TP
Techniques heuristiques

De Mathis ENGELS, Thomas BAUDUIN, Flavien CARPENTIER et Anthony BACKER

Table des matières

I. Présentation 4

- Interface d'accueil
- Menu « Import »
- Menu « Get code »
- Utilisation
  - Etape 1 - Se rendre sur la web app
  - Etape 2 - Importer de la donnée
  - Etape 3 - Choisir l'algorithme et la distance
  - Etape 4 - Lancer la visualisation
  - Etape 5 - Résultat de la visualisation
  - Etape 6 - Expérimentez!
- Remarques importantes

II. Reading the data and representing a solution 13

- Explication
- Pseudo code
- Exemple

III. Procedure for constructing an initial feasible solution 15

- Explication
- Pseudo code
- Résultat

IV. Procedure to verify if a solution is feasible or not 17

- Explication
- Pseudo code
- Exemple

V. Procedure to calculate the number of occupied seats in the solution 18

- Explication
- Pseudo code

Exemple
VI. Local search procedures 19
Explication
Pseudo code
Résultat
VII. Developing a metaheuristic algorithm 22
Explication
Pseudo code
Résultat
IIX. Rappel 26

I. Présentation

Pour ce TP, nous voulions le résoudre de manière originale et visuelle. De ce fait, nous avons développé une web app qui résout le problème MSA (Maximum Seats Allocation) visuellement.

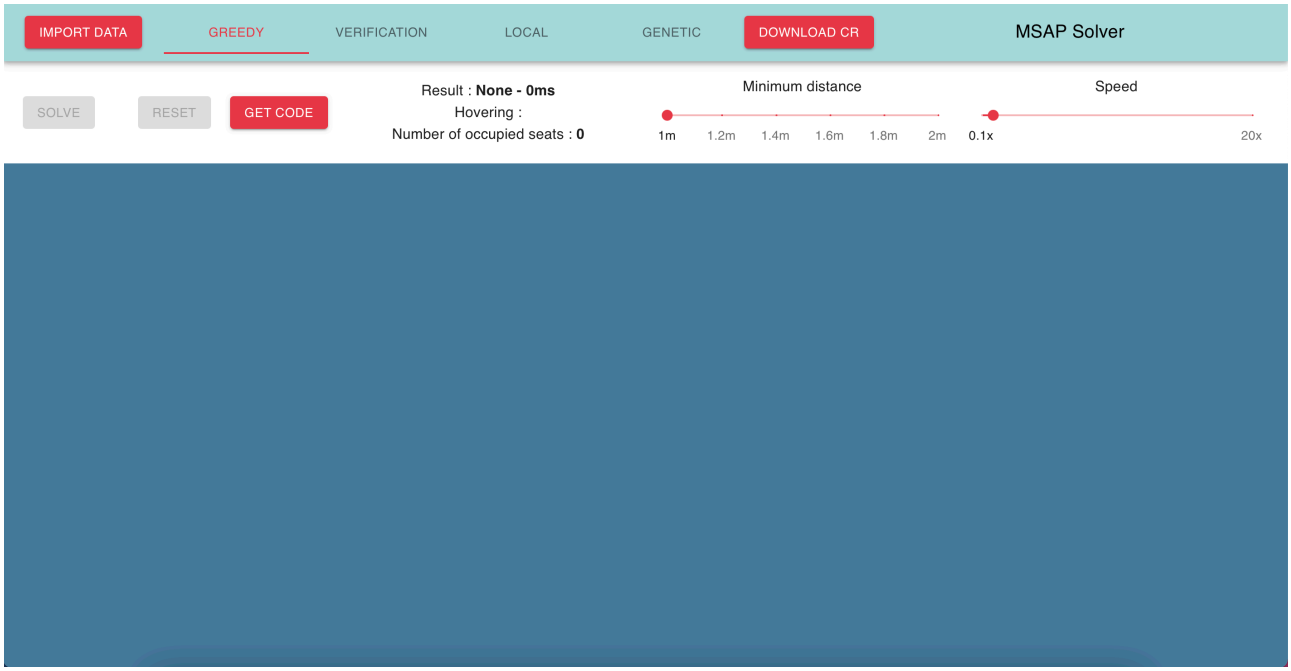
La web app est disponible sur Vercel : https://msap-solver.vercel.app

Le GitHub du projet est disponible à : https://github.com/LesCop1/comp-heur où un README.md est disponible, vous expliquant comment mettre en place la web app, sur votre ordinateur personnel.

