CR15: notes for the lecture of January 4, 2023
Part 1: Introduction to Scheduling Under Memory Constraints and Pebble Game Models

Loris Marchal
(Notes revised by Frédéric Vivien)

January 4, 2023

1 Algorithm Design and Data Movement

Let us consider here a very simple problem, the matrix product, to show how the design of the algorithm can influence the memory usage and the amount of data movement[1]. We consider two square $n \times n$ matrices A and B, and we compute their product $C = AB$.

Algorithm 1: SIMPLE-MATRIX-MULTIPLY(n, C, A, B)

```
for i = 0 → n − 1 do
    for j = 0 → n − 1 do
        C_{i,j} = 0
        for k = 0 → n − 1 do
            C_{i,j} = C_{i,j} + A_{i,k} B_{k,j}
```

We consider that this algorithm is executed on a simple computer, consisting of a processing unit with a fast memory of size M. In addition to this limited memory, a large but slow storage space is available. In the following, we assume that this space is the disk and has unlimited available storage space.[2] Our objective is to minimize the data movement between memory and disk, also known as the volume of I/O (input/output), that is the number of A, B and C elements that are loaded from the disk to the memory, or written back from the memory to the disk. We assume that the memory is limited, and cannot store more than half a matrix, i.e., $M < n^2/2$.

In Algorithm 1, all elements of B are accessed during one iteration of the outer loop. Thus, as the memory cannot hold more than one half of B, at least $n^2/2$ elements must be read. For the n iterations, this leads to $n^3/2$ read operations. This is huge as it is the same order of magnitude as the number of computations (n^3).

Fortunately, it is possible to improve the I/O behavior of the matrix product by changing the algorithm. We set $b = \sqrt{M/3}$ and assume that n is a multiple of b. We consider the blocked version of the matrix product, with block size b, as detailed in Algorithm 2. In this algorithm $C^b_{i,j}$ denotes the block of size b at position (i, j) (all elements $C_{k,l}$ such that $ib \leq k \leq (i+1)b - 1$ and $jb \leq l \leq (j+1)b - 1$).

Each iteration of the inner loop of the blocked algorithm must access 3 blocks of size b^2. Thanks to the choice of b, this fits in the memory, and thus, each of these $3b^2$ elements are read at most once and the b^2 elements of C are written each exactly once. This leads to at most $4/3M$ data

[1] A large part of this section is adapted from [9].

[2] Note that this study detailed for the case "main memory vs. disk" may well apply to other pairs of storage such as "fast small cache vs. large slower memory".

Algorithm 2: BLOCKED-MATRIX-MULTIPLY(n, C, A, B)

```
b ← √(M/3)
for i = 0, → n/b − 1 do
    for j = 0, → n/b − 1 do
        for k = 0, → n/b − 1 do
            Simple-Matrix-Multiply(n, C^b_{i,j}, A^b_{i,k}, B^b_{k,j})
```

movements. Since there are $(n/b)^3$ iterations of the inner loop, the volume of I/O of the blocked algorithm is $O((n/b)^3 \times 4/3M) = O(n^3/\sqrt{M})$.

2 (Black) pebble game for memory minimization

2.1 Definition

We present here the first theoretical model that was proposed to study the space complexity of programs. This model, based on a pebble game, was originally used to study register allocation. Registers are the first level of storage, the fastest one, but also a scarce resource. When allocating registers to instructions, it is thus crucial to use them with caution and not to waste them. The objective is thus to find the minimum amount of registers that is necessary for the correct execution of the program. Minimizing the number of registers is similar to minimizing the amount of memory. For the sake of consistency, we present the pebble game as a problem of memory size rather than register number minimization. This does not change the proposed model nor the results, but allows us to present all results of this document with the same model and formalism.

The pebble-game was introduced by Sethi [6] to study the space complexity of "**straight-line**" programs, that is, programs whose control flow does not depend on the input data. A straight-line program is modeled as a directed acyclic graph (DAG): a vertex represents an instruction, and an arc between two vertices $i \to j$ means that the results of the vertex i is used for the computation of j.

