Erwan Normand
Jean Queval

IA01 - A13
9 pages

Compte rendu du TP03 : Conduite d'expertise d'un SE ordre 0+

Survie sur une île déserte

Après un accident d'avion, vous vous réveillez, seul. Très vite vous comprenez : vous devez survivre dans ce milieu hostile, seul ou peut être pas… Lost, Man vs. Wild, Robinson Crusoé avoir lu ces livres et vu ces séries ne vous suffira pas, heureusement pour vous, le système expert est là pour vous aider !

Ce TP nous a permis de réaliser un système expert d'ordre 0+.
Un système expert est un programme informatique qui « raisonne sur des problèmes dont la résolution repose sur une expertise humaine dans un domaine délimité. » (Edward Feigenbaum)
L'utilisation d'un tel système est justifiée quand la résolution du problème ne connaît pas d'algorithme connu ou de modèle mathématique.
Pour raisonner, un SE se compose de trois parties : une base de règles qui indique le raisonnement à suivre, une base de faits que donne l'utilisateur, et un moteur d'inférence qui fait fonctionner les règles à partir de ces faits pour obtenir de nouveaux faits jusqu'à atteindre à la réponse à la question posée
Le SE proposé est d'ordre 0+ : les faits sont représentés comme des couples (attribut valeur).

Nous présenterons tout d'abord notre problématique et l'expertise, puis la représentation des connaissances, faits et règles, dans les bases de faits BF et la base de règle BR, ainsi que des jeux d'essais. Enfin, nous décrirons et comparerons les deux moteurs : en chaînage avant et en chaînage arrière, tous les deux en profondeur d'abord.

1. Problématique et expertise

1.1 Problématique

Le système expert va vous dire si vous pourrez survivre ou non dans votre environnement.
Survivre implique de satisfaire, entre autre, trois besoins fondamentaux : la faim, la soif et la santé physique *(La Pyramide des besoins inspirée d'Abraham Maslow).* La deuxième condition à remplir pour survivre est de rendre l'île sure, autrement dit il faut se débarrasser des dangers si ils existent (des animaux sauvages dangereux, par exemple)..
Afin d'assurer chacun de ces besoins, il existe plusieurs stratégies. Elles requièrent des objets et/ou des actions pour les réaliser ; qui elles même ne seront possibles qu'en fonction de l'état de la personne (états de faim, de soif, de fatigue), de ce qu'elle possède avec elle, de ce qui se trouve et de ce qui est accessible dans l'environnement.
Si ces trois besoins ne sont pas correctement satisfaits, la survie ne pourra être possible. S'ils sont partiellement satisfaits, la survie pourra être possible de un à quelques jours. S'ils sont

totalement satisfait la survie (physique : la solitude n'est pas agréable à un être humain) est possible. De plus si les dangers ne sont pas résolus, la survie, partielle ou totale est impossible.

Les but cherché est : la survie. En fonction des faits, le système indiquera si elle sera satisfaite, moyenne ou non satisfaite. Il y a donc 2 buts possibles (survie satisfaite), (survie moyenne) ou la survie n'est pas assurée.
Les faits correspondent à l'état de la personne, de ses possessions, et de ce qui est accessible dans l'environnement. Les règles déclenchées correspondront à des actions possibles pour la personne, et qui, si elles sont déclenchées viendront enrichir la base de faits. De nouveaux faits peuvent être ajoutés ou modifiés.

1.2 Expertise

Nous nous sommes vite rendu compte que le nombre de situation possible sur une île était énorme. En effet la biodiversité et les outils que l'on peut se construire rendent le nombre de faits presque infini. Nous avons donc décidé de restreindre le sujet, et de choisir les situations les plus probables. Nous traiterons le cas d'une île déserte tropicale, et la survie se passe à court terme (pas d'agriculture…).
Cependant certaines techniques plus ou moins compliquées paraissent indispensable pour survivre dans la nature. Par exemple faire du feu ou construire un couteau sans aucun outil n'est pas évident. On trouve de nombreux documents sur internet (scoutisme, tutoriaux, forums…) qui expliquent ces techniques, nous avons cité nos références ci-dessous.
Il existe aussi de nombreuses autre techniques, plus originales. L'évaporation de l'eau salé, par exemple. Nous avons cité quelques unes de ces techniques, les plus utiles, dans notre base de règle.
Pour les animaux et les fruits, les cas que nous avons traité sont les plus communs. Les animaux cités (serpent, moustique, crocodile…) sont les animaux les plus dangereux en terme de nombre de décès causés par an.

