Ici, mini-problèmes : algorithme

1 3 cas

1. Pb orienté optimisation et satisfaction des contraintes
2. Pb orienté "simulation"
3. Pb orienté "Apprentissag et réseaux de neurones"

2 Problème orienté optimisation

2.1 Pb n reines

```
Input : Tableau n×n TAB a valeurs dans nombres entiers supérieurs ou égaux à 0
    TAB[i,j] ...

Output : Un placement de n "reines" sur l'échiquier (marquage de n cases)

   tant que on a pas 2 cases marquées sur une même ligne, colonne, diagonales
    l'argent correspondant aux cases marquées soit maximal
```

Étape 1 Établier une spécification formelle via un modèle linéaire (PLNE). **modèle linéaire** : calculer un vecteur Z a valeurs dans des nombres (flottants ou entiers). tq (1) pour i dans J > 1..N, Z, doit etre entier (2) pour i dans 1..K, Σ Ak,i, , Zi ?? Bk (3) Maximiser Σ Ci Zi

```
Input :
    N
    J <= (1..N)
    Ak, k = 1..K
    Bk, k = 1..K
    Ci, i = 1..N
```

Ce qu'il faut savoir : * Les modèles linéaires → Beaucoup d'effort des editeurs de logiciels - Bibliothèque dédiées IBM (OSL, CPLEX), XPRESS, GUROBI - Bibliothèques de calcul généraliste MATLAB, SAS, … - Logiciels de gestion d'entreprises : SAP - Langages : CHIP, Prolog, CP-optimizer * Ces bibliothèques peuvent traiter des modèles avec => 10^5 variables ou contraintes * Rôle renctral en Algo car Beaucoup de Problèmes pratiques peuent être exprimés via ce formalisme

Le cas des n reines Vecteurs inconnu
Z indexe sur les cases de l'echiquier
Z=(Zi,j , i = 1..N, j = 1..N) Entier scrictement entre 0 et 1 (exclues)…
Sémantique : Zi,j = 1 ~ on lace une reine en case (i, j) 0 ~ on ne places pas

Quantité a maximum (critere qualité)

```
ΣTAB[i,j] * Zi,j
i = 1..N
j = 1..N
```

Contraintes

1. je veux exactement n reines Σi,j Zi,j = n
2. Pas + de 1 reine par ligne Pour toute ligne j = 1..N Σ Zi,j =< 1
3. Pas + de 1 reine par colonne Pour toute colonne i = 1..N Σ Zi,j =< 1
4. Pas + de 1 reine pour une diagonale montante, 2n-1, diagonales montante

Pour tout k = -(n-1)..(n-1)
Σ Zi,i+k =< 1 i tq 1 <= i <= n i tq 1 <= i+k <= n
2 types d'algos : - *exact* : Realise exactement la spécification - *approché* : Induisent une marge d'erreur

Prendre en compte temps d'exec et critère de qualité de l'approximation dans le cas d'un algo approché (également la robustesse).

Approché

Remarque : Il faut préciser "appoche" : - J'impose vraiment *n-reines* (sinon échec) - J'accepte moins de n-reine → maximiser l'argent donc pénalités (n - nbres de reines placées)

```
not stop;
tant que not stop faire
    choisir une case libre;
    si echec(choisir)
        alors stop;
    sinon
        placer une reine sur la case choisie;
        déduire le plus de choses possibles;
        mettre à jour stop;
```

→ algo glouton avec propagation de contraintes

Déduire des reines imposées, interdites

Exercice : Écrire algorithme pour réalistion ces déductions imposé/interdit → SDD, Algo (on ne prend pas en compte le gains ici)

- Un tableau deux dimensions i,j où :
  - 0 case interdite
  - -1 case libre
  - 1 case occupée

- Un tableau compteur ligne, en ligne j nbe de case libre
- statut ligne, statutligne[1] = 1, il y a une reine en ligne j
- Idem colonnes :
  - compteur colonnes
  - statut colonnes
- Liste de reines imposées (fait déclencheurs en propagation de contraintes) :
  - couples (i,j) correspond à des cases où une reine est imposée
  - ehec, booleen diagnostiquant les impossibiliftés

Algo :

```
Echec ← 0;
LISTE ← case qi'on vient de choisir dans la boucle principale de l'algo;
Tant que (Not Échec) && (Liste != Nil) Faire
|   (i,j) ← Tête(LISTE);
|   LISTE ← Queue(LISTE);
|  Pour toute case (i',j') telle que (i = i') || (j = j') || [(i+j) = (i'+j')] || [(i-j)=(i'-j')]
|   Si occupé[i',j'] = 1 alors Échec
|   Sinon
|   |   Si (occupé[i',j'] = -1) alors
|   |   |   occupé[i',j'] = 0;
|   |   |   compteur_lignes[i'] ← compteur_lignes[i]-1
|   |   |   compteur_colonnes[j'] ← compteur_colonnes[j]-1
|  |  |  Si (compteur_lignes[i] = 0) && (statut_lignes[i'] = 0) alors ÉCHEC;
|   |   |   Sinon
|  |  |  |  Si (compteur_lignes[i'] = 1) && (statut_lignes[0] = 0)
|   |   |   |   |   soit j0 ← unique j tq occupé [i',j] = 0;
|   |   |   |   |   occupé[i',j0] ← 1;
|   |   |   |   |   LISTE ← (i',  j0).LISTE;
|   |   |   |   |   statut_lignes[i'] = 1;
|   |   |   |   |   statut_colonnes[j0] = 1;
|   |   |   |   Fin Si
|   |   |   Fin Sinon
|   |   Fin si
|   Fi Sinon
Fin Tant que
```

Remarque : le schéma peut échouer parce qu'il ne trouve pas une solution estimée acceptable. Il peut aussi renvoyer une solution de qualité médiocre.

