Rapport du projet Honor

Ahmed Mhedhbi

April 2025

1 Introduction

Depuis la définition du calcul catalytique par Buhrman et al. en 2014 [BCK+14], de nombreux travaux ont exploré la puissance de ce modèle qui permet de réduire l'espace de travail interne d'une machine en utilisant une mémoire auxiliaire "catalytique" dont le contenu initial doit être restitué à la fin du calcul. En particulier, Cook et Mertz ont remis en cause une conjecture concernant l'évaluation d'arbre dès 2020 [CM20] et ont presque optimalement perfectionné leur technique par la suite [CM22], [CM24].

En parallèle, James Cook a publié une preuve de concept en C appliquant le même paradigme au problème de l'accessibilité dans un graphe orienté [Coo21]

- **Problématique :** Jusqu'où peut aller l'économie d'espace rendue possible par le calcul catalytique ? Ce projet se concentre sur l'étude du problème GEN, problème P-complet classique pour lequel les meilleures bornes actuelles reposent sur des programmes de branchement dits "incrémentaux" [GKM08] et souffrent de bornes inférieures exponentielles sous cette restriction.
- **Objectifs du projet :**
  1. *Composante théorique :* explorer si et comment l'inspiration du calcul catalytique permet d'améliorer, ne serait-ce que par un facteur constant, les bornes supérieures connues sur la taille des programmes de branchement pour GEN.
  2. *Composante appliquée :* implémenter en s'inspirant de Cook–Mertz (2024) [CM24], la méthode catalytique pour le problème TreeEval , puis l'étendre à GEN si possible.
- **Plan du rapport :** Le document s'articule comme suit :
  1. Notions préliminaires (modèles catalytique, registres, branchement) ;
  2. Mise en contexte (classes de complexité, définition de GEN)
  3. Implémentation de l'algorithme Cook–Mertz pour TreeEval
  4. Perspectives théoriques pour GEN ;
  5. Conclusion et questions ouvertes.

2 Notions préléminaires

2.1 Modèle de calcul catalytique

Nous pouvons considérer dans ce cas ci soit une machine de Turing, soit une machine à accès aléatoire (RAM) ou n'importe quel modèle raisonnable, le plus important que nous devons leur ajouter des bandes supplémentaires ou une région de mémoire additionnelle dont le contenu initial est inconnu. Cette bande (ou cette région) peut être modifiée pendant le calcul, mais doit retrouver exactement son contenu initial à l'état à la fin du calcul.

Définition 2.1 (Machine de Turing catalytique)**.** Soient $s, w : \mathbb{N} \to \mathbb{N}$ deux fonctions non décroissantes. Une machine de Turing catalytique M a trois bandes: une bande d'entrée en lecture seule, une bande de travail en lecture–écriture et une bande auxiliaire (dite "catalytique") en lecture–écriture.

On dit que M décide une langue $L \subseteq \{0,1\}^*$ en espace $s(n)$ et espace catalytique $w(n)$ si, pour tout mot d'entrée x de longueur n et toute configuration initiale $a \in 0, 1^{w(n)}$, M s'arrête en acceptation si $x \in L$ (sinon en rejet), accède à au plus $s(n)$ cellules de la bande de travail, accède à au plus $w(n)$ cellules de la bande auxiliaire initialement contenant a et retrouve exactement a à la fin.

Cette définition éclaire trois points: (i) le cadre de la machine de Turing et de la décidabilité, (ii) le modèle catalytique proprement dit, et (iii) la façon dont les deux se combinent. Nous verrons plus tard qu'une utilisation astucieuse de la mémoire catalytique nous permet potentiellement de dépasser les capacités d'une machine typique à espace borné.

