Université Catholique de Louvain-la-Neuve
Ecole Polytechnique de Louvain
Travail de fin d'études - Année académique 2010-2011

Promoteur : Jean Vanderdonckt
Copromoteur : Jean-Charles Delvenne
Lecteur : Adrien Coyette

Benchmarking d'algorithmes pour la génération automatique de layout d'interfaces graphiques

Olivier Angély

Waterloo, le 29 août 2011

Résumé

De nombreux travaux ont été effectués pour trouver des manières d'aider les développeurs d'interfaces graphiques dans leur tâche. Les outils les plus évolués parmi ceux crées au cours de telles études allient les deux aspects principaux du design d'interfaces graphiques, la sélection des 'widgets' et leur placement dans les différentes fenêtres de l'interface. Certains fournissent un environnement d'édition plus facile à utiliser, ou encore permettent d'évaluer la qualité des résultats. Mais ces outils ne sont pas parfaits, et même les plus aboutis d'entre eux ont encore des défauts. Le problème de l'optimisation du layout des interfaces graphiques et complexe et vaste, et de nombreuses pistes n'ont pas été explorées.

Dans ce contexte, nous avons réalisé une étude comparative de plusieurs méthodes de placement des éléments. Nous avons également abordé le problème d'un autre angle, en essayant d'appliquer une nouvelle méthode de placement, les forces dirigées, déjà utilisées en théorie des graphes. Nous avons rassemblé des informations sur les différentes méthodes et leur implémentation, afin de fournir au lecteur une bonne vue d'ensemble de chacune, et de permettre une utilisation facile des méthodes, voire même une réimplémentation future sur d'autres plates-formes. Une comparaison sur base de tests a aussi été réalisée, afin de mettre en évidence les forces et faiblesses de chacune des méthodes étudiées, permettant d'envisager des solutions pour remédier aux points négatifs.

Table des matières

Introduction

De nos jours, les interfaces graphiques sont utilisées partout : que ce soit dans nos téléphones portables, les pages internet, les applications diverses,... Tous les domaines sont concernés, même ceux n'ayant pas trait à première vue à l'informatique.

Le développement d'une bonne interface nécessite beaucoup de temps et de travail de la part de ses développeurs. Mais comment caractérise-t-on une 'bonne' interface ?

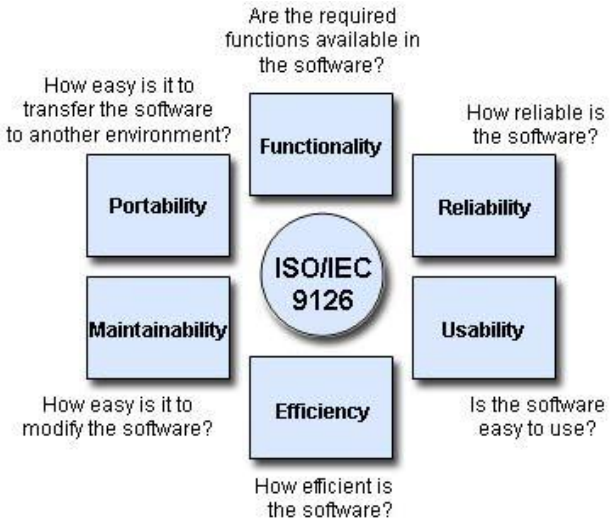


Fig. 1: Illustration de la norme ISO 9126

La norme ISO/CEI 9126 définit un langage commun pour modéliser les qualités d'un logiciel, ce qui peut être étendu dans ce cas-ci à une interface. Comme le montre la **Fig. 1**, cette norme sépare cette évaluation de qualité en 6 différents aspects. Une interface idéale devrait prendre en compte tous ces aspects. Cependant, on s'aperçoit vite que les interfaces actuelles n'en sont pas encore à ce stade : une interface développée avec soin sera conçue pour être fonctionnelle, reliable et efficiente. La portabilité et la facilité de maintenance de l'interface sont des domaines plus ardus, car dépendants du langage de programmation, de la méthode employée pour le layout, *et caetera*. Ainsi, une interface programmée en Java Swing ne pourra pas être aisément transférée dans un système fonctionnant dans un autre langage. De même, la modification de l'interface nécessitera de se plonger dans le code même, et la modification de la taille d'un élément peut contraindre le développeur à d'autres modifications, par effet 'boule de neige'.

Nous allons, dans ce travail, nous intéresser de plus près au dernier de ces aspects, l'usabilité.

Usability : extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use[1].

ISO définit donc l'usabilité comme le degré auquel un produit donné peut être utilisé par des personnes spécifiques afin d'atteindre des objectifs précis avec efficience, efficacité et satisfaction, tout cela dans un contexte d'utilisation donné. Il s'agit d'un critère en partie au moins subjectif, dépendant de l'appréciation qu'a l'utilisateur de l'interface. L'usabilité, selon Jakob Nielsen et Ben Shneiderman, se compose de

- la facilité d'apprentissage : un utilisateur novice pourra-t-il s'en sortir facilement avec l'interface ?
- l'efficience : avec quelle rapidité un utilisateur rôdé pourra-t-il accomplir ses tâches ?
- la facilité de mémorisation : le fonctionnement de l'interface est-il aisément retenu, permettant de retrouver rapidement l'efficacité après des périodes d'inutilisation ?
- les erreurs : les utilisateurs font-ils souvent des erreurs, de quelle gravité, et peuvent-ils facilement les corriger ?
- la satisfaction : l'interface est-elle agréable à utiliser ?

La satisfaction est le critère qui retiendra notre attention. Celle-ci relève en grande partie de l'aspect esthétique de l'interface, longtemps négligé dans ce domaine. Certaines études récentes traitent de l'influence de l'esthétique de l'interface sur l'impression que reçoit l'utilisateur face à l'interface, qui elle-même influe sur l'usabilité perçue de celle-ci. La beauté d'une interface est une qualité subjective, qui dépend de l'utilisateur. Cependant, des études ont pu mettre en évidence certaines tendances parmi les différentes personnes étudiées, ce qui a conduit à l'apparition de critères tentant de quantifier la beauté, ou du moins d'aider dans cette tâche. Ces critères sont nombreux, et peuvent être classés en plusieurs catégories. La liste ci-dessous, non-exhaustive bien entendu, donne un aperçu de ce classement :

- des critères dits 'physiques', qui évaluent la position des éléments dans l'interface, tels que la symétrie, la balance, l'alignement, la proportion,...
- des critères de composition de l'interface, qui cotent le choix des éléments, tels que la simplicité, l'économie, la neutralité,...
- des critères évaluant l'association (ou la dissociation) des éléments comme l'unité, la répartition, le groupement,...
- des critères d'ordre comme la séquentialité, la continuité,...
- des critères dits 'photographiques', basés sur des impressions visuelles, tels que l'arrondi, la stabilité,...

[1] ISO 9241-11 :1998(E), *Ergonomic requirements for office work with visual display terminals (VDTs) - Part 11 : Guidance on usability*

Ces critères, combinés ensemble, permettent de donner une valeur aux interfaces. Il devient dès lors intéressant d'exprimer le problème sous forme d'une optimisation :

$$\begin{aligned} &\max_{I} \ \mathbf{S}\left(\{c_i | c_i \in C\}\right) \\ &\text{with Position constraints} \\ &\qquad \text{Element constraints} \\ &\qquad \ldots \end{aligned}$$

où :

- I est l'interface ;
- C est l'ensemble des critères utilisés ;
- $\mathbf{S}$ est une fonction de ces critères.

Le problème que nous avons là est un problème extrêmement difficile à résoudre, et ce pour une très bonne raison : il n'existe aucune définition analytique connue de la fonction $\mathbf{S}$. Toute résolution analytique et algorithmique de cette optimisation est donc impossible à l'heure actuelle. C'est pourquoi de nombreuses études menées sur le sujet tentent de trouver des algorithmes permettant d'approcher l'optimisation de la fonction S ou de la substituer par d'autres fonctions qui sont, elles, calculables.

Ces algorithmes seraient utiles tout au long de la chaîne de conception et d'utilisation de l'interface : que ce soit au niveau de l'utilisateur de l'interface, qui verrait la qualité de son outil améliorée, ou au niveau du développeur et du concepteur de l'interface, qui, en profitant du feedback combiné des utilisateurs et des algorithmes, ainsi que de l'aide de ces derniers, pourraient produire plus rapidement de meilleures interfaces. Ce genre de méthodes permettraient donc d'économiser un temps précieux dans de nombreuses situations, que ce soit au niveau de la création, de l'évaluation ou d'autres tâches, tels le changement de support d'une application : imaginons une application tournant sur un ordinateur avec un écran large, devant être transplantée sur une machine plus petite, par exemple un iPad. Si l'on souhaite éviter des barres de défilement sur les bords de l'écran, un redesign de l'interface est obligatoire, ce qui demanderait du travail, alors que l'interface en elle-même ne change pas de fonction. Un algorithme permettant d'adapter une interface donnée à une fenêtre de taille spécifiée résulterait en un gain du temps normalement consacré à cette tâche, qui pourrait être réinvesti ailleurs.

Il s'agit donc d'un problème complexe et non-résolu à l'heure actuelle. De plus, pour ajouter à la contribution de ce travail, une nouvelle piste sera explorée en développant et en testant une nouvelle méthode de positionnement pour les interfaces.

1 Le problème

Reprenons pour ce faire la description sous forme d'optimisation énoncée dans l'introduction :

$$\begin{aligned} &\max_{I} \ \mathbf{S}\left(\{c_i | c_i \in C\}\right) \\ &\text{with Position constraints} \\ &\qquad \text{Element constraints} \\ &\qquad \ldots \end{aligned}$$

où :

- I est l'interface ;
- C est l'ensemble des critères utilisés ;
- $\mathbf{S}$ est une fonction de ces critères.

Le processus de création d'une interface est séparé en deux étapes, qui se recoupent et s'influencent entre elles : tout d'abord, il faut effectuer un choix judicieux des éléments, en fonction de la tâche à laquelle ils sont destinés, et peut-être même du type d'utilisateur ; ensuite, un placement approprié de ces éléments est nécessaire. Nous allons ici négliger l'aspect de sélection des éléments au profit de leur positionnement dans l'interface. Au vu des travaux réalisés dans le domaine, il apparait que les études sur la sélection des éléments sont à un stade plus avancé que leur positionnement, et considérer un seul des deux problèmes permet de concentrer les efforts dans cette direction.

1.1 Hypothèses et réécriture

Afin de pouvoir s'attaquer à la résolution de ce problème, nous allons poser certaines hypothèses sur la forme des données. Tout d'abord, afin de rendre le problème le plus simple possible et de nous concentrer sur l'aspect du positionnement des éléments, nous allons écarter les aspects relevant du contenu des éléments, c'est-à-dire leurs fonction, couleur, valeurs possibles, texte éventuel, *et caetera*. Ces derniers seront réduits à leur plus simple expression, et seront considérés comme des **rectangles**. Leur influence dans les critères utilisés sera déterminée uniquement par leurs **dimensions**, et leur **position** les uns par rapport aux autres et dans l'interface. Le raisonnement appliqué ici peut être étendu par la suite à d'autres critères plus complexes. De plus, nous allons considérer que toutes les données relatives aux éléments de l'interface peuvent être résumées en trois matrices : la matrice des dimensions des éléments, la matrice de leurs coordonnées dans la fenêtre spécifiée et la matrice d'adjacence.

La **matrice d'adjacence**, notée $\mathbf{A}$, est une matrice dont les éléments $A(i,j) \in \{0,1\}$ renseigne sur la présence d'un **lien d'adjacence** de l'élément i vers l'élément j. Le concepteur de l'interface, désirant pour une raison ou pour une autre qu'un élément i se trouve proche de l'élément j, entrera dans la matrice $A(i,j) = 1$.

Un lien réciproque entre deux éléments, à savoir $A(i,j) = A(j,i) = 1$, indique une **relation de groupement** entre les deux éléments. Le choix de cette forme pour les données du problème est motivée en partie par l'adaptation dans ce travail d'une fonction ayant trait au domaine de théorie des graphes. Ainsi, toutes les méthodes sont mises sur un pied d'égalité pour l'étude, en utilisant le même type de données. De plus, la description du problème obtenue est concise, simple à écrire et à modifier.

Le problème peut alors être réécrit

$$\begin{aligned} &\max_{G} \quad \mathbf{F}\,(A, dim, G) \\ &\text{with Adjacency constraints } (A) \\ &\qquad \text{Dimension constraints } (dim) \\ &\qquad \text{Screen constraints} \\ &\qquad \text{Spacing constraints} \\ &\qquad \ldots \end{aligned}$$

où :
- G est la matrice des positions des éléments dans l'interface ;
- A est la matrice d'adjacence ;
- dim est la matrice reprenant les dimensions des éléments ;
- F est une fonction de ces trois variables, basée sur des critères d'évaluation.

La fonction objectif F peut varier en fonction des algorithmes considérés, cependant la comparaison entre les différents critères sera assurée afin de constater les différences au niveau des résultats.

1.2 Objectif recherché

L'objectif de ce travail est d'effectuer une étude comparative d'une sélection de méthodes, afin de tester leurs performances respectives, et mettre en évidence leurs forces et leur faiblesses. Le but espéré est de pouvoir émettre des recommandations sur quelles méthodes utiliser dans quel cas de figure, ainsi que de rendre accessibles les différents algorithmes utilisés en fournissant des informations sur leur implémentation.

La section 2 reprendra un bref aperçu des méthodes et travaux notables effectués dans le domaine des interfaces graphiques, ceci pour donner une idée de l'avancement des recherches en la matière. Les section 3 à 5 s'intéresseront ensuite à la description des différents algorithmes implémentés et comparés dans le cadre de ce travail. Enfin, dans la section 6, nous reprendrons les résultats des différents tests réalisés afin de comparer les performances des méthodes considérées.

2 Etat de l'art

Dans la section précédente, nous avons défini précisément le problème étudié et les objectifs à atteindre du travail. Cette deuxième section contient un résumé des travaux notables dans le domaine du développement d'interfaces.

2.1 Processus d'analyse

Nous avons d'abord commencé par effectuer une recherche dans la littérature concernant les interfaces graphiques afin de trouver des articles en relation avec l'optimisation des interfaces. Ensuite, parmi les sources retenues, une sélection a été faite en fonction de leur lien avec le placement des éléments dans l'interface, qui est la branche de cette optimisation examinée ici. De plus, la présence de critères de comparaison et la qualité des résultats obtenus étaient un plus non négligeable. Finalement, les travaux restants après sélection ont été examinés selon la grille d'analyse suivante :

- **Description** de la recherche effectuée ;
- **Motivations** qui ont poussé à cette recherche ;
- **Pseudocode** du ou des algorithme(s) implémentés ;
- **Exemples** illustrant le fonctionnement de ceux-ci ;
- **Avantages** et **inconvénients** de la méthode utilisée ;
- **Analyse** critique des résultats obtenus.

2.2 SUPPLE

SUPPLE est un algorithme avancé qui aborde la génération d'interfaces graphiques comme un problème d'optimisation. Il crée des interfaces de manière à minimiser le travail de l'utilisateur, en considérant les contraintes liées au support sur lequel l'interface doit être affichée. Il tient donc à la fois compte des spécificités de l'interface elle-même, de celles du support et des caractéristiques de l'utilisateur auquel cette interface est destinée. De plus, SUPPLE traite non seulement du positionnement des éléments sur l'interface, mais également de leur sélection dans un panel de 'widgets' disponibles.

2.2.1 Motivation

La motivation derrière le développement de cet algorithme est de permettre à l'utilisateur d'accéder aux ressources informatiques indépendamment de la machine sur laquelle il ou elle travaille. C'est-à-dire qu'à l'aide de cet algorithme, une interface pourrait être efficacement affichée à la fois sur un écran d'ordinateur et sur un téléphone portable, sans avoir besoin de l'intervention d'un développeur pour redessiner celle-ci. De plus, les concepteurs de SUPPLE désirent pouvoir adapter l'interface en fonction des préférences et des capacités des utilisateurs eux-mêmes. Ainsi, leur utilisation de l'interface pourrait résulter

en des modifications de celle-ci pour leur permettre un accès plus aisé. La multitude de combinaisons de types d'utilisateurs et de support les ont donc poussés à trouver une solution automatisée à ce problème.

2.2.2 L'algorithme

Dans SUPPLE, le coût d'une interface est défini comme suit :

$$\$(\phi, \mathcal{T}) = \sum_{T \in \mathcal{T}} \sum_{i=1}^{|T|-1} N(\phi, e_{i-1}, e_i) + \mathcal{M}(\phi(e_i), v_{old}, v_{new})$$

où :
- N est un estimateur de l'effort engendré par la navigation entre les éléments ;
- $\mathcal{T}$ est la trace de l'utilisateur, c'est-à-dire un relevé de ses actions sur l'interface ;
- $\mathcal{M}$ est une fonction évaluant la perspicacité du choix d'un 'widget' pour un élément donné ;
- e_i est l'élément i ;
- ϕ est une instance de l'interface graphique I considérée.

Notons que la fonction $\mathcal{M}$ est uniquement dépendante du 'widget' choisi pour un certain élément, et non de la valeur assignée à l'élément durant son utilisation.

A l'aide de cette fonction, l'algorithme de représentation de SUPPLE est construit comme suit :

La condition $currentCost(vars) + remainingCostEstimate(vars) \geq bestCost$ permet d'éliminer de la recherche les noeuds supérieurs au meilleur score déjà atteint par une interface.

Ensuite, en exécutant $selectUnassignedVariable(vars)$ l'algorithme sélectionne la variable la plus contrainte, c'est-à-dire celle ayant le moins de valeurs possibles. L'algorithme effectue alors une recherche récursive sur toutes les valeurs possibles de cette variable. Ainsi, toutes les solutions sont explorées et l'on est certains d'aboutir à une solution optimale.

A chaque étape de la recherche, les nouvelles valeurs assignées sont propagées sur les contraintes, afin de vérifier si la solution partielle ne débouche pas sur des configurations impossibles, où certaines variables ne peuvent pas avoir de valeur dans le domaine.

