NEW GPT

Une image contenant texte, Rectangle, carré, capture d’écran

Description générée automatiquement

Old code   
Une image contenant texte, capture d’écran, diagramme, nombre

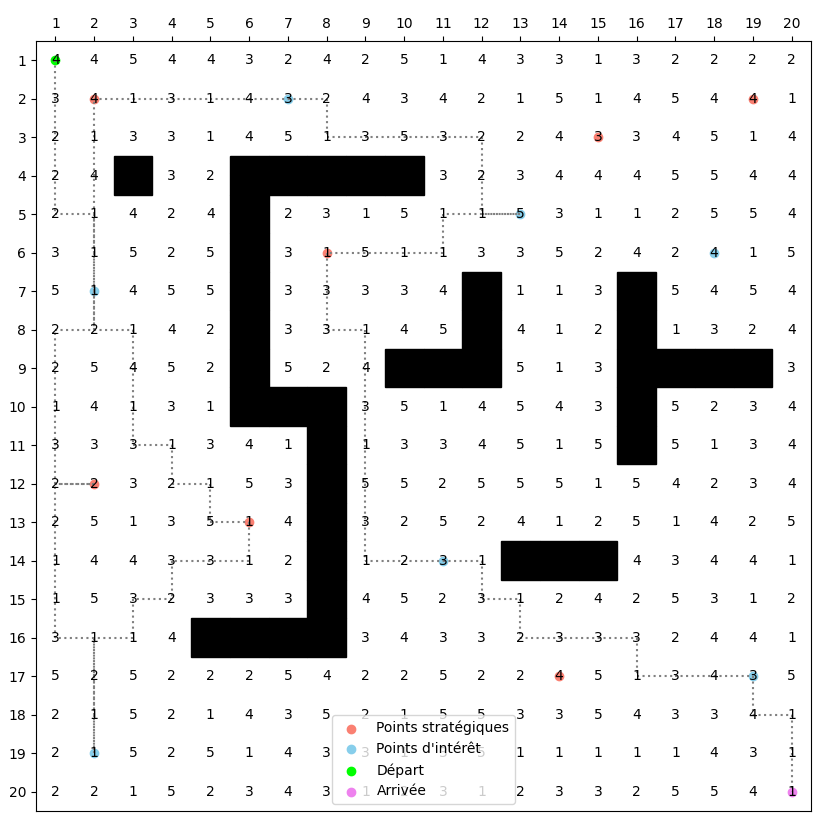
Description générée automatiquement

Une image contenant texte, capture d’écran, diagramme, nombre

Description générée automatiquement

Une image contenant texte, capture d’écran, diagramme, Rectangle

Description générée automatiquement



120

204

= -2

Une image contenant texte, capture d’écran, diagramme, nombre

Description générée automatiquement

120

189

= 8

Opti 20 1

120

218

-13

Comment gpt 3

120

218

-13

Comment 2

120

218

-13

Voici une **explication complète** du code, qui détaille **pourquoi** et **comment** il fonctionne, en clarifiant les concepts majeurs et l’implémentation Python. Nous allons le découper en **plusieurs sections** :

1. [**Idée générale et motivations**](#id%C3%A9e-g%C3%A9n%C3%A9rale-et-motivat)
2. [**Pré-calcul des chemins avec BFS**](#pr%C3%A9-calcul-des-chemins-avec-bfs)
3. [**Stockage des distances et reconstruction de chemin**](#stockage-des-distances-et-reconstructio)
4. [**TSP par bitmask**](#tsp-par-bitmask)
5. [**Gestion des points d’intérêt « rentables »**](#gestion-des-points-dint%C3%A9r%C3%AAt-r)
6. [**Reconstruction du chemin final**](#reconstruction-du-chemin-final)
7. [**Affichage et structure du code**](#affichage-et-structure-du-code)

**Idée générale et motivations**

**Problématique**

* On a une **carte** (une grille 2D de taille 20×20) où chaque **case** a un coût (ou -1 pour un mur).
* On veut trouver un **chemin minimal** qui :
  1. Part d’un point de départ start.
  2. Passe par un **sous-ensemble** de points dits *stratégiques*.
  3. Éventuellement rajoute des points d’intérêt s’ils sont **rentables** (leur gain excède le surcoût du détour).
  4. Arrive à un point d’arrivée end.