Références:

- http://www.toujourspret.com/plan/tech/
- http://vivrelanature.teamgoo.net/
- Echelle de la faim : http://healthiestregards.com/2010/12/28/hunger-scale/
- http://www.wikihow.com/Live-on-a-Deserted-Island

2. Représentation des connaissances

2.1 Base de faits BF

Nous avons choisis de représenter la base de faits BF comme une liste de faits `(fait_1 fait_2 ... fait_n)`, où un fait correspond à un couple `(attribut valeur)`.
Exemple de base de faits : `((faim 3) (bananes nombreuses) (couteau oui))`.

Quelques exemples fonctionnels se trouvent dans la partie 2.3 Jeux d'essais.

2.2 Base de règles BR

La base de règles BR est représentée comme une liste de règles `(R1 R2 ... RN)`.
Une règle R contient une liste d'actions, une liste de conditions et une chaîne de caractères, optionnelle : `((action_1 action_2 ... action_n) (condition_1 condition_2 condition_n) "description")`.
Actions et conditions sont des triplets `(attribut valeur opérateur)` et l'opérateur est optionnel. Pour une condition les opérateurs peuvent être `< <= >= !=`. `equal` utilisé par défaut si l'opérateur n'est pas précisé. Pour une action, les opérateurs peuvent être `+ - * /`.

Une règle ne peut être déclenchée que si toutes ses conditions sont satisfaites. Alors ses actions peuvent être exécutées pour mettre à jour la base de faits. Plusieurs règles peuvent atteindre un même but ; c'est-à-dire pour que le but soit atteint, au moins une règle doit être déclenchée.
Une règle est candidate si elle peut atteindre le but. C'est-à-dire si après mise à jour de la base de faits BF par les actions de la règle le but est vérifié dans BF.

Exemples de règles :

- R1 : `(((survie satisfaite)) ((faim 6 >=) (soif 6 >=) (sante 6 >=) (danger 0 =)) "Survie satisfaite")`
- R3 : `(((faim 3 +)) ((bananes beaucoup)) "Mangez les bananes avec la peau pour ne pas gacher de vitamines !")`
- R20 : `(((grande_feuille oui) (bois oui)) ((palmier present)))`

2.3 Jeux d'essais

Définissons les buts possibles : (survie satisfaite) ou (survie moyenne).

Voici quelques jeux d'essais :

- Vous venez d'arriver, vous avez un peu faim et soif et les fruits sont en abondance.
  ```
  (defparameter *BF* '((faim 3) (soif 4) (sante 8) (danger 0) (bananes beaucoup) (noix_de_coco verte) (temps pluie) (palmier present)))
  ```
- Vous avez un nouveau compagnon, pas très câlin.
  ```
  (defparameter *BF* '((faim 3) (soif 4) (sante 8) (danger 1) (bananes beaucoup) (noix_de_coco verte) (temps pluie) (palmier present) (crocodile oui)))
  ```
- Malgré votre statut de naufragé, rien de vous empêche de vous lancer dans la maroquinerie : aujourd'hui, on va faire des sacs à main en crocodile.
  ```
  (defparameter *BF* '((bananes beaucoup) (temps pluie) (palmier present) (noix_de_coco verte) (ile grande) (faim 3) (soif 4) (sante 8) (danger 1) (crocodile 1)))
  ```

3. Moteurs d'inférences

3.1 Fonctions de services

Nous avons tout d'abord écrit des fonctions de services. Elles nous permettent d'exploiter les règles, la base de faits, exploiter les actions et afficher. Elles constituent une couche d'abstraction de la construction des bases de faits et de règles, et permettent de simplifier le code des moteurs, qui est alors plus lisible.
Nous avons aussi créer des fonctions pour appliquer les actions d'une règle. Les voici:

Fonctions des règles

- `(actions regle)` : renvoie la liste des actions de la règle
- `(conditions regle)` : renvoie la liste des conditions de la règle
- `(description regle)` : renvoie la description de la règle, si elle existe, sinon NIL

Fonctions des faits, des actions et des conditions (ce sont des tuples)

- `(attribut tuple)` : renvoie l'attribut du tuple
- `(valeur tuple)` : renvoie la valeur du tuple
- `(operateur triplet)` : renvoie l'opérateur du triplet (action ou condition), s'il existe, sinon NIL
- `(type_condi)` : renvoie le type de la condition : numérique si elle a un opérateur, fait sinon

