** - representa um ítem na pilha, em que ‘value’ é o seu valor e ‘next’ é um ponteiro para o elemento seguinte. Estrutura “`stack_item`” é um ponteiro para uma estrutura do tipo “`stack_item_t`”.

Estrutura **`stack_t(stack_item_t* top, uint size)`** - representa uma estrutura do tipo pilha. ‘top’ é um ponteiro para o ítem no topo da pilha, e ‘size’ indica a quantidade de elementos presentes na pilha. Estrutura “`stack`” é um ponteiro para uma estrutura do tipo “`stack_t`”.

- Função **`stack_create(): stack`** - Devolve uma nova pilha vazia - $O(1)$
- Função **`stack_push(uint valor): void`** - Insere ‘valor’ no topo da pilha - $O(1)$

- Função **stack_pop(): uint** - Elimina o valor no topo da pilha e retorna-o - $O(1)$

- Lista

Esta estrutura é idêntica à estrutura 'Pilha', com exceção do tipo de valores armazenado. É uma lista de pilhas. A criação desta estrutura deve-se apenas à ausência de tipos parametrizados na linguagem C, e à não utilização de tipos genéricos ('void*').

- Caso de Teste (Testcase)

Estrutura **testcase_t(uint num_dominos, stack drops[])** - representa um caso de teste: 'num_dominos' indica quantos dominós tem o caso de teste, e 'drops' é uma representação do grafo em formato de lista de adjacências - cada posição do vector é uma pilha correspondente aos arcos que partem de cada vértice.

- Tarjan

Estrutura **tarjan_t(testcase testcase, uint index, stack stack, bool *stack_contents, uint *vertex_index, uint *vertex_lowlink, list sccs)** - estrutura auxiliar para a execução do algoritmo de Tarjan. 'testcase' indica o caso de teste com o input para o algoritmo de Tarjan; 'index' é o índice da procura em profundidade; 'stack' é uma pilha de vértices, 'stack_contents' é um vector auxiliar para determinar se um nó está na pilha de vértices em $O(1)$; 'vertex_index' e 'vertex_lowlink' são vectores que têm informação sobre os valores 'index' e 'lowlink' dos vértices; 'sccs' é a lista de SCCs, em que cada SCC é um conjunto (pilha) de vértices.

Em seguida apresento a função **count_unreachable_sccs(testcase tc)**, que dado um caso de teste, devolve o custo da solução óptima (mínimo de derrubes manuais).

```

1: function count_unreachable_sccs:
2:   - input: testcase
3:   - output: number of unreachable SCCs in the input graph
4:   // V = number of dominos in input testcase

5:   SCCs = list of SCCs in input graph (given by Tarjan's algorithm)

6:   // Map domino number to corresponding SCC number
7:   scc_number = array of integers, with length V.
8:   for each scc in SCCs do
9:     for each vertex v in SCC do
10:      scc_number[v] = SCC

11:   // Array to check which SCCs have edges to them from other SCCs
12:   stamp = array of integers, same length as 'SCCs' list
13:   initialize every stamp array position to 1
14:   for each scc in SCCs do
15:     for each vertex v in SCC do
16:       for each edge e of V do
17:         w = destination vertex of edge e
18:         if (v != w) then
19:           stamp[scc_number[w]] = 0;

20:   return the sum of 'stamp' array positions

```

Análise da Complexidade da Solução Implementada

Seja 'N' o número de casos de teste, 'V' o número de dominós de um determinado caso de teste (ou vértices no grafo), 'E' o número de "quedas" (arcos entre pares de vértices num caso de teste).

A leitura de um caso de teste tem complexidade espacial $O(V+E)$, e temporal $O(E)$.

O algoritmo de Tarjan para Componentes Fortemente Ligados corre em tempo e espaço $O(V+E)$, para um determinado grafo $G=(V,E)$ como input, em representação lista de adjacências (e com a optimização anteriormente descrita). O número de SCCs é sempre inferior a V. A soma dos vértices em cada SCC é igual a V.

A função 'count_unreachable_sccs' tem complexidade temporal e espacial $O(V+E)$. A execução da linha 5 tem complexidade $O(V+E)$, dada pelo algoritmo de Tarjan. O array criado na linha 7 ocupa $\Theta(V)$ e assume-se que este é criado em tempo constante. Os ciclos na linha 8 e 9 fazem $\Theta(V)$ iterações. O vector criado na linha 12 ocupa $\Theta(V)$ e assume-se que este é criado em tempo constante. A sua inicialização na linha 13 é feita em $O(V)$. Os loops das linhas 14-16 fazem $O(V+E)$ iterações - no máximo cada arco é visitado uma vez, e cada vértice é visitado uma vez (mais as vezes que são referenciados por arcos vindos de outros vértices). Na linha 17, o vértice de destino é obtido em $O(1)$. Na linha 18, o teste é feito em $O(1)$, e a actualização do vector na linha 19 é feita em $O(1)$.

