Université de Bourgogne, UFR Sciences et Techniques, Département I.E.M.

Projet « Algorithmique et complexité »
Pavage en carrés

dahmouni mohamed amine ; majd E-rami

Table des matières

2022-2023

1 Introduction

Le pavage d'un domaine avec des carrés de différentes tailles est un problème classique en mathématiques et en informatique. Le but est de trouver une disposition des carrés qui couvre entièrement le domaine sans chevauchement ni espaces vides. Dans ce rapport, nous explorons deux approches principales pour résoudre ce problème : la recherche de solutions mathématiques en utilisant des combinaisons de carrés parfaits et l'algorithme de backtracking pour le pavage avec des codes de Bouwkamp. Ces deux méthodes sont complémentaires, la première permettant de générer des solutions potentielles et la seconde vérifiant leur validité en termes de placement réel en 2D.

2 Présentation du problème et des données

Le problème abordé dans ce rapport consiste à paver un carré de taille n x n avec des carrés de différentes tailles, sans chevauchement ni espaces vides. Nous cherchons à trouver des combinaisons de carrés dont la somme des aires est égale à l'aire du carré initial et à vérifier si ces combinaisons peuvent être placées de manière à couvrir entièrement le carré initial en 2D.

Pour représenter un carré et ses pavés, nous utilisons une matrice d'entiers, où chaque élément de la matrice correspond à une cellule du carré initial. La valeur de chaque élément représente la taille du carré qui occupe cette cellule. Par exemple, un élément de valeur 0 indique que la cellule est vide, tandis qu'un élément de valeur 4 indique qu'un carré de taille 4x4 occupe cette cellule et les cellules adjacentes.

Les combinaisons de carrés parfaits sont représentées sous forme de listes d'entiers, où chaque entier correspond à la taille d'un carré. Par exemple, la liste [18 ; 15 ; 14 ; 10 ; 9 ; 8 ; 7 ; 4 ; 1] représente une combinaison de carrés de tailles 18x18, 15x15, 14x14, 10x10, 9x9, 8x8, 7x7, 4x4 et 1x1. Les codes de Bouwkamp sont également représentés de cette manière.

Pour les algorithmes et les fonctions auxiliaires, nous utilisons des structures de données telles que des listes, des options et des enregistrements pour stocker et manipuler les informations pertinentes. Par exemple, nous utilisons un enregistrement "position" pour stocker les coordonnées x et y d'une cellule dans la matrice.

Dans la section suivante, nous décrirons en détail les algorithmes de recherche de solutions mathématiques et de backtracking avec des codes de Bouwkamp, ainsi que les structures de données et les fonctions auxiliaires utilisées pour résoudre ce problème.

3 Explication des algorithmes et fonctions auxiliaires

Dans cette section, nous décrivons les algorithmes principaux et les fonctions auxiliaires utilisées pour résoudre le problème de pavage. Les algorithmes principaux sont : "find_square_sum" pour trouver les solutions mathématiques et "backtrack_paving" pour vérifier la validité des solutions en utilisant le backtracking et les codes de Bouwkamp.

3.1 Fonction pour obtenir les carrés parfaits jusqu'à n inclus :

La fonction "get_squares" permet d'obtenir tous les carrés parfaits jusqu'à n inclus. Elle prend en entrée un entier n et renvoie une liste de carrés parfaits.

```
let get_squares n =
  let rec aux i acc =
    if i*i > n then acc
    else aux (i+1) (acc @ [i*i])
  in
  aux 1 [];;
```

3.2 Fonction pour trouver les combinaisons de carrés parfaits dont la somme est égale à n :

La fonction "find_square_sum" est une fonction récursive qui trouve toutes les combinaisons de carrés parfaits dont la somme est égale à un nombre donné n. Cette fonction prend en entrée deux arguments :"n", qui représente le nombre dont on veut trouver la somme des carrés parfaits, et"squares", qui est une liste de carrés parfaits préalablement calculée. Voici le code de la fonction :

```
let rec find_square_sum n squares =
  match squares with
  | [] -> []
  | x :: xs ->
    let remaining = n - x in
    if remaining < 0 then []
    else if remaining = 0 then [[x]]
    else let solutions = find_square_sum remaining xs in
      if solutions <> [] then
        List.map (fun sol -> x :: sol) solutions @ find_square_sum n xs
      else
        find_square_sum n xs;;
```

La fonction fonctionne en utilisant un algorithme de type "diviser pour régner". Elle commence par vérifier si la liste des carrés parfaits est vide, auquel cas elle renvoie une liste vide, indiquant qu'aucune combinaison n'a été trouvée. Si la liste n'est pas vide, la fonction extrait le premier élément de la liste "squares", représenté par "x", et calcule la différence entre "n" et "x", stockée dans la variable remaining.

Ensuite, la fonction vérifie si "remaining" est négatif. Si c'est le cas, cela signifie que la combinaison actuelle n'est pas valide, et la fonction renvoie une liste vide. Si "remaining" est égal à 0, cela signifie que la combinaison actuelle est valide, et la fonction renvoie une liste contenant "x" comme seule solution.

