Couverture par partitionnement parallèle le long d'un côté

Formulation du problème :

Nous cherchons à **couvrir complètement une zone** contenant des **zones où le drone ne peut pas voler**. Nous cherchons également à **réduire le temps de complétion**, et donc la **consommation d'énergie**, en **optimisant le trajet** et en **réduisant le nombre de virages** effectués.

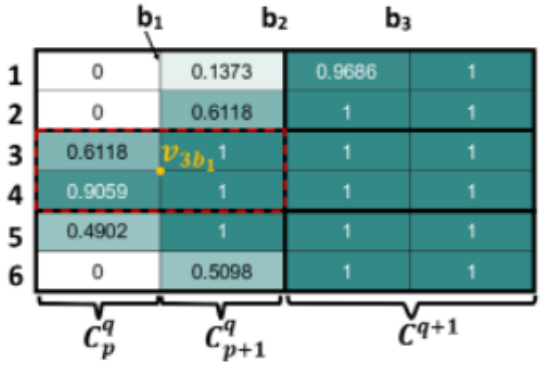
Fonctionnement :

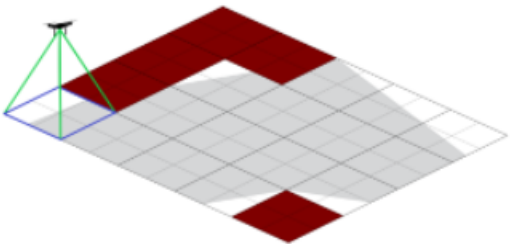
1. La zone est quadrillée ;

2. La zone est partitionnée en fonction de la zone de non-vol. Elle est maintenant constituée de sous-zones de tailles approximativement égales. Afin de réduire le pourcentage de zone non couverte, les bords des partitions sont placés sur les bords des cellules de la grille (certaines techniques ne couvrent pas les cellules situées sur le bords, ce qui engendre un pourcentage de zone non-couverte conséquent) ;

3. On détermine le trajet du drone dans chaque sous-zone.

L'image ci-dessous représente une zone à couvrir. Elle est quadrillée en cellules rectangulaires et subdivisée en 6 sous-zones de 4 cellules chacune.
Chaque cellule comporte un nombre qui représente un taux "aire appartenant à la zone / aire de la cellule". Ainsi, une cellule comportant le nombre "0.5" aura 50% de son aire appartenant à la zone à couvrir.
Un groupe de 4 cellules adjacentes (dont au moins 3 n'ont pas un taux à 0) ont un centre *v*. Lorsqu'il se déplace, le drone passe par ce centre.

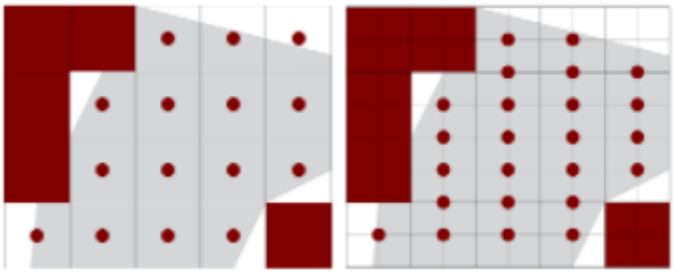




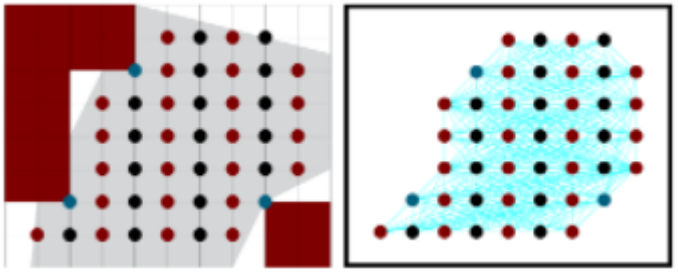
Ainsi, à un temps t, le drone couvre exactement une zone équivalente à 4 cellules "mineures". La taille de cette zone ("cellule majeure") est déterminée par la résolution et le champ de vision du drone.

La zone est rectangulaire, et pour réduire le nombre de virages nécessaires, le drone se déplacera la majorité du temps dans une direction telle que le plus petit côté de ce rectangle soit parallèle au côté le plus long de la grille.
Les détails de l'algorithme de partitionnement feront l'objet de recherches futures.

