Université de Sherbrooke

IFT 703 : Informatique cognitive

Rapport Projet - partie 2

Automne 2024

Enseignant :

Mohamed Mehdi Najjar

Réalisé par :

Achraf Haijoubi	Matricule : 24 014 567
Fatima-ezzahra Bour	Matricule : 24 171 487
Ayman Zouhair	Matricule : 24 170 433
Zineb Sebti	Matricule : 24 171 469

Table des figures

Table des matières

1 Introduction générale

Dans ce projet, nous abordons la problématique complexe de l'organisation des bagages dans un coffre d'automobile, un problème qui implique des contraintes variées liées à la taille, la forme, le poids et la stabilité des bagages. Le défi principal réside dans l'optimisation de l'utilisation de l'espace disponible tout en respectant l'intégrité physique des objets transportés. En effet, un mauvais placement des bagages peut entraîner des risques de dommages, de déséquilibre et d'incapacité à fermer le coffre correctement.

Le modèle ACT-R[1] , que nous utilisons dans ce projet, cherche à résoudre cette problématique en plaçant un nombre aléatoire de bagages dans le coffre d'une voiture, tout en prenant en compte plusieurs critères : les dimensions du coffre, le poids et la forme des valises, ainsi que leur type. L'objectif est de parvenir à une organisation stable des bagages, garantissant à la fois l'optimisation de l'espace et la sécurité des objets transportés. Cependant, plusieurs défis doivent être surmontés pour rendre ce modèle adaptable à une gamme de scénarios réalistes, où le nombre de bagages peut varier et où des priorités différentes (poids, taille, catégorie des valises) doivent être prises en compte.

La problématique principale de ce projet réside donc dans la gestion de ces multiples contraintes, avec la nécessité d'adapter un modèle intelligent capable de s'ajuster à des conditions de placement variées. Dans les sections suivantes, nous détaillerons les approches mises en œuvre pour répondre à cette problématique, y compris la manière dont nous avons ajusté et amélioré le modèle pour garantir une organisation efficace et stable des bagages dans le coffre, tout en optimisant l'espace disponible.

2 Représentation des connaissances

Dans ce chapitre, nous allons présenter les connaissances de notre modèle, notamment, les chunks modifiés, ainsi que les slots et procédures ajoutés. Toutefois, dans le cadre de notre travail, nous n'avons pas crée de nouveaux buts ou sous-buts, mais avons plutôt utilisé ceux déjà proposés .

2.1 Description des chunks et slots

```
(chunk-type arrange-state c1 c2 c3 c4 c5 c6 p1 p2 p3 p4 p5 p6 n1 n2 n3 n4
   n5 n6 encoding-status (memory-retention "f") sum1 sum2 result state)
```

- Les slot $\{c_1 \dots c_6\}$ correspondent aux catégories des six valises, et peuvent avoir pour valeur 1,2 ou 3.
- Les slot $\{p_1 \dots p_6\}$ correspondent aux poids des six valises, et peuvent avoir pour valeur 1,2,3,4 ou 5.
- Les slot $\{n_1 \dots n_6\}$ représentent les niveaux où les six valises sont placées. ils peuvent prendre les valeurs 1,2 ou 0 si une valise n'est pas placée.
- Le slot *encoding-status* permet de suivre l'état du processus d'encodage.
- Le slot *memory-retention*, par défaut initialisé à "f" (false) indique si une organisation précédente est organisée.
- Les slots *sum1* et *sum2* correspondent respectivement aux sommes des dimensions des valises placées dans le niveau 1 et 2.
- Le slot *result* est utilisé pour stocker le résultat de l'arrangement ("win" ou "lose").
- Le slot *state* représente l'étape courante du processus.

Listing 2.1 – Création des valises

```
(chunk-type learned-info  c1 c2 c3 c4 c5 c6  p1 p2 p3 p4 p5 p6  n1 n2 n3 n4
    n5 n6)
```

- Comme indiqué précédemment, les slots $\{c_1 \dots c_6\}$, $\{p_1 \dots p_6\}$ et $\{n_1 \dots n_6\}$ représentent respectivement les catégories, les poids et les niveaux des six valises, avec les mêmes valeurs et significations décrites plus haut.

2.2 Description des procédures ajoutées et/ou modifiées

2.2.1 procédure espace-initial

```
(p espace-initial
   =goal>
      state org-stage-1

      c1 =cat1
      c2 =cat2
      c3 =cat3
      c4 =cat4
      c5 =cat5
      c6 =cat6

   ==>
   !bind! =sum (+ (espace-occupe =cat1) (espace-occupe =cat2) (
       espace-occupe =cat3))
   =goal>
      state org-stage-2
      sum1 =sum
      n1 1
      n2 1
      n3 1
)
```

Cette procédure permet de :

— Initialiser par défaut les trois premiers bagages au niveau 1.

— Calculer l'espace occupé par le niveau 1 en additionnant les espaces des trois premiers bagages.

2.2.2 procédure cas-depassement-niveau-1

```
(p cas-depassement-niveau-1
   =goal>
      state org-stage-2

      sum1 =sum
   >  sum1 36

      c1 =cat1
      c2 =cat2
      c3 =cat3
      c4 =cat4
      c5 =cat5
      c6 =cat6
   ==>
   !bind! =ssum (- =sum (espace-occupe =cat3))
   !bind! =ssum2 (espace-occupe =cat3)
```

```
   =goal>
      state org-stage-3
      n3 2
      sum1 =ssum
      sum2 =ssum2
)
```

Cette procédure permet de gérer un cas de dépassement de niveau en réorganisant l'espace occupé par la troisième valise.
Sachant que l'espace maximal pouvant être occupé par un niveau est de 36 (6×6), on vérifie si l'espace calculé dépasse cette limite. Si c'est le cas, la troisième valise est placée au deuxième niveau, et les sommes sont mises à jour en conséquence (On soustrait l'espace occupé par cat3 de sum1 et on l'ajoute à sum2).

