Compléments sur les plus courts chemins

Retour sur l'algorithme de Bellman-Ford

- À chaque itération de l'algorithme de Bellman-Ford (tel qu'il a été présenté précédemment), on teste chaque arc du graphe en parcourant chaque sommet puis, pour chacun d'eux, la liste de ses arcs sortants (autrement dit la liste de ses successeurs).
- Si, au moment de traiter le sommet i à une itération donnée, sa marque λ_i n'a pas diminué depuis son traitement à l'itération précédente il est inutile de parcourir les arcs sortant de i car aucun d'eux ne permettra d'améliorer une marque λ (ces marques ne font que diminuer au cours de l'algorithme).
- Plutôt que de parcourir (et tester) tous les arcs du graphe à chaque itération on peut se contenter de maintenir une liste des sommets dont la marque λ a diminué (mais dont les arcs sortants n'ont pas encore été testés) et de ne traiter que les sommets de cette liste, jusqu'à épuisement.
- Cette amélioration, attribuée à Yens (1970), ne modifie pas la complexité de l'algorithme dans le pire des cas mais peut apporter des gains de temps substantiels en pratique lorsque le réseau ne contient pas de circuits absorbants.

Algorithme générique pour le problème de base

Données : Un réseau orienté $R = (V, E, c)$ **sans circuits absorbants** et un sommet particulier s.

Résultat : Les longueurs λ_j des plus courts chemins depuis s ainsi que les prédécesseurs immédiats $p[j]$ dans ces chemins.

Début

(1) **Pour tout** $j \in V$ **poser** $\lambda_j := \infty$ et $p[j] :=$ `NULL`

(2) $\lambda_s := 0$

(3) $L := (s)$ // Liste des sommets à traiter

(4) **Tant que** L n'est pas vide **faire**

(5) Retirer un sommet i de L // Lequel ?

(6) **Pour chaque** successeur j de i **faire**

(7) **Si** $\lambda_j > \lambda_i + c_{ij}$ **faire** // Amélioration -> mise à jour

(8) $\lambda_j := \lambda_i + c_{ij}$

(9) $p[j] := i$

(10) **Si** $j \notin L$ **faire** // Il faudra traiter ou retraiter j

(11) Ajouter j à L // À quelle place ?

(12) // Et si j est déjà dans L, faut-il changer sa position ?

(13) Retourner les longueurs λ_j et les prédécesseurs $p[j]$

Fin

Propriétés de l'algorithme générique (1)

- **L'algorithme générique s'arrête, après un nombre fini d'itérations de la boucle (4), si et seulement si le réseau ne contient aucun circuit absorbant accessible depuis la source s.**
- Si l'algorithme s'arrête, les valeurs λ_j ($< \infty$) sont égales à la longueur d'un plus court chemin de la source s au sommet j et les marques $p[j]$ (non nulles) fournissent un prédécesseur immédiat dans un plus court chemin de s à j.
- Les performances de l'algorithme générique dépendent de la manière dont sont gérés les ajouts et les retraits de la liste L. Les trois structures de données qui viennent immédiatement à l'esprit pour gérer L sont
  - une file (liste FIFO),
  - une pile (liste LIFO),
  - une queue de priorité (liste ordonnée selon les valeurs croissantes des meilleures distances connues λ_j).

Propriétés de l'algorithme générique (2)

- Si on utilise une file (liste FIFO) pour stocker l'ensemble L des sommets à traiter, on obtient l'algorithme de Bellman-Ford (tel qu'il est présenté actuellement dans de nombreux ouvrages).

  La complexité de l'algorithme obtenu est en $O(m\,n)$ dans le pire des cas (et donc en $O(n^3)$ pour un réseau simple mais dense).

- Si on utilise une pile (liste LIFO), on obtient un algorithme qui peut nécessiter, dans le pire des cas, un nombre *exponentiel* d'itérations !

- Si on utilise une stratégie mixte, consistant à introduire un sommet j en queue de liste si sa marque λ_j subit sa première diminution (elle passe alors d'une valeur infinie à une valeur finie) et à introduire j en tête de liste sinon, on obtient un algorithme (Pape, 1974) qui peut nécessiter, dans le pire des cas, un nombre *exponentiel* d'itérations !

Propriétés de l'algorithme générique (3)

- Si on utilise une queue de priorité et que l'on retire de L, à chaque itération, le sommet i possédant la plus petite marque λ_i on obtient essentiellement l'algorithme de Dijkstra.
  - Si tous les poids sont positifs ou nuls, chaque sommet (accessible depuis s) est introduit et retiré de L une seule fois.

    L'algorithme correspond alors exactement à l'algorithme de Dijkstra et sa complexité est polynomiale (en $O(m \log n)$ avec un tas binaire et en $O(m + n \log n)$ avec un tas de Fibonacci).
  - Si le réseau contient des arcs de poids négatif (mais pas de circuit absorbant), l'algorithme obtenu reste correct mais un sommet peut être introduit (et retiré) plusieurs fois de L.

    Dans cette variation de l'algorithme de Dijkstra, la boucle (4) peut être effectuée un nombre *exponentiel* de fois dans le pire des cas !

    En pratique, si le réseau ne compte que quelques arcs de poids négatif, cette approche peut offrir une alternative très compétitive à l'utilisation de Bellman-Ford.

Algorithme générique et circuits absorbants

- Si le réseau contient un circuit absorbant (accessible depuis la source s), la liste L ne se vide jamais et l'algorithme générique ne termine pas.
- Afin de conserver un algorithme fonctionnel, qui s'arrête après un nombre fini d'itérations, deux approches principales sont envisageables :
  1. Explorer périodiquement le graphe partiel défini par les prédécesseurs $p[i]$ à la recherche d'un circuit (qui sera alors forcément à coût négatif).
  2. Gérer la liste L comme une file (liste FIFO) et simuler les itérations de l'algorithme de Bellman-Ford en utilisant un sommet sentinelle.
     - À l'initialisation on insère la source s dans L suivie de la sentinelle.
     - Lorsque le sommet retiré de L est la sentinelle, une itération vient de se terminer. Si L est vide on s'arrête (les marques ont convergé), sinon on incrémente le nombre d'itérations.

       Si ce nombre d'itérations est égal à n (le nombre de sommets) on s'arrête, le réseau contient un circuit absorbant (accessible depuis la source). Sinon on réintroduit la sentinelle en queue de liste.

Plus court chemin de *s* à *t*

On considère un réseau simple $R = (V, E, c)$ dans lequel on cherche un plus court chemin d'un sommet origine s (la source) à un sommet destination t (le puits).

- Si la pondération est non négative, on peut utiliser l'algorithme de Dijkstra depuis la source s et stopper l'exécution de la boucle principale dès que le puits t est retiré de la queue de priorité.

  Dans le pire des cas, le sommet t sera le dernier à être traité et cet arrêt prématuré de l'algorithme ne permet pas d'en améliorer la complexité asymptotique. En pratique le gain peut être très intéressant voire essentiel.

- Si les poids des arcs peuvent être négatifs, l'utilisation de l'algorithme de Bellman-Ford (ou de la version générique de l'algorithme de Dijkstra) ne permet plus l'arrêt prématuré de l'algorithme. En effet, lors de l'application de tels algorithmes un sommet peut être introduit et retiré plusieurs fois de l'ensemble des sommets à traiter. Ce n'est qu'une fois toutes les étiquettes stabilisées que l'on peut affirmer qu'elles sont bien égales aux longueurs des plus courts chemins depuis s.

