João Miguel Louro Neto - 64787
Instituto Superior Técnico - Análise e Síntese de Algoritmos - Projecto 2
10 de Maio de 2013

EMPREGOS

Descrição do Problema

Dado um conjunto de estudantes, um conjunto de empregos, e um conjunto de candidaturas de estudantes a empregos (cada estudante pode candidatar-se a mais do que um emprego, e cada emprego pode ter mais do que um estudante a candidatar-se), indicar qual o número máximo de estudantes com emprego atribuído, considerando que qualquer estudante não fica com mais de um emprego atribuído, e qualquer emprego não poderá ser atribuído a mais de um estudante.

Abordagem ao Problema

Comecei por tentar uma abordagem **Greedy** ao problema - a qual falhou por completo. Em seguida, tentei formular uma solução para o problema através de **programação dinâmica**, também sem sucesso. Não fiquei convencido de inexistência de solução para o problema através de uma formulação de tal tipo, mas não consegui encontrá-la. Formular o problema através de um **problema de satisfação de restrições [4]** também era trivial, mas a procura apenas era relativamente eficiente para atribuições completas. A resolução do problema através de **programação linear** parecia trivial - o cálculo eficiente da solução nem tanto.

Foi então que me apercebi que este problema poderia ser resolvido como um problema de **fluxos**, sendo estudantes e empregos vértices, estando os estudantes ligados aos empregos aos quais se candidataram por um arco, criando ainda dois vértices adicionais - um vértice fonte S, com um arco para todos os estudantes, e um vértice destino T, para o qual todos os empregos têm um arco. Todos os arcos com custo unitário. A solução passaria então por achar o fluxo máximo (em que fluxos são inteiros) entre S e T. O valor do fluxo corresponderia ao número máximo de estudantes com emprego atribuído. A correcção desta formulação deve-se ao facto de cada estudante apenas poder enviar fluxo para um único trabalho, e cada trabalho apenas poder receber fluxo de um estudante, respeitando assim as restrições impostas na descrição do problema.

Com isto, implementei o algoritmo de **Relabel-to-Front** (em C++). Esta provou-se ineficiente - para grafos esparsos demorava demasiado tempo a completar. Pensei em implementar o algoritmo de **Edmonds-Karp**, mas não o fiz.

Entretanto ocorreu-me que este grafo era **bipartido** - existem métodos mais eficientes para calcular o **emparelhamento máximo[3]** em grafos deste tipo, nomeadamente o **algoritmo de Hopcroft-Karp[1]**, que tem a melhor complexidade no pior-caso para **grafos densos**. A implementação em C é bastante simples, dado que apenas necessitei de um **FIFO**, e provou estar bastante optimizada, pelo que não teve qualquer dificuldade em passar aos testes.

Solução

A solução final passou pela construção de um grafo não dirigido, bipartido através do input recebido. Após obter o **número de estudantes (S)** e o **número de empregos (J)**, criavam-se **S+J+1 vértices** (sendo este último o **vértice auxiliar NIL** para o **algoritmo de Hopcroft-Karp**). Para cada candidatura A[i,j] de cada estudante S[i], criava-se um **arco não dirigido** de S[i] para J[A[i,j]] - arco de S[i] para J[A[i,j]], e arco de J[A[i,j]] para S[i].

Após construção do grafo, este era passado directamente ao **algoritmo de Hopcroft-Karp**. Deste, o output que nos interessava era apenas o **número de emparelhamentos** após execução do algoritmo.

Apesar de experimentalmente estar provado que o algoritmo não é tão rápido na prática como é em teoria**[2]**, este continua a ter o melhor desempenho conhecido para o pior caso (grafo denso). Para grafos esparsos, este corre em tempo quase linear**[2]**. Também se prova experimentalmente que na prática é mais lento que técnicas push-relabel, e técnicas simples largura-primeiro e profundidade-primeiro**[2]**.