2.2 Programmes de registres

Définition 2.2 (Programme de registres)**.** Soient $s, t : \mathbb{N} \to \mathbb{N}$ deux fonctions non décroissantes. Un programme de registres P de taille $s(n)$ et de temps $t(n)$ comprend $s(n)$ registres $R_1, \ldots, R_{s(n)}$ contenant des valeurs dans un anneau [1] $\mathcal{R}$, ainsi qu'une suite de $t(n)$ instructions de la forme

$$R_i \;\leftarrow\; R_i \;+\; p\big(R_1, \ldots, R_{i-1}, R_{i+1}, \ldots, R_{s(n)}\big),$$

où p est un polynôme dans les autres registres. Nous autorisons que P remplace une de ses instructions par l'exécution d'un autre programme de registres P', ce qui introduit deux mesures de complexité distinctes : le nombre d'instructions élémentaires et le nombre d'appels récursifs.

Définition 2.3 (Programme de registres transparent et propre)**.** Soit P un programme de registres utilisant s registres $R_1, \ldots, R_s$, initialisés respectivement aux valeurs $\tau_1, \ldots, \tau_s \in \mathcal{R}$.

- On dit que P est *transparent* s'il existe un indice $i \in \{1, \ldots, s\}$ et une valeur $v \in \mathcal{R}$ tels qu'à la fin de calcul du P nous avons,
$$R_i \;=\; \tau_i \;+\; v.$$
- Si, de plus, pour tout $j \neq i$ on a
$$R_j \;=\; \tau_j,$$
alors on dit que P est *propre*.

[1] Rappel: un *monoïde* $(M, *)$ est un ensemble muni d'une opération associative possédant un neutre. Un *anneau* $(A, +, \cdot)$ est un ensemble où $(A, +)$ forme un groupe abélien (chaque élément admet un opposé), $(A, \cdot)$ est un monoïde, et $\cdot$ est distributive sur$+$. Les inverses n'existent nécessairement donc que pour l'addition.

Dans les deux cas, on appelle v la valeur calculée par P et R_i le *registre cible*, et l'on note

$$P : R_i \leftarrow R_i + v.$$

De plus, pour tout programme propre P calculant v dans R_i, il existe un programme propre P^{-1} tel que

$$P^{-1} : R_i \leftarrow R_i - v$$

pour le même registre R_i et la même valeur v (cela peut être fait en exécutant les instructions dans l'ordre inverse tout en inversant chaque signe)

Remarque: le programme de registres de façon *isolée* n'est pas un modèle non-uniforme; le lemme qui suit sert précisément de pont entre programmes de registres et uniformité[2].

Lemme 2.1. Pour tout $n \in \mathbb{N}$, posons $c := c(n)$, $s := s(n)$, $t := t(n) \in \mathbb{N}$ et soit $\mathcal{R} := \mathcal{R}_n$ un anneau. Soit $f := f_n$ une fonction booléenne à n variables, et soit $P := P_n$ un programme de registres uniforme en espace c, utilisant s registres sur $\mathcal{R}$ et comportant t instructions au total, qui calcule f proprement. Alors f peut être calculée en espace $O\big(c + s \log |\mathcal{R}| + \log t\big)$.

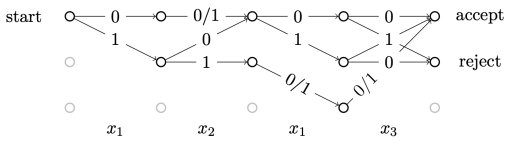
Intuition : Une MT uniforme génère l'instruction courante dans c bits, pour simuler un accès à un registre, il suffit de mémoriser son indice et sa valeur courante; cela coûte $\log |\mathcal{R}|$ bits par registre, donc $s \log |\mathcal{R}|$ au total. et un compteur sur $\log t$ bits qui indique quelle instruction on exécute; la somme donne la bounde annoncée.

Nous verrons plus loin toute la puissance des programmes de registres propres.

