INF431

Randomized Algorithms
CORRIGÉ
Version: 2520:2533M

1 Matrix Product Verification

In this part of the PC, we will develop a very efficient way to check if the product AB of two matrices A and B equals a third matrix C. Let us assume for simplicity that all matrices given are $n \times n$ matrices. Of course, we can simply compute the product AB and compare it element-wise with C. However, there is a more efficient way. Using randomness (and allowing a constant failure probability), we can do this check in time $O(n^2)$. We discuss later how to cope with the failure probability.

Question 1 Give a simple randomized algorithm that takes as input three $n \times n$ matrices A, B, and C. The output of the algorithm is a Boolean variable b satisfying the following.

– If $AB = C$, then b is true with probability one.
– If $AB \neq C$, then b is false with probability at least $1/2$.

Your algorithm shall have a time complexity of $O(n^2)$. ◇

Solution. Since we aim at a runtime of $O(n^2)$, we cannot simply multiply the two matrices A and B and compare the result element-wise with C. What we can do in quadratic time is a matrix-vector multiplication. Hence an idea is to compute $A(Br)$ and Cr for a suitable random vector r and compare the two results. These are three matrix-vector multiplications and one comparison of two vectors, so it is clear that this can be done in quadratic time. Clearly, if $AB = C$, then $A(Br) = Cr$. The hope is that if $AB \neq C$, then with sufficiently high probability, we also have $A(Br) \neq Cr$, so we see that $AB \neq C$. To summarize, our algorithm outputs true if and only if $A(Br) = Cr$ and we hope that this is a good indication for $AB = C$. The pseudocode of this algorithm, invented by Freivalds [1], is given below.

procédure FREIVALDS(A, B, C)
Choose $r \in \{0,1\}^n$ uniformly at random;
$x := B \cdot r$;
$y := A \cdot x$;
$z := C \cdot r$;
si $y = z$
alors renvoyer (**vrai**)
sinon renvoyer (**faux**);

For this plan to work out, it suffices to take $r \in \{0,1\}^n$ uniformly at random (note that this is equivalent to saying that each r_i is chosen independently and uniformly from $\{0,1\}$). As discussed above, if $AB = C$, then $A(Br) = Cr$, to the output is correct with probability one. Hence let $AB \neq C$. Then $D := AB - C$ is not the all-zero matrix. We need to show that with probability at least $1/2$, $Dr \neq 0$. Let i, j be such that $d_{ij} \neq 0$. To ease notation, let us assume that $j = 1$. Let $q_i = (Dr)_i = \sum_{k=1}^{n} d_{ik} r_k$. We have $q_i = 0$ if and only if $r_1 = -(\sum_{k=2}^{n} d_{ik} r_k)/d_{i1}$. For any outcome of $r_2, \ldots, r_n$, this happens with probability at most $1/2$, simply because at most one of the two possible values 0 and 1 for r_1 can fulfill this equation. Consequently, with probability at least a half, we have $q_i \neq 0$, which implies $A(Br) \neq Cr$.

Another way of obtaining this result is to rely on the result by Schwartz and Zippel from the course : each coordinate of Av is a polynomial of degree $d = 1$ in the coordinates of v, so that sampling over a set S^n with $|S| = 2$ gives a probability at least $1/2$ of detecting a nonzero row. □

A failure probability of $1/2$ is not what makes the user of an algorithm happy. However, if we run the algorithm multiple times, then each run has an independent probability of failing. Use this to improve the failure probability of the previous algorithm !

Question 2 Let $\varepsilon > 0$. Give an algorithm for the matrix product verification problem that has a failure probability of at most ε. How does its runtime depend on ε ? ◇

Solution. Let $T = \lceil \log_2(1/\varepsilon) \rceil$. We run Freivalds' algorithm T times. If one of these runs detects that $AB \neq C$, we return false, otherwise we return true. If $AB = C$, all T runs return true, so we also return true, which is correct. If $AB \neq C$, each run has a probability of at most $1/2$ of not detecting the inequality. Hence the probability that no run detects it, and that we wrongfully return true, is at most $(1/2)^T \leq \varepsilon$. Otherwise, we return false as desired. □

The *probability amplification* technique used above, of course, can be applied to any randomized algorithm with one-sided error. Similar approaches building on re-running the basic algorithm exist also for two-sided errors.

