Randomized Algorithms
Assignment 4 - Resubmission

Nikolaj Dybdahl Rathcke (rfq695)
Victor Petren Bach Hansen (grn762)

June 8, 2016

Summary of RA 5.4-5.6

In section 5.4, we revisit the problem of oblivious routing, studied in section 4.2, but now with the aspect of the probabalistic method. The example again focuses on oblivious permutation routing on the hypercube. Here the following theorem is presented

Theorem 5.8 Consider Consider any randomized oblivious algorithm for permutation routing on the hypercube with $N = 2^n$ nodes. If this algorithm uses k random bits, then its expected running time is $\Omega(2^{-k}\sqrt{N/n})$

This yields the corollary:

Corollary 5.9 Any randomized oblivious algorithm for permutation routing on the hypercube with $N = 2^n$ nodes must use $\Omega(n)$ random bits in order to achieve expected running time $\mathcal{O}(n)$.

Which shows that the algorithm presented in 4.2 uses N bits of randomness more than necessary. By using the probabilisitc method we can match this lower bound, as stated in theorem 5.10:

Theorem 5.10 For every n, there exists a randomized oblivious scheme for permutation routing on a hypercube with $N = 2^n$ nodes that uses $3n$ random bits and runs in expected time at most $15n$

Section 5.5 then presents a tool in the probabilistic method, called the *Lovász Local Lemma*. Given n events which each occurs independently with probability atmost $1/2$, we can then assert that none of the events will occur with with probability 2^{-n}. It generalizes this notion to the case where each of these events is independent of all but a small number of other events. The lemma is as follows:

Lemma 5.11 (The Lovász Local Lemma) Let $G(V, E)$ be a dependency for events $\mathcal{E}_1, \ldots, \mathcal{E}_n$ in a probability space. Suppose that there exists $x_i \in [0, 1]$ for $1 \leq i \leq n$ such that

$$Pr[\mathcal{E}_i] \leq x_i \prod_{(i,j)\in E} (1 - x_j)$$

Then

$$iPr[\cup_{i=1}^{n} \overline{\mathcal{E}_i}] \geq \prod_{i=1}^{n} (1 - x_i)$$

This lemma is then applied to show that any instance of SAT meeting certain conditions always has a satisfying assignment.

Section 5.6 describes how the method of conditional probabilities can be used to derandomize an algorithm. The section uses the set-balancing problem as an example as how to derandomize the algorithm described in example 4.5, such that it becomes deterministic. The method of conditional probability applied here results in the theorem

Theorem 5.15 The algorithm based on the method of conditional probabilities determines a vector b such that $\|Ab\|_{infty} \leq 4\sqrt{n \ln n}$, in time polynomial in n.

Summary of RA 8.1, 8.2 and 8.5

In chapter 8.1, we are told about the fundamental data structuring problem. That is, we revisit the operations:

- MAKESET(S): Create a new and empty set S.
- INSERT(k, S): Insert item with key k in S.
- DELETE(S): Delete item with key k from S.
- FIND(S): Return item with key k from S.
- JOIN(S_1, I, S_2): Construct a new set $S = S_1 \cup \{i\}S_2$, such that $\forall j \in S_1, k(j) < k(i)$ and $\forall j \in S_2, k(j) > k(i)$.
- PASTE(S_1, S_2): Construct a new set $S = S_1S_2$, such that $\forall i \in S_1$ and $\forall j \in S_2$, then $k(i) < k(j)$.
- SPLIT(k, S): Split the set S into sets S_1 and S_2, where $S_1 = \{j \in S \mid k(j) < k\}$ and $S_2 = \{j \in S \mid k(j) > k\}$.

It's discussed how the operations are implemented for a binary search tree and introduce a method to keep the trees balanced, namely the trees that are called *treaps* (Chapter 8.2).
Treaps are binary search trees where each node has an additional attribute, called a priority, so that it is a binary search tree with respect to the keys and a heap with respect to the priorities. The treaps have an interesting property, stated as a Theorem:

Theorem 8.1 Let $S = \{(k_1, p_1), ..., (k_n, p_n)\}$ be any set of key-priority pairs such that the keys and priorities are distinct. Then, there exists a unique treap $T(S)$ for it. The chapter then discusses how to implement the different operation for this kind of data structure and what their running times are. Lemma 8.6 tells us something about the depth of some node x:

Lemma 8.6 Let T be a random treap for a set S of size n. For an element $x \in S$ having rank k

$$\mathbb{E}[depth(x)] = H_k + H_{n-k+1} + 1$$

which tells us that we can always expect the depth to be $\mathcal{O}(\log n)$. We can also say something about the number of rotations which are needed when we make an update on the treap:

Lemma 8.7 Let T be a random treap for a set S of size n. For an element $x \in S$ of rank k

$$\mathbb{E}[R_x] = 1 - \frac{1}{k}$$

and

$$\mathbb{E}[L_x] = 1 - \frac{1}{n-k+1}$$

Where R_x denotes the length of the right spine in the left subtree and L_x is the length of the left spine of the right subtree. This analysis leads us to the final Theorem, which states something about the running times for the operations:

Theorem 8.8 Let T be a random treap for a set S of size n

1. The expected time for a FIND, INSERT, or DELETE operation on T is $\mathcal{O}(\log n)$.
2. The expected number of rotations required during an INSERT or DELETE operation is at most 2.
3. The expected time for a JOIN, PASTE, or SPLIT operation involving sets S_1 and S_2 of sizes n and m, respectively, is $\mathcal{O}(\log n + \log m)$.

Chapter 8.5 discusses hashing with $\mathcal{O}(1)$ search time. The final solution they present is by using a primary hash function to put element in bins so that the bin sizes are small. Another perfect hash function associated wiith each bin then resolves the collision by using secondary hash tables. This double-hashing scheme with a primary table of size $n = s$ uses space:

$$s + \mathcal{O}\left(\sum_{i=0}^{s-1} b_i^2\right)$$

which means it uses linear space if the bin sizes are linearly bounded in s, and still manages search operations in constant time. Chapter 8.2.2 shows that we can pick primary and secondary hashing function which ensures the bin sizes are linearly bounded by s.