SEGMENTATION D'IMAGES À L'AIDE D'AGENTS SOCIAUX

11 MAI 2021

Table des matières

Table des figures

1 | Description du travail d'article

Dans l'article, ils se sont basés sur deux méthodes de segmentation d'image utilisant des modèles provenant de la biologie, soient les fourmis et les araignées sociales. En premier, ils ont présenté les principes de ces deux méthodes en décrivant leurs algorithmes et leurs fonctionnements, ainsi les techniques d'optimisation des paramètres utilisés dans chaque algorithme, ensuite ils ont introduit les concepts des deux autres algorithmes classiques de segmentation d'image, région growing et Otsu, afin de comparer leurs résultats avec les deux premières méthodes, en se basant sur plusieurs critères de comparaison, à savoir le nombre de régions, correspondance entre les régions de l'image réelle et l'image segmenté ainsi le temps d'exécution, au final ils ont conclu que le meilleur résultat c'est en combinant la méthode des araignée avec celle des fourmis, pour la raison que les araignées sociales et les fourmis sociales ont des rôles complémentaires et la fusion des deux méthodes produira une nouvelle segmentation ayant des contours précis sur les images résultantes.

2 | Les fourmis

2.1 Algorithme des fourmis

2.1.1 Description de l'algorithme des fourmis et son amélioration

Les fourmis sont des agents réactifs qui ont la possibilité de se déplacer dans l'environnement, en se communiquant par le biais de phéromones qui s'évaporent par le temps. Elles peuvent mettre un pixel dans un état visité, marqué ou libre.
D'abord la fourmi vérifie si le pourcentage de pixels visités vérifie la condition établie par l'utilisateur, une fois qu'elle est confirmée la fourmi calcule le gradient morphologique(*en soustrayant la couleur minimale de la couleur maximale de ses patch voisins*) sur son propre pixel , puis le pixel est marqué si la condition du seuil du gradient est vérifié si non elle fait rien. Ensuite, Elle consulte ses voisins et se déplace vers celui ayant le gradient le plus élevé.

L'algorithme présenté décrit exactement le fonctionnement expliqué, il lui manquait seulement quelques ajustements, à savoir la notion de visité, lorsqu'une fourmi se déplace vers un pixel elle lui change d'état de libre à visité afin d'éviter de repasser dans ce pixel sauf si besoin ou pas de choix. Dans le but d'améliorer l'algorithme, on a introduit la notion de visité en faisant en sorte de mettre le gradient du pixel à nulle dès qu'une fourmi est introduite dans un pixel donné.

2.1.2 Analyse des paramétres

L'algorithme des fourmis sociales possède particulièrement trois (3) paramètres qui doivent être fixés par l'utilisateur tel que le nombre d'agents , le seuil du gradient morphologique et le pourcentage de pixel à visiter dans le but d'accomplir la tache de segmentation.

— **Le pourcentage de pixel à visiter *PerV*** : est un paramètre qui mesure le pourcentage de pixels visités dans l'image, il est utilisé comme condition d'arrêt. Les fourmis essayent de visiter le pourcentage effectué par l'utilisateur, tant que ce dernier n'est pas atteint le processus continue de travailler.

— **Seuil du gradient morphologique *Grad*** : Ce paramètre définit la notion du phéromone, tant que le gradient d'un patch dépasse ce seuil définie par l'utilisateur , le patch est classé comme marqué

— **Nombre d'agents *Nbagent*** : le paramètre qui définit le nombre de fourmis travaillant sur la segmentation d'image, plus on a un nombre important plus l'exécution est rapide.

2.1.3 Optimisation des paramétres

En premier nous avons fixé le pourcentage de pixels à visiter à 100 % pour nous assurer que tous les pixels sont visités.