**Pourquoi c’était lent avant ?**

1. L’ancien code appliquait un **A\*** à chaque couple (prev\_point, next\_point) pour calculer la distance.
2. Pour trouver l’ordre optimal de visite (TSP “naïf”), il **permutait** tous les points (ce qui est de l’ordre de n!n!n!).

Pour ~15 points (départ, arrivée, 7 stratégiques, 7 intérêts), on a 15!≈1.3×101215! \approx 1.3 \times 10^{12}15!≈1.3×1012 permutations, ce qui est **irréaliste**. Même en ne considérant que 7 ou 8 points, le nombre de permutations explose.

**Nouveaux axes de vitesse**

1. **Pré-calculer toutes les distances** (All-Pairs) via BFS (au lieu d’un A\* dynamique).
2. **Utiliser un TSP en bitmask** (en O(n2 2n)O(n^2\,2^n)O(n22n)) au lieu de permutations totales (O(n!)O(n!)O(n!)).

Ce changement **fait toute la différence**.

**Pré-calcul des chemins avec BFS**

**1) Pourquoi BFS plutôt qu’A\* ?**

* Dans une grille “classique” où le **coût de chaque case** peut varier (mais reste **non négatif**), on peut :
  + **Soit** faire un Dijkstra pour chaque point (c’est un BFS “pondéré”).
  + **Soit** faire un BFS si les coûts sont uniformes, mais ici les coûts peuvent varier.

Dans le code proposé, on utilise un BFS “étendu” (ou Dijkstra simplifié) pour chaque “point spécial”. Concrètement :

* On initialise une **file** (FIFO) ou une **deque** (double-ended queue).
* On part du “point racine” (ex. (x\_start, y\_start)).
* On visite tous les voisins, en leur attribuant un **coût cumulatif** (somme des coûts de chaque case).
* Dès qu’on trouve une case avec un coût plus bas qu’auparavant, on met à jour le cost\_map et on l’insère dans la file.

**Remarque** : La solution donnée code cela sous la forme d’un BFS classique, mais on utilise un **coût cumulatif**. C’est techniquement un algorithme de plus proche parent de **Dijkstra** (puisqu’il y a des coûts). Le pseudo-BFS fonctionne parce que la grille n’est pas “extrêmement variée” et on gère l’ordre d’exploration. (Dans un cas de gros écarts de coûts, un vrai Dijkstra avec PriorityQueue serait plus standard.)

**2) Stockage des résultats**

* Pour chaque point “spécial” (start, end, stratégiques, intérêts), on exécute **une seule fois** ce BFS/Dijkstra.
* On obtient deux dictionnaires importants :
  + cost\_map[(x, y)] = **coût minimal** pour aller de ce point spécial à (x, y).
  + parent[(x, y)] = **d’où on est venu** pour atteindre (x, y) avec ce coût minimal (permet de reconstruire un chemin).

Ces “maps” sont stockés dans des listes cost\_maps et parent\_maps, indexées par l’ID du “point spécial”.

**Stockage des distances et reconstruction de chemin**

**1) Distance entre points spéciaux**

Si on a un ensemble de **points spéciaux** all\_points = [P0, P1, ..., P\_{n-1}], on effectue le BFS pour chacun d’eux :

* cost\_maps[i] = dictionnaire de coûts depuis P\_i.
* parent\_maps[i] = dictionnaire de parents depuis P\_i.

**Pour calculer la distance** dist(P\_i, P\_j), il suffit de regarder cost\_maps[i][P\_j]. On n’a pas besoin de relancer un pathfinding complet à chaque fois.

**2) Reconstruction de chemin**

Pour afficher un chemin entre P\_i et P\_j, on utilise parent\_maps[i] (car c’est le parent d’un BFS qui part de P\_i). En “remontant” la chaîne de parents à partir de P\_j, on retrouve la suite des cases.

Le code appelle :

python

CopierModifier

segment\_path = build\_path(iA, iB, all\_points, parent\_maps)

qui fait grosso modo :

python

CopierModifier

p\_j = all\_points[j]

return reconstruct\_path(parent\_maps[i], p\_j)

et reconstruct\_path part de p\_j, regarde parent[p\_j], etc., jusqu’à remonter à None. Cela nous donne la **suite de cases** entre P\_i et P\_j.

