Guide d'utilisation de Mo'K

Gilles Bisson (gilles.bisson@imag.fr)

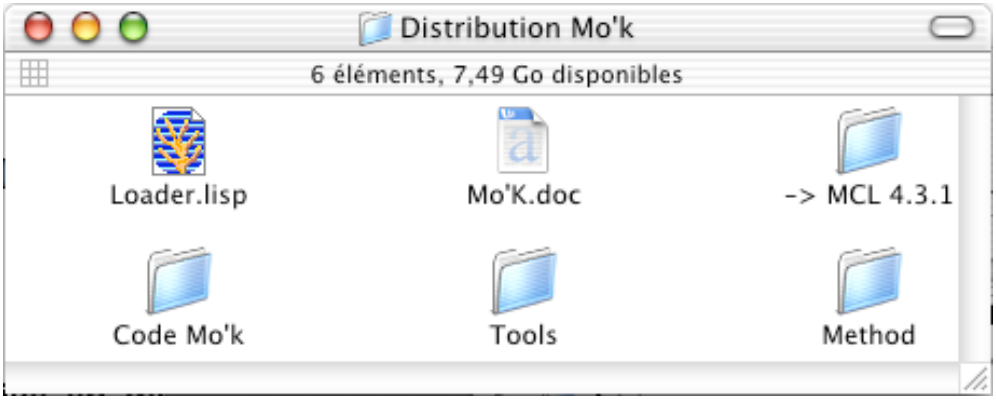
Version du octobre 7, 2002

Contenu du document

1. Mise en œuvre du système

1.1. Installation de Mo'K

Tout d'abord, il faut décompacter l'archive "Système Mo'K.sea" qui a été crée avec DropStuff 6.5. Un double clic sur l'archive est suffisant puisqu'il s'agit d'une archive auto-extractible. On doit obtenir alors un dossier "Distribution Mo'K" contenant les éléments suivant :



Ce dossier peut-être placé absolument n'importe où sur le disque. Par contre, les éléments qui sont (*éventuellement*) contenus dans le dossier nommé "-> MCL 4.x" doivent être copiés directement (autrement dit, au 1er niveau) dans le dossier de MCL 4.x (Mac Common Lisp) installé sur le disque dur. Comme ce dossier ne contient que des patches et des initialisations pour MCL, il est inutile d'effectuer cette opération si cela a déjà été fait lors d'une installation précédente. Une fois l'installation dans MCL terminée on peut sans problème mettre ce dossier dans la poubelle. La *version actuelle de MCL est la 4.3.1*, elle fonctionne sous MacOS 9 et X …

Il faut enfin configurer la mémoire de l'application MCL 4.x à 32 Mo minimum. Plus cette taille est importante plus le système sera à l'aise pour travailler et plus il pourra traiter un grand nombre de phrases. Typiquement il faut environ 64Mo pour pouvoir travailler dans de bonnes conditions avec 100 Kphrases. Pour modifier la taille mémoire, il suffit de faire les opérations suivantes :

❶ sélectionner MCL 4.x (cliquer 1 fois sur l'application)
❷ aller dans le sous-item "Mémoire" de l'item "Lire les informations" du menu "Fichier" du Finder
❸ de mettre 32000 (ou plus) dans le champ "souhaitée" de la boite de dialogue

1.2. Lancement de Mo'K

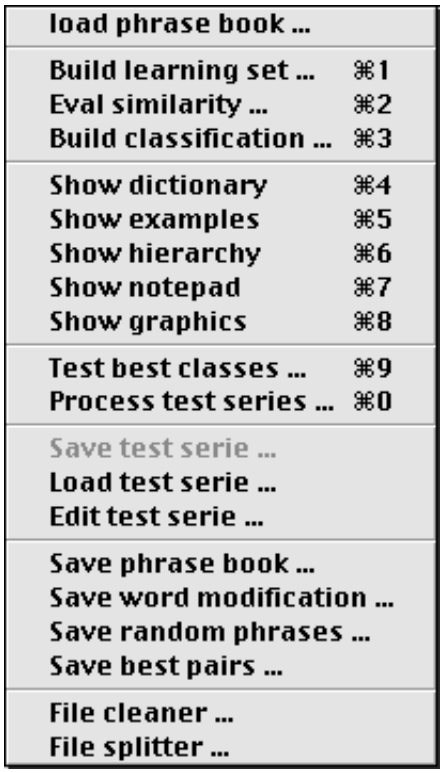
❶ Double-cliquer sur le fichier "Loader.lisp" dans le dossier "Mo'K"
❷ Attendre quelques secondes que la fenêtre "Listener" apparaissent
❸ Sélectionner alors la fenêtre "Loader.lisp" et évaluer son contenu en faisant au choix :

- soit Pomme-H
- soit en sélectionnant la commande "Execute All" du menu "Lisp"

Normalement après quelques secondes le menu "Mo'K" apparaît à la fin de la barre de menu ... On peut alors fermer la fenêtre du fichier "Loader.lisp". En cas de problème : <gilles.bisson@imag.fr>

2. Commandes de Mo'K

Le menu Mo'K contient les commandes suivantes. A un instant donné seules les commandes qui sont effectivement applicables dans le contexte sont activées, les autres restent en grisé. On a ici l'ensemble des commandes qui sont potentiellement actives dans la version actuelle du système.



2.1. Load phrase book

C'est la première commande à exécuter, elle permet de sélectionner un fichier de triplets (appelés « phrases ») et de le charger en mémoire. Le format général d'un fichier de phrases est le suivant :

```
{ ; un commentaire | verbe $ lien $ nom & nombre-d'occurrence}*
```

Lors du chargement les erreurs de syntaxe éventuelles sont affichées dans la fenêtre du Listener et l'on indique le numéro de la ligne fautive dans le fichier de départ.

2.2. Build learning set ...

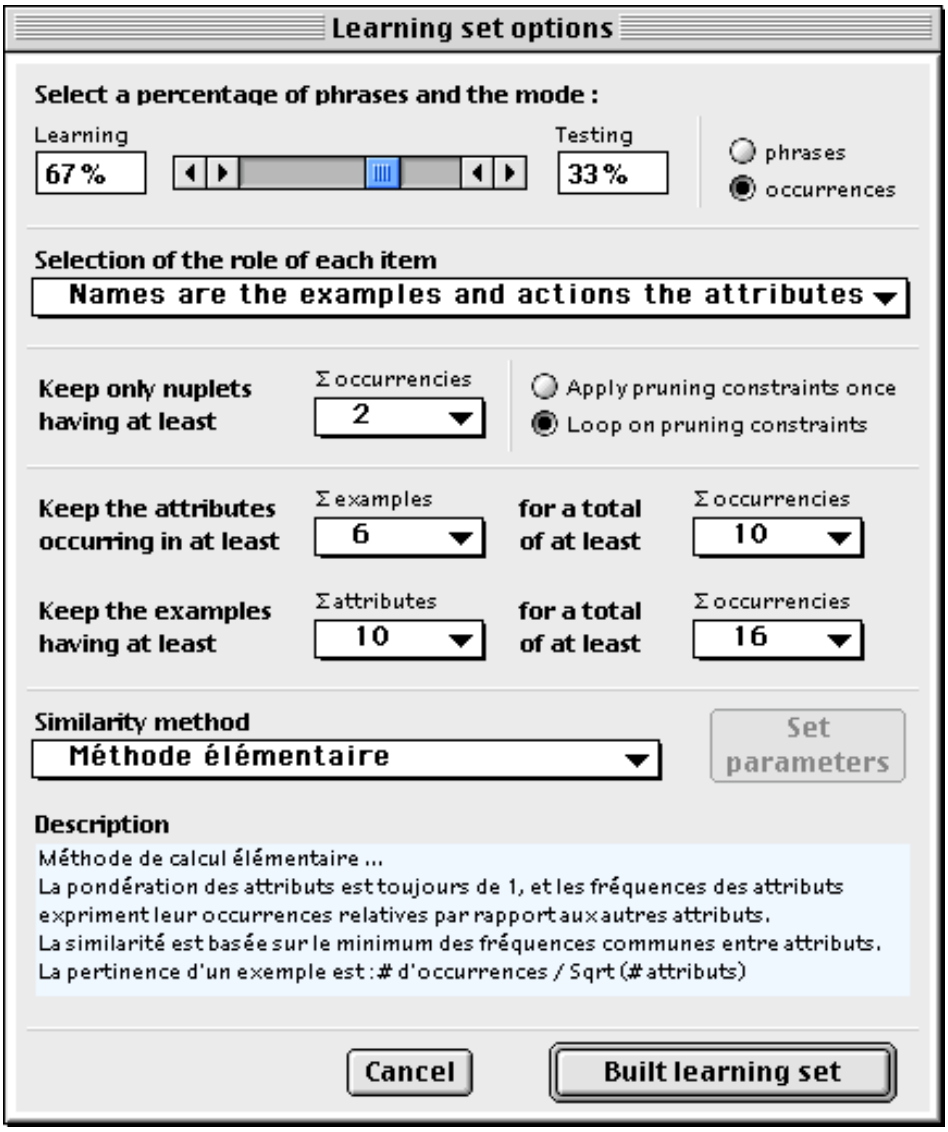
Cette commande permet de sélectionner les phrases qui vont être utilisées d'une part, lors de la phase d'apprentissage pour construire les classes (learning set) et, d'autre part, pour tester la qualité des classes (test set). L'utilisateur paramètre les caractéristiques de ces deux ensembles au moyen d'une boite de dialogue ci-dessous où l'on trouve successivement :

La sélection des deux pourcentages de phrases qui vont être respectivement utilisés pour les ensembles d'apprentissage et de test (attention d'utiliser uniquement la scrollbar pour choisir la partition, les champs ne marchent qu'en affichage). Par ailleurs, l'utilisateur peut également choisir si la partition est effectuée au niveau de la phrase (indépendemment du nombre d'occurrences) ou au niveau des occurrences. Par exemple, dans ce dernier cas, si l'on a un triplet ayant 20 occurrences, avec les options sélectionnées dans la boite de dialogue ci-dessous, 2/3 des occurrences environ iront

dans l'ensemble d'apprentissage et 1/3 dans l'ensemble test. Il est important de noter que ces proportions sont approximatives car la répartition est réalisée à l'aide d'un tirage aléatoire.

Le menu "Selection ..." permet de sélectionner qui jouent le rôle d'exemples et d'attributs.

On sélectionne ensuite le nombre minimal d'occurrences que doit vérifier une phrases pour être prise en compte par le système. Les phrases ne vérifiant pas le critère sont éliminées (mais restent en mémoire). Notons que, dans les exemples après élagage, on peut retrouver des couples <action - mot> ayant un nombre inférieur à la limite fixée : ils correspondent simplement à des couples dont le nombre d'occurrences a été diminué suite à l'élagage d'autres exemples et attributs.



On peut ensuite sélectionner les critères d'élagage propres aux exemples et attributs. Ainsi, pour les attributs, l'utilisateur peut préciser le nombre minimal d'exemples dans lesquels ils doivent apparaître ainsi que le nombre total minimal d'occurrences qu'ils doivent vérifier au total. Pour les exemples ces critères ont des significations similaires. Il faut noter que ces différents critères peuvent être appliqués des deux manières suivantes :

- Une seule fois : dans ce cas on élague tout d'abord les attributs, puis les exemples en tenant compte des attributs qui ont été préalablement supprimés.

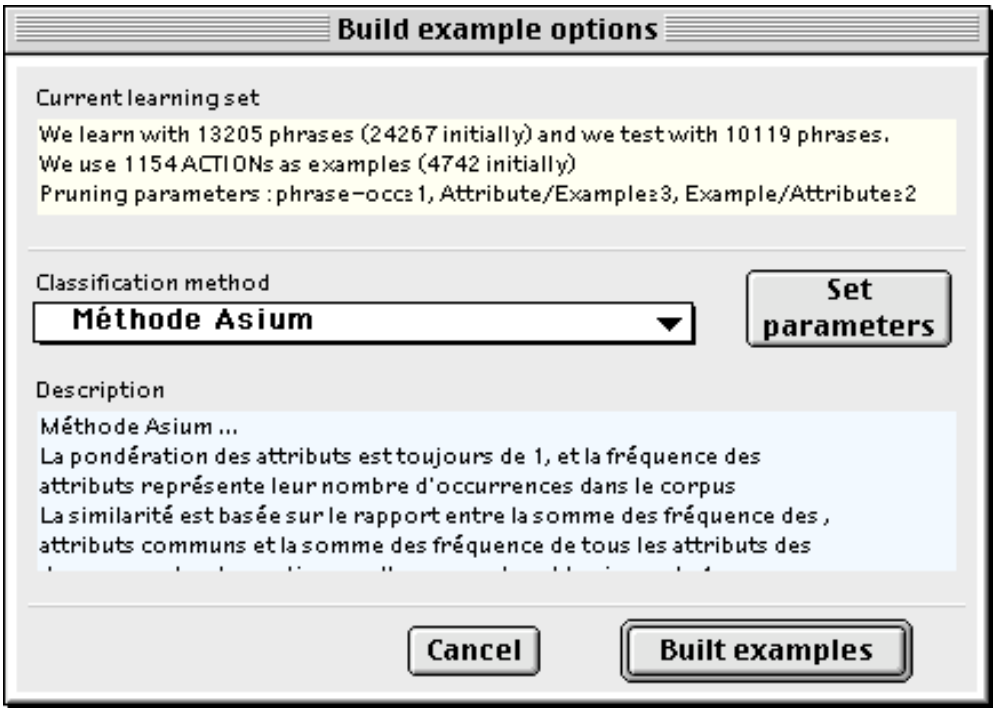
- En boucle : dans ce cas on boucle sur les deux phases de l'élagage jusqu'à ce qu'aucun élément ne soit enlevable. Ce mode se justifie par le fait que supprimer des exemples peut entraîner certains attributs à ne plus vérifier les contraintes (et réciproquement pour les attributs).