Si "remaining" est positif, la fonction effectue un appel récursif à elle-même avec remaining et "xs" (la liste des carrés restants) comme arguments. Elle stocke le résultat de cet appel récursif dans la variable "solutions". Si "solutions" n'est pas vide, la fonction construit de nouvelles combinaisons en ajoutant "x" à chaque solution trouvée, puis concatène ces nouvelles combinaisons avec les solutions trouvées en appelant récursivement la fonction avec "n" et "xs". Si "solutions" est vide, la fonction continue à chercher d'autres combinaisons en appelant récursivement la fonction avec "n" et "xs".

La fonction "find_square_sum" explore ainsi toutes les combinaisons possibles de carrés parfaits et renvoie une liste contenant toutes les combinaisons valides dont la somme est égale à "n".

Il est également possible d'envisager d'utiliser la programmation dynamique pour résoudre le problème de la recherche des combinaisons de carrés parfaits dont la somme est égale à un nombre donné "n".

Cependant, dans notre cas, l'utilisation de la programmation dynamique peut ne pas être la meilleure solution, car elle ne tient pas compte du critère d'unicité des carrés dans chaque combinaison. La programmation dynamique stockerait les solutions intermédiaires pour chaque sous-problème et les réutiliserait pour résoudre les problèmes de taille supérieure, ce qui pourrait conduire à des combinaisons non valides contenant des carrés répétés.

En utilisant l'approche "diviser pour régner" implémentée dans la fonction "find_square_sum", nous sommes en mesure de garantir l'unicité des carrés dans chaque combinaison, car chaque combinaison est construite en ajoutant un seul carré à la fois, et nous explorons toutes les combinaisons possibles sans répétition.

Ainsi, bien que la programmation dynamique puisse être une technique efficace pour résoudre certains problèmes, elle n'est pas nécessairement la meilleure option pour ce cas spécifique, en raison du

critère d'unicité des carrés dans les combinaisons valides. L'approche "diviser pour régner" utilisée dans la fonction "find_square_sum" est plus adaptée pour garantir que les combinaisons générées respectent ce critère d'unicité.

3.3 Fonction pour trouver le nombre de combinaisons de carrés parfaits dont la somme est égale à n :

La fonction "find_num_square_sum" prend en entrée un entier n et une liste de carrés parfaits, puis renvoie le nombre de combinaisons de carrés parfaits dont la somme est égale à n. Cette fonction est similaire à "find_square_sum", mais elle renvoie simplement le nombre de solutions plutôt que les solutions elles-mêmes.

```
let rec find_num_square_sum n squares =
  match squares with
  | [] -> 0
  | x :: xs ->
    let remaining = n - x in
    if remaining < 0 then 0
    else if remaining = 0 then 1
    else find_num_square_sum remaining xs + find_num_square_sum n xs;;
```

3.4 Fonctions pour placer et retirer un carré dans la matrice :

Les fonctions "place_square" et"remove_square" prennent en entrée une matrice représentant le carré initial, la taille d'un carré à placer ou à retirer et la position où le carré doit être placé ou retiré. Elles modifient la matrice en conséquence.

```
let place_square board square_size pos =
    for y = pos.y to pos.y + square_size - 1 do
      for x = pos.x to pos.x + square_size - 1 do
        board.(y).(x) <- square_size
      done
    done;;


  let remove_square board square_size pos =
    for y = pos.y to pos.y + square_size - 1 do
      for x = pos.x to pos.x + square_size - 1 do
        board.(y).(x) <- 0
      done
    done;;
```

3.5 Fonction pour trouver la prochaine position vide dans la matrice :

La fonction "find_next_empty_position" joue un rôle essentiel dans l'algorithme de pavage avec backtracking. Elle permet de déterminer la prochaine position vide dans la matrice "board" où un carré de taille "square_size" peut être placé.

```
let rec find_next_empty_position board square_size start_pos  =
    let rec find_in_row row col =
      if col + square_size > Array.length board.(0) then
        find_next_empty_position board square_size {x=0; y=row + 1}
      else if row + square_size > Array.length board then None
      else if board.(row).(col) = 0 && board.(row).(col + square_size - 1) = 0 &&
    board.(row + square_size - 1).(col) = 0 && board.(row + square_size - 1).(col +
    square_size - 1) = 0 then
        Some {x=col; y=row}
      else
        find_in_row row (col + max 1 board.(row).(col))
    in
    find_in_row start_pos.y start_pos.x;;
```

La fonction prend en entrée trois arguments :
— board : la matrice représentant l'état actuel du pavage.
— square_size : la taille du carré à placer.
— start_pos : la position à partir de laquelle la recherche de la position vide commence.

La fonction "find_next_empty_position" fonctionne en utilisant une approche récursive pour parcourir la matrice board. Elle commence par chercher une position vide dans la rangée actuelle à partir de la colonne indiquée par "start_pos". Si une position valide est trouvée, elle retourne cette position sous la forme d'un enregistrement avec les coordonnées (x, y). Si la position vide n'est pas trouvée dans la rangée actuelle, la fonction passe à la rangée suivante et continue la recherche.

La recherche de la position vide est effectuée en vérifiant si les cellules du coin supérieur gauche, du coin supérieur droit, du coin inférieur gauche et du coin inférieur droit de la zone où le carré de taille "square_size" doit être placé sont vides (c'est-à-dire que leur valeur est égale à 0).

Si la fonction atteint la fin de la matrice sans trouver de position vide, elle retourne None, indiquant qu'aucune position vide n'a été trouvée pour placer le carré.