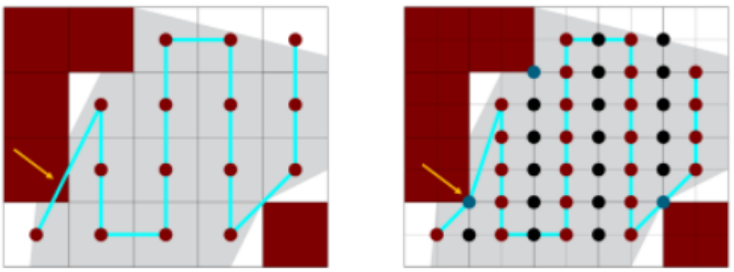
Chaque cellule majeure est représentée par un point, qui peut se trouver à l'extérieur de la zone. Après la subdivision en cellules mineures, seulement les points à l'intérieur de la zone peuvent être conservés. Dans l'exemple ci-dessous, le drone se déplace de bas en haut :



Durant le virages, le drone se déplace sur les bords des cellules.
Afin de sélectionner les meilleurs virages, on ajoute des points sur ces bords. On lie ensuite les points entre eux et on détermine le chemin final.

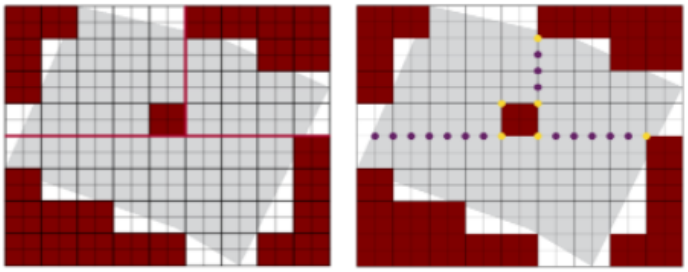


Les points marrons représentent les points du chemin principal. Les points noirs sont les points d'aide aux virages entre deux parties du chemin principal. Les points noirs situés sur le bord d'une zone de non-vol deviennent des points bleus, importants pour déterminer les virages.



Des points violets sont utilisés sur les bords des partitions. Ils deviennent jaunes s'ils se trouvent sur un bord de zone de non-vol. Ces points seront utiles pour éviter qu'une zone ne

soit couverte plusieurs fois, et pour trouver la meilleure façon de joindre deux chemins (deux couvertures de partitions) si un seul drone est utilisé.



Les points sont triés en 5 groupes (R1, R2, R3, R4 et R5) afin de les filtrer.
R1 inclut les points qui ne peuvent pas servir de virage.
R2 et R3 incluent les points marrons pouvant servir de virage.
R4 et R5 incluent les points noirs pouvant servir de virage.
On assigne des valeurs à chaque point noir ou marron. Ceux étant en début ou fin de colonne sont bien sûr plus susceptibles d'être sélectionnés comme virages.

Algorithm 1 Brown nodes key assignment

Input: $C_p^q \wedge C_{p+1}^q \wedge b_j$
1: $F_1 \leftarrow$ row of first vertex in C_p^q
2: $F_2 \leftarrow$ row of first vertex in C_{p+1}^q
3: $L_1 \leftarrow$ row of last vertex in C_p^q
4: $L_2 \leftarrow$ row of last vertex in C_{p+1}^q
5: **for** (each b_j where j is odd) **do**
6: **if** $(F_1 == F_2)$ **then**
7: $R_3^j = [F_1 : F_2]$
8: $R_1^j.min = F_1 + 1$
9: **end if**
10: **if** $(F_1 < F_2)$ **then**
11: $R_3^j = [F_2 + 1 : F_1]$
12: $R_1^j.min = F_2 + 2$
13: **end if**
14: **if** $(F_1 > F_2)$ **then**
15: $R_3^j = [F_1 + 1 : F_2]$
16: $R_1^j.min = F_1 + 2$
17: **end if**
18: **if** $(L_1 == L_2)$ **then**
19: $R_1^j = [R_1^j.min : L_1 - 1]$
20: $R_2^j = [L_1 : L_2]$
21: **end if**
22: **if** $(L_1 < L_2)$ **then**
23: $R_1^j = [R_1^j.min : L_1 - 2]$
24: $R_2^j = [L_1 - 1 : L_2]$
25: **end if**
26: **if** $(L_1 > L_2)$ **then**
27: $R_1^j = [R_1^j.min : L_2 - 2]$
28: $R_2^j = [L_2 - 2 : L_1]$
29: **end if**
30: **end for**
31: **for** (each v_{ib_i} where i $\in R_1^j$) **do**
32: v_{ib_j}.setKey(00)
33: **end for**
34: **for** (each v_{ib_i} where i $\in R_2^j$) **do**
35: v_{ib_j}.setKey(12)
36: **end for**
37: **for** (each v_{ib_i} where i $\in R_3^j$) **do**
38: v_{ib_j}.setKey(11)
39: **end for**