Enfin l'utilisateur peut sélectionner la méthode qu'il souhaite appliquer pour mesurer la similarité entre les différents exemples de l'ensemble d'apprentissage. Le fonctionnement de cette partie est le même que pour la commande "Eval similarity" qui est expliqué dans le paragraphe suivant.

2.3. Eval similarity ...

Cette commande permet (re)construire la matrice de similarité entre exemples qui est utilisé pour construire les classes de mots. Elle peut être appelée après que l'utilisateur ait construit un ensemble d'apprentissage afin de modifier la mesure de similarité sans pour autant modifier l'ensemble qui a été construit. La boite de dialogue proposée à l'utilisateur est composée de trois parties.

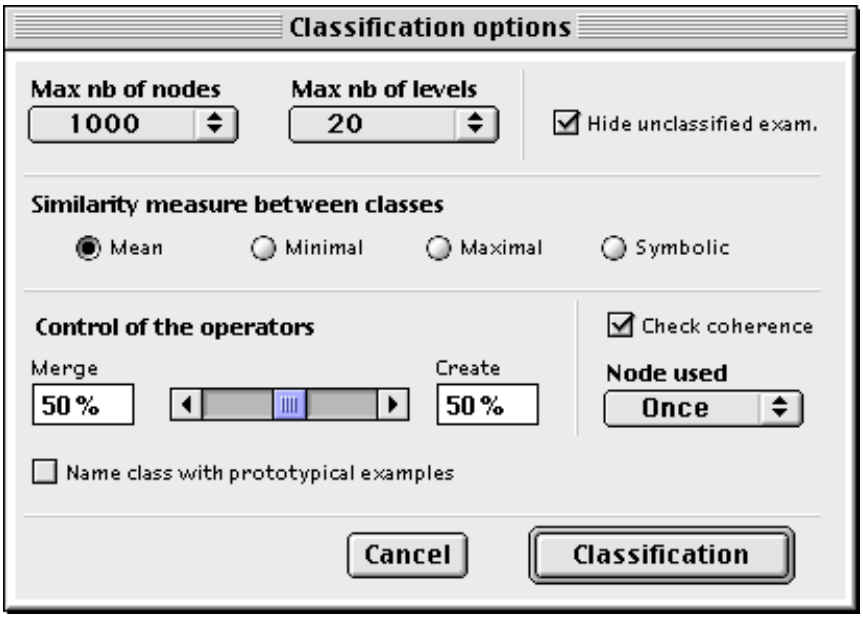
Dans la partie supérieure on trouve un rappel succint des informations qui sont présentes dans l'ensemble d'apprentissage courant et des critères d'élagage utilisés. En dessous, l'utilisateur peut sélectionner le type de méthode de comparaison qui est à employer pour évaluer la ressemblance entre les couples d'exemples. Lorsque la méthode est paramétrable, le bouton "Set parameters" devient actif et, en cliquant dessus, l'utilisateur fait apparaitre une boite de dialogue dans laquelle il pourra saisir les dits paramètres. Enfin, la fenêtre "description" située en bas de la boite de dialogue affiche les commentaires qui sont éventuellement associés à la méthode sélectionnée.



Moyennant un minimum de programmation il est possible de rajouter de nouvelles méthode de comparaison. La démarche est expliquée en détail dans la dernière partie (cf 3.) de ce document.

2.4. Build classification ...

Lance la construction de la hiérarchie. Pour l'instant l'algorithme implémenté est relativement figé et l'utilisateur n'a pas la possibilité de construire de nouvelles méthodes génériques de classification (au sens de ce qui est fait pour les mesures de similarités), on peut cependant configurer les 3 familles de paramètres suivants :



Tout d'abord, on peut contraindre la taille de la hiérarchie qui est construite à la fois en terme de profondeur (level) et de nombre total des nœuds engendrés. On peut ainsi contruire des hiérarchies portant uniquement sur les exemples les plus similaires. Lors des tests il est prudent de commencer avec une limite assez basse et de l'augmenter en fonction des résultats.

L'utilisateur peut également choisir le type de ressemblance « inter-classes » qui est à utiliser lors du regroupement des concepts. On trouve ici trois mesures classique : liens minimal, maximal et moyenne ainsi qu'une mesure dite « symbolique » qui permet de faire ce calcul directement à partir de la description des concepts (entension) en utilisant la même méthode que celle utilisée pour construire la matrice de ressemblance. Il faut noter que la construction de la description d'une classe se fait simplement en faisant l'union des descripteurs (et en sommant les occurrences).

Enfin, la méthode de classification utilise deux opérateurs différents de construction de nouveaux concepts : soit l'agrégation classique des classes (create) , soit la « fusion » qui consiste à ranger le nouvel élément dans une classe déjà existante sans qu'il y ait création d'un nouveau nœud dans la hiérarchie. Le curseur permet de favoriser l'un ou l'autre de ses opérateurs. La case « check coherence » permet de contrôler que l'opérateur de fusion ne construit pas des classes « trop allongées » par l'apparition d'un classique « effet de chaînage ». Le menu « node used » contrôle le degré de polymorphisme (spécialisation multiple des concepts) qui est autorisé dans la hiérarchie.

2.5. Show dictionary

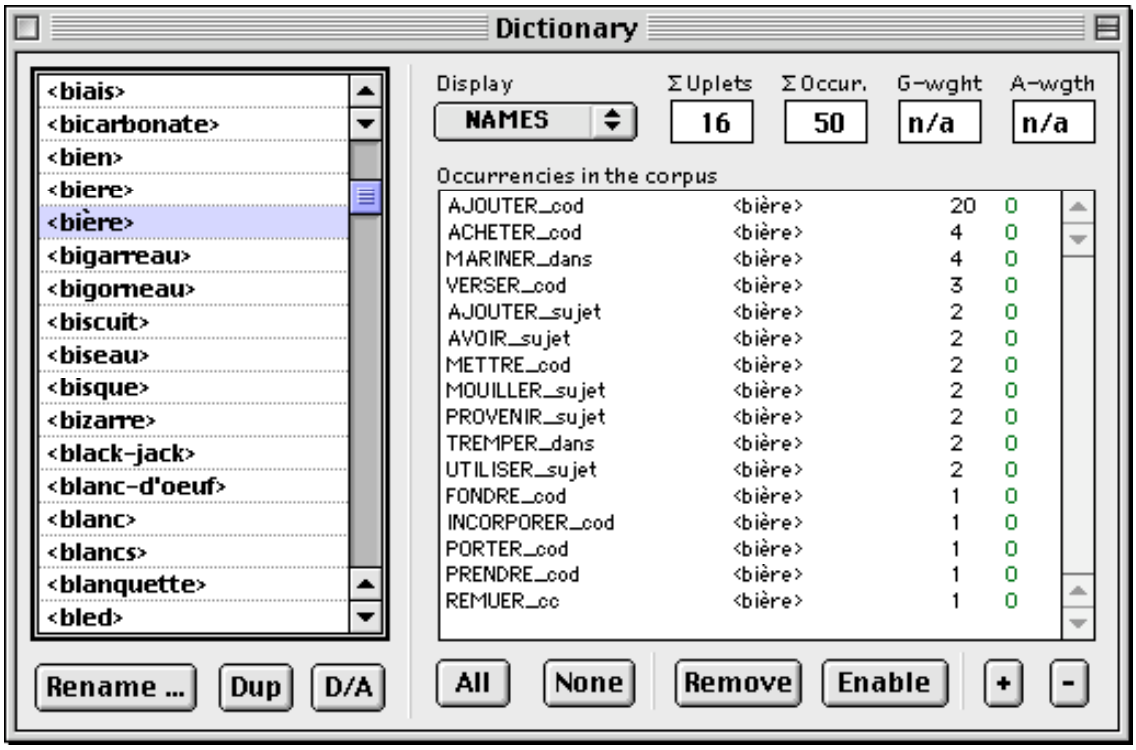
Cette commande affiche une fenêtre permettant de visualiser les différents mots apparaissant dans le corpus en indiquant pour chacun le nombre d'occurrences et les triplets dans lesquels il apparaît. Elle permet également de modifier interactivement le dictionnaire et ainsi soit d'introduire des

corrections dans le jeu d'exemples (réorthographication …), soit de tester des hypothèses de travail, par exemple ne garder, ou inversement supprimer, les compléments d'objet direct …

On utilise un code de couleur précis pour indiquer l'état des différents mots. Dans la liste des triplets, située en bas à droite, les phrases et mots apparaissant en "grisés" sont ceux qui ont été retirés du corpus lors de l'élagage car ne vérifiant pas l'un au moins des paramètres. Cependant, un mot non élagué (exemple ou descripteur) qui apparaît au sein d'une phrase élagué est distingué des autres en étant colorié en marron clair. Les deux colonnes de droites correspondent respectivement au nombre d'occurrences du triplet dans l'ensemble d'apprentissage et de test ; afin de faciliter la lecture cette dernière colonne est affichée en vert. Lorsqu'un mot (et triplet) a été explicitement supprimé de l'ensemble des phrases il est coloré en rouge, lorsque cette suppression est le résultat d'une fusion avec un autre mot (via un renommage) on colore les items concernés en bleu.

Les valeurs de nombres de Nuplets et d'Occurrences qui sont affichées en haut sont celles correspondant aux seuls items actifs c'est à dire des lignes apparaissant en noir dans la table. Le poids correspond au champ correspondant de l'objet. Le menu « display » permet de filtrer le type des items qui apparaissent dans la colonne de gauche.

Notons enfin que lorsque l'on clique sur un mot dans un triplet le sélecteur de gauche se place automatiquement sur ce mot, on peut ainsi très facilement naviguer dans le dictionnaire. Il est par ailleurs possible de se déplacer dans la liste des mots à l'aide des flêches du clavier.



Les possibilités d'édition concernent d'une part les « mots » (verbes, noms, actions …) et d'autre part les triplets correspondant aux différentes occurrences des mots. Les modifications effectuée sur l'une des listes se répercutent automatiquement sur l'autre. Par contre, les modifications ne sont pas directement répercutées dans le calcul de similarité (ce qui serait trop complexe) ; il faut donc relancer le calcul après chaque modification. Les fonctions suivantes sont disponibles :

- Renommage des mots (avec fusion automatique des occurrences si besoin)
- Duplication d'un mot (bouton « Dup »)
- Activation et désactivation d'un mot (avec suppression de l'ensemble des triplets)

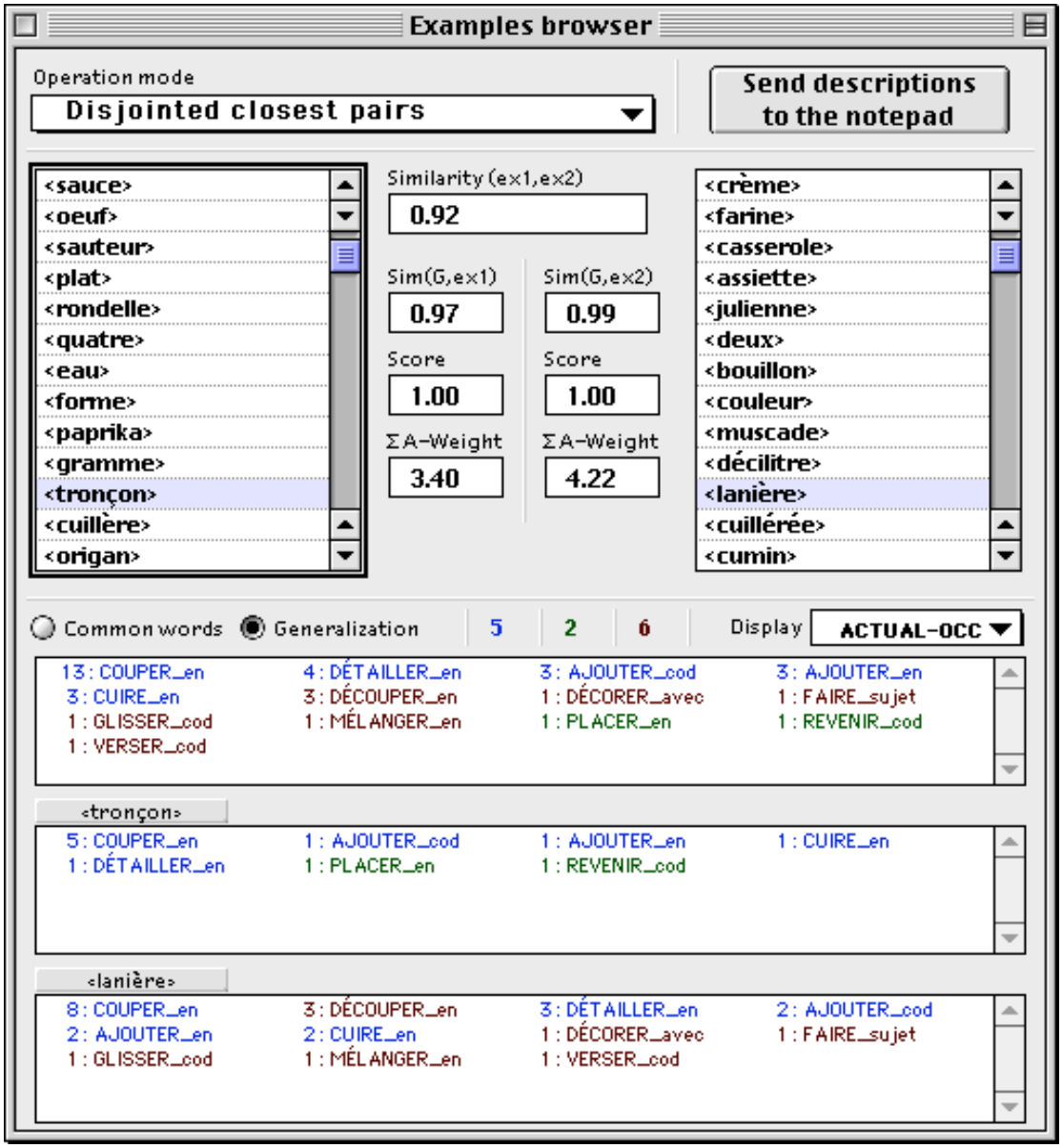
- Sélection des triplets : boutons « al » et « none » (sélection partielle avec la touche « shift »)
- Activation et désactivations d'une partie des occurrences (triplets) d'un mot
- Modification de la fréquence d'un triplet (boutons « + » et « - »)

Le jeu de phrases modifié est sauvable via la commande « Save phrase book … ». En outre les modifications peuvent être elles mêmes sauvées pour constituer des fichiers de suppression (« stop list ») ou de substitutions ré-utilisables via la commande « File cleaner … ».