2 Maximum Cuts in Graphs

Let $G = (V, E)$ be an undirected graph (graphe non orienté). For two disjoint subsets $S, T \subseteq V$ of the vertices (nœuds, sommets), let $E(S) := \{e \in E \mid e \subseteq S\}$ denote the set of edges (arêtes) fully contained in S and $E(S,T) := \{e \in E \mid |e \cap S| = 1 = |e \cap T|\}$ the set of edges having exactly one vertex each in S and T. We call $E(S, V \setminus S)$ the *cut induced by* S and denote its *size* by $\delta(S) := |E(S, V \setminus S)|$. Let $S^* \subseteq V$ with $\delta(S^*)$ maximal. For simplicity, we shall assume in the following that G has no isolated vertices (vertices with no incident edges).

In this part of the PC, we are interested in computing large cuts (sets S with $\delta(S)$ large). This problem, basic enough to be regarded in its own right, found applications among others in statistical physics and VLSI design. Unlike computing minimum cuts, this is a difficult problem. Assuming $P \neq NP$, this problem cannot be solved in polynomial time. Also, it is not possible to compute a solution S with $\delta(S) \geq (16/17)\delta(S^*)$ in polynomial time. If the unique games conjecture (see Khot [2]) is true, the constant of $(16/17) \approx 0.941$ reduces to approximately 0.878.

In the following questions, we will develop deterministic and randomized algorithms that compute a cut S with $\delta(S) \geq |E|/2$. Note that, trivially, $\delta(S^*) \leq |E|$, so all these algorithms compute a solution with $\delta(S) \geq 0.5\,\delta(S^*)$. This is the best known approximation ratio that can be achieved with classic methods. In 1995, Goemans and Williamson [3] invented an ingenious randomized algorithm (using semidefinite programming and randomized rounding) that computes cuts with $\delta(S) \geq 0.878\,\delta(S^*)$, but this is beyond the scope of this course.

We start with a not-so-easy greedy solution for the Maximum Cut problem. If you cannot solve it, or you cannot solve it completely, proceed to the next question, which can be solved independently from this one. The main message of this question is (i) that you can solve the Maximum Cut problem with the methods learned so far, but (ii) that this is not totally easy.

Question 3 Find a greedy (glouton) algorithm for the Maximum Cut problem and describe it using pseudo-code. What is its time complexity ? Show that it produces a cut S with $\delta(S) \geq |E|/2$. ◇

Solution. We start with a pair (S, T) of empty sets and then for each node $v \in V$, put v into that side that creates more cut edges $E(S, T)$.

procédure GREEDYMAXCUT($G = (V, E)$)
$S := \emptyset; T := \emptyset; X := V;$
tant que $X \neq \emptyset$
faire
- Choose $v \in X$;
- **si** $|E(\{v\}, S)| \geq |E(\{v\}, T)|$
- **alors** $T := T \cup \{v\}$
- **sinon** $S := S \cup \{v\}$;
- $X := X \setminus \{v\}$

renvoyer S

When implemented using the right data-structures (adjacency lists for the graph and arrays for sets of vertices (so that adding vertices and querying $v \in S$ can be done in constant time)), this algorithm has a time complexity of $O(\sum_{v \in V} \deg(v)) = O(|E|)$. We note that the while-loop satisfies the invariant

$$V = S \uplus T \uplus X \wedge |E(S,T)| \geq |E(S \cup T)|/2.$$

Consequently, when the while-loop is exited, we have $X = \emptyset$, hence $S \cup T = V$, and thus $|E(S,T)| \geq |E|/2$. □

The next question will show that our life becomes much easier if we consider randomized algorithms.