Ensuite pour le seuil du gradient morphologique nous avons commencé à l'optimiser avec la technique donnée dans l'article comme la différence minimale entre deux maxima locaux de l'histogramme de l'image ayant la distribution de pixels la plus élevé entre eux, en définissant un histogramme qui prend les gradient en axe des abscisses et le nombre de patchs ou de pixels en axe des ordonnés , par contre ça donne pas un bon résultat , c'est la raison pour laquelle nous avons penser à optimiser en utilisant l'outil behaviour space mais le seul problème qu'on a rencontré dans ce cas c'est que cet outil nous permet pas d'évaluer des figures, donc on s'est basculé à une solution d'algorithme d'optimisation que nous avons décrit qui travaille sur le même

principe de behaviour space mais en faisant varier les figures.

En conséquence, afin d'optimiser les paramètres de cet algorithme de fourmis sociales on s'est inspiré de l'outil behaviour space pour décrire un algorithme qui travaille sur le même principe mais plus adapté à notre cas .

- **Pourcentage de pixels à visiter *PerV*** : Fixer à 100 %
- **Seuil du gradient morphologique *Grad*** : nous avons expérimenté les valeurs comprises dans l'intervalle [1.0 2.4] avec un pas de 0.2 , car entre les voisins on trouve jamais une différence de couleur de patch qui dépasse 2.4 tellement les patchs qui sont voisins se dégradent faiblement.
- **Nombre d'agents *Nbagent*** : Nous avons expérimenté les valeurs comprises dans l'intervalle [5 20] avec un pas de 5.

2.1.4 Analyse des résultats obtenus

Une fois l'algorithme d'optimisation est exécuté plusieurs résultats de segmentation d'image sont obtenues :

- En testant les valeurs d'intervalle de nombre d'agents , nous avons remarqué qu'il n'y a pas vraiment de différence sur la façon de détecter les contours ,Par conséquent le nombre d'agents influence sur le temps d'exécution, par contre le fonctionnement de leurs taches reste le même.
- En testant les valeurs d'intervalle du seuil du gradient morphologique, nous avons remarqué qu'à partir de 1.2 jusqu'a 1.4 les fourmis détectent les contours mais d'une façon exagérée , puis au delàs de la valeur 1.8 les fourmis tracent parfaitement les contours mais elles commencent à manquer quelques uns.

2.1.5 Le cas idéal pour l'algorithme des fourmis

Le meilleur résultat obtenu avec la méthode de l'algorithme d'optimisation, c'est le cas ou nous avons pris le seuil du gradient à 1.6, puis aprés le nombre d'agent n'est pas important vu qu'en changeant sa valeur nous avons eu presque des même résultats,

cepandant ça influence sur le temps d'exécution , c'est pourquoi il vaut mieux prendre un nombre important d'agents.

La figure 2.3 illustre le meilleur résultat obtenu.

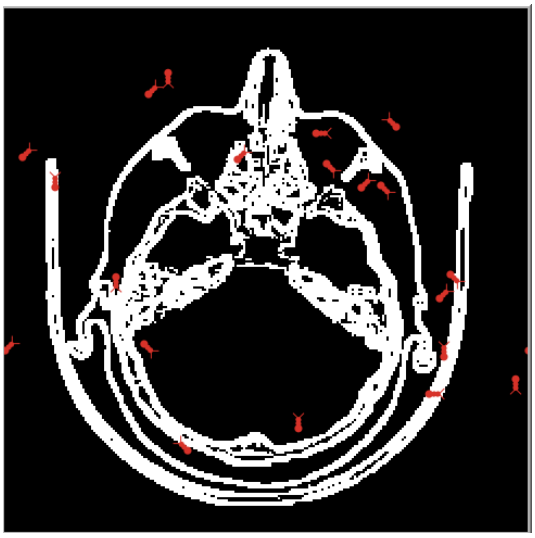

FIGURE 2.1 – Le résultat Obtenu avec nombre d'agent à 20, le nombre de pixel à visiter à 100% et un seuil du gradient morphologique à 1.2