2.2.3 procédure cas-non-depassement-niveau-1

```
    (p cas-non-depassement-niveau-1
   =goal>
      state org-stage-2

      sum1 =sum
   <= sum1 36

      c1 =cat1
      c2 =cat2
      c3 =cat3
      c4 =cat4
      c5 =cat5
      c6 =cat6
   ==>
   =goal>
      state org-stage-3
      sum1 =sum
      sum2 0
      n3 1
)
```

Cette procédure gère le cas où le niveau 1 n'est pas dépassé.
Elle intervient lorsque l'espace total calculé pour le niveau 1 (sum1) est inférieur ou égal à 36, respectant ainsi la limite maximale. Dans ce cas, l'état du système passe à l'étape suivante (org-stage-3), les sommes sont mises à jour (sum1 conserve sa valeur, sum2) est initialisée à 0), et la troisième valise reste au premier niveau (n3 = 1).

2.2.4 procédure cas-depassement-niveau-2

```
(p cas-depassement-niveau-2
   =goal>
      state org-stage-4

      sum2 =sum2
```

```
   >  sum2 36

      c1 =cat1
      c2 =cat2
      c3 =cat3
      c4 =cat4
      c5 =cat5
      c6 =cat6
   ==>
   !bind! =ssum2 (- =sum2 (espace-occupe =cat5))
   =goal>
      state encoding-phase
      encoding-status comparing_weight
      sum2 =ssum2
      n5 0
      n6 0
)
```

Cette procédure traite le cas où l'espace total calculé pour le niveau 2 dépasse la limite maximale autorisée (>36). Lorsque cette condition est détectée, l'espace occupé par la cinquième valise (cat5) est soustrait de la somme totale du niveau 2, et la somme ajustée (ssum2) est calculée. Ensuite, l'état du système passe à la phase d'encodage (encoding-phase), avec le statut d'encodage défini sur comparing_weight. Dans cette situation, ni la cinquième valise ($n5 = 0$) ni la sixième valise ($n6 = 0$) ne sont considérées comme placées.

2.2.5 procédure cas-non-depassement-niveau-2

```
(p cas-non-depassement-niveau-2
   =goal>
      state org-stage-4

      sum2 =sum2
   <= sum2 36

      c1 =cat1
      c2 =cat2
      c3 =cat3
      c4 =cat4
      c5 =cat5
      c6 =cat6
   ==>
   =goal>
      state encoding-phase
      encoding-status comparing_weight
      sum2 =sum2
      n5 2
      n6 0
)
```

Cette procédure gère le cas où le niveau 2 respecte la limite maximale d'espace occupé. Elle s'exécute lorsque l'espace total calculé pour le niveau 2 (sum2) est inférieur ou égal à 36. Dans ce cas, l'état du système passe à la phase d'encodage (encoding-phase), et le

statut d'encodage est défini sur comparing_weight. Les sommes restent inchangées (sum2 conserve sa valeur), la cinquième valise est placée au deuxième niveau ($n5 = 2$), et la sixième valise est marquée comme non placée ($n6 = 0$).

2.3 Description des fonctions ajoutées et/ou modifiées

2.3.1 fonction espace-occupe

```
   (defun espace-occupe (cat)
   (setf espace "error-calcul-espace")
   (setf catt cat)
   (when (equal cat "none") (setf catt 0))
   (case catt
      (1 (setf espace 9))
      (2 (setf espace 12))
      (3 (setf espace 18))
      (0 (setf espace 0))
   )
   espace
)
```

Cette fonction détermine l'espace occupé par chaque bagage en fonction de ses dimensions, ce qui correspond à sa catégorie.

2.3.2 fonction create-valises

```
(defun create-valises()
  ;; Génère un nombre aléatoire de valises entre 3 et 6
  (let* ((nb-valises (+ 3 (act-r-random 4))) ; Nombre aléatoire entre 3 et
      6
         (valise-list (loop repeat nb-valises collect (make-instance '
            valise)))) ; Crée les valises
    ;; Configure chaque valise
    (loop for valise in valise-list
       do (progn
            (setf (slot-value valise 'poids) (1+ (act-r-random 5))) ; poids
                aléatoire
            (setf (slot-value valise 'categorie) (1+ (act-r-random 3))) ;
               catégorie aléatoire
            (setf (slot-value valise 'couche) 1) ; par défaut, couche 1
            ;; Dimensions selon la catégorie
            (case (slot-value valise 'categorie)
              (1 (progn (setf (slot-value valise 'x) 3) (setf (slot-value
                 valise 'y) 3)))
              (2 (progn (setf (slot-value valise 'x) 6) (setf (slot-value
                 valise 'y) 2)))
              (3 (progn (setf (slot-value valise 'x) 6) (setf (slot-value
                 valise 'y) 3))))))
    ;; Trie les valises par poids décroissant puis par catégorie croissante
```

```
    (sort-by-poids-first valise-liste )))
```

Cette fonction a été modifiée pour générer un nombre aléatoire de valises compris entre 3 et 6. Les valises sont directement stockées dans une liste temporaire valise-list, créée dynamiquement par loop. On a ajouter une étape supplémentair qui trie les valises après leur création avec la fonction sort-by-poids-first , qu'on verra dans 2.3.4 .

2.3.3 fonction compare-valises

```
(defun compare-valises (x y)
;; Compare deux valises en fonction de leur poids et, en cas d'égalité, par
    catégorie.
  (cond
    ((/= (valise-poids x) (valise-poids y))
     (> (valise-poids x) (valise-poids y)))
    (t
     (< (valise-categorie x) (valise-categorie y)))))
```

Cette fonction compare deux valises x et y en fonction des critères suivants :

- **Critère principal :** le poids, classé par ordre décroissant.
- **Critère secondaire (en cas d'égalité sur le poids) :** La catégorie, classé par ordre croissant.

