Calcul de vecteur

$A\begin{pmatrix}1\\3\end{pmatrix}, \vec{u}\begin{pmatrix}0\\-1\end{pmatrix}, B\begin{pmatrix}3\\0\end{pmatrix}, \vec{v}\begin{pmatrix}-1\\0\end{pmatrix}$

$\overrightarrow{AB} = \begin{pmatrix}3\\0\end{pmatrix} - \begin{pmatrix}1\\3\end{pmatrix} = \boxed{\begin{pmatrix}2\\-3\end{pmatrix}}$

Normalisation :

$||\overrightarrow{AB}|| = \sqrt{2^2 + (-3)^2} = \boxed{\sqrt{13}}$

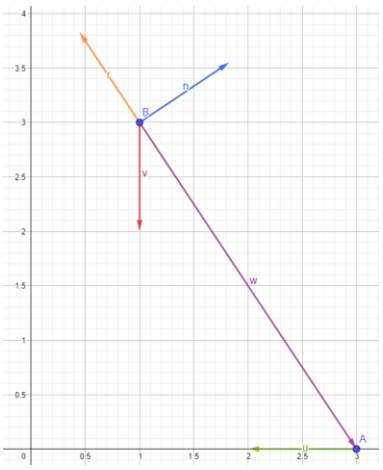
Répulsion (vecteur unitaire inverse)

$\vec{r} = -\frac{1}{\sqrt{13}}\begin{pmatrix}2\\-3\end{pmatrix} = \boxed{\begin{pmatrix}-2/\sqrt{13}\\3/\sqrt{13}\end{pmatrix}}$

Glissement (vecteur normal)

La norme d'un vecteur $\begin{pmatrix}a\\b\end{pmatrix} = \begin{pmatrix}-b\\a\end{pmatrix}$

$\vec{g} = \frac{1}{\sqrt{13}}\begin{pmatrix}3\\2\end{pmatrix} = \boxed{\begin{pmatrix}3/\sqrt{13}\\2/\sqrt{13}\end{pmatrix}}$



Choix de chemin

Difficultés

- Trouver le but rapidement
- Ne pas rester bloqué (obstacle || min. local)
- Optimiser le chemin (avoid détour inutile)

Choix possible

- ***Dijkstra*** → Explorer l'espace et essayer de trouver le «meilleur» chemin.
  - →I On ne sait pas où se trouve le but
  - →I On parcourt tout l'espace.
  - →I On trouve le meilleur chemin de cet espace

⇨ Recherche coûteuse mais résultat parfait.

- ***Best-first*** → Essayer d'atteindre le but en utilisant une heuristique
  - →I On choisit toujours le chemin qui diminue au mieux la distance avec le but.
  - →I Si blocage, on explore autour.

⇨ Recherche rapide mais pas optimal

- ***A**** → Combinaisons des deux précédents.
  - →I On applique l'heuristique de best-first.
  - →I On optimise le chemin parcourus en tenant compte de la distance à l'origine comme avec Dijkstra.

⇨ Bon compromis entre temps de recherche et qualité. (#IA)

Flocking en V

```
to adjust ;; ajuster la direction et position par rapport aux autres.
   set closest-neighbor min-one-of visible-neighbors [distance myself]
   let closest-distance distance closest-neighbor
   ;; if I am too far away from the nearest bird I can see, then try to
        get near them.
   if closest-distance > updraft-distance [
      turn-towards (towards closest-neighbor)
      set speed base-speed * (1 + speed-change-factor)
      set happy? false
      stop]
   ;; if my view is obstructed, move sideways randomly.
   if any? visible-neighbors in-cone vision-distance obstruction-cone [
      turn-at-most (random-float (max-turn * 2) - max-turn)
      set speed base-speed * (1 + speed-change-factor)
      set happy? false
      stop]
   ;; if i am too close to the nearest bird slow down.
   if closest-distance < too-close [
      set happy? false
      set speed base-speed * (1 - speed-change-factor)
      stop]
   ;; if all three conditions are filled, adjust.
   ;; to the speed and heading of my neighbor and take it easy.
   set speed [speed] of closest-neighbor
   turn-towards [heading] of closest-neighbor
   set happy? true
end
```

```
A <- agents a portee
a <- agent a portee le plus proche

Si Distance(a) < distance evitement
   eviter()
Sinon Si DistanceCentre(A) > distance cohesion
   cohesion()
Sinon
   aligner()
```

```
to flock
   find-flockmates
   if any? flockmates
   [ find-nearest-neighbor
     ifelse distance nearest-neighbor < minimum-separation
      [ separate ]
      [ align
        cohere ] ]
   avoid-obstacles ;; ce qui change.
end
```

Fuir: partir dans le sens opposé

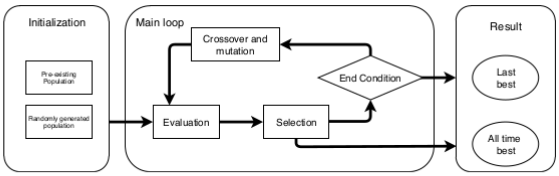
```
to avoid-obstacles
   ; avoid anything nearby that is not black
   set obstacles patches in-cone vision angle-avoidance with [pcolor !=
        black]
   if (any? obstacles)
      [flee]
end

to flee
   let obstacles-in-front obstacles in-cone 3 angle-flee
   if (any? obstacles-in-front)
   [
      rt 180
      rt random 10
      lt random 10
   ]
end
```

```
to inonde-case
   if type != but and type != obstacle[
      let case-max max-one-of neighbors [potential]
      let new-potential ([potential] of case-max) - 1
      if (potential < new-potential)[
         set potential new-potential
         ifelse potential < 0
         [set potential 0]
         [set continue true]
      ]
   ]
end

to inonder
   While continue [
      set continue false
      ask patches [inonde-case]
   ]
end
```

Algorithme génétique



- Initialisation.
- Boucle:
  - Évaluation.
  - Sélection.
  - Condition d'arrêt.
  - Génération d'une nouvelle population.
- Résultat.

0.405 = tanh((0.8*0.2) + (0.3*0.9))