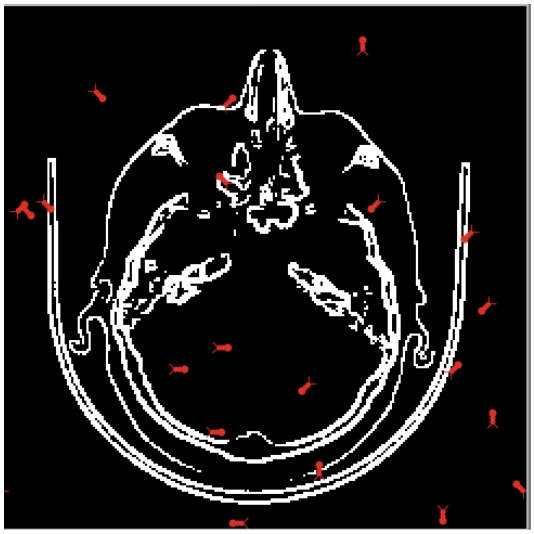

FIGURE 2.2 – Le résultat Obtenu avec nombre d'agent à 20, le nombre de pixel à visiter à 100% et un seuil du gradient morphologique à 2.4

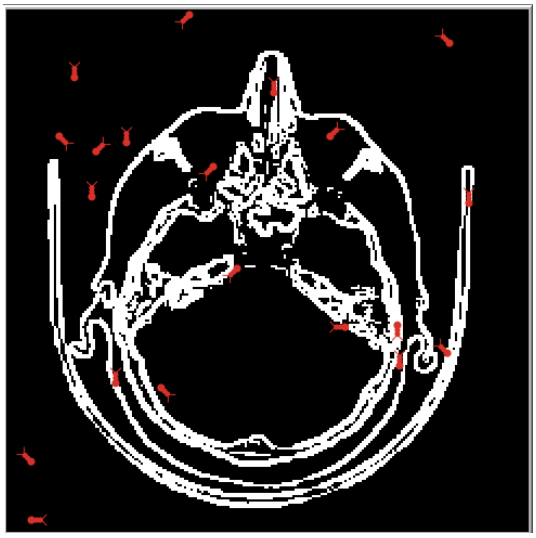

FIGURE 2.3 – Le résultat Obtenu avec nombre d'agent à 20, le nombre de pixel à visiter à 100% et un seuil du gradient morphologique à 1.6

3 Les araignées

3.1 Algorithme des araignées

3.1.1 Description de l'algorithme donné et de son amélioration

Les araignées sont des agents réactifs qui sont capables de se déplacer dans l'environnement, fixer une soie et revenir en arrière. Les araignées essaient de se déplacer dans leur voisinage, elles donnent la priorité aux pixels non soyeux et essaient de fixer des soies sur eux. S'ils échouent, ils reviennent à la dernière soie fixée. À la fin du processus, des groupes d'araignées se forment et sont appelés des colonies. L'algorithmes tel qu'il est donné dans l'article est incomplet (il s'agit d'un pseudo algorithme pas d'un algorithme détaillé), mais aussi fait appel à des fonctions qui ne sont pas expliquées dans l'article ni dans les articles référencés telle que la fonction **computeInt(pixel(a))**. Nous avons regardé dans les articles référencés pour mieux comprendre le principe puis nous avons mis des hypothèses pour développer l'algorithme et l'améliorer dans le but d'avoir des résultats rapprochés de ceux obtenus dans l'article. Lors de la création des araignées, nous les avons placées aléatoirement sur l'image importée, puis elles commencent à explorer leurs régions.

— Au départ les agents araignées commencent par calculer la liste des voisins telle que la différence entre la couleur du patch ou se trouve l'araignée et la couleur du voisin est inférieur à un seuil "thres", car les araignées cherchent des zones homogènes ou presque homogènes.

— Puis si cette liste n'est pas vide, l'araignée priorise le voisin qui n'a pas encore

de soie fixée, grâce à une variable que nous avons défini pour chaque patch et qui vaut 1 si aucune soie n'est fixée et vaut 0 si on a déjà une soie fixée sur ce patch.

