IA01 : RAPPORT TP02
LA GRANDE TRAVERSEE

Livrables attendus : un fichier lisp + un rapport présentant et argumentant le code lisp proposé.

Date de remise : 15 novembre 2020

Problématique :

Un homme, un loup, une chèvre et un chou doivent traverser une rivière. Ils sont actuellement sur la rive gauche et souhaitent donc aller sur la rive droite. A leurs pieds est amarrée une barque à deux places. L'homme étant seul à savoir ramer sera évidemment de tous les voyages. Cependant, lorsqu'il n'est pas présent sur une rive, les appétits se déchaînent et le loup a tendance à manger la chèvre, laquelle se laisserait facilement aller à manger le chou.

Objectif :

L'objectif du TP est de traiter informatiquement ce problème comme une recherche dans un espace d'états :

1. Définition et représentation des états du problème.

Dans notre problème un état sera décrit par la position de l'homme, de la chèvre, du chou et du loup. C'est-à-dire s'ils sont sur la rive gauche ou sur la rive droite.

Le personnage peut être sur la rive gauche ou sur la rive droite ; s'il n'est sur aucune des deux cela implique qu'il est dans la barque.

On représente un état par la connaissance de sa présence sur la rive droite (état final), la présence d'un protagoniste sur la rive gauche sera codée par un 1 et son absence par un -1.

Ensemble des états possibles :

Etats	Homme	Chou	Chèvre	Loup
E0	1	1	1	1
E1	1	1	-1	1
E3	1	-1	1	1
E3	1	-1	1	-1
E4	-1	-1	-1	-1
E5	-1	-1	-1	1
E6	-1	1	-1	-1
E7	-1	-1	1	-1
E8	1	1	1	-1

E0 : tous les protagonistes sont sur la rive gauche.

E1 : l'homme fait passer la chèvre sur la rive droite, le loup et le chou cohabitent paisiblement sur la rive gauche.

E2 : l'homme fait passer le chou sur la rive droite, et reprend la chèvre pour ne pas la laisser avec les choux, pendant ce temps le loup est seul sur la rive gauche.

E3 : l'homme dépose la chèvre sur la rive gauche et prend le loup à sa place, il dépose le loup sur la rive droite avec le chou. Pendant ce temps, la chèvre est seule sur la rive gauche. Le loup et le chou cohabitent sur la rive droite.

E4 : l'homme passe sur la rive droite avec la chèvre, tous cohabitent et ceci est la fin du problème.

Voici, une représentation grâce à une liste :

(((HOMME 1)(CHOU 1)(CHEVRE 1)(LOUP 1))

((HOMME 1)(CHOU 1)(CHEVRE -1)(LOUP 1))

((HOMME 1)(CHOU -1)(CHEVRE 1)(LOUP 1))

((HOMME 1)(CHOU -1)(CHEVRE 1)(LOUP -1))

((HOMME -1)(CHOU -1)(CHEVRE -1)(LOUP -1))

((HOMME -1)(CHOU -1)(CHEVRE -1)(LOUP 1))

((HOMME -1)(CHOU 1)(CHEVRE -1)(LOUP 1))

((HOMME -1)(CHOU -1)(CHEVRE 1)(LOUP -1)))

On a donc d'une manière simplifiée :

```
(setq *etats* '((1 1 1 1)(1 1 -1 1)(1 -1 1 1)(1 -1 1 -1)
                (-1 -1 -1 -1)(-1 -1 -1 1)(-1 1 -1 1)
                (-1 -1 1 -1)(-1 1 -1 -1)(1 1 1 -1) ))
```

On peut ici faire usage d'une liste globale car il s'agit de la définition du problème.

2. Etats initial et final, états interdits.

La configuration de l'état initial est la suivante : tous les protagonistes sont sur la rive gauche : il s'agit de l'état E0 ci-dessus.