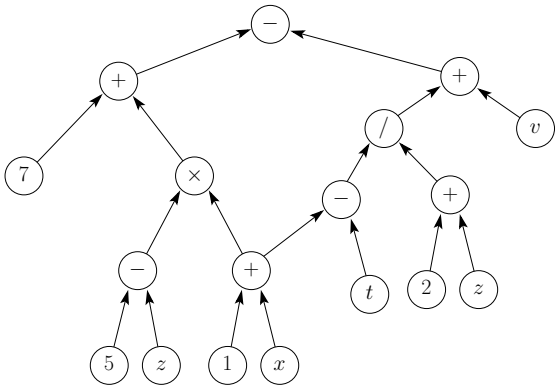

Figure 1: Graph corresponding to the computation of the expression $7 + (5 - z) \times (1 + x) - ((1 + x - t)/(2 + z) + v)$

When processing a vertex, all its inputs (as well as the result) must be loaded in memory, and the goal is to execute the program using the smallest amount of memory. Memory slots are modeled as pebbles, and executing the program is equivalent to playing a game on the graph with the following rules:

- (PG1) A pebble may be removed from a vertex at any time.

- (PG2) A pebble may be placed on a source node at any time.
- (PG3) If all predecessors of an unpebbled vertex v are pebbled, a pebble may be placed on v.

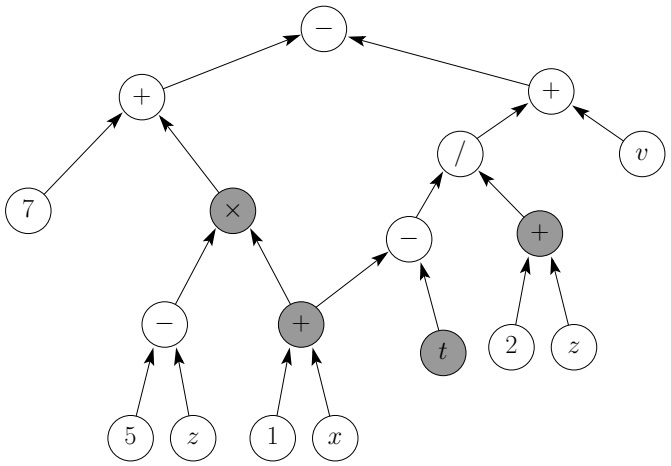

Figure 2: Playing the (black) pebble game on the graph of Figure 1. Dark nodes are the ones currently pebbled, meaning that four values are now in memory: $(5 - z) \times (1 + x)$, $1 + x$, t and $2 + z$.

The goal of the game is to put a pebble on each output vertex at some point of the computation, and to minimize the total number of pebbles needed to reach this goal. In this game, pebbling a node corresponds to loading an input in memory (rule PG2) or computing a particular vertex (rule PG3). From a winning strategy of this game, it is thus straightforward to build a solution to the original memory allocation problem.

2.2 Variants and hardness

Note that the game does not ensure that each vertex will be pebbled only once. Actually, in some specific graphs, it may be beneficial to pebble several times a vertex, that is, to compute several times the same values, to save a pebble needed to store its value. A variation of the game, named the *Progressive Pebble Game*, forbids any recomputation, and thus models the objective of minimizing the amount of memory without any increase in the computational complexity. In this latter model, the problem of determining whether a directed acyclic graph can be processed with a given number of pebbles has been shown NP-hard by Sethi [6]. The more general problem allowing recomputation is even more difficult, as it has later been proved PSPACE-complete by Gilbert, Lengauer and Tarjan [2]. Another variant of the game slightly changes rule PG3 and allows to *shift* a pebble to an unpebbled vertex if all its predecessors are pebbled. Van Emde Boas and van Leeuwen [10] shows that it can at most decrease the number of pebbles required to pebble the graph by one, but in the worst case the saving is obtained at the price of squaring the number of moves needed in the game.

Another variant, called the black-white pebble game has been proposed to model non deterministic execution [3, 4], where putting a white pebble on a node corresponds to guessing its value; a guessed vertex has to be actually computed later to check the value of the guess.

2.3 Special case: tree-shaped graphs

A simpler class of programs consists in trees rather than general graphs: for example, arithmetic expressions are usually described by in-trees (rooted directed trees with all edges oriented towards the root), unlike the one of Figure 1 which uses a common sub-expression twice.