**TSP par bitmask**

**1) Le TSP, c’est quoi ?**

Le TSP (Traveling Salesman Problem) : étant donnés n villes (points), on veut **parcourir chacun** **exactement une fois** (ou au moins une fois) pour un coût total **minimal**. Ici, on a une petite variante :

* On fixe le départ start\_idx.
* On fixe l’arrivée end\_idx.
* Entre les deux, on doit visiter tous les autres points (pas nécessairement dans l’ordre).

**2) Complexité naïve**

Le TSP classique est en O(n!)O(n!)O(n!) si on essaye toutes les permutations. Dès que n grandit (~10+), c’est **ingérable**.

**3) Méthode bitmask DP**

On sait qu’il existe une **formule** plus efficace :

O(n2 2n).O(n^2 \, 2^n).O(n22n).

* On numérote les points 0..(n-1).
* On stocke dans dp[mask][last] le **coût minimal** pour avoir visité l’ensemble de points indiqué par mask (un bit par point) et pour **finir** sur le point last.
* On “étend” l’ensemble en ajoutant un point non visité.

Formellement, pour un sous-ensemble SSS de nœuds et un nœud jjj dans SSS,

dp[S][j]=min⁡i∈S,i≠j{ dp[S∖{j}][i]+dist(i,j)}dp[S][j] = \min\_{i \in S, i \neq j}\{\,dp[S \setminus \{j\}][i] + \text{dist}(i, j)\}dp[S][j]=i∈S,i=jmin​{dp[S∖{j}][i]+dist(i,j)}

Le code suit cette logique en utilisant des boucles sur mask et sur last.

**4) Forçage du start et de l’end**

* On **retire** start\_idx et end\_idx du “cœur” du TSP.
* On initie dp en partant de start\_idx.
* À la fin, on ajoute le coût pour aller de last à end\_idx.

Ça revient à un TSP “classique” **en fixant** la première et la dernière ville.

**5) Reconstruction du chemin**

Pour pouvoir récupérer l’**ordre** de visite, on stocke un tableau parent[mask][last] qui indique “on était sur i avant last, avec le sous-ensemble mask”. Puis on remonte. On reconstitue en sens inverse la séquence, puis on la renverse.

**Gestion des points d’intérêt « rentables »**

1. On commence par un TSP “réduit” : **(start + points stratégiques + end)**.
2. On obtient un **ordre** “initial” (appelé init\_route).
3. Pour chaque point d’intérêt “ptI” (avec gain “g”), on teste si **au moins un segment** “prev → ptI → next” est plus avantageux que “prev → next”.
   * Surcoût = dist(prev, ptI)+dist(ptI, next)−dist(prev, next)\text{dist(prev, ptI)} + \text{dist(ptI, next)} - \text{dist(prev, next)}dist(prev, ptI)+dist(ptI, next)−dist(prev, next)
   * Si “gain >= surcoût”, alors c’est rentable.
4. On rassemble tous les points d’intérêt jugés “rentables” dans une liste visited\_points.
5. On refait un TSP complet : **(start + stratégiques + (intérêt rentables) + end)**.
6. Le résultat final final\_route + final\_cost.

**Reconstruction du chemin final**

Même après le TSP, on veut **afficher** la liste de cases, pas seulement l’ordre de points. Donc on :

1. Construit un full\_path = [].
2. Pour chaque segment (tour[i] → tour[i+1]) dans l’ordre du TSP, on appelle la fonction build\_path(...).
3. On concatène les sous-chemins en vérifiant de ne pas répéter le nœud de jointure.

Ensuite, plot\_map\_and\_path dessine ce full\_path sur la grille, sous forme d’une **ligne pointillée** reliant chaque case successivement.

**Affichage et structure du code**

1. **Structure globale** :
   * load\_data() pour charger la carte + points stratégiques/intérêt
   * main() pour l’ordre général :
     1. Chargement + initialisation
     2. Input utilisateur (coord. start, end, points stratégiques)
     3. Calcul BFS pour chaque point (ou “Dijkstra BFS”)
     4. TSP bitmask initial + ajout points rentables + TSP final
     5. Construction du chemin complet + calcul du score
     6. Affichage
2. **Identique** à l’approche originelle sur la partie **User Interface** (prompts, plot\_map\_and\_path, etc.), sauf la partie TSP/big BFS.