La démo de la web app

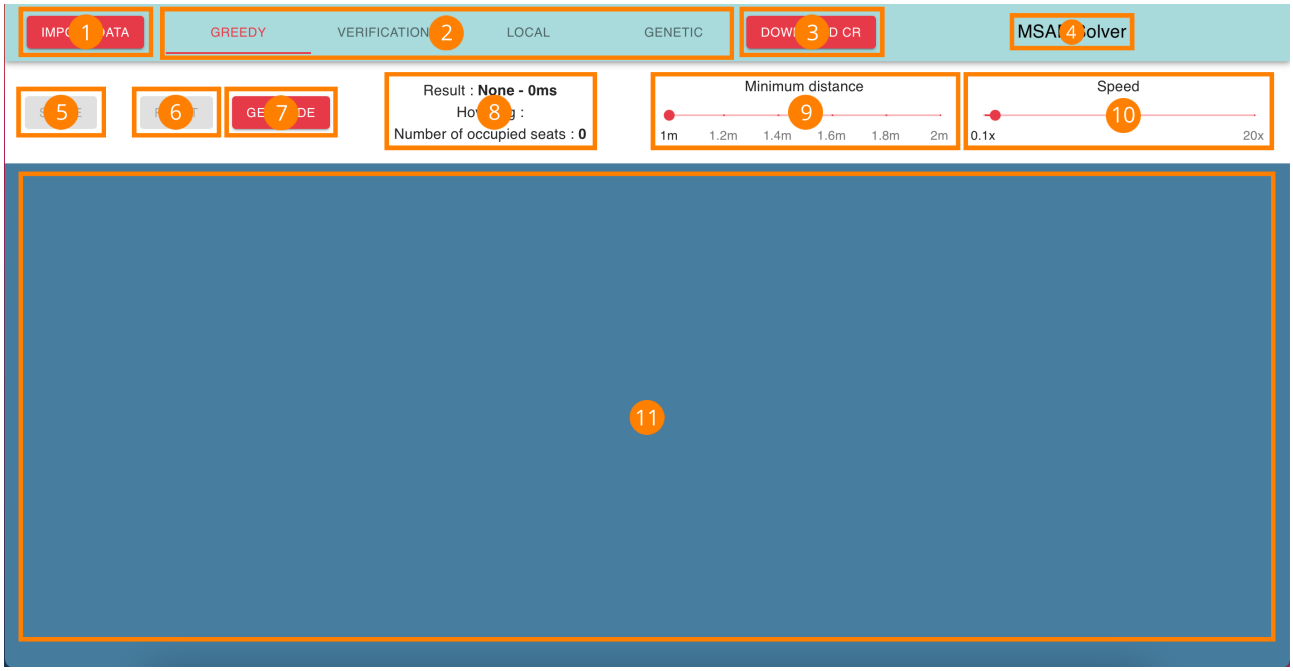
Avant de répondre aux questions, nous voulions vous présenter l'interface de la web app afin que vous puissiez l'utiliser à son plein potentiel.

Une fois que vous accéder à la web app, vous devriez voir ceci :



Décortiquons l'interface ensemble.

Interface d’accueil

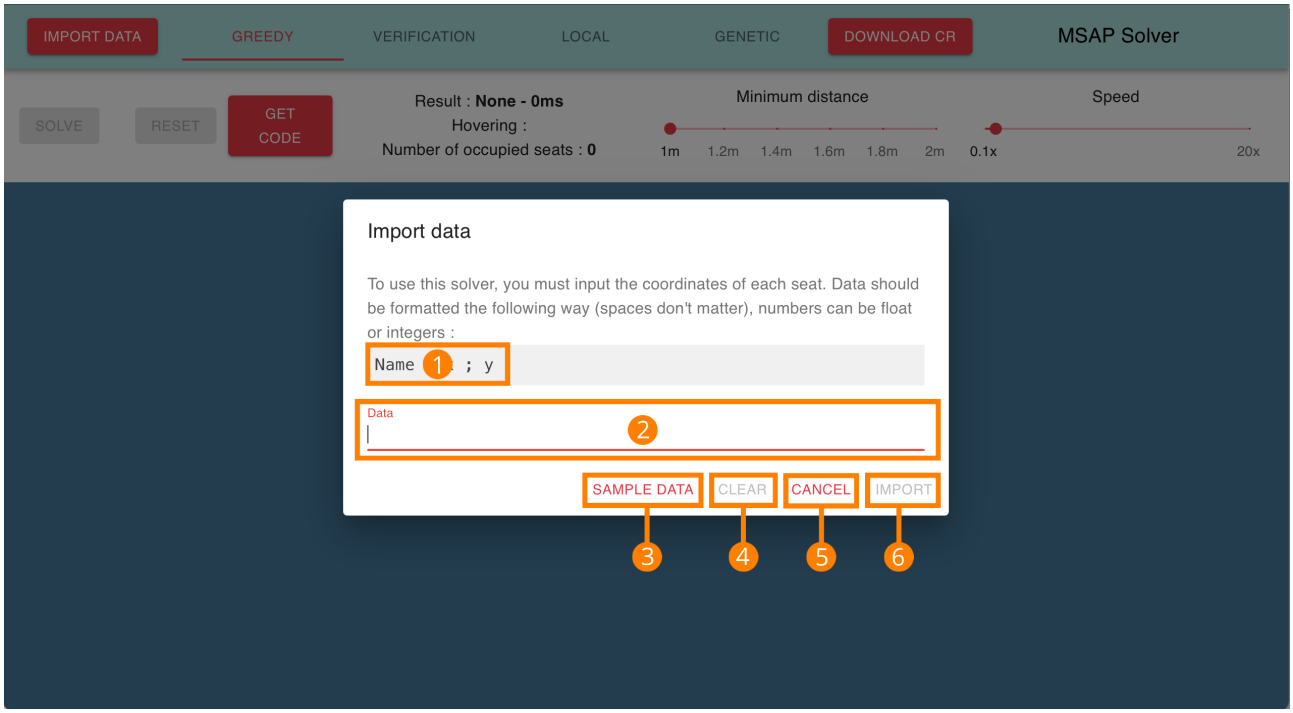


1 - Accès au menu « Import » / Supprimer la data. *
2 - Accès aux différents algorithmes. *
3 - Télécharger le compte rendu, celui que vous êtes en train de lire.
4 - Le nom de la web app.
5 - Lancer / Finir la visualisation de l’algorithme.
6 - Reset les couleurs des places.
7 - Accès au menu « Get code ».
8 - Données, en direct, de la simulation.
9 - Slider de la distance minimum pour l’algorithme. *
10 - Slider de la vitesse de la visualisation. Peut être modifié dynamiquement, c’est-à-dire lorsque la visualisation est en fonctionnement.
11 - Espace de visualisation des algorithmes.

* : Cliquer / Faire une modification de ses paramètres entrainera le recalcul de l’algorithme, merci d’être patient après une modification.

Passons au menu « Import ».

