Techniques génératives utilisées comme fonction de réparation : intégrer des contraintes dans les processus d'optimisation multicritère pour la conception architecturale

Claire Duclos-Prévet[1,*] and *François* Guéna[2]

[1] Laboratoire AS Explore de l'agence Architecture Studio, 75012 Paris et membre de l'UMR MAP-MAAC, ENSAPLV, 75019 Paris

[2] UMR MAP-MAAC, ENSAPLV, 75019 Paris

Résumé. L'intégration des contraintes est indissociable de l'optimisation multicritère. Pourtant, elle est rarement étudiée en conception architecturale computationnelle et performancielle. En combinant les processus évolutionnaires avec d'autres techniques génératives comme les processus à base de règles, l'auto-organisation ou les automates cellulaires, il est possible d'intégrer efficacement des contraintes dans des problèmes d'optimisation multicritère en utilisant la méthode des fonctions de réparation. Dans cet article, nous expérimentons ces techniques sur quatre cas d'étude issus de la pratique professionnelle et dérivons des lignes directrices pour le développement d'un solveur d'optimisation adapté à la réparation.

Mots-clés. algorithmes génétiques, méthodes d'intégration des contraintes, réparation, modélisation à base d'agents, solveur d'optimisation

Abstract. Constraint integration is a crucial aspect in multicriteria optimization, yet it often receives little attention in computational and performance-based architectural design. In this paper, we explore the integration of constraints through a combination of evolutionary processes with other generative techniques such as rule-based systems, self-organization, and agent-based models. By employing the repair function method, we demonstrate the efficient incorporation of constraints into multi-criteria optimization problems. Our experiments on four case studies drawn from professional practice yield guidelines for developing an optimization solver capable of constraint repair.

* Corresponding author: c.duclos.prevet@gmail.com

© The Authors, published by EDP Sciences. This is an open access article distributed under the terms of the Creative Commons Attribution License 4.0 (https://creativecommons.org/licenses/by/4.0/).

Keywords. genetic algorithms, contraints handling methods, integrated design, agent-based modeling, optimization solver

1 Introduction

Les modèles génératifs tels que les processus à base de règles, les modèles à base d'agents comme l'intelligence en essaim ou les automates cellulaires se distinguent des modèles paramétriques. Il s'agit de processus itératifs [1] auxquels la récursivité confère la faculté d'émergence. Un processus génératif est capable de créer, avec des règles simples et de manière autonome, une grande variété de formes non anticipées par le designer [2]. Cependant, cette particularité des techniques génératives les rend difficiles à contrôler. Ainsi, elles sont peu adaptées à la pratique, et restent cantonnées à des projets de papiers dont on peut trouver des exemples chez certaines agences d'architecture et de design identifiées dans la taxonomie de L. Ma [3].

Les processus génératifs évolutionnaires (fondés sur des algorithmes d'optimisation évolutionnaires comme les algorithmes génétiques) ne rencontrent pas cette limite, car les solutions générées sont en réalité définies, le plus souvent, par un modèle paramétrique. Ces méthodes sont très populaires dans la littérature scientifique [4], y compris leur intégration dans un processus BIM [5]. De nombreux outils ont été implémentés dans les modeleurs 3D démocratisant l'usage de l'optimisation (9 solveurs au total sur *Grasshopper*®), mais aussi les logiciels BIM avec Optimo sur *Dynamo*® ou *Generative Design Primer*®[6]. Les premières applications commencent à apparaître sur des cas réels pour optimiser la performance environnementale des bâtiments [5,6], bien que cela reste rare dans la pratique [9].

Un des sujets souvent négligés en optimisation appliquée à la conception architecturale est la question de l'intégration des contraintes, pourtant si courantes dans la pratique professionnelle des architectes. À notre connaissance, aucun outil à disposition des architectes ne permet de faciliter leur prise en compte. Récemment, il a été montré sur un unique cas d'application [10] qu'un algorithme génératif pouvait permettre de traiter efficacement des problèmes particulièrement contraints en faisant de la réparation de solution.

Nous proposons donc d'expérimenter cette méthode de la réparation par processus génératif sur plusieurs cas d'études issus de la pratique professionnelle de l'agence Architecture Studio. Quatre cas d'études permettent de traiter des problèmes distincts avec des contraintes de différentes natures et en utilisant différents types de techniques génératives.

2 Fonction de réparation et techniques génératives

Il existe de nombreuses méthodes pour intégrer des contraintes dans un problème d'optimisation, notamment avec les algorithmes génétiques [11]. La méthode la plus populaire est la fonction de pénalisation, car elle est la plus facile à reproduire et donne des résultats raisonnables pour de nombreux problèmes [12]. Elle consiste à pénaliser les valeurs

des fonctions objectifs pour les mauvaises solutions. Il y a cependant de nombreux problèmes pour lesquels cette méthode ne fonctionne pas.

2.1 Les fonctions de réparations

Lorsque la région des solutions faisables est proportionnellement étroite par rapport à l'ensemble des solutions possibles, les méthodes basées sur les fonctions de pénalité fonctionnent mal [11]. En optimisation combinatoire, il semblerait que la méthode la plus adaptée à la fois en termes de performance et en temps de calcul soit la fonction de réparation [13]. L'efficacité de cette méthode a déjà fait l'objet d'une étude comparative appliquée à un problème d'architecture [10, 14]. Cette dernière a confirmé ce résultat.

Une fonction de réparation est un algorithme qui cherche à modifier une solution infaisable pour la rendre faisable. Avec cette méthode, l'ensemble des solutions évaluées sont exploitables, ce qui permet d'éviter de perdre du temps en simulation inutile. Pour que cette méthode fonctionne, il faut qu'un tel algorithme existe, ce qui n'est pas toujours le cas. Cependant, les problèmes d'optimisation combinatoire, comme les problèmes rencontrés en architecture, sont réputés pour être assez adaptés à cette méthode [11]. Salcedo-Sanz [15] a identifié des typologies d'algorithmes de réparation décrits dans la littérature scientifique, mais il s'agit d'applications très éloignées de l'architecture.

