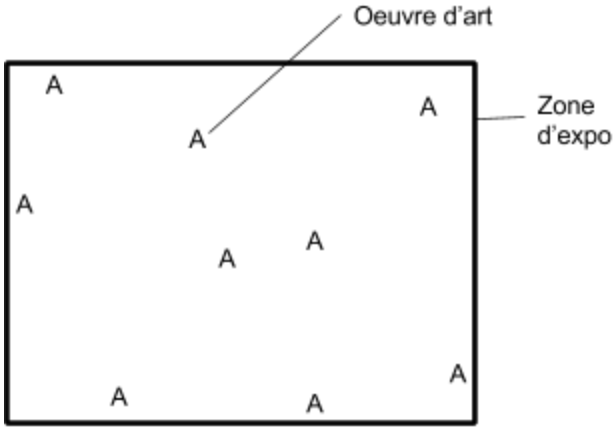
DM2 Optimisation : Exposition au Musée

Charles Jacquet, Elodie Ikkache

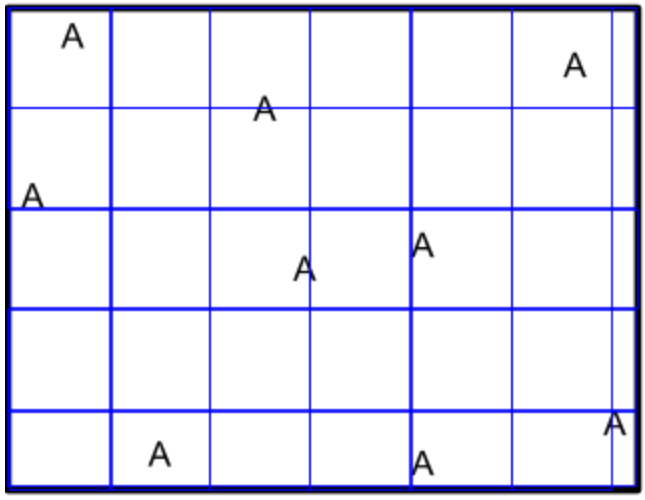
I. Approche 1 : Programmation linéaire à variables entières avec SCIP

Modélisation du problème, choix des variables:

La fonction **create_context** nous permet de récupérer la position des oeuvres d'art, les caractéristiques des caméras ainsi que les dimensions de la zone où se situent les oeuvres d'art.



La fonction **solve** permet de résoudre le problème à partir d'un choix d'une taille de grain. En effet, nous avons choisi de diviser la zone de l'exposition en un grille dont les cases sont de taille taille_grain^2 (qui a pour valeur par défaut 1). On ajoute au modèle deux variables par case : une pour la petite caméra (notée p_{ij} dans la suite du rapport), la seconde pour la grande caméra (notée g_{ij} dans la suite du rapport). La variable vaut 1 si on place une caméra de ce type à cet emplacement, 0 sinon. C'est un problème linéaire à variables entières.

