PERINET Anaëlle

Rapport Projet IA

Le but final du projet est de réaliser en CLIPS un labyrinthe dans lequel se déplace un aventurier de manière autonome. Lors de son parcours il peut rencontrer des monstres, des potions, des trésors, des portes, des épreuves et bien d'autres encore.

Dans un premier temps nous créons une base de faits qui sera la base du labyrinthe avec le prédicat *Laby(n_x,n_y)* qui indique qu'il existe un chemin entre n_x et n_y. Nous avons donc le choix de mettre un chemin à sens unique ou à double sens.

Dans un second temps on va initialiser l'aventurier avec son emplacement (nœud), sa force et son agilité comprises entre 1 et 10.

Dans un troisième temps on choisit de manière aléatoire grâce à la fonction « set-strategy random » la direction suivante où l'on souhaite aller c'est-à-dire au nœud suivant. La fonction « set-strategy » permet de sélectionner les faits de la base d'une manière bien précise, ici on utilise le mode aléatoire.

```
5   (deffacts initialisation_projet_IA
6       ;_____________________(laby nx ny)_____
7
8       (laby 1 2)
9
10      (laby 2 7)
11      (laby 2 3)
12
13      (laby 3 2)
14      (laby 3 4)
15      (laby 3 5)
16      (laby 3 9)
17
18      (laby 4 5)
19      (laby 4 3)
20      (laby 4 7)
21      (laby 4 8)
```

set-strategy depth

set-strategy random

```
86  (defrule choix_direction
87      (declare (salience 5))
88      (aventurier ?n ? ?)
89      (laby ?n ?ny)
90      (not (noeud_suiv))
91      (not(fin))
92  =>
93      (assert (noeud_suiv ?ny))
94  )
```

On peut donc voir qu'en mode normal « depth » lors de l'exécution du programme, le premier fait sera sélectionné tandis qu'en mode aléatoire « random », par exemple ici au nœud 4 (n_x), il va choisir de manière aléatoire entre les nœuds (n_y) 3, 5, 7 et 8.

De plus, on implémente une mémoire à l'aventurier pour lui éviter de revenir sur ses pas, sur le nœud qu'il vient de parcourir. Pour cela on ajoute un nouveau fait « mémoire » qui retient le nœud précédent lorsque l'aventurier passe au nœud suivant. Si l'algorithme sélectionne un « nœud mémoire » alors l'algorithme se relance et choisit un autre nœud.

Ensuite, on implémente dans la base des faits de nouveaux faits : les monstres et les trésors. Ils sont caractérisés, d'une part pour les monstres par leur nom, leur emplacement dans le labyrinthe et leur force d'attaque et d'autre part pour les trésors par leur emplacement dans le labyrinthe et leur butin. Tout au long du parcours nous allons vérifier pour chaque nœud s'il y a un monstre ou un trésor en

comparant le nœud (voir image ci-dessous) sur lequel se trouve l'aventurier et ceux sur lesquels se trouvent les monstres et les trésors.

```
155  (defrule tresor_trouve
156      (declare (salience 10))
157      (aventurier ?n ? ?)
158      ?t <- (tresor ?n ?m)
159      ?s <- (score ?pts)
160      (not(fin))
161  =>
162      (bind ?newscore (+ ?pts ?m))
163      (retract ?t ?s)
164      (printout t "*TRESOR* +" ?m " pieces")
165      (assert (score ?newscore))
166  )
```

```
176  (defrule monstre_inf
177      (declare (salience 12))
178      (aventurier ?n ?f ?)
179      ?m <- (monstre ?nom ?n ?fm)
180      (test(<= ?fm ?f))
181  =>
182      (retract ?m)
183      (printout t "*MONSTRE* |" ?nom "||Fo
184  )
```

En outre, pour les monstres, on compare la force de l'aventurier à celle des monstres pour savoir si l'aventurier est assez fort pour combattre les monstres ou non. Si l'aventurier est trop faible cela lui fait quitter le programme et son score tombe à 0. Pour les trésors, à chaque fois que l'aventurier arrive sur un nœud avec un trésor, le montant de ce trésor est additionné au score. Si le joueur récupère tous les trésors disponibles dans le labyrinthe, il atteint un score maximal alors il termine le jeu et le programme s'arrête.