Algorithm 1 Algorithme de recherche de SUPPLE

$vars$: ensemble des variables
$const$: ensemble des contraintes

if $propagate(vars, const) = 0$ **then**
  return
end if

if $currentCost(vars) + remainingCostEstimate(vars) \geq bestCost$ **then**
  return
end if

if $completeAssignment(vars)$ **then**
  $bestCost \leftarrow cost$
  $bestRendition \leftarrow vars$
  return
end if

$var \leftarrow selectUnassignedVariable(vars)$
for $\forall$ $value$ in $ordderValues(getValues(var))$ **do**
  $setValues(var, value)$
  $optimumSearch(vars, const)$
end for

$restoreDomain(var)$
$undoPropagate(vars)$

2.2.3 Exemples

Voici un petit exemple illustrant les capacités et le fonctionnement de SUPPLE. L'interface considérée, supposée située dans une salle de classe, est décrite par l'arbre suivant :

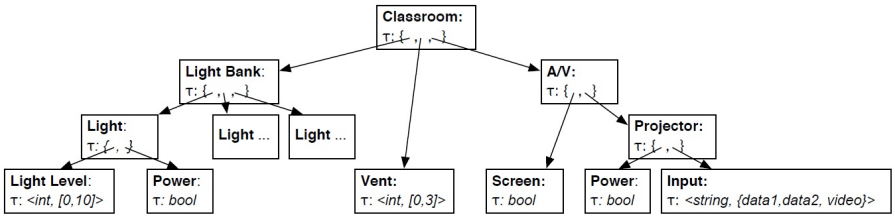


FIG. 2: Arbre définissant l'interface à optimiser

L'algorithme prend ces données en compte, et étant donné des contraintes physiques liés au support, telles un ensemble de 'widgets' mis à disposition et une taille d'écran, ainsi qu'une trace d'utilisateur spécifique, calcule la meilleure sélection et le placement optimal des différents éléments.

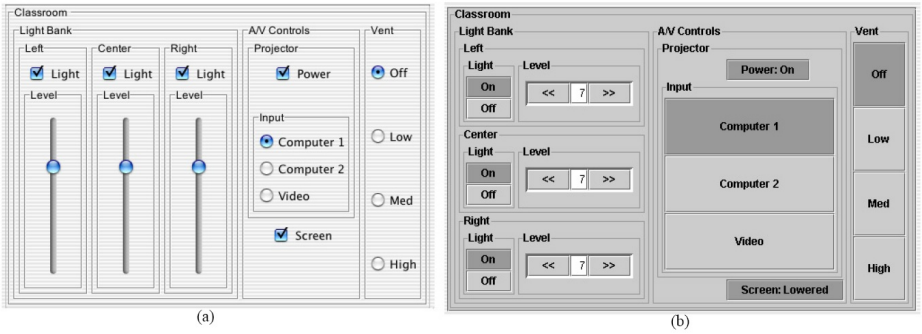
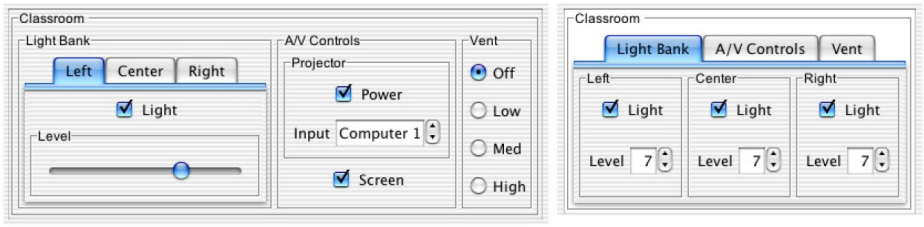


FIG. 3: L'interface de la salle de classe représentée sur **(a)** un appareil à pointeur, et **(b)** un appareil tactile

Les premiers résultats sont obtenus avec une trace d'utilisateur vide pour la figure **3 (a)**, et avec une trace signalant une utilisation fréquente des contrôles de luminosité pour **3 (b)**. De plus, la figure **3 (b)** est affichée sur un support tactile, ce qui se reflète dans la taille des boutons utilisés. La seconde illustration (**Fig. 4**) montre la même interface, cette fois représentée sur un support intermédiaire, avec le même ensemble de 'widgets' que pour la **Fig. 3 (a)**. La trace d'utilisateur est vide dans le cas **(a)** et fait état d'une transition fréquente entre les différents contrôles de lumière pour **(b)**. Enfin, la figure **5** montre l'interface affichée sur un support plus petit, de type téléphone portable. Les 'widgets' sont moins faciles



(a) (b)

FIG. 4: L'interface de la salle de classe sur un support plus petit, avec **(a)** une trace vide, et **(b)** de fréquentes transitions entre les contrôles de lumière

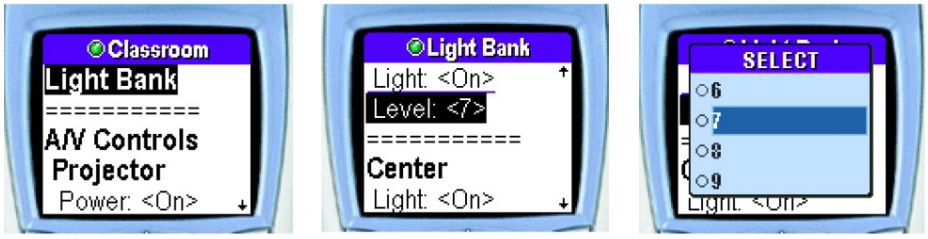


FIG. 5: L'interface de la salle de classe représentée sur un téléphone portable

à utiliser, mais l'on observe sur l'image que la navigation entre les différents contrôles de lumières reste assez facile.

2.2.4 Avantages, inconvénients et analyse

Un des points positifs majeurs de cet algorithme est qu'il intègre à lui seul les deux aspects de la construction d'une interface graphique, la sélection et le placement.

SUPPLE permet aux interfaces de s'adapter aux utilisateurs et aux différents supports, pour autant qu'il y ait des informations à leur sujet.

Ses performances semblent honnêtes sur les exemples sus-cités, avec moins de 2 secondes de temps d'exécution, grâce aux améliorations effectuées sur la recherche exhaustive.

L'algorithme nécessite cependant des traces d'utilisateurs pour fonctionner correctement, ce qui nécessite d'étudier le comportement de ces derniers. L'autre alternative est de créer une trace factice par analyse de l'interface, ce qui alourdit la tâche du développeur et/ou du concepteur. Ce problème est compensé par une application supplémentaire décrite dans une section ultérieure, ARNAULD.

Malgré que l'algorithme soit encore en plein développement, que des améliorations doivent encore être faites, celui-ci est prometteur, autant par son approche du problème que par ses résultats, et son évolution mérite d'être suivie avec attention.

2.3 ARNAULD

ARNAULD est un outil d'évaluation et d'aide au design des interfaces développé en parallèle avec SUPPLE, et basé sur l'interaction avec l'utilisateur. Les informations tirées de ces deux sources sont ensuite combinées pour paramétrer la fonction de coût utilisée dans le design des interfaces. L'interface est alors modifiée pour convenir à la nouvelle fonction, et les futurs layouts créés tiendront compte de ces nouvelles préférences. Deux méthodes sont utilisées pour obtenir de l'utilisateur les informations recherchées :

- critique d'exemples : l'utilisateur a la possibilité de modifier le choix de 'widgets' pour les différents éléments de l'interface. Les choix sont alors enregistrés comme des préférences, et résultent en des changements dans le layout de l'interface ;
- élection par préférence : ARNAULD demande également à l'utilisateur de choisir entre deux layouts différents pour une interface, ou deux 'widgets' pour un même élément. De nouveau, le programme tient un inventaire des choix effectués, qui sont pris en compte dans l'évaluation de la fonction à optimiser.

2.3.1 Motivation

Le développement de cette méthode est motivé par la difficulté rencontrée par les développeurs d'interfaces graphiques en tentant de déterminer des bons paramètres pour une fonction à optimiser. En effet, l'optimisation est de plus en plus utilisée dans ce domaine, afin d'obtenir des résultats de manière automatisée. Le problème étant que la forme et les paramètres de la fonction à optimiser sont des points difficiles à fixer, surtout si l'on veut tenir compte de la diversité des profils d'utilisateurs. Les paramètres peuvent changer avec les utilisateurs, et même avec le temps pour un même utilisateur. L'objectif des développeurs d'ARNAULD est d'éviter un processus d'ajustement de ceux-ci à la main, qui se révèle souvent long et sujet aux erreurs.

2.3.2 L'algorithme

L'algorithme utilisé est basé sur les méthodes dites à 'marge maximale', utilisées dans les machines à vecteur de support (SVM en abrégé).

Les SVM sont un ensemble de techniques d'apprentissage automatique servant dans la résolution de problèmes de discrimination et de régression. Les SVM sont des classifieurs peu complexes, et font partie des méthodes les plus rapides pour résoudre ce genre de problèmes. En fonction des problèmes, ce ne sont cependant pas les plus efficaces.

Les méthodes à marges maximale résolvent les problèmes d'optimisation d'interface en assignant à chaque variable la valeur satisfaisant le plus grand nombre possible de contraintes. Ces valeurs sont en outre choisies afin de maximiser la différence entre les deux membres de l'inégalité constituant la contrainte.

Formellement, les préférences des utilisateurs récupérées par les deux méthodes citées précédemment sont converties en inégalités de la forme

$$\sum_{k=1}^{K} u_k f_k(o'_{c_i}) - \sum_{k=1}^{K} u_k f_k(o_{c_i}) \geq m - \xi_i$$

et ce pour tout degré c_i de satisfaction de la contrainte, où :
- o et o' sont une paire de choix proposés à l'utilisateur, avec $o \succ o'$;
- les u_k sont les paramètres recherchés par la méthode ;
- m est une variable de marge partagées entre les contraintes ;
- ξ_i est une variable d'écart.

L'objectif à maximiser est alors de la forme $m - \alpha \sum \xi_i$, où α est un paramètre contrôlant la tolérance à la violation des contraintes. Ceci doit être combiné à une autre optimisation visant à tenir compte de la connaissance obtenue précédemment :

$$\begin{aligned} &\min \quad m' + \alpha' \sum \xi'_k \\ &\text{with } |u_k - u'_k| \leq m' + \xi'_k \ \forall k \in [1 \dots K] \end{aligned}$$

Les deux fonctions objectifs sont alors combinées pour ensuite être utilisées dans l'optimisation

2.3.3 Exemples

(a) (b)

FIG. 6: Critique d'exemples dans SUPPLE (1) : **(a)** interface initiale et **(a)** modification du 'widget' de contrôle de volume

Les figures **6** et **7** illustrent le procédé de critique d'exemples. L'utilisateur se retrouve face à l'interface de la figure **6(a)** et, n'étant pas satisfait par le choix du

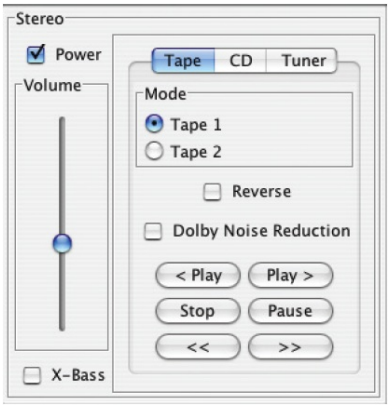


FIG. 7: Critique d'exemples dans SUPPLE (2) : Interface optimisée pour la nouvelle fonction de coût

'widget' du contrôle de volume, demande à SUPPLE que ce dernier soit remplacé par un 'slider', comme montré dans la figure **6(b)**. L'interface est alors modifiée (**Fig. 7**) pour convenir aux attentes de l'utilisateur, et ARNAULD stocke ces informations afin de mieux paramétrer les fonctions de coût de SUPPLE dans les optimisations futures.

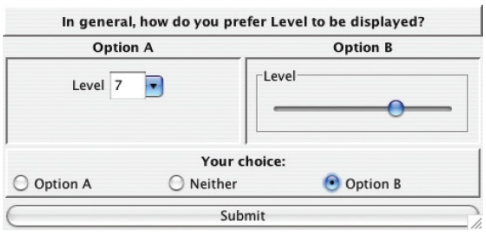


FIG. 8: Election par préférence (1) : Choix du 'widget' du contrôle de l'intensité des lumières

Les figures **8** et **9** donnent un aperçu du procédé d'élection par préférence, basé sur l'interface de la salle de classe déjà présentée dans la section sur SUPPLE. ARNAULD demande d'abord à l'utilisateur quel 'widget' lui convient le mieux pour représenter le contrôle de l'intensité des lumières (**Fig. 8**). Le choix de l'utilisateur, à savoir les 'sliders', a cependant des conséquences sur d'autres parties de l'interface de la salle de classe. ARNAULD propose alors deux alternatives à l'utilisateur (**Fig. 9**) : une ne respectant pas sa préférence, mais permettant à tous les éléments d'être affichés dans une seule fenêtre ; et une seconde, où les contrôles d'intensité des lumières sont représentés par des 'sliders' comme demandé, mais où les différents éléments sont répartis dans plusieurs onglets pour respecter les contraintes de taille de l'interface.

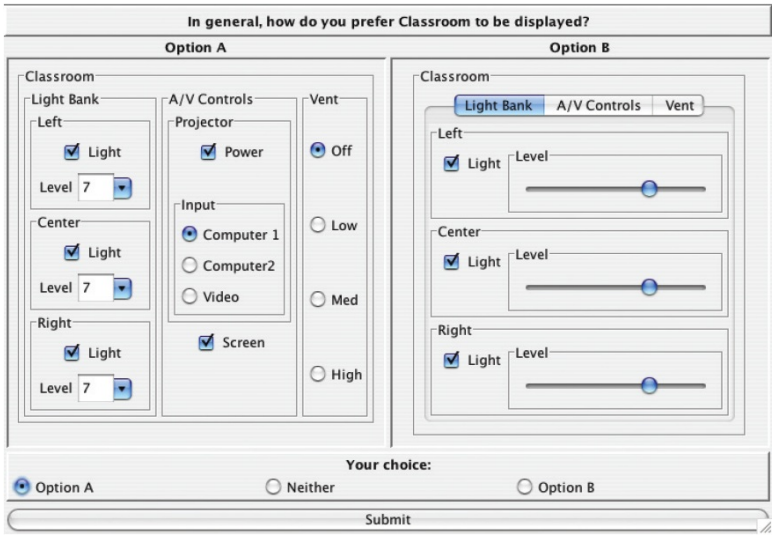


FIG. 9: Election par préférence (2) : Alternatives proposées à l'utilisateur, en conséquence de son choix

2.3.4 Analyse

Une étude pilote réalisée par les développeurs d'ARNAULD conclut que l'algorithme est pratique et facile d'utilisation, convenant aux exigence des sujets de l'expérience, malgré quelques améliorations à apporter, comme par exemple un bouton pour annuler les modifications faites.

L'algorithme produit des résultats assez rapidement, endéans la seconde pour un problème comprenant jusqu'à 40 variables. Les résultats sont suffisamment précis, et quelques secondes de plus permettent de diminuer fortement les erreurs subsistant.

Cependant, ARNAULD impose un passage par une série de questions-réponses pour choisir entre les différentes alternatives proposées, ce qui peut se révéler fastidieux pour certains. Le temps nécessaire pour cette tâche n'est heureusement pas trop long.

De nouveau, la méthode utilisée parait prometteuse, et la combinaison de celle-ci avec SUPPLE donne un outil puissant de création, modification et évaluation d'interfaces graphiques. Cependant, l'algorithme nécessite toujours quelques perfectionnements.

2.4 AIDE

AIDE est un outil d'assistance à la création et à l'évaluation d'interfaces graphiques. Il se base sur une série de critères d'évaluation, appelés métriques, de deux types différents : des mesures intervenant au niveau du design sématique, et

des mesures évaluant l'esthétique de l'interface. Ces métriques sont au nombre de 5 : l'efficacité, l'alignement, la balance horizontale et verticale, et les contraintes. AIDE est l'un des premiers outils permettant de combiner la création, l'évaluation et la modification d'interfaces en une seule application. Cependant, celui-ci est uniquement destiné au placement des différents 'widgets', supposant que ceux-ci ont été choisis judicieusement auparavant. AIDE n'est donc pas un outil auto-suffisant, et ses développeurs préconisent son utilisation en parallèle avec d'autres algorithmes.

2.4.1 Motivation

Comme déjà dit précédemment, la création d'interfaces n'est pas une tâche aisée. Celles-ci doivent être imaginées en fonction de la tâche et des utilisateurs à laquelles elles sont destinées. Les métriques, notamment celles citées dans le paragraphe précédent, peuvent aider à appréhender certains aspects difficiles de ce travail, de même que des aspects que l'on tend à négliger. Elles allègent donc le travail du développeur, lui faisant gagner du temps et lui permettant de concentrer son attention sur d'autres points du processus. De plus, l'automatisation de l'évaluation permet d'obtenir non seulement des résultats rapides, et ce à n'importe quelle étape du design, mais également leur incorporation dans des algorithmes de génération d'interfaces. C'est dans ce but que les développeurs se sont attelés à la programmation d'AIDE.

2.4.2 L'algorithme

L'algorithme propose à l'utilisateur de choisir les métriques à optimiser parmi celles disponibles, en attribuant des poids selon l'importance souhaitée du critère.

L'efficacité est le score de Layout Appropriateness de l'interface. Elle évalue la distance parcourue par le curseur de l'utilisateur lors de l'accomplissement de ses actions. Pour plus de détails, vous êtes invités à consulter la section dédiée à l'algorithme du même nom.

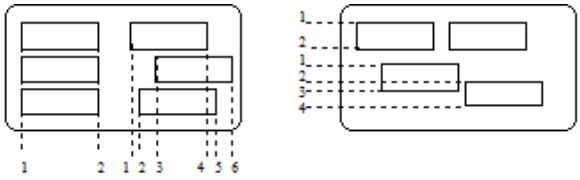


Fig. 10: Alignement : à gauche, un placement aligné et mésaligné en colonne ; à droite, un placement aligné et mésaligné en ligne

L'alignement (**Fig. 10**) évalue à quel point les différents éléments sont alignés verticalement et horizontalement.