Le principal reproche fait à cette méthode est que même si un algorithme de réparation existe (ce qui est difficile à savoir a priori), il faut encore le trouver, et cela peut être très chronophage d'autant qu'un algorithme de réparation est propre à un problème donné. Parfois, la réparation est tellement complexe que les temps de calcul pour réparer une solution ne sont plus adaptés à un processus d'optimisation. Enfin, un algorithme de réparation peut fortement transformer le génome des solutions d'une génération à l'autre, ce qui peut perturber le processus d'évolution [16].

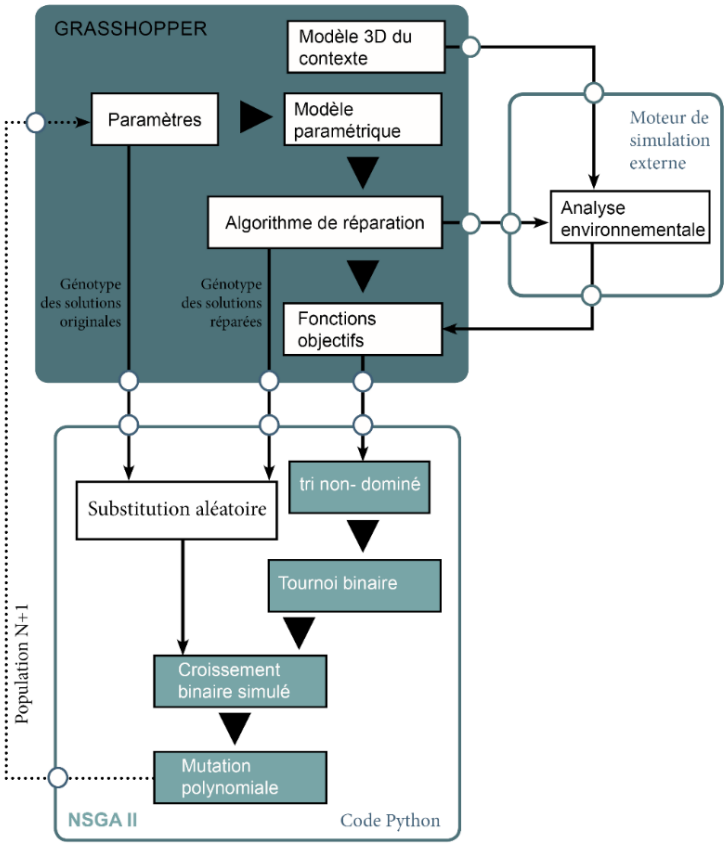

Figure 1. Architecture du solveur d'optimisation Urchin

Il existe deux manières d'implémenter un algorithme de réparation dans un processus d'optimisation évolutionnaire :

1. L'approche Baldwinienne dans laquelle le génome utilisé lors du croisement génétique est celui des solutions originales
2. L'approche Lamarckienne dans laquelle le génome des solutions originales est remplacé par celui des solutions réparées, pour une partie ou pour l'ensemble de la population [13].

La première approche peut être effectuée avec n'importe quel solveur d'optimisation. Pour tester l'approche lamarckienne, nous avons implémenté dans Grasshopper© notre propre solveur d'optimisation, nommé Urchin, qui utilise le très populaire algorithme NSGAII [17]. Ce solveur a été développé pour Grasshopper avec Ghpython sur Rhino 7. Le composant principal du solveur permet de lancer en arrière-plan un outil codé Python 3 qui permet d'utiliser la librairie d'optimisation Pymoo. Le fonctionnement général de la méthode partant du modèle paramétrique jusqu'au solveur en passant par la fonction de réparation et les outils de simulation sont décrits en figure 1.

Avec une approche Lamarckienne, il faut déterminer le bon taux de remplacement du génome des solutions originales par celui des solutions réparées, celui-ci varie selon les auteurs et le type de problèmes [13]. David Orvosh [18] a montré que le taux de remplacement optimal pour des problèmes d'optimisation combinatoire était de 5 %.

Nous testons la méthode des fonctions de réparation sur quatre cas d'études différents, tous issus de la pratique professionnelle de l'agence Architecture Studio. Ces cas d'étude ont été choisis pour représenter la diversité des problèmes rencontrés dans la pratique (génération d'un îlot urbain, optimisation de la morphologie d'une façade et de l'allocation du vitrage, optimisation d'un détail ou de l'ensemble du système de protection solaire), mais aussi la diversité des types de contraintes rencontrées (environnementales, constructives, esthétiques, conflits géométriques).

2.2 Les différentes techniques génératives

Les fonctions de réparations sont souvent des algorithmes qui vont répéter en boucle une opération jusqu'à ce que le critère de définition de faisabilité de la solution soit satisfait, ce qui correspond à la définition même d'un processus génératif [1], on peut aussi parler de « boucle conditionnelle ». Les processus génératifs sont aussi caractérisés par leur capacité à obtenir des résultats complexes à partir de règles simples, ce qui leur confère une capacité d'émergence [19] (c'est-à-dire qu'ils sont capables de créer des formes inattendues).

En architecture computationnelle, différentes techniques génératives peuvent être utilisées [20]. Parmi les plus populaires après les algorithmes génétiques, on trouve les grammaires de formes et les L-systèmes, mais aussi les modèles à base d'agents (MBA) [21] comme les automates cellulaires (AC) [22] ou l'intelligence en essaim [23]. Depuis quelques années, les algorithmes d'apprentissage profond comme les réseaux de neurones sont récurrents dans la littérature scientifique [24, 25]. Nous verrons avec nos différents cas d'études que la méthode des MBA s'est révélée particulièrement adaptée à la création de fonctions de réparation.