2.3.4 fonction sort-by-poids-first

```
(defun sort-by-poids-first (vlist)
  ;; Trie les valises par poids décroissant, puis par catégorie croissante
     en cas d'égalité.
  (stable-sort vlist #'compare-valises))
```

Cette fonction trie une liste de valises (vlist) en appliquant les critères définis dans la fonction compare-valises.

2.3.5 Méthode proposée pour la gestion de la hauteur variable des bagages

Les connaissances de notre modèle, notamment les chunks modifiés, ainsi que les slots et procédures ajoutés, ont été essentielles pour résoudre les trois premières limites liées au placement illogique des bagages, à la gestion du nombre variable de valises et à la prise en compte du poids. Ces ajustements ont permis d'améliorer la stabilité et l'efficacité du modèle. En ce qui concerne la quatrième limite, qui porte sur la gestion de la hauteur variable des bagages, nous ne l'avons pas réalisée concrètement. Cependant, nous présenterons dans cette soussection la méthode proposée et utilisée pour résoudre cette problématique.

La démarche pour résoudre la quatrième limite, liée à la hauteur variable des bagages, commence par un tri des bagages selon plusieurs critères hiérarchisés. Le premier critère de tri est la hauteur des bagages, qui peut prendre deux valeurs possibles : 1, représentant la hauteur du niveau 1 du coffre, et 2, représentant la hauteur du coffre, niveau 1 + 2. Ensuite, en cas d'égalité sur la hauteur, le tri se fait selon le poids des bagages, dans l'ordre décroissant. Enfin, si la hauteur et le poids sont identiques, la catégorie des bagages est prise en compte, triée dans l'ordre croissant. Une fois les bagages triés, l'algorithme vérifie si le premier niveau du coffre est libre pour le placement du bagage. Si la hauteur du bagage est égale à 2 et que le niveau 1 est libre, le bagage est placé dans ce niveau. Cependant, si le premier niveau est occupé, le bagage ne pourra être placé et sera marqué comme "non placé". Si la hauteur du bagage est égale à 1, l'algorithme traite le cas de manière classique, en vérifiant la disponibilité du niveau 1, comme prévu dans notre modèle. Enfin, si les deux niveaux sont occupés, le bagage est également marqué comme "non placé". Cette approche garantit un placement optimal tout en respectant les contraintes de hauteur, de poids et de catégorie.

3 Brève description d'un scénario de la réalisation de la tâche

Après génération d'une liste de 4 bagages (avec des catégories c1, c2, c3, c4 et des poids p1, p2, p3, p4) — la génération pouvant se faire de 3 à 6 valises, mais dans ce scénario, nous prenons le cas de 4 — le modèle effectue un tri des bagages en fonction de leur poids, du plus lourd au plus léger, et de leur catégorie, en commençant par les valises les plus grandes. Le modèle vérifie ensuite dans sa mémoire s'il a déjà rencontré cet arrangement. Si oui, il place les bagages directement dans leur position correcte. Si non, il commence par essayer de placer les 3 premiers bagages dans le niveau 1 du coffre, en vérifiant si l'espace occupé respecte la capacité du coffre. Si cela rentre, il place le 4e bagage dans le niveau 2. En cas de dépassement, il place 2 bagages dans le niveau 1 et les 2 autres dans le niveau 2. Si un bagage ne peut pas être placé dans l'un des deux niveaux, il est marqué comme non placé. Une fois l'organisation effectuée, le modèle mémorise l'agencement pour une utilisation future.

4 Structure modifiée : modèle d'organisation des bagages

4.1 Processus Général

Le modèle d'organisation des bagages suit les étapes suivantes :

- **Génération des Bagages**
  - Nombre de bagages : Aléatoire entre 3 et 6
  - Attribution aléatoire des caractéristiques :
    - Catégories (1, 2, 3)
    - Poids (1 à 5)
- **Préparation Initiale**
  - Tri des bagages :
    - Critère principal : Poids (ordre décroissant)
    - Critère secondaire : Catégorie (ordre croissant)

4.2 Productions du modèle

4.2.1 Production Start

- **Objectif :** Rappel du but présent
- **Mécanisme :** Identifier les niveaux actuels des bagages

4.2.2 Production remember-organisation

- **Condition :** Succès du rappel des niveaux des valises
- **Actions :**
  * Passer à la procédure *encode-cat1*
  * Retourner le niveau de chaque valise
  * Définir le slot "result" à win
  * Terminer l'exécution

4.2.3 Production doesn't-remember-organisation

— **Condition :** Échec du rappel des niveaux des valises
— **Action :** Diriger vers la procédure *espace-initial*

4.2.4 Organisation des Espaces

— **Espace initial** : Calcul de sum1 = cat1 + cat2 + cat3 ; transition vers `org-stage-2`.
— **Dépassement niveau 1** : Si sum1 > 36, retirer cat3, ajuster n3 = 2, et passer à `org-stage-3`.
— **Non-dépassement niveau 1** : Si sum1 $\leq$ 36, maintenir n3 = 1, et passer à `org-stage-3`.
— **Espace libre** : Ajouter cat4 + cat5 à sum2, ajuster n4, n5 = 2, et passer à `org-stage-4`.
— **Dépassement niveau 2** : Si sum2 > 36, retirer cat5, ajuster n5, n6 = 0, et passer à `encoding-phase`.
— **Non-dépassement niveau 2** : Si sum2 $\leq$ 36, maintenir n5, n6 = 2, 0, et passer à `encoding-phase`.