Question 4 Let S be a random subset of V, that is, a set of vertices that contains each vertex independently with probability $1/2$. Compute the expected size $\mathbb{E}[\delta(S)]$ of the cut induced by S. Use this to propose a very simple randomized algorithm that computes a cut in a graph (it is enough to describe your algorithm informally in natural language). Analyze its time complexity. ◇

Solution. Let S be a random set of vertices. We define for each edge $e \in E$ a binary random variable X_e that is one if and only if $e \in E(S, V \setminus S)$. By construction, $\sum_{e \in E} X_e = |E(S, V \setminus S)|$. Let $e = \{u, v\}$ be an edge of G. We compute (in full detail) that each edge has a probability of $1/2$ of being a cut edge :

$$\begin{aligned}
\Pr[X_e = 1] &= \Pr[e \in E(S, V \setminus E)] \\
&= \Pr[((u \in S) \wedge (v \notin S)) \vee ((u \notin S) \wedge (v \in S))] \\
&= \Pr[(u \in S) \wedge (v \notin S)] + \Pr[(u \notin S) \wedge (v \in S)] \\
&= \Pr[u \in S] \Pr[v \notin S] + \Pr[u \notin S] \Pr[v \in S] \\
&= \tfrac{1}{2} \cdot \tfrac{1}{2} + \tfrac{1}{2} \cdot \tfrac{1}{2} = \tfrac{1}{2}.
\end{aligned}$$

Since X_e is a binary random variable, we have $\mathbb{E}[X_e] = \Pr[X_e = 1] = 1/2$. Finally, by linearity of expectation, we have $\mathbb{E}[\delta(S)] = \mathbb{E}[|E(S, V \setminus S)|] = \mathbb{E}[\sum_{e \in E} X_e] = \sum_{e \in E} \mathbb{E}[X_e] = |E|/2$.

This suggests the following simple algorithm to compute a large cut : simply take a random set S of vertices ! More precisely, we start with $S = \emptyset$ and then, for each $v \in V$ independently, set $S := S \cup \{v\}$ with probability $1/2$ (and do nothing with the remaining probability of $1/2$). Clearly, this algorithm has a time complexity of $O(|V|)$. Note that this can even be less than the size of the input to the algorithm (the graph G), e.g., when there are $\Theta(n^2)$ edges. □

The previous question showed that randomized algorithms can be (i) more efficient, (ii) simpler, and (iii) easier to find than deterministic ones (at least when familiar with randomized algorithms). We now demonstrate another strong argument for randomized algorithms, and this will please even those who feel a little uneasy with an algorithm taking random decisions : Randomized algorithms can be used as a generic design method to find deterministic algorithms—find a randomized algorithm (which can be easy as seen above) and then *derandomize* it, that is, take out the randomness without making the algorithm much worse. Let us do this for the Maximum Cut problem :

To ease the notation, let us assume that our graph G has the vertex set $V = \{1, 2, \ldots, n\}$. Let us agree on the short-hand $[n] := \{1, 2, \ldots, n\}$. For a binary vector $x \in \{0,1\}^n$, let us define $\delta(x) := \delta(\{i \in [n] \mid x_i = 1\})$. So we simply allow ourselves to describe the set S via a binary vector in the natural way. Let $X_1, X_2, \ldots, X_n$ be independent binary random variables. Then the result of the previous question can be formulated as $\mathbb{E}[\delta(X_1, \ldots, X_n)] = |E|/2$. So far, we only introduced a simple notation. To check that you understood it (but not only for this reason), answer the following question.

Question 5 Prove (in few lines) :

$$\mathbb{E}[\delta(X_1, \ldots, X_n)] = \tfrac{1}{2}\mathbb{E}[\delta(0, X_2, \ldots, X_n)] + \tfrac{1}{2}\mathbb{E}[\delta(1, X_2, \ldots, X_n)].$$ ◇

Solution. Using the law of total probability, we compute

$$\begin{aligned}
&\mathbb{E}[\delta(X_1, \ldots, X_n)] \\
&\quad = \Pr[X_1 = 0]\, \mathbb{E}[\delta(X_1, X_2, \ldots, X_n) \mid X_1 = 0] + \Pr[X_1 = 1]\, \mathbb{E}[\delta(1, X_2, \ldots, X_n) \mid X_1 = 1] \\
&\quad = \tfrac{1}{2}\mathbb{E}[\delta(0, X_2, \ldots, X_n)] + \tfrac{1}{2}\mathbb{E}[\delta(1, X_2, \ldots, X_n)].
\end{aligned}$$ □

From the above and the simple observation that the maximum of two numbers is not smaller than their average, we obtain

$$\mathbb{E}[\delta(X_1,\ldots,X_n)] \leq \max_{x_1\in\{0,1\}} \mathbb{E}[\delta(x_1,X_2,\ldots,X_n)].$$