Algorithme de Dijkstra bidirectionnel

On considère le problème de la recherche d'un plus court chemin de s à t dans un réseau simple $R = (V, E, c)$ muni d'une pondération non négative de ses arcs.

L'idée des variantes bidirectionnelles de l'algorithme de Dijkstra consiste à effectuer deux explorations en parallèle :

- une exploration *en avant*, sur le réseau d'origine, depuis la source s et utilisant une queue de priorité Q_f (f pour *forward*) et des étiquettes λ_j ;
- une exploration *en arrière*, sur le réseau transposé, depuis le puits t et utilisant une queue de priorité Q_r (r pour *reverse*) et des étiquettes δ_j.

En pratique on alterne entre les traitements en avant et en arrière en utilisant une règle *ad hoc* pour décider si le prochain sommet traité est retiré de Q_f ou de Q_r.

Toute règle fournit un résultat correct (mais pas forcément les mêmes performances pratiques dans un contexte donné) pour autant que le critère d'arrêt de l'algorithme soit correct.

Critère d'arrêt

- Un critère d'arrêt naturel pourrait être de stopper l'algorithme dès qu'un sommet j va être traité pour la deuxième fois (une fois dans chaque direction) et de retourner le chemin $s \rightsquigarrow j \rightsquigarrow t$ où le premier sous-chemin provient de l'exploration en avant alors que le second provient de l'exploration en arrière.

  Un tel critère n'est pas correct car il est possible que le sommet j n'appartienne pas au plus court chemin de s à t !

- Un critère d'arrêt correct est donné par la démarche suivante :
  - On tient à jour la longueur μ du meilleur chemin connu jusqu'à présent de s à t, initialement $\mu = \infty$.
  - Lors du traitement d'un arc (i, j), dans l'exploration en avant, on met à jour la valeur de μ si j a déjà été retiré de Q_r et si $\mu > \lambda_i + c_{ij} + \delta_j$.
  - On adapte le test précédent pour mettre à jour μ lors du traitement d'un arc dans l'exploration en arrière.
  - On stoppe l'algorithme et on retourne le chemin de longueur μ lorsqu'un sommet va être traité pour la deuxième fois.

Algorithme A* (1)

- Particulièrement apprécié pour la résolution de problèmes de *pathfinding* dans les jeux vidéo ainsi que pour celle de puzzles et casse-tête divers mais possédant également de nombreuses applications plus « sérieuses », l'algorithme A* (prononcé « A star ») utilise une *heuristique* pour guider l'exploration d'un graphe et accélérer la recherche d'un plus court chemin de s à t.
- Plus précisément, considérons une **heuristique optimiste h** (aussi appelée **admissible**) qui fournit, pour chaque sommet j d'un réseau, une *borne inférieure* h_j sur la longueur d'un plus court chemin de j à t.

  L'algorithme A* utilise la même structure que l'algorithme de Dijkstra (version générique avec réintroduction possible d'un sommet dans la queue de priorité) à une différence près :

  À chaque itération, le prochain sommet traité est le sommet j pour lequel la somme $\lambda_j + h_j$ est minimale.

Algorithme A* (2)

- En d'autres termes, l'algorithme A* choisit à chaque itération le sommet j offrant la meilleure estimation de la longueur d'un plus court chemin de s à t.

  Cette estimation est la somme de deux termes :

  - la longueur λ_j du meilleur chemin connu pour l'instant de s à j,
  - la borne inférieure h_j sur la longueur d'un plus court chemin de j à t.

- Plus les valeurs fournies par l'heuristique h sont proches des vraies distances jusqu'à t, meilleures seront les chances que le plus court chemin de s à t passe par le sommet j sélectionné et meilleures seront les performances de l'algorithme.

  - À une extrême, l'heuristique est un *oracle* omniscient retournant toujours la longueur d'un plus court chemin de j à t. L'algorithme A* traite alors uniquement les sommets d'un plus court chemin de s à t.
  - À l'autre extrême, l'heuristique retourne toujours 0 comme borne inférieure (on suppose les poids des arcs non négatifs) et A* n'est rien d'autre que Dijkstra.

Algorithme A* (Hart, Nilsson, Raphael, 1968)

Données : Un réseau orienté $R = (V, E, c)$ muni d'une pondération *non négative* de ses arcs (ou du moins sans circuit absorbant), deux sommets particuliers s et t et une heuristique admissible $h : V \to \mathbb{R}$.

Résultat : Le longueur λ_t d'un plus court chemin de s à t ainsi que les prédécesseurs immédiats $p[j]$ permettant de reconstruire un plus court chemin de s à t ($\lambda_t = \infty$ s'il n'existe pas de chemin de s à t dans R).

Début

(1) **Pour chaque** sommet $i \in V$ **poser** $\lambda_j := \infty$ et $p[j] :=$ `NULL`

(2) **Poser** $\lambda_s := 0$ et **calculer** h_s

(3) $L := (s)$ // Queue de priorité des sommets à traiter, basée selon les priorités $\lambda + h$

(4) **Tant que** $L \neq \varnothing$ **faire**

(5) Retirer de L le sommet i de plus petite priorité $\lambda_i + h_i$

(6) **Si** $i = t$ **sortir de la boucle** // Destination atteinte -> on s'arrête

(7) **Pour chaque** successeur $j \in Succ[i]$ **faire**

(8) **Si** $\lambda_j > \lambda_i + c_{ij}$ **faire** // Amélioration -> mise à jour des marques

(9) **Si** $\lambda_j = \infty$ **calculer** h_j // Découverte de j -> calcul de h_j

(10) **Poser** $\lambda_j := \lambda_i + c_{ij}$ et $p[j] := i$

(11) Ajouter j à L (s'il n'y est pas déjà) // Mise à jour prio. si j déjà dans L

(12) Retourner la longueur λ_t et les prédécesseurs $p[j]$

Fin

Performances et Propriétés

Soit $R = (V, E, c)$ un réseau simple sans circuit absorbant.

- L'algorithme A* avec une heuristique admissible (c.-à-d. ne surestimant jamais la distance jusqu'à la destination t) est un algorithme exact qui fournit toujours une solution optimale (c.-à-d. un plus court chemin de s à t si t est accessible).
- Cependant, même si les poids des arcs sont non négatifs, il est possible qu'un sommet doive être traité plusieurs fois (cela dépend de l'heuristique).
- Dans le pire des cas, certaines heuristiques peuvent nécessiter le traitement d'un nombre exponentiel de sommets.
- Une modification possible consiste à ne tester une amélioration (ligne (8) du pseudocode) que si le sommet j n'a pas encore été retiré de L (il est donc dans L ou alors $\lambda_j = \infty$). Chaque sommet est alors traité au plus une fois et l'algorithme obtenu nécessite au plus n itérations de la boucle principale.

  Cependant, avec une telle modification, la méthode n'est plus un algorithme exact en général mais une heuristique.

Heuristique consistante (1)

Une heuristique $h : V \to \mathbb{R}$, pour le problème du plus court chemin de s à t dans un réseau R, est dite **consistante** (ou) si $h_t = 0$ et si pour tout arc (i, j)

$$h_i \leq c_{ij} + h_j.$$

Autrement dit, une heuristique est consistante si, quel que soit l'arc (i, j) considéré, l'estimation de la distance de i à t n'est pas pire que le coût de l'arc (i, j) plus l'estimation de la distance de j à t.

Théorème. Une heuristique consistante est optimiste (mais la réciproque n'est pas toujours vraie).