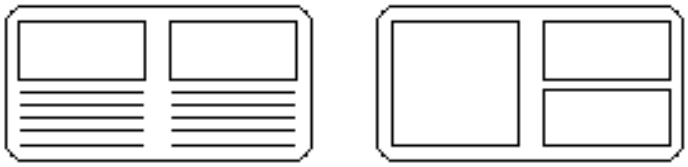


FIG. 11: Balance : à gauche, un placement balancé symétrique ; à droite, un placement balancé asymétryque

La balance évalue la bonne répartition de l'espace utilisé par les éléments dans l'écran, en considérant tour à tour les moitiés haut et bas, puis droite et gauche. La répartition idéale n'implique pas la symétrie (voir **Fig. 11**), mais au moins une même occupation dans chacune de ces paires.

Les contraintes regroupent toutes les indications spécifiées par le concepteur de l'interface. A l'aide d'une fonction d'importance associé par ce dernier à la contrainte, un poids peut être calculée pour tout état de la contrainte, qu'elle soit violée ou non. Ainsi un score de respect des contraintes peut être attribué au layout.

Par définition, un layout optimal calculé par l'algorithme doit respecteur toutes les contraintes spécifiées. Ensuite, en fonction des critères sélectionnés, AIDE place les 'widgets' sur une grille de layout, de manière la plus satisfaisante possible.

2.4.3 Exemples

Les quelques images rassemblées ici servent à idonner une idée du fonctionnement d'AIDE.

L'interface de contrôle d'AIDE est représentée sur la **Fig. 12**. On observe que le critère de balance est exclu de l'optimisation, et que des poids différents ont été attribués à l'efficience (4) et à l'alignement (1).

La **Fig. 13(a)** montre l'interface optimisée selon le critère d'alignement. Les différentes valeurs des critères sont également calculés et interprétés dans le panneau de contrôle (**Fig. 13(b)**).

La dernière figure (**Fig. 14**) montre une interface créée manuellement. Le tableau suivant regroupe les valeurs des différents critères :

Metric	**Interpretation**	**Value**
Efficiency	Very good	103
Alignment	Very good	114
Bottom-Top Balance	Well Balanced	102
Left-Right Balance	Slightly Right Heavy	121
Constraints	1 of 3 violated	50

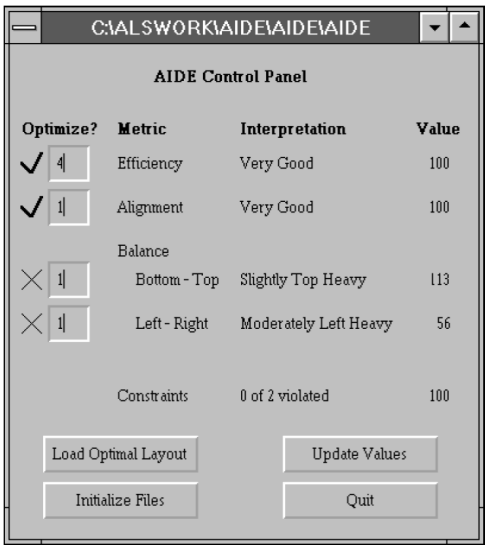
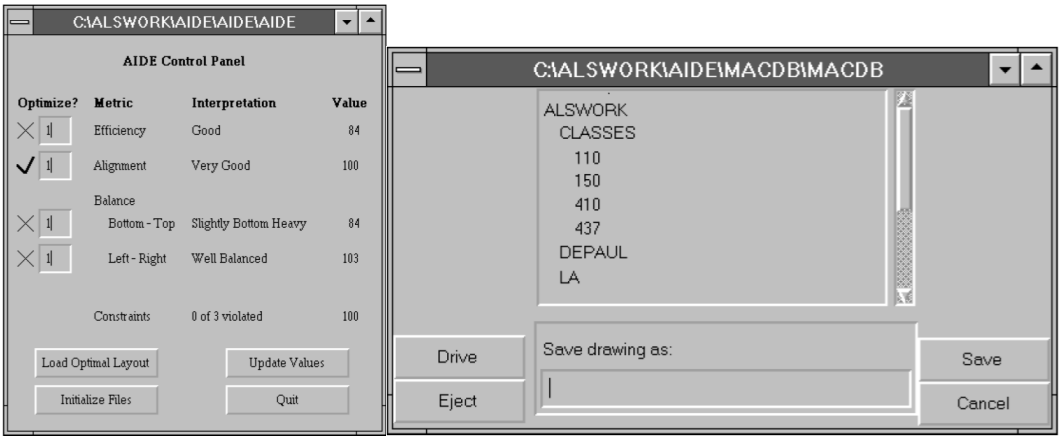


FIG. 12: AIDE - Interface de contrôle



(a) (b)

FIG. 13: AIDE - Optimisation selon le critère d'alignement : **(a)** interface de contrôle et **(b)** interface optimisée

L'efficacité des opérations est augmentée car beaucoup de séquences d'action des utilisateurs commencent par la sélection du disque (*drive*). Cependant le prix à payer pour cela est la violation d'une contrainte, jugée importante au vu du score de 50 attribué au respect de l'ensemble des contraintes. C'est au développeur de l'interface de juger de la valeur de l'amélioration apportée à l'interface, et de peser le pour et le contre de ce choix.

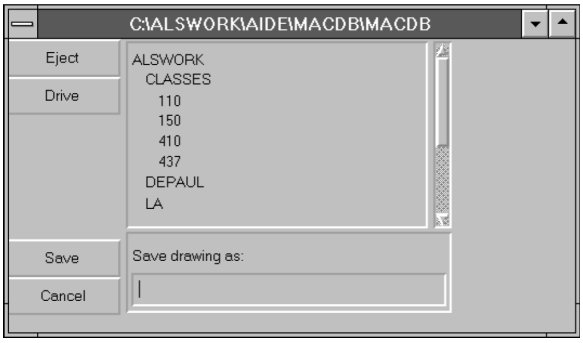


FIG. 14: AIDE - Interface crée manuellement

2.4.4 Analyse

Comme dit précédemment, AIDE n'est pas conçu pour développer des interfaces automatiquement, et doit être utilisé avec soit l'intervention d'un développeur, soit en addition d'autres algorithmes.

Cependant, l'évaluation effectuée par AIDE peut se révéler très utile en tant que telle, que ce soit pour évaluer ou même modifier des interfaces selon certains critères. Par exemple, une interface entièrement basée sur la balance verticale et horizontale pourra être adaptée pour tenir compte de l'alignement, résultant ainsi en un layout plus esthétique. De ce point de vue, l'algorithme atteint bien le but recherché par ses concepteurs.

Il n'y a pour l'instant que peu de critères d'évaluation disponibles, mais les développeurs ne comptent pas en rester là, et ont l'intention de rajouter d'autres métriques au champ d'action de AIDE. Cela résulterait en une plateforme polyvalente d'évaluation d'interfaces.

2.5 Conclusion

Afin de conclure cette section, nous avons effectué une comparaison des différents algorithmes présentés. La comparaison est effectuée selon les axes suivants :

- rapidité d'exécution ;
- qualité des résultats obtenus ;
- l'algorithme s'occupe-t-il du placement des éléments ?
- l'algorithme s'occupe-t-il de la sélection de ces éléments ?
- une intervention extérieure d'un développeur, chercheur, ou autre est-elle nécessaire pour fournir des informations essentielles ?
- l'algorithme effectue-t-il une évaluation de la qualité de ses résultats ?
- l'algorithme est-il utilisable seul ?

	Rapidité	Résultats	Placement	Sélection
Supple	$< 2s$	good	**V**	**V**
Arnauld	$< 4s$	good	**X**	**V**
AIDE		good	**V**	**X**
Supple + Arnauld	$< 6s$	very good	**V**	**V**

	Intervention ?	Evaluation	Stand-alone
Supple	**V**	**X**	**V**
Arnauld	**X**	**X**	**X**
AIDE	**X**	**V**	**V**
Supple + Arnauld	**X**	**X**	**V**

3 Algorithme à forces dirigées (SPRING)

Dans la section précédente, nous avons passé en revue différents travaux réalisés dans le domaine de l'optimisation des interfaces graphiques. Nous allons maintenant aborder le premier des algorithmes étudiés ici. Il s'agit d'une méthode initalement utilisée en théorie des graphes : les forces dirigées. Le choix de celle-ci a été fait en raison de la similitude entre la représentation d'un graphe et celle d'une interface. De plus, la méthode à forces dirigées a tendance à produire des représentations de graphes où les noeuds sont bien répartis sur l'aire disponible. Dès lors, on peut s'attendre à ce que les 'layouts' produits par cette méthode aient une bonne balance verticale et horizontale. Finalement, les forces dirigées pouvant être appliquées assez facilement sur des graphes allant jusqu'à plusieurs dizaines de noeuds, ce choix semble un bon point de départ pour un algorithme de positionnement.

Nous ferons par la suite référence à cet algorithme sous le nom SPRING (et dérivés).

3.1 Les bases

Les méthodes à forces dirigées sont utilisées pour calculer la représentation d'un graphe. Ces méthodes sont utilisées entre autres pour représenter de graphes planaires sans croisement d'arètes. Elles ne nécessitent que des informations contenues dans la structure du graphe elle-même, par exemple dans la matrice d'adjacence des noeuds, et renvoie les coordonnées de tous les noeuds du graphe dans la solution qu'elles calculent. Celui-ci peut ensuite être reconstruit et représenté à l'aide de cette même matrice.

Les méthodes de base fonctionnent assez bien sur des petits graphes, mais dès que la taille de ceux-ci devient trop importante (quelques centaines de noeuds), celles-ci perdent de leur intérêt. En effet, le modèle physique utilisé possède souvent plusieurs optima locaux, et plus le problème est grand, plus il est difficile de trouver l'optimum global parmi les locaux. Des améliorations sont évidemment possible, mais dans le cadre de ce travail, la taille des interfaces considérées ne justifiait pas une implémentation directe de telles modifications.

Voici une description de l'algorithme de base :

On considère le graphe et les relations entre noeuds à travers la matrice d'adjacence dont on dispose. Nous remplaçons tous les noeuds par des anneaux, et les arètes par des ressorts, que l'on attache aux anneaux correspondant au noeuds qu'elles relient. Pour obtenir la représentation planaire, il suffit alors de 'relâcher' le système, qui va alors tendre vers un état de moindre énergie (un équilibre stable donc).

Tous les ressorts sont identiques, et réagissent selon une force d'intensité logarithmique :

$$F_S(d) = c_1 \cdot \log\left(\frac{d}{c_2}\right)$$

où c_1 et c_2 sont des constantes, et d est la longueur du ressort.
Observons que la distance optimale pour le ressort, c'est-à-dire celle où la force qu'il exerce est nulle, est égale à c_2. De plus, deux noeuds n'étant reliés par aucune arète se repoussent tels des particules de même charge, selon une force fonction de la distance, identique pour tous les noeuds :

$$F_R(d) = \frac{c_3}{d}$$

où c_3 est une constante et d est toujours la distance entre les noeuds considérés. On suppose celle-ci non nulle.

Le problème étant assez difficile à résoudre tel quel, des méthodes d'optimisation combinatoire entrent en jeu à ce stade : la plus couramment utilisée dans ce cas est la méthode du gradient, ou 'plus forte pente'.

Une force dérivant d'un potentiel, le gradient en un point de l'énergie du système (c'est-à-dire la plus forte pente) est le vecteur de la résultate des forces en ce même point. Dès lors, le minimum peut être recherché itérativement en avançant chaque noeud dans la direction de la résultante des forces qui y sont appliquées. La taille du pas est habituellement une fraction de l'intensité de la résultante.

Le pseudocode suivant résume les étapes de l'algorithme :

Algorithm 2 Méthode à forces dirigées : SPRING_G(G)

```
Placement des noeuds de G à des positions aléatoires
for i = 1 → M do
    Calculer la résultante des forces en chaque noeud
    Bouger chaque noeud de c4·(résultante en ce noeud)
end for
Afficher le graphe
```

où :
- *c_4 est la constante citée précédemment, définissant la taille du pas*
- *G représente le graphe considéré, et toutes les informations nécessaires.*

3.2 Application au problème donné

Plusieurs problèmes apparaissent lorsqu'on essaie de transférer cette méthode au domaine de la représentation des interfaces graphiques :

- nous n'avons plus des noeuds mais des éléments avec une aire et des dimensions non nulles. Il faut en tenir compte lorsqu'on exprime le champ d'attraction et de répulsion ;
- la zone d'affichage a également des dimensions précises ;

– les éléments peuvent se superposer entre eux, mais pas pénétrer la bordure de la fenêtre. Les bords de la zone d'affichage devront être considérés comme éléments à part entière, avec des forces d'attraction et de répulsion bien définies ;

3.2.1 Les champs d'attraction et de répulsion

Comme dit auparavant, les éléments à placer, c'est-à-dire les 'widgets' de l'interface, ont des dimensions non nulles, contrairement à ce qui a été dit auparavant. Les champs d'attraction et de répulsion de ces derniers doivent donc être modifiés.

Nous aimerions donc que
– deux éléments reliés par une arète s'attirent ;
– deux éléments non-reliés se repoussent ;
– que deux éléments qui sont superposés se repoussent, avec une intensité inversément proportionnelle à la distance entre leurs centres de gravité.

Les forces s'exercent uniformément sur les éléments, et peuvent donc être simplifiées en un vecteur sur leur centre de gravité. Nous allons en plus considérer que tous les moments sont nuls, c'est-à-dire qu'aucun élément ne pourra pivoter. Cela simplifie grandement le problème au niveau physique.

Les composantes de la **force d'attraction** peuvent être considérées séparément :

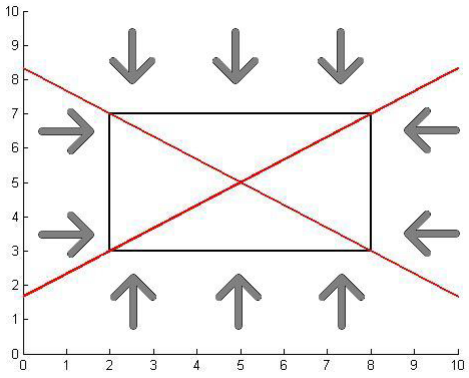


FIG. 15: Champ d'attraction d'un 'widget'

– un élément sera attiré vers un autre dans la direction perpendiculaire à son plus proche côté. Le champ d'attraction d'un 'widget' donné possède donc des irrégularités, et peut dès lors être qualifié de brisé (voir **Fig. ??**) ;

– la composante parallèle à ce côté le plus proche sera une composante d'alignement, dont nous discuterons ultérieurement.

L'intensité choisie pour la force d'attraction est logarithmique, mais cette fois bornée pour éviter les intensités infinies et/ou trop grandes :

$$\max\left(c_1 \cdot \log\left(\frac{d(i,j)}{D_{ideal}}\right), L\right)$$

où :
- c_1 est une constante d'ajustement ;
- $d(i,j)$ la distance entre les éléments i et j ;
- D_{ideal} la distance 'idéale' entre ces deux éléments ;
- L la borne inférieure sur la force d'attraction.

Lorsque les éléments sont éloignés, la force est grande. Elle décroit lorsqu'ils se rapprochent, jusqu'à devenir nulle lorsque la distance optimale est atteinte.

Si jamais deux éléments se superposent (la distance centre à centre est inférieure à l'optimum), la force d'attraction décroit très rapidement dans les nombres négatifs (jusqu'à rencontrer la borne inférieure), ce qui résulte en un écartement des éléments considérés.

Deux éléments non-reliés par une arète **se repoussent** mutuellement dans des directions opposées, prises à partir de leurs centres respectifs.

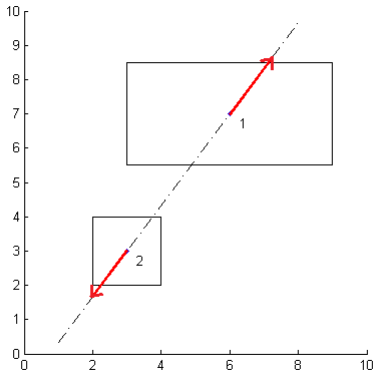


FIG. 16: 'Widgets' se repoussant mutuellement

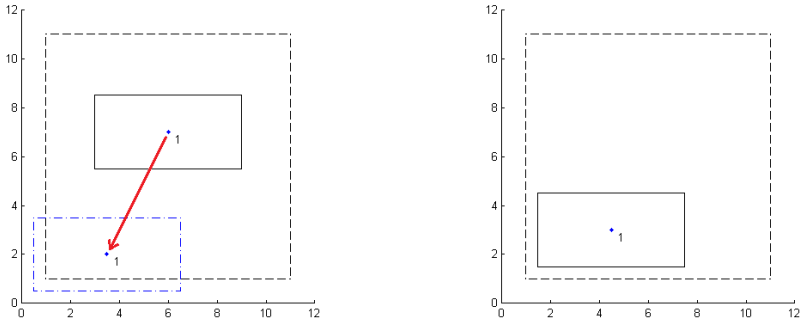
La forme choisie pour l'intensité de la répulsion est très semblable à celle de l'attraction, la différence se trouvant dans la direction de la force et le signe qu'on lui attribue. Par exemple, si l'objet considéré est situé dans le quadrant supérieur ou inférieur par rapport à l'objet répulseur, la force qu'il subira sera

$$f_x = \frac{dim(j,2)}{dim(j,1)} \cdot \frac{d(i,j)}{d(i,j)} \cdot f_y$$
$$f_y = \max\left(L, \min\left(U, c_1 \cdot \log\left(\frac{d(i,j)}{D_{ideal}}\right)\right)\right)$$

où :
- c_1 est une constante d'ajustement ;
- $d(i,j)$ la distance entre les éléments i et j ;
- D_{ideal} la distance 'idéale' entre ces deux éléments ;
- L et U sont les bornes inférieures et supérieures sur la force de répulsion ;
- dim est la matrice des dimensions des éléments de l'interface.