This last inequality immediately suggests how to derandomize the randomized Maximum Cut algorithm. Instead of randomly deciding whether vertex 1 goes into S or not, find an $x_1^* \in \{0,1\}$ such that $\mathbb{E}[\delta(x_1^*,X_2,\ldots,X_n)] = \max_{x_1\in\{0,1\}} \mathbb{E}[\delta(x_1,X_2,\ldots,X_n)]$. Note that $\mathbb{E}[\delta(x_1^*,X_2,\ldots,X_n)]$ is the expected size of the cut that stems from deterministically putting vertex 1 into S (if $x_1^*=1$) or not (if $x_1^*=0$) and then deciding for all other vertices randomly whether to put them in S or not. In a sense, we computed which outcome of the random variable X_1 is better for us and greedily took this decision. Of course, there is no reason to stop this cleverness after the first decision. So the full scheme looks like this :

- find an $x_1^* \in \{0,1\}$ such that
  $e_1 := \mathbb{E}[\delta(x_1^*,X_2,\ldots,X_n)] = \max_{x_1\in\{0,1\}} \mathbb{E}[\delta(x_1,X_2,\ldots,X_n)]$.
- find an $x_2^* \in \{0,1\}$ such that
  $e_2 := \mathbb{E}[\delta(x_1^*,x_2^*,X_3,\ldots,X_n)] = \max_{x_2\in\{0,1\}} \mathbb{E}[\delta(x_1^*,x_2,X_3,\ldots,X_n)]$.
- find an $x_3^* \in \{0,1\}$ such that
  $e_3 := \mathbb{E}[\delta(x_1^*,x_2^*,x_3^*,X_4,\ldots,X_n)] = \max_{x_3\in\{0,1\}} \mathbb{E}[\delta(x_1^*,x_2^*,x_3,X_4,\ldots,X_n)]$.
- find ...
- find an $x_n^* \in \{0,1\}$ such that
  $e_n := \mathbb{E}[\delta(x_1^*,x_2^*,x_3^*,\ldots,x_{n-1}^*,x_n^*)] = \max_{x_n\in\{0,1\}} \mathbb{E}[\delta(x_1^*,x_2^*,x_3^*,\ldots,x_{n-1}^*,x_n)]$.

Question 6 Give a sketchy argument why $e_k \geq e_{k-1}$ for all $k \in [n]$, where $e_0 := \mathbb{E}[\delta(X_1,\ldots,X_n)]$, and conclude that $e_n \geq |E|/2$. ◇

Solution. Analoguous to above, we have

$$\begin{aligned} e_k &= \max\{\mathbb{E}[\delta(x_1^*,\ldots,x_{k-1}^*,0,X_{k+1},\ldots,X_n)],\mathbb{E}[\delta(x_1^*,\ldots,x_{k-1}^*,1,X_{k+1},\ldots,X_n)]\} \\ &\geq \tfrac{1}{2}\mathbb{E}[\delta(x_1^*,\ldots,x_{k-1}^*,0,X_{k+1},\ldots,X_n)] + \tfrac{1}{2}\mathbb{E}[\delta(x_1^*,\ldots,x_{k-1}^*,1,X_{k+1},\ldots,X_n)] \\ &= e_{k-1}. \end{aligned}$$

By induction, we obtain $e_n \geq e_0$. We have $e_0 = |E|/2$ from the analysis of the randomized Maximum Cut algorithm. □

It remains to show that we can efficiently compute the x_k^*.

Question 7 Describe how you can compute $\mathbb{E}[\delta(x_1^*,\ldots,x_k^*,X_{k+1},\ldots,X_n)]$ efficiently for given $x_1^*,\ldots,x_k^* \in \{0,1\}$. From this, conclude that you can compute an x_{k+1}^* such that $\mathbb{E}[\delta(x_1^*,\ldots,x_k^*,x_{k+1}^*,X_{k+2},\ldots,X_n)]$ equals

$$\max\{\mathbb{E}[\delta(x_1^*,\ldots,x_k^*,0,X_{k+2},\ldots,X_n)],\mathbb{E}[\delta(x_1^*,\ldots,x_k^*,1,X_{k+2},\ldots,X_n)]\}$$

in time $O(\deg(k+1))$. ◇

Solution. Let $x_1^*,\ldots,x_k^* \in \{0,1\}$. Define $S := \{i \in [k] \mid x_i^* = 1\}$, $T := \{i \in [k] \mid x_i^* = 0\}$, and $X = [n] \setminus [k]$. Then

$$\mathbb{E}[\delta(x_1^*,\ldots,x_k^*,X_{k+1},\ldots,X_n)] = |E(S,T)| + |E(S\cup T,X)|/2 + |E(X)|/2$$