« Système d'auto-organisation », « modélisation à base d'agents », « simulations à base d'agents », « systèmes multi-agents », tous ces termes renvoient à la notion d'intelligence artificielle distribuée selon C.M. Macal [26]. Il est difficile de trouver un consensus sur la définition du terme MBA, car il est employé dans de nombreuses disciplines avec des définitions variables. Nous entendons par MBA est un processus itératif composé de multiples individus nommés « agents » capables d'entrer en interaction entre eux et avec leur environnement. Ces individus sont identifiables, autonomes, modulaires, sociables et leur état est variable au cours du temps [27]. De leurs interactions, ils déduisent des informations locales entraînant pour chaque agent l'exécution de règles simples qui permettent d'atteindre un objectif global.

L'on peut distinguer deux types de MBA populaires : l'intelligence en essaim (IE), et les automates cellulaires (AC). L'IE se distingue par des MBA fortement bio-inspirés où les agents ont des comportements semblables à ceux des insectes ou plus généralement des animaux sociaux. Il est possible de faire de l'IE sans nécessairement s'inspirer du comportement d'un animal précis, mais en imaginant des règles simples capables d'atteindre un objectif. Les agents d'un processus d'IE sont simples, presque tous identiques, mais surtout ils se déplacent dans l'espace, contrairement aux agents des AC. Ces derniers, appelés alors « cellules », sont disposés sur une trame orthogonale à une, deux ou trois dimensions, et c'est leur état qui évolue au fil des itérations en fonction de l'état des cellules voisines.

3 Applications sur quatre cas d'études

Nous avons utilisé les méthodes de réparation Baldwinienne et Lamarckienne à l'aide de technique générative sur quatre cas d'études différents. Afin de pouvoir comparer l'efficacité des différentes méthodes et mesurer l'impact du taux de remplacement sur l'efficacité de l'algorithme génétique, nous avons effectué cinq explorations par cas d'étude, avec dix générations de trente individus chaque (au total : 300 évaluations), comprenant une exploration Baldwienne et quatre lamarckiennes à des taux de remplacement différents (5 %, 15 %, 50 % et 100 %). Nous avons également réparé un groupe de contrôle de 300 solutions générées au hasard pour vérifier que l'algorithme de réparation ne perturbe pas le processus d'évolution de l'algorithme génétique.

Pour chaque cas d'étude, nous avons déterminé le taux de remplacement approprié pour trouver les meilleures solutions (non dominées) en agrégeant l'ensemble des solutions trouvées lors des 6 explorations et effectué un tri non dominé similaire à celui effectué par l'algorithme NSGAII. Ainsi, sur les figures 3, 4, 7 et 9, le rang 1 correspond aux solutions situées sur le front de Pareto (les meilleures solutions), le rang 2 est constitué des solutions légèrement inférieures, etc.

L'ensemble des modèles ont été réalisés sur Grasshopper et les visualisations sont des captures d'écrans du modeleur Rhinocéros. Les simulations d'ensoleillement et de lumière ont été réalisées à l'aide des plugins Honeybee et Ladybug [28]. Les analyses des résultats des optimisations ont été réalisées sur Python 3 et les graphiques produits avec Microsoft Excel.

3.1 Intelligence en essaim pour éviter les collisions d'objets

Le problème des collisions d'objets est un cas courant en optimisation appliquée à la conception architecturale et urbaine. On peut le retrouver sur des exemples de conception d'enveloppe [29], d'allocation de fenêtres, de génération de plan [30], ou bien sur des cas d'urbanisme [31].

Figure 2. Résultats des analyses des trois fonctions objectifs du cas d'étude 1

Nous avons rencontré ce premier problème pour la rétroconception de logements à Aubervilliers en île de France. Au cœur d'une ancienne forteresse militaire sont disposés sans logique géométrique apparente treize bâtiments de même hauteur et de même forme que celle de l'ellipse extrudée. L'objectif de cette première exploration est d'interroger la disposition des bâtiments en fonction des apports solaires et des vis-à-vis.

Le modèle paramétrique permettant de définir l'ensemble de 1.1^{1097} solutions (faisables et non faisables) permet de faire varier les coordonnées de chaque bâtiment en X et Y, et leur orientation (soit 39 variables au total). L'analyse des vis-à-vis a été réalisée avec des lancers de rayon.

Parmi les solutions définies par le modèle paramétrique, une très large majorité ne respecte pas une, voire deux contraintes essentielles : (i) les bâtiments ne doivent pas se superposer et (ii) ni sortir de la limite constructible. Pour intégrer ces contraintes, nous avons développé un algorithme de répulsion adapté aux ellipses. Nous avons utilisé l'intelligence en essaim pour créer un système d'auto-organisation [20, 32] qui force les ellipses à se repousser les unes les autres. Ici les agents sont les empreintes des bâtiments. Leur comportement varie en fonction du nombre de bâtiments avec lequel il rentre en collision, du type de collision, et la distance avec la limite constructible.