4.2.5 Encodage des Niveaux

— **Objectif** : Encoder chaque catégorie (`cat1` à `cat6`) et son niveau associé (`n1` à `n6`).
— **Mécanisme** : Transférer les valeurs dans la mémoire imaginale et avancer dans le processus.

4.2.6 Finalisation

— **Objectif** : Mémoriser les caractéristiques finales des bagages.
— **Action** : Stocker catX, poidsX, nivX pour $X = 1..6$ et passer à la phase de finalisation.

5 Scénario d'exécution

Dans ce chapitre, nous détaillons les différentes étapes du scénario d'exécution du modèle ACT-R, illustrées par des captures d'écran de la fenêtre du Stepper. Le Stepper est un outil qui permet de visualiser le processus de décision et d'exécution du modèle, étape par étape. À travers ces captures, nous analyserons le fonctionnement interne du modèle et les choix effectués par le système.

Dans ce scénario, le modèle commence par générer 4 valises avec des catégories respectives de 3, 3, 2 et 2, et des poids correspondants de 5, 3, 1 et 1. Les valeurs de `CAT5`, `CAT6`, `POIDS5` et `POIDS6` sont égales à `none`.

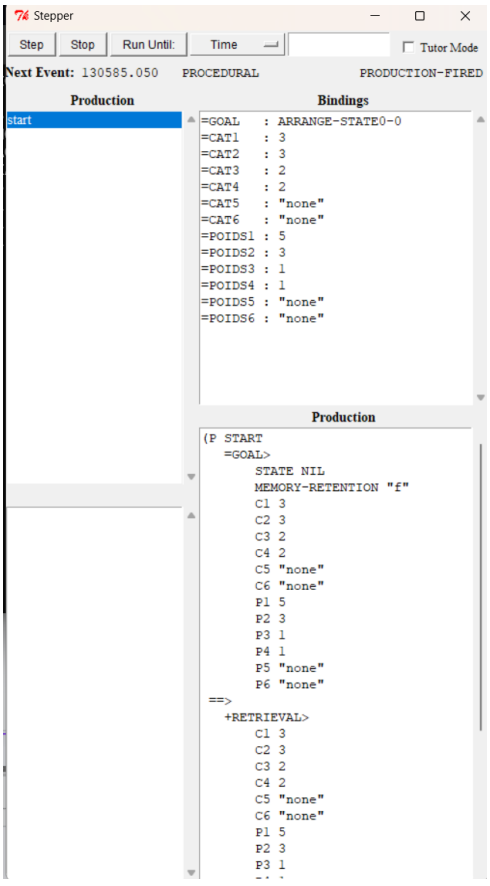

FIG. 5.1 – Stepper (start)

Le modèle ne se rappel pas de l'organisation et l'état du goal passe à **ORG STAGE-1.**

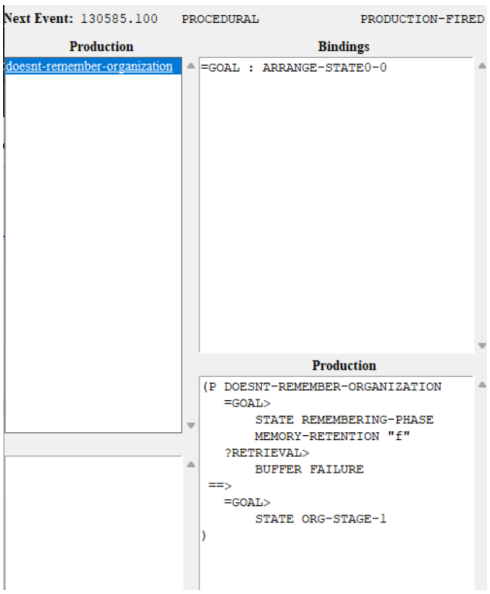

FIG. 5.2 – Stepper (doesnt-remember-organization)

La procédure *espace-initial* calcule, dans le LHS, l'espace occupé par les 3 premières valises (dans ce cas sum = 48) et places les 3 valises dans le niveau 1 et puis donne au state du goal **ORG-STAGE-2.**

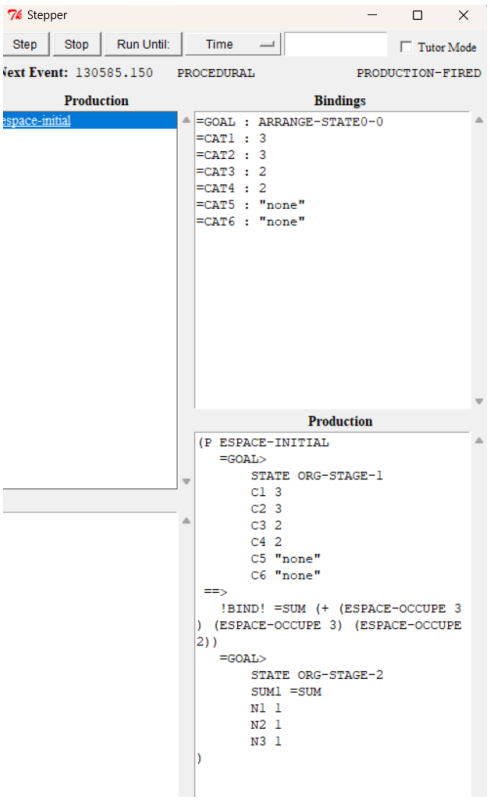

FIG. 5.3 – Stepper (espace-initial)

La procédure *cas dèpassement-niveau 1* se déclenche car l'espace occupé par les 3 valises est supérieur à 36 alors le modèle place la valise au niveau 2 et l'ètat du goal passe à **ORG-STAGE-3**.

NB : Si sum avait été infèrieur à 36, le modèle ACT-R serait passé à la procédure *cas-non-depassement-niveau-1*.

