- Exercice 02 – Local Beam Seach
  - Une version modifiée de l'algorithme Best First Search
  - La sélection des nœuds se fait à base d'une probabilité conditionnelle
  - Chaque itération implique plusieurs chemins
  - Chemins ordonnés et choisis selon la longueur du chemin
  - Enchainement plus court ou moins coûteux

- Exercice 02 – Local Beam Seach
  - input du programme

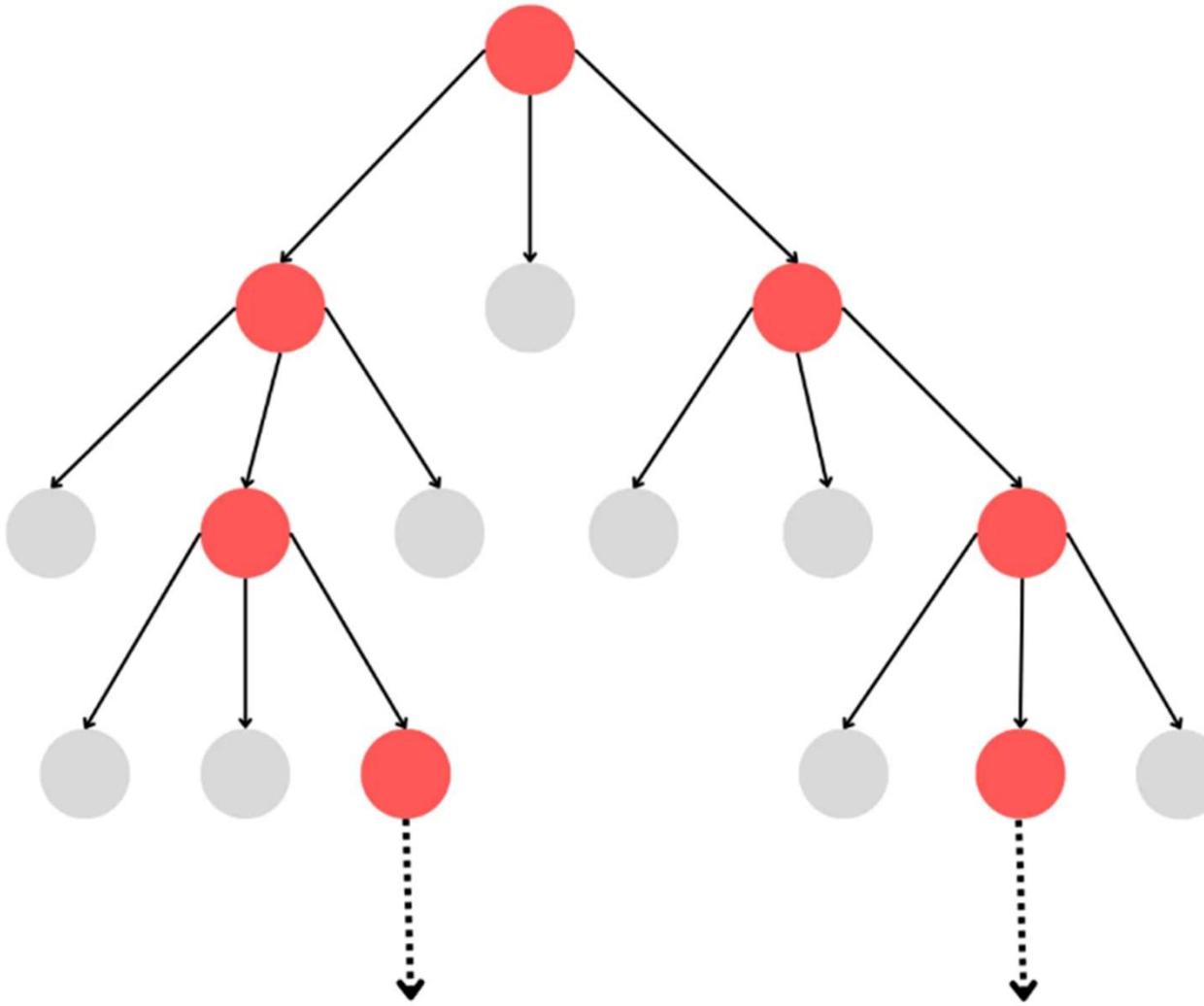
```
Input: START and GOAL states

Local Variables: OPEN, NODE, SUCCs, W_OPEN, FOUND

Output: Yes or No
```