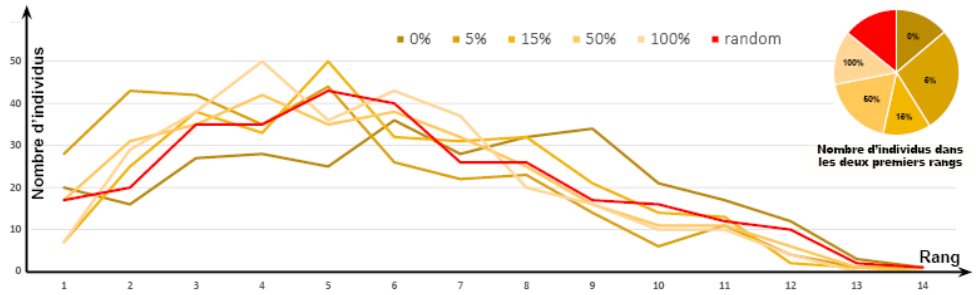

Figure 3. Nombre d'individus par rang pour les 6 explorations pour le cas d'étude 1

Lorsque l'on teste le modèle paramétrique, il apparaît que ce problème est particulièrement contraint, la taille de la région faisable est tellement étroite que sur 1 000 solutions générées aléatoirement, aucune ne respecte les deux contraintes énoncées. Quel que soit le taux de remplacement utilisé, l'algorithme génétique n'obtient pas beaucoup plus de solutions dans les premiers rangs (les meilleures solutions) que le groupe de contrôle, il a donc eu du mal à converger. Nous notons tous de même que le taux le plus efficace fut celui recommandé pour les problèmes d'optimisation combinatoire (5 %).

3.2 Simple boucle conditionnelle pour intégrer une contrainte environnementale

En pratique, une exigence environnementale peut soit être formulée comme un critère d'optimisation (comme essayer d'éclairer au mieux une pièce par exemple), soit comme une contrainte (comme obtenir un éclairement naturel de 500 lux sur 50 % du temps pour 60 % de la surface de la pièce). Dans le second cas, les solutions qui ne respectent pas cette règle seront considérées comme inexploitables par l'architecte. Il est alors préférable d'évincer ces solutions du processus d'exploration. C'est aussi une façon de hiérarchiser les critères d'optimisation. Cette sorte de préoptimisation permet d'assurer qu'un critère environnemental jugé prioritaire atteint un objectif minimum.

Pour le projet du tribunal judiciaire de Saint-Laurent-du-Maroni, en Guyane, le système de protection solaire nécessitait de protéger le bâtiment des intempéries. Une exploration a été réalisée pour minimiser l'ensoleillement tout au long de l'année tout en maximisant la lumière naturelle dans les salles d'audience et les bureaux. Nous nous sommes inspirés de ce projet pour notre second cas d'étude.

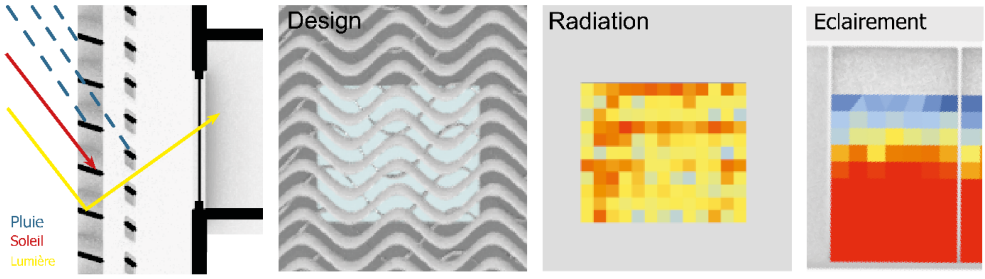

Figure 4. Coupe, élévations et résultats des analyses pour le cas d'étude 2

Deux couches de lames en aluminium en forme de vagues, comme illustré en figure 4, sont utilisées pour favoriser une lumière indirecte. Le modèle paramétrique compte un total de 11 variables (la distance entre les deux couches, le nombre de lames, le nombre de vagues par lames, leur amplitude, l'inclinaison et la largeur des lames). L'espace de solutions est de 54.6^{1013} solutions. L'éclairement naturel a été calculé pour les pires conditions, le 21 décembre et les précipitations ont été analysés avec des lancers de rayons en utilisant la vitesse et de l'orientation des vents dominants locaux et une hypothèse sur la taille des gouttes.

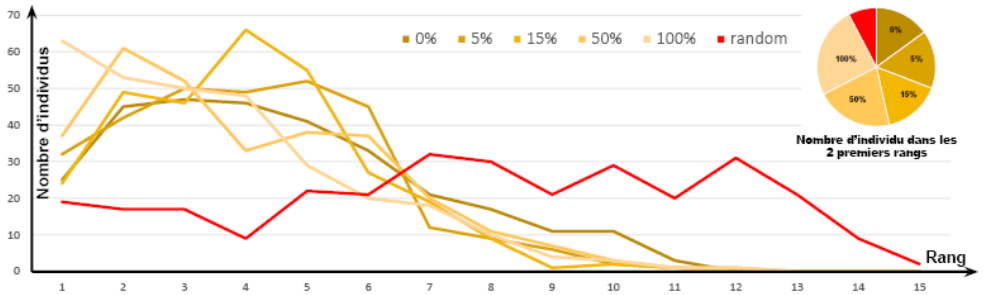

Figure 5. Graphique du nombre d'individus en fonction du rang pour les 6 explorations pour le cas d'étude 2

Pour ce second cas, 36 % des solutions générées n'ont pas nécessité de réparation. Contrairement à l'exemple précédent, l'algorithme génétique ne semble pas avoir eu de difficulté à converger puisque la quantité de solutions des meilleurs rangs y est significativement plus importante lorsque l'on a utilisé l'algorithme, comme on peut l'observer en figure 5. Dans ce cas, c'est un remplacement de 100 % des génotypes des solutions réparées qui est le paramétrage le plus performant. La quantité d'individus dans les meilleurs rangs augmente avec le pourcentage de remplacement.

3.3 Automate cellulaire pour supprimer les aberrations

Lorsque l'on définit des ensembles de solutions à partir de modèles paramétriques, il arrive relativement fréquemment de produire des aberrations, c'est-à-dire des solutions qui imposent d'intégrer des contraintes qui ne viendrait pas à un architecte d'énoncer (tellement elles sont évidentes). Éviter les collisions entre deux bâtiments fait partie de ce que nous appelons une aberration.