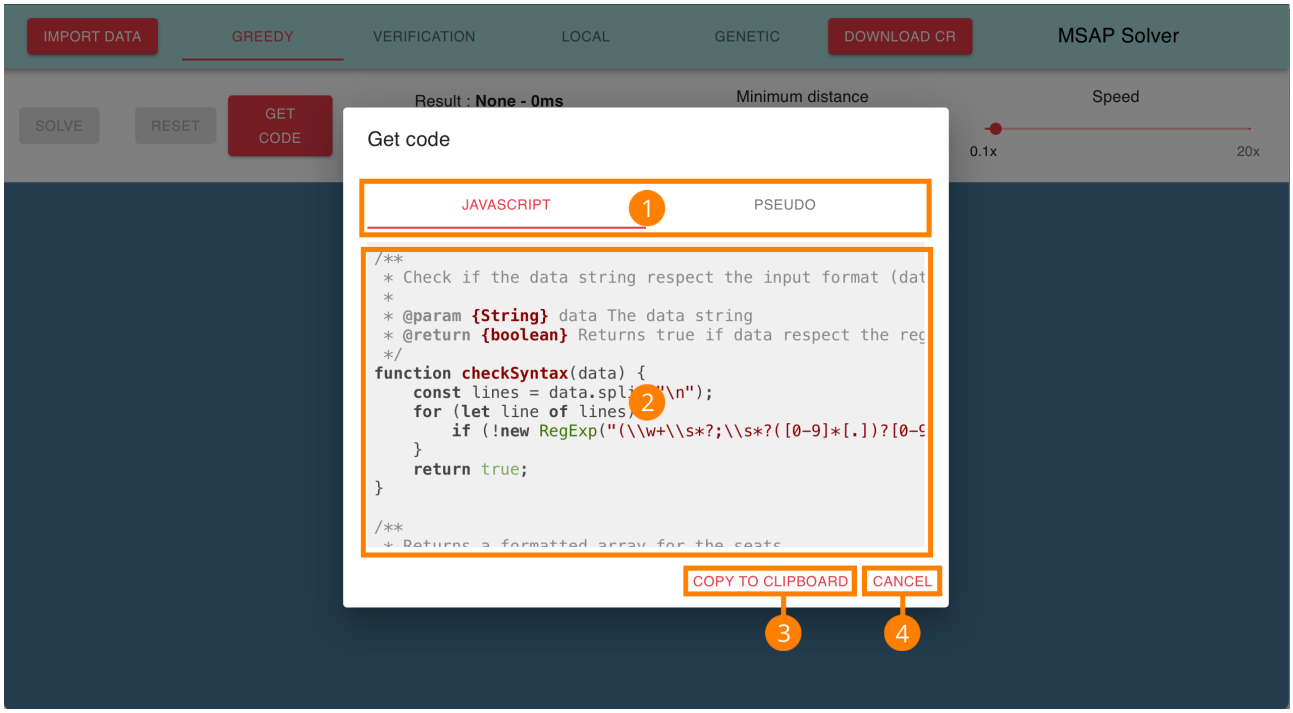
Menu « Import »



1 - Le format à respecter pour la donnée.
2 - L'espace dans lequel doit être saisi la donnée.
3 - Rentrer dans l'espace (2), une donnée d'exemple (Celle qui était disponible avec le sujet).
4 - Supprime la donnée qui est dans (2). *
5 - Ferme le menu « Import ».
6 - Importe la donnée présente dans (2). **

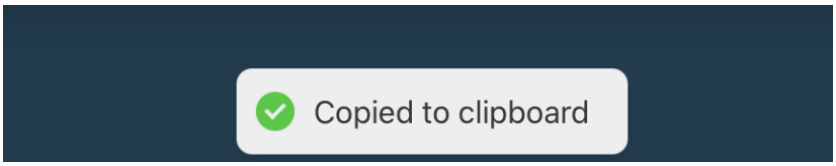
* : Le bouton sera actif seulement si il y a de la donnée dans (2).
** : Le bouton sera actif seulement si la donnée respecte le format (1).

Menu « Get code »



1 - Langage du code.
2 - Code.
3 - Copie le code dans le presse-papier. *
4 - Ferme le menu « Get code ».

* : Lorsque vous cliquez sur ce bouton, une confirmation apparait en bas de la page pour confirmer que le contenu a bien été copié.