La configuration de l'état final est la suivante : le loup, la chèvre, et le chou sont sur la rive droite ainsi que l'homme. C'est le but de notre problème : que tous les personnages aient traversés la rivière. Il s'agit de l'état E4 ci-dessus. d

Certains protagonistes ne peuvent être ensemble sans la présence de l'homme : si le loup et la chèvre restent ensemble, le loup mangera la chèvre ; si le chou et la chèvre restent ensemble seuls, la chèvre mangera le chou. Et, s'ils sont tous les trois sur la rive, seul le loup survivra. Cela représente les états interdits :

Homme	Chou	Chèvre	Loup
-1	1	1	-1
-1	1	1	1
-1	-1	1	1

```
(setq *etatnonvalide* '((-1 1 1 -1)(-1 1 1 1)(-1 -1 1 1)))
```

3. Actions permettant de passer d'un état à l'autre.

Une action peut être représentée par une liste (nh nchou nchevre nloup) où nh nchou nchevre et nloup sont respectivement le nombre de de protagoniste à retirer ou à ajouter de la rive gauche à la suite d'une traversée.

Par exemple, la traversée de l'homme avec la chèvre est représentée par (2 0 2 0). Les protagonistes seront retirés de l'état si on passe de la rive gauche à la rive droite, et ajoutés si on passe de la rive droite à la rive gauche.

Par exemple, depuis l'état (1 1 1 1) on applique l'action (2 0 2 0), l'état obtenu sera

(1-2 1-0 1-2 1-0)=(-1 1 -1 1) Ce qui représente le faite que l'homme et la chèvre sont passés de la rive gauche à la rive droite.

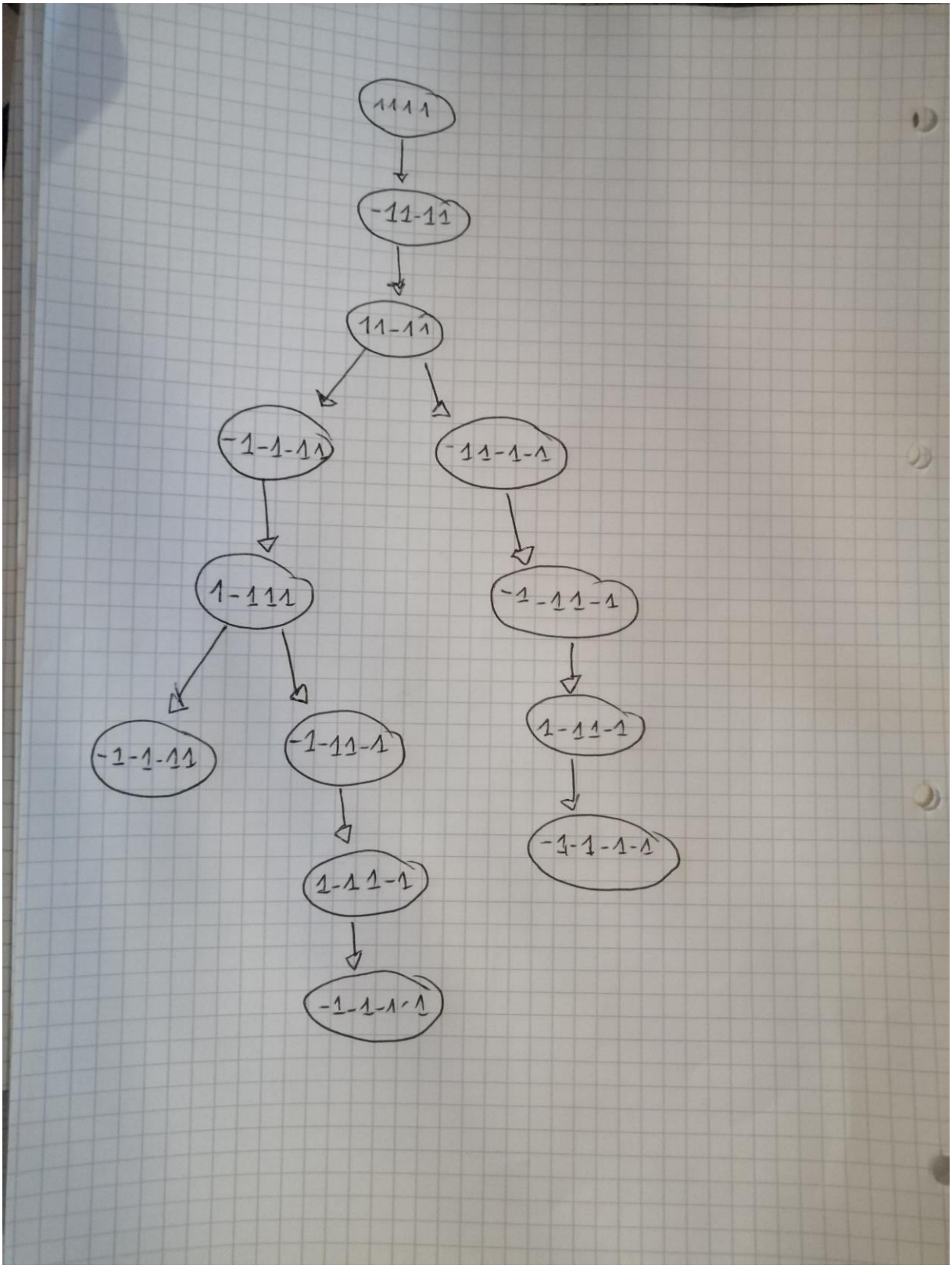
Cette représentation est suffisante puisque si l'homme est sur la rive gauche, cela veut dire qu'on passe de la rive gauche à la rive droite, et si l'homme n'est pas présent sur la rive gauche cela veut dire qu'on pas de la rive droite à la rive gauche.