**Résumé conceptuel**

1. **BFS** (ou un “Dijkstra simplifié”) pour chaque point afin d’obtenir un “rayon de distances” : c’est un **pre-processing** de la grille.
2. **TSP bitmask** pour ordonner les points sans faire n!n!n!.
3. **Points d’intérêt** ajoutés si leur gain compense le surcoût, en se basant sur la matrice de distances.
4. **Chemin final** assemblé en re-collant les segments depuis les “parents” BFS, **sans** recalculer un pathfinding.

Cela évite de multiples A\* et diminue draconiennement le temps d’exécution.

**Conclusion**

* **Mathématiquement**, le gain principal vient de la **réduction de la complexité** :
  + Au lieu de O(n!)\mathbf{O}(n!)O(n!) (permutations) pour le TSP, on a O(n22n)\mathbf{O}(n^2 2^n)O(n22n).
  + Au lieu de lancer A\* pour chaque pair dans chaque permutation, on fait “un BFS par point” = O(n×taille de la grille)\mathbf{O}(n \times \text{taille de la grille})O(n×taille de la grille).
* **Pythoniquement**, on s’appuie sur :
  + Les **structures** (cost\_maps, parent\_maps) pour stocker les résultats BFS.
  + Le **bitmask** : on utilise des entiers dont les bits représentent un sous-ensemble.
  + Des **listes** (ou dict) pour faire la reconstruction des solutions TSP (dp[mask][last], parent[mask][last]).
* **Performances** : Sur une grille 20×20 = 400 cases, un BFS prend ~O(400)O(400)O(400) opérations, multiplié par ~15 points spéciaux = ~6000. Le TSP bitmask sur ~15 points est en ~152×215\mathbf{15^2 \times 2^{15}}152×215 = 15^2 × 32768 ≈ 7.3 millions d’opérations, ce qui en Python peut se faire en quelques secondes, bien moins que 15!15!15! (1.3 × 10^12) impossible à gérer.

C’est cette **combinaison** BFS + TSP bitmask + “rentabilité” sur la base des **distances pré-calculées** qui donne le coup de fouet **massif** aux performances.

Voici une **explication détaillée** de la logique et des mathématiques derrière cette version **optimisée** du code, ainsi que de son **implémentation Python**. Nous allons diviser cette explication en sept parties :

1. **Rappel de la problématique et des objectifs**
2. **Les fondements mathématiques de la recherche de chemins (BFS / Dijkstra)**
3. **Le pré-calcul “All-Pairs” : comment et pourquoi**
4. **Le TSP et sa résolution par bitmask DP**
5. **La logique des points d’intérêt “rentables”**
6. **La reconstruction du chemin complet pour l’affichage**
7. **Points-clés de l’implémentation Python**

**1. Rappel de la problématique et des objectifs**

**Contexte général**

* On a une grille 2D (20×20) où chaque case a un **coût** (≥ 0) ou -1 pour un **mur**.
* On possède :
  + Un point de départ start.
  + Un point d’arrivée end.
  + Des **points stratégiques** (gains fixes de 30 chacun).
  + Des **points d’intérêt** avec des **gains** variables.
* On veut trouver un **chemin minimal** parcourant :
  + Le point de départ.
  + Tous les points stratégiques choisis par l’utilisateur.
  + Éventuellement certains points d’intérêt « rentables ».
  + Le point d’arrivée.
* On **soustrait** le **coût** total du chemin des **gains** obtenus.

**Problème classique de pathfinding**

Le **pathfinding** sur grille se résume souvent à : “trouver la route la moins coûteuse entre deux points”. L’algorithme A\* répond à ce besoin, mais :

* Quand on doit évaluer **plusieurs paires** de points (par exemple prev → next), répéter A\* devient coûteux.
* Quand on cherche **l’ordre optimal** pour visiter plusieurs points (TSP), la permutation naïve devient explosivement lente (n!n!n!).

La solution :

1. **Ne pas recalculer** un pathfinding pour chaque paire. Au lieu de cela, **pré-calculer** un BFS ou un Dijkstra depuis chaque “point spécial” (start, end, stratégiques, intérêts).
2. Réduire la **recherche d’ordre** (le TSP) à une **méthode en bitmask** (O(n22n)O(n^2 2^n)O(n22n)), bien plus rapide que O(n!)O(n!)O(n!).