2.6. Show examples

Cette commande affiche une fenêtre permettant de visualiser et de comparer les différents exemples générés. Le menu "Operation mode" qui permet de sélectionner le type d'affichage parmi :

- Similarité entre deux exemples
- Recherche du plus proche voisin
- Recherche des 400 meilleures paires d'exemples
- Recherche des 400 meilleures paires **disjointes** d'exemples



En dessous de ce menu se trouvent les deux listes (dans lesquelles ont peut naviguer à l'aide des flêches haut et bas du clavier) permettant de sélectionner les couples d'exemples, entre ces listes sont affichées la similarité entre les exemples courants, la similarité entre ces exemples et leur généralisation, la pertinence de chaque exemple et enfin la somme du poids des attributs. La valeur affichée est bien évidemment fonction de la méthode de ressemblance préalablement choisie.

Le bas de la fenêtre est composé de 3 champs affichant successivement (de haut en bas) :

- La description des parties communes entre les deux exemples
- La description de l'exemple de gauche
- La description de l'exemple de droite

Par l'intermédiaire des deux radio-boutons, l'utilisateur peut visualiser dans le champ du haut soit l'intersection, soit la généralisation des descriptions des deux exemples. En outre, à l'aide du menu déroulant il peut choisir le type d'information (Fréquence, Nombre d'occurrences,...) qu'il souhaite faire apparaitre devant chaque attribut. Entre ces deux items on trouve trois compteurs indiquant respectivement le nombre d'éléments communs entre les deux exemples et le nombre d'éléments spécifiques au premier et deuxième exemples.

Notons que lorsque l'on clique sur l'un des littéraux dans une description, la fenêtre dictionnaire est, si ce n'était pas le cas, ouverte (mais **pas** rendue active si elle l'était déjà, en d'autres termes elle reste au second plan si elle l'éatit déjà), et le curseur est positionnée sur la description du mot. De même, on peut cliquer sur le nom des exemples qui apparaissent sous la forme de petit bouton rectangulaire au dessus des descriptions.

Enfin dans le coin supérieur gauche on trouve le bouton "Send descriptions to the notepad" qui permet d'envoyer dans le Calepin (notepad) les descriptions associées aux exemples courants. Cette fonction ne fonctionne que lorsque deux exemples sont sélectionnés.

2.7. Show hierarchy ...

Cette commande affiche le graphe de classification courant. Pour l'instant, du fait de l'absence de certaines optimisations, il est conseillé (particulièrement pour le masquage) de ne travailler qu'avec des graphes ne contenant au plus que quelques centaines de nœuds.

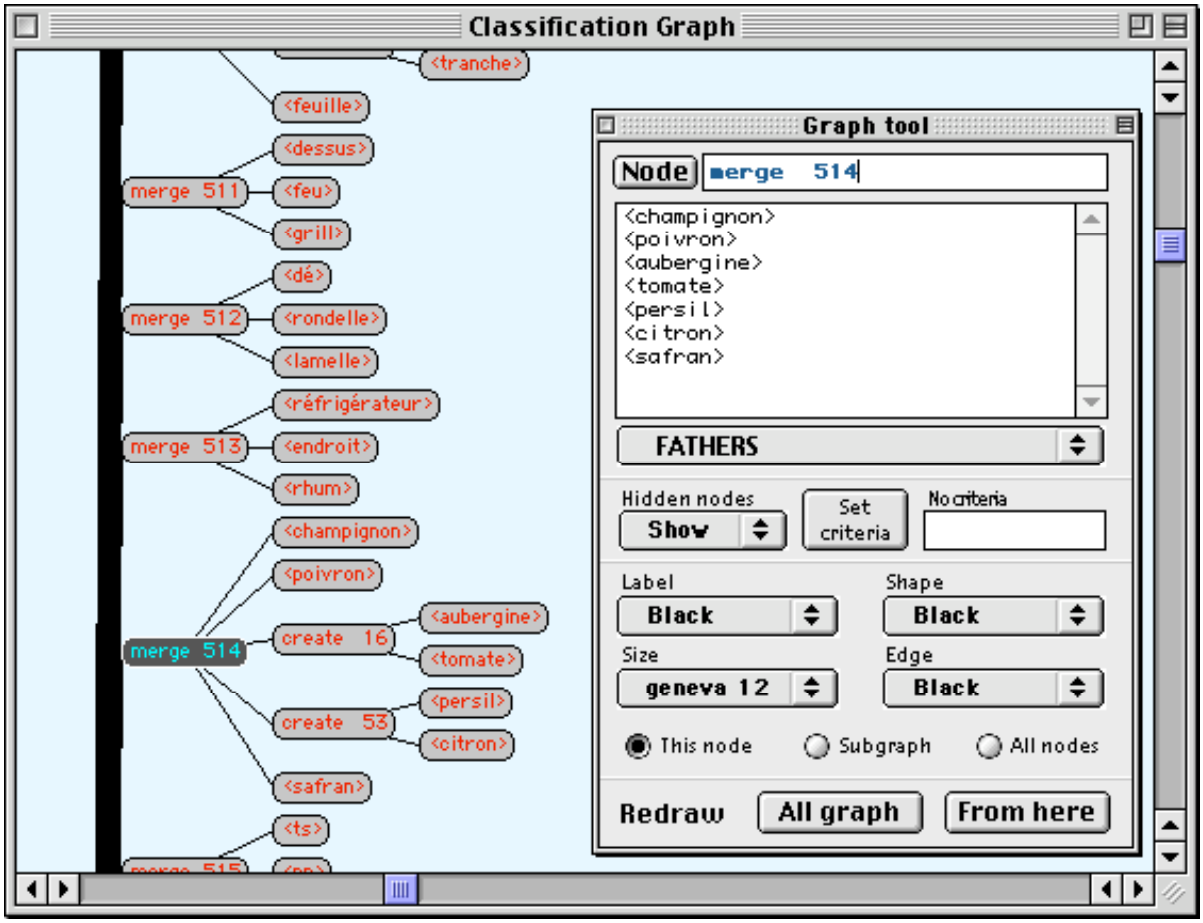
Le grapheur de Mo'K offre plusieurs possibilités de manipulation directement à partir de la souris :

- Cliquer sur un noeud : Fait apparaitre la palette flottante "Graph Tool"
- Double cliquer : Provoque le repliement/dépliement du sous-arbre associé au nœud
- CTRL-double cliquer : Même chose mais en conservant les fils du nœud courant
- SHIFT-cliquer : Masquage/démasquage d'un nœud

La palette "Graph-Tool" permet une plus large plage de possibilités. De haut en bas on trouve :

Le nom du nœud qui est éditable, toute modification étant reportée dans l'ensemble du système (Browser d'exemples y compris) si la modification ne concerne pas une feuille du graphe (exemple). Notons que le bouton placé à gauche du nom permet de se repositionner directement sur le nœud correspondant dans le graphe.

La zone d'information surmontant un menu déroulant qui permet de sélectionner le type de données que l'on veut faire apparaitre concernant le nœud courant. Lorsque cette fenêtre contient un nom correspondant à un nœud du graphe cliquer sur ce nom positionne le graphe sur celui-ci.



Le mode de gestion des “nœuds cachés” parmi (Show, Keep X,Y, Keep X, Hide). Brièvement, dans le mode “Show” les nœuds sont tous visualisés. En “Keep X,Y” les nœuds cachés ne sont pas visibles mais l’organisation spatiale du graphe reste inchangée. En “Keep X”, les nœuds visibles gardent leur position en X mais le graphe est redessiné en Y. Enfin, le mode “Hide” effectue l’affichage optimale du graphe en tassant les nœuds visibles au maximum.

La définition d’un critère de filtrage. Pour cela on sélectionne dans le menu pop-up situé au dessus le type d’information sur laquelle on veut faire le filtrage, par exemple “la pertinence”, puis on appuie sur le bouton “Set criteria”.On peut alors rentrer la valeur du filtre dans la zone placée à droite du bouton, par exemple, “ > 2.3” qui signifie que seuls les nœuds ayant au moins cette valeur de pertinence seront visibles. Lorsque le critère concerne une liste (par exemple les attributs communs) ont peut définir un nombre minimum d’éléments en entrant par exemple “length > 0”.

Dans la partie suivante l’utilisateur peut choisir la couleur de l’étiquette, du fond et du bord des nœuds, ces choix pouvant être appliqués soit au nœud courant, soit au sous-graphe associé, soit à l’ensemble de l’arbre. Le menu SIZE permet de régler le niveau de zoom

Enfin, lorsque le graphe est de taille trop importante, l’utilisateur peut choisir de ne visualiser que le sous-arbre associé au nœud courant en utilisant le bouton “From here”.

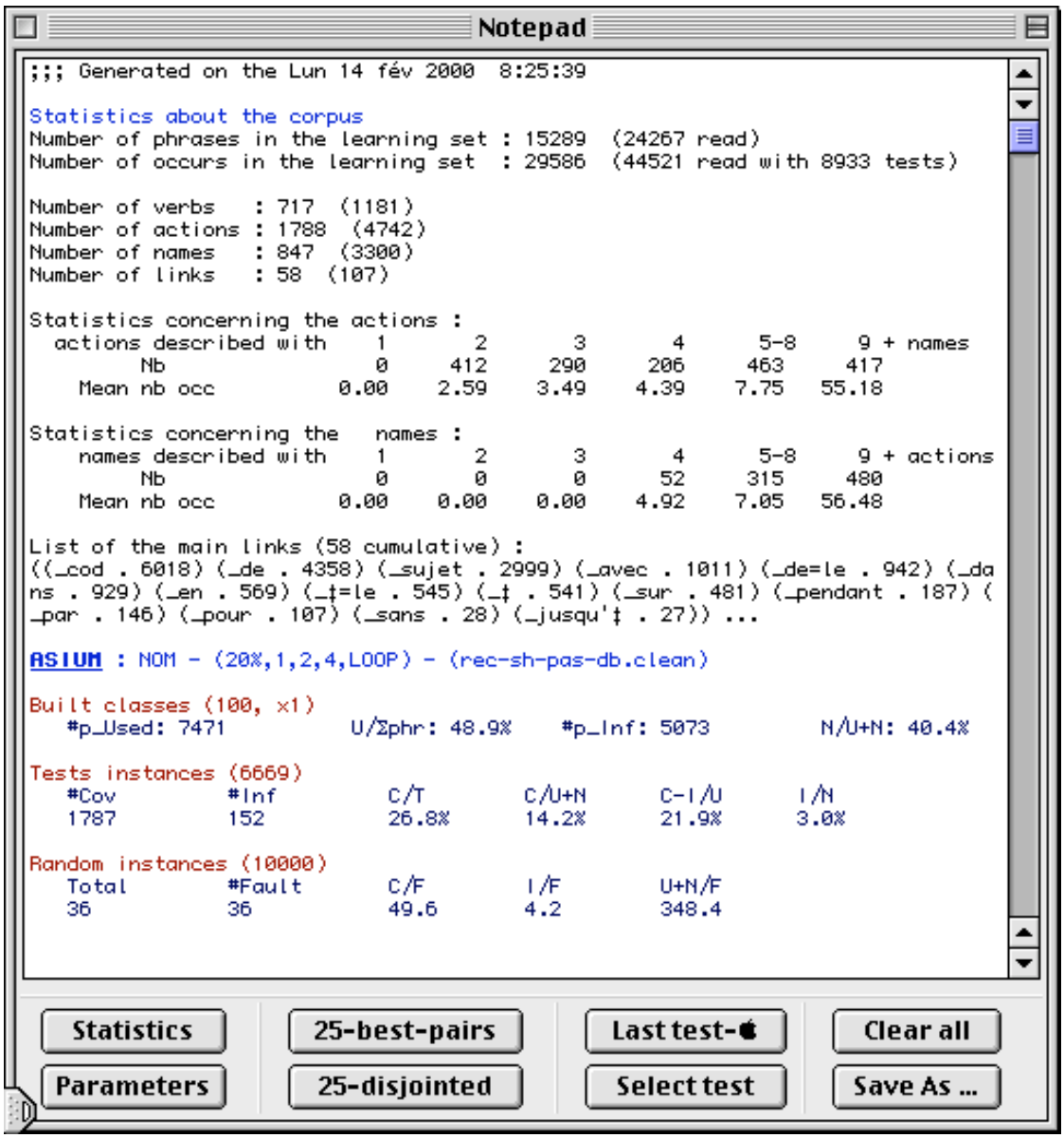
2.8. Show Notepad ...