—for each edge in $E(S\cup T)$ it is already determined whether is it a cut edge or not, each other edge has a probability (not independent, but we don't care) of $1/2$ of becoming a cut edge. Hence we can redo the argument with the indicator random variables X_e from above. Consequently, we can easily compute $\mathbb{E}[\delta(x_1^*,\ldots,x_k^*,X_{k+1},\ldots,X_n)]$ by counting edges ; this takes a total time of $O(|E|)$.

To decide the optimal value for x^*_{k+1}, we first convince ourselved that with $X := [n] \setminus [k+1]$, we have

$$\begin{aligned}
&\mathbb{E}[\delta(x^*_1, \ldots, x^*_k, 0, X_{k+2}, \ldots, X_n)] \\
&\qquad = |E(S,T)| + |E(S, \{k+1\})| + |E(S \cup T \cup \{k+1\}, X)|/2 + |E(X)|/2, \\
&\mathbb{E}[\delta(x^*_1, \ldots, x^*_k, 1, X_{k+2}, \ldots, X_n)] \\
&\qquad = |E(S,T)| + |E(\{k+1\}, T)| + |E(S \cup T \cup \{k+1\}, X)|/2 + |E(X)|/2.
\end{aligned}$$

Consequently, to find the optimal value for x^*_{k+1}, we do not need to compute $\mathbb{E}[\delta(x^*_1, \ldots, x^*_k, 0, X_{k+2}, \ldots, X_n)]$ and $\mathbb{E}[\delta(x^*_1, \ldots, x^*_k, 1, X_{k+2}, \ldots, X_n)]\}$, but we can much faster decide by setting $x^*_{k+1} = 0$ if and only if $|E(S, \{k+1\})| \geq |E(\{k+1\}, T)|$. This last decision can be made by checking the edges incident with vertex $k+1$, so this takes time $O(\deg(k+1))$ when we store the graph in the adjacency list model (or any other data-structure that allows to enumerate the neighbors of the nodes in time proportional to their degree). □

Note that if we decide the values of the x^*_k as in the previous paragraph, then our derandomization of the randomized Maximum Cut algorithm re-discovered the greedy algorithm we started with. This is a good news, because we thus found the greedy algorithm using a general recipe instead of problem-specific thinking. The general recipe was

1. Find a simple randomized algorithm for your problem and analyze the expected solution value.
2. In an arbitrary order, re-consider all random decisions done by the algorithm and re-set their outcome to that outcome that gives the largest expected solution value (where the expectation is taken over all random decisions not re-considered yet).

As a final remark, let us comment that in the Maximum Cut example, the greedy algorithm was not that difficult to find that one feels the absolute need to do the detour via randomized algorithms and derandomization to find it. However, there are problems for which the best known deterministic algorithms were only discovered by derandomizing a randomized algorithm. An example are algorithms that build on a technique called *randomized rounding* [4, 5].

Références

[1] R. Freivalds. Probabilistic machines can use less running time. *In IFIP Congress*, pages 839–842, 1977.

[2] S. Khot. On the power of unique 2-prover 1-round games. *In Proceedings of the Thiry-fourth Annual ACM Symposium on Theory of Computing*, STOC '02, pages 767–775. ACM, 2002.

[3] M. X. Goemans et D. P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *J. ACM*, 42:1115–1145, 1995.

[4] P. Raghavan. Probabilistic construction of deterministic algorithms : Approximating packing integer programs. *J. Comput. Syst. Sci.*, 37:130–143, 1988.

[5] P. Raghavan et C. D. Thompson. Randomized rounding : A technique for provably good algorithms and algorithmic proofs. *Combinatorica*, 7:365–374, 1987.