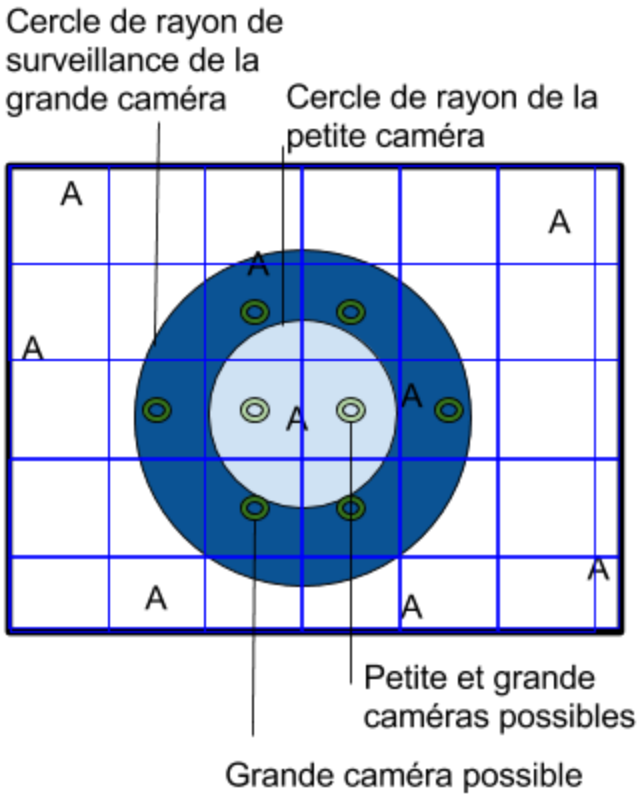


Les contraintes:

La première contrainte est qu'il ne peut pas y avoir deux caméra au même endroit. Donc pour chaque case, $0 <= p_{ij} + g_{ij} <= 0$.

La seconde contrainte est que chaque oeuvre d'art doit être surveillée. Pour cela, on regarde pour chaque oeuvre d'art quels sont les emplacements possibles pour les caméras. L'un de ces emplacements au moin doit avoir effectivement une caméra :

$Somme_{petite\_zone}(p_{ij}) + Somme_{grande\_zone}(g_{ij}) >= 1$



Minimiser le coût des caméras:

Le but étant de trouver une configuration de caméra le moins cher possible, on cherche à minimiser $prix_{petite\_caméra} * Somme(p_{ij}) + prix_{grande\_caméra} * Somme(g_{ij})$

La solution:

La **solve** permet d'ajouter toutes les variables et toutes les contraintes au modèle, ainsi que l'objectif puis de faire appel à la fonction optimize du modèle pour trouver une solution. Si cette dernière existe (cela dépend de la taille de grain choisi), alors solve crée un fichier texte contenant les coordonnées de toutes les caméras à positionner dans la salle, ainsi que la valeur minimum de la function.

II. Approche 2 : Recherche locale

La recherche locale consiste à partir d'une solution connue, pas optimisée, et à se déplacer de solution proche en solution proche vers une solution plus optimale. La recherche s'arrête soit lorsqu'une solution optimale est obtenue, soit au bout d'un certain nombre d'itérations.

On utilise la même fonction **create_context** afin de lire le ficher et récupérer les données.

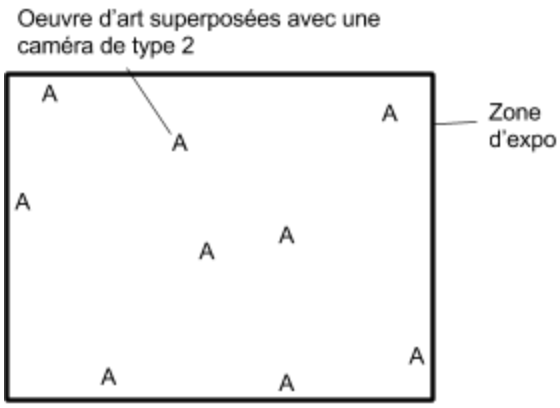
Solutions de départ:

La fonction **solve** implémente la recherche locale.

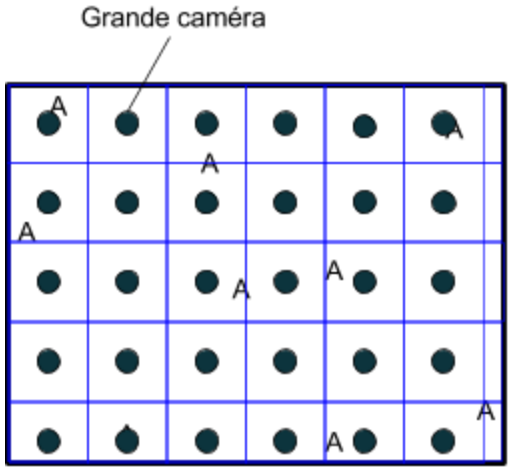
Pour cette approche, nous avons codé une technique d'optimisation avec la possibilité de choisir entre deux solutions initiales:

- La première consiste à placer une grande caméra par oeuvre d'art.
- La seconde consiste à placer des grandes caméras selon une grille comme dans l'approche par contraintes.