Le Notepad (ou Calepin) offre le moyen de noter ou de visualiser des informations sur le jeu de données en cours d’analyse (c’est notamment le cas des tests) et de les conserver dans un fichier. C’est donc le moyen de garder une trace des sessions. Le Notepad se compose principalement d’une

fenêtre d'édition dans lequel l'utilisateur peut librement insérer ou retirer du texte ainsi que d'une barre de boutons placé au dessous de cette fenêtre.

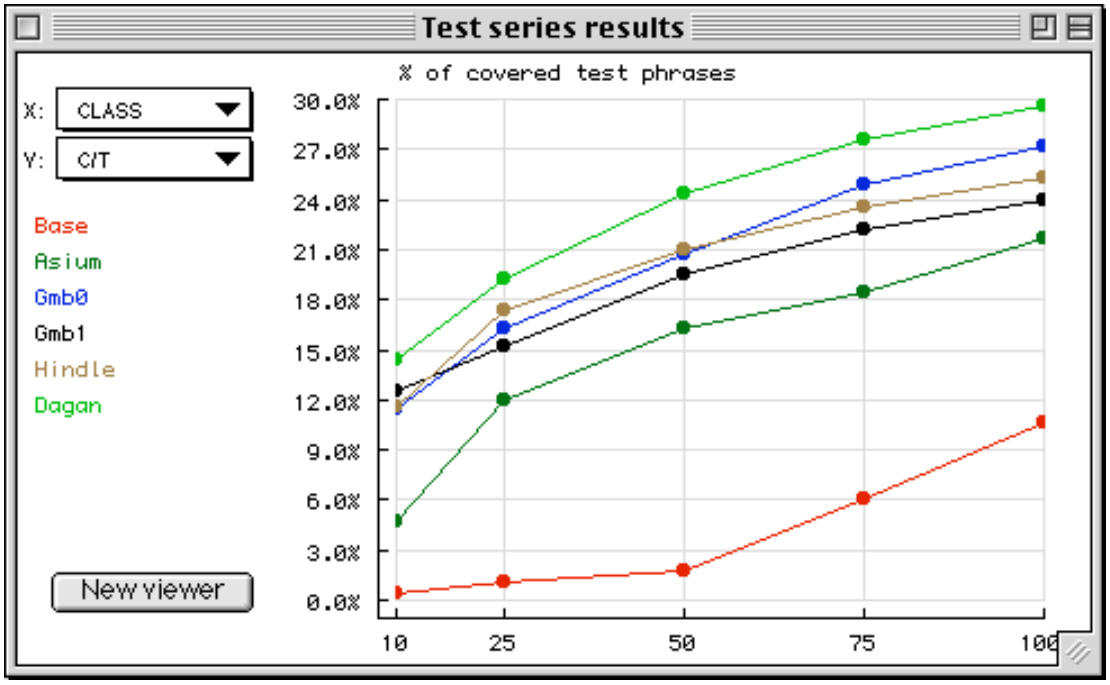
Les commande sont actuellement les suivantes (toute proposition d'amélioration est acceptée) :

- Statistics : Affiche des informations statistique sur le jeu de phrases en mémoire
- Parameters : Redonne les paramètres d'élagage sélectionnés
- 25-best-pairs : Affiche les 20 couples d'exemples les plus similaires
- 25-disjointed : Affiche les 20 couples disjoints d'exemples les plus similaires
- Last test : Affiche le dernier test (+ les phrases erronée si la touche "pomme" est appuyée)
- Select test : Affiche une BD permettant de sélectionner les test a afficher au sein d'une série
- Clear all : Efface le contenu du Notepad (en fait la zone comprise en le début et le curseur)
- Save As : Permet de sauver le contenu de la fenêtre dans un fichier texte



2.9. Show graphics ...

Cette commande permet de visualiser sous forme de courbes les résultats obtenus par les différentes mesures de similarité lorsque l'on effectue une série automatique de tests. La fenêtre contient en haut à droite deux menus déroulants permettant de choisir respectivement le type d'échelle à utiliser en abcisse et le type de résultat que l'on souhaite faire apparaitre en ordonné. Les différents résultats possibles sont décrits dans la partie du document qui concerne les commandes de test. En dessous du menu on trouve la liste des différentes mesures affichées avec leur code de couleur. Notons que si l'utilisateur fait apparaitre la fenêtre avant de lancer les jeux de tests, les courbes sont automatiquement remises à jour à chaque nouveau résultat obtenu.

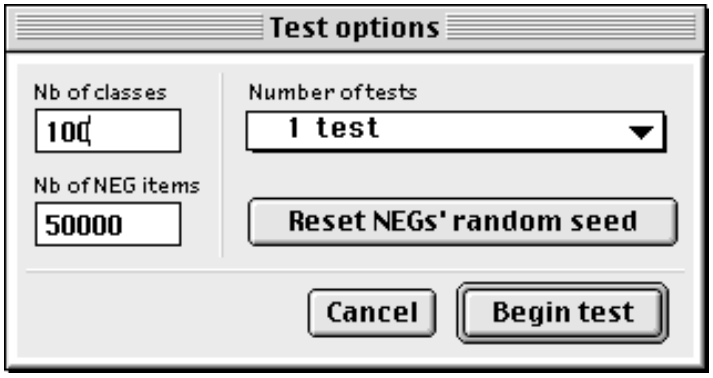


Le bouton "New viewer" qui est place en bas à gauche de la fenêtre permettre de faire apparaitre des clones de la fenêtre graphique ce qui permet de comparer aisément plusieurs types de courbes.

2.10. Test best classes ...

Cette commande a pour objet de tester la capacité prédictive et la correction des regroupements effectuées par une mesure de ressemblance donnée. L'utilisateur paramètre les tests à l'aide de la boite de dialogue ci-dessous. Il doit tout d'abord choisir le nombre de classes (disjointes) à générer, chaque classe étant constituée par un couple d'exemples (action ou nom selon le cas). Ensuite, il doit choisir le nombre d'exemples "négatifs" à engendrer, ces exemples étant construits en tirant aléatoirement une action et un nom parmi les mots qui n'ont pas été élagués. Les tests peuvent être effectués 1 ou plusieurs fois ce qui permet d'établir des moyennes significatives. Ce nombre est sélectionné à l'aide d'un menu déroulant en haut à droite.

Soulignons que, par défaut et pour une session de travail donnée, Mo'K utilise tout le temps la même "graine" pour générer l'ensemble des exemples aléatoires ; ainsi, lorsque l'on teste plusieurs mesures de ressemblances, les tests vont porter sur les **mêmes** jeux d'exemples négatifs ce qui assure l'homogénéité des résultats. Le bouton "Reset" à droite de la boite à pour but de modifier cette graine et, ce faisant, la série d'exemples négatifs qui est engendrée. Notons que lorsque l'on effectue une série de tests (commande suivante) la graine des négatifs est également modifiée.



Le test de capacités prédictives est effectué à l'aide des phrases tests qui ont été "mises de côté" lors de la phase d'élagage du corpus. Le processus est simple : pour chaque phrase "test" le système regarde si le concept qu'elle décrit (le concept étant soit une action soit un nom selon la manière dont les exemples sont constitués) est présent dans une (ou plusieurs) des classes générées. Si c'est le cas on regarde si l'attribut présent dans la phrase test est aussi présent dans la description de cette classe. Les résultats des tests sont visualisable sous la forme de plusieurs valeurs affichées dans le NotePad lorsque l'on clique sur le bouton "Last test" de ce dernier. Notons que les tests effectués par Mo'K peuvent être modifiés, cette possibilité est décrite dans la partie "Gestion des tests dans Mo'K".

Voici la liste commentée des mesures qui sont actuellement effectuées durant les tests et décrites telles qu'elles apparaissent dans le NotePad :

```
Nom méthode : type exemples - paramètres d'élagage - nom du fichier

Built classes (nombre classes, nombre de boucle de tests)
```

U=used : nombre de phrases (triplets) utilisées pour construire les classes
U/∑phr : pourcentage de phrases par rapport à l'ensemble d'apprentissage
N=inf : nombre de phrases inférées par le processus de construction des classes
N/N+U : pourcentage de phrases inférées parmi les phrases contenues dans les classes

```
Tests instances (nombre phrases test T)
```

#Cov : nombre de phrases reconnues par les classes
#Inf : nombre de phrases reconnues grâce aux phrases inférées dans les classes
C/T : pourcentage de reconnaissance
C/U+N : pourcentage de reconnaissance par rapport au total des phrases mémorisées et inférées
C-I/U : pourcentage d'utilisation des phrases inférées dans les classes
I/N : rapport entre le nombre de phrases reconnues par inférence et le nombre d'inférences

```
Random instances (nombre de phrases générées)
```

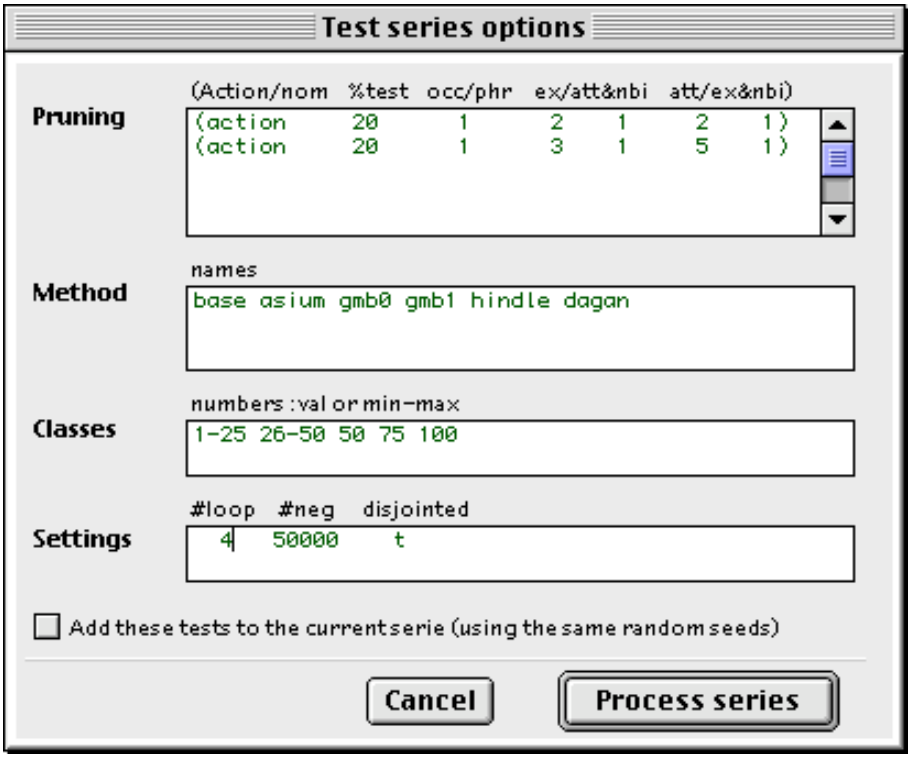
Total : nombre total d'erreurs au cours des boucles de test
#Fault : nombre moyen de phrases NEG reconnues
C/F : rapport entre la couverture et le nombre d'erreurs
I/F : rapport entre les phrases reconnues par inférence et le nombre d'erreurs
U+N/F : rapport entre le nombre de pharses mémorisée et inférées et le nombre d'erreurs

Pour tester la corrections des classes induites, on se base donc sur le nombre de phrases aléatoires (NEG) qui sont reconnues par le système. Bien évidemment certaines de ces phrases peuvent être "sémantiquement correctes" bien que non présentes dans le corpus, c'est pourquoi, dans le NotePad, en plus d'indiquer le nombre d'inférences effectuées par le système, on peut obtenir la liste de ces phrases (en appuyant sur la touche "pomme"). C'est donc à l'utilisateur de juger si elles

sont correctes ou non et de réajuster le score en fonction (via la sauvegarde des tests dans la version actuelle).

2.11. Process test series ...

Cette commande permet d'automatiser les tests de capacité prédictive des regroupements effectués par les différentes mesures de similarité. On peut ainsi faire varier automatiquement les paramètres d'élagage, de construction de classes ainsi que la liste des mesures à utiliser.



La boite de dialogue contient quatre champs permettant d'exprimer sous la forme de liste les différentes contraintes et options à utiliser pour réaliser ces tests. Les paramètres à donner sont les mêmes que ceux que l'on trouve dans les boites de dialogues spécifiques.

Ainsi, dans l'exemple ci-dessus on va effectuer une série de tests portant sur :
- 2 types d'élagage différents,
- portant sur 6 mesures de similarité,
- les tests porteront sur les classes 1 à 25, 26 à 50, ainsi que sur les 50, 75 et 100 premières,
- on effectuera 4 fois l'ensemble des tests en prenant 50000 NEGs.