2.3.1 Complete binary trees

We first consider **Complete binary trees** and establish the following result for the variant when shifting pebbles is not allowed:

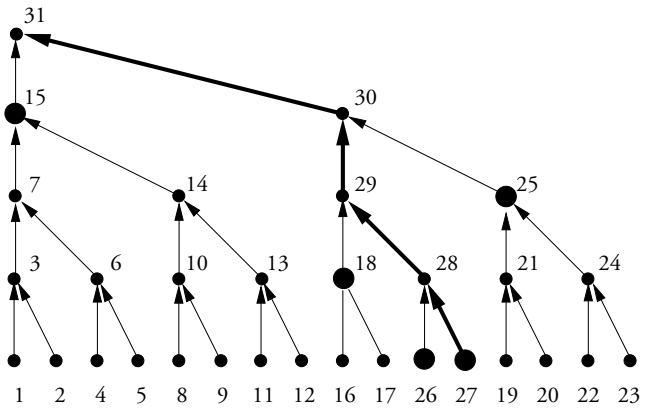

Figure 3: Complete binary tree of depth $k = 4$ and pebbles when the last leaf (27) is pebbled. This figure as well as the followings used to illustrate the pebble games are borrowed from [5].

Theorem 1. *Any pebbling strategy (with or without recomputation) for the complete balanced binary tree $T(k)$ of depth k (for $k/geq1$) uses at least $k+2$ pebbles and $2^{k+1}-1$ steps. There is a pebbling strategy that reaches both bounds.*

Proof. For the first lower bound, for a given strategy, consider the last time when it pebbles a leaf, called v. We assume that no vertex on the path P from v to the root is already pebbled (otherwise pebbling v at this time is unnecessary and can be avoided). Since v is the last pebbled leaf, for any other leaf v', there should be a pebble on the path from v' to some node on the path P from v to the root. Minimizing this number is obtained by pebbling all the descendants not in P of nodes of P, which needs k pebbles. Together with the pebble on v, it makes $k+1$ pebbles. When the parent of v is pebbled, $k+2$ pebbles are required

The second lower bound simply derives from the fact that each vertex of the tree should be pebble once.

Finally, the following postorder pebbling strategy reaches both bounds:

1. Pebble the left subtree, leave a pebble on its root
2. Pebble the right subtree, leave a pebble on its root
3. Pebble the root, remove the pebbles at the roots of the two subtrees.

A tree of depth 0 (single node) can be pebbled with one pebble. Hence, a tree of depth k can be pebbled with $k+2$ pebbles using this strategy. Furthermore, it only pebbles once each vertex. □

2.3.2 General trees

We continue to focus on the variant without shifting pebbles. Even in this case, the problem gets much simpler than for general graph: Sethi and Ullman [7] designed an optimal scheme which relies on the following theorem.[3]

[3]When pebble shifting is allowed, the minimum number of pebbles needed to pebble a tree is the Strahler number (as outlined in [1]), which is a measure of a tree's branching complexity that appears in natural science, such as in the analysis of streams in a hydrographical bassin [8] or in biological trees such as animal respiratory and circulatory systems https://en.wikipedia.org/wiki/Strahler_number.

Theorem 2. *An optimal solution for the problem of pebbling an in-tree with the minimum number of pebbles using rules PG1 – PG3 is obtained by a depth-first traversal which orders subtrees by non-increasing values of $P(i)$, where the peak $P(v)$ of the subtree rooted at v is recursively defined by:*

$$P(v) = \begin{cases} 1 & \text{if } v \text{ is a leaf} \\ \max(k+1, \max_{i=1\ldots k} P(c_i) + i - 1) & \text{where } c_1, \ldots c_k \text{ are the children of } v \\ & \text{such that } P(c_1) \geq P(c_2) \geq \cdots \geq P(c_k) \end{cases}$$

The first step to prove this result is to show that depth-first traversals are dominant, i.e., that there exists an optimal pebbling scheme which follows a depth-first traversal. Pebbling a tree using a depth-first traversal adds a constraint on the way a tree is pebbled: consider a tree T and any vertex v whose children are $c_1, \ldots c_k$, and assume w.l.o.g. that the first pebble that is put in the subtree rooted at v is in the subtree rooted at its leftmost child c_1. Then, in a depth-first traversal, the following pebbles must be put in the subtree rooted at c_1, until c_1 itself holds a pebble (and all pebbles underneath may be removed). Then we are allowed to start pebbling other subtrees rooted at $c_2, \ldots, c_k$. These traversals are also called *postorder*, as the root of a subtree is pebbled right after its last child.

To prove that depth-first traversals are dominant, we notice that whenever some pebbles have been put on a subtree rooted at some vertex v, it is always beneficial to completely pebble this subtree (until its root v, which uses a single vertex) before starting pebbling other subtrees.