Este algoritmo tem uma vantagem - a facilidade de implementação. Apenas necessita de um tipo abstracto de dados FIFO, e o resto do algoritmo pode ser implementado em 70 linhas de código ou menos, traduzindo directamente do pseudocódigo.

Implementação

A solução foi implementada em ANSI C com relativa facilidade (excepto problemas na depuração de problemas associados ao algoritmo de Hopcroft-Karp).

O tipo básico de dados utilizado '**uint**' foi definido como sendo '**long unsigned int**'. Segue-se uma breve listagem das estruturas de dados relevantes utilizadas - o que representam e que operações são possíveis sobre as mesmas.

Para efeitos de cálculo da complexidade computacional, assume-se que **operações** de **reserva e libertação de memória** decorrem em **tempo constante**.

- Fila (FIFO) - A utilização de uma fila serve para suportar **todas as operações** relacionadas em manter uma fila durante a execução do algoritmo de Hopcroft-Karp em **tempo constante**.

Estrutura **fifo_item_t(int value, fifo_item next)** - representa um ítem na fila. O tipo "fifo_item" é um ponteiro para uma estrutura do tipo "fifo_item_t".

- Função **fifo_item_create(uint value): fifo_item** - aloca memória e inicializa uma estrutura do tipo fifo_item com o valor fornecido - **O(1)**
- Função **fifo_item_destroy(fifo_item item): uint** - destrói e liberta a memória utilizada pelo item, e devolve o valor do ítem destruído - **O(1)**

Estrutura **fifo_t(uint size, fifo_item first, fifo_item last)** - representa uma estrutura do tipo fila. 'first' é um ponteiro para o ítem à frente da pilha, 'last' é um ponteiro para o

ítem no fim da fila, e 'size' indica a quantidade de elementos presentes na fila. O tipo "fifo" é um ponteiro para uma estrutura do tipo "fifo_t".

- Função **fifo_create(): stack** - Aloca espaço e inicializa uma pilha sem ítens - **O(1)**
- Função **fifo_destroy(fifo f): void** - Destrói a pilha e todos os seus N elementos, libertando a memória utilizada por estes - **O(N)**
- Função **fifo_queue(fifo f, uint valor): void** - Insere 'valor' no fim da fila. Utiliza a função fifo_item_create(uint value) de forma a alocar espaço e inicializar uma nova estrutura fifo_item - **O(1)**
- Função **fifo_dequeue(): uint** - Elimina o ítem à frente da fila e devolve o seu valor. Utiliza a função fifo_item_destroy(fifo_item item) para libertar o espaço utilizado pelo fifo_item e devolver o seu valor. - **O(1)**

- Caso de Teste (Testcase)

Estrutura **testcase_t(uint num_students, uint num_jobs, uint num_vertices, fifo[] adjacencies, uint[] pair_g1, uint[] pair_g2, uint[] dist, uint matching)** - representa um caso de teste, a ser alimentado ao algoritmo de Hopcroft-Karp. 'num_students' reprensenta o número de estudantes, 'num_jobs' representa o número de empregos e 'num_vertices' representa o número de vértices no grafo (num_students + num_jobs + vértice NIL). 'adjacencies' é um vector de filas - a fila no índice i é uma lista dos vértices adjacentes ao vértice de índice i. pair_g1, pair_g2 e dist são vectores. as posições i em pair_g1 e pair_g2 representam os vértices emparelhados com o vértice i (no caso do valor ser NIL, indica que não está emparelhado). 'matching' indica o número de emparelhamentos. O tipo "testcase" é um ponteiro para uma estrutura do tipo "testcase_t".