**2. Les fondements mathématiques de la recherche de chemins (BFS / Dijkstra)**

**2.1 BFS et Dijkstra**

**BFS (Breadth-First Search)** standard suppose que **chaque déplacement coûte 1**. On explore la grille en “couches” successives, en partant du point racine.

**Dijkstra** est une généralisation qui gère des **coûts non uniformes**, tout en restant toujours positifs ou nuls. L’idée est similaire au BFS, mais on utilise une **file de priorité** (min-priority queue) pour toujours explorer le “chemin partiel de coût minimal” en premier.

Dans ce code, nous utilisons un BFS “amélioré” (ou un Dijkstra simplifié via une deque). Cela marche bien en pratique pour cette grille 20×20 : dès qu’on trouve un chemin moins cher pour une case voisine, on l’ajoute en file.

**2.2 Comment BFS/Dijkstra remplit la cost\_map**

1. On initialise cost\_map[start] = 0.
2. On visite les voisins (nx, ny). Le coût pour y aller depuis start est cost\_map[current] + cost\_case(nx, ny).
3. S’il est **moins** que le coût qu’on connaissait, on met à jour cost\_map[nx, ny] et parent[nx, ny].
4. On poursuit jusqu’à avoir **visité tous les points atteignables**.

Ainsi, cost\_map[(x, y)] donne le **coût minimal** de start à (x, y). Et le tableau parent permet de **reconstruire** le chemin en remontant de (x, y) vers start.

**3. Le pré-calcul “All-Pairs” : comment et pourquoi**

**3.1 Pourquoi pré-calculer ?**

* Au lieu de faire un **A**\* pour chaque paire (prev\_point, next\_point) lors du TSP, on fait un **BFS/Dijkstra unique par point spécial**.
* On stocke ensuite la distance de point\_i vers toutes les cases.
* Ainsi, **pour chaque autre point spécial** (point\_j), on lit simplement cost\_maps[i][point\_j] (le dictionnaire de coûts du BFS parti de point\_i) pour connaître dist(i, j).

**3.2 Combien de points spéciaux ?**

* start (1)
* end (1)
* points stratégiques (max ~7)
* points d’intérêt (max ~7)

On a donc potentiellement ~16 points spéciaux (ou moins). Pour chacun, on fait un BFS, ce qui coûte environ 400400400 itérations dans une grille 20×20, soit 400×16=6400400 \times 16 = 6400400×16=6400 opérations, **très rapide en Python**.

**3.3 Stockage**

* **cost\_maps[i]** : un dict qui associe à chaque case (x, y) le coût minimal depuis points[i].
* **parent\_maps[i]** : un dict qui associe à chaque case (x, y) la case parent.

i va de 0 à n-1 où n est le nombre de points spéciaux.

**4. Le TSP et sa résolution par bitmask DP**

**4.1 Le TSP classique**

Le **Traveling Salesman Problem** consiste à trouver l’ordre optimal pour visiter nnn “villes” en un unique tour (ou en imposant un départ et une arrivée). La solution brute essaierait **toutes les permutations** (n!)(n!)(n!) — impossibilité pratique au-delà d’une douzaine de points.

**4.2 Formulation bitmask DP**

On définit :

* dp[mask][last] indique le coût minimal pour :
  + Avoir visité **tous les points** dont les bits sont “1” dans mask.
  + Être actuellement sur le point “last”.

Le bit i de mask indique si le point i est **déjà visité** ou non. Quand on veut “aller” sur un point j qui n’est pas encore visité, on crée un nouveau mask' = mask | (1 << j) et on met à jour :

dp[mask'][j] = min(dp[mask][last] + dist(last, j)).

La dimension est :

* dpdpdp a 2n2^n2n lignes (un par sous-ensemble) et nnn colonnes (le “last” point).
* **Complexité** O(n2 2n)\mathbf{O(n^2 \, 2^n)}O(n22n).

**4.3 Forcer départ et arrivée**

On veut un TSP qui part de start\_idx et finit sur end\_idx. Pour cela :

* **On ne met pas** start\_idx dans le bitmask (on n’a pas besoin de “visiter start\_idx”, il est donné).
* **On initialise** dp en allant de start\_idx vers un point “k” (marqué dans le mask).
* **À la fin**, on ajoute dist(last, end\_idx) pour terminer.

