Fonction 1 Algorithme de séparation et évaluation progressive (*branch and bound*)

```
1:  function BRANCH_AND_BOUND(graphe)                                   ▷ O(n^3 · Σ_{i=1}^{n} i!)
2:      pile ← ∅
3:      nombre_obstructions_min ← ∞

4:      voisins_triés ← ∅
5:      for all noeud ∈ graphe do          ▷ n itérations pour n noeuds ⇒ Θ(n^2 log n)
6:          voisins_triés ← {voisins(noeud) triés par hauteur croissante}  ▷ Θ(n log n)
7:      end for

8:      while pile ≠ ∅ do
                  ▷ nombre d'itérations pour n noeuds ≤ Σ_{i=1}^{n} i! ⇒ O(n^3 · Σ_{i=1}^{n} i!)
9:          chemin ← dépiler(pile)                                      ▷ Θ(1)

10:         if longueur(chemin) ≥ nombre_obstructions_min then
11:             nombre_obstructions ← COMPTER_OBSTRUCTIONS(chemin)       ▷ Θ(n)
12:             if nombre_obstructions ≥ nombre_obstructions_min then    ▷ Θ(1)
13:                 continue
14:             end if

15:             if longueur(chemin) = |graphe| then                      ▷ Θ(1)
16:                 afficher solution                                    ▷ Θ(n)
17:                 nombre_obstructions_min ← nombre_obstructions
18:                 continue
19:             end if
20:         end if

21:         nouveaux_chemins ← ÉTENDRE_CHEMIN(chemin)                    ▷ O(n^3)
22:         empiler(pile, nouveaux_chemins)     ▷ |nouveaux_chemins| < n ⇒ O(n)
23:     end while
24: end function
```

Fonction 2 Extension du chemin partiel en lui ajoutant un nouveau noeud

```
1: function ÉTENDRE_CHEMIN(chemin : [noeuds], voisins_triés : map)        ▷ O(n^3)
2:     nouveaux_chemins ← ∅

3:     voisins ← voisins_triés[dernier sommet de chemin]                  ▷ Θ(1)
4:     for all voisin ∈ voisins \ chemin do                  ▷ |voisins| < n ⇒ O(n^3)
5:         a_un_voisin_condamné ← false

6:         if ∃ ≥ 2 noeuds ∉ chemin then
7:             for all voisin_degré_2 ∈ voisins(voisin) \ chemin do
                                                  ▷ | voisins(voisin) | < n ⇒ O(n^2)
8:                 voisins_deg3_non_visités ← 0
9:                 for all voisin_degré_3 ∈ voisins(voisin_degré_2) \ chemin do
                                              ▷ | voisins(voisin_degré_2) | < n ⇒ O(n)
10:                    voisins_deg3_non_visités ← voisins_deg3_non_visités + 1

11:                    if voisins_deg3_non_visités > 1 then               ▷ Θ(1)
12:                        break
13:                    end if
14:                end for

15:                if voisins_deg3_non_visités ≤ 1 then                   ▷ Θ(1)
16:                    a_un_voisin_condamné ← true
17:                    break
18:                end if

19:            end for
20:        end if

21:        if ¬a_un_voisin_condamné then
22:            nouveaux_chemins ← nouveaux_chemins ∪ {(chemin ∪ voisin)}
                                                  ▷ |chemin| + 1 < n ⇒ O(n)
23:        end if
24:    end for

25:    return nouveaux_chemins
26: end function
```

Fonction 3 Comptage du nombre d'obstructions dans une solution

```
1:  function COMPTER_OBSTRUCTIONS(chemin : [noeuds])                    ▷ Θ(n)
2:      nombre_obstructions ← 0
3:      hauteur_max ← 0
4:      for all noeud ∈ chemin do          ▷ n itérations pour n noeuds ⇒ Θ(n)
5:          if hauteur(noeud) < hauteur_max then          ▷ op. baromètre : Θ(1)
6:              nombre_obstructions ← nombre_obstructions + 1
7:          else
8:              hauteur_max ← hauteur(noeud)
9:          end if
10:     end for
11:     return nombre_obstructions
12: end function
```