2.3 Programmes de branchement

Définition 2.4 (Programme de branchement)**.** Un programme de branchement est un graphe orienté acyclique muni d'un neoud de départ unique, de neoud de requête étiquetés par des variables x_i avec deux arêtes marquées 0 et 1, et de deux nœuds de sortie étiquetés 0 et 1 sans sorties. Le calcul commence au noeud de départ et, à chaque étape, suit l'arête correspondant à la valeur du bit lu.

Bien que l'espace soit formellement mesuré par le logarithme de la taille du programme de branchement — c'est-à-dire par le nombre de bits nécessaires pour suivre notre position dans le graphe à chaque instant — on dispose de notions plus fines qui sont pertinentes. On dit qu'un programme est stratifié si ses neouds peuvent être organisés en couches, en commençant par la source à la couche 0, de sorte que toutes les arêtes sortant d'un neoud de la couche i mènent à des neouds de la couche $i + 1$. Dans ce cas, nous pouvons parler du temps et de l'espace du programme en termes de longueur (nombre de couches) et de largeur (taille maximale d'une couche), respectivement.



Un programme de branchement de largeur 3 et longueur 4 pour $(x_1 \wedge x_2) \vee (x_1 \oplus x_3)$.

[2]On parle de modèle *non-uniforme* lorsqu'on autorise, pour chaque taille d'entrée n, un dispositif distinct (un circuit, un programme de registres, etc.). Le lemme suivant montre qu'il est possible de décrire, *en espace logarithmique*, la famille complète de ces programmes, rétablissant ainsi l'uniformité du modèle.

Lien avec les programmes de registres: Tout programme de registres utilisant k registres sur un corps fini $\mathcal{K}$ et comportant $t(n)$ instructions peut être simulé par un programme de branchement de largeur au plus $|\mathcal{K}|^k$ et de longueur $t(n)$.

3 Mise en contexte

3.1 Définitions et inclusions des classes de complexité pertinentes

Nous définissons les classes autour L (l'ensemble des langages reconnus par une MT (machine de Turing) fonctionnant en espace logarithmique), NL (l'ensemble des langages reconnus par MT *non déterministe* fonctionnant en espace logarithmique) P (l'ensemble des langages reconnus par une MT fonctionnant en temps polynomiale) NP (l'ensemble des langages reconnus par une MT *non déterministe* fonctionnant en temps polynomiale) ;PSPACE (l'ensemble des langages reconnus par une MT fonctionnant en espace polynomiale)
Il est bien établi que:

$$\mathrm{L} \subseteq \mathrm{P} \subseteq \mathrm{NP} \subseteq \mathrm{PSPACE},$$

et, par le théorème de hiérarchie de l'espace (Cook–Reckhow, 1973), $\mathrm{L} \not\subseteq \mathrm{PSPACE}$. La question qui nous intéresse est $\mathrm{L} \neq \mathrm{P}$?

3.2 Le problème GEN

Nous écrivons $[n] = \{1, 2, \ldots, n\}$. Pour $T \subseteq [n]^3$ et $S \subseteq [n]$, la *fermature* de $\langle S \rangle_T$ est la plus petite $S' \supseteq S$ telle que

$$\forall (i,j,k) \in T : \quad i \in S' \wedge j \in S' \implies k \in S'.$$

Problème GEN :

- *Donné :* une fonction $g : [n] \times [n] \to [n]$, définissant $T^g = \{(i, j, g(i, j)) : i, j \in [n]\} \subseteq [n]^3$;
- *Déterminer :* si $n \in \langle \{1\} \rangle_{T^g}$.

Il est facile de trouver un algotithme en temps polynomiale pour le problème GEN.

$T := \{1\}$;
tant que $\exists\, i, j \in T$ tel que $k := g(i, j) \notin T$ **faire**
  $T := T \cup \{k\}$;
fin tant que;
accepter si $n \in T$.