3.2.2 La fenêtre d'affichage

Comme dit précédamment, la zone en-dehors de la fenêtre d'affichage doit être imperméable aux éléments à afficher. Si le déplacement d'un élément le pousse en-dehors de la fenêtre, celui-ci sera réduit au déplacement maximum possible dans la fenêtre, sans toutefois changer la direction donnée (**Fig. 17**).



(a) (b)

FIG. 17: Illustration de l'imperméabilité des bords : **(a)** déplacement prévu initialement et **(b)** position finale de l'élément

Nous aimerions également que les bords soient des éléments à part entière, afin de pouvoir exprimer des relations d'adjacence entre ceux-ci et certains éléments du graphe : par exemple, pouvoir spécifier qu'une barre de titre ou une bannière de présentation doit se trouver au sommet de la page, au-dessus des autres éléments.

Il faut donc modifier la matrice d'adjacence du graphe, en lui incluant 4 colonnes supplémentaires, permettant d'exprimer les relations nécessaires entre

les éléments et les bords. L'équation **(1)** illustre la nouvelle forme de la matrice d'adjacence. Les bords ont des coordonnées bien précises, et sont immobiles. Cependant, ceux-ci participent au champ de force du système en attirant les éléments qui leur sont reliés, et en repoussant ceux qui ne le sont pas. Les intensités des forces d'attraction et de répulsion des bords sont de la même forme que celles des éléments.

$$A = \left[\begin{array}{cccccc|cccc} & & & & & & & & & \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ \underbrace{\qquad\qquad\qquad\qquad\qquad\qquad}_{\text{élements}} & & & & & & \underbrace{\qquad\qquad\qquad\qquad}_{\text{côtés}} & & & \end{array}\right] \tag{1}$$

3.2.3 Pseudocode de l'algorithme modifié

Algorithm 3 Méthode à forces dirigées appliquée aux interfaces : SPRING(I)

```
Placement des éléments de I à des positions aléatoires
while les déplacement des éléments sont supérieurs à un seuil ϵ do
    Calculer la résultante des forces en le centre de gravité de chaque élément i
    Calculer les déplacements D_i = c_4·(résultante en ce centre de gravité)
    if un élément i sort de la zone d'affichage then
        Réduire son déplacement D_i au maximum possible à l'intérieur de la
        fenêtre
    end if
    Bouger chaque élément de D_i
end while
Afficher l'interface
```

où :

- ϵ est le seuil de tolérance en-deçà duquel les déplacements sont considérés comme indiscernables, stoppant ainsi l'itération de l'algorithme ;
- c_4 est la constante définissant la taille du pas ;
- I représente l'interface considérée, et toutes les informations nécessaires (matrice d'adjacence, dimensions et nombre des éléments, etc).

3.3 Problèmes rencontrés et améliorations

L'algorithme tel que décrit au point précédent présente plusieurs vices de fonctionnement qui nuisent grandement à son efficacité :

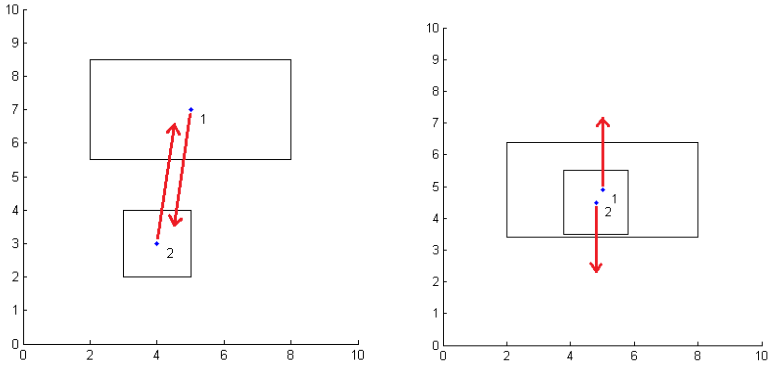
- Tout d'abord, lors de l'exécution, il est très improbable que deux éléments devant échanger de place se croisent en se chevauchant.

– Ensuite, étant donné une matrice d'adjacence, il se peut qu'il y ait trop d'éléments pour une seule fenêtre de l'interface.

– De plus, l'algorithme est un algorithme probabiliste, qui nécessite un nombre plus ou moins important d'itérations pour obtenir une solution correcte.

– Finalement, le grand nombre d'itérations à effectuer rend le traitement manuel des résultats long et fastidieux.

3.3.1 Croisement des éléments



(a) (b)

FIG. 18: Illustration du problème lié au croisement de deux éléments : **(a)** direction initiale des forces et **(b)** répulsion mutuelle des deux éléments

Le problème du croisement des éléments est dû à la forme logarithmique choisie pour l'intensité de la force. En effet, lorsque la distance séparant deux éléments devient proche de zéro, la force, elle, tend vers l'infini (voir **Fig. 18**). Malgré les bornes imposées sur celle-ci, la répulsion reste trop importante pour permettre le croisement de deux éléments.

Plusieurs solutions sont envisageables à ce stade :
– passer le problème en 3 dimensions, en permettant aux éléments de se déplacer 'verticalement' selon l'axe z ;
– ajouter une composante aléatoire ('température'), décroissante au fil du temps, au déplacement des différents éléments, comme dans l'algorithme de Fruchterman & Reingold ;
– faire évoluer les éléments sur des plans d'altitudes différentes, donnant ainsi une distance minimale à deux éléments qui se chevauchent.

La solution qui a été retenue dans cette étude est la première. Les éléments partent d'une position aléatoire en 3 dimensions, et sont attirés vers le plan $z = 0$.

L'intérêt de la 3e dimension, telle qu'elle est implémentée dans cette méthode, est l'accroissement de la distance entre les éléments : les centres de gravité de deux éléments superposés ne coincident pas nécessairement, car la 3e coordonnée n'est probablement pas la même. Dès lors, il est possible que l'intensité des forces ne soit pas suffisamment élevée pour empêcher le croisement.

Dans l'implémentation, cela se traduit par un facteur de proportionnalité

$$\frac{\text{distance-plan}}{\text{distance-3D}}$$

qui multiplie les intensités logarithmiques des forces d'attraction et de répulsion :

$$\max\left(c_1 \cdot \log\left(\frac{d(i,j)}{D_{ideal}}\right) \cdot \frac{\mathbf{d(i,j)}}{\mathbf{d_{3D}(i,j)}}, L\right)$$

pour l'attraction, où :
- c_1 est une constante d'ajustement ;
- $d(i,j)$ la distance entre les projections orthogonales des éléments i et j sur le plan $z = 0$;
- D_{ideal} la distance 'idéale' entre ces deux éléments ;
- L la borne inférieure sur la force d'attraction ;
- $d_{3D}(i,j)$ est la distance entre les éléments i et j, mais calculée dans l'espace en 3 dimensions.

$$f_x = \frac{dim(j,2)}{dim(j,1)} \cdot \frac{d(i,j)}{d(i,j)} \cdot f_y$$
$$f_y = \max\left(L, \min\left(U, c_1 \cdot \log\left(\frac{d(i,j)}{D_{ideal}}\right) \cdot \frac{\mathbf{d(i,j)}}{\mathbf{d_{3D}(i,j)}}\right)\right)$$

pour la répulsion, où :
- c_1 est une constante d'ajustement ;
- $d(i,j)$ la distance entre les éléments i et j ;
- D_{ideal} la distance 'idéale' entre ces deux éléments ;
- L et U sont les bornes inférieures et supérieures sur la force de répulsion ;
- dim est la matrice des dimensions des éléments de l'interface.
- $d_{3D}(i,j)$ est la distance entre les éléments i et j, mais calculée dans l'espace en 3 dimensions.

Un premier bilan de cette amélioration, basé sur quelques tests simples, montre que, bien que celle-ci ne résolve pas entièrement le problème de l'efficacité de l'agorithme, elle améliore déjà grandement la probabilité d'obtenir une solution acceptable. Cela contribue à réduire le nombre d'itérations nécessaires, et donc la mémoire et le temps nécessaires à l'exécution de l'algorithme.

3.3.2 Gestion du nombre d'éléments

Dans certains cas, la somme des aires de tous les éléments à placer dans l'interface est largement supérieure à l'espace disponible dans une seule fenêtre. Dans cette situation, superposer les éléments n'est pas une solution viable. Il n'y a donc qu'une seule solution possible : diviser l'interface en plusieurs fenêtres de charges acceptables.

Il faut donc une série de fonctions capables

- d'évaluer le nombre de liens entre éléments ou groupes d'éléments ;
- de les séparer en groupes élémentaires, c'est-à-dire ne contenant que des éléments reliés par des relations de groupement ;
- de placer ces groupes dans les différentes fenêtres, en essayant de préserver au maximum les relations entre éléments de groupes différents.

Par hypothèse, nous savons que, dans le graphe représentant l'interface, une arète dirigée (1 en (i,j) et 0 en (j,i) dans la matrice d'adjacence A, ou inversément) représente une relation d'adjacence, et une arète bidirectionnelle (1 en (i,j) et en (j,i) dans A) une relation de groupement.

Une simple routine gloutonne (voir **Algorithme 4**) éliminant les groupes élémentaires un à un fera l'affaire pour la **séparation**.

Algorithm 4 Algorithme glouton de séparation des éléments

N : ensemble des éléments non-traités
A : matrice d'adjacence des éléments
G : ensemble des groupes élémentaires
g : groupe couramment traité
i : index de l'élément de g couramment traité
while $N \neq \emptyset$ **do**
  $g \leftarrow N(1)$
  $N \leftarrow N \backslash N(1)$
  while tous les éléments de g n'ont pas été traités **do**
    $i \leftarrow$ le premier élément non-traité de g
    $g' \leftarrow$ tous les éléments groupés avec l'élément i de g (j tels que $A(i,j) = A(j,i) = 1$)
    $g \leftarrow$ l'union disjointe de g et g'
  end while
  $G \leftarrow G \cup g$
end while
Renvoyer G

Pour **trier les groupes obtenus par fenêtre**, il suffit de placer dans une fenêtre un des groupes (par exemple celui ayant le plus de liens avec le bord supérieur, ou le plus grand), et ensuite, par une routine gloutonne, ajouter itérativement les groupes ayant le plus de liens avec les éléments déjà présents dans la fenêtre, et ce jusqu'à atteindre le seuil de saturation. Ensuite, on répète le processus avec une nouvelle fenêtre de l'interface, jusqu'à ce qu'il n'y ait plus de groupes élémentaires à placer.

3.3.3 Réduction du nombre d'itérations

Bien que le passage en 3 dimensions ait amélioré l'efficacité de l'algorithme, ce n'est pas encore suffisant pour le qualifier d'efficace. Le problème majeur réside dans la position initiale attribuée aux différents éléments de l'interface. Si les éléments sont mal placés ou mal dispersés au début d'une itération, il est peu problable que celle-ci aboutisse à un résultat concluant. Ceci est fortement lié au problème de croisement des éléments.

Un algorithme effectuant un pré-traitement du problème, plaçant les éléments à des positions plus favorables, serait donc extrêmement utile. Celui-ci pourrait se substituer au positionnement initial aléatoire. Plusieurs alternatives ont été testées, mais pour l'instant aucune n'a apporté des résultats significatifs.

3.3.4 Evaluation des solutions

Au terme de l'exécution de l'algorithme sur une interface donnée, nous obtenons un nombre important (proportionnel à la taille de l'interface) de solutions candidates. Lorsque la taille du problème augmente, il devient très vite long de visionner toutes ces solutions afin d'en trouver une correcte, si bien sûr celle-ci se trouve parmi les résultats.

Il est donc intéressant d'ajouter à l'algorithme une routine évaluant les solutions. Même si l'évaluation est réalisée sommairement, ce serait un apport non-négligeable, autant pour le travail réalisé que pour la sélection des solutions, et donc la rapidité d'exécution.

Pour noter les solutions, nous avons besoin d'une série de critères d'esthétisme de l'interface. Il en existe beaucoup, d'utilités variables par rapport aux hypothèses et au but recherché ici, j'ai donc choisi de me limiter aux suivants :

- **Aire superposée totale** :
  comme précisé auparavant, dans cette méthode, il est permis aux éléments de se superposer. Ce cas de figure est même inévitable lorsque la charge en 'widgets' est trop importante pour la zone d'affichage.
  Cependant, la surcharge d'une fenêtre, et plus particulièrement la superposition des éléments, même si bien gérée par l'interface graphique, limite l'accès à l'information et décroit l'efficacité de l'utilisation de l'interface.
  Pour deux instances de la même interfaces (à charge égale donc), on effectue donc la somme totale de l'aire occupée par des éléments et 'cachée' par superposition. La note de l'instance est ensuite calculée selon la formule suivante :

  $$Area = \left(1 - \min\left(\frac{A}{area_{ref}}, 1\right)\right)^3$$

  où :
  - A est l'aire superposée totale ;

– $area_{ref}$ est l'aire de référence, prise par hypothèse comme la somme des aires de tous les éléments de l'interface.

J'ai choisi, après plusieurs tests d'efficacité du critère, d'élever le membre de droite à la puissance 3 afin d'accroitre le poids de la superposition dans la note totale de l'interface. En effet, il est beaucoup plus dérangeant d'avoir une interface avec des éléments cachés derrière des autres que d'avoir une interface qui n'est pas parfaitement balancée.

La note est située entre 0 et 1. L'instance qui possède la valeur la plus élevée sera préférée.

– **Symétrie 'pondérale', ou balance, horizontale** :
la balance d'une interface est un critère fréquamment évoqué. Si l'on considère que chaque élément de l'interface possède un poids (une densité de masse par extension), la balance consiste à égaler la somme des poids sur chaque partie de l'interface (haut et bas, gauche et droite, pris par paires comme indiqué). Par exemple, si un élément est placé dans la moitié gauche de l'interface, nous nous trouvons dans un état d'instabilité. Il faut donc ajouter un ou plusieurs éléments dans la moitié droite de l'interface pour atteindre la balance. Attention cependant, la balance n'implique pas la symétrie visuelle.
La balance horizontale est cotée selon la formule suivantet :

$$Sym_h = 2 * \frac{min(S_h, area_{ref} - S_h)}{area_{ref}}$$

où :
– S est le poids total dans une des moitiés (gauche ou droite) de l'interface ;
– $area_{ref}$ est de nouveau l'aire de référence (somme des aires de tous les éléments de l'interface).

La symétrie verticale est évaluée de manière similaire. Les deux notes sont également situées entre 0 et 1. L'instance possédant la valeur la plus élevée sera la plus 'balancée'.

La note finale attribuée à une instance de l'interface est le produit de toutes les notes suscitées. Ce produit se situe également entre 0 et 1, une note plus élevée signifiant un meilleur respect global des critères.

3.4 Pseudocode final

Le pseudocode suivant reprend toutes les étapes de l'algorithme après améliorations, du pré- au post-traitement.

Algorithm 5 Méthode à forces dirigées appliquée aux interfaces : SPRING_FULL(I)

```
Répartir les éléments de I en s fenêtre(s)
for j = 1 to s do
  for k = 1 to M do
    Placement des éléments de I(s) à des positions aléatoires, en 2 dimensions
    (on néglige les dimensions des éléments pour l'instant)
    while les déplacement des noeuds sont supérieurs à un seuil ε_2D do
      Calculer la résultante des forces en chaque noeud i
      Calculer les déplacements D_i = c_4·(résultante en ce noeud)
      if un noeud i sort de la zone d'affichage then
        Réduire son déplacement D_i au maximum possible à l'intérieur de
        la fenêtre
      end if
      Bouger chaque noeud de D_i
    end while
    Attribution des positions aléatoires selon l'axe z
    (les éléments sont considérés avec leurs dimensions)
    while les déplacement des éléments sont supérieurs à un seuil ε_3D do
      Calculer la résultante des forces en le centre de gravité de chaque
      élément i
      Calculer les déplacements D_i = c_4·(résultante en ce centre de gravité)
      if un élément i sort de la zone d'affichage then
        Réduire son déplacement D_i au maximum possible à l'intérieur de
        la fenêtre
      end if
      Bouger chaque élément de D_i
    end while
  end for
  On évalue la note de chacune des M instances de la fenêtre s, et on choisit
  la meilleure
end for
Afficher les s fenêtre(s) de l'interface I
```

Le code source de toutes les fonctions relatives à la méthode SPRING est disponible en annexe.

4 La méthode Bottom-Right

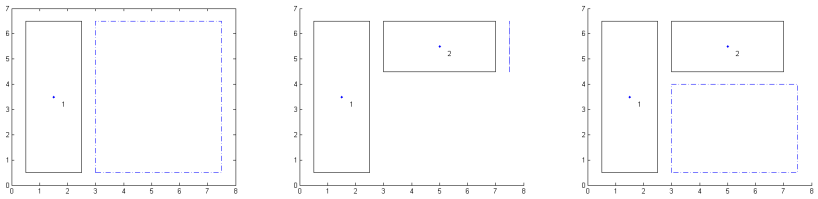
Nous avons décrit dans la section précédente la première méthode étudiée dans le cadre de ce travail. Cette section-ci est consacrée à la seconde, la méthode Bottom-Right. Il s'agit d'une méthode assez simple, développée vers la fin des années '90. Etant le plus simple des algorithmes étudiés ici, Bottom-Right servira de point de référence pour la comparaison avec les autres méthodes.

Je ferai référence à cet algorithme sous le nom BOTTOM-RIGHT, ou BR (et dérivés) en abrégé.

4.1 La méthode de base

La méthode Bottom-Right (Bas-Droite, littéralement) est un algorithme glouton de placement des éléments dans une interface, traitant le problème comme une instance du problème 'Knapsack'.

L'algorithme commence par trier les éléments selon un critère donné, l'aire dans ce cas-ci. Ensuite, les éléments étant pris dans l'ordre, le premier qui rentre dans la zone disponible est placé en haut à gauche de celle-ci. Il est ensuite retiré de la liste d'éléments à traiter. Récursivement, l'algorithme va effectuer le même raisonnement avec les zones à droite et en-dessous de l'élément nouvellement placé (voir **Fig. 19**), et ainsi jusqu'à ce qu'il n'y ait plus d'éléments à positionner.