Algorithm 2 Black nodes key assignment

Input: b_j
1: **if** (j is even) **then**
2: $R_4^j.min = \min\{R_3^{j-1}.min, R_3^{j+1}.min\}$
3: $R_4^j.max = \max\{R_3^{j-1}.max, R_3^{j+1}.max\}$
4: $R_4^j = [R_4^j.min, R_4^j.max]$
5: $R_5^j.min = \min\{R_2^{j-1}.min, R_2^{j+1}.min\}$
6: $R_5^j.max = \max\{R_2^{j-1}.max, R_2^{j+1}.max\}$
7: $R_5^j = [R_5^j.min, R_5^j.max]$
8: **for** (each v_{ib_j} where i $\in R_4^j$) **do**
9: v_{ib_j}.setKey(21)
10: **end for**
11: **for** (each v_{ib_j} where i $\in R_5^j$) **do**
12: v_{ib_j}.setKey(22)
13: **end for**
14: **end if**

Nous pouvons ensuite relier les points en suivant 4 règles :

1. Les noeuds marrons en R1 d'une colonne j ne peuvent être liés qu'avec soit :
   - des noeuds voisins marrons en R1
   - des noeuds voisins marrons en R2 ou R3 sur la même colonne j

2. Les noeuds marrons en R2 (respectivement R3) sont liés aux noeuds noirs de la colonne j+1 en R5 (resp. R4). Les noeuds marrons en R2 (resp R3) sur la colonne j sont liés aux noeuds noirs en R2 (resp R3) sur la colonne j+2.

3. Deux nœuds noirs ne peuvent être liés, et les nœuds noirs en R4 et R5 sont omis du graphe.

4. Si un nœud bleu est en haut (resp en bas) d'une colonne, alors tous les liens au-dessous (resp au-dessus) sont supprimés, afin d'éviter les zones de non-vol.

Algorithm 3 Edge assignment to the nodes

Input: b_j
Output: edges

1: **for** (each v_{ib_j} where i $\in R_3^j$) **do**
2: **for** (each $v_{ab_{j+2}}$ where a $\in R_3^{j+2}$) **do**
3: v_{ib_j}.setEdge($v_{ab_{j+2}}$)
4: **end for**
5: **end for**
6: **for** (each v_{ib_j} & $v_{ab_{j+2}}$ where i $\in R_3^j$ and a $\in R_3^{j+2}$) **do**
7: **for** (each $v_{nb_{j+1}}$ where n $\in R_4^{j+1}$) **do**
8: v_{ib_j}.setEdge($v_{nb_{j+1}}$)
9: $v_{ab_{j+2}}$.setEdge($v_{nb_{j+1}}$)
10: **end for**
11: **end for**
12: **for** (each v_{ib_j} where i $\in R_2^j$) **do**
13: **for** (each $v_{ab_{j+2}}$ where a $\in R_2^{j+2}$) **do**
14: v_{ib_j}.setEdge($v_{ab_{j+2}}$)
15: **end for**
16: **end for**
17: **for** (each v_{ib_j} & $v_{ab_{j+2}}$ where i $\in R_2^j$ and a $\in R_2^{j+2}$) **do**
18: **for** (each $v_{nb_{j+1}}$ where n $\in R_5^{j+1}$) **do**
19: v_{ib_j}.setEdge($v_{nb_{j+1}}$)
20: $v_{ab_{j+2}}$.setEdge($v_{nb_{j+1}}$)
21: **end for**
22: **end for**

Il s'agit d'un graphe semi-orienté (on ne connaît les directions des vecteurs qu'à la fin).
Le point de départ est connu : c'est le premier ou le dernier noeud marron sur la colonne la plus à gauche. Ces deux points vont générer deux chemins différents, et le plus rapide sera choisi.