Fonction de gestion de la base de faits BF

- `(ajouter_faits BF actions)` : pour chaque actions d'une règle, on analyse si le l'attribut du fait est présent ou non dans la base de fait. S'il ne l'est pas, on l'insère dans BF, sinon on le modifie à l'aide de la prochaine fonction.
- `(modifier_fait BF action)` : les variables lisp sont des pointeurs, pour modifier les valeurs on doit donc faire une copie de la valeur à l'aide de la fonction "copy-seq". On empile ensuite le fait composé de l'attribut et de la nouvel valeur du fait que nous calculons ci-dessous. La fonction retourne ensuite la nouvelle BF entière.
- `(valeur_apres_action (BF action)` : si l'attribut est présent dans BF et si cette action est numérique, alors la fonction réalise le calcul de la nouvelle valeur, sinon elle renvoie la valeur de l'action. Le résultat retourné est borné par la fonction `limite`.
- `(limite max valeur)` : retourne 0 si la valeur calculée est négative, max si elle dépasse la valeur limite, ou la valeur sinon.

Affichage

`tabulation` et `affichage_description` sont des fonctions utilisées par le moteur arrière pour afficher les descriptions des règles et rendre compte de la profondeur.

3.1 Moteur en chaînage avant en profondeur d'abord

En chaînage avant, toutes les règles possibles sont déclenchées jusqu'à atteinte du but. Ou l'impossibilité de l'atteindre: cela se produit s'il n'y a plus de règles déclenchables avant que le but ne soit atteint. Ce modèle est intéressant car il permet de visualiser la progression du

raisonnement jusqu'au but. Cependant elle effectue un trop grand nombre d'opérations : toutes les branches de l'arbre possible sont testées. Pour réduire cette complexité l'algorithme s'arrête lorsqu'une solution est trouvée, et décris celle-ci.

Description des principales fonctions:

`(declencher-moteur-avant *BR* *BF* 10 *buts*)`
La fonction entrée par l'utilisateur. Elle teste les buts dans l'ordre (survie satisfaite, survie moyenne), et s'excuse si aucune solution n'est possible ! Si le but est trouvé, celui-ci sera modifié. A la fin de cette fonction on lui redonne sa valeur initiale.

`(moteur-avant BR BF max deja_fait but)`
Pour tester chacun de ces buts, on utilise cette fonction récursive. L'algorithme va tester, pour chaque chaque règle possible si le but est atteint ou non. S'il est atteint, il affichera toutes les règles dotées d'une description dans l'ordre. Il modifiera aussi la valeur du but, pour envoyer ainsi un message à toutes les opérations en cour et arrêter la recherche. S'il ne l'est pas, il ré effectuera toutes les règles possible, sauf celles déjà effectuées.

`(regle_possible? BR deja_vu max BF)`
Teste si chaque règle est applicable ou non, et retourne la liste des règles possibles.

`(possible? regle deja_vu max BF)`
Teste si une règle est possible. Pour qu'elle le soit il ne faut pas qu'elle soit deja présente dans la liste "deja_vu", et que ses conditions soient respectées par la base de fait.

`(test_conditions condis BF)`
Cette fonction teste si les conditions sont respectées par la base de fait, retourne T si elles le sont et NIL sinon. Voici l'algorithme:

test_conditions (conditions BF)
SI (first (conditions) == NIL) return T //conditions d arret si tous les éléments ont été traités.
SI (type (first (conditions)) == fait) ALORS //type(condi) est la fonction pour tester le type d'une règle
    SI first (conditions) EST PRESENT DANS BF return TEST_CONDI (rest (conditions)
    SINON return NIL //La condition est manquante, la règle ne peut être déclarée
SINON SI (type (first (conditions)) == numérique) ALORS
    SI (respect_règle_numérique(first (conditions), BF) = T return TEST_CONDI (rest (conditions)
    SINON return NIL

3.2 Moteur en chaînage arrière en profondeur d'abord

En chaînage arrière, le moteur va parcourir l'arbre des règles jusqu'à retrouver les faits permettant le but possible.

Le moteur est organisé en plusieurs fonctions :
`(declencher_moteur_arriere *BR* *BF* *buts*)`
La fonction entrée par l'utilisateur. Elle teste les buts dans l'ordre (survie satisfaite, survie moyenne), et s'excuse si aucune solution n'est possible !

