**Explications et commentaires**

1. Je retourne vrai lorsque la voiture rouge est sur le point de sortir, c'est-à-dire que "pos[0] >= 4". Je construis également le *State* en gardant les informations importantes du *State* précédent.

2. La fonction *initFree* débute en initialisant la matrice avec des vrais pour dire que tout est libre. Ensuite, pour chaque voiture, je prends la position, la direction et la longueur de la voiture pour mettre les cases occupées par la voiture à faux.
   La fonction *move* vérifie pour chaque voiture d'abord la direction qu'elle utilise. Ensuite, en utilisant sa position, sa direction et sa longueur, je vérifie si la voiture peut avancer ou reculer. Si c'est possible, j'ajoute le *State* résultant dans la liste des *States*.

3. La fonction *solve* a été implémentée en utilisant l'algorithme décrite dans l'énoncé. C'est une simple recherche BFS utilisant un fil (FIFO). "La liste contient au début l'état initial, tant que la liste n'est pas vide on extrait le premier état, s'il est final on termine l'algorithme, sinon on ajoute ses fils qui n'ont pas encore été visités à la fin de la liste. On ajoute ensuite les fils à la liste *visited*." (Tiré directement de l'énoncé)

4. Le but est d'afficher dans l'ordre les étapes, c'est-à-dire les *States*, entre l'état initial et l'état final de la solution. Le *State* qu'on possède en paramètre est le *State* final. Sachant que chaque *State* possède l'information du *State* précédent, sauf pour le premier, on peut empiler les *States* dans un monceau en commençant par le dernier, le *State* qui le précède, et ainsi de suite. Au final, lorsqu'on dépilera ce monceau, les *States* qu'on retire sera dans l'ordre. Il suffit alors d'afficher le mouvement de la voiture de chaque *State* à fur et à mesure qu'on les dépile et de compter le nombre d'états.

5. Pour implémenter l'algorithme A*, comme précisé dans l'énoncé, c'est l'algorithme de *solve* qui est utilisé avec une différence : l'utilisation d'un fil avec priorité qui utilise la fonction d'estimation heuristique *f* des états.

**Comparaison des différentes heuristiques**

| | **A* avec f = nbEtatsVisites** | **A* avec fonction 1** | **A* avec fonction 2** |
|---|---|---|---|
| solve 22 | Solution A* en 4846 états vérifiés en 13040586 nanosecondes<br>ratio temps/états: 2691.046<br>46 mouvements | Solution A* en 1018 états vérifiés en 4890815 nanosecondes<br>ratio temps/états: 4804.337<br>150 mouvements | Solution A* en 838 états vérifiés en 4820439 nanosecondes<br>ratio temps/états: 5752.314<br>68 mouvements |
| solve 1 | Solution A* en 1071 états vérifiés en 2180945 nanosecondes<br>ratio temps/états: 2036.3632<br>16 mouvements | Solution A* en 245 états vérifiés en 647320 nanosecondes<br>ratio temps/états: 2642.1226<br>123 mouvements | Solution A* en 172 états vérifiés en 657583 nanosecondes<br>ratio temps/états: 3823.157<br>30 mouvements |
| solve 40 | Solution A* en 3068 états vérifiés en 5718108 nanosecondes<br>ratio temps/états: 1863.7981<br>81 mouvements | Solution A* en 1633 états vérifiés en 4384616 nanosecondes<br>ratio temps/états: 2685.0068<br>155 mouvements | Solution A* en 1427 états vérifiés en 3777250 nanosecondes<br>ratio temps/états: 2646.9866<br>141 mouvements |

On remarque qu'avec l'implémentation de la fonction 1 comme heuristique (la distance entre la voiture Rouge et la sortie), on trouve une solution beaucoup plus rapidement. L'algorithme prend plus de temps par états visités, mais visite au final beaucoup moins d'états. La solution trouvée n'est pas optimisée par contre.

Pour l'implémentation de la fonction 2 comme heuristique (la distance de la voiture rouge `a la sortie et de nombre de voitures qui lui barre le passage), on trouve des solutions plus optimisées que la fonction 1, mais tout de même pas les plus optimales. L'algorithme visite aussi moins d'états qu'avec la fonction 1. Malgré tout ça, le temps utilisé reste assez similaire, car bien qu'il y a moins d'états visités, chaque état est plus long à vérifier.

En observant le temps pour chacune des algorithmes, on déduit que plus l'estimation de l'heuristique est complexe, plus chaque état prendra de temps. Par contre, si l'algorithme permet de réduire de façon significative le nombre d'états à visiter, le temps total pour trouver une solution sera réduit.

Ainsi, avec la fonction 2, on est capable de trouver des solutions non-optimales beaucoup plus rapidement qu'une recherche naïve en largeur. Les solutions trouvées avec la fonction 2 restent plus optimales qu'avec la fonction 1 en utilisant un temps semblable.

**Bonus**

L'algorithme proposée est la suivante : prendre le nombre d'états visités (*n*) et la fonction 2 multipliée par un facteur d'importance. Ainsi, quand on est au début de l'algorithme, c'est-à-dire dans les premiers mouvements et la valeur de *n* est plus petite que la fonction 2, l'algorithme sera guidée par la fonction 2. À fur et à mesure qu'il y a d'états visités, le nombre d'états visités devient important et influence la décision de quel état visiter.

Le résultat est qu'on trouve une solution aussi optimale que lorsqu'on cherchait en largeur, tout en gardant un temps de calcul et un nombre d'états visité assez proches des valeurs avec juste la fonction 2. Ceci donne une fonction heuristique intéressante, car on obtient des solutions plus rapidement tout en restant optimal.

| | **A* avec f = nbEtatsVisites** |
|---|---|
| solve 22 | Solution A* en 1102 etats verifies en 5925207 nanoseconds<br>ratio temps/etats: 5376.776<br>46 mouvements |
| solve 1 | Solution A* en 214 etats verifies en 567778 nanoseconds<br>ratio temps/etats: 2653.1682<br>16 mouvements |
| solve 40 | Solution A* en 2003 etats verifies en 3816471 nanoseconds<br>ratio temps/etats: 1905.3774<br>81 mouvements |