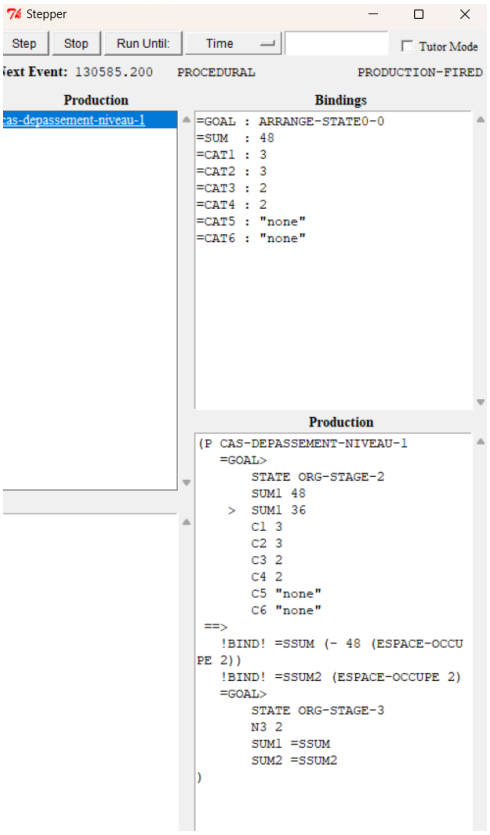

FIG. 5.4 – Stepper (cas dèpassement-niveau 1)

On calcule l'espace au niveau 2 (ici c'est 24) et on place la 4ème valise au niveau 2 et l'ètat du goal est changé en **ORG-STAGE-4**

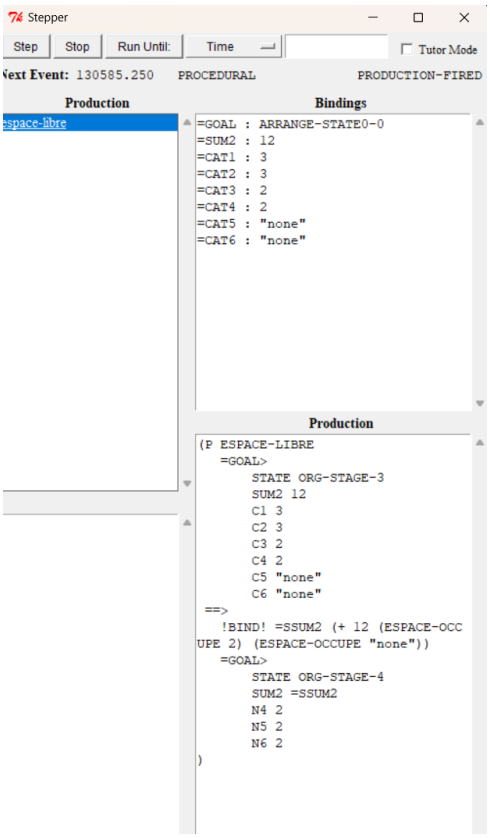

FIG. 5.5 – Stepper (espace-libre)

Le modèle passe à la procédure *cas-non-dépassement-niveau-2* car l'espace occupé au niveau 2 est sum2 = 24 < 36. **NB** : Si sum2 avait été supérieur à 36, le modèle ACT-R serait passé à la procédure *cas-dépassement-niveau-2*.

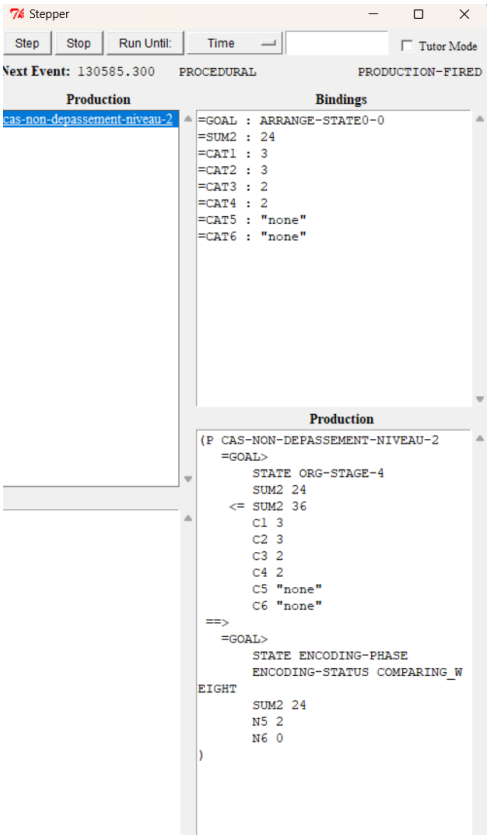

FIG. 5.6 – Stepper (cas-non-dépassement-niveau-2)

La procédure *memorize* permet d'enregistrer la configuration des bagages en utilisant le module *IMAGINAL* d'ACT-R.