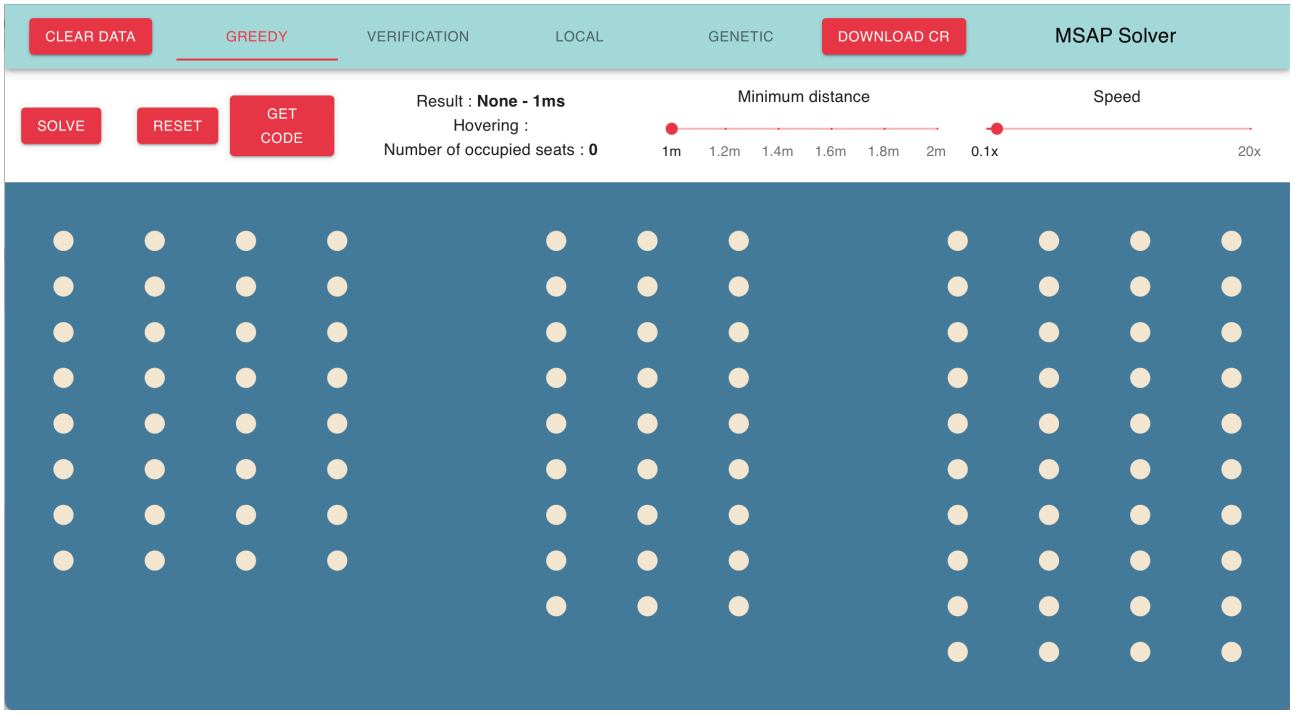
Utilisation

Etape 1 - Se rendre sur la web app

Tout premièrement, rendons-nous sur la web app, en allant sur http://localhost:3000 ou sur Vercel :

Etape 2 - Importer de la donnée

Ensuite cliquons sur le bouton « Import Data ». Lorsque le menu s'ouvre, cliquons sur « Sample Data » puis « Import ».
Vous devriez avoir ceci :



Les ronds que vous apercevez sont les places.
Les places peuvent avoir 3 couleurs différentes :

- Blanc : La place n'a pas d'état, elle n'est pas prise et ne pose pas de problème.*
- Vert : La place est prise.
- Rouge : La place ne peut pas être prise car elle est trop proche d'autre place.

* : Pour les algorithmes « Local » et « Genetic » peu de place rouge vont apparaître, nous avons fait ce choix afin de gagner en performance.

Etape 3 - Choisir l'algorithme et la distance

Il y a deux types d'algorithmes. Le premier type d'algorithme fait de la résolution alors que l'autre fait de la vérification.
Les algorithmes « Greedy », « Local » et « Genetic » font partie du premier type.
« Verification » fait partie du deuxième type.

Si vous choisissez un algorithme du premier type, il vous suffit de sélectionner celui que vous désirez avec la distance souhaitée.
Merci d'être patient lorsque vous sélectionnez ce type d'algorithme. Plus de détails dans « Remarques importantes ».

Si vous voulez utiliser l'algorithme de vérification, vous devriez rentrer les sièges pris. Lorsque vous cliquerez sur un rond (une place), elle deviendra verte. Si vous recliquez sur une place déjà verte, elle redeviendra blanche.

Pour sélectionner la distance souhaitée, faites simplement glisser le slider.

Etape 4 - Lancer la visualisation

Cliquer sur le bouton « Solve » ou « Verify » pour lancer la visualisation.

Par exemple, pour le greedy avec 1m de distance, la visualisation est :

Vous pouvez apercevoir les données en direct dans la zone (8) voir page 6.
Si vous le souhaitez, vous pouvez modifier le slider « Speed » pendant l'exécution de la visualisation afin d'augmenter la vitesse de celle-ci.

Deux nouveaux boutons sont apparus : « End » et « Stop ».
Le bouton « End » vous amène directement à la fin de la visualisation.
Le bouton « Stop » arrête la visualisation.

Etape 5 - Résultat de la visualisation

En cliquant sur le bouton « End » ou en attendant la fin de la visualisation, vous aurez le résultat de l'algorithme.

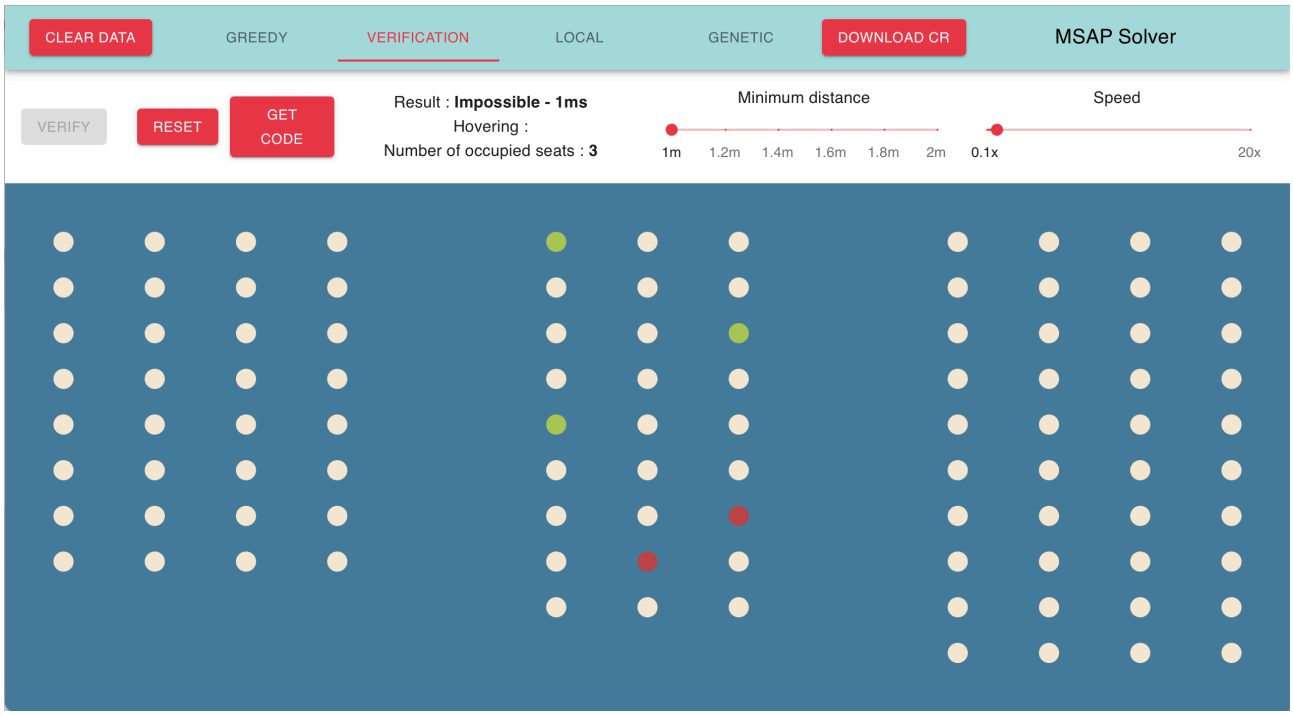
Pour « greedy » avec 1m, le résultat est :