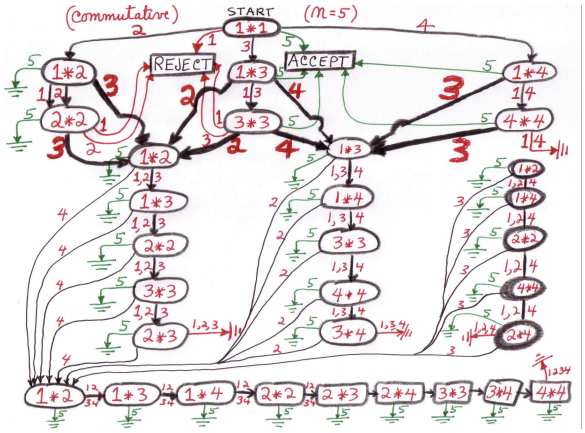
L'algorithme effectue au plus n insertions, chacune nécessitant parcourir n^2 paires d'indices au pire, soit un temps $O(n^3)$. Il suffit d'un tableau de marquage de taille n (un bit par case) et deux pointeur de $O(\log n)$ bits pour parcourir les paires, d'où une complexité en espace $O(n)$.

Enfin, GEN est P-complet par réduction log-espace, ce qui en fait un candidat naturel pour distinguer L de P.

3.3 Programmes de branchement incrémentaux (Gál–Koucký–McKenzie, 2008)

Un *programme incrémental* limite la "mémoire" : en tout point du calcul, il ne peut tester qu'un produit impliquant au moins un élément encore utile. Plus formellement, soit $\pi = (v_0, \dots, v_\ell)$ un chemin dans le graphe. Un entier $i \in [n]$ est dit *utile* pour π si la dernière arête parcourue évalue $j * i = k$ (ou $j * i = k$) et qu'aucune arête antérieure n'a déjà testé $k * j = i$. Pour chaque neoud u on pose

$$U(u) = \big\{ i \mid i \text{ est utile pour un chemin partant de } u \big\}, \qquad \max U(P) = \max_u |U(u)|.$$



Un programme de branchement incrémental qui résout GEN version commutative.

Borne supérieure. Une construction directe montre qu'il existe, pour tout n, un programme incrémental *déterministe* calculant n–GEN dont la taille est $O(n^2 2^n)$ (Thm 6.2).

Borne inférieure. À l'inverse, tout programme incrémental (déterministe ou même non déterministe) qui ne lit que cette mémoire utile doit contenir au moins 2^{n^ε} neouds pour une constante $\varepsilon > 0$ (Thm 4.4). L'argument s'appuie sur le *Symmetrization Lemma* : en ré-étiquetant les variables on transforme un programme k-naire en un programme n-aire tout en gardant la même valeur de $\max U(P)$; nous transfèrrons ensuite des bornes de profondeur provenant des circuits monotones.

Ces résultats fournissent la première séparation exponentielle connue pour un sous-modèle naturel de programmes de branchement incrémentaux, et motivent le recours à d'autres techniques.

3.4 Approches modernes et objectifs du projet

Le modèle catalytique ajoute à la machine classique une bande auxiliaire (parfois de taille exponentielle) dont le contenu initial est inconnu mais doit être restitué *à l'identique* à la fin du calcul. L'idée c'est de s'inspirer des approches catalitique à fin d'utiliser/réutiliser un *petit espace*.

Le problème TreeEval. Le problème d'évaluation d'arbre $\textsc{TreeEval}_{h,k}$ est paramétré par une hauteur h et une taille d'alphabet k. L'entrée est un arbre binaire complet de hauteur h, où chaque feuille est étiquetée par un élément de $[k]$ et chaque neoud interne par une fonction de $[k] \times [k]$

vers $[k]$. La sortie est la valeur de la racine de l'arbre, celui-ci étant évalué de bas en haut de manière naturelle. Nous omettrons souvent les indices et écrirons simplement TreeEval. La taille de l'entrée pour $\text{TreeEval}_{h,k}$ est

$$(2^{h-1}-1)\,k^2\log k \;+\; 2^{h-1}\log k \;=\; O\big(2^h\,\mathrm{poly}(k)\big).$$

Appartenance à P. Ce problème appartient à P : il peut être résolu en temps polynomial en évaluant tous les neouds, en partant des feuilles, dans un ordre qui garantit que les deux enfants d'un neoud sont traités avant leur parent.