DÉMONSTRATION. Soit $(i_1 = i, i_2, \ldots, i_l = t)$ la suite des sommets visités par un plus court chemin de i à t. Pour chaque arc (i_k, i_{k+1}) on a $h_{i_k} - h_{i_{k+1}} \leq c_{i_k i_{k+1}}$ car h est consistante. En sommant sur tous les arcs du plus court chemin on obtient

$$h_i = \sum_{k=1}^{l-1} \left(h_{i_k} - h_{i_{k+1}}\right) \leq \sum_{k=1}^{l-1} c_{i_k i_{k+1}} = \text{distance de } i \text{ à } t.$$

Heuristique consistante (2)

Si h est une heuristique consistante (pour le réseau $R = (V, E, c)$ et la destination t) alors l'algorithme A* traite chaque sommet au plus une fois et il est équivalent à l'algorithme de Dijkstra appliqué au réseau $R' = (V, E, c')$ muni des coûts réduits

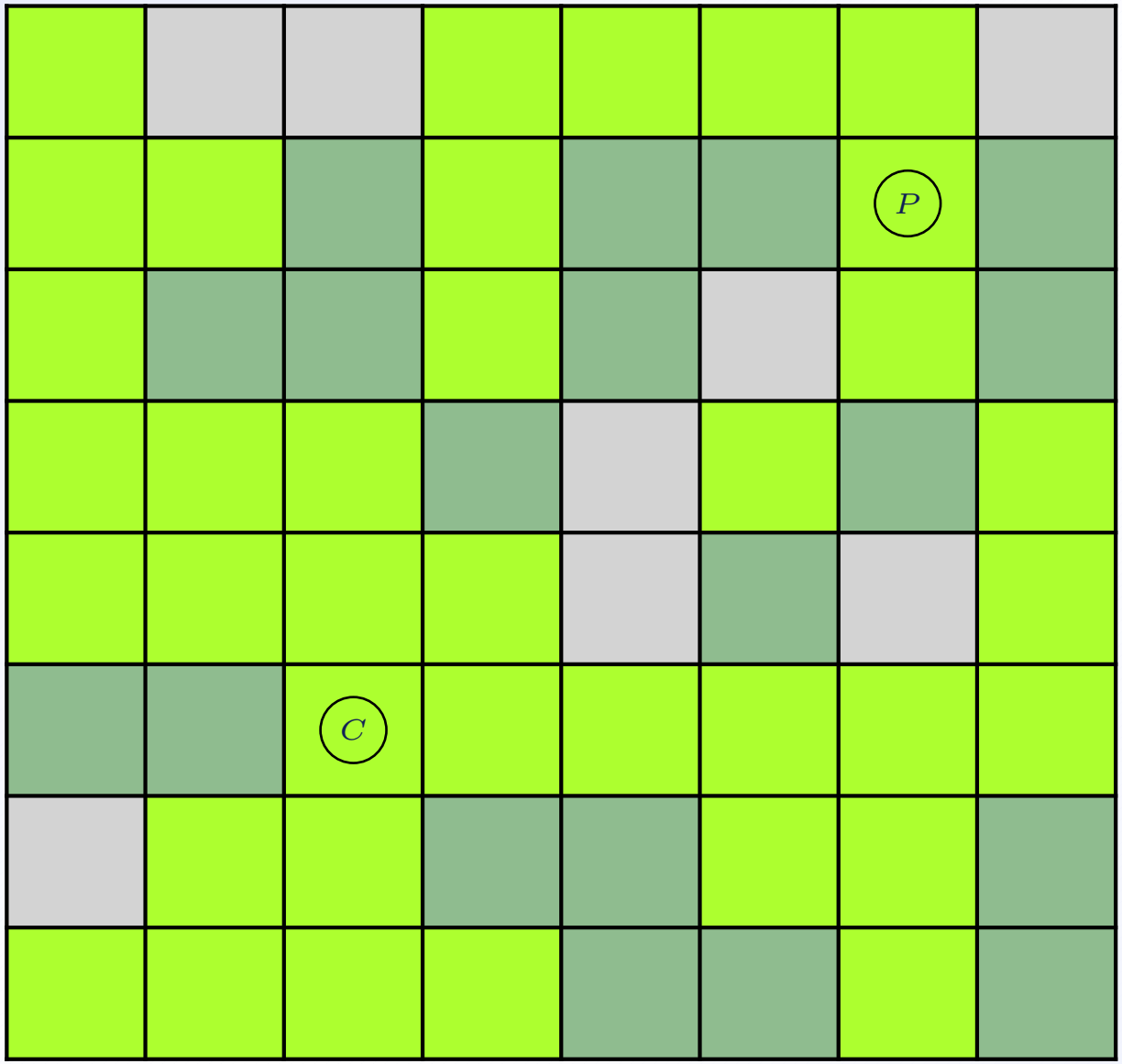
$$c'_{ij} = c_{ij} - h_i + h_j.$$

- L'heuristique h étant consistante, on a pour chaque arc (i, j)

$$h_i \leq c_{ij} + h_j \quad \Longleftrightarrow \quad c_{ij} - h_i + h_j \geq 0$$

  et les coûts réduits sont non négatifs.
- Dijkstra traite les sommets dans l'ordre croissant des distances modifiées $\lambda'_i = \lambda_i - h_s + h_i$.
- A* traite les sommets dans l'ordre croissant des priorités $\lambda_i + h_i$.
- Comme h_s est une constante, les deux algorithmes traitent les sommets dans le même ordre.

Exemple (1)