— Une fois un voisin est choisi, l'araignée se déplace vers lui, fixe une soie entre son patch actuel et lui et affecte à la variable "lastFixedSilk" le patch de départ.

— Sinon, si aucun voisin ne vérifie la condition, l'araignée revient au patch ou elle a fixé la dernière soie, si elle n'a fixé aucune soie elle revient à son patch de début d'exécution.

A la fin de cet algorithme un ensemble d'araignées qui détecte la même région est dite colonie.

Pour différencier les régions trouvées par chaque colonie nous avons attribué une couleur unique pour chaque araignée lors de sa création. La figure 3.1 représente l'algorithme de l'article et la figure 3.2 représente l'algorithme amélioré.

Algorithm 2 Spiders method

Require: Pixels: Matrix of pixels $\in \mathbb{N}^2$, Nbagent: number of agents $\in \mathbb{N}$, NbIt: Iteration number $\in \mathbb{N}$ and Thres: grayscale Threshold $\in \mathbb{N}$.

```
1:  while Ite − > 0 do
2:    for Each agent a do
3:      T ← computeInt(Pixel(a)).
4:      if T ≤ Thres then
5:        Move(a, Position(Pixel(T))).
6:        Silkfixing(Position(Pixel(a)), Position(Pixel(T))).
7:        LastFixedSilk(a) ← Pixel(a).
8:      else
9:        Moveback(a, LastFixedSilk(a)).
10:     end if
11:   end for
12: end while
```

FIGURE 3.1 – Algorithme des araignées de l'article.

Algithm 2 : Spiders Developed

Pixels: Matrix of pixels ∈ N^2, Nbagent: number of agents ∈ N, NbIt: Iteration number ∈ N and Thres: grayscale Threshold ∈ N.

```
for Each agent a do // Initialisation
      lastFixedSilk = curent-patch
end for

while Ite – > 0 do
  for Each agent a do
        listNeighbors =neighbors with (pcolor of the current patch – pcolor of neighbor)  < thres
        if  listNeighbors != empty
            nextPatch =  one of listNeighbors with ( NoSilks =1)
            if next-Patch != nobody  // we prioritize patches not silked yet
                NoSilks = 0
                lastFixedSilk = current-patch
                move (nextPatch)
                fixSilk ( current- patch, next-Patch)
            else
                nextPatch = one of listNeighbors // we choose one patch randomely
                 move (nextPatch) // move without fixing a silk
        else
            move-back( lastFixedSilk)
        end If
    end For
end While
```

FIGURE 3.2 – Algorithme des araignées amélioré .

3.1.2 Analyse des paramètres

Nous avons principalement cinq (05) paramètres qui permettent aux araignées de mener à bien leurs tâches de segmentation d'image en régions.

— **Thres :** un seuil que les araignées doivent respecter lors de leurs déplacements et lors de la fixation de soie. Ce paramètre assure que les araignées sont entrain de fixer des soies dans une région ou la couleur des patch est homogène ou presque homogène et les empêche de dépasser les frontières de leurs régions, de cette façon chaque agent araignée détecte une seule région.

— **NbIte :** Le nombre d'itération qu'on peut correspondre au cycle de vie des araignées dans lequel chacune des araignées exécute un ensemble d'instructions. Ce paramètre est important car plus le cycle de vie des araignées est long plus

elles ont la chance de détecter toutes les régions.

— **Nbagent :** le nombre d' araignées consacrées pour la tâche de segmentation. Comme les araignées sont placées d'une manière aléatoire dans l'environnement (l'image), il est recommandé d'attribuer à ce paramètre une valeur un peu élevée pour augmenter les chances d'avoir des araignées dans toutes les zones de l'image.