Ensuite, on assigne un poids à chaque vecteur, en fonction des angles et des types de noeuds, et on génère le chemin :

Algorithm 4 Path Generation

Input: v_{ij} : startpoint, $\mathcal{G}$: Graph,
Output: χ

1: $upward = true$
2: **if** (key(v_{ij})==11) **then**
3: $upward = false$
4: **end if**
5: **repeat**
6: **if** (v_{ij}.color=brown) **then**
7: **if** ($upward == false$) **then**
8: v_{next}=v_{i-1j}
9: **else**
10: v_{next}=v_{i+1j}
11: **end if**
12: **if** (v_{next} is an adjacent neighbor to v_{ij}) **then**
13: **if** (key(v_{next}) = 00) **then**
14: $\chi = \chi \cup \{(v_{ij}, v_{next})\}$
15: $\mathcal{I} = \mathcal{I} - \{v_{next}\}$
16: $v_{\cdot} = v_{ij}$
17: **else**
18: **if** (key(v_{next}) = 11 key(v_{ij}) = 11) **then**
19: $k_1 = 00, k_2 = 12$
20: **else**
21: **if** (key(v_{next}) = 12 key(v_{ij}) = 12) **then**
22: $k_1 = 00, k_2 = 11$
23: **end if**
24: **end if**
25: Find brown vertex $v_{pq} \in \mathcal{I}$ key(v_{pq}) = $k_1$key(v_{pq}) = k_2 & $v_{ij} \rightsquigarrow^* v_{pq}$
26: **if** ($v_{pq} \neq$ null) **then**
27: Find $p_k \in \varrho(v_{ij} \rightsquigarrow^* v_{pq})$ using Equations (10) and 11
28: $\chi = \chi \cup E(v_{ij} \rightsquigarrow_{p_k} v_{pq})$
29: $\mathcal{I} = \mathcal{I} - V(v_{ij} \rightsquigarrow_{p_i} v_{pq})$
30: Assign time cost t_{pq} using Equation (12)
31: $v_{\cdot} = v_{pq}$
32: $upward$ =!$upward$
33: **else**
34: $\chi = \chi \cup \{(v_{ij}, v_{next})\}$
35: $\mathcal{I} = \mathcal{I} - \{v_{ij}, v_{next}\}$
36: **end if**
37: **end if**
38: $v_{ij} = v_{\cdot}$
39: **end if**
40: **end if**
41: **until** ($\mathcal{I} = \phi$)

Points positifs

- Prise en compte de possibles zones de non-vol dans la zone.
- La réduction de la durée du parcours offre un avantage dans la désirabilité du produit.
- La réduction de la consommation d'énergie peut nous permettre d'embarquer sur le drone des équipements plus lourds.
- La réduction de la consommation d'énergie peut aussi permettre de couvrir le champ plus rapidement, si le drone n'a pas besoin de pause pour se recharger.
- Méthode créée spécialement pour les parcours de zones par des drones (pas de zones théoriquement couvertes mais qui ne le sont pas à cause du fonctionnement des drones : mouvement, etc).

Points négatifs

- Il arrive de passer plusieurs fois au même endroit, ce qui est dû au partitionnement de la zone en sous-zones ayant chacune leur propre parcours : plus il y a de trous, plus cela arrive. Bien que ce problème soit amoindri si plusieurs drones couvrent la zone, ce ne sera probablement pas notre cas.

- Dans le cas de bords fortement concaves sur une petite longueur, il peut arriver que le trajet sorte de la zone à couvrir. Nous devrons vérifier que cela ne signifie pas que le drone devra sortir du champ, en fonction de son champ de vision et de la “zone tampon” entre la surface cultivée et les bords du champ (délimitation légale / clôture, etc).

Axes d’optimisation

- Plutôt que de réduire la distance entre les arrêts et les départs de trajets dans chaque partition, le partitionnement pourrait être modifié afin de créer des zones dont la durée de couverture coïncide avec l’autonomie de la batterie du drone. En effet, une pause dans le trajet devra sans doute être réalisée dans tous les cas.

- Nous pourrions, pour chaque couverture de sous-zone, essayer de réduire la distance entre les lieux de départ et d’arrivée et la borne de rechargement.

- Si le drone doit s’arrêter durant le trajet (prise de contrôle par un humain ou nécessité de recharge de la batterie), il pourrait être intéressant de recalculer l’entièreté du chemin plutôt que de le continuer en reprenant à l’endroit où le drone s’est arrêté.

- Au lieu de choisir le chemin le plus rapide pour chaque sous partie, nous pourrions choisir chacun de ces chemins en testant les combinaisons (car il faudra les relier).

- Nous pourrions définir la position du centre v en fonction des taux des 4 cellules adjacentes.

- Il pourrait être intéressant de calculer différents chemins parcourant les X prochains points pour définir le plus optimisé (dans le cas où cela ne serait pas celui parallèle au plus long côté, ex : angle concave).

Notes

- Dans le cas où plusieurs drones couvrent le champ, avec chacun leur zone attribuée, la contrainte d’une distance de sécurité doit être ajoutée.