Plaine, coût = 1

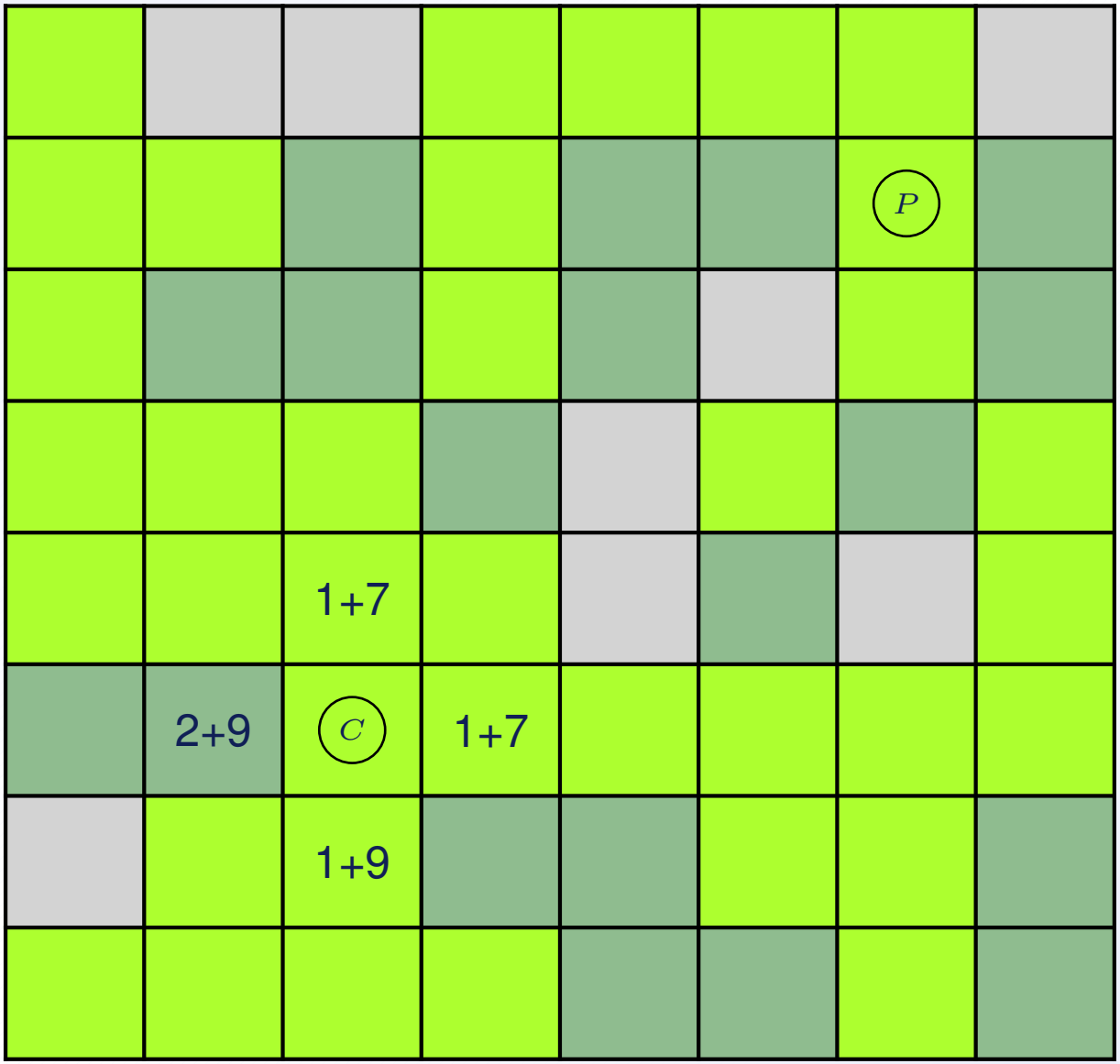
Forêt, coût = 2

Montagne, coût = 5

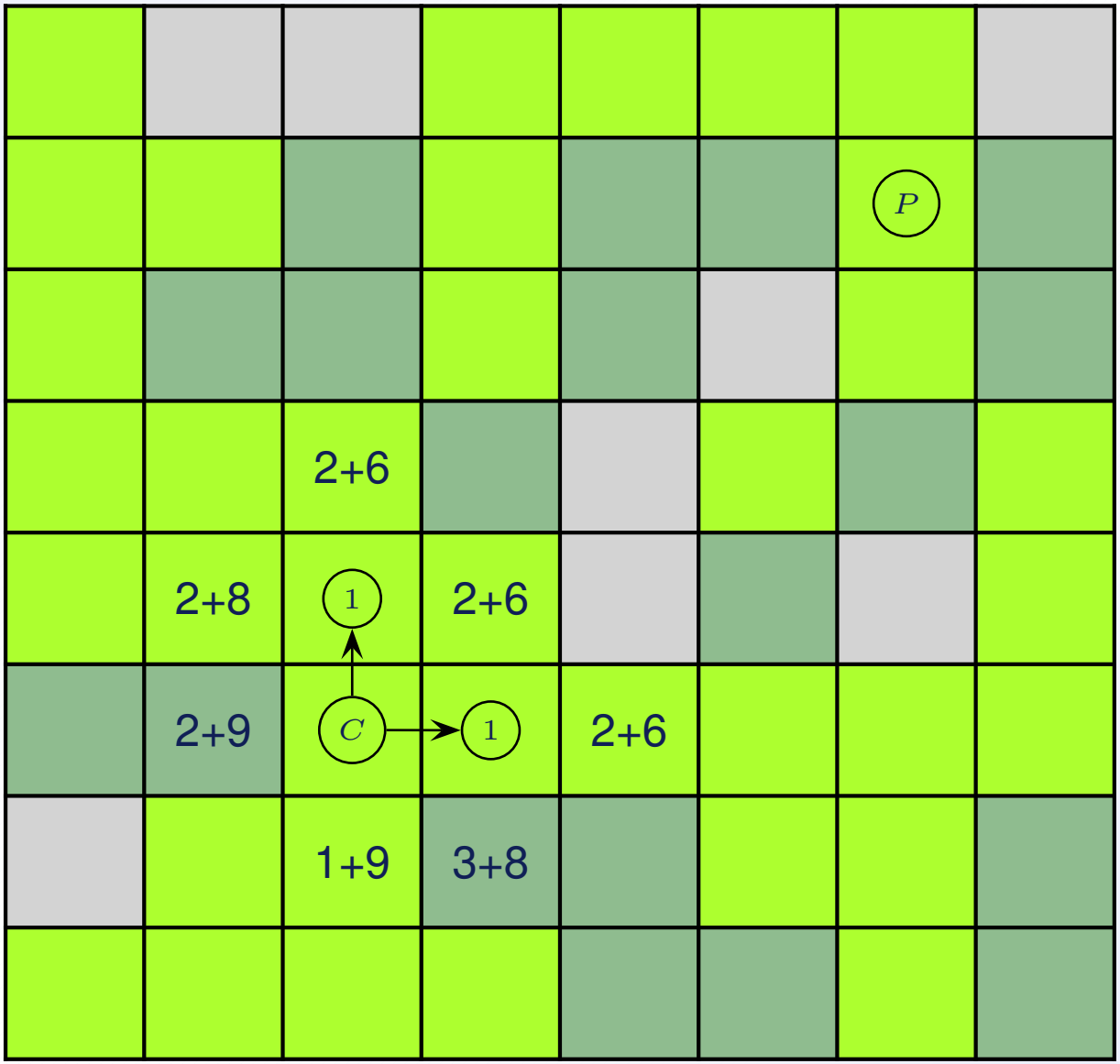