La fonction "find_next_empty_position" est utilisée dans la fonction "backtrack_paving" pour déterminer où placer le carré de taille "square_size" à chaque étape de l'algorithme de backtracking. Elle aide à explorer toutes les combinaisons possibles de placement de carrés pour vérifier la validité d'une solution.

3.6 Fonction de backtracking pour vérifier la validité des solutions :

La fonction "backtrack_paving" est une fonction récursive essentielle à l'algorithme de pavage. Elle permet de vérifier la validité des solutions trouvées par la fonction "find_square_sum" en plaçant les carrés sur le plateau de manière à respecter les contraintes du problème. Cette fonction de backtracking explore toutes les combinaisons possibles de placement des carrés dans la matrice board et s'assure qu'elles correspondent à une solution valide.

```
let rec backtrack_paving board bouwkamp_code =
      match bouwkamp_code with
      | [] -> Some board
      | square_size :: remaining_sizes ->
        let rec try_positions pos_opt =
          match pos_opt with
          | None -> None
          | Some pos ->
              place_square board square_size pos;
              match backtrack_paving board remaining_sizes with
              | Some solution_board -> Some solution_board
              | None ->
                remove_square board square_size pos;
                try_positions (find_next_empty_position board square_size {x=pos.x+1;
    y=pos.y})
        in
        try_positions (find_next_empty_position board square_size {x=0; y=0});;
```

La fonction prend en entrée deux arguments :
La fonction prend en entrée trois arguments :
— "board" : la matrice représentant l'état actuel du pavage.
— "bouwkamp_code" : une liste de carrés parfaits représentant une combinaison trouvée par la fonction "find_square_sum".

Le processus de backtracking se déroule de la manière suivante :

1. Si la liste "bouwkamp_code" est vide, cela signifie que tous les carrés ont été placés avec succès dans la matrice "board". La fonction retourne alors la matrice "board" comme solution valide.

2. Si la liste "bouwkamp_code" n'est pas vide, la fonction récupère la taille du premier carré à placer ("square_size") et essaie de le placer dans la matrice "board" en utilisant la fonction "find_next_empty_position".
3. Si la fonction "find_next_empty_position" retourne une position valide, la fonction "place_square" est appelée pour placer le carré de taille "square_size" à cette position. Ensuite, la fonction "backtrack_paving" est appelée récursivement avec la matrice "board" mise à jour et la "bouwkamp_code"liste réduite (sans le carré placé).
4. Si la récursion retourne une solution valide, la fonction "backtrack_paving" retourne cette solution. Sinon, elle utilise la fonction "remove_square" pour retirer le carré de taille "square_size" de la position précédemment trouvée et continue à chercher une nouvelle position pour placer le carré.
5. Si la fonction "find_next_empty_position" ne trouve pas de position valide pour placer le carré, le backtracking se poursuit en revenant à l'étape précédente et en essayant d'autres combinaisons de placement.

La fonction de backtracking permet de vérifier systématiquement toutes les combinaisons possibles de placement des carrés parfaits pour déterminer si une solution trouvée par la fonction"find_square_sum" est valide. Grâce à cette approche, l'algorithme peut identifier les solutions qui respectent les contraintes du problème, telles que l'unicité des tailles des carrés et le respect des dimensions de la matrice.

Ps. Veuillez noter que ça marche aussi pour des rectangles, le fonction "backtrack_paving" peux prendre n'import quelle matrice en entrais, tant que le "bouwkamp_code" est valide pour cette matrice, il trouvera la solution, il nous reste qu'a générer l'image depuis la matrice résultat.

3.7 Fonction de backtracking améliorer :

Après analyse du code précédent et une étude plus approfondi du sujet, on a eu l'idée d'une tout autre approche au problème, pour plus d'explication regarder la Complexité et analyse 4.3

```
(* Fonction pour trouver la prochaine position vide dans le tableau *)
let rec find_next_empty_position board square_size start_pos  =
let rec find_in_row row col =
      if col = Array.length board.(0) then
        find_in_row (row+1) (0)
      else if col + square_size > Array.length board.(0) then None
      else if row + square_size > Array.length board then None
      else if is_zone_clear board {x=col; y=row} {x=col + square_size - 1; y=row +
    square_size - 1} 0 then
        Some {x=col; y=row}
      else
        find_in_row row (col + max 1 board.(row).(col))
in
find_in_row start_pos.y start_pos.x;;

(*fonction de pavage iterer une seul fois*)
  let single_iteration_paving board bouwkamp_code =
    let rec try_placing_square square_sizes =
      match square_sizes with
      | [] -> Some board
      | square_size :: remaining_sizes ->
        let pos_opt = find_next_empty_position board square_size {x=0; y=0} in
        match pos_opt with
        | None -> None
        | Some pos ->
            place_square board square_size pos;
            let result = try_placing_square remaining_sizes in
            (match result with
            | None -> remove_square board square_size pos; None
            | _ -> result)
    in
    try_placing_square bouwkamp_code;;
```

```
(*fonction de pavage qui test tout les permutation d'un bouwkamp_code*)
let rec backtrack_paving_ameliorer board bouwkamp_code_permutations =
  match bouwkamp_code_permutations with
  | [] -> None
  | bouwkamp_code :: remaining_permutations ->
    let solution_board_opt = single_iteration_paving board bouwkamp_code in
    match solution_board_opt with
    | Some solution_board -> Some solution_board
    | None -> backtrack_paving_ameliorer board remaining_permutations;;
```

1. find_next_empty_position : Cette version de la fonction est différent de la précédente, elle retourne que la position la plus haute et la plus à droite disponible seulement s'il y a de la place pour le carré, sinon elle ne retourne rien, ce qui fait arrêter la fonction suivante.