Non-existence d'algorithme en espace logarithmique. Cependant, ce n'est pas un algorithme en espace logarithmique. L'espace utilisé dépend de l'ordre d'évaluation des neouds, puisque la valeur d'un enfant peut être oubliée dès que son parent est évalué. Un argument fondé sur un "jeu de pebbling" montre que même la version la plus économe en espace doit, à un moment donné, stocker simultanément h valeurs, nécessitant un espace

$$\Omega\big(h\log k\big) \;\subseteq\; \omega(h+\log k) \quad \text{pour } h,k \text{ non constants.}$$

Algorithme de pebbling. Nous appelons cette méthode l'*algorithme de pebbling*. Plus précisément, la version la plus efficace de cet algorithme utilise un espace $\Theta(h\log k)$.

Pendant près de vingt ans, nous avons soupçonné que cette borne était optimale. Pourtant Cook et Mertz (2020) ont montré

$$\text{TreeEval} \in \text{SPACE}\Big(\tfrac{\log^2 n}{\log\log n}\Big),$$

ouvrant la voie à d'autres d'améliorations. Leur technique repose sur des *programmes de registres propres* qui déversent les calculs intermédiaires dans la bande catalytique, puis invoquent un *programme inverse* pour remettre exactement les registres d'origine, libérant ainsi l'espace de travail. En 2024 les mêmes auteurs repoussent la frontière à

$$\text{TreeEval} \in \text{SPACE}(\log n\,\log\log n),$$

Objectifs spécifiques de ce projet. Nous chercherons à transposer cette philosophie au cadre des programmes de branchement incrémentaux pour GEN.

Enfin, l'ambition à long terme est d'améliorer les bornes connues sur la taille de programmes de branchement résolvant GEN à l'aide ces approches.

4 Implémentation de l'algorithme Cook–Mertz – TreeEval

Cette section fournit une version unifiée qui intègre :

- les **préliminaires algébriques**
- la **version warm-up** (Lemme 9) de Cook–Mertz.
- la **version optimisée** par segmentation (Lemme 10).
- une preuve de concept de la version warm-up, implémentée en Python et en C++.

4.1 Préliminaires algébriques

L'algorithme se base sur le lemme suivant :

Lemme 4.1. Soit $\mathcal{K}$ un corps fini et posons $m = |\mathcal{K}| - 1$. Soit $\omega \in \mathcal{K}$ une racine primitive de l'unité d'ordre m. Considérons un polynôme $p : \mathcal{K}^n \to \mathcal{K}$ de degré $d < m$, et des éléments $\tau_i, x_i \in \mathcal{K}$ pour tout $i \in [n]$. Alors

$$\sum_{j=1}^{m} (-1)^j \, p\big(\omega^j \tau_1 + x_1, \, \ldots, \, \omega^j \tau_n + x_n\big) \;=\; p\big(x_1, \ldots, x_n\big).$$

Toute la démarche qui mène à ce résultat se trouve dans le papier de Tree [CM24].

Choix du *corps* $\mathcal{K}$ et d'une racine primitive. Nous travaillons dans un corps fini $\mathcal{K} = F_q$, de taille $q \geq 2\log k + 3$, pour lequel l'ordre multiplicatif $m = q - 1$ domine $2\log k$. Fixons un générateur ω de $\mathcal{K}^\times$. Toutes les opérations de base (addition, multiplication, calcul de puissances) s'effectuent en $O(\log q)$ bits, donc en $O(\log n)$ espace. un *corps* choix possible de corps qui satisfait à la fois la condition et l'encodage binaire de l'entrée est $\mathbb{F}_{2^{\lceil \log(2\lceil \log k \rceil + 2) \rceil}}$.