`(moteur_arriere but BF BR)`
Retourne t ou nil si le but est vérifié ou pas : le moteur regarde tout d'abord si la base de faits BF vérifie le but. Sinon elle liste l'ensemble des conflits, c'est-à-dire les ensembles de règles candidates (qui vérifient le but). Dès qu'un ensemble est vérifié, le but l'est aussi.

`(ensemble_conflits but BF BR)`
Retourne la liste des ensembles de règles candidats. Pour cela, la fonction passe en revue toute la base de règles, et dès qu'une règle vérifie le but, elle est ajouté à la liste. Enfin, la fonction ajoute à la liste l'ensemble des règles qui ne peuvent vérifier le but seules, mais en combinaison avec d'autres règles.
Par exemple : `(ensemble_conflits '(faim 6 >=) *BF* *BR*)` retourne `((R3) (R4) (R9) (R10) (R16) (R37) (R5 R11 R28))`. R3 vérifie le but seule, mais pas R5, R11 et R28. Ces trois dernières ne peuvent atteindre le but qu'en association avec l'une ou les deux autres règles. Mais à ce stade, on ne sait pas quelles combinaisons de ces trois règles peuvent vérifier le but.

`(verifie_but? tuples but BF)`
Prédicat qui indique si les tuples (BF ou actions d'une règle) vérifient le but. La fonction cherche si l'attribut du but est présent dans la liste des tuples, puis teste le but par rapport aux valeurs des tuples. Si le but est juste un couple (attribut valeur), elle teste l'égalité des valeurs. Si le but contient un opérateur, elle utilise cet opérateur à la place de l'égalité pour tester les valeurs.

`(ensemble_candidat? ensemble but BF)`
Prédicat qui indique si l'ensemble atteint le but. La fonction passe en revue toutes les règles, et calcule la valeur obtenue par l'ensemble des règles après application de leurs actions, pour l'attribut du but. Enfin elle renvoie si cette valeur atteint le but. Alors l'ensemble est candidat.

`(regle_verifiee? regle BF BR)`
Prédicat qui indique si la règle est vérifiée. La fonction établit chacune de ses conditions comme de nouveaux sous-buts à établir. S'ils sont tous vérifiés, la règles est vérifiée.

`(ensemble_verifie? ensemble but BF BR)`
Semi-prédicat qui renvoie l'ensemble s'il est vérifié, c'est-à-dire s'il est candidat et si ces règles sont vérifiées, sinon nil. Cette fonction est utilisée par `moteur_arriere` sur chacun des ensembles de `ensemble_conflits`.
Si l'ensemble contient une seule règle, on sait qu'elle est déjà candidate, il suffit simplement de la vérifier avec `regle_verifiee?`. Alors l'ensemble est vérifié.
Si l'ensemble contient plusieurs règles, la fonction vérifie chacune des règles à la suite les unes des autres. A chaque nouvelle règle vérifiée, la fonction cherche si l'ensemble des règles vérifiées est candidat. S'il est, alors l'ensemble des règles vérifiées est candidat et vérifié : la fonction cherche les combinaisons de règles candidates et vérifiées. Dès qu'une combinaison est candidate et vérifiée, la fonction la renvoie.

3.3 Résultats
Testons le fonctionnement détaillé des moteurs avec le premier jeu d'essai :

```
(defparameter *BF* '((faim 3) (soif 4) (sante 8) (danger 0) (bananes
beaucoup) (noix_de_coco verte) (temps pluie) (palmier present)))
```

3.3.1 Moteur avant

A la première itération de “moteur-avant”, les règles possibles sont (R3 R20 R11). Il les appliquera toutes indépendamment des autre. Appliquons R20, par exemple, (grande_feuille oui) et (bois oui) font maintenant partis de la base de fait. Voici la suite des opérations qui permettent d’atteindre le but:

```
R42 => (soif + 1)
 *BF* : ((bananes beaucoup) (temps pluie) (palmier present)
(noix_de_coco verte) (faim 3) (soif 5) (sante 8) (danger 0)
(grande_feuille oui)  (bois oui)))

R11 => (faim 1 +) (recipient moyen)
 *BF* : ((bananes beaucoup) (temps pluie) (palmier present)
(noix_de_coco verte) (faim 4) (soif 5) (sante 8) (danger 0)
(grande_feuille oui)  (bois oui) (recipient moyen)))

R13 => (soif 1 +)
 *BF* : ((bananes beaucoup) (temps pluie) (palmier present)
(noix_de_coco verte) (faim 4) (soif 6) (sante 8) (danger 0)
(grande_feuille oui)  (bois oui) (recipient moyen)))

R3 => (faim 3 +)
 *BF* : ((bananes beaucoup) (temps pluie) (palmier present)
(noix_de_coco verte) (faim 7) (soif 6) (sante 8) (danger 0)
(grande_feuille oui)  (bois oui) (recipient moyen)))

R1 => (survie satisfaite). En effet, on a bien (soif > 6) (faim > 6)
(sante > 6) (danger = 0). Le but “survie satisfaite” est atteint et la
recherche s’arrête ici.
```