Actions possibles :

	nh	nchou	nchevre	nloup
A1	2	2	0	0
A2	2	0	2	0
A3	2	0	0	2
A4	2	0	0	0
A5	2	2	0	0

```
(setq *actions* '((2 0 0 0) (2 0 2 0) (2 0 0 2) (2 0 0 0) (2 2 0 0)))
```

4. Graphe de l'espace d'état



Je ne représente sur le graphe, que les états qui n'ont jamais été visités.

5. Fonction Lisp `successeurs(etat)` qui retourne la liste des successeurs d'un état.

Déclaration de la liste globale des états possibles :

```
(setq *etats* '((1 1 1 1)(1 1 -1 1)(1 -1 1 1)(1 -1 1 -1)
            (-1 -1 -1 -1)(-1 -1 -1 1)(-1 1 -1 1)
            (-1 -1 1 -1)(-1 1 -1 -1)(1 1 1 -1)
            ))
```

La liste peut être globale puisqu'il s'agit de la modélisation du problème.

Déclaration de la liste globale des actions possible :

```
(setq *actions* '((2 0 0 0) (2 0 2 0) (2 0 0 2) (2 0 0 0) (2 2 0 0)))
```

De même, la liste des actions possible fait partie de la modélisation du problème.

Définition de la fonction successeurs qui renvoie les successeurs possibles d'un état.

```
(defun successeurs(etat)
 (let ((succ (list))(res nil))
     (dolist (a *actions*)
       (setq res (calcul etat a))
       (if (member res *etats* :test #'equal)
          (progn (if (not(member res succ :test #'equal))(push res succ))
            (if (not(member (list res etat) *predecesseur* :test #'equal))
                          (push (list res etat) *predecesseur*))
        )
       )
       )
   (return-from successeurs (reverse succ))
  )
 )
```

Cette fonction applique chaque action à l'état passé en paramètre, si l'état résultant est possible (il appartient donc à la liste d'état) alors on l'ajoute à la liste des successeurs de l'état courant. Pour les besoins de la recherche en profondeur, on construit la liste prédécesseur afin de pouvoir avoir une trace des liens de parenté entre le nœud courant et ses successeurs.