2. single_iteration_paving : Cette fonction prend une de la permutation du "bouwkamp_code" et test de paver dans l'ordre fourni, si le carre est pavée, on retourne la matrice et on fait arrêter la fonction suivante, sinon on ne retourne rien et ont continu.

3. backtrack_paving_ameliorer : Cette fois-ci la fonction principale faite en sort de test toutes les permutations de "bouwkamp_code" fourni jusqu'à retour d'une solution.

À première vue, c'est changement en l'air de rendre le traitement plus compliquer, mais en réalité cette implémentation est plus rapide que la précédente, néanmoins elle fait face à des problèmes sur un autre niveau.

4 Complexité et analyse

Dans cette section, nous allons analyser la complexité temporelle et spatiale des fonctions "find_square_sum" et "backtrack_paving". Nous expliquerons notre raisonnement de manière simple, tout en utilisant des expressions mathématiques pour illustrer nos arguments.

4.1 Complexité de find_square_sum :

4.1.1 Complexité temporelle

La complexité temporelle de la fonction "find_square_sum" dépend du nombre de branches explorées dans l'arbre de recherche. À chaque étape de la fonction, nous avons deux choix : inclure le premier carré de la liste ("x") ou ne pas l'inclure. Cela signifie que, pour chaque élément de la liste des carrés, nous avons deux branches possibles, ce qui entraîne un arbre binaire de recherche.

La profondeur maximale de cet arbre est de n, qui est la taille de la liste des carrés. Par conséquent, le nombre total de nœuds dans l'arbre est de l'ordre de 2^n, et la complexité temporelle de la fonction est $O(2^n)$.

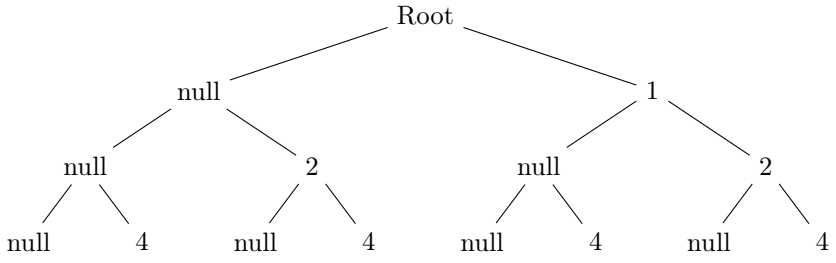

FIGURE 1 – arbre de recherche pour n = 3

taille du coté	temps d'execution (s)
1	0,000000
10	0,000000
20	0,000998
30	0,004000
40	0,091080
50	1,216840
60	14,505640
70	145,550646
80	2106,694942

TABLE 1 – table des temps d'execution

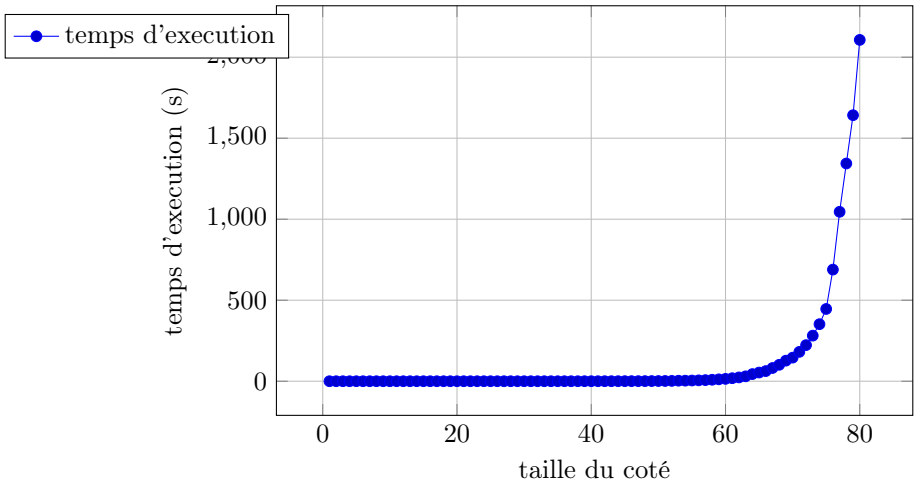

FIGURE 2 – temps d'exécution en fonction de la taille du carré

4.1.2 Complexité spatiale

La complexité spatiale de "find_square_sum" est principalement due à la mémoire utilisée par la récursion et le stockage des solutions intermédiaires et finales. Dans ce cas, la profondeur maximale de la récursion est égale à la taille de la liste "squares"squares, qui est "n".

À chaque niveau de récursion, nous avons besoin de stocker les informations suivantes :

— La liste "solutions" qui contient les solutions partielles trouvées jusqu'à présent.
— La liste "xs" qui contient les carrés restants à explorer et sa taille maximale est "n".