C'est donc en tout 240 tests élémentaires qui seront effectués automatiquement par Mo'K. Les résultats de ces tests peuvent être affichés dans le Notepad et ils sont également visualisables sous forme de courbes grâce à la commande "Show Graphics".

On peut également indiquer au système que l'on veut ajouter le résultats des tests à ceux précédemment effectués, par exemple pour introduire de nouveaux points dans les courbes. Dans ce cas, le système réutilise automatiquement les mêmes graines aléatoires que pour les séries de tests précédents de manière à travailler sur les même exemples de tests et les mêmes NEGs (il faut cependant garder les champs PRUNING et SETTINGS absolument identiques).

Attention : dans la version actuelle l'attribut disjointed (classes disjointes ou non) du champ SETTINGS doit rester à T car les routines de tests ne sont pas adaptées aux classes non disjointes.

2.12. Save test series ...

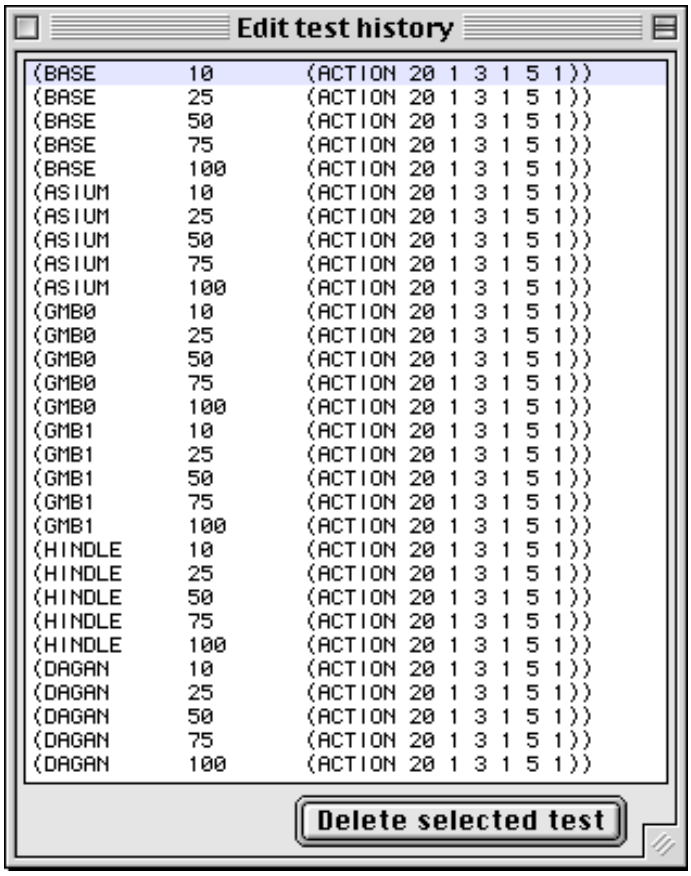
Permet de sauver les résultats issus de la dernière série de tests effectués. Une fois sauvée l'utilisateur peut éventuellement éditer ce fichier pour ajuster le nombre de phrases erronées par exemple.

2.13. Load test series ...

Permet de relire les résultats issus d'une série de tests. Pour l'instant on ne peut que les visualiser sous la forme de courbes grace à la commande "Show graphics".

2.14. Edit test series ...

Cette commande permet d'éditer la liste des test effectuer et de retirer certains résultats. On peut ainsi par exemple modifier le code d'une mesure de similarité, retirer les tests qui s'y rapporte et refaire une série de tests ne portant que sur cette mesure spécifique.



Notons que lorsque l'on clique sur un item de la liste en ayant la touche option (ALT) appuyée, on accède à la représentation interne du test ce qui permet de vérifier les mesures effectuées.

2.15. Save phrases book ...

Cette commande permet de sauver les phrases en mémoires à un instant donné, c'est à dire celles sélectionnées par le critères d'élagage courant et contenant les modifications apportées via le dictionnaire. On peut ainsi se servir du système d'élagage de Mo'K de manière à constituer un jeu de données plus petit. Le format de sauvegarde des exemples est celui décrit dans <Load phrase book>.

2.15. Save word modification ...

Cette commande permet de construire des liste de suppressions (« stop list ») ou de substitions à partir de l'ensemble des modifications qui ont été été effectuées dans le dictionnaire. On peut ainsi sélectionner le type de mots à sauver et le type de modification qui ont été réalisées. Il est possible de fusionner ces listes dans un fichier existant (check box en bas à gauche).

2.16. Save random phrases ...

Permet de générer N exemples fictifs construit par agrégation d'une action et d'un nom pris aléatoirement dans le dictionnaire. L'idée est de pouvoir générer des jeux de « contre-exemples ». Le format des exemples générés est le même que celui utilisée par <Load phrase book>.

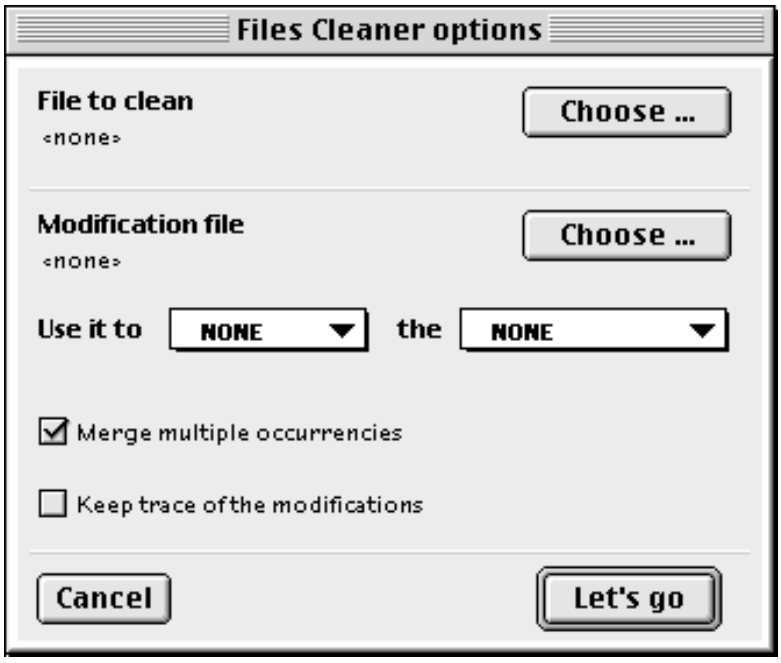
2.17. Save best pairs ...

Permet de sauver les N meilleures paires d'exemples trouvées par les méthodes courantes. Les exemples sauvés sont donc ceux qui apparaissent dans la boite d'affichage d'exemples lorsque le mode de recherche des 400 meilleurs paires est activé.

2.18. File cleaner ...

Cette commande permet de “nettoyer” un fichier de phrases à l’aide d’un fichier de substitutions de mots et de fichiers de sélections (ou suppressions) de mots. Lors de l’exécution de la commande, une boite de dialogue permet à l’utilisateur de sélectionner l’opération qu’il souhaite réaliser. On rentre ainsi successivement les informations suivantes :

- Nom du fichier à modifier
- Nom du fichier décrivant les mots à modifier
- L’opération à effectuer parmi : SELECT, DELETE, MODIFY. Les deux première permettent de sélectionner (respectivement supprimer) les triplets contenant une liste spécifique de mots.
- Les items concernés par le fichier parmi : VERBS, LINKS, WORDS, VERB&WORDS, ALL
- Le fait que l’on souhaite fusionner les triplets identiques en sommant le nombre d’occurrences
- Le fait que l’on souhaite mémoriser les modifications sous forme de commentaires



Les format des deux types fichiers de modifications sont les suivants :

1) Pour le fichier de substitutions de mots (MODIFY)

```
...
champigons                <- ancienne forme
champignon                <- nouvelle forme
hakkock
haddock
...
```

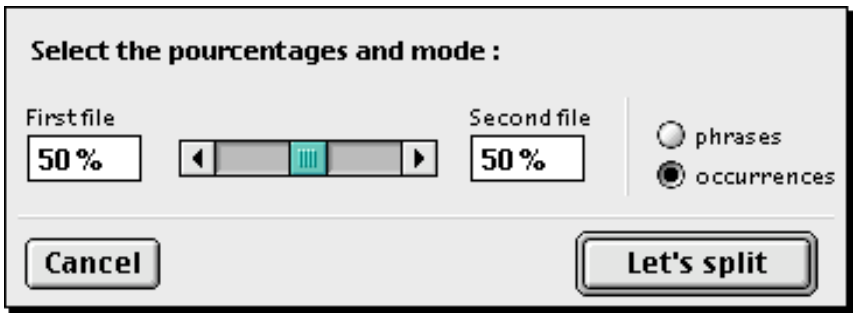
2) Pour les fichier de suppression/sélection de mots (SELECT, DELETE)

```
...
reconnait
par-contre
aux trois quarts
...
```

Après la sélection des différentes options le système construit un fichier "XXX.clean" corrigés. Si aucun nom d'opération n'est rentré (NONE) mais que l'option de fusion est active, le système se borne à fusionner les triplets communs. Notons enfin que la conservation des modifications ne concerne que le dernier fichier, il n'y a donc pas de cumul de commentaires d'un filtrage à l'autre

2.19. File splitter ...

Cette commande permet simplement de découper en deux morceaux un fichiers contenant un corpus de phrases. Les options de découpages sont les mêmes que celles proposée pour générer les tests.



3. Introduction de nouvelles méthodes dans Mo'K

3.1. Introduction

Le système Mo'K est construit de manière à pouvoir redéfinir et étendre les méthodes de mesure de ressemblance et de classification entre mots. Pour ce faire, les algorithmes de calcul sont associés à des classes dont le comportement est décrit à l'aide de méthodes (au sens "objet"). Le but de cette partie est donc d'expliquer comment introduire de nouveaux algorithmes dans le système. Il est également nécessaire de décrire les structure de données de Mo'k puisque c'est là que les méthodes génériques vont lire et écrire les informations calculées. Nous allons successivement détailler :

- Les structures de données utilisées dans Mo'K
- L'architecture du système concernant l'appel des méthodes génériques
- La description des méthodes génériques proprement dites

Important : les codes des méthodes sont (à l'exception de BASE) décrits dans le dossier "Method". L'utilisateur est libre d'y ajouter son code comme il le souhaite, l'ensemble des fichiers contenus dans ce dossier étant automatiquement chargé et évalué lors du lancement de Mo'K.

3.2. Les structures de données

3.2.1. Les variables globales

Le système gère un certain nombre de variables globales. Certaines de ces variables contiennent des informations purement numériques (nombre de phrases lues, ...), d'autres servent à indexer les structures de données contenant les informations issues du corpus. Voici les principales :

Table des variables "globales"	
`*nb-phrase-corpus*`	nombre de phrases (nuplets) lues dans le fichier initial
`*nb-phrase*`	nombre de phrases (nuplets) après élagage
`*nb-occ-corpus*`	nombre total d'occurrences lues dans le fichier initial
`*nb-occ`	nombre total d'occurrences après élagage
`*nb-example*`	nombre d'exemples générées par le système
`*word-index*`	tableau contenant les mots classés par ordre lexicographique
`*dictionary*`	hash-table (clef : mot) contenant la description de tous les MOTs
`*example-table*`	tableau contenant la description de tous les EXEMPLEs

Afin de comprendre le rôle de ces variables il faut s'intéresser aussi aux structures de données. Dans les tableaux et les hash-tables les informations sont stockées à l'aide de "structures" (ou record), qui contiennent des "slots" (ou champs). Lorsque nous faisons référence à une structure, le nom est écrit en lettres capitales. Il y a quatre types de structures : MOT, PHRASE, EXEMPLE, LITTERAL, que l'on peut séparar en deux catégories. D'une part celles permettant de stocker les mots du corpus (MOT, PHRASE), d'autre part celles permettant de gérer les informations utilisées lors de la classification (EXEMPLE, LITTERAL). Nous allons étudier maintenant la première catégorie.

3.2.2. Les structures stockant les phrases du corpus

Les structures PHRASE et MOT sont crées lors du chargement du corpus. La structure PHRASE sert simplment à stocker un nuplet issu du corpus. Voici son format :

Structure de PHRASE	
`active`	indicateur indiquant si la phrase est élaguée ou non
`verbe`	chaîne correspondant au verbe du nuplet
`lien`	chaîne correspondant au lien du nuplet
`action`	chaîne correspondant à l'action (couplet verbe_lien)
`nom`	chaîne correspondant au nom du nuplet
`occ`	entier correspondant au nombre d'occurrences du nuplet (- les tests)
`occ-test`	entier correspondant au nombre d'occurrences mise de coté pour les tests

La structure MOT permet de stocker les mots du corpus. En pratique, on crée une structure MOT pour chacune des parties de PHRASE (verbe, lien, action et nom). Voici son format :

Structure de MOT	
`valide`	indicateur indiquant si le mot a été élagué (NIL) ou non (T)
`type`	symbole parmi : verbe, lien, action, nom
`label`	chaine de caractères correspondant à son nom
`id-mot`	indice du mot dans la table `*word-index*`
`ref-exemple`	pointeur sur la structure de l'exemple éventuellment associé au mot
`poids`	poids associé au mot
`nb-phrase`	nombre de phrases dans lesquelles le mot apparaît (après élagage)
`nb-occ`	nombre d'occurrences totales du mot (après élagage)
`list-phrase`	liste de l'ensemble des PHRASE dans lesquels le mot apparaît

Afin de mieux comprendre les interactions entre ces différents éléments, voici un exemple des valeurs qui sont prises par ces structures pour un corpus de 3 phrases :

Début de la table *word-index* (en réalité triée par ordre lexicographique)

0	1	2	3	4	5
"[CHAUFFER]"	"_cod"	"CHAUFFER_co	"<casserole>"	"<four>"	"VERSER"

• • •

MOT : "[CHAUFFER_cod]"	
valide	T
type	verbe
id-mot	2
poids	-
ref-exemple	-
nb-phrase	2
nb-occ	20
list-phrase	(...)