(a) (b) (c)

FIG. 19: Procédé de placement de BR : **(a)** le premier élément placé, l'algorithme examine la zone à droite de celui-ci ; **(b)** le second élément placé, l'algorithme se rend compte qu'il n'y a pas de place à droite et **(c)** s'attaque alors à la zone en-dessous de ce dernier

Si jamais, lors de l'itération, plus aucun 'widget' ne peut être placé dans la fenêtre courante, l'algorithme crée une nouvelle fenêtre pour l'interface, et recommence l'itération sur la liste d'éléments restants.

4.1.1 Problèmes rencontrés et améliorations

Plusieurs problèmes apparaissent dès les premiers tests :

– les représentations d'interfaces obtenues à l'aide de l'algorithme BR ne sont pas balancées ;

– l'algorithme ne respecte pas les relations d'adjacence entre éléments ;
– l'algorithme ne tient pas compte des relations de groupement.

4.1.1.1 Déséquilibre dans la balance

L'algorithme BR commence par remplir les fenêtres de l'interface par la gauche et par le haut. Dès lors, on peut s'attendre à ce que la moitié de gauche soit plus chargée que la moitié de droite, et de même pour les moitiés supérieures et inférieures, comme illustré sur la figure **20**. Dès lors, en reprenant les critères de l'évaluateur de la fonction SPRING, les interfaces produites par cet algorithme auront tendance à obtenir un mauvais score en balance.

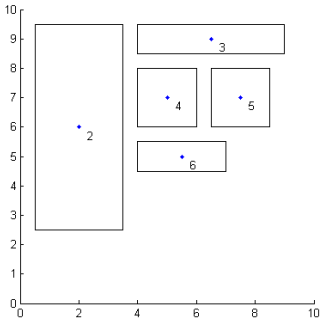


FIG. 20: Layout mal balancé calculé par BR

Malheureusement, ce problème est inhérent à la conception même de l'algorithme, et aucune modification mineure n'apportera une solution viable au problème. C'est un des défauts majeurs de l'algorithme BR.

4.1.1.2 Relations d'adjacence et de groupe

En plaçant un à un les éléments par ordre décroissant de taille, l'algorithme BR viole les contraintes d'adjacence et de groupe entre éléments. Dès lors, des éléments devant être placés côte à côte, pour des raisons de compréhension, peuvent se retrouver à deux extrémités d'une fenêtre, voire sur des fenêtres différentes. L'introduction des contraintes de groupement dans l'algorithme BR donne naissance à un autre algorithme, nommé Bottom-Right par Groupe (BRG), décrit plus loin dans cette section.

Le non-respect des contraintes d'adjacence, bien que moins gènant que les contraintes de groupement, reste un problème majeur. Changer le tri des éléments en fonction d'un critère combinant l'aire et les liens d'adjacence pourrait permettre d'augmenter le taux de respect de ces contraintes. Cependant, cette solution n'a pas été examinée dans le cadre de ce travail, et figure donc dans la liste des améliorations à apporter.

4.1.2 Pseudocode final

Algorithm 6 Algorithme Bottom-Right (BottomRight)

N : ensemble des éléments non-traités
I : interface et toutes les informations concernant les éléments
dim : matrice des dimensions des éléments
$Z \leftarrow I(1)$: zone considérée pour le placement des éléments
Tri des éléments de N par ordre de taille
$s \leftarrow 1$
while $N \neq \emptyset$ **do**
  $I(s) \leftarrow N(1)$ (placé en haut à gauche)
  $R \leftarrow$ la zone à droite de $N(1)$
  $B \leftarrow$ la zone en-dessous de $N(1)$
  $N \leftarrow N \backslash N(1)$

  $[N, I] \leftarrow [N', I'] = BR(N, I, R, s)$ (appel de la méthode de placement de BR sur la zone R avec comme liste d'éléments N)
  $[N, I] \leftarrow [N', I'] = BR(N, I, B, s)$

  $s \leftarrow s + 1$
end while
Renvoyer I

Algorithm 7 Méthode de placement de BottomRight (BR)

N : ensemble des éléments non-traités, triés par taille
I : interface et toutes les informations concernant les éléments
Z : zone considérée pour le placement
s : index de la fenêtre considérée
if un élément de N peut encore être placé dans Z **then**
  $I(s) \leftarrow N(1)$ (placé en haut à gauche de Z)
  $R \leftarrow$ la zone de Z à droite de $N(1)$
  $B \leftarrow$ la zone de Z en-dessous de $N(1)$
  $N \leftarrow N \backslash N(1)$

  $[N, I] \leftarrow [N', I'] = BR_{place}(N, I, R, s)$ (appel de BR_{place} sur la zone R avec comme liste d'éléments N)
  $[N, I] \leftarrow [N', I'] = BR_{place}(N, I, B, s)$
end if
Renvoyer $[N, I]$

4.2 Algorithme Bottom-Right par groupe

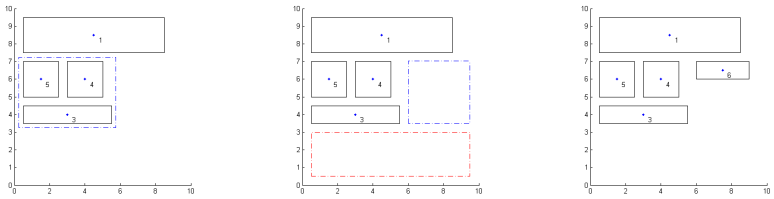
L'algorithme Bottom-Right par groupe est une amélioration de l'algorithme BR, qui, comme son nom le suggère, intègre les contraintes de groupement d'éléments dans son fonctionnement. Il s'agit d'un apport significatif car cela résout l'un des problèmes majeurs de l'algorithme BR.

Nous utiliserons l'abréviation BRG pour faire référence à cet algorithme.

4.2.1 Les bases

L'algorithme utilise le même principe que la méthode BR pour placer les divers éléments de l'interface. L'intégration des contraintes de groupement se fait au niveau de la sélection des éléments à placer : celle-ci ne se fait plus élément par élément mais bien groupe par groupe.

Tout d'abord, l'algorithme fait appel à la fonction de séparation de la méthode SPRING (SPRING_SPLIT), pour diviser l'ensemble des éléments en groupes élémentaires. L'algorithme va ensuite choisir le groupe le plus grand, et le positionner dans la surface la plus petite possible de la fenêtre. Ensuite, ce groupe sera considéré comme un seul gros élément, et l'algorithme tentera récursivement de placer d'autres groupes à droite et en-dessous de celui nouvellement placé, en suivant le même principe que la méthode BR. La figure **(21)** illustre ces étapes.



(a) (b) (c)

FIG. 21: Procédé de placement de BRG : **(a)** Le groupe nouvellement placé (3-4-5) est assimilé au rectangle en pointillés ; **(b)** l'algorithme explore ensuite les possibilités de placement à droite et en-dessous ; **(c)** prochain placement

Le placement des éléments se fait en utilisant la fonction de placement de BR (BR_{place}), utilisée sur toutes les permutations de l'ordre de placement des éléments. La représentation occupant la surface la plus petite est sélectionnée, et le conteneur équivalent est alors le plus petit rectangle englobant tous les éléments. C'est ce dernier, et non plus le détail des placements, qui sera pris en compte par la suite dans l'algorithme.

4.2.2 Problèmes et améliorations

De nouveau, après quelques tests, on s'aperçoit que l'algorithme est sujet à quelques défauts, hérités de la méthode BR :

– comme son proche parent, les représentations d'interfaces obtenues à l'aide de BRG ne sont pas balancées ;
– l'algorithme ne respecte pas les relations d'adjacence entre éléments ;

Les contraintes de groupement sont par contre dans l'ensemble respectées, le but de l'amélioration est donc atteint.

4.2.2.1 Déséquilibre dans la balance

Tant dans le placement des groupes dans la fenêtre que dans le placement des éléments à l'intérieur d'un seul groupe, la méthode utilisée a tendance à surcharger les zones haut et gauche de la fenêtre. Comme pour l'algorithme BR, ce problème n'est pas résolvable sans changer la méthode de placement, et reste un des défauts majeurs de l'algorithme BRG.

4.2.2.2 Relations d'adjacence

Héritant ce défaut de son parent, BRG viole dans certains cas les contraintes d'adjacence pour des raisons de dimensions d'éléments ou de groupes. Le même type de solution que pour BR, à savoir changer le tri des éléments en fonction d'un critère combinant l'aire et les liens d'adjacence pourrait minimiser l'impact de ce problème. La solution, non étudiée dans le cadre de cette étude, figure dans les améliorations à envisager.

4.2.3 Pseudocode final

Algorithm 8 Algorithme Bottom-Right (BottomRight_G)

I : interface et toutes les informations concernant les éléments
$Z \leftarrow I(1)$: zone considérée pour le placement des éléments

$G \leftarrow$ séparation des éléments en groupes élémentaires
$g \leftarrow$ groupe ayant le plus de lien avec le bord supérieur
$s \leftarrow 1$

$[G, I] \leftarrow [G', I'] = BRG_{place}(g, I, I(s), s)$ (placement du premier groupe)
$R \leftarrow$ zone à droite de g
$B \leftarrow$ zone en-dessous de g et R

$[G, I] \leftarrow [N', I'] = BRG_{main}(G, I, R, s)$ (remplissage à droite)
$[G, I] \leftarrow [N', I'] = BRG_{main}(G, I, B, s)$ (remplissage en bas)

while $G \neq \emptyset$ **do**
  $s \leftarrow s + 1$

  $[G, I] \leftarrow [G', I'] = BRG_{main}(G, I, I(s), s)$ (remplissage des fenêtres suivantes)
end while
Renvoyer I

Algorithm 9 Sous-fonction récursive principale (BRG_{main})

G : groupes non-placés
I : interface et toutes les informations concernant les éléments
Z : zone considérée pour le placement
s : numéro de la fenêtre à modifier

$[G, I] \leftarrow [G', I'] = BRG(G, I, Z, s)$ (appel de la méthode de sélection)
$R \leftarrow$ zone à droite de g
$B \leftarrow$ zone en-dessous de g et R

$[G, I] \leftarrow [N', I'] = BRG_{main}(G, I, R, s)$ (remplissage à droite)
$[G, I] \leftarrow [N', I'] = BRG_{main}(G, I, B, s)$ (remplissage en bas)

Renvoyer $[G, I]$

Algorithm 10 Fonction de sélection (BRG)

G : groupes non-placés
I : interface et toutes les informations concernant les éléments
Z : zone considérée pour le placement
s : numéro de la fenêtre à modifier

$G \leftarrow$ tri des éléments de G par aire, en ordre décroissant
$k \leftarrow 1$

while aucun groupe de G n'a pas été placé avec succès **and** $k \leq$ longueur de G **do**
  $g \leftarrow G(k)$
  $[g', I'] \leftarrow BRG_{place}(g, I, Z, s)$
  $k \leftarrow k + 1$
end while

if un groupe a été placé avec succès **then**
  Renvoyer $[G \backslash g, I']$
else
  Renvoyer $[G, I]$
end if

Algorithm 11 Fonction de placement (BRG_{place})

g : groupe à placer
I : interface et toutes les informations concernant les éléments
Z : zone considérée pour le placement
s : numéro de la fenêtre à modifier

$M \leftarrow$ toutes les permutations possibles des éléments de g
$a = \infty$
J est le placement des éléments retenu

for $k = 1 \rightarrow$ longueur de M **do**
  $g \leftarrow G(k)$
  $[g', I'] \leftarrow BR(g, I, Z, s)$

  if $g' = \emptyset$ **and** $a >$ aire utilisée par g **then**
    $a \leftarrow$ aire utilisée par g
    $J \leftarrow I'$
  end if

end for

if aucun placement n'a été retenu **then**
  Renvoyer $[g, I]$
else
  Renvoyer $[\emptyset, J]$
end if

5 Layout appropriateness (LA)

La section précédente était consacrée à l'algorithme Bottom-Right et son dérivé, l'algorithme Bottom-Right par groupe. Nous allons maintenant dans cette section nous intéresser à une quatrième et dernière méthode, le Layout Appropriateness. Il s'agit d'un algorithme déjà existant, réimplémenté et légèrement amélioré sur MATLAB pour les besoins de ce travail.

Cet algorithme est basé sur la mesure du même nom, qu'il utilise pour évaluer la qualité d'une interface. Cette mesure nécessite une description des séquences d'actions (au niveau des éléments) effectuées par l'utilisateur, et la fréquence de ces dites séquences. Ces données peuvent être obtenues en observant un système déjà existant, ou en effectuant une analyse simplifiée d'une interface. La note d'une interface ('appropriateness') est ensuite calculée en attribuant un coût à chaque séquence et en pondérant le tout par la fréquence d'utilisation.

La méthode n'utilise pas la même base pour représenter les interfaces, néanmoins il est possible de transformer les descriptions utilisées dans ce travail pour en tirer les séquences d'actions et les fréquences nécessaires. La comparaison peut donc être effectuée. De plus, la nouvelle métrique introduite peut être utilisée pour évaluer sous une autre perspective les interfaces produites dans le travail.

5.1 La mesure

Le calcul du Layout Appropriateness nécessite plusieurs données :

- l'ensemble des éléments utilisés ;
- les séquences effectuées par les utilisateurs ;
- les fréquences de chacune de ces séquences.

Tout cela peut être résumé en une matrice dite 'de transfert', où les probabilités de passage d'un noeud à l'autre sont encodées : $0 \leq F(i,j) \leq 1$ où i est la probabilité de passer en i depuis j. Notons que pour une même colonne de cette matrice, la somme des éléments ne peut dépasser 1. Elle peut cependant être strictement inférieure à 1 puisque certains noeuds peuvent être terminaux, c'est-à-dire à la fin de la séquence d'actions.

Le coût d'une séquence d'actions est ici relié à la distance parcourue sur l'écran par l'utilisateur, selon la loi de Fitts. D'autres fonctions de coût peuvent être envisagées, mais le point ne sera pas exploré dans ce travail.

Le coût d'une représentation est alors calculé à l'aide des coûts des séquences pondérés par la fréquence d'utilisation de la séquence.

$$Cost = \sum_{\forall(i,j)} F(i,j) \cdot C(i,j)$$

où F est la matrice des probabilités de transition, et C la matrice des coûts. Notons que minimiser cette fonction de coût n'équivaut pas à minimiser le coût

de la séquence d'actions la plus utilisée, mais bien la moyenne (pondérée en fonction des fréquences) des séquences d'action.

Pour calculer le LA d'une représentation quelconque, il faut d'abord trouver une représentation LA-optimale. Ceci est possible à l'aide de l'algorithme décrit au point suivant. La note d'une interface est ensuite calculée selon la formule suivante :

$$LA = 100 * \frac{\text{coût du layout LA-optimal}}{\text{coût du layout évalué}}$$

5.2 L'algorithme

Plusieurs simplifications ont été faites pour implémenter cet algorithme :

- tous les 'widgets' sont de taille unitaire ;
- les distances parcourues sont évaluées centre à centre ;
- les éléments sont positionnés dans une grille rectangulaire, de taille spécifiée au départ ;
- les cellules de la grille doivent être de même taille.

Des contraintes supplémentaires sont aussi ajoutées : les éléments peuvent être soit placés à un point bien précis de la grille, soit directement à droite ou en-dessous d'un autre élément. Les deux dernières contraintes peuvent être utilisées pour transformer un élément plus large en blocs unitaires.

L'algorithme choisi est une recherche exhaustive de toutes les possibilités de placement sur la grille. Cependant, le nombre de configurations à examiner explose rapidement lorsque la taille de la grille ou le nombre d'éléments augmente :

$$\text{Nombre de possibilités} = \frac{N!}{K! \cdot (N-K)!} \cdot K!$$

où K est le nombre d'emplacements sur la grille, et N le nombre d'éléments à placer.

Nous allons donc utiliser un algorithme de type Branch-and-Bound pour réduire le nombre d'itérations de la méthode. L'algorithme va prendre en paramètres un point de départ pour les séquences d'actions, et va explorer les différentes possibilités de placement, construisant ainsi un arbre, mais en tenant compte des points suivants :

- L'ordre de placement des éléments est déterminé par l'algorithme : on choisit à tout instant de placer l'élément qui est le point d'arrivée le plus probable si l'on part de tous les éléments déjà placés sur l'interface, point de départ compris. Le premier élément à être placé sera donc celui vers lequel le point de départ a la plus grande probabilité de transfert.
- On calcule pour chaque représentation (partielle ou non) une borne inférieure sur la note de l'interface. A chaque itération, l'interface explorée est celle

dont la borne inférieure est la plus basse. A bornes égales, on choisit celle ayant le plus d'élément déjà placés.
- Une borne supérieure, calculée comme étant la note d'une représentation de l'interface obtenue par un algorithme plus simple (par exemple BR), est utilisée pour écarter les layouts partiels dont le coût est trop élevé.

Lorsqu'une solution où tous les 'widgets' sont placés a un coût inférieur ou égal à toutes les autres solutions (partielles ou non) de l'arbre, elle est considérée comme LA-optimale.

5.3 Application au problème donné

Pour pouvoir comparer cet algorithme avec ceux décrits précédemment, il faut pouvoir passer de la représentation par matrice d'adjacence à celle par matrice de transition.

L'**algorithme 12** permet de séparer les 'widgets' en éléments unitaires, et construit la représentation par transitions correspondant à la matrice d'adjacence donnée.

5.4 Problèmes et améliorations

Outre le problème de conversion traité au point précédent, la méthode LA présente quelques points négatifs :
- les contraintes d'adjacence et de groupement ne sont pas toujours respectées ;
- il est impossible de prendre en compte l'espacement des éléments.

5.4.1 Contraintes d'adjacence et de groupement

Le respect de ces contraintes est un problème inhérent à la conception de l'algorithme.

En effet, la méthode choisit la représentation qui minimise son critère de Layout Appropriateness. Les considérations qui suggèrent au développeur de placer deux éléments côte à côte peuvent ne pas être reprises dans le critère LA, et la contrainte sera donc ignorée.

On pourrait envisager de remédier à cela en utilisant la contrainte 'dure' de positionnement (imposer le placement directement à droite ou en-dessous), mais ce serait écarter des solutions qui pourraient être meilleures selon le critère LA.