C Preux chevalier

P Princesse

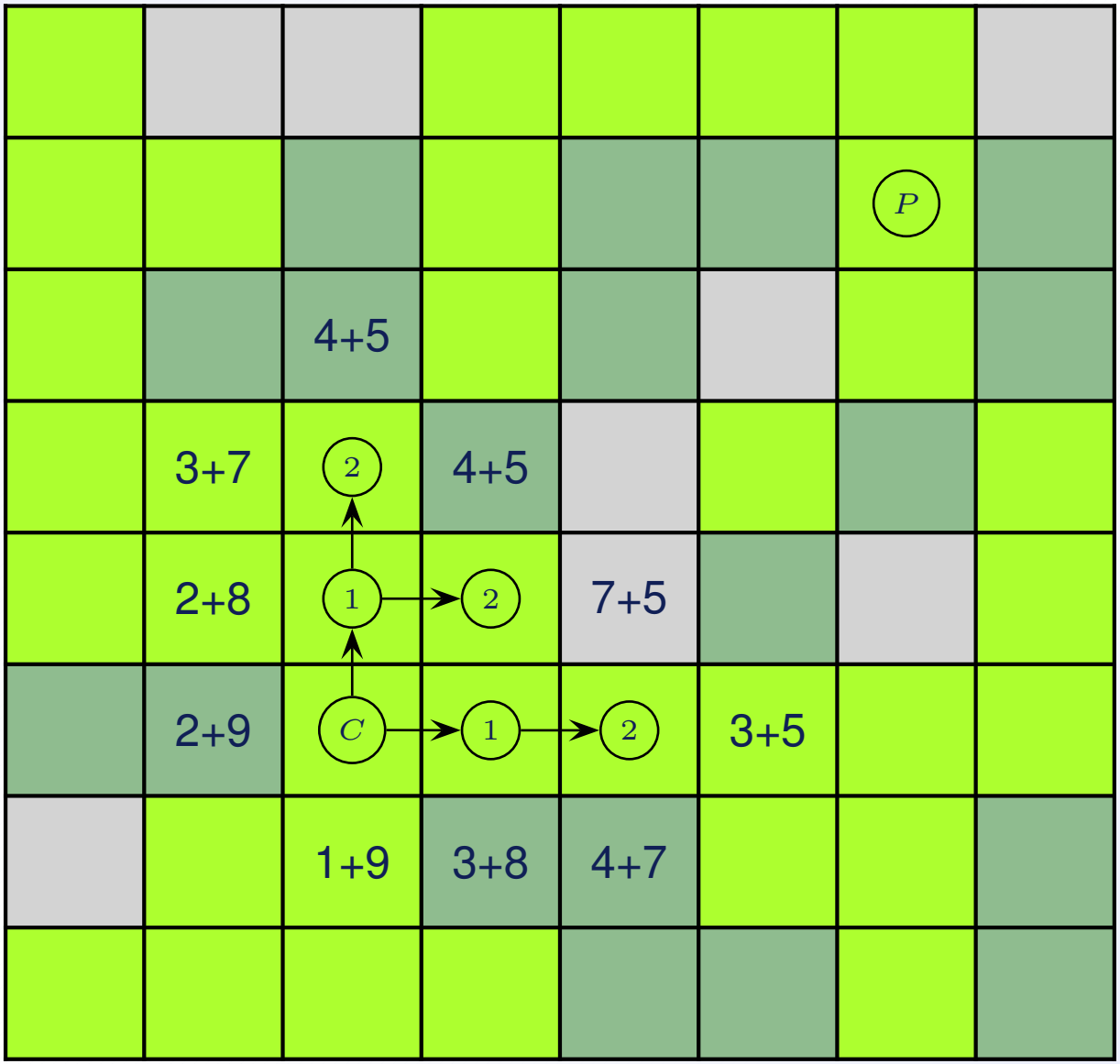
Exemple (2)



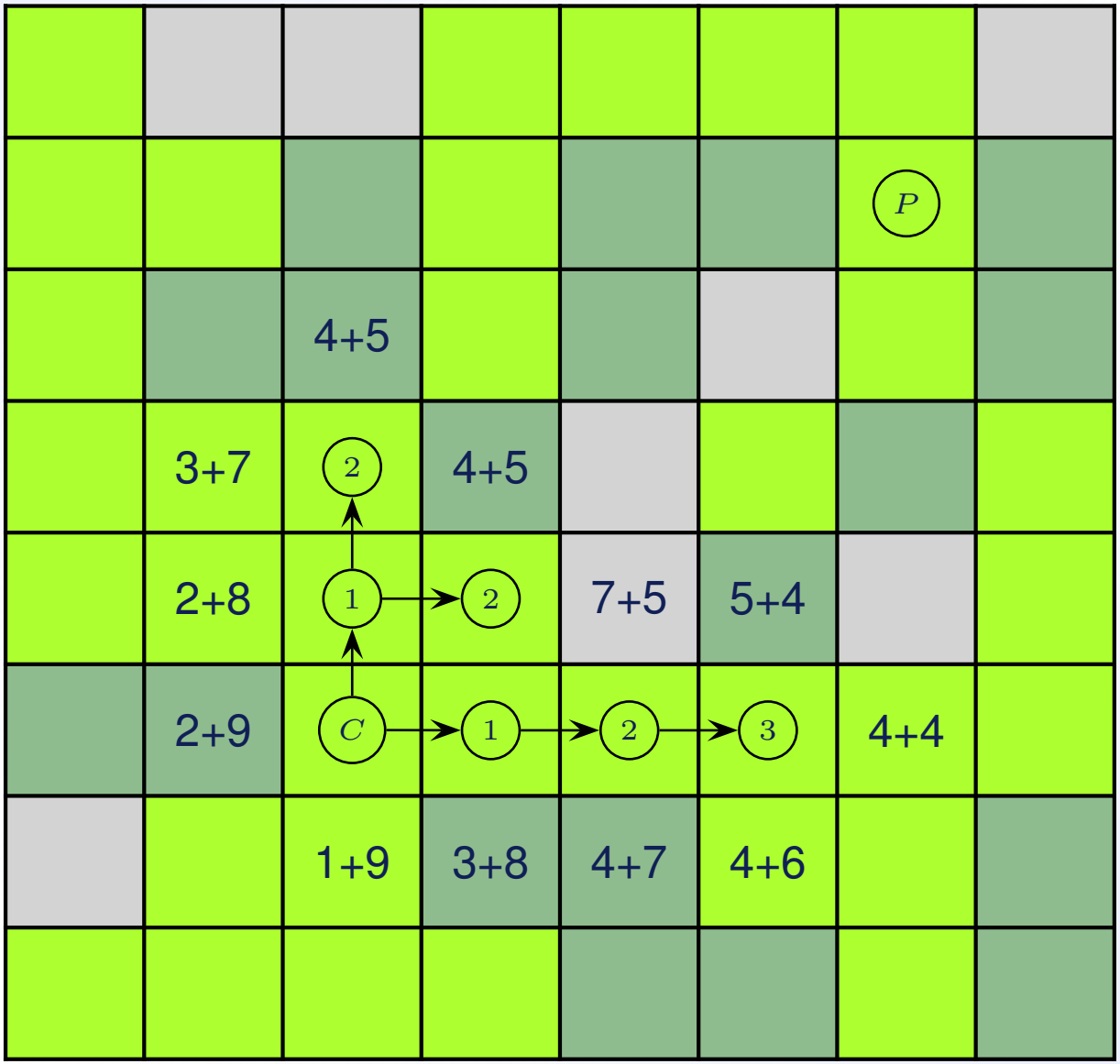
Exemple (3)