MOT : (casserole)	
valide	T
type	verbe
id-mot	3
poids	-
ref-exemple	-
nb-phrase	2
nb-occ	13
list-phrase	(...)

On a ici que les descriptions associées aux mots : Chauffer_COD et Casserole. Cependant, les autres mots Four, Verser, COD, DANS et Verser_DANS, sont décrits de manière similaire.

Structure PHRASE	
active	T
verbe	"[CHAUFFER]"
lien	"_cod"
action	"CHAUFFER_co
nom	"<casserole>"
occ	12

Structure PHRASE	
active	T
verbe	"[CHAUFFER]"
lien	"_cod"
action	"CHAUFFER_su
nom	"<four>"
occ	8

Structure PHRASE	
active	T
verbe	"[VERSER]"
lien	"_dans"
action	"VERSER_dans"
nom	"<casserole>"
occ	5

chauffer	COD	casserole	12
chauffer	COD	four	8
verser	DANS	casserole	5

Corpus initial

Nous n'avons pas représenté ici la hash-table `*dictionary*` qui permet simplement de retrouver la structure d'un mot à partir de la chaîne de caractère qui composent son nom (c'est la "clef"). Par ailleurs, dans la structure MOT deux champs (poids et ref-exemple) ne sont pas valués, il le seront lors de la phase suivante qui consiste à construire le jeu d'exemples d'apprentissage.

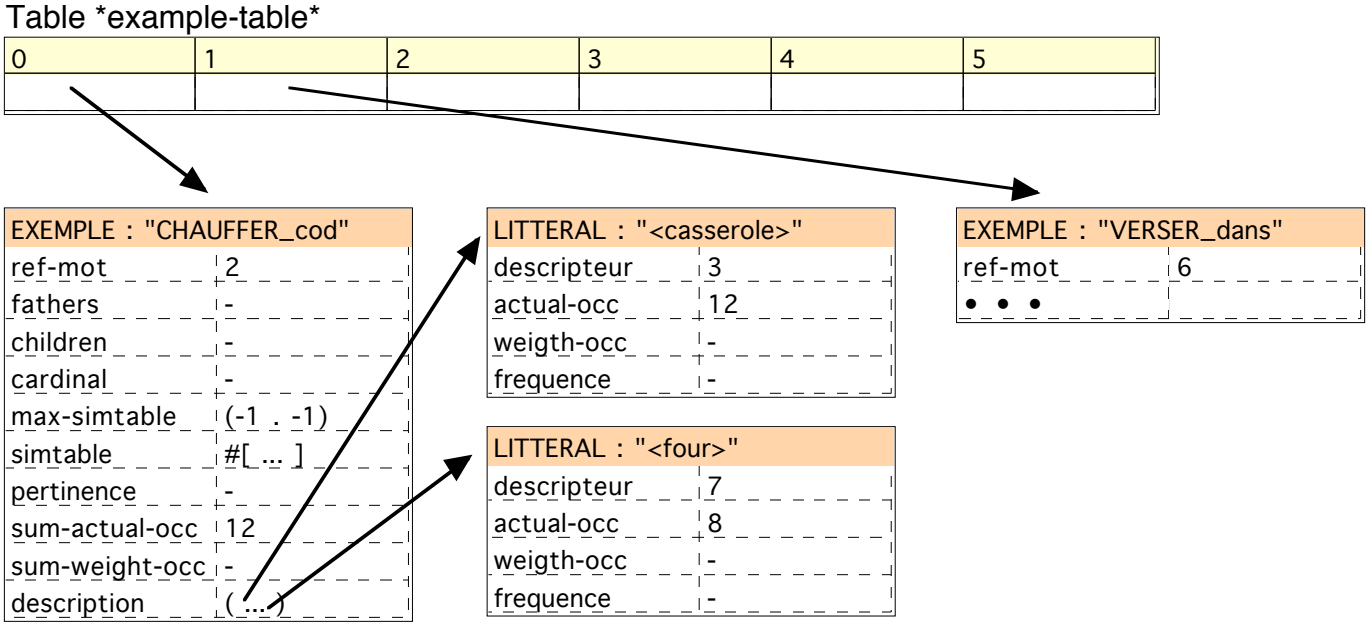
3.2.3. Les structures stockant les exemples

Après les choix du type d'apprentissage que l'on veut faire (Nom ou Action), des paramètres d'élagage et de la méthode de classification, le système construit la base d'exemples. Les exemples sont décrits à l'aide de deux structures EXEMPLE et LITTERAL qui servent également à décrire les classes produites par généralisation lors de la classification. Voici le format de EXEMPLE :

Structure de EXEMPLE	
`id-exemple`	indice de l'exemple dans la table `*example-table*`
`ref-mot`	pointeur sur la structure du mot
`fathers`	liste des N° d'exemples plus généraux (hiérarchie)
`children`	liste des N° d'exemples plus spécifiques (hiérarchie)
`cardinal`	nombre total de mots généralisés dans la classe

`max-simtable`	paire pointée formée de la similarité maximale trouvée dans la table `simtable` de l'exemple avec l'indice correspondant : (similarité . indice)
`simtable`	table de similarité entre l'exemple courant N et ceux dont l'indice est ≤N
`pertinence`	valeur de pertinence de l'exemple
`sum-actual-occ`	somme des occurrences des attributs composants le nom
`sum-weight-occ`	somme des occurrences pondérées des attributs composants le nom
`description`	liste des LITTERAL (ou attributs) composants l'exemple

Voici un exemples des structures qui sont générées dans le cas ou les exemples sont engendrés à partir des actions (on reprend ici l'exemple précédent avec comme action "CHAUFFER_cod").



Chaque exemple est "décrit" à l'aide d'une liste d'attributs. Ainsi, si l'on veut classer des Actions, les exemples correspondent aux Actions et les attributs aux Noms. La structure LITTERAL sert à stocker chacune des occurrence d'un attribut, cependant, elle sert également à ranger des informations (statistiques ou autres) qui seront utilisées pour comparer les exemples. Voici son format :

Table de LITTERAL	
`descripteur`	indice de l'attribut dans la table `*word-index*`
`actual-occ`	nombre d'occurrence de l'attribut avec l'exemple
`weigth-occ`	pondération locale de l'attribut
`frequence`	fréquence relative de l'attribut (valeur utilisée pour comparer les attributs)

Notons qu'afin d'optimiser le temps de calcul des similarités, on code les attributs sous la forme d'entiers (en pratique il s'agit de leur indice dans `*word-index*`). De cette manière la recherche des attributs communs se résume à des comparaisons d'entiers sans allocation de mémoire par MCL.

Chaque fois d'un exemple est crée, on place dans le champ `ref-exemple` du MOT qui lui correspond son numéro dans la table des exemples. Les champs Father, Children et Cardinal sont utilisés durant la classification pour gérer la hiérarchie. Enfin, les champs Simtable et Max-simtable servent à stocker les similarité entre exemples ; nous ne les détaillons pas ici car on y accède jamais directement.

Comme on peut le constater, après la construction des exemples de nombreux champs n'ont pas encore de valeur, ils sont remplis par l'appel aux méthodes associées à l'algorithme de comparaison. Les liens existant entre méthodes et structures sont décrits dans le paragraphe suivant.

3.3. Enchainement des méthodes

Actuellement, lors de la phase de la construction des exemples (commande <Build examples>), voici l'ordre d'appel des méthodes génériques (cf 3.2) et les résultats qu'elles engendrent :

Etape du processus de comparaison	Résultat placé dans la structure
❶ Phase d'initialisation	
Pour chaque MOT de la base	
Appel de *init-att-weight*	-> MOT.poids
Appel de *init-obj-weight*	-> MOT.poids
❷ Phase de construction des exemples	
Pour chaque EXEMPLE	
Pour chaque attribut (c-à-d LITTERAL)	
Appel de *eval-att-weight*	-> EXEMPLE.LITTERAL.weigth-occ
Appel de *eval-obj-weight*	-> EXEMPLE.sum-weigth-occ
Pour chaque LITTERAL	
Appel de *init-similarity*	-> EXEMPLE.LITTERAL.frequence
❸ Phase de calcul des similarités	
Pour chaque couple d'EXEMPLE (e1, e2)	
Appel de *eval-similarity*	-> EXEMPLE-e1.simtable (e2)
❹ Phase finale	
Pour chaque EXEMPLE	
Appel de *eval-score*	-> EXEMPLE.pertinence

3.4. Implémentation des méthodes génériques

Nous allons maintenant détailler l'implémentation des méthodes génériques, c'est à dire préciser le comportement attendu et les entrées/sorties. Lors de cette description, le passage de paramètres dans les fonctions génériques utilise les conventions suivantes :

- SELF : désigne l'instance de la classe de la méthode utilisée
- OBJ : structure du MOT jouant le rôle d'exemple
- ATT : structure du MOT jouant le rôle d'attribut
- LITT : structure d'un LITTERAL dans une description d'exemple
- EX : structure de l'EXEMPLE
- PHR : structure d'une PHRASE (ou triplet)

Pour illustrer la manière dont est implémenté un algorithme nous détaillons ici l'implémentation de la classe BASE qui est présente dans Mo'K. Contrairement aux autres méthodes celle-ci est implémentée non pas dans le dossier "`Method`" mais dans le fichier "`Code Mo'K/method.lisp`".

3.4.1. Déclaration d'un nouvel algorithme

La déclaration d'une classe est la première chose à faire pour décrire un nouvel algorithme. Cet algorithme peut hériter soit de la classe MOK-SIMILARITY-METHOD générale, soit d'une classe déjà existante dans le système (par exemple BASE qui est prédéfinie). Dans tous les cas on hérite de tous les comportements de l'algorithme précédent et on décrit les méthodes qui sont différentes.

```
(defclass my-method (MOK-SIMILARITY-METHOD) ())
```

On peut documenter la méthode à l'aide de l'option `:documentation` de Common-Lisp. Dans ce cas c'est cette chaîne qui sera utilisée dans le menu de sélection d'algorithme en lieu et place du nom de la classe implémentant la méthode. Dans le cas de BASE l'appel est le suivant :

```
(defclass BASE (MOK-SIMILARITY-METHOD) ()
  (:documentation "Méthode de base"))
```

3.4.2. Méthode de documentation de l'algorithme

La méthode de description renvoie le message, sous forme de listes de chaînes, qui est affiché dans la boite de dialogue de construction d'exemples lorsque l'utilisateur sélectionne un algorithme. Notons que l'on n'utilise pas ici (cf ci-dessus) la possibilité de documentation offerte par Common-Lisp qui reste trop limitée et ne permet pas facilement de composer des messages multilignes.

describe-method	
Définition	`(defmethod describe-method ((self my-method)) ...)`
Résultat	une liste de chaînes de caractères décrivant l'algorithme
Stockage	-

```
(defmethod describe-method ((self BASE))
   '("Méthode de calcul élémentaire ..."
     "La pondération des attributs est 1, et les fréquences des attributs"
     "expriment leurs occurrences relatives par rapport aux autres attributs."
     "La similarité est basée sur le minimum des fréquences communes."
     "La pertinence d'un exemple est : # d'occurrences / Sqrt (# attributs)"))
```

Cette méthode est cependant purement informative et rien n'interdit de faire :

```
(defmethod describe-method ((self BASE)) nil)
```

3.4.3. Méthodes de pondération des attributs

Ces deux méthodes permettent d'initialiser et de calculer le poids des attributs.

init-att-weight	
Définition	`(defmethod init-att-weight ((self my-method) att) ... )`
Résultat	une valeur numérique
Stockage	`-> att.poids`

Dans le cas de l'algorithme BASE le poids des attributs vaut toujours 1. Le DECLARE sert uniquement à éviter les hurlements du compilateur disant que ATT n'est pas utilisé.

```
(defmethod init-att-weight ((self BASE) att) (declare (ignore att)) 1)
```

eval-att-weight	
Définition	`(defmethod eval-att-weight ((self my-method) att obj phr) ... )`
Résultat	une valeur numérique
Stockage	`-> dans le champ Weigth-Occ du littéral correspondant à Att`

L'argument PHR correspond à la phrase courante dans laquelle ATT et OBJ "co-occurrent".