Une autre alternative, explorée ici, est de donner un bonus à la fréquence des objets groupés lors du passage de la matrice d'adjacence vers la matrice de transition.

Algorithm 12 Algorithme de conversion des données

n : nombre d'éléments de l'interface
A : matrice d'adjacence des éléments
dim : matrice des dimensions des éléments
c : paramètre pour les liens de groupement

F : matrice des transitions
B : vecteur des contraintes 'en-dessous'
R : vecteur des contraintes 'à droite'

for $i = 1 \rightarrow n$ **do**
  $F(:, i) \leftarrow A(:, i)$

  pour tous les j $(0 \leq j \leq n)$:
  if A(j,i) = A(i,j) = 1 **then**
    $F(j, i) \leftarrow F(j, i) + c \cdot F(j, i)$ (mise en exergue des relations de groupe)
  end if

  $s \leftarrow \sum_{\forall j} F(j, i)$
  $F(:, i) \leftarrow \frac{1}{s} \cdot F(:, i)$
end for

$k \leftarrow 0$ (compteur des éléments rajoutés)
for $i = 1 \rightarrow n$ **do**
  if $dim(i, 1) > 1$ **then**
    Division du widget en éléments de largeur unitaire
    Assignement des contraintes de proximité dans B et R
  end if
  if $dim(i, 2) > 1$ **then**
    Division de chaque tranche verticale (de largeur unitaire) en éléments de hauteur unitaires
    Assignement des contraintes de proximité dans B et R
  end if

  $d \leftarrow \lceil dim(i, 1) \rceil \cdot \lceil dim(i, 2) \rceil$
  $L \leftarrow 1/d \cdot F(i, :)$
  $C \leftarrow 1/d \cdot F(:, i)$
  $F(i, :) \leftarrow L$
  $F(:, i) \leftarrow C$

  $FR \leftarrow C \cdot \mathbf{1}^{1 \times (d-1)}$
  $FB \leftarrow \mathbf{1}^{1 \times (d-1)} \cdot L$
  $F \leftarrow \begin{bmatrix} F & FR \\ FB & \mathbf{0}^{(d-1) \times (d-1)} \end{bmatrix}$
end for

Renvoyer $[F; B; R]$

5.4.2 Espacement des éléments

Tous les algorithmes testés jusqu'à présent offraient la possibilité de spécifier une distance d'écartement entre les différents éléments d'une interface. Ce n'est pas le cas de la méthode LA. Il faut pour cela inclure une petite routine déterminant, sur base de la distance d'écartement, si un placement d'élément est valide : étant donné une distance d'écartement $dist \in \mathcal{N}$, la méthode LAX_isValid va vérifier si les $dist$ premiers emplacements sur la grille dans toutes les directions autour du nouvel élément sont bien libres. Si c'est le cas, le placement sera autorisé ; dans le cas contraire, celui-ci est purement et simplement ignoré (**Fig. 22**).

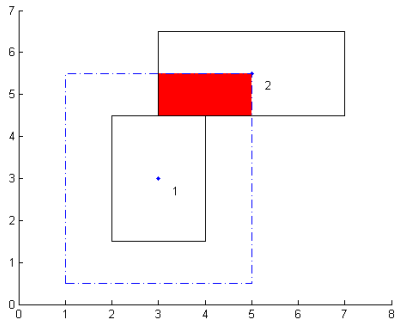


FIG. 22: LA - Exemple d'un placement invalide ($dist = 1$)

5.5 Pseudocode final

Les peudocodes **Algorithme 13** et **14** reprennent les fonctions principales de la méthode de Layout Appropriateness.

Algorithm 13 Algorithme de Layout Appropriateness

n : nombre d'éléments de l'interface
F : matrice des transitions
B : vecteur des contraintes 'en-dessous'
R : vecteur des contraintes 'à droite'
S : vecteur de position du point de départ sur la grille
$height, width$: dimensions de l'interface
$dist$: distance d'écartement des éléments

$C \leftarrow$ blocs liés à un autre par une contrainte de proximité
$NC \leftarrow$ tous les autres blocs
M : arbre exploré par l'algorithme
$Val \leftarrow \infty$: vecteur des estimateurs en chaque noeud

$T \leftarrow$ seuil calculé par une méthode plus simple (BR)
$o \leftarrow 0$

while $o = 0$ **do**
  $i, Frame \leftarrow$ index et représentation du noeud le plus intéressant à explorer (valeur minimale)

  if $Frame$ ne contient pas tous les noeuds **then**
    $[Val; M; C; NC] \leftarrow LAX(n, B, R, F, height, width, Val, M, i, C, NC, T, S, dist)$;
  else
    $o \leftarrow i$
  end if
end while
Renvoyer $Val(o)$ et $M(o)$

Algorithm 14 Sous-fonction d'exploration de LA (LAX)

n : nombre d'éléments de l'interface
F : matrice des transitions
B : vecteur des contraintes 'en-dessous'
R : vecteur des contraintes 'à droite'
S : vecteur de position du point de départ sur la grille
$height, width$: dimensions de l'interface
C : blocs contraints non-placés en chaque noeud
NC : blocs non-contraints non-placés en chaque noeud
M : matrice représentant l'arbre exploré par l'algorithme
Val : vecteur des estimateurs en chaque noeud
i : index du noeud exploré
T : seuil d'élimination des noeuds
$dist$: distance d'écartement des éléments

$Frame, RemC, RemNC \leftarrow$ informations relatives au noeud i

$node \leftarrow$ élément de $RemNC$ ayant le plus de liens avec ceux déjà placés

for toutes les positions j non-occupées de $Frame$ **do**
  Dans $Frame$ **:**
  Placement de $node$ en j
  Placement itératif de tous les noeuds de $RemC$ liés à j (même indirectement)

  if Tous les noeuds liés à j ont pu être placés **and** le placement est valide (écartement de $dist$) **then**
    $V \leftarrow$ estimation du coût de $Frame$
    if $V \leq T$ **then**
      $M \leftarrow [M; Frame]$
      $Val \leftarrow [Val; V]$
      $RemC1 \leftarrow$ noeuds contraints non-placés sur $Frame$
      $RemNC1 \leftarrow$ noeuds non-contraints non-placés sur $Frame$
      $C \leftarrow [C; RemC1]$
      $NC \leftarrow [NC; RemNC1]$
    end if
  end if
end for
Renvoyer $[Val; M; C; NC]$

6 Tests et analyse

Nous avons abordé dans la section précédente la dernière méthode à comparer, le Layout Appropriateness. Nous allons maintenant exposer les tests effectués, leurs résultats et les analyses effectuées sur base de ceux-ci. Les points forts et les points faibles des différentes approches seront mis en lumière, et la section se terminera sur les améliorations à apporter pour corriger ces derniers.

6.1 Critères de comparaison

La comparaison des différents algorithmes a été effectuée selon plusieurs axes. Les principaux points ayant retenu notre attention sont décrits ci-dessous.

6.1.1 Complexités temporelle et spatiales

La **complexité temporelle** d'un algorithme est le nombre d'opérations élémentaires effectuées par celui-ci. Ce nombre s'exprime en fonction de la taille n des données.

La **complexité spatiale** d'un algorithme est l'homologue de la complexité temporelle dans le monde plus physique de la mémoire nécessaire à l'exécution d'un algorithme. Elle évalue, toujours en fonction de la taille n des données, le nombre d'éléments à stocker lors d'une exécution de la méthode.

Par exemple, pour un algorithme nécéssitant un nombre d'opérations caractérisé par une fonction linéaire en n, nous dirons que sa complexité est linéaire, d'ordre n, ou encore, en abrégé, $\mathcal{O}(n)$. La complexité spatiale s'exprime avec la même notation que la complexité temporelle.

Ces deux critères permettent d'évaluer le temps et les ressources mémoires consommés par un algorithme pour s'exécuter, tout en restant indépendant des détails techniques liés à la machine, à l'utilisation des ressources processeur,... Elle permet également de donner des critères d'évaluation à des algorithmes qui ne sont pas implémentés physiquement sur une machine, ce qui permet de les comparer du point de vue théorique.

6.1.2 Temps d'exécution et ressources processeur

Afin de donner une idée plus précise et plus physique de la complexité temporelle des différents algorithmes, le temps d'exécution de chacune des méthodes sera précisé. Ceci afin d'évaluer le poids de l'algorithme sur le système, et de déterminer à quel type d'usage celui-ci peut être destiné.

Tous les tests sont effectués sur la même machine, afin que les données restent comparables. Voici les caractéristiques de l'ordinateur utilisé :

- **Carte mère :** ASUS P8P67
- **Processeur :** INTEL Core i7-2600
- **RAM :** 8GB

– **Système d'exploitation :** Windows 7 Professionel
– **Version de MATLAB :** 7.0.1

6.1.3 Qualité de l'interface

La qualité de l'interface est évaluée à l'aide des différents critères existants, en particulier ceux repris dans l'évaluateur de la méthode SPRING et le critère de Layout Appropriateness. Le respect des premiers n'impliquant pas nécessairement le respect du dernier, nous garderons les deux critères séparés lors de l'analyse.

De plus, si certains défauts évidents apparaissent sur les résultats obtenus, ceux-ci seront mentionnés même s'ils ne relèvent pas directement des critères suscités.

Juger la qualité des résultats, en parallèle avec le temps et les ressources nécessaires pour les obtenir, permet d'obtenir un point de comparaison solide pour les différents algorithmes.

6.2 Complexité

Le tableau suivant reprend les complexités des différents algorithmes examinés :

	temporelle	spatiale
BR	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
BRG	$\mathcal{O}(n^{n+1} * width)$	$\mathcal{O}(n^n)$
LA	$\mathcal{O}(n^n)$	$\mathcal{O}(n^n \cdot width \cdot height)$
SPRING	$\mathcal{O}(i * n^2)$	$\mathcal{O}(n^2)$

où n est le nombre d'éléments, *width* et *height* les dimensions de la fenêtre d'affichage et i le nombre d'itérations requis pour arriver à une solution pour l'algorithme SPRING.

6.2.1 Bottom-Right

La méthode de placement de Bottom-Right s'effectue en temps $\mathcal{O}(n)$, et s'appelle elle-même deux fois. Dans le pire des cas, aucun de ces appels subordonnés n'aboutit à un placement, et Bottom-Right doit itérer n fois sur sa méthode de placement, ce qui conduit à une complexité $\mathcal{O}(n^2)$.

La complexité spatiale en $\mathcal{O}(n)$ s'explique par les matrice de positions ($n \times 2$), dimensions ($n \times 2$),... stockées par l'algorithme.

6.2.2 Bottom-Right par groupe

Le pire cas pouvant arriver est le suivant : tous les n éléments à placer se trouvent dans un seul groupe. La méthode de placement de BRG va donc effectuer toutes les permutations possibles (au nombre de $n!$) de l'ordre des

éléments, et les stocker dans une matrice, d'où la complexité spatiale en $\mathcal{O}(n^n)$. Ensuite, la méthode de placement appelle une sous-routine de complexité $\mathcal{O}(n)$ sur chacune de ces permutations, en tentant de les placer dans des fenêtres de largeur différentes. Il y aura donc en tout $\mathcal{O}(n^n \cdot width)$ appels de complexité $\mathcal{O}(n)$. Bien entendu, ce cas de figure est extrêmement rare, et la complexité moyenne est beaucoup plus faible.

Notons que le facteur faisant exploser la complexité est la taille des groupes. En gardant des groupes élémentaires de taille raisonnable, la complexité peut être tenue dans des valeurs tout à fait acceptables : limiter les groupes à k éléments résulterait en une complexité temporelle $\mathcal{O}(\frac{n^2}{k} \log n + width \cdot k^{k+1})$. La complexité spatiale pourrait être réduite en utilisant une autre fonction tirant aléatoirement une permutation des éléments considérés. Cependant, cela enlèverait la certitude d'avoir trouver le bon placement. Etant donné que les exemples ne contenaient pas de groupes de taille importante, nous avons préféré conserver cette certitude pour les tests.

6.2.3 Layout Appropriateness

De nouveau, examinons le cas où des éléments de taille unitaire doivent être placés sur une grille de taille $width \times height$. Avec autant d'éléments qu'il y a d'emplacements sur la grille ($n = width \times height$), et ceux-ci n'étant reliés par aucune contrainte de placement 'en bas de...' et 'à droite de...', nous sommes en présence d'un problème où n éléments doivent être placés dans n emplacements, de manière à minimiser une fonction de coût donnée. Le pire cas est bien évidemment atteint lorsque toutes les $n!$ possibilités doivent être explorées pour trouver l'optimum, ce qui conduit à une complexité en $\mathcal{O}(n^n)$. Etant donné que toutes les assignations sur la grille sont gardées en mémoire, la complexité spatiale de pire cas est bien $\mathcal{O}(n^n \cdot width \cdot height)$.

Comme pour la méthode BRG, ce cas est rare, et la complexité moyenne de l'algorithme est bien en-dessous de celle de pire cas. Le facteur déterminant dans la complexité est le nombre de cases disponibles pour un placement, qui est en fait lié à la taille de la grille et à la taille des différents éléments. Plus les éléments sont petits et plus la grille est grande, plus le nombre de cases disponibles est élevé, et plus nombreuses sont les possibilités devant être explorées.

6.2.4 Forces dirigées

A chaque itération de SPRING, une évaluation des forces du système est nécessaire. La complexité de l'évaluation est de $\mathcal{O}(n^2)$. Le nombre d'itérations i est une fonction inconnue et difficile à caractériser des différents paramètres de l'interface : nombre d'éléments, dimensions, relations d'adjacence.

- Un nombre d'éléments élevé dans une fenêtre donnée augmentera le nombre d'itérations nécessaires ;
- Des éléments allongés, avec de grandes dimensions résulteront en un nombre plus élevé d'itérations, à cause du problème de croisement ;

– Une bonne description des relations d'adjacence peut réduire le nombre d'itérations à effectuer.

Ce nombre pourrait être grandement réduit si l'algorithme était associé à une méthode de prépositionnement efficace, qui respecterait les contraintes de groupement et d'adjacence. La complexité spatiale en $\mathcal{O}(n^2)$ est liée aux matrices stockées par l'algorithme, notamment la matrice d'adjacence $(n \times n)$.

6.3 Interface à 6 éléments

Cet exemple se base sur une interface simple à 6 éléments, placés dans une fenêtre de 10cm sur 10cm, ou plusieurs si besoin. Les matrices suivantes reprennent les caractéristiques de l'interface :

n	dimensions
1	8×2
2	3×6
3	5×1
4	2×2
5	2×2
6	3×1

et $A = \left[\begin{array}{cccccc|cccc} 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{array}\right]$ (2)

Rappelons que pour la matrice d'adjacence, l'élément $A(i, j) = 1$ signifie que i devrait être placé proche de j.

Sur base de ces données, nous pouvons commencer les tests. Les deux tableaux suivants regroupent les résultats obtenus :

	probabilité	temps	# itérations
BR	1	$0.01s$	1
BRG	1	$0.2s$	1
LA*	1	$16min$	1
LA	1	$4h30$	1
SPRING	0.011	$1s$	$n = 275 \Rightarrow p_{OK} \geq 0.95$

	LA	aire	balance H	balance V	total
BR	25.3228	1	0.88	0.76	0.6688
BRG	[23.3794; 4]	[1; 1]	[0.85; 0.47]	[0.77; 0.59]	[0.6545; 0.2773]
LA*	18.8743	1	0.62	0.82	0.5084
LA	21.2374	1	0.98	0.9	0.882
SPRING	23.6915	1	0.99	0.9089	0.8998

Du premier coup d'oeil, on observe que les méthodes gloutonnes (Bottom-Right et Bottom-Right par groupe) ont un temps d'exécution bien inférieur aux deux autres méthodes. La différence entre les lignes LA et LA* réside dans la prise en compte des distances inter-éléments : les résultats classifiés LA* ont été obtenus avec une distance $dist = 0$, tandis que les résultats de la ligne LA sont calculés pour $dist = 0.5$.

6.3.1 Bottom-Right

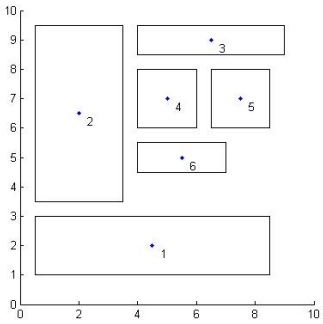


FIG. 23: Interface par la méthode BR

La méthode Bottom-Right, comme annoncé dans la section qui lui est consacrée, n'a pas tenu compte des relations d'adjacence pour le placement des éléments. L'élément 1, par exemple, censé se retrouver proche du bord supérieur de la fenêtre, a été placé tout en-dessous. Les relations de groupement sont dans ce cas-ci respectées, mais ce n'est que pure coincidence. Bottom-Right est la méthode plus rapide de toutes, mais son score de Layout Appropriateness est le plus élevé parmi les 4, et la balance verticale et horizontale laissent à désirer. Ce dernier point s'explique, comme montré sur la **Fig. 23**, par une surcharge des parties gauche et supérieure de l'interface.