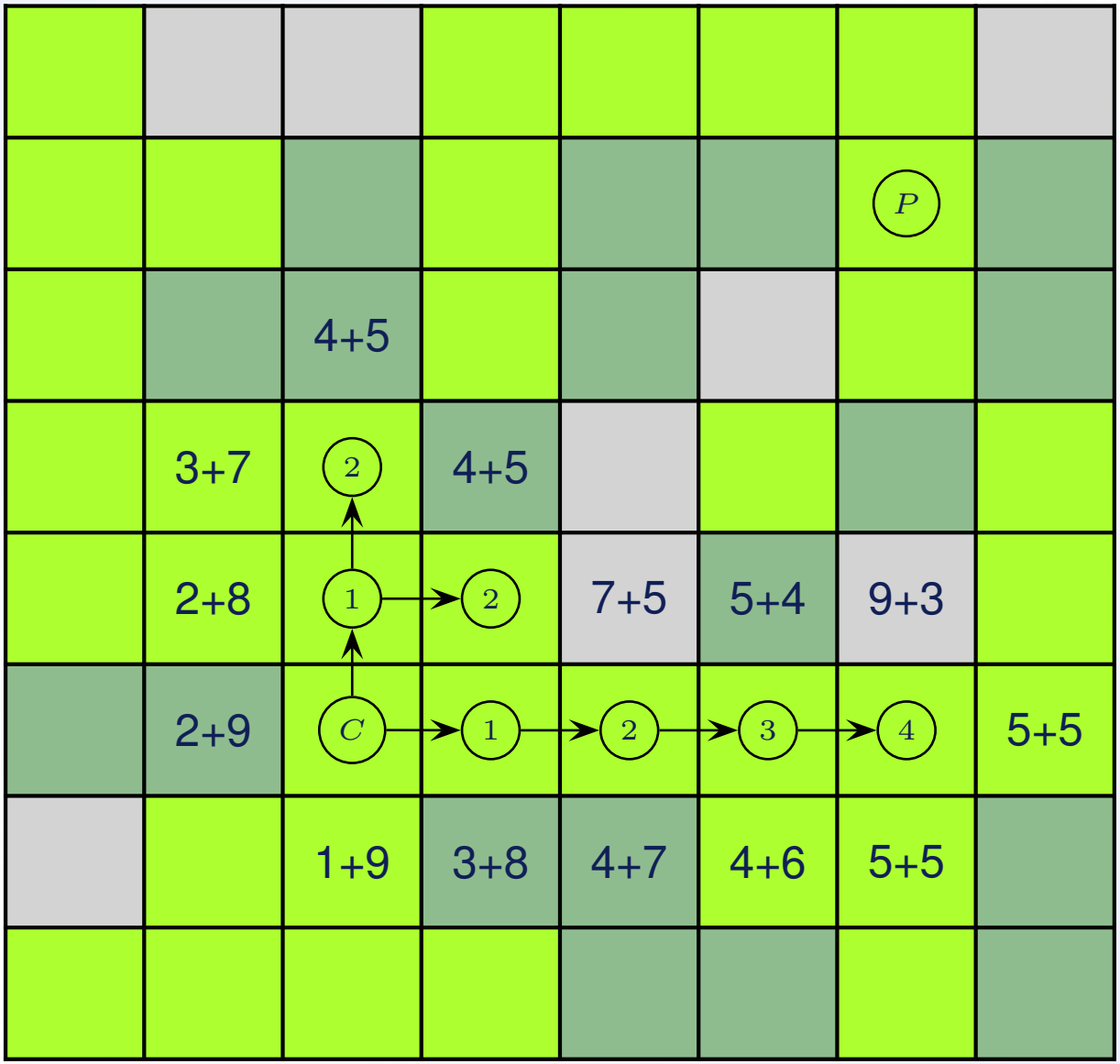
Exemple (4)



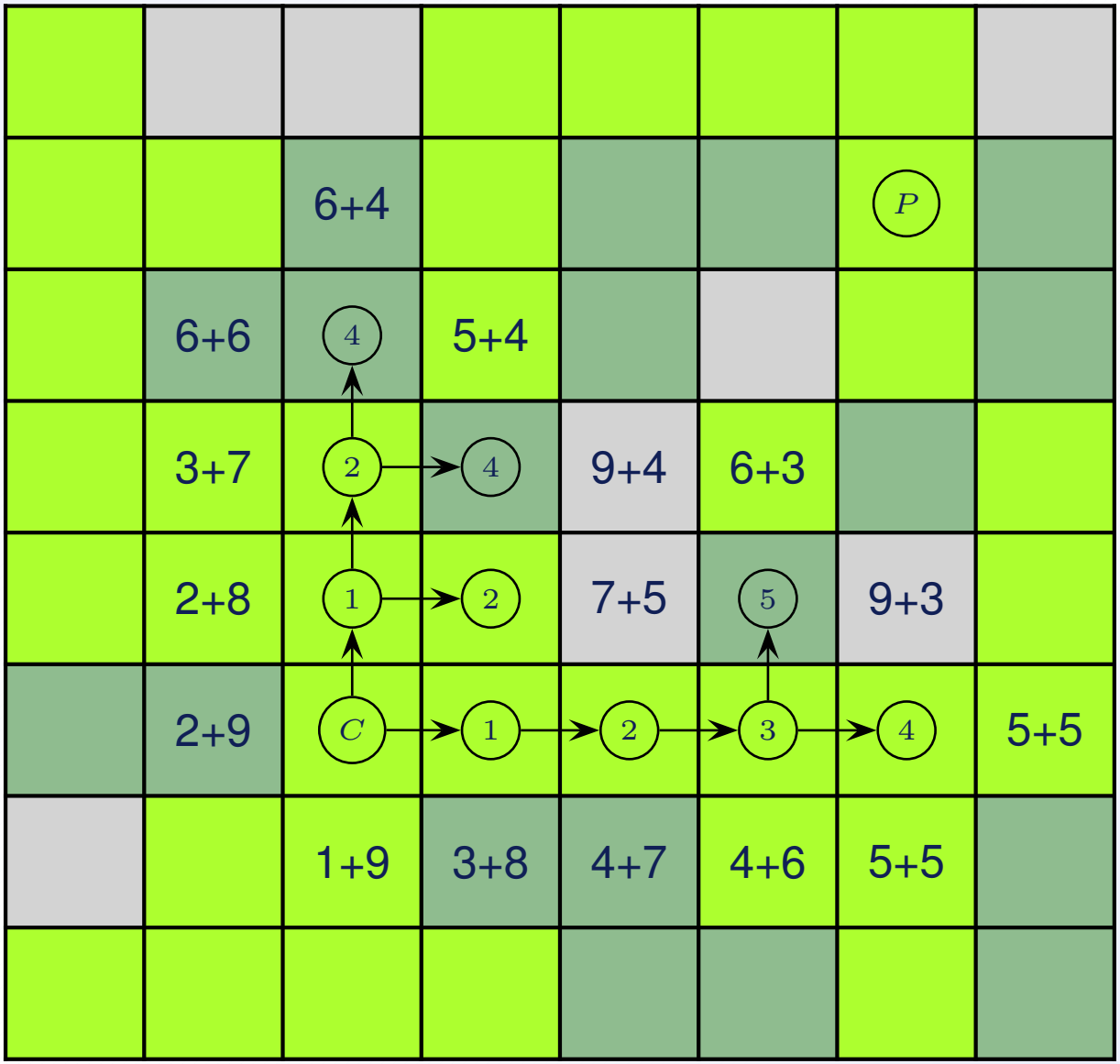
Exemple (5)



