I/O-Efficient Algorithms 2012 — Homework

This assignment is to be made and handed in by each student separately. Your answers have to be sent by e-mail to the lecturer by Sunday 8 July, 23:59. Keep in mind that not all questions have well-defined answers. Some may even turn out, unintentionally, to be ‘trick questions’, and it is okay if you cannot solve all problems. Nevertheless, the more you can solve, the better it is for your grade. As a guideline: exercises 1, 2a, 2b, 2c, 3a, 3b, 4a, 4b and 5a are kind-of basic and you should be able to give a satisfactory answer to most of them; exercises 2d, 2e, 3c, 5b, 5c, and 5d, marked with asterisks, are meant to be more challenging. You may use results from the course material without proving them. In total, the homework assignment should take an afternoon to complete, but it might take a bit more time if you have to look up basic knowledge about internal-memory data structures.

Exercise 1.

Prove that topological sorting of a graph with \( N \) vertices and \( \Theta(N) \) edges needs at least \( \Omega(\min(\text{sort}(N), N)) \) I/Os in the worst case.

Exercise 2.

Consider the following algorithm, which takes as input a two-dimensional \( N \times N \) matrix \( I \) which contains the colours of a digital image in row-major order, the coordinates \((\text{row}, \text{col})\) of one white pixel, and a colour \( c \neq \text{white}; \) the algorithm then finds the white component that contains pixel \((\text{row}, \text{col})\) and changes the colour of this component to \( c \).

1. initialize an empty stack \( S \)
2. push \((\text{row}, \text{col})\) onto \( S \)
3. while \( S \) is not empty
   4. do pop \((\text{row}, \text{col})\) from \( S \)
   5. if \( \text{row}, \text{col} \in \{1, \ldots, N\} \) and \( I[\text{row}, \text{col}] = \text{white} \)
      6. then \( I[\text{row}, \text{col}] \leftarrow c \)
      7. push \((\text{row}, \text{col} - 1), (\text{row} - 1, \text{col}), (\text{row}, \text{col} + 1), (\text{row} + 1, \text{col})\) onto \( S \)

(a) Where is the I/O-bottleneck in this algorithm, and what would be an easy way to alleviate this bottleneck in most practical situations?

(b) How many I/O’s does your solution need in the worst case? What would be an input that would bring out this worst-case behaviour?

(c) Is your solution cache-aware or is it cache-oblivious? Explain your answer.

(d*) What technique could be used to get an algorithm that is I/O-efficient in the worst case?

(e*) Would such a solution be cache-aware or cache-oblivious? Explain your answer.
Exercise 3.

The following two algorithms each take as input a weighted graph with \( n \) vertices numbered consecutively from 1 to \( n \), given as two arrays:

- an array \( E \) that contains the edges of the graph, where each edge is given as a triple \((i, j, w)\) of two vertex numbers \( i \) and \( j \) and a weight \( w \); each edge is stored twice (once as \((i, j, w)\) and once as \((j, i, w)\)), and the whole array is sorted lexicographically;
- an array \( V \) such that \( V[i] \) is the index of the first edge \((i, j, w)\) in \( E \), and \( V[n+1] = |E|+1 \) (pointing past the end of \( E \))

Each algorithm outputs a set of edges that forms minimum spanning tree of the input graph.

**PRIM**(number of vertices \( n \), edge array \( E \), vertex array \( V \))

1. initialize an array \( Reached[1..n] \) and set all entries to false
2. initialize an empty priority queue \( Q \) to store edges in order of increasing weight
3. \( Reached[1] \leftarrow true \)
4. for edge \( \leftarrow V[1] \) to \( V[2] - 1 \) \( \triangleright \) (all edges incident on vertex 1)
5. do insert \( E[edge] \) into \( Q \)
6. while \( Q \) is not empty
7. do extract a minimum-weight edge \((i, j, w)\) from \( Q \)
8. if \( Reached[j] = false \)
9. then output \((i, j)\)
10. \( Reached[j] \leftarrow true \)
11. for edge \( \leftarrow V[j] \) to \( V[j + 1] - 1 \)
12. do insert \( E[edge] \) into \( Q \)