Permutation des blocs de mémoire $(\hat{x}, \hat{y}, \hat{z})$. Suivant Goldreich, nous conservons trois blocs de taille $\log k$ dans la bande catalytique et appliquons soit :

$$(\hat{x}, \hat{y}, \hat{z}) \;\longmapsto\; (\hat{y}, \hat{z}, \omega^j \hat{x}),$$

soit :

$$(\hat{x}, \hat{y}, \hat{z}) \;\longmapsto\; (\hat{x}', \hat{z}, \omega^j \hat{y}),$$

de sorte que le résultat courant s'écrive toujours dans le troisième composant, le deuxième et le troisième composant servent comme des résultats intermédiaires.

4.2 Algorithme TreeEval-Warm (Lemme 9) à la Goldreich

Afin d'utiliser le lemme 4.1, nous devons définir l'évaluation dans chaque noeud dans l'arbre comme une fonction polynomiale qui calcule bit à bit la valeur dans un noeud interne u (tout pour contourner les restrictions qui donnent des bornes inférieures):

$$\big(f_u(y,z)\big)_i \;=\; \sum_{\alpha,\beta,\gamma \in [k]^3} [\alpha_i = 1]\,[f_u(\beta,\gamma) = \alpha]\,[y = \beta]\,[z = \gamma].$$

$$e(y,\beta) \;=\; \prod_{i=1}^{\lceil \log k \rceil} \big(1 - y_i + (2y_i - 1)\beta_i\big),$$

qui vérifie $e(y,\beta) = [y = \beta]$ pour $y_i \in \{0,1\}$. Alors, nous avons:

$$q_u(y,z) = \sum_{\substack{\alpha,\beta,\gamma \in [k]^3 \\ \alpha_i = 1}} [f_u(\beta,\gamma) = \alpha]\; e(y,\beta)\, e(z,\gamma) \tag{1}$$

L'algorithme en details:

Algorithm 1 TreeEval-Warm$(u, \tau, \hat{x}, \hat{y}, \hat{z})$

if u est une feuille **then**
    extraire les bits de v_u et mettre à jour $\hat{z}$
    return
end if
for $j = 1$ **to** m **do** ▷ Masquage : multiplication par ω_j
    for $i = 1$ **to** $\log k$ **do**
        $\hat{x}_i \leftarrow \omega^j \cdot \hat{x}_i$
        $\hat{y}_i \leftarrow \omega^j \cdot \hat{y}_I$
    end for ▷ Permutation circulaire des registres
    $(\hat{x}, \hat{y}, \hat{z}) \leftarrow (\hat{y}, \hat{z}, \hat{x})$
    TreeEval-Warm$(u_0, 0, \hat{x}, \hat{y}, \hat{z})$
    $(\hat{x}, \hat{z}) \leftarrow (\hat{z}, \hat{x})$
    TreeEval-Warm$(u_1, 0, \hat{x}, \hat{y}, \hat{z})$
    ▷ Mise à jour via $q_{u,i}$
    for $i = 1$ **to** $\log k$ **do**
        $\hat{y}_i \leftarrow \hat{y}_i - q_{u,i}(\hat{x}, \hat{y})$
    end for
    ▷ Appels inverses et restauration
    $(\hat{x}, \hat{z}) \leftarrow (\hat{z}, \hat{x})$
    TreeEval-Warm$(u_0, 1, \hat{x}, \hat{y}, \hat{z})$
    $(\hat{x}, \hat{z}) \leftarrow (\hat{z}, \hat{x})$
    TreeEval-Warm$(u_1, 1, \hat{x}, \hat{y}, \hat{z})$
    ▷ Permutation inverse et démasquage
    $(\hat{x}, \hat{y}, \hat{z}) \leftarrow (\hat{x}, \hat{z}, \hat{y})$
    for $i = 1$ **to** $\log k$ **do**
        $\hat{x}_i \leftarrow \hat{x}_i / \omega^j$
        $\hat{y}_i \leftarrow \hat{y}_i / \omega^j$
    end for
end for