Voici la fonction calcul utilisé ci-dessus :

```
(defun calcul(etat a)
 (if (equal (car etat) 1)
    (list(-(car etat)(car a))(-(cadr etat)(cadr a))(-(caddr etat)(caddr a))(-(cadddr etat)(cadddr a)))
  (list(+(car etat)(car a))(+(cadr etat)(cadr a))(+(caddr etat)(caddr a))(+(cadddr etat)(cadddr a)))
  )
 )
```

Cette fonction renvoie le résultat du calcul quand on applique l'action a à un état e. Si l'homme est sur la rive gauche : (car etat) est à 1 alors on fait une soustraction, sinon on fait une addition. Cela représente le fait que l'on passe de la rive droite à la rive gauche et vice versa.

La liste prédécesseur aura une utilité dans la suite.

6. Ecrire et tester la fonction Lisp `rech-prof(etat)` qui explore l'espace d'états en profondeur d'abord et affiche toutes les solutions possibles. Dessiner l'arbre de recherche correspondant.

Première version – donne un résultat correct mais peu concluant

```
(defun rech-prof(etat etat-visite)
 (if (equal etat '(-1 -1 -1 -1)) (progn (push etat etat-visite) (print(reverse etat-visite))))
  (push etat etat-visite)
 (dolist (fils (successeurs etat))
  (if (member fils etat-visite :test #'equal)
     nil
     (progn
      (rech-prof fils etat-visite)
      )
   )
  )
)
```

```
(rech-prof '(1 1 1 1) nil)
```

```
((1 1 1 1) (-1 1 -1 1) (1 1 -1 1) (-1 1 -1 -1) (1 1 1 -1)
 (-1 -1 1 -1) (1 -1 1 -1) (-1 -1 -1 -1))
((1 1 1 1) (-1 1 -1 1) (1 1 -1 1) (-1 -1 -1 1) (1 -1 1 1)
 (-1 -1 1 -1) (1 -1 1 -1) (-1 -1 -1 -1))
```

On peut garder cette version qui est parfaitement correcte.

Seconde Version avec un meilleur affichage

```
(defun rech-prof(etat etat-visite)
 (if (equal etat '(-1 -1 -1 -1)) (progn (push etat etat-visite) (afficher (reverse etat-visite))))
  (push etat etat-visite)
 (dolist (fils (successeurs etat))
  (if (member fils etat-visite :test #'equal)
     nil
     (progn
      (rech-prof fils etat-visite)
      )
   )
  )
 )
```

```
(defun afficher (liste)
  (dolist (x liste)
   (format t "~s~%" x)
   )
 (format t "~%~%")
  )
```

```
(rech-prof '(1 1 1 -1) nil)
```