— **LastFixedSilk :** un paramètre associé à chaque araignée, son objectif est de sauvegarder la trace du dernier patch ou l'araignée a fixée une soie et de se déplacer vers ce patch dans le cas de blocage (aucun voisin ne vérifie la condition de thres).

— **NotSilked :** un paramètre associé à chaque araignée, son objectif est d'établir la priorité, dans le cas où l'araigné a plusieurs voisins qui vérifie la condition de thres, elle va choisir celui qui n'a pas de soi déjà fixé (NotSilked =1).

3.1.3 Optimisation des paramètres

Pour l'optimisation des paramètres nous avons implémenté un algorithme en s'inspirant de la tâche que réalise l'outil behavior space. cet algorithme permet de tester combinaisons de plusieurs valeurs des paramètres, pour chaque combinaison nous retournons une image que nous analysons par la suite. Nous n'avons pas utilisé le behavior space car il ne permet pas de retourner des images.

— **Thres :** nous avons testé dans un intervalle de [0.08, 0.24] avec un pas de 0.08, car au-delà de 0.24 le voisinage n'est pas considéré homogène et toutes les araignées franchissent la frontière de leurs régions. Les valeurs de thresh inférieur à 0.08 ne sont pas prises en compte car avec 0.08 est déjà considéré comme homogène donc toutes les valeurs inférieur 0.08 vérifient cette contrainte d'homogénéité.

— **NbIte :** pour estimer le nombre d'itération nécessaire pour que les araignées détectent toutes les régions, nous avons testé cet ensemble de valeurs [20000 30000 50000]. Si avec x Nbagent et 10000 itérations on peut bien segmenter, pas

la peine de faire 20000 itérations.

— **Nbagent :**nous avons donné une liste [5 10 15] pour voir la relation du nombre d'agents avec le nombre d'itération.

3.1.4 Analyse des résultats obtenus

Nous avons obtenu plusieurs résultats après avoir exécuter l'algorithme d'optimisation des paramètres.

— Avec un nombre d'agents = 5, des régions sont restées sans araignées (le nombre de régions est supérieur au nombre d'agents), donc le nombre d'agents ne doit pas être inférieur à 6 dans le cas de notre image, sinon un nombre de régions ne sera jamais détecté (Figure 3.5).

— Avec un nombre d'itération = 20000 nous avons eu de bon résultat seulement dans le cas où le nombre d'agent est élevé (15), cela se justifie par le fait d'avoir beaucoup d'agents, chaque régions va avoir plusieurs araignées chacune avec un cycle de vie de 20000 itérations. Par contre avec 10000 itérations même si on a un nombre élevé d'agent (15), si la région est un peu grande les araignées n'auront pas suffisamment de temps pour la segmenter(Figure 3.3). Avec Nbite= 50000 nous avons obtenu une bonne segmentation mais dans le cas où le nombre d'agent est élevé ce nombre d'itération ne fait qu' augmenter le temps d'exécution sans rien ajouter au performance.

— Avec Thresh = 0.16, nous avons obtenu de très bons résultats. Or avec Thresh = 0.24 des régions différentes sont considérées comme une seule région car les araignées sortent de leur périmètre de segmentation vers d'autres régions(Figure 3.4). Le cas de thresh = 0.08 a donné de mauvais résultats, une araignée ne fixe pas des soies sur tous les patchs d'une région car évidement les régions ne sont pas parfaitement homogènes.