A função principal do programa é a seguinte:

```
1: function main:
2:   num_tests = get_num_tests();
3:   for i=0; i<num_tests; i++
4:     testcase = read_test_case();
5:     print count_unreachable_sccs(testcase);
6:     testcase_destroy(testcase);
```

A função `get_num_tests()` lê um inteiro do input e devolve-o - tempo e espaço constante. A função `read_test_case()` lê um caso de teste do input, em tempo $\Theta(E)$, e produz uma estrutura `testcase_t`, a qual ocupa espaço $O(V+E)$.

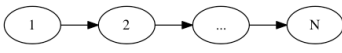
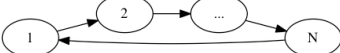
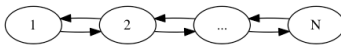
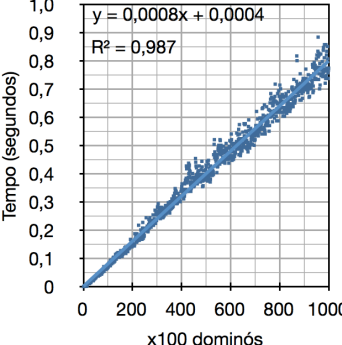
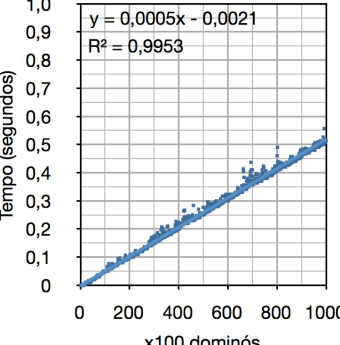
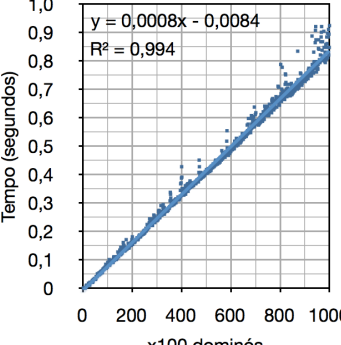
A destruição de testcases (`testcase_destroy()`) demora $O(E)$ - é necessário libertar memória para cada um dos arcos individualmente. Assume-se que a libertação de uma região de memória é feita em tempo constante.

O ciclo na linha 3 é feito $\Theta(N)$ vezes.

É seguro afirmar que o programa tem complexidade temporal $O(N*(V+E))$, e complexidade espacial $O(V+E)$ pois apenas é mantida em memória uma única instância de caso de teste de cada vez.

Avaliação experimental

Para confirmar a complexidade da solução, indicada na secção anterior, foram criados quatro tipos de testes, aos quais se irá variar o tamanho do input (entre 1 e 1.000.000 dominós). Irá utilizar-se 'N' para referir ao número total de dominós em cada caso de teste.

Teste 1	Teste 2	Teste 3
		
Existe um arco entre os dominós i e $i+1$, para qualquer i entre 1 e $N-1$.	Existe um arco entre os dominós i e $i+1$, para qualquer i entre 1 e $N-1$, e entre o dominó N e o dominó 1.	Existe um arco entre cada dominó i e $i+1$, e entre cada dominó $i+1$ e i , para qualquer i entre 1 e $N-1$.
		

Cada um dos casos de teste foi incluído 10 vezes no ficheiro, com N a variar entre 100 e 10.000 em incrementos de 100. Os resultados apresentados são a média de correr o programa 3 vezes para cada um dos ficheiros de teste. O tempo apresentado é o tempo real. Os testes foram feitos numa máquina virtual com 1024MB DDR3 e um CPU core i5-2557M a 1.70GHz a correr Debian Squeeze 6.0.7.

Dado que o coeficiente de correlação foi superior a 0,98 em todos os casos, tendo sido utilizada uma recta de regressão linear, pode-se concluir que o algoritmo implementado cresce, de facto, linearmente com o número de dominós e arcos, tal como esperado.

Referências Bibliográficas

[1] [http://en.wikipedia.org/wiki/Condensation_\(graph_theory\)](http://en.wikipedia.org/wiki/Condensation_(graph_theory))

[2] http://en.wikipedia.org/wiki/Tarjan's_strongly_connected_components_algorithm