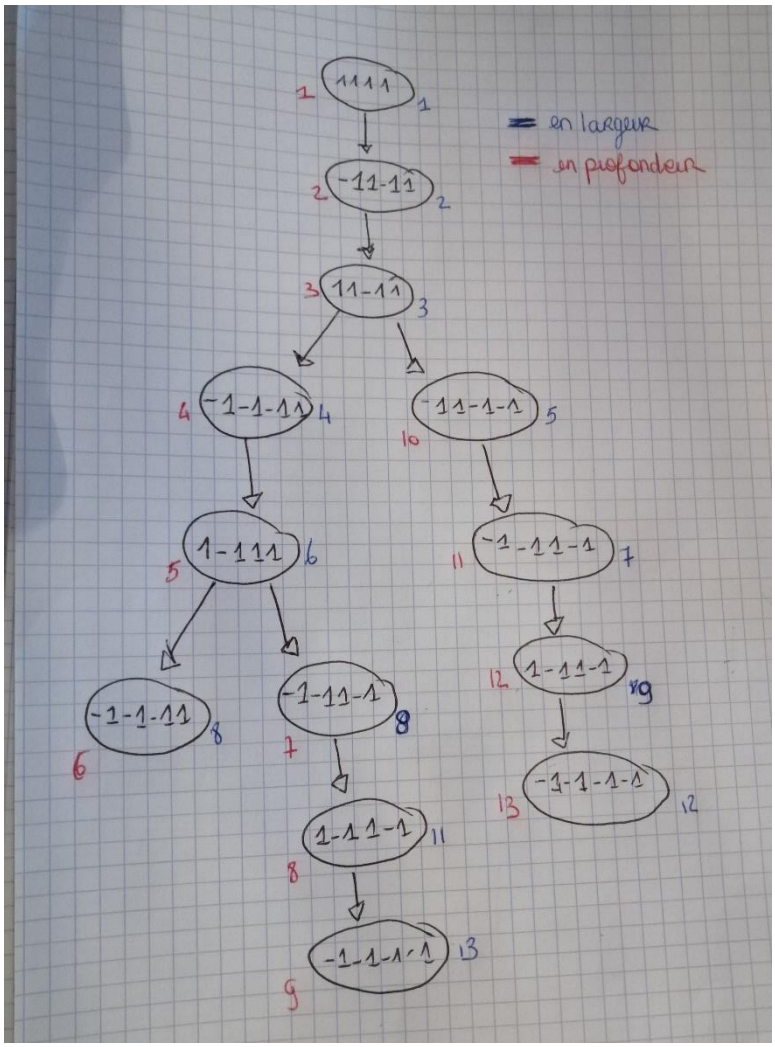
```
CG-USER(67): (1 1 1 1)
(-1 1 -1 1)
(1 1 -1 1)
(-1 1 -1 -1)
```

```
(1 1 1 -1)
(-1 -1 1 -1)
(1 -1 1 -1)
(-1 -1 -1 -1)


(1 1 1 1)
(-1 1 -1 1)
(1 1 -1 1)
(-1 -1 -1 1)
(1 -1 1 1)
(-1 -1 1 -1)
(1 -1 1 -1)
(-1 -1 -1 -1)
```

On remarque que la fonction nous renvoie plusieurs solutions possibles. Ces solutions sont correctes, elles correspondent aux solutions trouvées plu haut. Cet affichage reprend un peu plus le concept d'arborescence (mais peut être complètement omis).

Voici l'arbre de recherche correspondant :



7. Ecrire et tester la fonction Lisp `rech-larg(etat)` qui explore l'espace d'états en largeur d'abord et affiche toutes les solutions possibles. Dessiner l'arbre de recherche correspondant.

Implémentation de la liste globale *predecesseurs*. La liste nous permet de garder la trace du chemin parcouru en répertoriant l'état courant et ses successeurs.

Par exemple : si A a deux successeurs : E et R. On va ajouter (A E) et (A R) dans la liste *predecesseurs*.

Définition de la fonction rech-larg1 : cette fonction nous permet de construire la liste *predecesseurs* par le biais de la fonction successeur plus haut.

```
(defun rech-larg1 (candidate-list old-states)
  (if (null candidate-list) nil
    (progn
  (if (member '(-1 -1 -1 -1) candidate-list :test #'equal)
     (afficher (reverse (append '((-1 -1 -1 -1)) old-states)))) ;; pour vérifier que le parcours est bon
```

```
  (rech-larg1 (diff (flatten
                (mapcar (lambda (xx)(successeurs xx)) candidate-list))
                old-states)
           (append candidate-list old-states)))
    )
  )
```

Cette fonction est récursive et parcourt l'arborescence en largeur. En effet, notons que l'appel récursif se fait sur chaque successeur de la liste des états candidats. Pour ne pas faire de boucle infinie, il est nécessaire de garder une trace des états visités. C'est pourquoi nous faisons la différence ensembliste entre la liste des successeurs des candidats et la liste des états visités appelés old-states. On ajoute les éléments de la liste candidate-list aux états visités puisqu'à l'issu de cet appel récursif les éléments ont été traités.