On voit que l'algorithme a pris 1ms et qu'il y a 28 sièges occupés.

Pour « verification » avec 1m de distance on sélectionne les places « 33, 53, 37, 57, 49 ».

On remarque, qu'après avoir rentré des places, le bouton « verify » est activé. On décide de faire fonctionner la visualisation en cliquant dessus.

Malheureusement, on remarque que le résultat est « impossible » et nous avons 2 sièges en rouge. Ces sièges rentrent en conflit.



Le bouton « Verify » est maintenant bloqué, vous devez enlever tous les sièges en rouge afin de le ré-exécuter.
Afin de retirer les sièges rouges, vous devez cliquer dessus afin qu'ils deviennent blancs.
Après modification, on obtient :

Sans le siège 57, cette configuration est donc possible.

Etape 6 - Expérimentez!

A partir de ce point, vous pouvez tester chaque algorithme et vous devriez avoir la connaissance nécessaire de l'outil afin de l'utiliser à son plein potentiel.

Il reste un bouton que nous n'avons pas parlé, le bouton « Reset ». Il permet d'effacer toutes les couleurs et de remettre tous les sièges en blanc.
Ce bouton n'est pas nécessaire. A chaque fois que vous cliquez sur «Solve », «Reset » sera appelé. Ce bouton est surtout utile pour «Verification ».

Remarques importantes

Il est important de préciser que cet outil est une web app et que ses performances varient en fonction de la capacité de votre ordinateur / navigateur web.

Notre outil, codé principalement en Javascript, n'utilise pas de web-worker ce qui le réduit à **un simple thread**. Ceci engendre des **ralentissements si la complexité de l'algorithme est grande**.

De ce fait, nous vous invitons à être patient quand vous utiliser le *MSAP Solver* avec les paramètres suivants :

- « Genetic » (les calculs prennent de 1 à 4-6 secondes en fonction de la distance, du nombre de places, et la puissance de votre machine).

- « Local » (les calculs prennent de 1 à 2-3 secondes en fonction de la distance, du nombre de places, et la puissance de votre machine).

Merci d'être patient et de ne pas cliquer répétitivement sur un bouton lorsque le résultat n'est pas instantané.

Il faut comprendre que le calcul des algorithmes se réalise lorsqu'une de ses 3 conditions change :

- **Data importée**
- **Distance**
- **Algorithme**

Merci de votre compréhension.

II. Reading the data and representing a solution

Explication

Nous avons décidé de représenter la donnée différemment que dans l'exemple fourni.
Le format que nous avons adopter est le suivant : nom ; x ; y

Nous trouvons que ce format est plus adapté, il évite les erreurs de frappe et est bien structuré.

Une fois ce format respecté, la donnée est transformée en tableau d'objet tel que :
[{name: name1, x: x1, y: y1}, …]

Notre manière de représenter une solution est un tableau, contenant les index des places sélectionnées se basant sur le tableau d'entrée.
[number, number, ….]

Pseudo code

```
PROCEDURE checkSyntax(data: String): Boolean
   lines <- SEPARER PAR \n data
   POUR line DANS lines FAIRE
      SI line NE RESPECT PAS regex ALORS
         RENVOIE FAUX
      FIN SI
   FIN POUR

   RENVOIE VRAI
FIN


PROCEDURE importSeatsData(data: String): [{name: String, x: Number, y: Number}]
   SI checkSyntax(data) ALORS
      arr <- []
      lines <- SEPARER PAR \n data

      POUR line DANS lines FAIRE
         lineData <- line SEPARER PAR ; line
         newSeat <- {
            name <- lineData[0]
            x <- lineData[1]
            y <- lineData[2]
         }
         newSeat AJOUTER A arr
      FIN POUR

      RENVOIE arr
   FIN SI

   RENVOIE []
FIN
```

Exemple

Un exemple avec le string d'entrée :
thomas ; 0 ; 0
mathis ; 0 ; 75
flavien ; 0 ; 150

Renvoie le tableau suivant :
[{name: « thomas », x: 0, y: 0}, {name: « mathis », x: 0, y: 75}, {name: « flavien », x: 0, y: 150}]

Si dans cette configuration, le siège « mathis » serait sélectionné, le tableau de sortie serait :
[1]
(car dans le tableau de base, l'index 1 correspond au siège intitulé « mathis »).

III. Procedure for constructing an initial feasible solution

Explication

Pour cette procédure greedy, nous avons opté pour un algorithme semi-intelligent.

L'algorithme va regarder les précédentes places prises (et non toutes, au quel cas, l'algorithme ne serait pas du tout intelligent). Si pour toutes les places prises, la distance entre la place actuelle et les places prises est supérieure à la distance rentrée en paramètre, alors il place une nouvelle place prise.