6.3.2 Bottom-Right par groupe

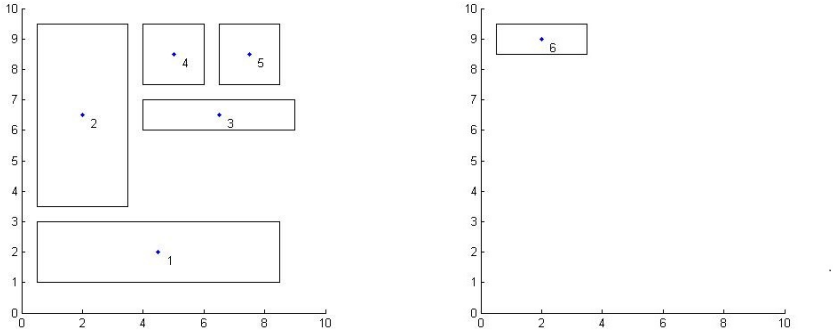


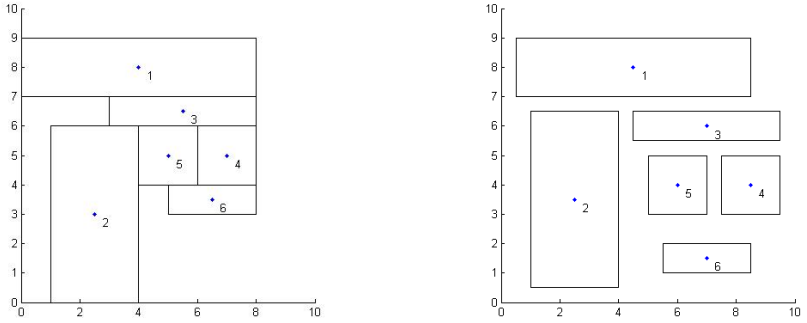
FIG. 24: Interface par la méthode BRG

La méthode Bottom-Right par groupe, quant à elle, a ici un résultat à deux

fenêtres (**Fig. 24**). Il est évident que la deuxième est totalement superflue, son unique élément pouvant aisément se placer dans l'espace vide à droite de la première, donnant alors exactement le même résultat la méthode BR. Ceci est dû à la méthode de placement des groupes : les éléments 1 à 5 formant un groupe, l'algorithme a d'abord commencé par tous les placer dans l'interface. Mais une fois que l'algorithme passé dans une zone plus basse de l'interface, il lui est impossible de remonter pour placer un autre élément dans une zone libre plus haut. Il est donc contraint de créer une nouvelle fenêtre pour le placement.

Le score combiné de Layout Appropriateness de la méthode BRG est donc plutôt élevé. Les scores de balance comparables à ceux de la méthode BR, et ce pour une bonne raison : les résultats sont presques identiques. La vitesse d'exécution est toujours très rapide, quoiqu'un peu plus lente que celle de la méthode BR.

6.3.3 Layout Appropriateness



(a) sans distance entre éléments

(b) avec une distance $dist = 0.5$

FIG. 25: Interface par la méthode LA

Tout d'abord, il faut remarquer que le temps d'exécution de LA avec distances inter-éléments est beaucoup plus élevé que si celles-ci sont négligées. En effet, *dist* étant plus petit que 1, il a fallu examiner le problème non pas sur base de blocs unitaires, mais de dimension 0.5×0.5. Ceci afin de pouvoir tenir compte des distances inter-éléments dans la grille en imposant des zones vides autour des différents 'widgets'. Ainsi, la taille de la grille explorée est passée de 10×10 à 20×20, ce qui équivaut à multiplier le nombre d'emplacements sur la grille, et donc de positions possibles, par 4. C'est la raison de la forte augmentation du temps de calcul. Afin de rester sur un pied d'égalité avec les tests effectués sur les autres méthodes, nous allons uniquement comparer les résultats où les distances inter-éléments sont respectées. Le seuil utilisé pour la méthode LA est celui obtenu par l'algorithme Bottom-Right.

L'interface produite par LA affiche un score de Layout Appropriateness plus bas que toutes les autres, ainsi qu'une bonne balance horizontale et verticale. Le prix à payer en temps de calcul est élevé, mais les résultats sont concluants. De plus, dans cet exemple-ci, les interfaces créées respectent relativement bien les relations d'adjacence et de groupement.

6.3.4 Forces dirigées

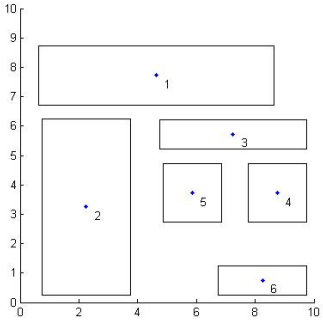


FIG. 26: Interface par la méthode SPRING

En ce qui concerne les résultats de la méthode SPRING, la première observation est que la fenêtre aurait pu être réduite à 10×9 sans impacter la qualité ni la densité de l'interface. SPRING représente comme espéré l'interface considérée avec une très bonne balance horizontale et verticale, le meilleur score parmi les 4 méthodes. Seul l'algorithme LA affiche un meilleur score en Layout Appropriateness. Cependant, le temps de calcul est le deuxième plus élevé, et la certitude d'aboutir à la bonne solution n'est que de 95%. Pour arriver à un niveau de certitude $p > 99\%$, il faut effectuer plus de 400 itérations, poussant le temps de calcul à environ 7 minutes.

6.3.5 Sensibilité du modèle

Un point important est à noter : la sensibilité de certains algorithmes au modèle sous-jacent des relations entre les éléments, défini par la matrice d'adjacence. Alors que Bottom-Right, qui ne se préoccupe pas du tout de ces relations, n'affichera aucun changement tant que les dimensions de la fenêtre ou des éléments ne sont pas modifiées, les autres algorithmes réagiront à certaines modifications de la matrice A.

Pour ce faire, nous avons altéré la matrice A de l'exemple (voir équation **3**) de la manière suivante :

$$A = \left[\begin{array}{cccccc|cccc} 0 & \mathbf{0} & \mathbf{0} & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & \mathbf{0} & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{array}\right]$$

Le groupe formé par les éléments 1 à 5 a été cassé en 3 groupes élémentaires. Les tableaux suivants regroupent les résultats de ce nouveau test

	probabilité	temps	# itérations
BR	1	$0.01s$	1
BRG	1	$0.2s$	1
LA	1	$4h30$	1
SPRING	0.04	$1s$	$n = 75 \Rightarrow p_{OK} \geq 0.95$

	LA	aire	balance H	balance V	total
BR	21.4656	1	0.88	0.76	0.6688
BRG	13.85	1	0.8	0.76	0.6080
LA	*N.D.*	*N.D.*	*N.D.*	*N.D.*	*N.D.*
SPRING	14.257	1	0.8519	0.9732	0.8291

L'amélioration des objectifs de Layout Appropriateness est à attribuer au changement des relations entre éléments, impliquant un changement de la matrice de transition. On observe cependant que l'algorithme Bottom-Right par groupe affiche cette fois l'interface en une seule fenêtre, qui possède un score LA plus faible que la méthode SPRING. Mais cette dernière lui reste toujours supérieure en matière de balance horizontale et verticale.

Un autre changement notable se situe au niveau des performances de SPRING : la probabilité d'obtenir un résultat a quasiment quadruplé, sur un simple changement de la matrice A, ce qui a grandement diminué le temps d'exécution total. C'est tout le contraire pour la méthode LA[2], dont le temps d'exécution a augmenté de plusieurs heures, malgré que le seuil d'exclusion des noeuds se soit grandement amélioré.

6.4 Interface à 13 éléments

	probabilité	temps	# itérations
BR	1	$0.05s$	1
BRG	1	$0.73s$	1

[2]Les résultats du test sont indisponibles. Celui-ci a été lancé 3 fois, pendant plus de 7 heures chaque fois. Les deux premières tentatives se sont soldées respectivement par une erreur de mémoire et un crash de MATLAB ; la dernière est toujours en cours.

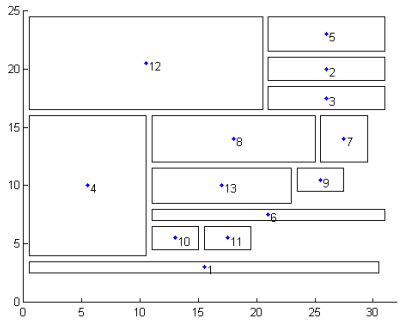
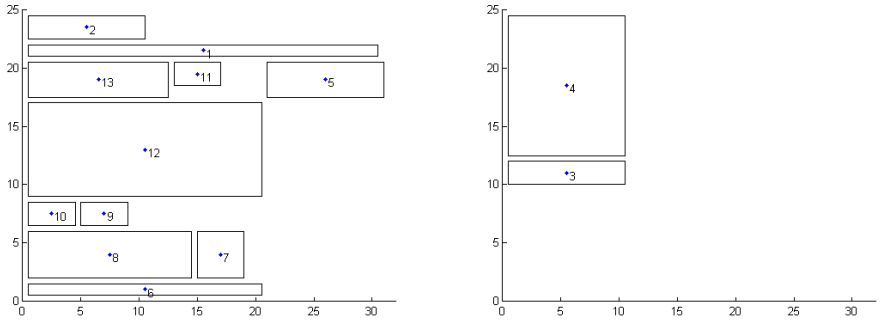


FIG. 27: Interface à 13 éléments, par la méthode BR



(a) première fenêtre

(b) seconde fenêtre

FIG. 28: Interface à 13 éléments, par la méthode BRG

	LA	aire	balance H	balance V	total
BR	193.7761	1	0.7669	0.8402	0.6444
BRG	13.85	1	[0.8008; 0.4436]	[0.7669; 0.8947]	0.6080

Tandis que BR et BRG affichent des résultats acceptables en matière de temps d'exécution, les tests sur l'interface à 13 éléments, inspirée de celle de Skype, font acte de la limite des méthodes Spring et LA sous leur implémentation actuelle. La méthode de Branch-and-Bound améliorée utilisée dans LA doit examiner de trop nombreux emplacements sur la grille si l'on tient compte de la distance inter-éléments, résultant ainsi en un temps d'exécution supérieur à 5 heures. La méthode Spring, quant à elle, souffre du positionnement initial aléatoire de ses éléments : elle ne parvient pas à aboutir à un optimum global, le nombre de possibilités de placement étant trop élevé.

6.5 Récapitulatif

La méthodes Bottom-Right, et la méthode Bottom-Right par groupe par extension, sont des moyens simples et rapides d'arriver à une solution de placement. Elles ne demandent pas beaucoup de ressources système, et leur code n'est pas très long, ce qui permet des les implanter sur des machines ne possédant pas une grande mémoire ou un processeur très puissant. Cependant, rien ne garantit la qualité des résultats produits : en effet, Bottom-Right ne respecte pas les contraintes de placement logique (adjacence et groupement), et son extension par groupe, malgré l'intégration des contraintes de groupement, possède toujours le même défaut que BR en ce qui concerne les contraintes d'adjacence. De plus, les layouts produits par ces méthodes ont tendance à être mal balancés, surchargeant la zone en haut à gauche et délaissant celle en bas à droite. Une piste reste encore à étudier pour améliorer ces algorithmes : sélectionner l'élément à placer non plus en fonction de l'aire, mais des liens avec les éléments déjà placés dans l'interface, ou même une fonction combinée de ces deux critères, ce qui permettrait de mieux tenir compte des relations d'adjacence.

La méthode à forces dirigées, Spring, a produit des interfaces bien balancées comme attendu. Cependant, elle est fortement handicapée par le positionnement aléatoire des éléments, qui contraint à faire de nombreuses itérations pour aboutir à une solution. Dès lors, celle-ci a besoin de ressources système un peu plus importantes que les méthodes BR et BRG, même si la mémoire nécessaire à son fonctionnement reste assez faible. Plusieurs pistes sont à envisager : adjoindre à Spring une méthode de prépositionnement des éléments, qui écarterait ou du moins minimiserait le facteur aléatoire dans le positionnement initial, et améliorer de l'expression du champ de forces des éléments, qui est peut-être lui aussi une cause des mauvais résultats de Spring.

La méthode de Layout Appropriateness, une fois améliorée pour tenir compte des distances entre éléments, donne des résultats plus que potables en matière de LA et de balance. Cependant, le procédé de Branch-and-Bound utilisé consomme énormément de mémoire pour stocker les noeuds explorés, même si le seuil d'exclusion des noeuds, l'évaluation de la borne inférieure sur le score LA et l'ordre de placement défini par les liens permettent de réduire grandement leur nombre. Un changement de l'algorithme d'exploration, remplaçant la méthode de Branch-and-Bound par une autre, plus efficace et moins gourmande en mémoire voire en temps d'exécution, ne pourrait que bénéficier à l'algorithme.

Conclusion

Bilan des objectifs

L'objectif premier de ce travail était de réaliser une étude comparative de ces algorithmes, en comparant leurs caractéristiques et leurs performances. Ceci a été réalisé tout au long de ce travail, tant en décrivant leur implémentation dans les chapitres 3, 4 et 5, qu'en effectuant les différents tests dans le chapitre 6. Nous avons identifié les défauts et les qualités des algorithmes, et sommes en mesure de donner des recommandations en ce qui concerne leur utilisation. Les informations concernant leur implémentation, ainsi que le code MATLAB, sont accessibles pour une utilisation future. Finalement, ce travail apporte une contribution à la recherche sur les interfaces graphiques avec les méthodes BRG et SPRING. Elles nécessitent encore toutes deux des améliorations, et même si les performances de SPRING sont limitées quant à la taille du problème traité, les premiers résultats restent tout de même encourageants.

Analyse critique

Comme déjà annoncé à la fin du chapitre 6, chaque méthode possède son lot de qualités et de défauts :

- La méthode **Bottom-Right** est, parmi toutes celles comparées, la méthode la plus rapide et la moins exigeante en ressources système. Ses résultats sont corrects, mais les layouts produits sont souvent mal balancés, et ne respectent ni les relations d'adjacence, ni de groupement.
- Son extension **Bottom-Right par Groupe** intègre quant à elle les relations de groupement. Elle reste rapide, et est peu gourmande en ressources système. Cependant, les layouts qu'elle produit sont également mal balancés, et violent parfois les relations d'adjacence entre éléments. De plus, le placement par groupe peut conduire à la séparation des éléments en d'avantage de fenêtres qu'avec la méthode BR.
- La méthode à **forces dirigées, SPRING**, est une méthode probabiliste, ce qui s'avère être un défaut dans l'état actuel de son implémentation. En effet, l'attribution aléatoire des positions des éléments en début d'appel de la fonction, et la convergence vers un optimum local et non global, impose de multiplier les itérations. Malgré la complexité plutôt faible de la méthode, le temps d'exécution devient alors plus conséquent. De plus, cette approche probabiliste perd rapidement de son efficacité lorsque le nombre d'éléments considérés augmente.
  Cependant, les layouts produits par SPRING ont un bon score LA, et sont bien balancées.

– La méthode de **Layout Appropriateness** souffre quant à elle de sa recherche Branch-and-Bound, qui, même avec les améliorations réalisées, demande beaucoup de temps et de mémoire pour son exécution. De plus, la balance du layout n'est pas garantie, car il ne s'agit pas d'un des critères d'optimisation de la méthode. Par contre, le score LA du layout obtenu est le meilleur possible, et, avec la conversion de la matrice d'adjacence en matrice de probabilités de transition, l'algorithme tient compte des relations d'adjacence et de groupement.

Après mûre réflexion sur le sujet, nous sommes arrivés à la conclusion que la méthode Spring, développée dans le cadre de ce travail, n'est pas à même de gérer à elle seule le positionnement des éléments dans l'interface. Il s'agit plutôt d'une méthode de positionnement finale, destinée à améliorer le layout produit par une méthode plus rapide, qui tiendrait compte des relations d'adjacence et de groupement, mais pas nécessairement des critères tels que la balance ou le Layout Appropriateness.
La meilleure solution serait donc de la coupler à un algorithme de prépositionnement rapide possédant ces caractéristiques.

Travaux à venir

Poursuivre l'étude comparative

Le premier objectif futur est de continuer la comparaison en considérant d'autres algorithmes, afin de réunir les informations sur leur fonctionnement et leur implémentation, qui seront ainsi à la disposition de ceux qui en ont besoin.

Amélioration des méthodes

– **Méthode de sélection de BR et BRG :**

La méthode de sélection des éléments à placer ne tient compte que de la surface occupée par ceux-ci. C'est la cause du non-respect des contraintes d'adjacence et de groupement. Pour remédier à cela, une prochaine étape serait le développement d'une fonction de sélection alliant les critères de la taille et des liens avec les éléments déjà placés.

– **Performances de SPRING :**

Afin d'améliorer les performances de SPRING, ou de la confirmer dans son rôle de méthode de placement final, le développement d'une méthode de prépositionnement est nécessaire. Celle-ci doit tenir compte des relations d'adjacence et de groupement, et être rapide. Réétudier l'expression du champ de forces des éléments pourrait s'avérer nécessaire, pour s'assurer que des améliorations ne puissent pas y être apportées.

– **Méthode de recherche de LA :**

Le temps d'exécution de l'algorithme LA, ainsi que la mémoire consommée, augmentent rapidement lorsque la taille du problème augmente. La recherche des solutions par Branch-and-Bound n'est pas la plus efficace des

méthodes connues, aussi un travail futur pourrait implémenter un autre algorithme de recherche, améliorant ainsi les performances de LA.

– **Evaluation :**

Les différents évaluateurs utilisés dans ce travail cotent l'aire superposée, le Layout Appropriateness et la balance horizontale et verticale. L'intégration de nouveaux critères dans ces fonctions permettrait d'obtenir une plateforme de cotation polyvalente.

Références

[1] ISO 9241. Ergonomic requirements for office work with visual display terminals (vdts) - part 11 : Guidance on usability, 1998.

[2] P. Botella, X. Burgués, J.P. Carvallo, X. Franch, G. Grau, J. Marco, and C. Quer. Iso/iec 9126 in practice : what do we need to know ?

[3] Isabel F. Cruz and Roberto Tamassia. Graph drawing tutorial.

[4] Krzysztof Gajos and Daniel S. Weld. Preference elictation for interface optimization.

[5] Krzysztof Gajos and Daniel S. Weld. Supple : Automatically generating user interfaces. *IUI'04*, January 2004.

[6] Stephen G. Kobourov. Force-directed drawing algorithms. *CRC Press*, 2004.

[7] Carrie A. Lee. Hci & aesthetics : The future of user interface design, 2007.

[8] Morten Moshagen and Meinald T. Thielsch. Facets of visual aesthetics. *International Journal of Human-Computer Studies*, 68(10), May 2010.

[9] David Chek Ling Nao, Lian Seng Teo, and John G. Byrne. A mathematical theory of interface aesthetics.

[10] Andrew Sears. Aide : A tool to assist in the design and evaluation of user interfaces.

[11] Andrew Sears. Layout appropriateness : A metric for evaluating user interface widget layout. *IEEE Transactions on Software Engineering*, 19, July 1993.