Définition de la fonction flatten

utilisée au-dessus. Cette fonction permet de remettre la liste des successeurs à plat. C'est-à-dire qu'on aura une liste de profondeur constante. Sinon on aurait une liste qui augmente en profondeur à chaque itération ce qui ne fonctionnerait pas.

```
(defun flatten (L)
  (if L (append (car L) (flatten(cdr L)))))
```

Définition de la fonction diff :

elle permet de faire la différence ensembliste de deux listes. Cela permet de ne pas avoir de boucle dans notre recherche. En effet, si un état a été visité, on ne peut pas repasser par ce dernier.

```
(defun diff (L M)
  (cond ((null L) nil)
        ((member (car L) M :test 'equal) (diff (cdr L) M))
        (t  (cons (car L) (diff (cdr L) M))) ))
```

Définition de recherche-chemin :

affiche toutes les solutions possibles à partir de la liste des prédécesseurs.

```
(defun recherche-chemin(chemin etat ei)
  (if (not(member etat chemin :test #'equal))
     (progn (push etat chemin)
     (if (equal etat ei)
        (print chemin)
       (progn
        (dolist (element *predecesseur*)
          (if (equal (car element) etat)
             (recherche-chemin chemin (cadr element) ei))
          )
        )
       )
       )
    )
  )
```

On aura par exemple :

```
(setq *predecesseur* '((B A) (C A)(D B) (E B) (E C) (F C)(K E)))
(recherche-chemin nil 'K 'A)
```

Résultat :

```
CG-USER(71):
(A B E K)
(A C E K)
NIL
```

Cette fonction part de l'état final, recherche les prédécesseurs de cet état dans la liste des prédécesseurs. Si l'état ne fait pas parti du chemin, si l'état courant est l'état initial on a terminé, donc on affiche le chemin. Pour chaque élément de la liste prédécesseur on regarde si on a un prédécesseur (on remonte dans l'arborescence). Si oui, l'état courant devient le prédécesseur. On remonte ainsi jusqu'à l'état initial.

Définition de la fonction rech-larg, qui est la fonction qui fait le lien entre les dernières fonctions :

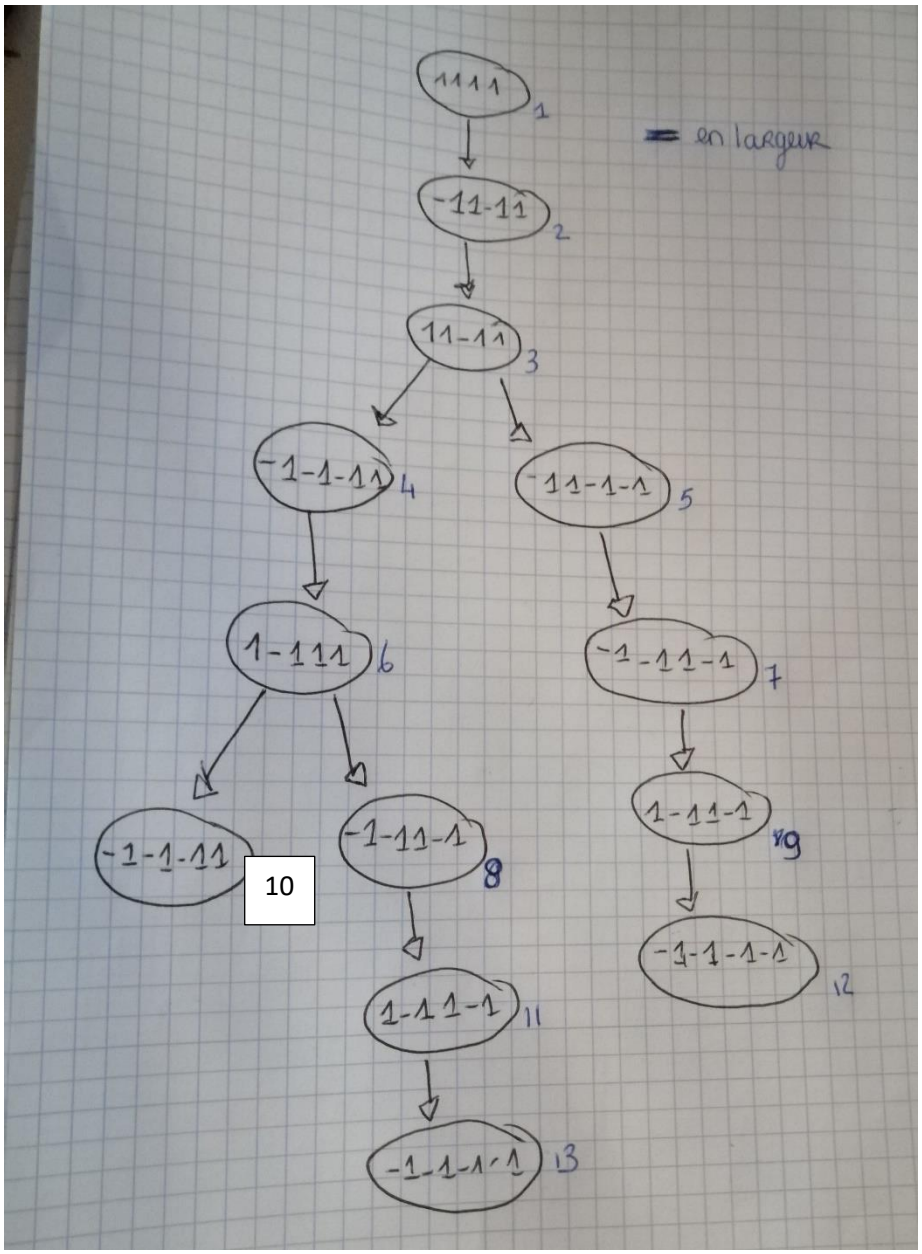
```
(defun rech-larg (etat)
 (setq *predecesseur* nil)
 (rech-larg1 (list etat) nil)
 (recherche-chemin nil '(-1 -1 -1 -1) '(1 1 1 1))
 )
```