Cet algorithme n'est pas optimisé et ne nous donne pas toujours les meilleurs résultats. Néanmoins, il reste très rapide.

Pseudo code

```
PROCEDURE isInRange(a: {name: String, x: Number, y: Number}, b: {name: String, x: Number, y: Number}, d: Number): Boolean
   RENVOIE RACINE_CARRE(PUISSANCE(a.x - b.x, 2) + PUISSANCE(a.y - b.y, 2)) < d
FIN


PROCEDURE greedy(seats: [{name: String, x: Number, y: Number}, …], distance: Number): [Number]
   seatsTaken <- []
   POUR i ALLANT DE 0 A TAILLE(seats) FAIRE
      noSeatsTakenInRange <- VRAI

      POUR j DANS seatsTaken FAIRE
         SI isInRange(seats[i], seats[j], distance) ALORS
            noSeatsInRange <- FAUX
            SORTIR POUR
         FIN SI
      FIN POUR

      SI noSeatsTakenInRange ALORS
         i AJOUTER A seatsTaken
      FIN SI
   FIN POUR

   RENVOIE seatsTaken
FIN


PROCEDURE main(dataString: [{name: String, x: Number, y: Number}, …], distance: Number): {time: Number, result: [Number], numberOfOccupiedSeats: Number}
   formattedString <- importSeatsData(dataString)
   timeStart <- TEMPS_MAINTENANT
   result <- greedy(formattedString, distance)
   RENVOIE {
      time <- TEMPS_MAINTENANT - timeStart
      result <- result
      numberOfOccupiedSeats <- TAILLE(result)
   }
FIN
```

Résultat

Les résultats sont, pour les données fournies :

- 1m : 28 places
- 1.2m : 27 places
- 1.4m : 22 places
- 1.6m : 16 places
- 1.8m : 11 places
- 2m : 10 places

Avec moins 1ms par calcul.

IV. Procedure to verify if a solution is feasible or not

Explication

Pour vérifier si une solution est bonne, l'algorithme va regarder toutes les places prises et va s'assurer que la distance entre celles-ci est respectée.

Pseudo code

```
PROCEDURE isInRange(a: {name: String, x: Number, y: Number}, b: {name: String, x: Number, y: Number}, d: Number): Boolean
    RENVOIE RACINE_CARRE(PUISSANCE(a.x - b.x, 2) + PUISSANCE(a.y - b.y, 2)) < d
FIN


PROCEDURE VERIFY(seats: [{name: String, x: Number, y: Number}, …], takenSeats: [Number], distance: Number): Boolean
    POUR i ALLANT DE 0 A TAILLE(takenSeats) - 1 FAIRE
        POUR j ALLANT DE i + 1 A TAILLE(takenSeats) FAIRE
            SI isInRange(seats[takenSeats[i]], seats[takenSeats[j]], distance) ALORS
                RENVOIE FAUX
            FIN SI
        FIN POUR
    FIN POUR

    RENVOIE VRAI
FIN


PROCEDURE main(dataString: [{name: String, x: Number, y: Number}, …], takenSeat: [Number], distance: Number): {time: Number, result: [Number]}
    formattedString <- importSeatsData(dataString)
    timeStart <- TEMPS_MAINTENANT
    result <- verify(formattedString, takenSeat, distance)
    RENVOIE {
        time <- TEMPS_MAINTENANT - timeStart
        result <- result
    }
FIN
```

Exemple

Un exemple avec le string d'entrée :
1 ; 0 ; 0
2 ; 0 ; 75
3 ; 0 ; 150

Et les places 1 et 3 sélectionnées, l'algorithme renvoie vrai.

V. Procedure to calculate the number of occupied seats in the solution

Explication

Pour calculer le nombre de place(s) occupée(s) dans la solution, notre manière de représenter nous avantage.

Il suffit de calculer la taille du tableau de sortie.

Pseudo code

```
PROCEDURE numberOfOccupiedSeats(array: [Number]): Number
    RENVOIE TAILLE(array)
FIN
```

Exemple

Si on a le tableau de sortie :
[1, 4, 6, 7]

La réponse est 4.

VI. Local search procedures

Explication

Nous avons décidé de développer un algorithme LS Swap.

Cet algorithme est aléatoire donc les résultats peuvent varier, il reste celui qui trouve les meilleures solutions, le plus rapidement.
Dans la web app, l'algorithme a les paramètres suivants : maxTime=500, maxIteration=2000

L'algorithme boucle jusqu'à ce qu'il atteigne un temps maximum ou un nombre d'itérations maximum. Avant de boucler, l'algorithme va prendre une solution initiale, ici une solution de l'algorithme « greedy ».
Puis dans la boucle, il va choisir aléatoirement une place prise à enlever et une place à rajouter.
Lorsqu'il rajoute cette place, il va regarder si celle-ci est trop proche des autres places prises. Si une place est trop proche de la nouvelle place prise, l'algorithme supprime l'ancienne place prise.

Une fois cette vérification / suppression de place trop proche, l'algorithme va chercher pour une place qui aurait la possibilité d'être prise, si c'est le cas, elle sera prise.

On compare si cette nouvelle solution est meilleur que la précédente solution (par exemple, lors de la première itération, on compare au greedy).
Et si celle-ci est meilleure, on l'a remplace et on continue de boucler.