Pour la liste "solutions", elle est créée à chaque appel récursif et stocke les combinaisons de carrés dont la somme est égale à "remaining". Le pire cas se produit lorsque toutes les combinaisons possibles sont des "solutions" valides. Dans ce cas, la taille maximale de solutions serait de l'ordre de $O(2^n)$ (où chaque élément est inclus ou exclu).

Pour la liste "xs", elle est partagée entre les appels récursifs et n'est pas modifiée, donc nous pouvons considérer que la mémoire requise pour cette liste est constante.

En somme, la complexité spatiale de "find_square_sum" est dominée par la mémoire requise pour stocker les solutions intermédiaires et finales à chaque niveau de récursion. Ainsi, la complexité spatiale est de l'ordre de $O(2^n)$.
on remarque aussi que le nombre de résultat valid suit une courbe exceptionnel.

taille du coté	nombre de resultat
1	1
10	3
20	55
30	847
40	9398
50	91021
60	855369
70	7596889
80	62065278

TABLE 2 – table des nombre de resultat

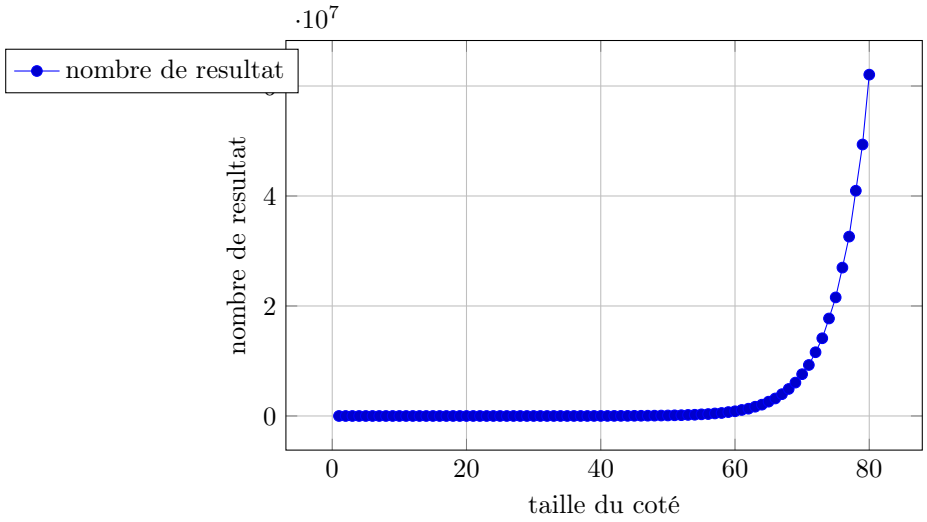

FIGURE 3 – nombre de resultat en fonction de la taille du carré

4.2 Complexité de backtrack_paving :

4.2.1 Complexité temporelle

La complexité temporelle de "backtrack_paving" dépend fortement de l'ordre des carrés dans "bouwkamp_code" Si les carrés sont triés de manière décroissante (du plus grand au plus petit), alors l'algorithme de backtracking est susceptible de trouver une solution plus rapidement, car il explore d'abord les positions pour les plus grands carrés et réduit ainsi l'espace de recherche pour les petits carrés.

Dans le meilleur des cas, si la liste "bouwkamp_code" est triée de manière optimale, la solution pourrait être trouvée dès la première tentative, ce qui signifie que l'algorithme ne devra pas revenir en arrière. Dans ce cas, la complexité temporelle serait proportionnelle à la taille de la liste "bouwkamp_code", soit $O(k)$, où "k" est le nombre de carrés dans la liste.

Cependant, dans le pire des cas, surtout si tout les carré sons de taille 1, l'algorithme de backtracking pourrait être amené à explorer toutes les combinaisons possibles de positions pour les carrés. La complexité temporelle serait alors exponentielle en fonction du nombre de carrés dans la liste "bouwkamp_code", soit $O((m * n)^k)$ où "n" et "m" sont les dimensions de la grille.

En pratique, la complexité temporelle réelle de "backtrack_paving" dépendra de l'ordre des carrés dans "bouwkamp_code", ainsi que de la disposition de la grille et de la nature des solutions. Dans certains cas, la solution pourrait être trouvée rapidement, tandis que dans d'autres, l'algorithme pourrait prendre beaucoup plus de temps.

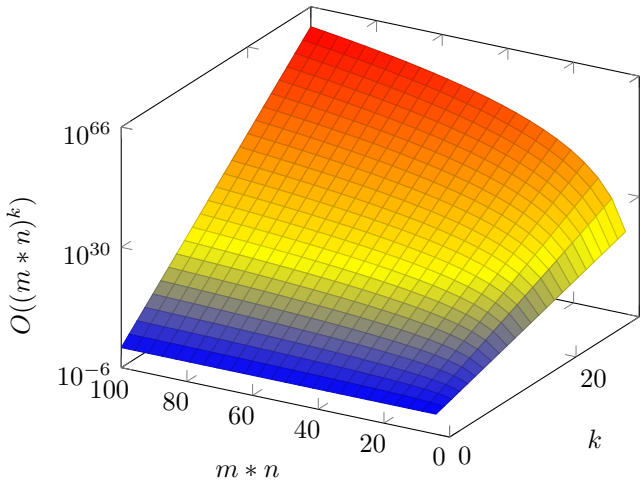

FIGURE 4 – plot 3D de $z = y^x$ ou $y = m * n$ et $x = k$.