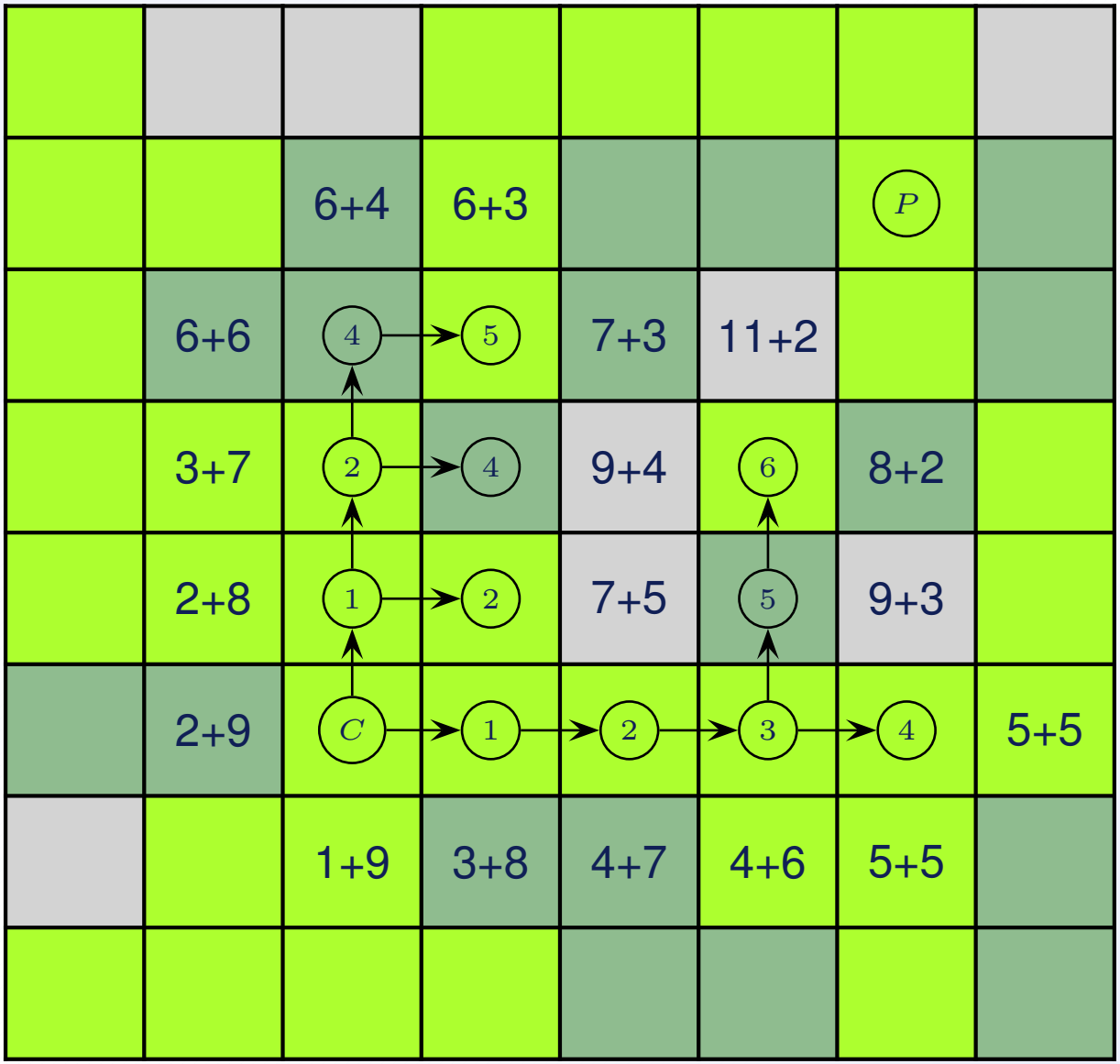
Exemple (6)



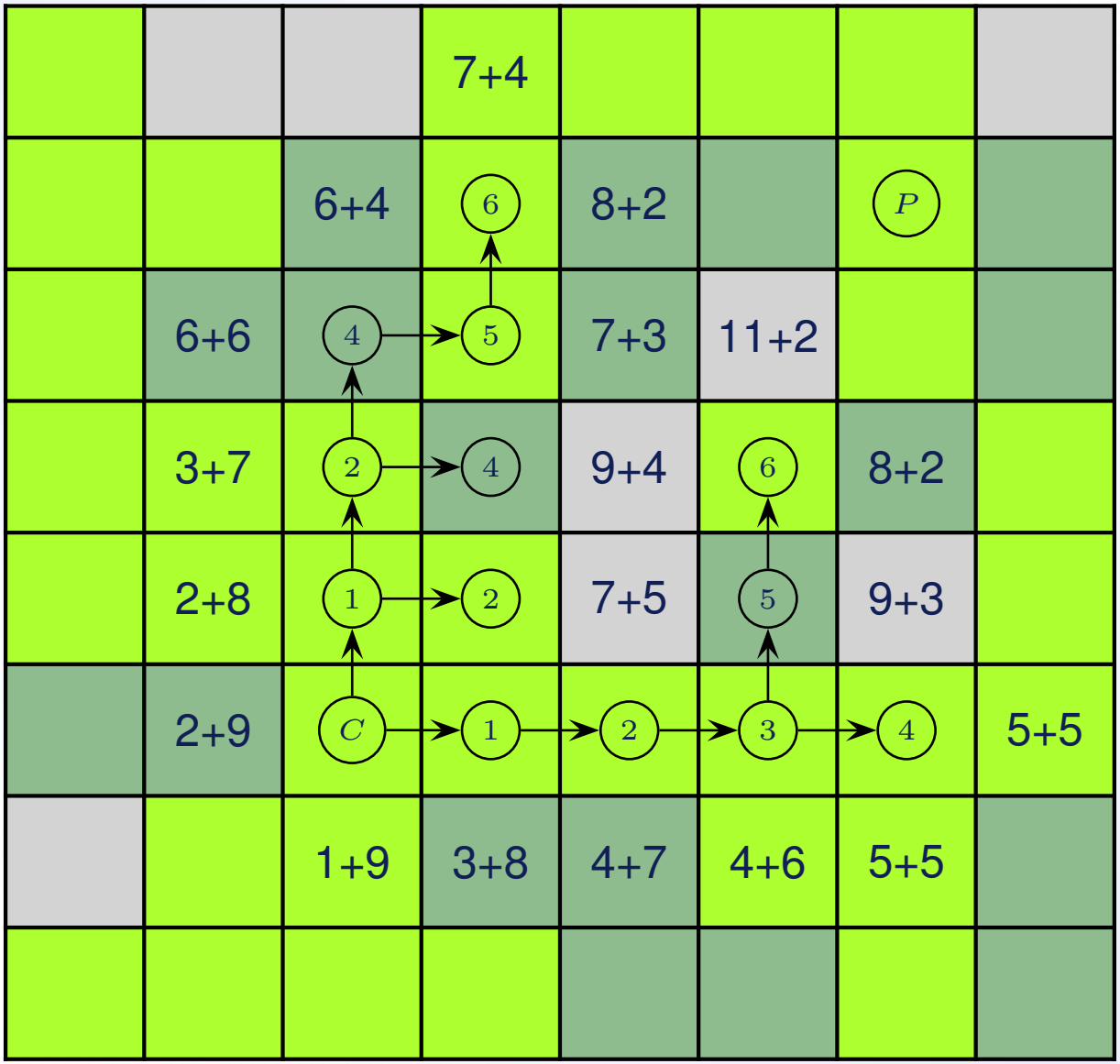
Exemple (7)