Pseudo code

```
PROCEDURE isInRange(a: {name: String, x: Number, y: Number}, b: {name: String, x: Number, y: Number}, d: Number): Boolean
   RENVOIE RACINE_CARRE(PUISSANCE(a.x - b.x, 2) + PUISSANCE(a.y - b.y, 2)) < d
FIN


PROCEDURE isSeatAvailable(seatsData: [{name: String, x: Number, y: Number}, …], I: Number, takenSeats: [Number], distance: Number): Boolean
   POUR CHAQUE takenSeat DANS takenSeats FAIRE
      SI isInRange(seatsData[i], seatsData[takenSeat], distance)
         RENVOIE FAUX
      FIN SI
   FIN POUR

   RENVOIE VRAI
FIN


PROCEDURE greedy(seats: [{name: String, x: Number, y: Number}, …], distance: Number): [Number]
   seatsTaken <- []
   POUR i ALLANT DE 0 A TAILLE(seats) FAIRE
      noSeatsTakenInRange <- VRAI

      POUR j DANS seatsTaken FAIRE
```

```
            SI isInRange(seats[i], seats[j], distance) ALORS
                noSeatsInRange <- FAUX
                SORTIR POUR
            FIN SI
        FIN POUR

        SI noSeatsTakenInRange ALORS
            i AJOUTER A seatsTaken
        FIN SI
    FIN POUR

    RENVOIE seatsTaken
FIN


PROCEDURE swap(seatsData: [{name: String, x: Number, y: Number}, …], seatsTaken: [Number],
distance: Number, index: Number, newPlaceTaken: Number): [Number]
    newSeatsTaken <- seatsTaken

    ENLEVER(newSeatsTaken, 0, 1)
    AJOUTER(newSeatsTaken, newPlaceTaken)

    POUR i ALLANT DE TAILLE(newSeatsTaken) FAIRE
        SI isInRange(seatsData[newSeatsTaken[i]],
seatsData[newSeatsTaken[TAILLE(newSeatsTaken) - 1]], distance) ALORS
            ENLEVER(newSeatsTaken, i, 1)
            i <- i - 1
        FIN SI
    FIN POUR

    POUR i ALLANT DE TAILLE(seatsData) FAIRE
        SI isSeatAvailable(seatsData, i, newSeatsTaken, distance) ALORS
            AJOUTER(newSeatsTaken, i)
        FIN SI
    FIN POUR

    RENVOIE TRIE(newSeatsTaken)
FIN


PROCEDURE local(seatsData: [{name: String, x: Number, y: Number}, …], distance: Number,
maxIteration: Number, maxTime: Number): [Number]
    seatsTaken <- greedy(seatsData, distance)

    iteration <- 0
    startTime <-TEMPS_MAINTENANT

    TANT QUE (iteration <= maxIteration) && (TEMPS_MAINTENANT - startTime <= maxTime)
        indexToDelete <- PARTIE_ENTIERE(ALEATOIRE_0_1 * TAILLE(seatsTaken))
        newPlaceTaken <- TABLEAU(TAILLE(seatsData)).REMPLIR(CLE_TABLEAU()).FILTRER((val)
-> !INCLUE(seatsTaken, val))
                                [PARTIE_ENTIERE(ALEATOIRE_0_1 * (TAILLE(seatsData) -
TAILLE(seatsTaken))]

        newSeatsTaken <- swap(seatsData, seatsTaken, distance, indexToDelete, newPlaceTaken)
        SI TAILLE(newSeatsTaken) > TAILLE(seatsTaken) ALORS
            seatsTaken <- newSeatsTaken
        FIN SI

        iteration <- iteration + 1
```

```
    FIN TANT QUE

    RENVOIE seatsTaken
FIN


PROCEDURE main(dataString: [{name: String, x: Number, y: Number}, …], distance: Number):
{time: Number, result: [Number], numberOfOccupiedSeats: Number}
    formattedString <- importSeatsData(dataString)
    timeStart <- TEMPS_MAINTENANT
    result <- local(formattedString, distance, 2000, 500)
    RENVOIE {
        time <- TEMPS_MAINTENANT - timeStart
        result <- result
        numberOfOccupiedSeats <- TAILLE(result)
    }
FIN
```

Résultat

Les résultats sont, pour les données fournis :

- 1m : 28 places - 467ms
- 1.2m : 27 places - 397ms
- 1.4m : 22 places - 315ms
- 1.6m : 16 places - 232ms
- 1.8m : 14 places - 194ms
- 2m : 12 places - 167ms

VII. Developing a metaheuristic algorithm

Explication

Nous avons décidé de faire un algorithme génétique pour l'algorithme métaheuristique.

Plus cet algorithme passera de temps à calculer, plus il sera fiable.
Dans la web app, cet algorithme est programmé pour 3 essais, une population de 200, et 10 générations.

L'algorithme commence par créer, pour chaque enfant de la population, une solution de base, entièrement aléatoirement. Une première place est choisie aléatoirement, puis l'algorithme va boucler tant qu'il y a moins de 3 éléments dans la solution ou qu'il n'a pas encore atteint triesMax (nombre d'essais maximum).
Lorsqu'il boucle, l'algorithme choisi de nouveau, une place aléatoirement. Si cette place peut être placée, la place est ajoutée. Par contre, si la place ne peut pas être placée, un essai est consommé.
Une fois sorti de cette boucle, on débute l'algorithme génétique.

La boucle des générations commence ici.
On commence par calculer les meilleurs membres de la population, ils seront retenus comme les parents.
Nous créons ensuite, une nouvelle population, qui se base à 75% sur les places en commun des parents.
Puis nous faisons de la mutation sur tous les membres de cette nouvelle population.
La mutation consiste à supprimer entre 1 et 3 éléments de leurs solutions. Les chances sont:

- 100% -1 élément.
- 25% -2 éléments.
- 0.625% -3 éléments.

Une fois cette mutation appliquée, chaque membre de la population essaie d'ajouter le maximum de places possibles dans leurs solution.

Et la boucle des générations se termine ici.
On renvoie le meilleur des deux parents de la dernière génération.