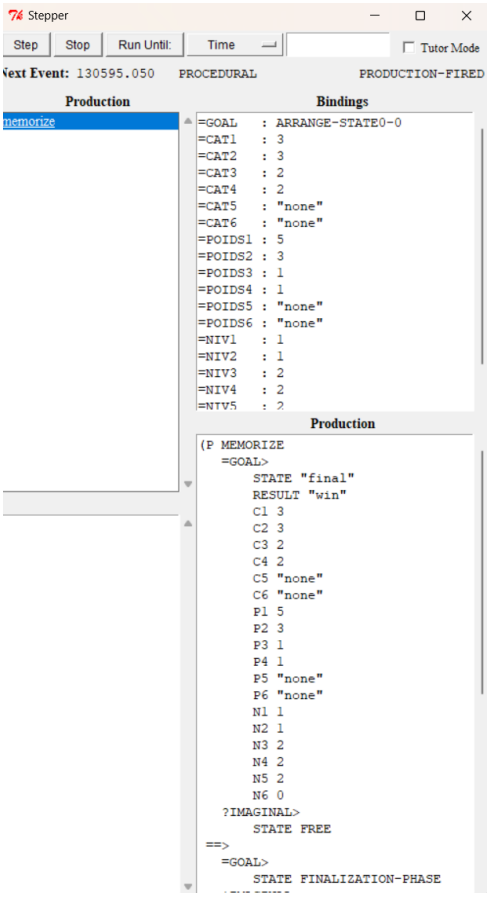

FIG. 5.7 – stepper (cas dèpassement-niveau 1)

Enfin, la procédure **clear-new-imaginal-chunk** vide le contenu du buffer imaginal pour libérer de l'espace pour un nouveau chunk.

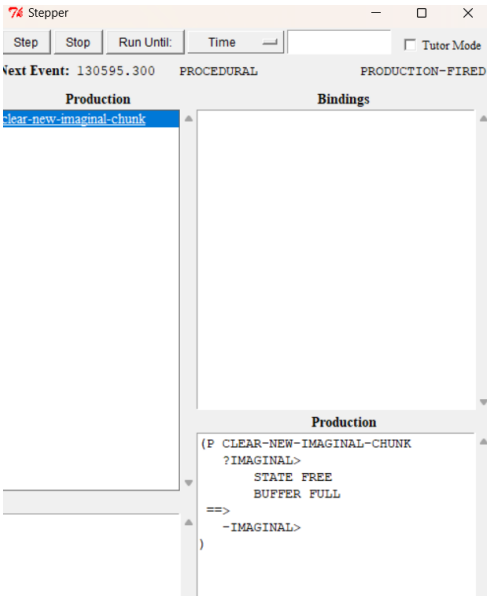

FIG. 5.8 – stepper (clear-new-imaginal-chunk)

Le résultat de notre modèle pour ce scénario montre l'organisation des valises en fonction de leurs caractéristiques. Chaque valise est décrite par son poids, sa catégorie, ses dimensions et son placement sur une couche donnée. Les niveaux sont affichés pour illustrer la disposition spatiale des valises, et il est important de noter que, dans ce cas, toutes les valises ont été placées aux niveaux 1 et 2, sans qu'aucune ne reste non placée.

```
La valise pese: 5, est de categorie 3, mesure 6x3 et est positionnee a la couche 1
La valise pese: 3, est de categorie 3, mesure 6x3 et est positionnee a la couche 1
La valise pese: 1, est de categorie 2, mesure 6x2 et est positionnee a la couche 2
La valise pese: 1, est de categorie 2, mesure 6x2 et est positionnee a la couche 2
Niveau 1:
 ____________
|____________|
|____________|
|____________|

 ____________
|____________|
|____________|
|____________|


Niveau 2:
 ____________
|____________|
|____________|

 ____________
|____________|
|____________|

Valises non placees :
0.6666667#<SB-IMPL::STRING-OUTPUT-STREAM {2B940F41}> is closed
```

FIG. 5.9 – Résultat graphique du scénario

6 Aprentissage du modèle

Dans ce chapitre, nous générons plusieurs graphes d'apprentissage pour évaluer la capacité de notre modèle ACT-R à apprendre. À chaque expérience, nous modifions le nombre de blocs ou le nombre d'expériences par bloc, afin d'observer l'impact de ces variations sur la performance du modèle. Ces ajustements permettent d'analyser comment le modèle s'adapte et apprend au fil du temps, en observant l'évolution des performances en fonction de ces paramètres.