Pour enrichir le jeu, on a implémenté des épreuves qui se basent sur le même fonctionnement que les monstres mais cette fois-ci avec de l'agilité. Pour cela, nous avons rajouter un nouveau fait « épreuve » dans la base des faits qui est caractérisé par le nœud où se trouve cette épreuve et le nombre de points d'agilité nécessaire pour réussir l'épreuve. L'aventurier possède donc des points d'agilité, si sur le nœud où il se situe il y a une épreuve, on compare les points d'agilité de l'aventurier et le nombre de points d'agilité nécessaire pour passer l'épreuve. Si l'aventurier passe l'épreuve alors il continue le jeu sinon le programme s'arrête et le score s'affiche.

Par conséquent, nous avons ajouté dans la base de fait un nouveau fait « potion » qui permet à l'aventurier de gagner des points d'agilité. Ce fait se caractérise par le nœud où se trouve la potion et les points d'agilités contenus dans la potion. Si l'aventurier se trouve sur un nœud où il y a une potion on additionne les points d'agilités.

Nous avons également implémenté un nouveau fait « porte » qui se caractérise par le nœud où se trouve la porte et le nom de la clé qu'il faut pour déverrouiller la porte. Pour cela nous avons implémenté un nouveau fait « clé » qui se caractérise par le nœud où se trouve la clé et le nom de la clé. Si le joueur se trouve sur un nœud où il y a une clé alors il la récupère et on supprime la clé concernée de la base des faits. S'il est sur un nœud où il y a une porte verrouillée c'est-à-dire qu'il n'a pas encore récupéré la clé correspondante alors l'aventurier fait marche arrière, sinon la porte se déverrouille et l'aventurier continue d'avancer dans le labyrinthe. Pour savoir si le joueur possède la clé pour déverrouiller la porte ou non il suffit de vérifier si la clé est dans la base des faits.

On a utilisé la fonction « random (1 N) » lors de l'initialisation de la base des faits pour l'emplacement monstres et leur force. On a choisi de considérer directement des valeurs entières pour les noms de lieu/noeud et non lettre + numéro car pour le "random" il faut utiliser la fonction "str-cat" pour concaténer les deux (lettre + chiffre). Cependant cela va être sous format d'un string donc cela complique les choses pour les conditions.

Exemple d'exécution du programme :

```
|N2|
|N3| *POTION*, tu recuperes 4 points d'agilite. Tu as 7 d'agilite maintenant.
|N5| *MONSTRE* |Bellick||Force : 5| Il est plus faible que toi, tu gagnes !
|N4| *TRESOR* +300 pieces
|N8| *EPREUVE D'AGILITE* de niveau; 5 tu es assez agile (7), tu reussi ! *CLE*, tu possedes maintenant la cle_1
|N7|
|N2|
|N3|
|N9| *PORTE DEVERROUILLE*, cle_1 possede !

|N6| *MONSTRE* |Patrick||Force : 1| Il est plus faible que toi, tu gagnes !*TRESOR* +500 pieces

|SCORE : 800|

CLIPS> (reset)
CLIPS> (run)

|N2|
|N3| *POTION*, tu recuperes 4 points d'agilite. Tu as 7 d'agilite maintenant.
|N9| *PORTE VERROUILLE*, cle_1 non possede ...
|N3|
|N2|
|N7|
|N8| *EPREUVE D'AGILITE* de niveau; 5 tu es assez agile (7), tu reussi ! *CLE*, tu possedes maintenant la cle_1
|N4| *MONSTRE* |Patrick||Force : 9| Il est plus fort que toi, tu as perdu !

|SCORE : 0|
```

Durant l'exécution du programme nous avons rencontré quelques conflits dans l'agenda et donc pour y remédier nous avons utilisé la fonction (declare (salience N)) qui permet d'exécuter selon un ordre les règles. Par exemple, la règle « Quitter » est prioritaire puisque c'est une règle qui doit être vérifiée à chaque fois. Au début, nous avons eu des difficultés à utiliser cette fonction, par exemple des fois le programme ne nous disait par lorsqu'il y avait un monstre car des règles moins importantes qui permettait de continuer à avancer étaient vérifiée avant celle des monstres.

Nous avons également eu des difficultés à mettre la mémoire en application, car nous n'arrivions pas à comparer les réponses, pour résoudre ce problème on a dû rajouter un fait « noeud_suiv » qui contient la valeur du nœud afin de le comparer avec le nœud mémoire.

Afin de répondre au mieux à ce projet, nous avons eu besoin de ressources internet, nous nous sommes aidé des sites : stackoverflow.com et csie.ntu.edu.tw/~sylee/courses/clips/bpg/top.html