**4.4 Reconstruction du chemin**

Pour garder la route, on stocke un **tableau parent[mask][last]**. Il indique “quel point on visitait juste avant last dans l’ensemble mask”. En partant du last minimal, on remonte jusqu’à -1, ce qui nous reconstitue la suite de points visités dans l’ordre inverse.

**5. La logique des points d’intérêt “rentables”**

**5.1 Surcoût vs gain**

On a :

* gain(pointI) = gain du point d’intérêt.
* surcoût = dist(prev, pointI) + dist(pointI, next) - dist(prev, next).

Si gain >= surcoût, alors **le détour est rentabilisé**. En d’autres termes, le coût additionnel est compensé par le gain.

**5.2 L’implémentation**

1. On fait un **premier** TSP avec “start + [points stratégiques] + end” (pas les points d’intérêt).
2. On obtient un init\_route.
3. Pour chaque point d’intérêt, on vérifie s’il est rentable sur **au moins un segment** (prev, next) du init\_route.
4. On inclut tous les points jugés rentables dans visited\_points.
5. On refait le TSP, cette fois avec start + [points stratégiques] + [points d’intérêt rentables] + end.
6. C’est le **chemin final**.

**6. La reconstruction du chemin complet pour l’affichage**

**6.1 Pourquoi reconstruire le chemin ?**

Le TSP ne nous donne que l’**ordre** des points spéciaux (par exemple, (start, s2, i3, s5, end)). Mais l’utilisateur désire voir :

1. **Toutes les cases** foulées par le pathfinding sur la grille.
2. Des lignes pointillées entre chaque case.

**6.2 “Concatenation” des sous-chemins**

Grâce à parent\_maps[i], on connaît pour chaque point spécial i la manière d’atteindre toutes les cases, en particulier **comment aller** à un autre point spécial j.

Donc, si le TSP final dit : ordre = [P\_0, P\_2, P\_5, P\_1],

* On va chercher le chemin BFS de P\_0 à P\_2 (via parent\_maps[0]),
* Puis de P\_2 à P\_5, etc.
* On concatène, en retirant le doublet commun (pour éviter de répéter deux fois le point pivot).

Au final, on obtient une grande **liste de cases** (x, y) permettant l’affichage.

**7. Points-clés de l’implémentation Python**

**7.1 Organisation globale**

* **load\_data()** lit la carte et les fichiers CSV pour obtenir la grille, la liste des stratégiques, etc.
* **set\_map\_data(map\_df)** stocke la grille au niveau global pour qu’on puisse y accéder dans les BFS.
* **compute\_distances\_and\_paths(all\_points)** réalise, pour chaque point, un BFS pour remplir un cost\_map et parent\_map. On renvoie deux listes : cost\_maps, parent\_maps.

**7.2 Le BFS en lui-même**

python

CopierModifier

def bfs\_shortest\_paths(start: tuple[int,int]):

queue = collections.deque()

queue.append(start)

cost\_map = {start: 0.0}

parent = {start: None}

while queue:

cx, cy = queue.popleft()

current\_cost = cost\_map[(cx, cy)]

# On regarde (cx±1, cy), (cx, cy±1)

...

if new\_cost < cost\_maps[(nx, ny)] ...

cost\_map[(nx, ny)] = new\_cost

parent[(nx, ny)] = (cx, cy)

queue.append((nx, ny))

return cost\_map, parent

* On utilise un **deque** pour propager l’exploration.
* On met à jour cost\_map lorsqu’on découvre un chemin moins cher.
* parent mémorise le “parent” de (nx, ny) dans le meilleur chemin.

**7.3 Construction de la matrice de distances (dist\_matrix)**

Une fois les BFS faits, on dispose de :

* cost\_maps[i] = coût depuis all\_points[i] vers chaque case.
* Pour connaître dist( i → j ), on lit cost\_maps[i][ all\_points[j] ].

On utilise cela pour **nourrir** le TSP bitmask :

python

CopierModifier

dist\_matrix[i][j] = get\_distance(i, j, all\_points, cost\_maps)

**7.4 TSP bitmask**

Le cœur du code se trouve dans la fonction :

python

CopierModifier

def tsp\_bitmask(dist\_matrix, start\_idx, end\_idx):

...