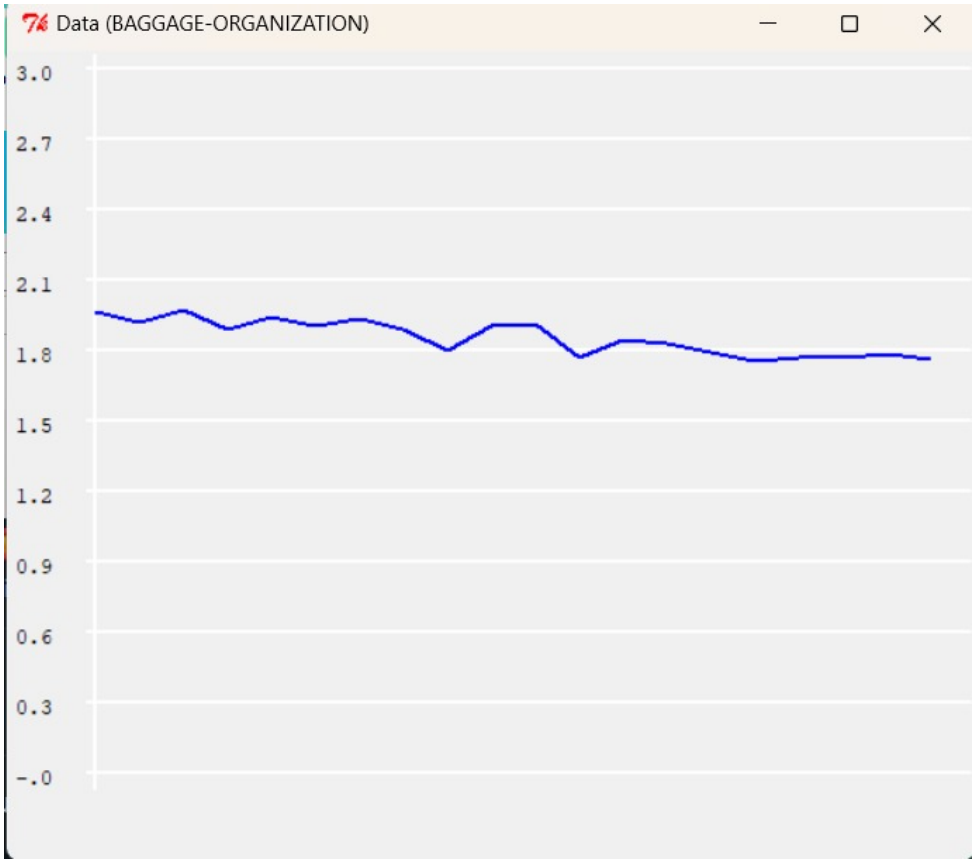

Fig. 6.1 – Graphe de 20 blocs de 100 expériences

Le graphe ci-dessus est généré avec la commande **(show-learning 20)** et montre une diminution du nombre moyen passant d'approximativement de 2 à environ 1,8 sur 20 blocs d'apprentissage de 100 expériences chacun. Cela reflète une amélioration progressive des performances, indiquant que notre modèle ACT-R arrive à apprendre.

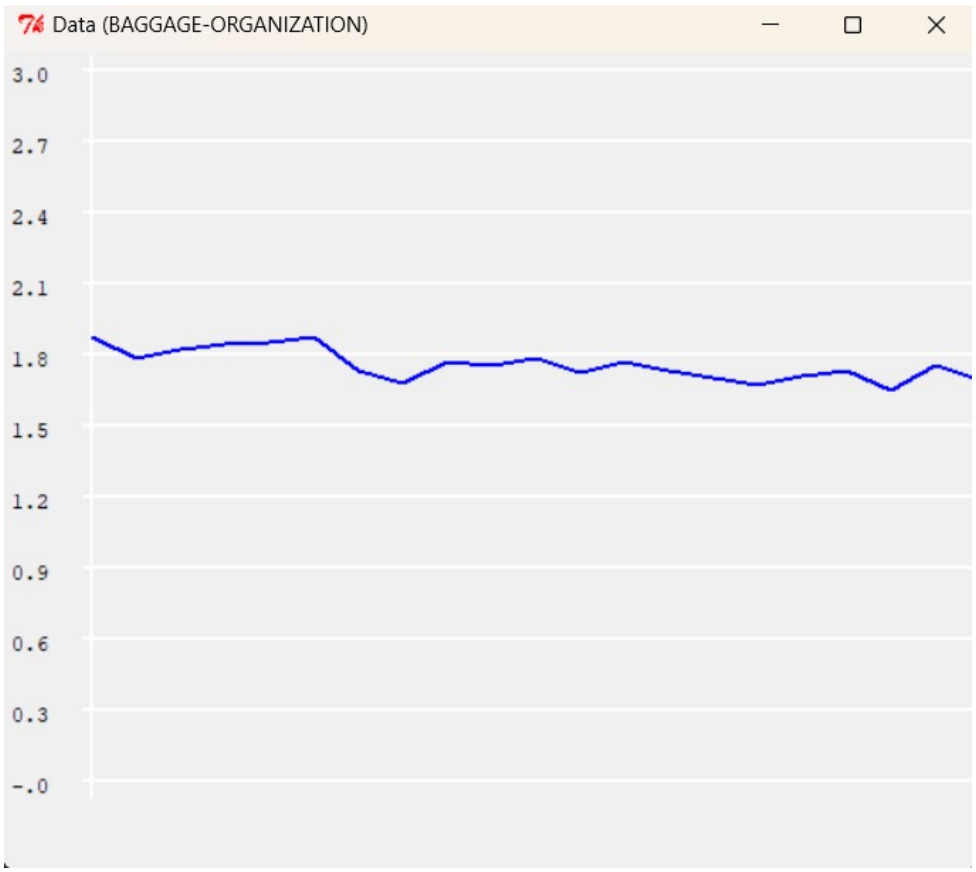

FIG. 6.2 – Graphe de 100 blocs de 100 expériences

Le graphe ci-dessus est généré avec la commande **(show-learning 100)** et montre aussi une diminution du nombre de tentatives passant d'environ 1,9 à environ 1,7 pour 100 blocs d'apprentissage de 100 expériences chacun.

Dans le code, nous avons modifié le nombre d'expériences par bloc en le portant à 1000 afin d'évaluer la capacité de notre modèle à apprendre sur un grand nombre de répétitions."

```
(defun show-learning (n &optional (graph t))
   (let ((points))

      (dotimes (i n)
         (push (run-blocks 1 1000) points)) ;; ici pour des blocs de 1000
      (setf points (reverse points))
      (when graph
         (draw-graph points)
      )
   )
)
```