Par exemple, dans BASE on renvoie pour chaque attribut le nombre d'occurences de cet multiplié par son poids (initialisé avec `init-att-weight`). Notons qu'ici ce produit n'est pas très utile car le poids est toujours de 1, on fait juste cela pour la rendre plus facilement héritable ...

```
(defmethod eval-att-weight ((self BASE) att obj phrase)
   (declare (ignore obj)) (* (get-poids att) (get-occ phrase)))
```

3.4.4. Méthodes de pondération des objets (et exemples)

Ces deux méthodes permettent d'initialiser et de calculer le poids des exemples. En toute rigueur la seconde méthode devrait plutôt s'appeler <`init-ex-weight`> puisqu'elle travaille sur des exemples.

init-obj-weight	
Définition	`(defmethod init-obj-weight ((self my-method) obj) ... )`
Résultat	une valeur numérique
Stockage	`-> obj.poids`

Dans BASE , le système affecte un poids de 1 à chaque objet :

```
(defmethod init-obj-weight ((self BASE) obj) (declare (ignore obj)) 1)
```

eval-obj-weight	
Définition	`(defmethod eval-obj-weight ((self my-method) ex) ... )`
Résultat	une valeur numérique
Stockage	`-> ex.sum-weigth-occ`

Notons qu'en interne, le système effectue automatiquement la somme des poids de chaque attribut (c-à-d des valeurs renvoyées par <`init-obj-weight`>) et place le résultat dans le champ <`sum-weight-occ`>. De ce fait, dans BASE on n'a que renvoyer la valeur précalculée :

```
(defmethod eval-obj-weight ((self BASE) ex) (get-sum-weight-occ ex))
```

Cette méthode est en principe largement héritable mais il peut être nécessaire de la spécialiser si l'on a par exemple des poids négatifs à traiter (Hindle) ou si le calcul fait intervenir une fonction modificatrice du résultat (Log ou autre). On peut alors redéfinir complètement la boucle de calcul en itérant sur les littéraux décrivant l'exemple, ainsi :

```
(defmethod eval-obj-weight ((self BASE) ex)
   (let ((som 0))
      (dolist (litt (get-description ex)) (incf som (my-function litt))) som))
```

3.4.5. Méthode d'évaluation de la pertinence d'un exemple (ou classe)

Cette méthode vise à évaluer le degré de pertinence d'un exemple. La pertinence a été introduite dans le système pour effectuer l'élagage automatique de la hiérarchie de classe. Dans la méthode GMB1 elle est cependant utilisé pour pondérer la mesure de ressemblance et amener le système à ne comparer que des exemples ayant des contenus informationnels quantitativement proches.

eval-score	
Définition	`(defmethod eval-score ((self my-method) ex) ... )`
Résultat	une valeur numérique
Stockage	`-> ex.pertinence`

Dans l'algorithme BASE, on calcule le rapport entre le poids (pondéré) de l'exemple et la racine carré du nombre de descripteurs.

```
(defmethod eval-score ((self BASE) ex)
  (/ (get-sum-weight-occ ex) (sqrt (length (get-description ex)))))
```

3.4.6. Méthodes d'évaluation de la similarité entre deux exemples

La première méthode est utilisée pour initialiser le champ `frequence` de chaque littéral. C'est cette valeur qui est "a priori" utilisée pour comparer les attributs communs lors du calcul de similarité. Le nom "fréquence" fait référence au fait que l'on compare "a priori" des profils de fréquences, cependant chaque algorithme est libre de placer dans ce champ l'information qui lui est propre.

init-similarity	
Définition	`(defmethod init-similarity ((self my-method) ex lit) ... )`
Résultat	une valeur numérique
Stockage	`-> lit.frequence`

Dans le cas de BASE, on renvoie le rapport entre la valeur pondérée du littéral et la somme pondérée. Le résultat est multiplié par la constante `*freq-accuracy*` de manière à se ramener à un entier et optimiser l'évaluation ultérieure de la similarité. Nous reviendrons plus en détail sur ces considérations dans la partie 3.5 de ce document

```
(defmethod init-similarity ((self BASE) ex lit)
   (round (* (/ (get-weight-occ lit) (get-sum-weight-occ ex)) *freq-accuracy*)))
```

La méthode `eval-similarity` effectue le calcul effectif de la similarité.

eval-similarity	
Définition	`(defmethod eval-similarity ((self my-method) ex1 ex2) ... )`
Résultat	une valeur numérique
Stockage	`-> ex1.simtable (ex2) ou ex2.simtable (ex1)`

En pratique le "lieu de stockage" de la similarité dépend de l'indice des exemples, de toute manière l'appel à la fonction <`get-sim-value (ex1 ex2)`> permet de retrouver la valeur de similarité en s'affranchissant de ce problème. Dans le cas de BASE on calcule la somme du minimum des fréquences entre littéraux communs. Notons que dans le système la liste des attributs est automatiquement triée par numéros d'indices croissants, de la sorte le calcul de l'intersection des listes peut s'effectuer de manière optimale (complexité linéaire).

La méthode peut paraître complexe à écrire mais en fait toutes les mesure de ressemblance basées sur la comparaison des attributs communs possèdent la même structure : une boucle et une partie mise à jour des résultats (similarités) marquée en bleu ci-dessus. Il serait donc envisageable de simplifier l'écriture en mettant dans le système la partie générique et en implémentant les parties variables sous la forme de MACRO), mais la version actuelle présente l'avantage d'offrir le maximum de souplesse.

```
(defmethod eval-similarity ((self BASE) ex1 ex2)
   (let ((result 0) desc1 desc2)
      (setq ex1 (get-description ex1))     ; liste triée des descripteur de ex1
      (setq ex2 (get-description ex2))     ; idem pour ex2
      (ccl::while (and ex1 ex2)
         (if (= (setq desc1 (get-descripteur (car ex1)))      ; si identique
                (setq desc2 (get-descripteur (car ex2))))
            (incf result (min (get-frequence (pop ex1)) (get-frequence (pop ex2))))
          (if (< desc1 desc2) (pop ex1) (pop ex2))))          ; si différents
      result))
```

La méthode `explain-similarity` est utilisée par l'interface (boite "Show examples" et "Notepad") pour afficher les parties communes entre deux exemples et les valeurs de ressemblance entre chaque littéral. Elle se borne à construire une *trace* du calcul de similarité.

explain-similarity	
Définition	`(defmethod explain-similarity ((self my-method) ex1 ex2) ... )`
Résultat	une liste formé de : ((valeur-commune . "attribut") ...)
Stockage	`-> ex1.simtable (ex2) ou ex2.simtable (ex1)`

La fonction est sensée construire une liste ayant la structure suivante : ((valeur . "mot") (...) ...).

```
(defmethod explain-similarity ((self BASE) ex1 ex2)
   (let (result desc1 desc2)
      (setq ex1 (get-description ex1))
      (setq ex2 (get-description ex2))
      (ccl::while (and ex1 ex2)
         (if (= (setq desc1 (get-descripteur (car ex1)))
                (setq desc2 (get-descripteur (car ex2))))
            ; construction de l'étiquette ... attention de bien respecter le format
          (progn(push  (cons  (min (get-frequence  (car  ex1))
                                   (get-frequence  (car  ex2)))
                              (aref  *word-index*  (get-ref-mot  (car  ex1))))
                       result)
                 (pop  ex1)  (pop  ex2))
          (if (< desc1 desc2) (pop ex1) (pop ex2))))
      (sort  result  #'>  :key  #'car)))  ; on tri  les  éléments  par  valeur  décroissante
```

Notons que dans la boite de dialogue d'exploration des exemples l'utilisateur peut sélectionner le champ de LITTERAL qu'il souhaite faire apparaitre dans la description des exemples. Pour profiter de cette possibilité de l'interface il suffit de faire appel à la fonction <`get-selected-slot`> en lieu et place de la fonction <`get-frequence`> dans la méthode ci-dessus.

3.4.7. Méthode d'affichage des valeurs

Ces méthodes sont utilisées pour formater les résultats calculés par le système. Ce formattage est rendu nécessaire par le fait que les valeurs peuvent avoir des ordre de grandeurs très différentes (entier, pourcentage, ...). On a actuellement des méthodes concernant l'ensemble des valeurs (ou slots) affichables dans l'interface de Mo'K.

Format-XXX	
Définition	`(defmethod format-XXX ((self my-method) value) ... )`
Résultat	une chaîne de caractères formatée représentant VALUE
Stockage	-

Elle sont définies de la manière suivante dans BASE :

```
(defmethod format-frequency ((self BASE) value) (form% value))
(defmethod format-similarity ((self BASE) value) (form% value))
(defmethod format-weight ((self BASE) value) (formf value))
(defmethod format-score ((self BASE) value) (formf value))
(defmethod format-weight-occ ((self BASE) value) (formf value))
(defmethod format-actual-occ ((self BASE) value) (formi value))
```

4. Gestion des tests dans Mo'K

Les tests de couverture de classes de Mo'K sont décrits dans le fichier "Test.lisp" du dossier "Code Mo'k". Il est possible de rajouter de nouveaux tests qui seront automatiquement pris en compte dans le menu associé à la fenêtre de visualisation des courbes. Dans la version actuelle du système les tests imprimés dans le NodePad ne sont pas eux facilement modifiables.
Lorsqu'il fait un test de couverture pour une mesure de similarité et un jeu de test donné, le système remplie une structure contenant les champs suivants :

```
(defstruct (test-result (:conc-name get-))
  (file      ); nom du fichier de test
  (pruning   ); paramètre d'élagage
  (method    ); nom de la mesure de similarité
  (nb-loop   ); nombre de boucle effectuée avec cet élagage et cette méthode
  (disjoined ); classe disjointes ou non
  (nb-class  ); nombre de classe testé
  (nb-neg    ); nombre de NEG
  (nb-tested    :type list)    ; nombre de triplets testés
  (nb-covered   :type list)    ; nombre de triplets reconnus
  (nb-infered   :type list)    ; nombre de triplets reconnus grâce aux inférences
  (occ-tested   :type list)    ; nombre d'occurrences testées
  (occ-covered  :type list)    ; nombre d'occurrences reconnues
  (occ-infered  :type list)    ; nombre d'occ. reconnues grâce aux inférences
  (phr-used    :type list)    ; nombre de triplets utilisés pour apprendre
  (phr-inf-done  :type list)    ; nombre de triplets inférés par apprentissage
  (occ-used      :type list)    ; nombre d'occurrences utilisées pour apprendre
  (occ-inf-done  :type list)    ; nombre d'occurrences inférés par apprentissage
  (nb-fault      :type list)    ; nombre d'exemples aléatoires classés
  (list-error    :type list))   ; listes des NEG reconnus
```

Dès lors à partir de ces valeurs il est possible de définir plusieurs tests qui les combinent. Par exemple, la mesure de couverture C/T (% de phrases reconnues) est codée de la mnière suivante :

```
(defgeneric C/T (self) (:documentation "% of covered test phrases"))

(defmethod C/T ((result test-result))
   (append (eval-mean-ratio 'nb-covered 'nb-tested result) '(percent)))
```

Il est important de noter que les champs de la structure qui contiennent un résultat de test sont en fait des listes de valeurs, chaque valeur correspondant au résultat produit par l'un des boucles. Il est donc nécessaire de calculer la moyenne de ces valeurs avant d'effectuer un calcul. On dispose dans le système des fonction suivantes :

```
(mean  nom-champ  test-result)
  Renvoie la valeur moyenne dans le champ
(eval-mean  nom-champ  test-result)
  Renvoie la liste : (valeur-moyenne   écart-type)
(eval-mean-ratio  nom-champ1  nom-champ2  test-result)
  Renvoie la liste : (moyenne-champ1/moyenne-champ2   0)
```

La définition d'un test s'effectue à l'aide de deux fonctions (cf C/T ci-dessus). Le DefGeneric permet d'associer une chaine de documentation au test, cette chaine ets celle qui est affichée au dessus des courbes dans la fenêtre graphique. Le DefMethod définie la fonction de calcul, elle doit toujours renvoyer une liste composée de trois éléments :

- valeur : la valeur calculée par le test
- écart : l'écart type éventuellement associé à cette valeur
- type : le type du résultat parmi les mot-clés (INTEGER FLOAT PERCENT)

5. Utilisation avancée

Nous allons examiner maintenant plusieurs points particuliers concernant l'écriture des méthodes, notamment en ce qui concerne l'optimisation des calculs et la personnalisation des comportements du système. Mais, au préalable, il faut décrire plus en détail la structure de la classe MOK-SIMILARITY-METHOD qui est à la racine des algorithmes (c'est la super-classe de BASE). En pratique, cette classe contient un certain nombre de champs qui définissent des propriétés générales des algorithmes et qui sont hérités par les sous-classes. Voici sa définition :

```
(defclass  MOK-SIMILARITY-METHOD  (standard-method)
   ((similarity-type  :accessor  similarity-type  :initform  'sym)   ;  non  utilisé
     (sim-value-type   :accessor  sim-value-type   :initform  '(float  32))
    (offset            :accessor  offset            :initform  0)      ;  usage  interne
    (delta             :accessor  delta             :initform  0)      ;  usage  interne
     (freq-accuracy    :accessor  freq-accuracy    :initform  30000)  ;  encodage  entier
     (example-struct   :accessor  example-struct   :initform  #'make-exemple)
     (litteral-struct  :accessor  litteral-struct  :initform  #'make-litteral))
    (parameter-def     :accessor  parameter-def     :initform  nil)
     (parameter-value  :accessor  parameter-value  :initform  nil)))
```

Lorsque l'utilisateur veut modifier des valeurs de ces champs il peut (et doit) le faire par l'intermédiaire de la méthode `initialize-method` qui est systématiquement appelée après que l'utilisateur ait choisi un algorithme de comparaison, sa définition est la suivante :

Initialize-method	
Définition	`(defmethod  initialize-method ((self  my-method))  ...  )`
Résultat	-
Stockage	-

Dans le cas de BASE cette appel ne fait rien, mais dans la suite de ce chapitre nous allons voir des utilisations de cette méthode et le rôle des champs de MOK-SIMILARITY-METHOD.

