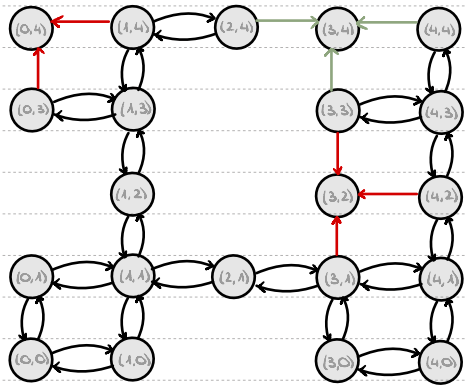
# Project - Part 1

## Exercice 1: Guided project

1. 
   - Actions: monter, descendre, aller à droite, aller à gauche (on suppose que les déplacements en diagonale ne sont pas possibles)
   - States: toute la grille est visible. L'agent voit où il se trouve, où sont les trous ainsi que les escaliers.
   - Cost/rewards: le but est de minimiser le chemin pour aller jusqu'aux escaliers, donc on choisit d'attribuer 100 points à chaque déplacement.
   - Start state: la case (0,0)
   - Goal state: la case (3,4)

2. 

3. 
```
grid_graph = {
    (0,0): [((0,1), 1), ((1,0),1 )],
    (0,1): [((0,0), 1), ((1,1), 1)],
    (0,3): [((0,4), 1), ((1,3), 1)],
    (0,4): [],
    (1,0): [((0,0), 1), ((1,1), 1)],
    (1,1): [((0,1), 1), ((1,0), 1),  ((2,1), 1), ((1,2), 1)],
    (1,2): [((1,1), 1), ((1,3), 1)],
    (1,3): [((1,2), 1), ((0,3), 1), ((1,4), 1)],
    (1,4): [((0,4), 1), ((1,3), 1), ((2,4), 1)],
    (2,1): [((1,1), 1), ((3,1), 1)],
    (2,4): [((1,4), 1), ((3,4, 1)],
    (3,1): [((2,1), 1), ((3,0), 1), ((4,1), 1), ((3,2))],
    (3,0): [((3,1), 1), ((4,0), 1)],
    (3,2): [],
    (3,3): [((3,4), 1), ((4,3), 1), ((3,2), 1)],
    (3,4): [],
    (4,0): [((3,0), 1), ((4,1), 1)],
    (4,1): [((3,1), 1), ((4,0), 1), ((4,2), 1)],
    (4,2): [((4,1), 1), ((3,2), 1), ((4,3), 1)],
    (4,3): [((4,2), 1), ((3,3), 1), ((4,4), 1)],
    (4,4): [((4,3), 1), ((3,4), 1)]
}
```

4. On va comparer les algorithmes A* et BFS
   - Pour A*, nous avons choisi d'utiliser l'heuristique « distance de Manhattan » (https://fr.wikipedia.org/wiki/Distance_de_Manhattan) qui consiste à utiliser comme heuristique: $h(n) = |x_f - x_i| + |y_f - y_i|$. Cependant, on doit un peu le modifier lorsqu'il s'agit des cases avec des trous ainsi

que certaines s'y trouvant trop proches. Voici donc notre tableau:

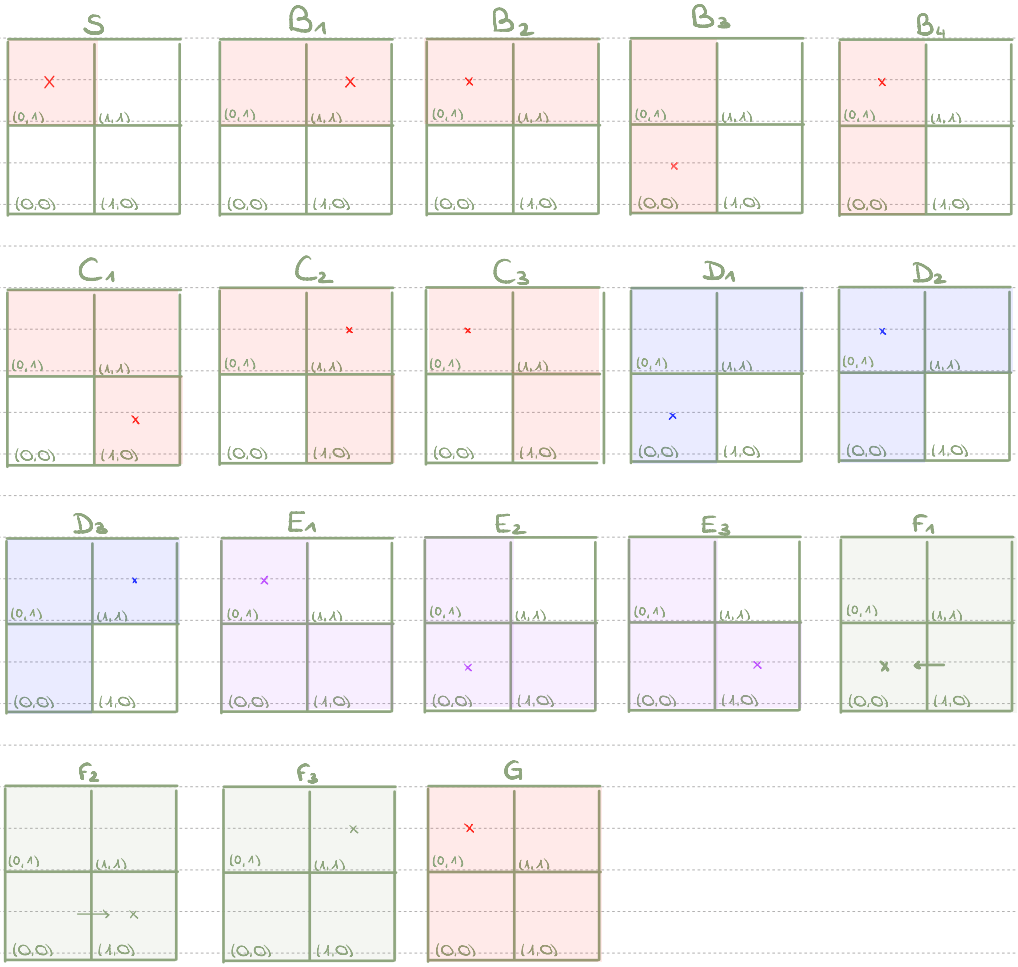
| Cases | Coût | Heuristique |
|---|---|---|
| (0,0) | 100 | 7 |
| (0,1) | 100 | 6 |
| (0,3) | 100 | 20 |
| (0,4) | 100 | 100 |
| (1,0) | 100 | 6 |
| (1,1) | 100 | 5 |
| (1,2) | 100 | 4 |
| (1,3) | 100 | 3 |
| (1,4) | 100 | 2 |
| (2,1) | 100 | 4 |
| (2,4) | 100 | 1 |
| (3,0) | 100 | 4 |
| (3,1) | 100 | 20 |
| (3,2) | 100 | 100 |
| (3,3) | 100 | 1 |
| (3,4) | 100 | 0 |
| (4,0) | 100 | 5 |
| (4,1) | 100 | 4 |
| (4,2) | 100 | 20 |
| (4,3) | 100 | 2 |
| (4,4) | 100 | 1 |

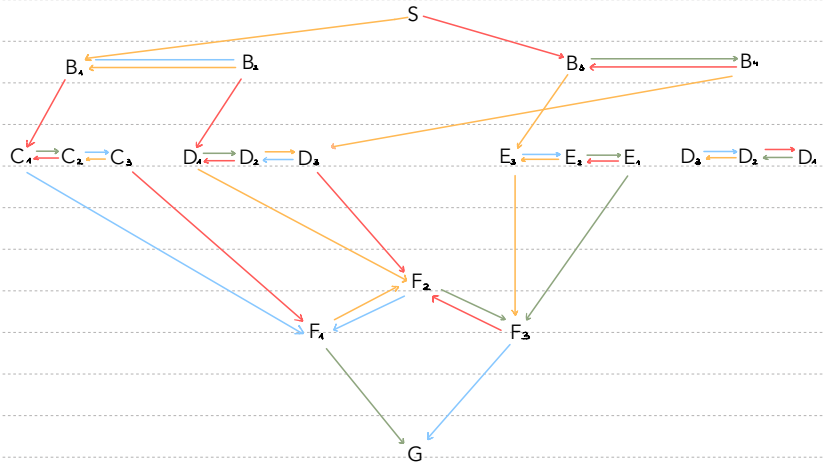
- Noeuds visités:
  A* : (0,0), (0,1), (0,3), (1,0), (1,1), (1,2), (1,3), (1,4), (2,1), (2,4), (3,2), (3,3), (3,4), (4,2), (4,3), (4,4)
  Total: 16 noeuds visités
  BFS: (0,0), (0,1), (0,3), (0,4), (1,0), (1,1), (1,2), (1,3), (1,4), (2,1), (2,4), (3,0), (3,1), (3,2), (3,4), (4,0), (4,1), (4,2)
  Total: 18 noeuds visités
  On remarque donc que l'algorithme A* visite moins de noeuds que BFS avant de trouver un chemin entre le départ et les escaliers.

## Exercice 2: Personal project

1. 
   - Actions: monter, descendre, aller à droite, aller à gauche (on suppose que les déplacements en diagonale ne sont pas possibles)
   - States: toute la grille est visible. L'agent voit où il se trouve, où sont les trous ainsi que les escaliers.
   - Cost/rewards: le but est de tondre entièrement la pelouse et de revenir à sa base une fois terminé, en faisant le chemin le plus court possible. Un déplacement équivaut à 4 points.
   - Start state: la case (0,1)
     Goal state: la case (0,1)

2.