Le cas d'étude que nous présentons ici est un bon exemple d'aberration. S'agissant de la conception d'une façade placée sur une trame orthogonale, ces contraintes peuvent être aisément intégrées en utilisant un automate cellulaire en deux dimensions.

Ce troisième exemple a déjà fait l'objet d'une publication [33]. Pour le concours de la tour de bureaux Axian située à Madagascar, nous avons, pour une des variantes étudiées en phase esquisse, cherché à générer un design de façade qui permette de réduire au maximum les apports solaires tout au long de l'année tout en conservant une qualité de vue vers l'extérieur pour chaque bureau.

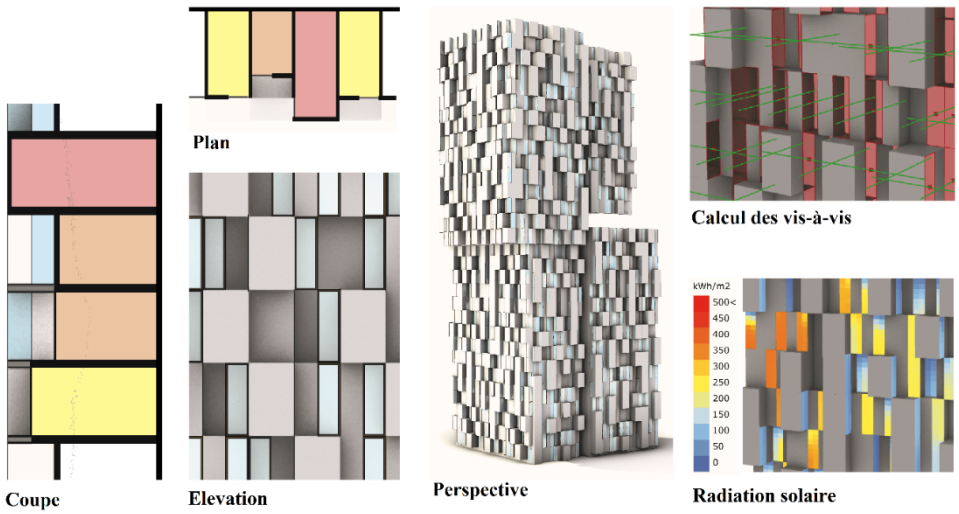

Figure 6. Coupe, élévations et plan, perspective et résultats des analyses pour le cas d'étude 3

Nous avons proposé de créer une façade poreuse avec des profondeurs de bureaux variables. Le principe de cette enveloppe est illustré en figure 6. Cette porosité permet à l'enveloppe de créer des ombres propres afin de s'autoprotéger des rayons du soleil sans nécessiter un système de protection supplémentaire qui réduirait drastiquement la visibilité depuis les bureaux vers l'extérieur. Ainsi, les façades ont été divisées selon une trame rectangulaire (un rectangle par bureau). Les bureaux sont disposés selon trois façons : avec un débordement nul, moyen ou grand. Pour chaque module de bureau, une partie devait être vitrée entraînant un total de 10 configurations de bureaux possibles.

L'optimisation a été réalisée seulement sur une partie de la tour, un morceau de façade orienté plein sud (avec des masques), composé de 150 modules de bureaux. Le modèle paramétrique compte donc le même nombre de variables (soit 1^{10150} solutions au total). Lorsque le problème est formulé de cette manière, il y a des incompatibilités de voisinage. Il arrive que deux éléments vitrés se fassent face, entraînant un vis-à-vis non désiré, ou qu'un élément vitré donne sur un bureau voisin plutôt que sur l'extérieur. Pour éviter ces aberrations, un certain nombre de règles ont été intégrées à l'aide d'un AC.

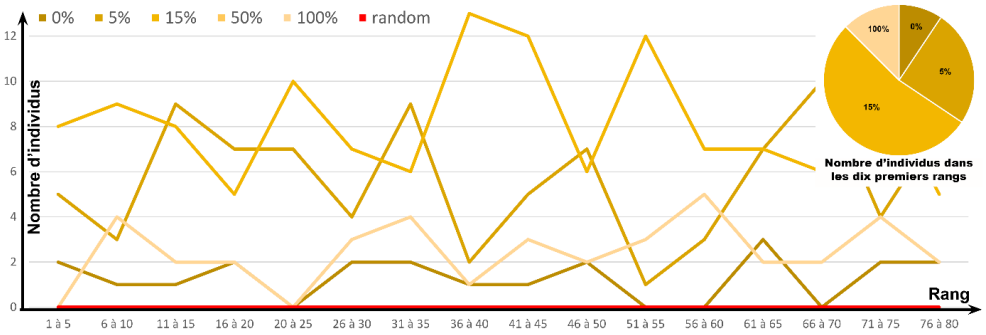

Figure 7. Graphique du nombre d'individus en fonction du rang pour le cas d'étude 3

Pour ce problème extrêmement contraint, car sur 1 000 solutions générées aléatoirement, toutes présentaient au moins une aberration, l'algorithme génétique a été particulièrement performant, car aucune des solutions situées dans les 80 meilleurs rangs ne provient du groupe de contrôle. Un taux de remplacement se distingue nettement des autres, celui de 15 % comme on peut le voir en figure 7.

3.4 Succession de boucles pour un problème multicontraint

Il n'est pas rare qu'il faille intégrer plusieurs contraintes de natures différentes pour un même problème d'optimisation, comme c'est le cas pour ce dernier exemple impliquant à la fois des contraintes constructives et des contraintes esthétiques. Aussi, trouver un unique algorithme qui permet de résoudre tous les problèmes simultanément relève du fantasme. On peut essayer d'élaborer un MBA [27] qui permettrait d'intégrer toutes les contraintes en les pondérant. Si les contraintes ne rentrent pas en concurrence, il est aussi possible de les traiter séparément et successivement avec différente fonction de réparation.