- Função **testcase_create(uint num_students, uint num_jobs): testcase** - Aloca espaço e inicializa uma estrutura testcase capaz de armazenar informação de todos os S estudantes e J empregos - isto inclui o espaço necessário para os vectores adjacencies (mais inicialização de todas as filas), pair_g1, pair_g2 e dist. As listas de adjacências são inicializadas estando vazias. - **O(S+J)**
- Função **testcase_destroy(testcase t): void** - Destrói a estrutura testcase, libertando o espaço ocupado por esta e todas as filas em 'adjacencies' - **O(S*J)**
- Função **testcase_add_application(fifo f, uint student, uint job): void** - Insere um arco não dirigido ao caso de teste - um arco de student até job, e outro arco de job até student. Os arcos são inseridos no fim de cada uma das filas de adjacências de cada um dos vértices. - **O(1)**

Análise da Complexidade da Solução Implementada

Seja S o número de estudantes, J o número de empregos, e A o número de candidaturas (representadas como arcos entre vértices S e J). A é O(S*J) para grafos densos. Em termos de representação em grafo, seja V o número de vértices (O(S+J)) e E o número de arcos (O(A)).

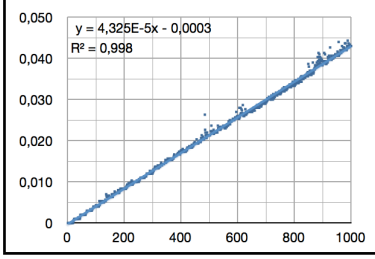
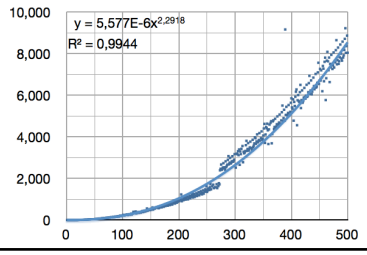
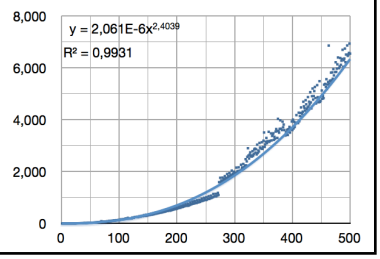
A leitura e inicialização de um caso de teste tem complexidade O(S+E). Depois da inicialização do caso de teste, este é fornecido directamente ao algoritmo de Hopcroft-

Karp. Este corre em $O(E\sqrt{(V)})$. Após cálculo do resultado, liberta-se toda a memória utilizada pelo caso de teste em $O(A)$.

Como tal, pode concluir-se que o programa corre em $O(S+E\sqrt{(V)})$. No caso de um grafo denso tem-se $E=V^2$, pelo que no pior caso o programa apresenta uma complexidade de $O(V^{2.5})$.

Avaliação experimental

Para confirmar a complexidade da solução, indicada na secção anterior, foram criados três tipos de testes, aos quais se irá variar o tamanho do input. N representa o número de estudantes, e também o número de empregos. O eixo vertical representa o tempo em segundos, o eixo horizontal representa o valor de N (em milhares).

Teste 1	Teste 2	Teste 3
Estudante i candidata-se ao emprego i	Todos os estudantes se candidatam a todos os empregos (pior caso)	Cada estudante tem um número aleatório de candidaturas, todas elas a empregos aleatórios.
		

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 8GB DDR3 e 2 virtual cores de um CPU i7-3770T a 2.40GHz a correr Debian Squeeze 6.0.7.

Dado que o coeficiente de correlação foi superior a 99% em todos os casos, tendo sido utilizada uma recta de regressão linear, pode-se concluir que o algoritmo implementado cresce, de facto, como esperado (excepto no teste 1, no qual o algoritmo parece crescer linearmente por ser um grafo esparso - regressão em potência $y=3,179E\text{-}5x^{1,0469}$, com $R^2=0,9953$).

Referências Bibliográficas

[1] J. Hopcroft and R. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. SIAM Journal of Computing, 2:225-231, 1973
[2] http://en.wikipedia.org/wiki/Hopcroft-Karp_algorithm
[3] http://en.wikipedia.org/wiki/Matching_(graph_theory)
[4] http://en.wikipedia.org/wiki/Constraint_satisfaction_problem