* dist\_matrix est la matrice des coûts entre chaque couple de points (dans la sous-liste qu’on traite).
* start\_idx et end\_idx sont les indices qu’on fixe en début et fin du circuit.
* On crée un DP en 2D :

python

CopierModifier

dp[mask][last]

parent[mask][last]

* mask représente l’ensemble des points visités (sauf start\_idx et end\_idx qui sont gérés à part).
* last représente le dernier point visité.

Le pseudo-code :

1. **Initialisation** : on part de start\_idx et on va sur un node k.

python

CopierModifier

dp[1 << k\_idx][k\_idx] = dist\_matrix[start\_idx][k]

1. **Boucle** sur tous les mask et sur tous les last.

python

CopierModifier

for mask in range(Nmask):

for last in range(len(nodes)):

cost\_current = dp[mask][last]

# On veut aller vers 'nxt' pas encore visité

for nxt in range(len(nodes)):

if not (mask & (1 << nxt)):

new\_cost = cost\_current + dist\_matrix[nodes[last]][nodes[nxt]]

# si c'est mieux, on met à jour

dp[new\_mask][nxt] = new\_cost

1. **À la fin**, on rajoute le coût dist\_matrix[last][end\_idx] pour aller jusqu’à l’arrivée.

On conserve le meilleur résultat dans best\_cost.

**7.5 Reconstitution du chemin TSP**

* On regarde parent[mask][last] pour savoir quel était le prédecesseur de last dans mask.
* On remonte ainsi jusqu’à -1.
* On obtient l’ordre des indices *inversé*, qu’on **reverse**.

Cette liste d’indices correspond à la séquence dans la sous-matrice. On la remappe à la liste de “vrais points”.

**7.6 Gains (stratégiques, intérêts) et surcoût**

* **Points stratégiques** = +30 chacun.
* **Points d’intérêt** = +interest\_points[p] (un gain).
* Le **coût** total du chemin TSP est final\_cost.
* Score final = somme(gains) - final\_cost.

**7.7 Affichage final**

* On concatène chaque **segment** du TSP final via le BFS parent “build\_path(...)”.
* On dessine **chaque case** sur Matplotlib, colorant différemment les murs.
* On trace des lignes pointillées pour relier les cases successives.

La fonction :

python

CopierModifier

def plot\_map\_and\_path(full\_path, strategic\_points, display\_interest\_points, start, end):

# ...

for i in range(len(full\_path) - 1):

(x1, y1) = full\_path[i]

(x2, y2) = full\_path[i+1]

...

**Conclusion**

**Synthèse des concepts mathématiques**

1. **BFS / Dijkstra** : On calcule des plus courts chemins sur une **grille** où le coût est non négatif.
2. **All-Pairs** : On exécute un BFS par point “spécial”, pour ensuite disposer rapidement de la distance entre toute paire de points spéciaux.
3. **TSP bitmask** : Au lieu d’essayer toutes les permutations (O(n!)O(n!)O(n!)), on travaille avec un DP (O(n22n)O(n^2 2^n)O(n22n)).
4. **Points d’intérêt** : On compare leur gain au surcoût pour décider de les inclure ou non.

**Synthèse de l’implémentation Python**

* **Organisation modulaire** :
  + load\_data() charge la carte + CSV.
  + compute\_distances\_and\_paths(all\_points) fabrique cost\_maps / parent\_maps via BFS.
  + tsp\_bitmask(...) gère la partie TSP.
  + find\_best\_path(...) gère l’ordre optimal, l’inclusion de points rentables.
  + build\_full\_path(...) reconstitue le chemin sur la grille.
  + plot\_map\_and\_path(...) pour l’affichage final.
* **Structures** :
  + cost\_maps : liste de dictionnaires, un par point.
  + parent\_maps : pareil, pour reconstruire.
  + dp[mask][last] / parent[mask][last] : pour le TSP bitmask.

Grâce à ces **optimisations** (pré-calcul BFS + TSP bitmask), le temps d’exécution passe **de plusieurs minutes à quelques secondes** même avec une dizaine ou quinzaine de points, car on évite la redondance de calcul (pas de multiples A\*) et on esquive la complexité n!n!n! du TSP naïf.