Espace utilisé. Dans la version warm-up (TreeEval-Warm), chaque itération emploie trois blocs de registres de taille $\ell = \lceil \log k \rceil$ et un compteur de longueur $O(\log m) = O(\log \log k)$, le tout sur une profondeur récursive h. L'espace total est donc

$$\mathcal{O}\big((h + \ell)\, \log \log k\big) = \mathcal{O}\big((h + \log k)\, \log \log k\big) = \mathcal{O}(\log n \log \log n) \quad \text{bits.}$$

comme annoncé.

preuve de concept Nous avons implémenté la version warm-up (TreeEval-Warm) :

- En **Python** lien : prototype fonctionnel basé sur le pseudocode de Cook–Mertz et l'intuition de Goldreich ;
- En **C++** lien : implémentation optimisée

Ces deux versions servent de prototype fonctionnel et offrent un aperçu plus concret du fonctionnement de ces algorithmes.

Conséquence via le Lemme 10. On choisit un sous-corps $\mathbb{F}_{q^r} \subset \mathbb{F}_{q^\ell}$ avec $r \mid \ell$, ce qui identifie chaque élément de $\mathbb{F}_{q^\ell}$ à un vecteur de ℓ/r coordonnées dans $\mathbb{F}_{q^r}$ et ramène le nombre de registres de ℓ à ℓ/r. En réinjectant cette compression dans l'analyse d'espace, on obtient une utilisation globale de

$$\mathcal{O}\big(h \log\log k + \log k\big).$$

5 Prospectives théoriques pour GEN

5.1 Formulation par circuits

Nous pouvons décrire l'opération d'ajout d'un nouvel élément k à la fermeture $T \mapsto \langle T \rangle$ par le circuit à portes $\vee$ et $\wedge$ donné par

$$v_{i,j} \;=\; v_{i-1,j} \;\vee\; \bigvee_{a,b\in[n]} \big(v_{i-1,a} \wedge v_{i-1,b} \bigwedge_{k=1}^{\log n} [\, g(a,b)_k \wedge j_k \,]\big),$$

où $v_{i,j} = 1$ signifie que $j \in T_i$. Ce circuit a profondeur $\Theta(n)$ et taille $\Theta(n^4)$.

1. **Transposer le Lemme 9 de Cook–Mertz.** Une configuration $T \subseteq [n]$ se code par un mot $x \in \{0,1\}^n$, identifié à un élément de $\mathbb{F}_{2^n}$. Pour chaque paire (a,b) nous voulons un *programme de registres propre* qui, sur l'entrée x:

   - lit *seulement* les deux bits x_a, x_b;
   - écrit proprement dans un registre cible le bit $\big[g(a,b) \in \langle T \rangle\big]$;
   - s'inverse, restaurant exactement x.

   Cette routine est l'analogue direct de leur Lemme 9, sauf qu'ici la fonction manipulée prend n bits en entrée et n'en renvoie qu'un seul.

2. **Compression (à la Lemme 10).** Pour diminuer le nombre de registres actifs on choisit un sous-corps $\mathbb{F}_{2^r} \subset \mathbb{F}_{2^n}$ avec $r \mid n$. Chaque élément de $\mathbb{F}_{2^n}$ devient alors un vecteur de taille n/r sur $\mathbb{F}_{2^r}$: on passe de n registres à n/r, mais la routine est allongée d'un facteur $2^n - 1$. On obtient un compromis

$$\#\text{registres} \simeq \frac{n}{r}, \qquad \text{longueur} \simeq (2^n - 1)\,\mathrm{poly}(n).$$

3. **Assemblage incrémental.** On parcourt les n^2 paires selon l'ordre $(1,1),(1,2),\ldots,(n,n)$. Pour chaque paire on exécute la routine compressée puis son inverse, sans jamais dépasser n/r registres actifs, exactement dans l'esprit des programmes de branchement incrémentaux de Gál–Koučký–McKenzie. (explorer la petite marge de n^2)

4. **Analyse du coût.** Groupons les appels par blocs afin de réutiliser les registres; un calcul sommaire donne alors une taille globale

$$\mathrm{Size}(n) \;=\; 2^{\,O(n/r)}\,\mathrm{poly}(n).$$

   et essayons d'optimiser r.