Lors d'un concours pour un ensemble scolaire à Marseille, nous avons été confrontés à un problème multicontraint lors de l'optimisation de son système de protection solaire. Ce dernier devait permettre d'éviter les surchauffes estivales tout en assurant un certain niveau de confort visuel (comprenant à la fois l'accès à la lumière naturelle et à la qualité de la vue vers l'extérieur).

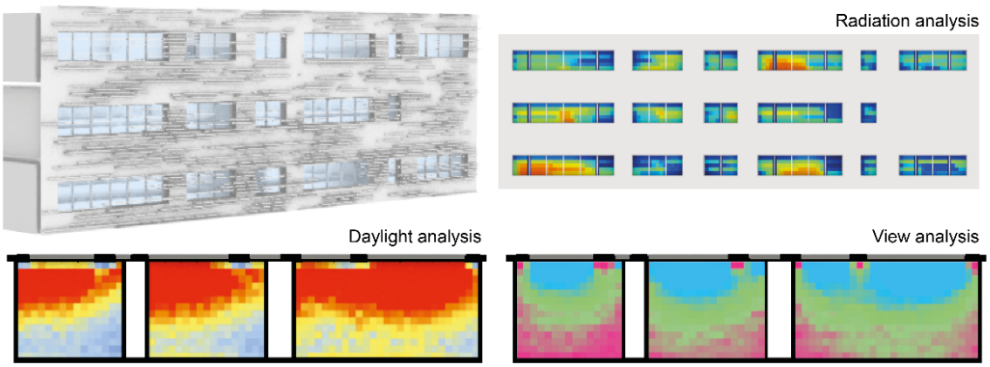

Figure 8. Modélisation 3D et résultats d'analyse pour le cas d'étude 4

Comme l'illustre la figure 8, l'enveloppe du bâtiment devait être protégée avec une grande quantité de lames horizontales de tailles différentes entraînant un effet de disposition aléatoire. Pour que des solutions générées à l'aide d'un modèle paramétrique puisse être viables, il fallait intégrer au modèle les contraintes suivantes :

(i) interdiction pour une lame de démarrer ou finir devant une fenêtre

(ii) limiter le nombre de type de lame et quantité maximale de mètres linéaires de lame

(iii) obtenir une répartition homogène des lames sur l'ensemble de la façade et conserver un effet de disposition aléatoire.

Un MBA (où chaque lame est un agent) a d'abord été développé pour tenter de réparer les solutions avec un unique algorithme. Pour chaque solution générée par le processus d'optimisation, le nombre d'itérations nécessaires pour qu'elle soit réparée rendait le processus de réparation trop chronophage pour être incompatible avec un processus d'optimisation. Une méthode plus rapide a donc ensuite été étudiée. Un modèle paramétrique permettant d'allouer 252 lames à l'aide d'une grille de points et de 504 variables (une par point), et de 252 autres variables pour la largeur de chacune des lames (soit un total de $2{,}6^{101130}$ solutions) a donc été défini.

Trois fonctions de réparation successives ont ensuite été utilisées. Une première fonction permet d'assurer une réparation homogène des lames en translatant les lames des zones les plus denses vers les zones les moins denses. Une seconde fonction permet de translater, agrandir ou rétrécir les lames dont les extrémités sont situées devant des fenêtres pour éviter les porte-à-faux. Enfin, une dernière fonction permet de réduire si nécessaire de manière proportionnelle la longueur de chaque lame pour respecter la contrainte sur la quantité maximale de lames.

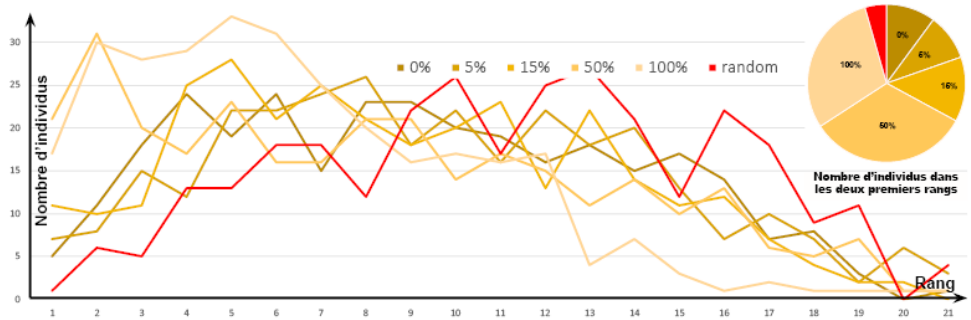

Figure 9. Graphique du nombre d'individus en fonction du rang pour le cas d'étude 4

Ce problème est aussi très contraint, puisque sur 1 000 solutions générées aléatoirement aucune ne respecte toutes les contraintes. Pour ce dernier exemple, NSGAII ne semble pas perturbé par la réparation (si l'on compare les résultats à celui du groupe contrôle en rouge sur la figure 9). Il arrive à converger notamment lorsque le taux de remplacement est important (50 ou 100 %).

4 Conclusions

Les expérimentations présentées ici ne suffisent pas pour tirer des conclusions définitives sur la meilleure façon d'implémenter une fonction de réparation. Cela constitue une première adaptation d'une méthode ayant fait ses preuves sur des problèmes très éloignés de la conception architecturale.