4.2.2 Complexité spatiale

La complexité spatiale de "backtrack_paving" est principalement due à la mémoire utilisée par la récursion. Dans ce cas, la profondeur maximale de la récursion est égale à la taille de "bouwkamp_code", qui est "k". À chaque niveau de récursion, nous avons besoin de stocker les informations suivantes :

- La matrice "board" de taille $n * m$.
- La liste "remaining_sizes" qui est une liste de carrés restants à placer et sa taille maximale est k.
- La structure "pos" qui contient deux entiers

Pour la matrice "board", elle est référencée et modifiée en place, donc elle n'est pas dupliquée à chaque appel récursif. Par conséquent, la mémoire requise pour la matrice "board" est constante et égale à $n * m$.

Pour la liste "remaining_sizes", la taille maximale est k. Cependant, puisque cette liste est partagée entre les appels récursifs et n'est pas modifiée, nous pouvons considérer que la mémoire requise pour cette liste est constante.

Enfin, la structure "pos" est créée à chaque appel récursif et nécessite un espace de mémoire pour stocker deux entiers. La mémoire requise pour "pos" est proportionnelle à la profondeur de la récursion.

En somme, la complexité spatiale de "backtrack_paving" est dominée par la mémoire requise pour stocker la matrice "board" et la structure "pos" à chaque niveau de récursion. Ainsi, la complexité spatiale est de l'ordre de $O(n * m + 2 * k) = O(n * m + k)$.

4.3 backtrack_paving_ameliorer :

4.3.1 Complexité temporelle

Vu qu'on sait qu'une combinaison optimale existe avec la Complexité $O(k)$, , pour quoi pas tester tous les combinaison d'un "bouwkamp_code", si on fait le calcule, pour une liste de taille "k", on a textcolorgreen"$k!$" permutation possible, on sait qu'il y 4 solution optimale (graphiquement, c'est la même, mais retourner de 90 degré) et chacune de, c'est solution optimale a besoin d'appliquer le backtrack une seule fois, alors pas besoin de tester les autres dispositions, ce qui donne $O(k)$ pour la partie backtrack. Ainsi, se "backtrack_paving_ameliorer" a un complexité temporelle de $O(k * k!)$.

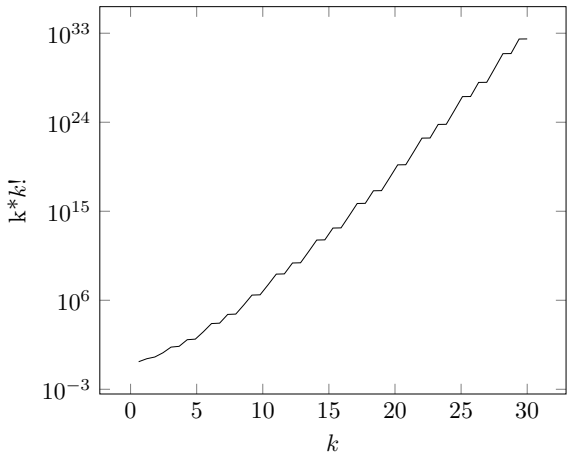

FIGURE 5 – plot de la fonction factoriel.

En comparent la figure 4 et 5 ont constat bel et bien une amélioration au niveau du temps, ou le temps n'est plus dépendant de la taille du carré, mais dépendant uniquement de la taille de la liste "bouwkamp_code" ,

4.3.2 Complexité spatiale

Pour la complexité temporelle, on a constaté une amélioration, mais c'est le contraire pour la Complexité spatiale, là, on doit stocker tous la permutation à essayer donc $O(k * k!)$, et stocker la matrice de taille m*n, Ainsi, la complexité spatiale est de l'ordre de $O(n * m + k * k!) = O(k * k!)$.
ce qui est une complexité pire, et qui va nous causer du problème a la suit principalement des problèmes de débordement d'entier et de taille de liste.

le problème de la taille de liste peut être résolu facilement, en ne testent qu'un nombre de permutations qui tien dans la mémoire à la fois :

```
(*fonction pour contourner le probleme de memoire*)
let find_solution_in_batches board bouwkamp_code batch_size =
  let k = List.length bouwkamp_code in
  let total_permutations = factorial k in
  let start_idx = ref 0 in
  let found_solution = ref None in
  while !found_solution = None && !start_idx < total_permutations do
    let end_idx = min (!start_idx + batch_size - 1) (total_permutations - 1) in
    let perms = perm bouwkamp_code !start_idx end_idx in
    let res = backtrack_paving_ameliorer board perms in
    let test_permutations () =
    if res <> None then (
      found_solution := res;
      raise Exit
    )in
    (try test_permutations () with Exit -> ());

    start_idx := end_idx + 1;
  done;

  (!found_solution);;
```

5 Implémentation

Dans cette section, nous discutons des détails pertinents de notre implémentation, tels que les points difficiles et les points compliqués.

5.1 Difficultés rencontrées

5.1.1 Stack overflow pour find_square_sum

Étant donné que cette fonction utilise la récursion pour générer toutes les combinaisons possibles de carrés parfaits dont la somme est égale à "n", elle peut provoquer un débordement de pile lorsque le nombre de solutions est énorme. Une approche pour résoudre ce problème pourrait consister à utiliser l'itération plutôt que la récursion ou à appliquer la récursion terminale, mais meme ca ne resout pas le problem a cause du nombre immense de solution, regarder table des nombre de resultat.