5.1. Gestion optimale de la mémoire

Il est crucial de tenir compte du problème du temps de calcul et de l'utilisation de la mémoire dès lors que l'on veut implémenter des algorithmes travaillant sur de nombreux exemples (> 1000). Prenons ainsi le cas d'un calcul de similarité portant sur 4000 exemples. La méthode `eval-similarity` (la plus critique) sera appelée 8 millions de fois si la similarité est symétrique (16/2) et conduira au stockage de 8 millions de valeurs. Dans le cadre du langage utilisé, MCL version 4.2, on doit signaler les faits suivants (la version 4.3 semble permettre certaines optimisations).

Dans MCL, l'arithmétique entière est beaucoup plus efficace que la flottante, on peut réduire significativement les temps de calcul (2 à 5 fois) et l'occupation mémoire (les opérations flottantes "Cons" en moyenne 16 octets par opérations) en travaillant sur des entiers. Dans l'exemple de la méthode BASE c'est d'ailleurs ce que l'on fait en multipliant systématiquement les résultats par la constante `freq-accuracy`. Par ailleurs, du point de vue du stockage le problème est le même puisque les valeurs entières occupent moins de place. Nous allons détailler ce point.

Dans Mo'K les valeurs de similarité sont rangées dans des tables génériques dont chaque cellule occupe 4 octets. Dès lors, dans l'exemple précédent, on utilise au moins 32 Mo pour stocker les résultats. Si la valeur à ranger est un entier, cette valeur de 32Mo est celle effectivement utilisée car le stockage est "immédiat" (c'est à dire local à la table). Par contre, si l'on travaille avec une valeur

flottante, le format par défaut est le type LONG-FLOAT qui est codé sur 8 octets. Comme cette valeur n'est pas directement rangeable dans la table on a donc un surcoût de 64Mo à gérer ce qui porte l'occupation mémoire totale à 96 Mo ! On peut limiter cette taille en travaillant avec des SHORT-FLOAT qui, s'ils sont eux aussi pas directement stockables dans la table pour des raisons de "tags", n'amènent un surcout que de 32Mo seulement, soit 64Mo d'occupation totale.

Afin de contourner ce problème de stockage, on offre la possibilité de contrôler la manière dont les similarité sont codées dans le système par le biais de la valeur mise dans le champs `sim-value-type` associé aux classes de méthodes. Voici la liste des valeurs possibles :

Valeur	Codage des similarité
`T`	format générique mais peu efficace
`(float 32/64)`	flottants codés sur 32 ou 64 bits
`(integer 8/16/32)`	entiers codés sur 8, 16 ou 32 bits
`(integer 8/16/32 (min . max))`	idem avec mise à l'échelle automatique dans [min .. max]

Le dernier mode de travail est **extrêmement intéressant** puisqu'il permet de ramener toutes valeurs comprises dans l'intervalle [min .. max] sous la forme d'un entier (non signé) codé sur N bits. De la sorte, si l'on souhaite travailler sur une très grosses matrice en ayant une similarité comprise entre 0 et 2 il suffira de placer la valeur `(integer 8 (0 . 2))` dans le champ `sim-value-type` :

```
(defmethod initialize-method ((self my-method))
  (setf (sim-value-type self) '(integer 8 (0 . 2)))
```

La gestion de cette compression est, à la perte de précision prêt, totalement transparente pour l'utilisateur mais nécessite tout de même de connaitre exactement l'intervalle des valeurs prise par la mesure de similarité. Notons que la compression entraine un très léger ralentissement des calculs qui peut être limité en prenant 0 comme borne min de l'intervalle des valeurs.

ATTENTION : lorsque l'on stocke une valeur de similarité avec la directive `(integer 8/16/32 (min . max))`, la valeur de similarité retournée par la mesure **doit -être un nombre flottant**. Un non respect de cette contrainte conduit à des calculs **faux** et/ou des **plantages** systèmes.

5.2. Arithmétique rapide

Bien que l'arithmétique entière soit bien plus performante que la flottante il est parfois difficile de s'en passer par exemple, lorsqu'on veut calculer des rapports ou des logarithmes. En pratique, le principal problème de l'arithmétique flottante n'est pas tant sa vitesse de calcul que le fait qu'elle entraine énormément d'allocations de mémoire. Sans rentrer dans les détails, disons que chaque opération utilise environ 8 octets qui sont à récupèrer ensuite par le Garbage Collector.

Pour limiter ce problème, le fichier "Fast-Math.lisp" du dossier "Grapher" contient un ensemble de fonctions mathématiques qui travaillent directement par effet de bord et qui n'entrainent donc aucune allocation mémoire. Toutefois la plupart d'entre elles ne font pas ou très peu de contrôle des arguments. Dès lors, si l'utilisation de ces fonctions accélère grandement les calculs de similarité, il faut être conscient du fait que toute erreur de programmation peut entrainer au mieux (?) des résultats complètement **faux**, au pire, des **corruptions** de la mémoire et des **plantages** du système.

Lors des calculs, les variables dans lesquels sont stockées les valeurs flottantes sont classiquement déclarées dans un LET, mais elles doivent **impérativement** être initialisées à l'aide de la fonction FREG qui leur affecte un "registre flottant" (entre 0 et 9). Voici un exemple de déclaration

```
(let ( (result (freg 0 0.0))   ; déclare RESULT et l'initialise à 0.0
       (aux1 (freg 1 25.0 ))   ; déclare AUX1 et l'initialise à 25.0
       (aux2 (freg 2)) )       ; déclare AUX2 (valeur indéfinie)
     ...)
```

Les fonctions d'affectations sont les suivantes :

```
(freg number &optional init)
  Renvoie un pointeur sur le registre number (0..9) et initialise ce registre
  la valeur d'initialisation doit être un flottant

(copy-0.0)
  Renvoie une copie de la valeur 0.0 qui est une constante du système et donc à
  ce titre qu'il est très malvenu de modifier.

(setf-0.0 fvar)
  Affecte la valeur 0.0 à un registre flottant sans CONSer
```

Les fonctions mathématiques actuellement disponibles dans la bibliothèque sont les suivantes. Il est à noter que l'ensemble des opérations fonctionnent ainsi :

- L'appel (fn op1 op2 ... opn result) affecte le résultat de (fn op1 ... opn) dans RESULT
- Les paramètres attendues doivent (sauf exception indiquée) **tous** être des flottants.

```
(%fadd op1 op2 ... result) ; addition n-aire (n >=2)
(%fsub op1 op2 ... result) ; soustraction n-aire ou changement de signe
                             si n=1 (et avec effet de bord si n=0)
(%fmul op1 op2 ... result) ; multiplication n-aire (n >=2)
(%fdiv op1 op2 ... result) ; division n-aire (n >=2)

(%fast-log value result &optional base)
  Cette fonction calcule le Log de VALUE dans la BASE indiquée (défaut 2). La
  valeur peut être un flottant ou un entier. Dans ce dernier cas le calcul est
  effectué à l'aide d'une table de logarithme précalculée. Le résultat est
  par contre toujours un flottant.

(%fast-xlogx value result &optional base 0)
  Cette fonction calcule X.log(X), le fonctionnement est le même que ci-dessus

(%fabs value &optional result)
  Prend la valeur absolue de VALUE (avec effet de bord si RESULT est absent)

(%fsqrt value &optional result)
  Prend la racine carrée de VALUE (avec effet de bord si RESULT est absent)
```

5.3. Ajout de slots dans une classe d'algorithme

Lorsque l'on déclare une classe d'algorithme on peut ajouter, si besoin, des slots à la classe pouvant être utilisés comme des "variables globales" permettant de mémoriser, en cours des calculs, des informations spécifiques à la méthode. Dans l'exemple ci-dessous l'accès à ce slot est effectué dans chaque méthode à l'aide de l'appel : `(schtroumpf SELF)`. Le changement de méthode réinitialise automatiquement ces variables.

```
(defclass autre-method (MOK-SIMILARITY-METHOD)
  ((buffer-schtroumpf :accessor schtroumpf :initform 0)))
```

5.4. Ajout de slots dans LITTERAL et EXEMPLE

Les structures de données LITTERAL et EXEMPLE possède un nombre fixe de champs. Or dans certains algorithme il peut être souhaitable de pouvoir **ajouter** dynamiquement de nouveaux champs. C'est tout à fait possible dans la mesure où les opérateurs de création des structures ne sont pas directement appelés dans le système mais sont en fait stockés dans deux champs de la classe MOK-SIMILARITY-METHOD :

- `litteral-struct : champ contenant le constructeur de LITTERAL`
- `example-struct : champ contenant le constructeur de EXEMPLE`

Admettons que, dans l'algorithme MY-ALGO nous voulions rajouter un nouveau champ RANG dans la structure littéral. Les déclarations à effectuer sont les suivantes :

```
; construction normale de la classe
(defclass my-algo (BASE)
   (:documentation "On rajoute un champ RANG"))

; déclaration de la nouvelle structure pour litteral
(defstruct (my-litteral
              (:include litteral) ; on doit partir de la définition de LITTERAL
              (:conc-name get-))  ; nom de la fonction d'accès
  (rang  0))                      ; on ajoute RANG initialisé à 0

; On indique au système le nom constructeur appelé make-xxx par défaut
(defmethod initialize-method ((self my-algo))
   (setf (litteral-struct self) #'make-my-litteral))
```

Ensuite, la gestion de ce nouveau champ RANG est bien évidemment laissée à la charge de l'utilisateur l'accès ce fait par : `(get-rang)`. Attention il est important de rajouter des champs aux structures et non de les redéfinir, afin que MO'K continu à accèder normalement au champ standard.

Notons que l'ajout de nouveaux champs est automatiquement répercuté dans la fenêtre d'exploration des exemples et il est ainsi possible de visualiser leur contenu comme celui des autres. Par contre, il faut alors écrire une nouvelle méthode d'impression de la valeur qui devra s'appellera `format-xxx`, dans le cas de notre exemple précédent, on pourra par exemple écrire :

```
(defmethod format-rang ((self my-algo) value) (format nil "~5D" value))
```

5.5. Méthodes paramétrables

Certaines méthodes de comparaisons utilisent des paramètres de configuration. Afin de pouvoir saisir et mémoriser de tels paramètres la classe MOK-SIMILARITY-METHOD contient les 2 champs suivants :

- `parameter-def : champ contenant la description de la boite de dialogue`
- `parameter-value : champ contenant les valeurs sélectionnées`

Le champ `parameter-def` est formé d'un couple (`"titre de la boite" fonction-d'analyse`), la "fonction d'analyse" étant la fonction LISP (unaire) qui sera appliquée sur la chaîne de caractères renvoyée par la boite de dialogue. C'est le résultat de cette application qui est ensuite rangée dans le champ `parameter-value`. Voici un exemple de définition de boite de paramétrage :

```
; construction normale de la classe
(defclass my-algo (BASE)
   (:documentation "On ajoute une boite de paramétrage"))
```

```
; On indique au système le nom constructeur appelé make-xxx par défaut
(defmethod initialize-method ((self my-algo)
    (setf (parameter-def self) '("Coefficients a et b ?" parse-from-string))
    (setf (parameter-value self) '(3.14 6)))
```

Les valeurs rentrées dans le champ correspondent aux valeurs par défaut proposées à l'utilisateur, si cette valeur est une liste, les parenthèses engobantes sont automatiquement supprimées par l'interface. Pour analyser l'entrée, on utilise ici la fonction `parse-from-string` qui a pour but de mettre une chaine (c'est ce que renvoie la BD) sous la forme d'une liste de symboles LISP ; lorsqu'il n'y a qu'un seul élément elle renvoie celui-ci comme un atome. Voici un exemple de son fonctionnement :

```
(parse-from-string "5.45 methode-a 8")  -> (5.45 METHODE-A 8)
(parse-from-string "5.45")              -> 5.45
```

Bien évidemment l'utilisateur est libre de choisir une autre fonction de traduction. Notons enfin que lorsque le champs `parameter-def` est laisé vide (NIL) le bouton de sélection des paramètres dans la boite de construction d'exemples reste inactif.

6. Méthodes implémentées

Les méthodes sont implémentées dans le dossier “Method” et l’utilisateur est libre d’en rajouter. Les fichiers contenu dans ce dossier sont automatiquement chargées et interprétés au lancement du système. Cependant, lorsque l’on écrit une nouvelle méthode, il est important de noter que 1) les fichiers sont chargés selon l’ordre lexicographique de leur nom 2) qu’une classe ne peut pas héritée d’une autre non encore implémentée. Attention donc d’avoir une liste de fichiers dans le “bon ordre”.

Voici une liste des méthodes fournies avec Mo’K :

- Méthode de base : effectue juste une comparason des profils de fréquence des descripteurs

- Gmb: méthodes expérimentales “maison” en cours de mise au point ... Dans l’implémentation actuelle la valeur de similarité n’est pas normalisée et on est donc obligé d’allouer trop de mémoire pour stocker les similarité.
  - GMB0 : comparaison de profils plus choix glouton
  - GMB1: comparaison de profil plus aggrégation d’exemples de tailles similaires

- Méthode ASIUM : Implémentation optimale

- Hindle : implémentation à améliorer (coder la similarité en entier)

- Hindle avec poids positifs : on s’arrange pour avoir des pondération d’attributs >0.

- Grefenstette : implémentation à améliorer (coder la similarité en entier). L

- Dagan : implémentation à améliorer (coder la similarité en entier)

- Grishman : implémentation à améliorer (coder la similarité en entier)