5. **Pont avec la simulation temps vers espace $t \mapsto \sqrt{t \log t}$ de Williams [Wil25].** Williams montre que tout calcul en temps t se simule en espace $O(\sqrt{t \log t})$ En particulier un circuit borné-fan-in de taille s s'évalue en $O(\sqrt{s}\,\mathrm{polylog}\, s)$ espace, puis se convertit en un programme de branchement de taille $2^{O(\sqrt{s}\,\log^k s)}$. Si l'on parvient à décrire GEN par des circuits de taille quadratique par rapport à la taille d'entrée (la borne actuelle), nous retombons sur la barrière classique. Un gain (même sub-quadratique) sur la taille des circuits se répercuterait automatiquement sur la taille des programmes de branchement via cette simulation, aura des potentiels sur L vs P. (peut être la diffuculté vient d'ici).

6 Conclusion

La question demeure ouverte. Nous soulevons toutefois de nouvelles problématiques :

Questions ouvertes.

- Poursuivre la compression : peut-on choisir r de manière adaptative pour chaque portion de l'instance ?
- Optimiser le *pebbling* du graphe de dépendances avant de le convertir en programme de branchement, à l'instar des récentes accélérations temps–espace ?

References

[BCK+14] Harry Buhrman, Richard Cleve, Michal Koucky, Bruno Loff, and Florian Speelman. Computing with a full memory: catalytic space. In *Symposium on Theory of Computing, STOC 2014*, pages 857–866, 2014.

[CM20] James Cook and Ian Mertz. Catalytic approaches to the tree evaluation problem. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020*, pages 752–760. ACM, 2020.

[CM22] James Cook and Ian Mertz. Trading time and space in catalytic branching programs. In *37th Computational Complexity Conference, CCC 2022*, volume 234 of LIPIcs, pages 8:1–8:21. Schloss Dagstuhl, 2022.

[CM24] James Cook and Ian Mertz. Tree evaluation is in space o(log n ·log log n). In *Proceedings of the 56th Annual ACM Symposium on Theory of Computing, STOC 2024*, pages 1268–1278. ACM, 2024.

[CMW+12] S. Cook, P. McKenzie, D. Wehr, M. Braverman, and R. Santhanam. Pebbles and branching programs for tree evaluation. *ACM Transactions on Computation Theory*, 3(2):4, 2012.

[Coo74] S. Cook. An observation on time-storage trade-off. *Journal of Computer and System Sciences*, 9(3):308–316, 1974.

[Coo21] James Cook. How to borrow memory. `https://www.falsifian.org/blog/2021/06/04/catalytic/`, January 2021.

[FMST24] Marten Folkertsma, Ian Mertz, Florian Speelman, and Quinten Tupker. Fully characterizing lossy catalytic computation. CoRR, abs/2409.05046, 2024.

[GKM08] A. Gál, M. Koucký, and P. McKenzie. Incremental branching programs. *Theory of Computing Systems*, 43(2):159–184, 2008.

[Gol24] Oded Goldreich. On the Cook–Mertz tree evaluation procedure. Technical report, Weizmann Institute of Science, 2024.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. Time–space tradeoffs for directed computations. *Journal of the ACM*, 24(2):309–316, 1977.

[Wil25] Ryan Williams. Simulating time with square-root space. Preprint, arXiv:2502.17779, 2025.