Avec nos expérimentations sur quatre cas d'études, l'approche lamarckienne donne de meilleurs résultats que l'approche baldwinienne. Cela justifie le besoin d'un solveur d'optimisation adapté permettant le remplacement des génotypes des solutions originales par ceux des solutions réparées. Aussi, il apparaît que la règle du taux de remplacement à 5 % pour les problèmes discrets [18] n'est pas systématiquement vérifiée. À l'avenir, nous pourrions chercher à savoir si la dimension de la région faisable peut avoir un impact sur le taux de remplacement optimal.

Pour un cas sur quatre, l'exploration conduite par l'algorithme génétique semble perturbée. Seul un taux de remplacement de 5 % se distingue du groupe contrôle en quantité de solutions performantes trouvées. C'est aussi le problème pour lequel les solutions réparées s'éloignent le plus des solutions originales en termes de génotype, ce qui corrobore avec les recommandations de Smith et ses co-auteurs [16].

Enfin, il apparaît que s'il est faisable de trouver une fonction de réparation sur un problème d'optimisation combinatoire, il est parfois complexe de définir une fonction suffisamment peu chronophage, notamment lorsqu'il y a beaucoup de contraintes. Les modèles à base d'agents sont pertinents lorsque quelques itérations seulement suffisent pour réparer la solution.

Les fonctions de réparation peuvent être utilisées sur différents types de problèmes et pour des contraintes de différentes natures : constructives, esthétiques, environnementales, de coûts de construction, ou des aberrations géométriques comme les conflits. Pour cela un algorithme spécifique doit être développé pour chaque problème. Des techniques génératives comme de simples boucles conditionnelles, ou encore des automates cellulaires ou l'intelligence en essaim, sont pertinentes pour créer des fonctions de réparation.

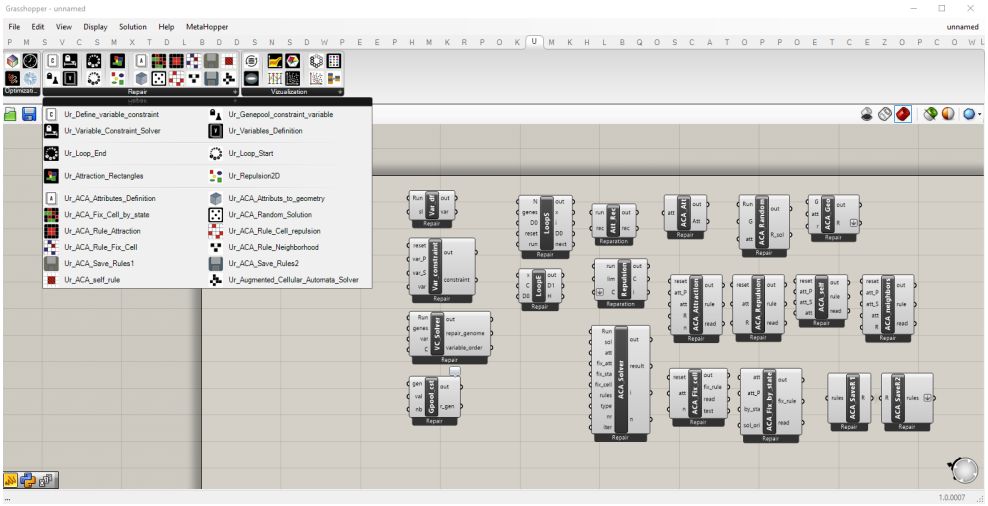

Figure 10. Composants Urchin pour la réparation de solution

Considérées comme les méthodes les plus efficaces pour les problèmes d'optimisation combinatoire sous contrainte, les méthodes à base de fonctions de réparation sont difficiles à implémenter lorsque l'on ne maîtrise pas la programmation informatique, comme cela est souvent le cas chez les architectes. Ainsi, à la suite de cette étude comparative, les algorithmes développés pour créer les fonctions de réparations pour ces quatre applications ont été étendus pour atteindre un niveau d'abstraction qui facilite leurs réemplois sur

différents problèmes rencontrés en conception architecturale. Ces composants Grasshopper s'ajoutent au solveur d'optimisation « Urchin » déjà présenté et permettent d'aboutir à une première version d'un nouveau plugin Grasshopper dédié à l'optimisation multicritère sous-contrainte. Au total, une vingtaine de composants de ce plugin, présentés en figure 10, sont dédiés à la réparation. Une grande partie d'entre eux permettent de définir des automates cellulaires 2D complexes avec des règles personnalisables. Il contient aussi des MBA pour faire de la répulsion de régions 2D et des outils pour faire des boucles conditionnelles compatibles avec le solveur d'optimisation.

Cette bibliothèque d'outil Grasshopper dédiée à la réparation reste évidemment incomplète. Elle ne concerne aujourd'hui que les problèmes en deux dimensions. À l'avenir, il serait intéressant d'adapter ces outils à la 3D et de vérifier l'efficacité de la méthode des fonctions de réparation sur des problèmes plus complexes en trois dimensions comme la génération de forme urbaines avec des voxels ou l'allocation spatiale en 3D.