[12] Noam Tractinsky. Toward the study of aesthetics in information technology, 2004.

[13] Jean Vanderdonckt. Visual design methods in interactive applications.

[14] Jean Vanderdonckt. Visual techniques in multimedia applications.

A Illustrations du fonctionnement des méthodes

A.1 Méthode Spring

Les matrices suivantes reprennent les données de l'exemple, une interface à 6 éléments à afficher dans une fenêtre de 10×10 :

n	dimensions
1	8×2
2	3×6
3	5×1
4	2×2
5	2×2
6	3×1

$$\text{et } A = \left[\begin{array}{cccccc|cccc} 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{array}\right] \tag{3}$$

La figure **29** illustre le meilleur résultat obtenu à la sortie de la méthode. Sur le CD fourni en annexe, vous trouverez une vidéo[3] MATLAB intitulée *'Exemple_SPRING_6'* qui illustre toutes les étapes du positionnement des éléments.

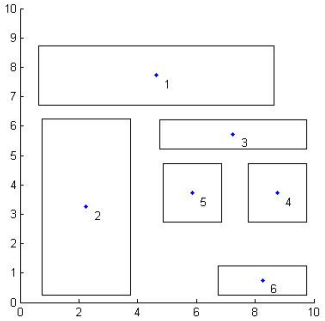


FIG. 29: Interface par la méthode SPRING

A.2 Méthode BR

Les quelques images regroupées ici illustrent les résultats obtenus grâce à l'algorithme Bottom-Right.

La **Fig. 30** montre l'interface obtenue pour une fenêtre de 8×7 et les éléments de dimensions suivantes :

$$dim = \left[\begin{array}{cc} 2 & 6 \\ 4 & 2 \\ 2 & 2 \\ 1.5 & 3 \end{array}\right]$$

[3] Afin de la visionner, placez-la dans le répertoire de travail de MATLAB, et exécutez la commande suivante : *M = aviread('Exemple_SPRING_6') ; movie(M) ;*

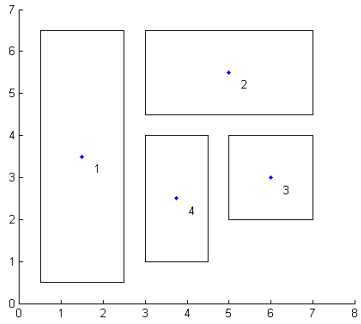


FIG. 30: Layout à 4 éléments calculé par BR

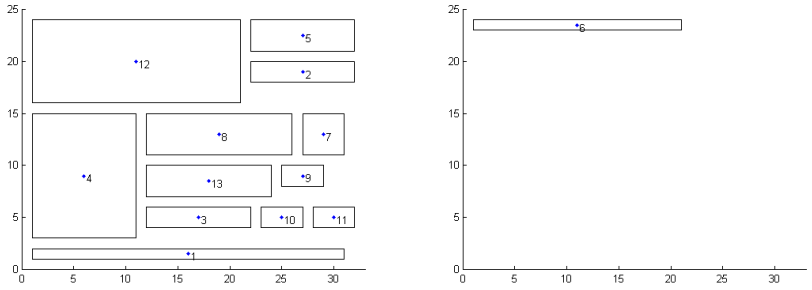
Il s'agit de l'exemple utilisé pour illustrer les différentes étapes d'une itération de l'algorithme. L'exemple **Fig. 31** montre une interface plus complexe, représentée dans des fenêtres de 33×25. Elle contient les éléments de dimensions suivantes :

$$dim = \begin{bmatrix} 30 & 1 \\ 10 & 2 \\ 10 & 2 \\ 10 & 12 \\ 10 & 3 \\ 20 & 1 \\ 4 & 4 \\ 14 & 4 \\ 4 & 2 \\ 4 & 2 \\ 4 & 2 \\ 20 & 8 \\ 12 & 3 \end{bmatrix}$$

On remarque que la première fenêtre est très fortement chargée, tandis que la seconde ne contient qu'un seul élément. Un algorithme plus évolué aurait pu effectuer une séparation plus équitable. Enfin, la troisième figure (**Fig. 32**) montre la même interface représentée cette fois sur un support de 32×17. L'adjacence des éléments a totalement changé, l'algorithme ne se préoccupant que de la taille des éléments dans son placement.

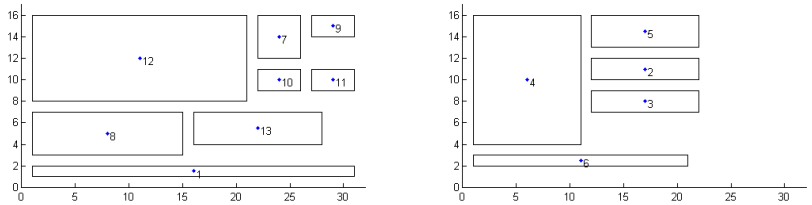
A.3 Méthode BRG

L'exemple utilisé est celui déjà présenté dans la méthode BR. Les dimensions des éléments n'ont pas changé, mais on rajoute cette fois aux données la matrice d'adjacence de l'interface :



(a) Première fenêtre (b) Seconde fenêtre

FIG. 31: Layout à 13 éléments calculés par BR, fenêtre 33×25

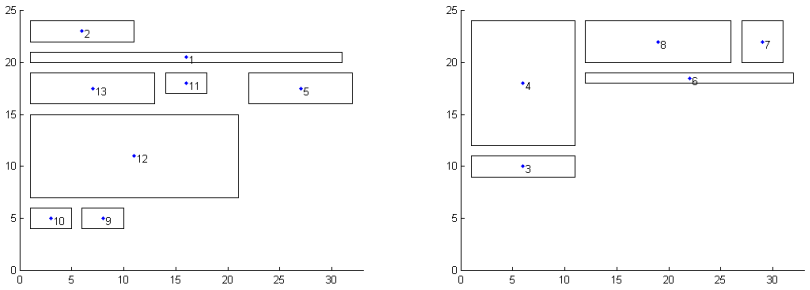


(a) Première fenêtre (b) Seconde fenêtre

FIG. 32: Layout à 13 éléments calculés par BR, fenêtre 32×17

$$dim = \begin{bmatrix} 30 & 1 \\ 10 & 2 \\ 10 & 2 \\ 10 & 12 \\ 10 & 3 \\ 20 & 1 \\ 4 & 4 \\ 14 & 4 \\ 4 & 2 \\ 4 & 2 \\ 4 & 2 \\ 20 & 8 \\ 12 & 3 \end{bmatrix} \text{ et } A = \left[\begin{array}{cccccccccccccc|cccc} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \end{array}\right]$$

L'algorithme effectue le placement de la figure **33** en tenant compte des relations de groupement. Le résultat obtenu est bien différent de celui de la méthode BR, et ce pour les mêmes contraintes de taille.



(a) Première fenêtre (b) Seconde fenêtre

FIG. 33: Layout à 13 éléments calculés par BRG, fenêtre 33×25

La **Fig. 34** contient les différentes fenêtres de l'interface représentée sur des fenêtres plus petites (32×17). L'interface obtenue est moins chargée que celle de BR, les groupes ayant été maintenus et répartis sur les différentes fenêtres.

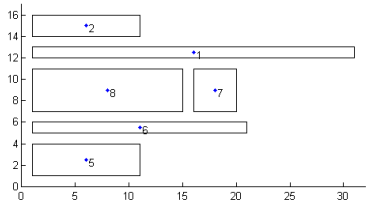
A.4 Méthode LA

L'exemple suivant illustre le fonctionnement de l'algorithme LA. L'interface est définie par :

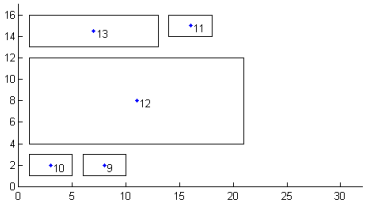
n	dimensions
1	16×4
2	6×12
3	10×2
4	4×4
5	4×4
6	6×2

et $A = \left[\begin{array}{cccccc|cccc} 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{array}\right]$

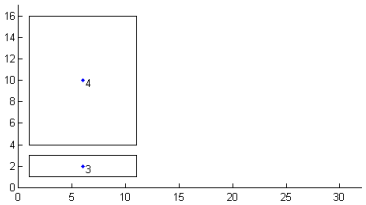
La distance *dist* souhaitée entre les éléments est de 1, et les relations de groupes sont considérées comme 2 fois plus fortes qu'une relation d'adjacence. La fenêtre utilisée pour l'affichage est de taille 20×20 L'algorithme commence d'abord par transformer la matrice d'adjacence en matrice de fréquences de transition :



(a) Première fenêtre



(b) Seconde fenêtre

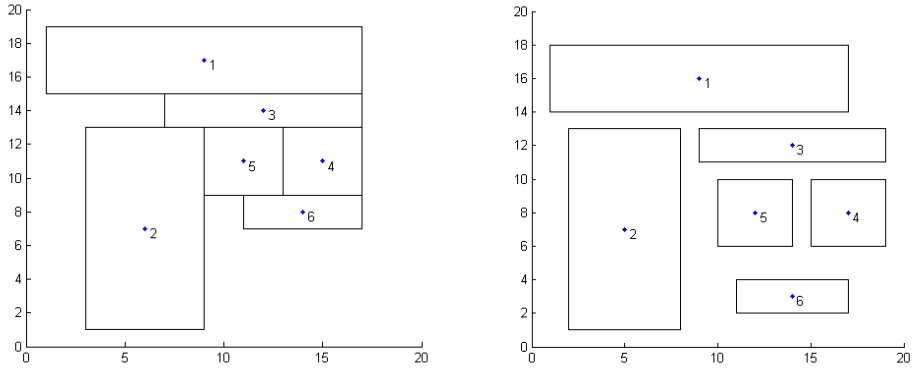


(c) Troisième fenêtre

FIG. 34: Layout à 13 éléments calculés par BRG, fenêtre 32×17

$$F = \begin{bmatrix} 0 & 1.0000 & 0.2857 & 0 & 0 & 0 \\ 0.5000 & 0 & 0.1429 & 0 & 0 & 0 \\ 0.5000 & 0 & 0 & 0.5000 & 0.5000 & 0 \\ 0 & 0 & 0.2857 & 0 & 0.5000 & 0 \\ 0 & 0 & 0.2857 & 0.5000 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Ensuite, l'algorithme décompose les différents 'widgets' en éléments unitaires (il y en a 200 en tout), encodant en même temps les relations de position 'en bas' et 'à droite' dans les matrices prévues à cet effet. Il ne reste plus qu'à explorer toutes les possibilités de placement pour tous les éléments, en tenant compte de la distance inter-éléments (*dist*). La figure **35** reprend l'interface sélectionnée par l'algorithme.



(a) sans distance entre éléments (b) avec une distance $dist = 1$

FIG. 35: LA - Interface obtenue après optimisation

B Fichiers MATLAB

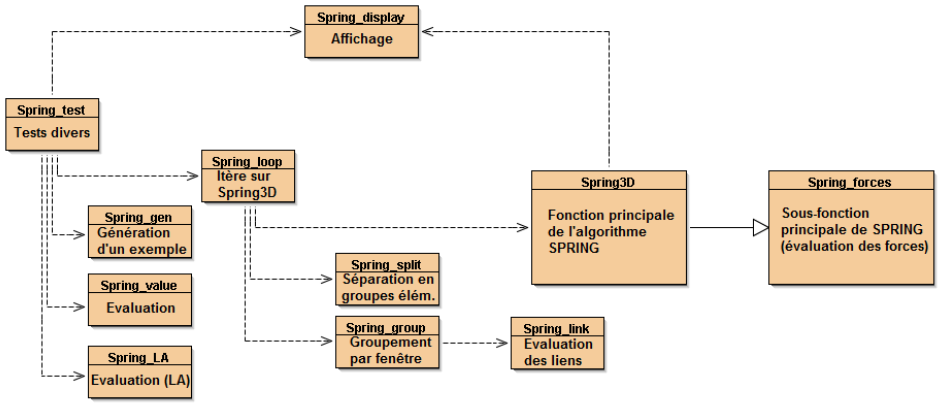


FIG. 36: Arborescence des fichiers de la méthode SPRING

La méthode SPRING (**Fig. 36**) représente un ensemble d'environ 430 lignes de codes, réparties comme suit entre 11 fonctions :

- **Spring_display (10 lignes) :** sert à afficher les résultats de toutes les méthodes implémentées ;
- **Spring_forces (60 lignes) :** sous-fonction principale de Spring3D, évalue les forces exercées sur chaque élément ;
- **Spring_gen (30 lignes) :** génère les données des exemples de type 'barre de tâche' ;
- **Spring_group (60 lignes) :** répartit les groupes élémentaires dans les différentes fenêtres d'affichage ;
- **Spring_LA (10 lignes) :** évalue le score LA d'un layout ;
- **Spring_link (10 lignes) :** calcule le nombre de liens d'adjacence entre deux éléments ou groupe d'éléments ;
- **Spring_loop (15 lignes) :** permet d'itérer sur la méthode Spring3D. Renvoie le meilleur résultat obtenu ;
- **Spring_split (25 lignes) :** sépare les éléments en groupes élémentaires ;
- **Spring_test (130 lignes) :** fonction de test principale ;
- **Spring_value (30 lignes) :** évalue le score d'un layout (aire superposée, balance) ;
- **Spring3D (50 lignes) :** fonction principale de la méthode SPRING.

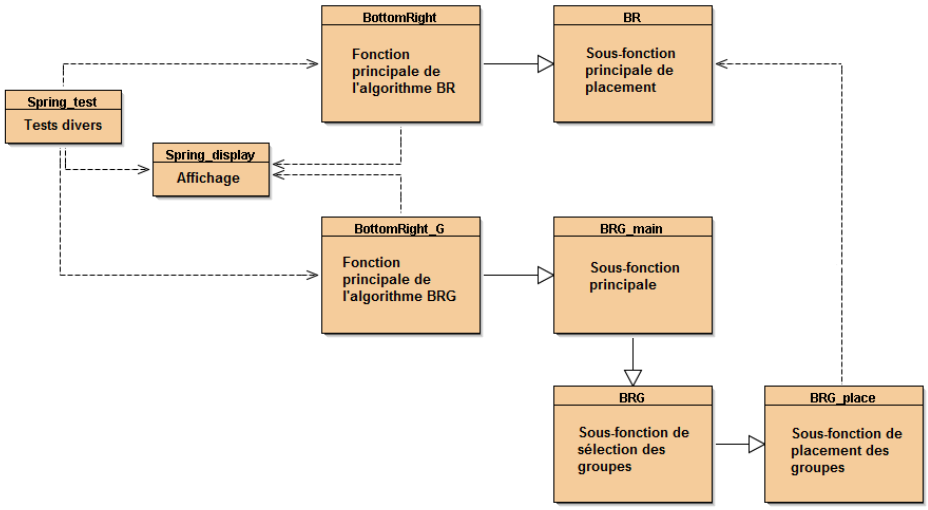


FIG. 37: Arborescence des fichiers de la méthode BR et BRG

Les méthodes BR et BRG (**Fig. 37**) rassemblent 6 fonctions pour un total de 130 lignes de code :

- **BottomRight(20 lignes) :** fonction principale de la méthode BR ;
- **BottomRight_G (35 lignes) :** fonction principale de la méthode BRG ;
- **BR (20 lignes) :** fonction de placement de la méthode BR ;
- **BRG (30 lignes) :** sous-fonction de BRG_main. Effectue la sélection du groupe élémentaire à placer.
- **BRG_main (20 lignes) :** sous-fonction principale de BottomRight_G.
- **BRG_place (35 lignes) :** sous-fonction de BRG. Effectue le placement (à l'aide de BR) du groupe sélectionné dans la plus petite aire possible.

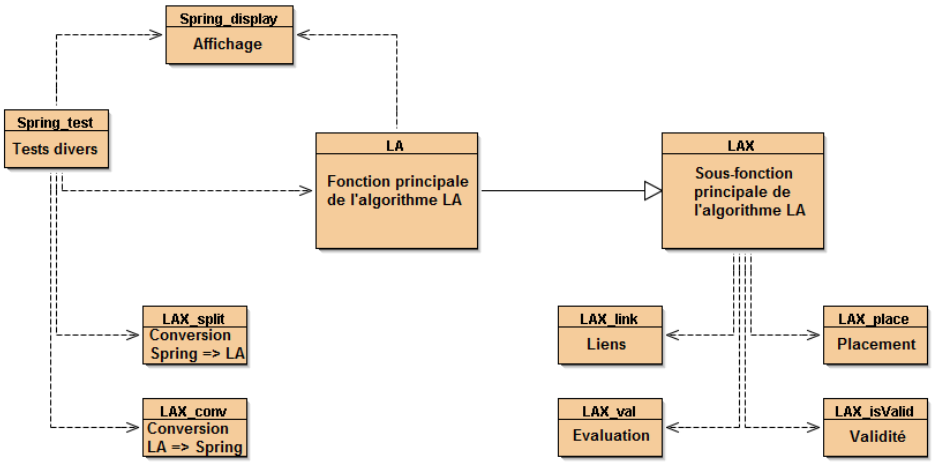


FIG. 38: Arborescence des fichiers de la méthode LA

La méthode LA (**Fig. 38**) compte 8 fonctions totalisant 195 lignes de code :

- **LA (35 lignes) :** fonction principale de la méthode LA ;
- **LAX (35 lignes) :** sous-fonction principale de LA, explore toutes les possibilités de placement pour l'élément sélectionné ;
- **LAX_conv (10 lignes) :** convertit la sortie de LA en données de même forme que les sorties de Spring3D, BottomRight et BottomRight_G ;
- **LAX_isValid (20 lignes) :** vérifie si le placement d'un élément respecte les distances inter-éléments ;
- **LAX_link (15 lignes) :** calcule la 'link value' d'un élément ;
- **LAX_place (20 lignes) :** place tous les noeuds contraints au noeud sélectionné ;
- **LAX_split (50 lignes) :** convertit les données d'entrée des tests en données compatibles avec LA ;
- **LAX_val (10 lignes) :** calcule l'estimation (borne inférieure) du score LA d'un layout.