3.2.1 Moteur arrière

Le moteur explore toutes les possibilités en profondeur d'abord.
La profondeur de la recherche est indiqué par le chiffre au début de la ligne. Pour chaque but cherché, le moteur indique le but et les règles qu'il cherche à vérifier. Quand un but est vérifié le moteur l'indique ainsi que les règles qui ont pu l'établir, et passe au but suivant qu'il fallait établir pour vérifier la règle. Pour qu'un but ne soit pas vérifié, aucun ensemble de règle vérifiées ne peut vérifier le but.

```
0 : But: (SURVIE SATISFAITE) - Regles cherchées : (R1)
1 : --But: (FAIM 6 >=) - Regles cherchées : (R3)
2 : ----But vérifié: (BANANES BEAUCOUP) (Fait : (BANANES BEAUCOUP))
1 : --"Mangez les bananes avec la peau pour ne pas gacher de vitamines !"
1 : --But vérifié: (FAIM 6 >=) (Regles : (R3)) ; la faim est ok
1 : --But: (SOIF 6 >=) - Regles cherchées : (R6)
```

```
2 : ----But non vérifié: (NOIX_DE_COCO MARRON)
1 : --But: (SOIF 6 >=) - Regles cherchées : (R7)
2 : ----But non vérifié: (NOIX_DE_COCO ORANGE)
1 : --But: (SOIF 6 >=) - Regles cherchées : (R12)
2 : ----But non vérifié: (RECIPIENT BEAUCOUP)
1 : --But: (SOIF 6 >=) - Regles cherchées : (R26)
2 : ----But: (FEU OUI) - Regles cherchées : (R31)
3 : ------But: (BOIS OUI) - Regles cherchées : (R20)
4 : --------But vérifié: (PALMIER PRESENT) (Fait : (PALMIER PRESENT))
3 : ------But vérifié: (BOIS OUI) (Regles : (R20))
3 : ------But: (COUTEAU OUI) - Regles cherchées : (R32)
4 : --------But: (SILEX TROUVE) - Regles cherchées : (R33)
5 : ----------But non vérifié: (ILE GRANDE)
4 : --------But non vérifié: (SILEX TROUVE)
3 : ------But non vérifié: (COUTEAU OUI)
2 : ----But non vérifié: (FEU OUI)
1 : --But: (SOIF 6 >=) - Regles cherchées : (R27)
2 : ----But: (EAU NON-POTABLE) - Regles cherchées : (R40)
3 : ------But non vérifié: (RIVIERE SAUVAGE)
2 : ----But non vérifié: (EAU NON-POTABLE)
1 : --But: (SOIF 6 >=) - Regles cherchées : (R44)
2 : ----But non vérifié: (CACHETTE RHUM)
1 : --But: (SOIF 6 >=) - Regles cherchées : (R8 R13 R42)
3 : ------But vérifié: (NOIX_DE_COCO VERTE) (Fait : (NOIX_DE_COCO VERTE))
3 : ------But: (COUTEAU OUI) - Regles cherchées : (R32)
4 : --------But: (SILEX TROUVE) - Regles cherchées : (R33)
5 : ----------But non vérifié: (ILE GRANDE)
4 : --------But non vérifié: (SILEX TROUVE)
3 : ------But non vérifié: (COUTEAU OUI)
3 : ------But: (RECIPIENT MOYEN) - Regles cherchées : (R9)
4 : --------But non vérifié: (NOIX_DE_COCO MARRON)
3 : ------But: (RECIPIENT MOYEN) - Regles cherchées : (R10)
4 : --------But non vérifié: (NOIX_DE_COCO ORANGE)
3 : ------But: (RECIPIENT MOYEN) - Regles cherchées : (R11)
4 : --------But vérifié: (NOIX_DE_COCO VERTE) (Fait : (NOIX_DE_COCO VERTE))
3 : ------"Ouvrez la noix de coco comme vous le pouvez pour manger la pulpe.
Gardez la carapace comme récipient."
3 : ------But vérifié: (RECIPIENT MOYEN) (Regles : (R11))
3 : ------But vérifié: (TEMPS PLUIE) (Fait : (TEMPS PLUIE))
2 : ----Buts vérifiés: ((SOIF 1 +)) (Regle : R13)
3 : ------But: (GRANDE_FEUILLE OUI) - Regles cherchées : (R20)
4 : --------But vérifié: (PALMIER PRESENT) (Fait : (PALMIER PRESENT))
3 : ------But vérifié: (GRANDE_FEUILLE OUI) (Regles : (R20))
3 : ------But vérifié: (TEMPS PLUIE) (Fait : (TEMPS PLUIE))
2 : ----Buts vérifiés: ((SOIF 1 +)) (Regle : R42)
1 : --"Récuperez l'eau de la pluie à l'aide des récipients"
1 : --"Utilisez les plus grandes feuilles que vous voyez et placez les dans un
trou pour recolter l'eau de la pluie"
1 : --But vérifié: (SOIF 6 >=) (Regles : (R13 R42))
1 : --But vérifié: (SANTE 6 >=) (Fait : (SANTE 8))
1 : --But vérifié: (DANGER 0 =) (Fait : (DANGER 0))
0 : "Survie satisfaite"
0 : But vérifié: (SURVIE SATISFAITE) (Regles : (R1))
```