The second step to prove Theorem 2 is to justify the order in which the subtrees of a given vertex must be processed to give a minimal number of pebbles. This result follows from the observation that after having pebbled $i-1$ subtrees, $i-1$ pebbles should be kept on their respective roots while the i^{th} subtree is being pebbled. When the root is pebbled, $k+1$ pebbles are needed, which is why it also appear in the maximum. The ordering of the children is established by a simple interchange argument.

2.4 Space-Time Tradeoffs

2.4.1 Example with the FFT graph

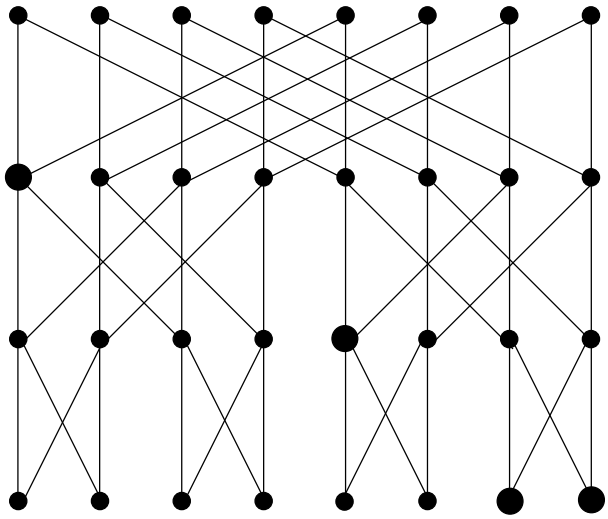

Figure 4: FFT graph with 8 input/output vertices (depth $k = 3$)

The FFT graph (see Figure 4) contains many complete binary trees; hence, it inherits some of the previous results: we need at least $k+2$ pebbles to pebble the FFT graph of depth k. A

pebbling scheme reaching this number of pebbles is obtained by pebbling each output one after the other. A simple optimization is possible: when two vertices of the third level are pebbled, two outputs (on the fourth level) can be pebbled at once. However, even with the optimization, this results in a large number of re-computations: all level-3 and 4 vertices are pebbled once, each level-2 vertex is pebbled twice, and each input (level-1) vertices are pebbled four times. Using a larger number of pebbles allows to reduce the number of steps. For example, with $2n = 2^{k+1}$ pebbles, each level can be pebble one after the other, which allows to reach the minimum number of steps.

Material used for this course and further documentation

Refer to chapter 10 ("Space-Time Tradeoffs") of the book "Models of Computation" by J. Savage [5] for more information on the black pebble game.

References

[1] Philippe Flajolet, Jean-Claude Raoult, and Jean Vuillemin. The number of registers required for evaluating arithmetic expressions. *Theoretical Computer Science*, 9(1):99–125, 1979.

[2] John R. Gilbert, Thomas Lengauer, and Robert Endre Tarjan. The pebbling problem is complete in polynomial space. *SIAM J. Comput.*, 9(3):513–524, 1980.

[3] Thomas Lengauer. Black-white pebbles and graph separation. *Acta Informatica*, 16(4):465–475, 1981.

[4] Friedhelm Meyer auf der Heide. A comparison of two variations of a pebble game on graphs. *Theoretical Computer Science*, 13(3):315–322, 1981.

[5] John E. Savage. *Models of Computation: Exploring the Power of Computing*. Addison-Wesley, 1998.

[6] Ravi Sethi. Complete register allocation problems. In *Proceedings of the 5th Annual ACM Symposium on Theory of Computing (STOC'73)*, pages 182–195, New York, NY, USA, 1973. ACM Press.

[7] Ravi Sethi and J.D. Ullman. The generation of optimal code for arithmetic expressions. *J. ACM*, 17(4):715–728, 1970.

[8] Arthur N Strahler. Hypsometric (area-altitude) analysis of erosional topography. *Geological Society of America Bulletin*, 63(11):1117–1142, 1952.

[9] Sivan Toledo. A survey of out-of-core algorithms in numerical linear algebra. In *External Memory Algorithms and Visualization*, pages 161–180. American Mathematical Society Press, 1999.

[10] Peter van Emde Boas and Jan van Leeuwen. Move rules and trade-offs in the pebble game. In *Theoretical Computer Science 4th GI Conference*, pages 101–112. Springer, 1979.