Pseudo code

```
PROCEDURE isInRange(a: {name: String, x: Number, y: Number}, b: {name: String, x: Number, y: Number}, d: Number): Boolean
   RENVOIE RACINE_CARRE(PUISSANCE(a.x - b.x, 2) + PUISSANCE(a.y - b.y, 2)) < d
FIN


PROCEDURE shuffleArray(array: []): []
   POUR i ALLANT DE <- TAILLE(array) - 1 A 0 FAIRE
      j <- PARTIE_ENTIERE(ALEATOIRE_0_1 * (i + 1))
      [array[i], array[j]] <- [array[j], array[i]]
   FIN POUR
```

```
    RENVOIE array
FIN


PROCEDURE genetic(seatsData: [{name: String, x: Number, y: Number}, …], distance: Number,
maxTries: Number, populationSize: Number, generationMax: Number): [Number]
    generation <- 0
    population <- []

    bestTwo <- [NULL, NULL]

    POUR i ALLANT DE 0 A populationSize FAIRE
        freePlaces <- TABLEAU(TAILLE(seatsData)).REMPLIR(CLE_TABLEAU())
        shuffleArray(freePlaces)
        takenPlace <- [freePlaces[0]]
        ENLEVER(freePlaces, 0, 1)

        tries <- 0
        TANT QUE (tries < maxTries) || (TAILLE(takenPlace) < 3) FAIRE
            shuffleArray(freePlaces)
            randomPlace <- freePlace[0]

            inRange <- FAUX
            POUR j ALLANT DE 0 a TAILLE(takenPlace) FAIRE
                SI isInRange(seatsData[randomPlace], seatsData[takenPlace[j]], distance) ALORS
                    inRange <- VRAI
                    FIN POUR
                FIN SI
            FIN POUR

            SI inRange ALORS
                AJOUTER(takenPlace, randomPlace)
                ENLEVER(freePlaces, 0, 1)
            SINON
                tries <- tries + 1
            FIN SI
        FIN TANT QUE

        population[i] <- takenPlace
    FIN POUR

    TANT QUE generation < generationMax FAIRE
        POUR i ALLANT DE 0 A populationSize FAIRE
            POUR j ALLANT DE 0 A 2 FAIRE
                SI !bestTwo[j] || TAILLE(population[i]) > TAILLE(bestTwo[j]) ALORS
                    bestTwo[j] <- population[i]
                    FIN POUR
                FIN SI
            FIN POUR
        FIN POUR

        commonPlacesOfParents <- TABLEAU(TAILLE(seatsData)).REMPLIR(CLE_TABLEAU())
                                .FILTER((val) => INCLUE(bestTwo[0], val) && INCLUE(bestTwo[1], val))

        POUR i ALLANT DE 0 A populationSize FAIRE
            shuffleArray(commonPlacesOfParents)
            population[i] <- PRENDRE(commonPlacesOfParents, 0,
PARTIE_ENTIERE((TAILLE(commonPlacesOfParents) - 1) * 0.75))

            ENLEVER(population[i], PARTIE_ENTIERE(ALEATOIRE_0_1() * (TAILLE(population[i]) - 1)), 1)
```

```
        rng <- ALEATOIRE_0_1()
        SI rng > 0.75 ALORS
            ENLEVER(population[i], PARTIE_ENTIERE(ALEATOIRE_0_1() * (TAILLE(population[i]) - 1)),
1)
            SI rng > 0.975 ALORS
                ENLEVER(population[i], PARTIE_ENTIERE(ALEATOIRE_0_1() * (TAILLE(population[i]) -
1)), 1)
            FIN SI
        FIN SI

        firstCompute <- VRAI
        freePlaces <- []
        potentialFreeTakenPlaces <- []

        TANT QUE TAILLE(potentialFreeTakenPlaces) > 0 || firstCompute FAIRE
            SI firstCompute ALORS
                firstCompute <- FAUX
            SINON
                AJOUTER(population[i], potentialFreeTakenPlaces[0])
            FIN SI

            freePlaces <- TABLEAU(TAILLE(seatsData)).REMPLIR(CLE_TABLEAU())
                                .FILTER((val) => !INCLUE(population[i], val))

            potentialFreeTakenPlaces <- []

            POUR j ALLANT DE 0 A TAILLE(freePlaces) FAIRE
                inRange <- FAUX

                POUR k ALLANT DE 0 A TAILLE(population[i]) FAIRE
                    SI isInRange(seatsData[j], seatsData[population[i][k]], distance) ALORS
                        inRange <- VRAI
                        FIN POUR
                    FIN SI
                FIN POUR

                SI !inRange ALORS
                    AJOUTER(potentialFreeTakenPlaces, j)
                FIN SI
            FIN POUR
            shuffleArray(potentialFreeTakenPlaces)

        FIN TANT QUE
    FIN POUR

    generation <- generation + 1
  FIN TANT QUE

  RENVOIE bestTwo[0]
FIN


PROCEDURE main(dataString: [{name: String, x: Number, y: Number}, …], distance: Number):
{time: Number, result: [Number], numberOfOccupiedSeats: Number}
  formattedString <- importSeatsData(dataString)
  timeStart <- TEMPS_MAINTENANT
  result <- genetic(formattedString, distance, 3, 200, 10)
  RENVOIE {
    time <- TEMPS_MAINTENANT - timeStart
    result <- result
```

```
        numberOfOccupiedSeats <- TAILLE(result)
    }
FIN
```

Résultat

Les résultats sont, pour les données fournis :

- 1m : 27-28 places - 4047ms
- 1.2m : 25-27 places - 3902ms
- 1.4m : 20-22 places - 2940ms
- 1.6m : 15-16 places - 1598ms
- 1.8m : 14 places - 1349ms
- 2m : 11-12 places - 1062ms

IIX. Rappel

Vous pouvez consulter la web app ici : https://msap-solver.vercel.app
Ou bien l'installer sur votre machine personnelle en suivant le README.md de ce GitHub : https://github.com/LesCop1/comp-heur

Les codes sources / pseudo codes sont disponibles sur la web app avec le bouton « Get code ».