3.

```
grid_graph = {
    (0, 1, 1, 0, 0, 0): [((1, 1, 1, 1, 0, 0), 4), ((0, 0, 1, 0, 1, 0), 4)],   # S qui est envoyé sur B1, B3
    (1, 1, 1, 1, 0, 0): [((0, 1, 1, 1, 0, 0), 4), ((1, 0, 1, 1, 0, 1), 4)],   # B1 qui est envoyé sur B2, C1
    (0, 1, 1, 1, 0, 0): [((1, 1, 1, 1, 0, 0), 4), ((0, 0, 1, 1, 1, 0), 4)],   # B2 qui est envoyé sur B1, D1
    (0, 0, 1, 0, 1, 0): [((0, 1, 1, 0, 1, 0), 4), ((1, 0, 1, 0, 1, 1), 4)],   # B3 qui est envoyé sur B4, E3
    (0, 1, 1, 0, 1, 0): [((0, 1, 1, 1, 0, 0), 4), ((1, 1, 1, 1, 1, 0), 4)],   # B4 qui est envoyé sur B2, D3
    (1, 0, 1, 1, 0, 1): [((1, 1, 1, 1, 0, 1), 4), ((0, 0, 1, 1, 1, 1), 4)],   # C1 qui est envoyé sur C2, F1
    (1, 1, 1, 1, 0, 1): [((1, 0, 1, 1, 0, 1), 4), ((0, 1, 1, 1, 0, 1), 4)],   # C2 qui est envoyé sur C1, C3
    (0, 1, 1, 1, 0, 1): [((1, 1, 1, 1, 0, 1), 4), ((0, 0, 1, 1, 1, 1), 4)],   # C3 qui est envoyé sur C2, F1
    (0, 0, 1, 1, 1, 0): [((0, 1, 1, 1, 1, 0), 4), ((1, 0, 1, 1, 1, 1), 4)],   # D1 qui est envoyé sur D2, F2
    (0, 1, 1, 1, 1, 0): [((0, 0, 1, 1, 1, 0), 4), ((1, 1, 1, 1, 1, 0), 4)],   # D2 qui est envoyé sur D1, D3
    (1, 1, 1, 1, 1, 0): [((0, 1, 1, 1, 1, 0), 4), ((1, 0, 1, 1, 1, 1), 4)],   # D3 qui est envoyé sur D2, F2
    (0, 1, 1, 0, 1, 1): [((0, 0, 1, 0, 1, 1), 4), ((1, 1, 1, 1, 1, 1), 4)],   # E1 qui est envoyé sur E2, F3
    (0, 0, 1, 0, 1, 1): [((0, 1, 1, 0, 1, 1), 4), ((1, 0, 1, 0, 1, 1), 4)],   # E2 qui est envoyé sur E1, E3
    (1, 0, 1, 0, 1, 1): [((0, 0, 1, 0, 1, 1), 4), ((1, 1, 1, 1, 1, 1), 4)],   # E3 qui est envoyé sur E2, F3
    (0, 0, 1, 1, 1, 1): [((1, 0, 1, 1, 1, 1), 4), ((0, 1, 1, 1, 1, 1), 4)],   # F1 qui est envoyé sur F2, G
    (1, 0, 1, 1, 1, 1): [((0, 0, 1, 1, 1, 1), 4), ((1, 1, 1, 1, 1, 1), 4)],   # F2 qui est envoyé sur F1, F3
    (1, 1, 1, 1, 1, 1): [((1, 0, 1, 1, 1, 1), 4), ((0, 1, 1, 1, 1, 1), 4)],   # F3 qui est envoyé sur F2, G
}
```

On a utilisé la notation: (x,y,T1,T2,T3,T4) avec les T qui correspondent au 4 cases avec 1 = rasé, 0 = non

4. On va comparer les algorithmes A* et DFS
   - Pour l'algorithme A*, nous avons choisi l'heuristique suivante:

| États | Heuristique |
|---|---|
| S | 4 |
| B1 | 2 |
| B2 | 4 |
| B3 | 2 |
| B4 | 4 |
| C1 | 1 |
| C2 | 3 |
| C3 | 2 |
| D1 | 3 |
| D2 | 4 |
| D3 | 3 |
| E1 | 2 |
| E2 | 3 |
| E3 | 1 |
| F1 | 1 |
| F2 | 2 |
| F3 | 1 |
| G | 0 |

Les heuristiques ont été choisies selon le nombres de cases non tondues. De plus, elles favorisent les chemins permettant d'accéder à une surface à tondre.

   - États visités:
     A* : S, B1, B2, B3, C1, C2, C3, D1, D3, E1, E2, E3, F1, F2, F3, G
     DFS: S, B1, B2, D1, D2, D3, F1, F2, G

   On remarque que DFS explore moins de noeuds. Cependant, il ne choisi pas un chemin optimal, alors que A* le fait.