L'avantage de la première solution est que nous sommes certains que c'est réellement une solution au problème, tandis que la seconde peut en réalité ne pas couvrir toutes les oeuvres d'art, en fonction de leur disposition et du rayon de portée de la caméra.



Solution 1 : une caméra par oeuvre,



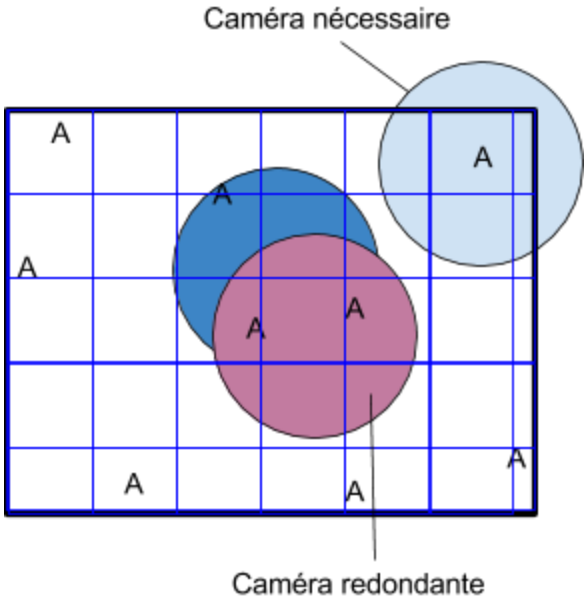
Solution 2 : une caméra par case de la grille

Recherche locale:

Tant que les itérations permettent de réduire le nombre de caméras, (jusqu'à 10 itérations autorisées), on cherche à réduire le nombre de caméras en se servant du fait que leurs zones de surveillance se chevauchent.

Ainsi, si une caméra CAM est la seule à surveiller une oeuvre, alors toutes les autres caméras n'ont plus à leur charge la surveillance des oeuvres dans le rayon d'action de CAM.

De plus, si l'ensemble des oeuvres surveillées par une caméra CAM sont aussi surveillées par une seconde caméra, alors on supprime CAM.



On traite ainsi toutes les caméras "en marche" pour en éliminer un maximum. Tant qu'on en enlève, on recommence à parcourir l'ensemble des caméras.

Enfin, une fois cette étape finie, on parcourt l'ensemble des caméras. Si pour une caméra, les oeuvres dont elle est en charge se situent toutes dans le rayon d'une petite caméra, alors la caméra passe du type 2 au type 1.

Il est possible (sur un set moins étendu que input_9) de visualiser la pièce avec les oeuvres, les caméras ainsi que les zones de surveillance grâce à la fonction **display_gallery**.

III. Comparaison des performances

	Programmation par contraintes	Recherche locale 1: *une grande caméra par oeuvre d'art*	Recherche locale 2: *placer des grandes caméras à équidistance les unes des autres*
Nombre de variables	$2 * 800^2 = 1{,}280{,}000$	5000	$(800 * \frac{1}{5})^2 = 25{,}600$
Temps de calcul	~30 minutes	~ 9 minutes	~ 10 minutes

Résultat (nombre de caméras)	565 grandes + 1,550 petites	2,092 grandes + 497 petites	3,226 grandes + 5 petites
Résultat (coût)	2680	4681	6537

La meilleure solution reste la programmation linéaire à variables entières. Elles est certes plus lente et nécessite beaucoup plus de mémoire, mais elle donne le coût le plus bas.

NB: nous avons codé un modèle pour solveur linéaire capable d'accepter des fractions en tant que pas (0.5 ou 0.25) , mais n'avions pas d'ordinateurs assez puissants pour le faire tourner en temps raisonnable.

Photo de notre solution avec un pas de 1