5.1.2 Inconsistance du temps d'exécution de backtrack_paving

Le temps d'exécution de cette fonction dépend fortement de l'ordre des carrés dans "bouwkamp_code" et de la disposition de la grille. Si les carrés sont ordonnés de manière optimale (par exemple, du plus grand au plus petit, ou du Plus élevé à gauche, plus bas à droite.), l'algorithme peut trouver une solution rapidement. Sinon, il peut prendre beaucoup plus de temps pour explorer toutes les combinaisons possibles.

5.1.3 débordement d'entier

L'un des défis majeurs rencontrés au cours du développement de notre algorithme de pavage a été le problème de débordement d'entier. Lorsque nous avons travaillé avec de grands nombres, en particulier lors du calcul des factorielles et des permutations, nous avons découvert que les entiers 64 bits n'étaient pas suffisants pour représenter les valeurs obtenues. Les entiers dépassaient la capacité maximale de stockage et débordaient, entraînant un retour aux valeurs négatives.

Ce problème de débordement d'entier a entraîné des erreurs de calcul et a rendu difficile la manipulation des grandes quantités de permutations et de combinaisons nécessaires pour résoudre le problème du pavage.

5.2 Optimisations et améliorations possibles

5.2.1 Enumération exhaustive et sélective de décompositions entières

Plutôt que de générer toutes les combinaisons possibles de carrés parfaits, on peut utiliser une approche d'énumération exhaustive et sélective pour générer uniquement les décompositions entières qui correspondent à des solutions valides. Cela peut réduire considérablement le nombre de solutions à explorer et améliorer les performances de l'algorithme.

pour plus de détails Je vous recommande la thèse de monsieur Gambini (1999) sur le sujet

5.2.2 Utiliser la programmation dynamique pour find_square_sum

Bien que l'utilisation de la programmation dynamique puisse ne pas être très utile dans notre cas en raison du critère d'unicité des carrés, elle pourrait être appliquée pour améliorer l'efficacité de "find_square_sum" si nécessaire. Cette approche permettrait de stocker les résultats intermédiaires et d'éviter de recalculer les mêmes solutions à plusieurs reprises.

5.2.3 Amélioration de la gestion de la mémoire

La gestion de la mémoire pourrait être améliorée pour réduire la consommation d'espace, par exemple en utilisant des structures de données plus efficaces ou en libérant la mémoire inutilisée plus rapidement.

voici quelleque proposition :

— Réduire l'allocation de mémoire : Essayez de minimiser les allocations de mémoire en réutilisant les structures de données et en évitant les opérations coûteuses, telles que les concaténations de listes. Utilisez des fonctions de traitement des listes telles que "List.map", "List.fold_left", et "List.fold_right", qui sont optimisées pour minimiser les allocations.

— Optimiser la taille des structures de données : Pour les tableaux et les autres structures de données, essayez d'utiliser des types de données plus petits, tels que "int32" ou "int64" au lieu de "int", pour économiser de la mémoire. Cela peut également réduire la fragmentation de la mémoire.

— Gérer manuellement la mémoire : Dans certaines situations, il peut être nécessaire de gérer manuellement la mémoire en utilisant des fonctions de bas niveau telles que "Gc.alloc", "Gc.free" et "Gc.compact". Toutefois, cette approche doit être utilisée avec prudence, car elle peut entraîner des problèmes de sécurité et de stabilité si elle est mal gérée.

5.2.4 l'augmentation de la capacité de calcul

nous avons exploré l'utilisation de bibliothèques de mathématiques spécifiques, telles que la bibliothèque Zarith pour OCaml, qui permettent de travailler avec des entiers de taille arbitraire. L'utilisation de cette bibliothèque nous permettra d'éviter les problèmes de débordement d'entier et d'améliorer la précision et la fiabilité de nos algorithmes.

Il est important de noter que la gestion des entiers de taille arbitraire peut entraîner une légère diminution des performances en raison de la complexité accrue de la manipulation de ces nombres. Cependant, dans notre cas, les avantages en termes de précision et de fiabilité l'emportent sur les inconvénients liés à la performance. En reconnaissant et en résolvant ce problème de débordement d'entier, nous pourrons créer un algorithme de pavage plus robuste et fiable pour traiter un large éventail de scénarios.

En résumé, notre implémentation présente quelques défis, tels que le débordement de pile dans "find_square_sum" et l'inconsistance du temps d'exécution de "backtrack_paving". Cependant, en appliquant des optimisations et des améliorations appropriées, il est possible d'améliorer les performances de l'algorithme et de résoudre ces problèmes. L'utilisation d'un "bouwkamp_code" optimal, l'énumération exhaustive et sélective de décompositions entières, l'amélioration de la gestion de la mémoire et l'application de la programmation dynamique sont autant de stratégies qui pourraient être envisagées pour améliorer l'efficacité de notre implémentation.