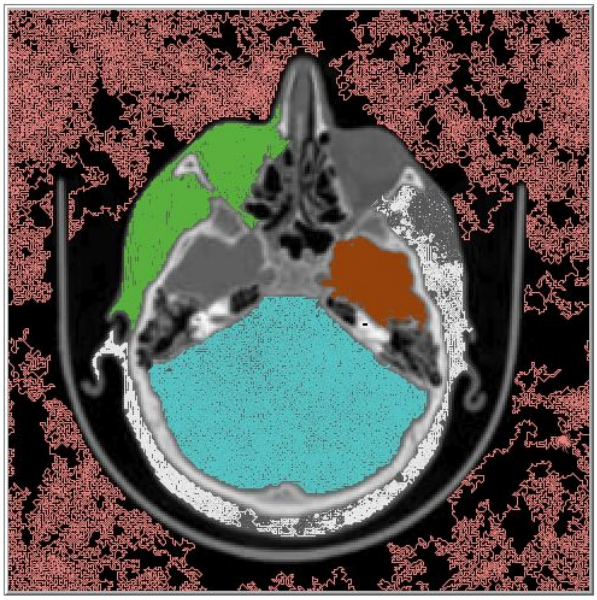

FIGURE 3.3 – Résultat de l'algorithme avec Thres =0.16 et Nbite= 20000

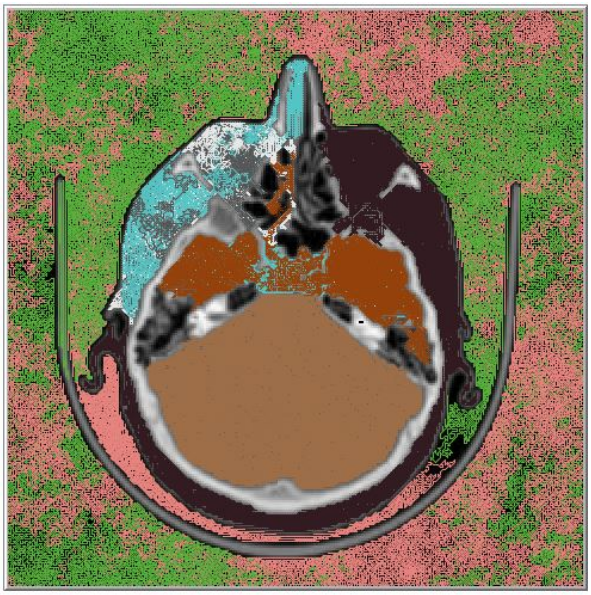

FIGURE 3.4 – Résultat de l'algorithme avec Thres = 0.24 et Nbite = 50000

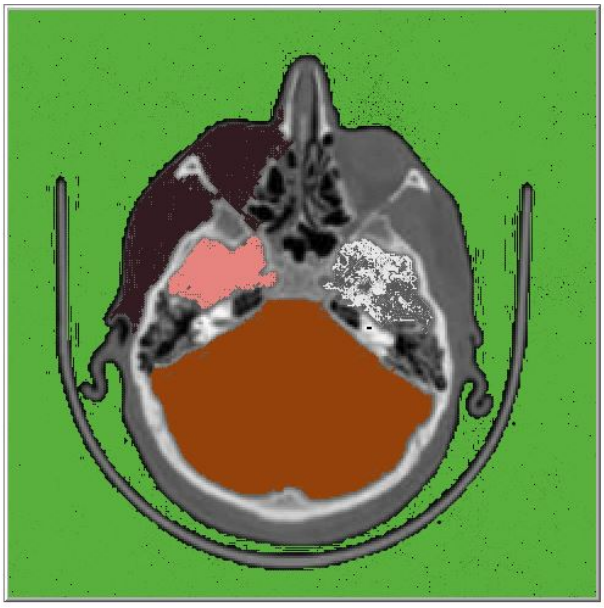

FIGURE 3.5 – Résultat de l'algorithme avec nbagent =5 Thres =0.16 et Nbite= 50000

3.1.5 Le cas idéal pour l'algorithme des araignées

Il est exigé dans cette approche de placer les araignées d'une manière aléatoire sur l'image importée. Ceci est la limite principale de cet algorithme car dans la majorité des cas plusieurs araignées se placent dans la même régions, et donc pas mal de régions restent sans araignées (impossible de les détecter, la figure 3.6). Pour cela il est

envisageable d'utiliser un nombre important d'araignées pour s'assurer que toutes les régions vont être explorées et segmentées. La figure 3.7 est un cas où nous avons mis 7 araignées manuellement sur l'image (chaque région avec une araignée), Thresh = 0.16 et NbIte = 50000. Le résultat obtenu est parfait, car chaque araignée a ségmenté sa région, nous avons obtenue 7 région différentes.