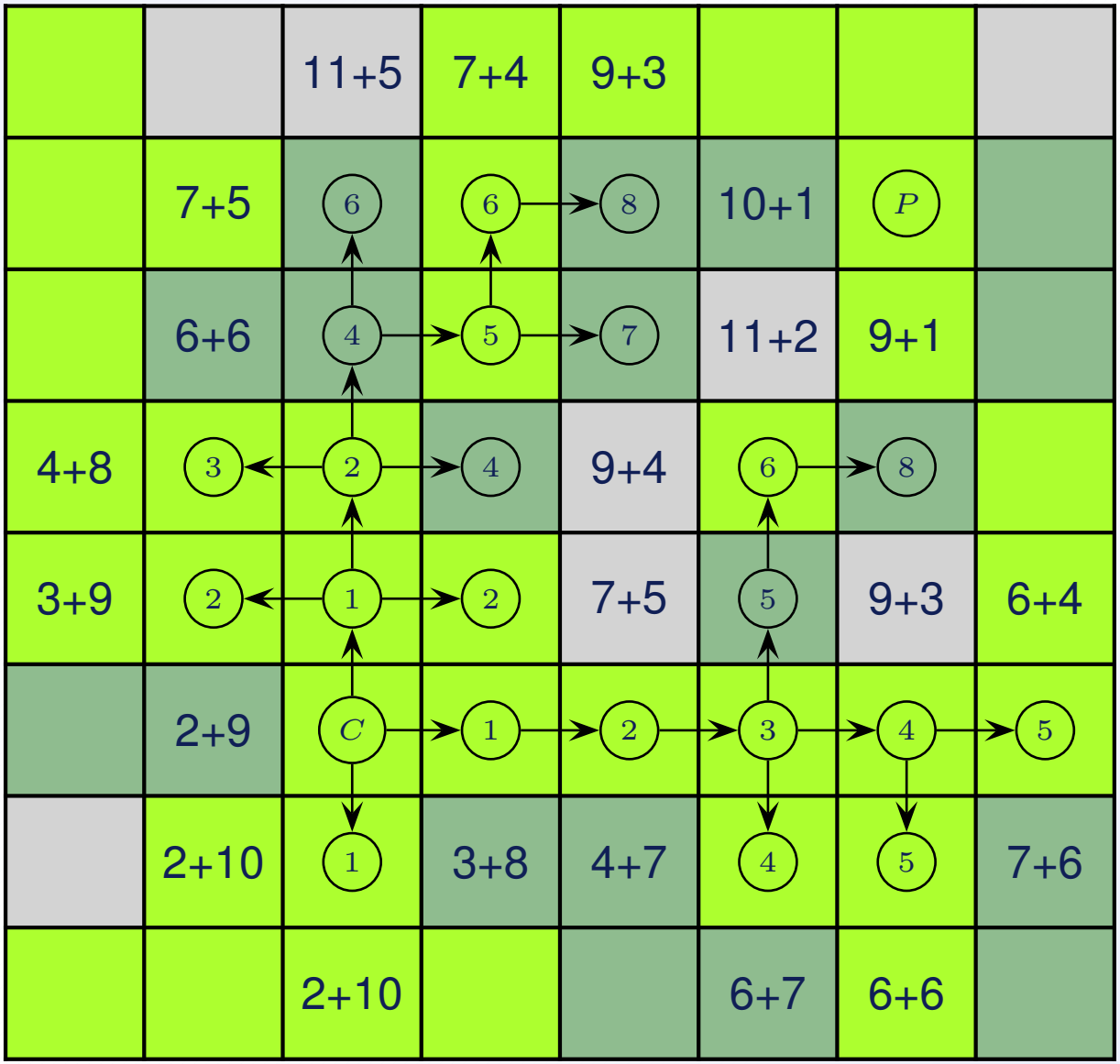
Exemple (8)


6+4
6+3
P
6+6
4
5
7+3
11+2
3+7
2
4
9+4
6
8+2
2+8
1
2
7+5
5
9+3
2+9
C
1
2
3
4
5+5
1+9
3+8
4+7
4+6
5+5

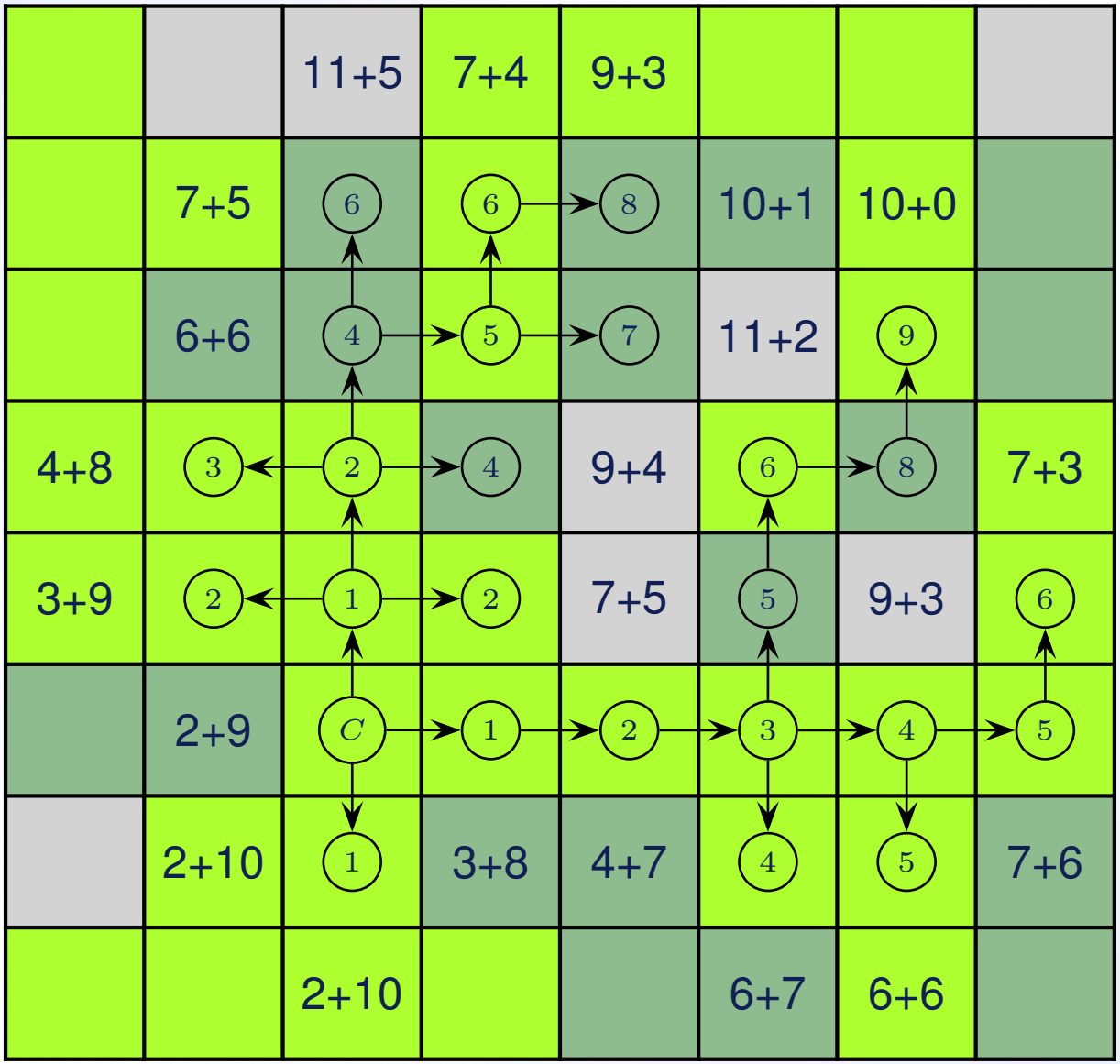
Exemple (9)


7+4
6+4
6
8+2
P
6+6
4
5
7+3
11+2
3+7
2
4
9+4
6
8+2
2+8
1
2
7+5
5
9+3
2+9
C
1
2
3
4
5+5
1+9
3+8
4+7
4+6
5+5

Exemple (10)


11+5
7+4
9+3
7+5
6
6
8
10+1
P
6+6
4
5
7
11+2
9+1
4+8
3
2
4
9+4
6
8
3+9
2
1
2
7+5
5
9+3
6+4
2+9
C
1
2
3
4
5
2+10
1
3+8
4+7
4
5
7+6
2+10
6+7
6+6

Exemple (11)


11+5
7+4
9+3
7+5
10+1
10+0
6+6
11+2
4+8
9+4
7+3
3+9
7+5
9+3
2+9
2+10
3+8
4+7
7+6
2+10
6+7
6+6

Exemple (12)