  - Output du programme : YES / NO

- Exercice 02 – Local Beam Seach
  - Méthode

- NODE = Root_node: Found = false:
- if NODE is the goal node, then Found = true else find SUCCs of NODE. if any with its estimated cost and store in the OPEN list:
- while (FOUND false and not able to proceed further) do
  - sort OPEN list:
  - select the top W elements from the OPEN list and put them in the W_OPEN list and empty the OPEN list.
  - for each node in the W_OPEN list
    - { if NODE = goal state then FOUND = true find SUCCs of NODE. if any with its estimated cost and store in the OPEN list. }
  - ending while loop
- if FOUND = true then return Yes otherwise return No.
- Stop.

- Exercice 02 – Local Beam Seach
  - Exemple d'exécution
    - W = 2; B = 3



- Exercice 02 – Local Beam Seach
  - Ecrire un programme python pour implémenter l'algorithme LBS
  - Utilisez une fonction principale beam_search() qui itère plusieurs fois afin de trouver le plus court chemin
  - Les paramètres de cette fonction doivent être la distance entre deux nouds « distance » + la largeur du beam « berta »

- Exercice 02 – Local Beam Seach : Solution
  - Définition de la fonction beam_search

LBS.py > beam_search

```python
from numpy import array


#main function
#beta here is width of beam and distances can be considered as weights.
def beam_search(distances, beta):
    #initialising some record
    paths_so_far = [[list(), 0]]

```

- Exercice 02 – Local Beam Seach : Solution
  - Traverser à travers les nœuds voisins ligne par ligne

LBS.py > ...

```python
#traverse through the neighbouring vertices row by row.
for idx, tier in enumerate(distances):
    if idx > 0:
        print(f'Paths kept after tier {idx-1}:')
        print(*paths_so_far, sep='\n')
    paths_at_tier = list()


    for i in range(len(paths_so_far)):
        path, distance = paths_so_far[i]

        # Extending the paths
        for j in range(len(tier)):
            path_extended = [path + [j], distance + tier[j]]
            paths_at_tier.append(path_extended)

    paths_ordered = sorted(paths_at_tier, key=lambda element: element[1])

    # The best paths are saved
    paths_so_far = paths_ordered[:beta]
    print(f'\nPaths reduced to after tier {idx}: ')
    print(*paths_ordered[beta:], sep='\n')
return paths_so_far
```

- Exercice 02 – Local Beam Seach : Solution
  - Définir la matrice de distance et déterminer les meilleurs chemins

LBS.py > beam_search

```
35
36    #Distance matrix
37    dists = [[1, 4, 6, 8],
38             [5, 2, 3, 4]]
39    dists = array(dists)
40
41    # Calculating the best paths
42    best_paths = beam_search(dists, 2)
43    print('\nThe best paths:')
44    for beta_path in best_paths:
45        print(beta_path)
```

- Exercice 02 – Local Beam Seach : Solution
  - Résultat d'exécution

```
PS C:\Users\KHALED\Strategies> & C:/Users/KHALED/AppData/Local/Programs/Python/Python39/python.exe c:/Users/KHALED/Strategies/LB
S.py

Paths reduced to after tier 0:
[[2], 6]
[[3], 8]
Paths kept after tier 0:
[[0], 1]
[[1], 4]

Paths reduced to after tier 1:
[[0, 3], 5]
[[0, 0], 6]
[[1, 1], 6]
[[1, 2], 7]
[[1, 3], 8]
[[1, 0], 9]

The best paths:
[[0, 1], 3]
[[0, 2], 4]
PS C:\Users\KHALED\Strategies>
```