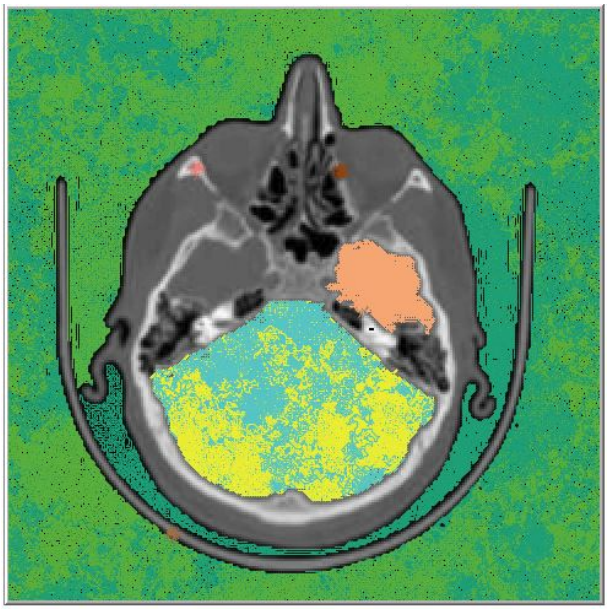

FIGURE 3.6 – 7 araignées placées aléatoirement sur l'image.

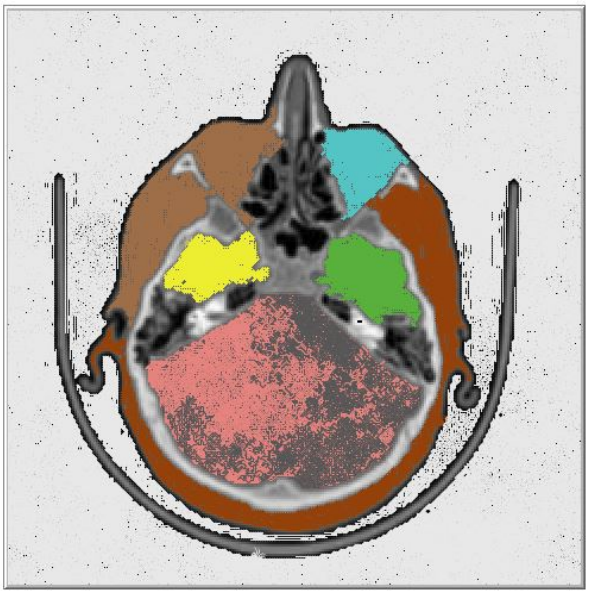

FIGURE 3.7 – 7 araignées placées manuellement sur l'image

4 | Conclusions

Dans ce projet, nous avons implémenté deux méthodes de segmentation. La première correspond à une technique basée sur les contours, les fourmis sociales, qui a produit une bonne segmentation où la méthode a récupéré les contours d' image. Au contraire, la méthode des araignées sociales, méthode par région, a produit un traitement temporel non négligeable. A noter que les résultats de la méthode des araignées sociales sont influencés par la répartition des agents sur la matrice et le nombre de pas à faire. A travers une comparaison entre l'algorithme des araignées et celui des fourmies, nous avons mis en avant quelques inconvénients sur la méthode des araignées sociales. En particulier, nous avons vu que pour cette méthode le temps d'exécution était particulièrement long. Comme nous pouvons le voir dans les figures des araignées , la méthode des araignées sociales produit une nouvelle région construite par des pixels non soyeux. Ces pixels sont ceux de contours. Par conséquent, les araignées sociales et les fourmis sociales ont des rôles complémentaires et la fusion des deux méthodes produira une nouvelle segmentation ayant des contours précis sur les images résultantes.