Références

[1] I. Caetano, L. Santos, and A. Leitão (2020), Computational design in architecture: Defining parametric, generative, and algorithmic design, *Frontiers of Architectural Research,* **9**(2), pp. 287–300.
[2] J.E. Harding and P. Shepherd (2017), Meta-Parametric Design, *Design Studies,* **52**, pp. 73–95.
[3] L. Ma (2015), Invention architecturale et algorithmes non-lineaires, https://www.theses.fr/2015VERS023S.
[4] S. Zhao and E. de Angelis (2018), Performance-based Generative Architecture Design: A Review on Design Problem Formulation and Software Utilization, *Journal of Integrated Design and Process Science*. **22**(3), pp. 55–76.
[5] W. Ma, X. Wang, J. Wang, X. Xiang, and J. Sun (2021), Generative Design in Building Information Modelling (BIM): Approaches and Requirements, *Sensors*. **21**(16), pp. 5439.
[6] Generative Design Primer, https://www.generativedesign.org.
[7] X. Shen, A. Singhvi, A. Mengual, M. Spastri, and V. Watson (2018), Evaluating the Multi-Objective Optimization Methodology for Performance-Based Building Design in Professional Practice, *ASHRAE and IBPSA*, pp. 646–653.
[8] M. Bernal, V. Okhoya, T. Marshall, C. Chen, and J. Haymaker (2020), Integrating expertise and parametric analysis for a data-driven decision-making practice, *International Journal of Architectural Computing,* **18** (4), pp. 424–440. *CUMINCAD*.
[9] S. Li, L. Liu, and C. Peng (2020), A review of performance-oriented architectural design and optimization in the context of sustainability: Dividends and challenges, *Sustainability*, **12** (4), pp. 1427,.
[10] C. Duclos-Prévet, F. Guéna, and M. Efron (2022), Constraint handling methods for a generative envelope design using genetic algorithms: The case of a highly constrained problem, *International Journal of Architectural Computing*, **20** (3), pp. 587–609.
[11] C.A. Coello Coello (2002), Theoretical and numerical constraint-handling techniques used with evolutionary algorithms: a survey of the state of the art. *Computer Methods in Applied Mechanics and Engineering*, **191** (11), pp. 1245–1287.
[12] Z. Michalewicz and M. Schoenauer (1996), Evolutionary algorithms for constrained parameter optimization problems, *Evolutionary computation*. **4** (1), pp. 1–32,.

[13] C.A. Coello Coello (2016), Constraint-handling techniques used with evolutionary algorithms. Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion, pp. 563–587.
[14] C. Duclos-Prévet, F. Guéna, and M. Efron (2021), Constrained Multi-Criteria Optimization for Integrated Design in Professional Practice. Gomez, P and Braida, F (eds.), *Designing Possibilities - Proceedings of the XXV International Conference of the Ibero-American Society of Digital Graphics (SIGraDi 2021)*, Online, 8-12 November 2021, pp. 29–40. *CUMINCAD*.
[15] S. Salcedo-Sanz (2009), A survey of repair methods used as constraint handling techniques in evolutionary algorithms, *Computer science review*, **3** (3), pp. 175–192.
[16] A. Smith, D. Coit, T. Bäck, D. Fogel, and Z. Michalewicz (1998), Penalty Functions.
[17] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan (2002), A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE transactions on evolutionary computation*, **6** (2), pp. 182–197.
[18] D. Orvosh and L. Davis (1993), Shall We Repair? Genetic Algorithms Combinatorial Optimization and Feasibility Constraints, Presented at the Proceedings of the 5th International Conference on Genetic Algorithms.
[19] G. Stiny: Shape Rules (1994), Closure, Continuity, and Emergence,, *Environ Plann B Plann Des*, **21** (7), pp. S49–S78.
[20] V. Singh and N. Gu (2012), Towards an integrated generative design framework,, *Design Studies*, **33** (2), pp. 185–207,.
[21] J. Ferber (1997), Les systèmes multi-agents: un aperçu général, *Techniques et sciences informatiques*, **16**(8).
[22] J. Von Neumann and A.W. Burks (1966), Theory of self-reproducing automata *IEEE Transactions on Neural Networks*, **5** (1), pp. 3–14.
[23] G. Beni (2004), From swarm intelligence to swarm robotics, *International Workshop on Swarm Robotics*, pp. 1–9. *Springer*.
[24] R. Hu, Z. Huang, Y. Tang, O. Van Kaick, H. Zhang, and H. Huang (2020), "Graph2Plan: learning floorplan generation from layout graphs", *ACM Trans. Graph.* **39**(4).
[25] H. Zheng and P.F. Yuan (2021), A generative architectural and urban design method through artificial neural networks, *Building and Environment*, **205**, pp. 108-178.
[26] C.M. Macal (2016), Everything you need to know about agent-based modelling and simulation *Journal of Simulation,* **10** (2), pp. 144–156.
[27] C.M. Macal and M.J. North (2005), Tutorial on agent-based modeling and simulation, *Proceedings of the Winter Simulation Conference*, p. 14.
[28] M.S. Roudsari, M. Pak, and A. Smith (2013), Ladybug: a parametric environmental plugin for grasshopper to help designers create an environmentally-conscious design, *Proceedings of the 13th international IBPSA conference* held in Lyon, France Aug. pp. 3128–3135.
[29] S.B. Parascho (2013), Design Tools for Integrative Planning Stouffs, Rudi and Sariyildiz, Sevil (eds.), Computation and Performance – *Proceedings of the 31st eCAADe Conference*, **2**, Faculty of Architecture, Delft University of Technology, Delft, The Netherlands, 18-20 September 2013, pp. 237-246. *CUMINCAD*.
[30] P.J.R. and R.K. Veloso (2019), Multi-agent space planning: a literature review (2008-2017). Ji-Hyun Lee (Eds.) "Hello, Culture!" [18th International Conference, CAAD Futures 2019, Proceedings / ISBN 978-89-89453-05-5] Daejeon, Korea, pp. 52-74. *CUMINCAD*.
[31] R. Koenig and G. Schmitt (2016), Backcasting and a New Way of Command in Computational Design, presented at the CAADence in Architecture.

[32] J. McCormack, A. Dorin, and T. Innocent (2004), Generative Design: A Paradigm for Design Research, *DRS Biennial Conference Series*..
[33]C. Duclos-Prévet, F. Guéna, and M. Efron (2022), Algorithme génétique ou automate cellulaire : le cas d'une optimisation multicritère sous contraintes pour la conception d'une enveloppe, SHS Web of Conferences. EDP Sciences.