6 Résultats

Dans cette partie du rapport, nous présenterons les résultats visuels obtenus à partir de notre implémentation du problème de pavage de carrés parfaits. Nous montrerons les images des pavages générés par notre algorithme de backtracking pour différentes valeurs de "n".

pour les résultat proposer on teste avec des bouwkamp_code optimale, mais normalement ça peut prendre des mois pour trouver une solution pour les carré de grand taille, pour plus de précision veuillez vous référencee a Gambini (1999), Bouwkamp (1947) ,Bouwkamp et al. (1960) ,Bouwkamp et al. (1964), ou visitez www.squaring.net .

FIGURE 6 – carré 33 X 32 avec $bouwkamp\_code = [18; 15; 14; 10; 9; 8; 7; 4; 1]$

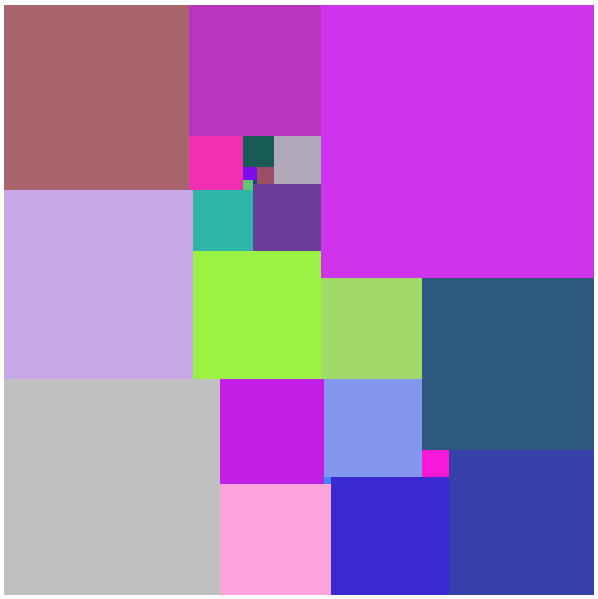

FIGURE 7 – carré 175 X 175 avec $bouwkamp\_code = [55; 39; 81; 16; 9; 14; 4; 5; 3; 1; 20; 56; 18; 38; 30; 51; 64; 31; 29; 8; 43; 2; 35; 33]$

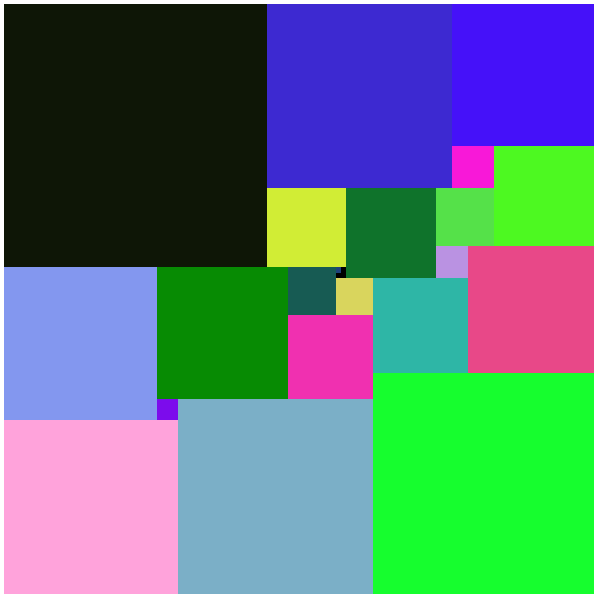

FIGURE 8 – carré 112 X 112 avec $bouwkamp\_code = [50; 35; 27; 8; 19; 15; 17; 11; 6; 24; 29; 25; 9; 1; 7; 18; 16; 42; 4; 37; 33]$

7 Conclusion

Dans cette étude, nous avons exploré le problème du pavage de carrés parfaits dans des rectangles et des carrés. Nous avons présenté et analysé les algorithmes "find_square_sum" et "backtrack_paving", qui nous ont permis de générer des solutions de pavage pour différentes valeurs de n. Nous avons également discuté des complexités temporelle et spatiale de ces algorithmes, ainsi que des défis rencontrés lors de leur implémentation, tels que les problèmes de débordement de pile et l'inconsistance du temps d'exécution.

Nous avons également abordé des pistes pour améliorer la performance de notre implémentation, notamment en utilisant l'énumération exhaustive et sélective de décompositions entières, en optimisant la gestion de la mémoire en OCaml et en recherchant des "bouwkamp_codes" optimaux. Les résultats obtenus ont été présentés sous forme de graphiques et d'images pour illustrer les pavages générés.

Enfin, il est intéressant de noter que le problème du pavage peut également être étendu à d'autres formes géométriques, telles que les triangles. Le pavage de triangles pourrait être une nouvelle direction de recherche passionnante, où des méthodes similaires pourraient être adaptées pour résoudre des problèmes de pavage impliquant des triangles et d'autres polygones. Cette extension pourrait permettre d'élargir notre compréhension des problèmes de pavage et d'explorer de nouvelles approches pour résoudre des problèmes géométriques complexes.

Références

Bouwkamp, C. (1947). On the dissection of rectangles into squares.

Bouwkamp, C., Duijvestijn, A., and Haubich, J. (1964). Catalogue of simple perfect squared rectangles of order 9 through 18.

Bouwkamp, C., Duijvestijn, A., and Medema (1960). Catalogue of simple squared rectangles of order nine through fourteen and their elements.

Gambini, I. (1999). Quant aux carrés carrelés.