On a le résultat suivant :

```
CG-USER(72):
(1 1 1 1)
(-1 1 -1 1)
(1 1 -1 1)
(-1 -1 -1 1)
(-1 1 -1 -1)
(1 -1 1 1)
(1 1 1 -1)
(-1 -1 1 -1)
(-1 -1 1 -1)
(1 -1 1 -1)
(1 -1 1 -1)
(-1 -1 -1 -1)
NIL
((1 1 1 1) (-1 1 -1 1) (1 1 -1 1) (-1 -1 -1 1) (1 -1 1 1)
 (-1 -1 1 -1) (1 -1 1 -1) (-1 -1 -1 -1))
((1 1 1 1) (-1 1 -1 1) (1 1 -1 1) (-1 1 -1 -1) (1 1 1 -1)
 (-1 -1 1 -1) (1 -1 1 -1) (-1 -1 -1 -1))
NIL
```

On peut remarquer que le parcours se fait bien en largeur (j'ai laissé cet affichage pour cela). Ensuite, on a les deux chemins possibles qui sont des chemins valides.

Voici l'arbre de recherche correspondant :



8. Comparer les deux méthodes.

La méthode de recherche en profondeur est beaucoup plus simple à implémenter. En effet, les chemins possibles sont trouvés de manière assez directe, tandis qu'il faut retrouver les prédécesseurs des sommets explorés et appartenant à la solution afin de remonter dans l'arborescence de recherche pour la recherche en largeur. De plus, la recherche en largeur nécessite l'implémentation de nombreuses fonctions supplémentaires ce qui ajoute à la complexité de l'algorithme. Au niveau de la complexité, la complexité de la recherche en largeur est supérieure à celle en profondeur puisqu'on doit construire la liste des prédécesseurs par la recherche en largeur puis reconstruire les chemins possibles grâce à cette liste.

Cela dit, le parcours en largeur a l'avantage de rechercher le plus court chemin. Même si ici, on fait en sorte que toutes les solutions possibles soient trouvées.