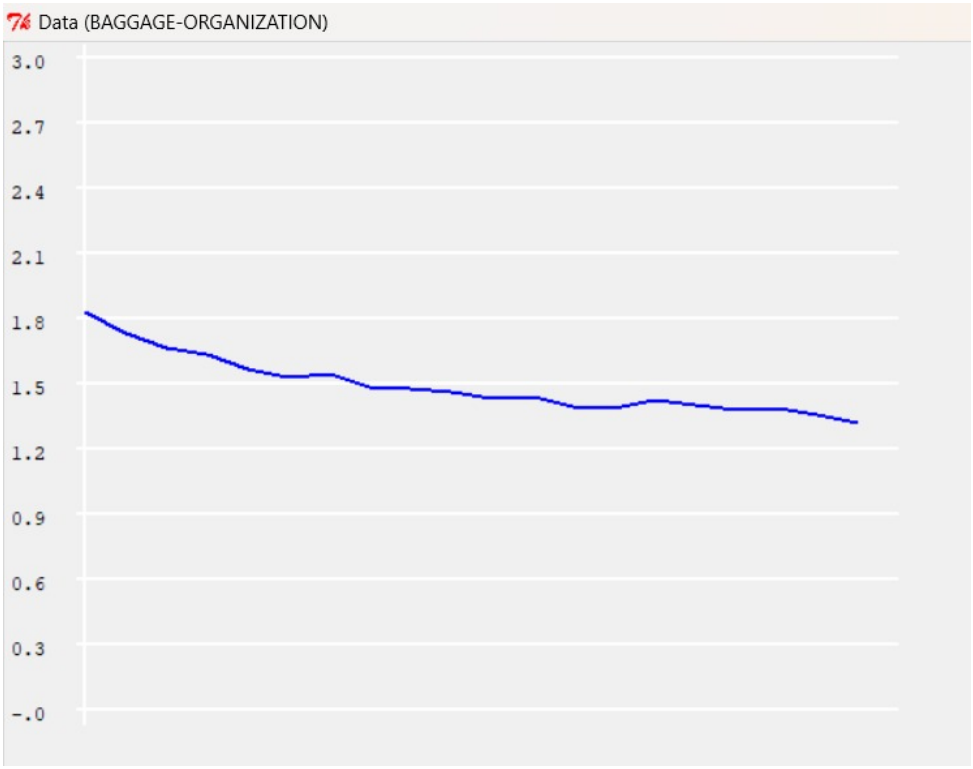

FIG. 6.3 – Graphe de 20 blocs de 1000 expériences

Le graphe ci-dessus généré avec la commande (show-learning 20), utilisant un total de 20 blocs de 1000 expériences, montre une amélioration notable dans l'apprentissage du modèle ACT-R. On observe une diminution significative de la moyenne des tentatives, passant de 1,8 à environ 1,3.

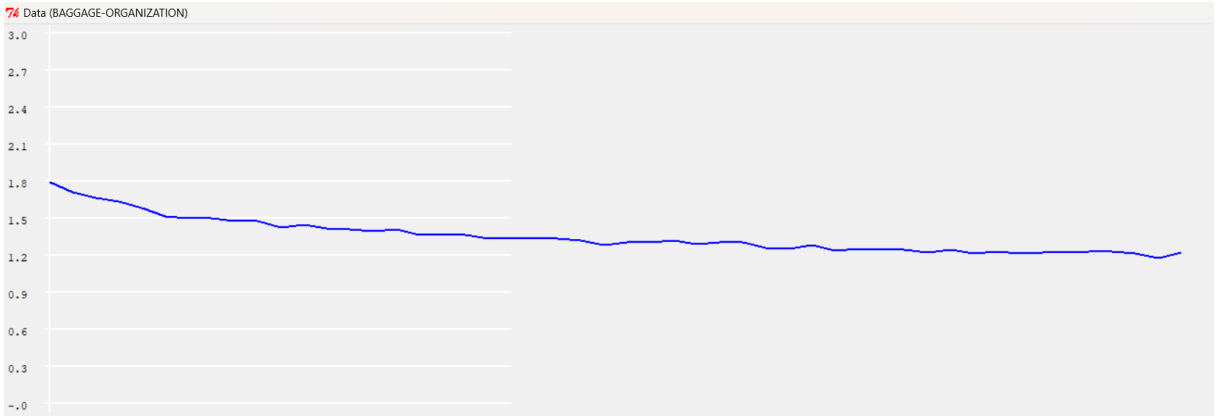

FIG. 6.4 – Graphe de 100 blocs de 1000 expériences

Pour le dernier graphe, nous avons utilisé la commande (show-learning 100), qui génère un total de 100 blocs de 1000 expériences. Ce graphe montre une diminution de la moyenne des tentatives, passant de 1,8 à environ 1,3. Cependant, on remarque que après un certain nombre de blocs, les performances du modèle stagnent et la courbe s'aplatit, se stabilisant autour de 1,3. Cela montre que malgré l'augmentation du nombre d'expériences, le modèle atteint un plateau dans son apprentissage, ne parvenant plus à améliorer ses performances après un certain seuil de répétitions.

Conclusion

En conclusion, ce projet nous a permis de développer et d'améliorer un modèle ACT-R capable d'organiser des bagages de manière optimale dans un coffre, en tenant compte de leurs dimensions, de leur poids et de leur catégorie. À travers les différentes étapes de cette démarche, nous avons surmonté plusieurs défis techniques et théoriques, notamment le placement illogique des bagages, la gestion du nombre variable de valises et la prise en compte des priorités de poids et de catégorie. Nous avons également exploré et tenté de résoudre une quatrième limite liée à la hauteur variable des bagages.

L'adaptation du modèle pour intégrer ces contraintes a permis d'obtenir une solution robuste et flexible, capable de s'ajuster à divers scénarios de placement. En outre, les graphes d'apprentissage et les résultats des simulations ont montré que notre approche permet une organisation stable et efficace des bagages, tout en respectant les critères d'optimisation d'espace et de sécurité.

Le choix de ce modèle présente néanmoins certaines lacunes, notamment en ce qui concerne la modélisation réaliste des contraintes physiques, comme la hauteur variable des bagages et la gestion de l'espace en fonction des formes irrégulières des valises. Pour améliorer notre approche, des expériences concrètes pourraient être envisagées afin d'ajuster ces paramètres de manière plus réaliste, par exemple en intégrant des données empiriques sur les dimensions et les poids des bagages réels. De plus, des ajustements supplémentaires pourraient être réalisés en modifiant les règles de priorisation, pour tenir compte d'autres facteurs comme la flexibilité du coffre ou les interactions entre les bagages. Ces améliorations permettraient de rendre le modèle encore plus performant et adapté à des scénarios pratiques de gestion des espaces.

Bibliographie

[1] *ACT-R 7.28+ Reference Manual*. URL : http://act-r.psy.cmu.edu/actr7.x/reference-manual.pdf.