Le moteur a cherché à vérifier les conditions de la règle R1, qui permet d'établir le but (survie satisfaite) :

- (faim 6 >=) est vérifiée par la règle R3 : les bananes permettent de manger à sa faim.
- (soif 6 >=) est vérifiée par l'ensemble des règles R13, R42.
- La santé et le danger sont vérifiés par la base de faits.

Alors R1 est vérifié et le but atteint : la survie est satisfaite.

3.4 Comparaisons

Les deux moteurs trouvent bien la même survie, mais on voit que les règles ne se déclenchent pas forcement. Le moteur arrière ne va que développer des règles qui auront au final un lien avec la faim, la soif, la santé ou le danger. Le moteur avant, lui, va développer toutes les règles possibles, et retournera le premier ensemble de règle qui a permit de trouver le but. Ainsi certaines règles déclenchés paraissent inutiles. De plus, le nombre d'opérations effectuées par le moteur avant est très élevé lorsque la base de fait contient initialement de nombreux faits. Quand au moteur arrière, il peut boucler à l'infini sur des règles si les faits ne sont pas correctement initialisés dans la base de faits.
Les deux moteurs sont complémentaires : le moteur arrière est utile pour retracer ce qui est nécessaire pour vérifier le but (la survie), le moteur avant va construire le raisonnement à partir de la base de fait.

Le moteur arrière pourrait permettre, aux noeuds de l'arbre, de poser des questions à l'utilisateur sur des faits pour orienter la recherche dans l'arbre. La résolution serait alors très efficace.
Tel que conçu, le moteur arrière affiche tout son raisonnement, où il explore toutes les branches. Une version pratique de ce moteur n'afficherai que la partie du raisonnement qui atteindrait au but.

4. Conclusion

Dans ce TP, nous avons cherché à réaliser une SE sur un problème complexe. Nous avons écrit deux moteurs, de chaînages différents, ce qui nous a permis de l'approcher de deux façons différentes.
Mais malgré que nous ayons encadré le sujet juste pour une île déserte, les possibilités d'actions ce sont tout de même retrouvées nombreuses, voire presque infinies. Par exemple, certaines techniques de survies sont complexes. Il en résulte des raisonnements longs et avec une grande profondeur, où nous atteignons facilement une profondeur de 6. Par exemple : explorer l'île, pour trouver un silex, trouver du bois, pour pouvoir fabriquer un couteau, trouver des noix de coco, et pouvoir boire leur jus en les perçant avec le couteau. Voici un exemple très simple à expliquer, mais déjà rapidement complexe à écrire en terme de règles.

Nous aurions pu en conséquence écrire plus de règles, plus de faits. Cependant, nous avons presque atteint les limites de nos moteurs, malgré du temps passé à améliorer leurs complexités, avec 50 règles formées ainsi. En effet, la gestion de scores (comme les niveaux de faim, de soif, de santé et de danger) est assez lourde.

En résumé, ce TP nous a permis d'essayer de traiter un problème réel, complexe.
Et si vous prenez l'avion, pensez à nous ! Nous espérons que vous n'aurez pas à vous servir de ce TP.