Advanced Algorithms and Datastructures

Exam Notes

Author:
Jenny-Margrethe Vej – 250986 – rwj935

Block 4, April-June, 2014

Contents

1 Maximum Flow

Wuuuh, Max Flow!! You Rock! Kick some ass!

1.1 Flow Networks

1.1.1 Flow Networks and Flow

Let $G = (V, E)$ be a flow network with a capacity function c. Let s be the source of the network, and let t be the sink. A flow in G is a real-valued function $f : V \times V \to \mathbb{R}$ that satisfies the following two properties:

Capacity Constraints: For all $u, v \in V$, we require $0 \leq f(u, v) \leq c(u, v)$
Flow Conservation: For all $u \in V - \{s, t\}$, we require

$$\sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v) \tag{1}$$

When $(u, v) \notin E$, there can be no flow from u to v, and $f(u, v) = 0$. The value $|f|$ of a flow f is defined as

$$|f| = \sum_{v \in V} f(s, u) - \sum_{v \in V} f(v, s) \tag{2}$$

1.1.2 An example of a flow

For example the transporting example from the book. A firm wants to transport goods from 1 place to anther, using a third part driver.

Guessing this particular section is not thaaaaat important for the exam. ;o)

1.1.3 Modelling problems with antiparallel edges

Antiparallel edges is 2 edges going to/from (v, u) - so 2 edges between 2 vertices but with opposite directions. To come around that, we transform our network into an equivalent one containing no antiparallel edges (adding an extra vertex for that, so we can split one of the edges). The resulting network is equivalent to the original one, due to the fact, that you do not add or subtract anything from the

capacity. It is the same:

1.1.4 Networks with multiple sources and sinks

A maximum flow problem may have several sources and sinks, rather than just one of each. To fix that, we just add a supersource and a supersink with infinity capacity from s to each of the multiple sources.

1.2 The Ford-Fulkerson Method

1: FORD-FULKERSON-METHOD(G, s, t)
2: initialise flow f to 0
3: **while** there exists an augmenting path p in the residual network G_f **do**
4: augment flow f along p
5: **end while**
6: **return** f

1.2.1 Residual Network

Suppose that we have a flow network $G = (V, E)$ with source s and sink t. Let f be a flow in G, and consider a pair of vertices $u, v \in V$. We define the residual capacity $c_f(u, v)$ by

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & \text{if } (u,v) \in E, \\ f(v,u) & \text{if } (v,u) \in E, \\ 0 & \text{otherwise} \end{cases}$$

2 Fibonacci Heaps

3 NP-Completeness

4 Randomised Algorithms

5 Hashing

OMG LOLZ'n SHIZZLES! You'r gonna nail this shit!

5.1 Direct-address tables

We have a scenario where we want to maintain a dynamic set of some sort. Each element has a key drawn from a universe $U = \{0, 1, ..., m-1\}$ where m is not too large, and no two elements have the same key.

Representing it by a direct-address table, or array $T[0...m-1]$, each slot or position corresponds to a key in U. If there is an element x with the key k, then $T[k]$ contains a pointer to x - otherwise $T[k]$ is empty represented by NIL

Illustration in the book on page 254

1: DIRECT-ADDRESS-SEARCH(T, k)
2: **return** $T[k]$
3:
4: DIRECT-ADDRESS-INSERTT, x
5: **return** $T[key[x]] \leftarrow x$
6:
7: DIRECT-ADDRESS-DELETET, x
8: **return** $T[key[x]] \leftarrow NIL$

Downsides of direct addressing: If the universe U is large, storing a table T of size $|U|$ may be impractical or even impossible, given the memory available on a typical computer. Furthermore, the set K of keys actually stored may be so small relative to U that most of the space allocated for T would be wasted.

5.2 Hash tables

To overcome the downsides of direct addressing, we can use hash tables. When K is much smaller than U, a hash table requires much less space than a direct-address table. It can also reduce storage requirements to $\Theta(|K|)$.

This bound is for average-case time whereas for direct addressing it holds for the worst-case time.

With direct addressing, an element with k is stored in slot k. With hashing, this element is stores in slot $h(k)$; that is, we use a hash function h to compute the slot from the key k. There is one hitch: 2 keys may hash to the same slot - called a collision. See page 256 for illustrations.

A hash function h must be deterministic in that a given input k should always produce the same output $h(k)$.

Collision resolution by chaining
In chaining we place all the elements that hash to the same slot into the same linked list. See figure on page 257.

1: DIRECT-HASH-SEARCH(T, x)
2: **search** for an element with key k in list $T(T, k)$
3:
4: DIRECT-HASH-INSERTT, x
5: **insert** x at the head of the list $T[h(x.key)]$
6:
7: DIRECT-HASH-DELETET, x
8: **delete** x from the list $T[h(x.key)]$

Analysis of hashing with chaining
Given a hash table T with m slots that stores n elements, we define the load factor α for T as n/m which is the same as the average number of elements stores in a chain.

Worst case performing: $\Theta(n)$ + the time to compute the hash function (all n keys hash to the same slot, creating a list of length n.

Average case performing of hashing depends on how well the hash function h distributes the set of keys to be stores among the m slots, on the average.

Theorem 11.1 In a hash table in which collisions are resolved by chaining, an unsuccessful search takes an average-case time $\Theta(1 + \alpha)$, under the assumption of simple uniform hashing. (OBS: Beviset herfor står på side 259!! LÆS OG FORSTÅ)

Theorem 11.2 In a hash table in which collisions are resolved by chaining, a successful search takes average-case time $\Theta(1+\alpha)$, under the assumption of simple uniform hashing. (OBS: Beviset herfor står på side 259-260!! LÆS OG FORSTÅ)

5.3 Hash functions

What makes a good hash function?
A good hash function satisfies (approximately) the assumption of simple uniform hashing: each key is equally likely to hash to any of the m slots, independently of where any other key has hashed to. In practice, we can often employ heuristic techniques to create a hash function that performs well.

Interpreting keys as natural numbers
Most hash functions assume that the universe of keys is the set $\mathbb{N} = \{0, 1, 2, ...\}$ of natural numbers.

5.3.1 The division method

5.3.2 The multiplication method

5.3.3 Universal hashing

5.4 Open addressing

5.5 Perfect hashing

6 Exact Exponential Algorithms

7 Approximation Algorithms

8 Computational Geometry

9 Linear Programming and Optimisation