**KRUSKAL**(number of vertices \( n \), edge array \( E \), vertex array \( V \))

1. initialize a union-find data structure \( U \) on the vertex numbers \{1, ..., n\}, such that each number is in a set by itself
2. sort \( E \) by increasing weight \( w \)
3. for each edge \((i, j, w)\) in \( E \) in order from \( E[1] \) to \( E[|E|] \)
4. do if set containing \( i \) in \( U \) \( \neq \) set containing \( j \) in \( U \)
5. then output \((i, j)\)
6. merge the sets containing \( i \) and \( j \) in \( U \)

(a) What are the I/O bottlenecks in PRIM, if implemented with the most I/O-efficient priority queue you know?

(b) What are the I/O bottlenecks in KRUSKAL, if implemented with the most I/O-efficient sorting algorithm and union-find data structure you know?

(c*) Do you have any ideas about how to alleviate the bottlenecks in the above algorithms in at least some cases in practice?
Exercise 4.

Consider the following two data structures to store $2^{30}$ records whose keys are distinct 64-bit numbers:

- a $B$-tree, in which each internal node has roughly $2^{10}$ children and each leaf stores $2^{10}$ records;
- a hash table with $\frac{3}{2} \cdot 2^{30}$ slots, using double hashing\(^1\).

Consider a situation in which one of these data structures is used to answer queries of the form: give me the record with key $k$. The queries arrive and need to be answered one by one in random order. You may assume that all queries are successful: there is never a query for a record with key $k$ if there is no such record in the data structure. You may assume that some algorithm has been using the data structure intensively, so that many blocks of it may already be in cache.

(a) Which of the two data structures would have the best expected query time for a sequence of uniformly distributed random successful queries when the main memory fits $2^{32}$ records? Explain your answer.

(b) Which of the two data structures would have the best expected query time for a sequence of uniformly distributed random successful queries when the main memory fits only $2^{28}$ records? Explain your answer.

---

\(^1\)To store a record with key $x$, a hash function $h(x)$ is used that computes in which slot of the table the record should be stored; if that slot is already occupied, a second hash function $h'$ is used to try other slots $h(x) + i \cdot h'(x)$ for $i$ starting at zero and increasing one by one, until an empty slot is found.
Exercise 5.

In the lectures and in the course material we have seen several depth-first-search algorithms:

- standard linear-time I/O-naive DFS;
- the phase approach (see Meyer, Section 4.2);
- a solution with buffered repository trees and priority queues (see Meyer, Section 4.6).

Below are a number of questions about the running times of these algorithms. The questions are really about orders of magnitude, so you may ignore insignificant terms in your calculations; it is okay if the end result is off by a factor five.

(a) If we ignore all the hidden constants in the bounds and have $M = 2^{30}$, $B = 2^{14}$, $|V| = 2^{31}$, $|E| = 2^{32}$, and one I/O takes 10 ms, what would be the running times of the three algorithms?

(b*) For what values of $M$, $B$, $|V|$ and $|E|$ would the first algorithm be faster than the other two algorithms and done within a week? Hint: consider how small/large each of $M$, $B$, $|V|$ and $|E|$ should be if we keep the other values as specified above. Include your calculations in your answer.

(c*) For what values of $M$, $B$, $|V|$ and $|E|$ would the second algorithm be faster than the other two algorithms and done within a week? Hint: consider how small/large each of $M$, $B$, $|V|$ and $|E|$ should be if we keep the other values as specified above. Include your calculations in your answer.

(d*) For what values of $M$, $B$, $|V|$ and $|E|$ would the third algorithm be faster than the other two algorithms and done within a week? Hint: consider how small/large each of $M$, $B$, $|V|$ and $|E|$ should be if we keep the other values as specified above. Include your calculations in your answer.