2 pistes d'améliorations :

1. On garde le schéma glouton et on le rend non déterministe (“randomize”), de façon à pouvoir l'executer plusieurs fois de suite. Fixe le nb de N “réplications”, pour i = 1..N faire Executer l'algo glouton (non déterministe); récuperer le “meilleur” résultat obtenu; Question : comment est-ce que je rends mon algo “non déterministe” ? Algo courant (deterministe) Itération courante : Cases imposées interdites, Libres (Random avec proba si prob supérieure premier sinon deuxieme meilleur)

2. À l'issue de l'execution de l'algo glouton, on récupère une solution REINE (ex : vecteur indexe sur les cases et à valeurs en [0,1]) On essaie alors d'ameliorer cette solution en lui appliquant des opérateurs de “Transformation locale” (local search)

(…)
Schéma GRASP

```
Pour i = 1..N (Replication) Faire
    Creer une solution REINE via
    la procédure gloutonne "randomizée";
    not stop;
    tant que not stop;
     Appliquer a reine l'opérateur 0 pour une valeur ad hoc de parametre;
        mettre à jour stop;
    Consinier le meilleur objet REINE obtenu;
```

Pour mettre en oeuvre ce schema, il me faut définir le (ou les)opératur 0 + la façon d'aleer chercher les bonnes valeurs de paramètres

Ici difficultés (liée au fait qu'il y a beaucoup de contrainessur l'obet cherché) → j'ai du mal à définir un opératuer 0 qui s'applique à un placement REINE satisfaisant les contraintes et qui le maintienne dans les contraines

Question améliorer ?

1ère Approche, replication

2° approche, on applique sur l'objet REINE produit par l'algo glouton une boucle dite de transformation locale (“local search”) :

```
not stop;
tant que not stop faire
    perturber REINE;
```

Cette 2° approche repose sur le design “d'operateurs”, c'est à dire de prcédures `TRANSFO(Reine, )` Reine : adresse, : valeur.

Opérateurs génériques `Build/Destroy`

- J'enleve p reines parmi les n reines placées (p ~ x% · n)
- Je me retrouve avec q = n-p (ou un peu moins) reines placées; J'applique la propagation de contraintes de ces q reines (j'interdis et impose des cases)

- Je relance le procede glouton “randomise” à partir de la situation obtenue

l'objet RBIND transforme de ? de ces 3 étapes
L'opérateur BUILD.DESTROY ainsi défini, prend comme paramètre.
`BUILD.DESTROY(Reine, )`
Questions sous-jacentes

1. Comment je choisis p et les reines de la liste ?
2. Qu'est ce qu'on met derriere “Tant que not stop faire perturber (Reine)” ?

Question 1

```
Fixer les idées →
    n = 100
    L'objet REINE est un vecteur a taille 10000
      À chaque etape p = 20 (20%) → on enlève 20 reines
    Pas possibilité d'enumerer toutes les possibilités
```

1. Possibilité : Enlever les reines qui ont été placées en 1°
2. Possibilité : Enlver les reines faibles (qui portent le moins d'argent)

NB : Il est souhaitable de génerer plusieurs paquets de p reines à faire a les tester tous et selectionner le + approprié.
Je peux mixer les 2 critères
Spécification de la boucle “local search”

```
not stop; solcour ← reine;
tant que not stop faire
    not stop 1;
    Tant que not stop 1 faire
     generer un paquet  de reines à enlever; //  (on peut générer eventuellement tous les paquet
     Tester l'application de BUILD.DESTROY(REINE, ); // Utilise une copie de REINE
        Si OK(Tester) alors
            stop1;
            Appliquer BUILD.DESTROY(Reine, );
        Sinon mettre à jour stop1;
```

Reste à préciser
`OK(Tester)` ~ Dans quelles conditions j'estime que le paquet de reines à retirer justifie l'application de BUILD.DESTROY(Reine,)

1. Approche : OK si l'application de l'operateur ameliore REINE (place plus de reine ou fait gagner + d'argent) (*Descente ou Hill-climbing*)
2. Approche : OK toujours vrai → je genere 1 paquet et j'applique (*Marche aléatoire ou Random Walk*)

→ Approche mixte : Recul simulé, Tabou, Genetique

(…)

Rappel

Exploration en Arbre (méthode exacte) → en largeur, liste de noeud créés : pb + décisions prises, placement de reines ; liste de triplés ((de signe +/-),i,j)

ex : Imposé reine en case (2,3) Interdire en (5,4) → {(+,2,3),(-,5,4)}

Variable de controle de l'exploration en largeur de l'arbre : LISTE : Liste de noeuds

ex : LIST : {{+,2,3},{+,5,4}} ← Noeud 1, {(+,2,3),(-5,4),{(-,2,3)}}

Squelette de l'Algo

```
*SDD* : LISTE,
                REINE_COUR,
                VAL_COUR, (meilleurs objets obtenus)
                OCCUPÉ[n][n] : de 1..n, 0/1/-1,
                libre_col, libre_lig, statut_col, statut_lig

Initialisation
    OCCUPÉ ← 1;
    ------
    LISTE ← {nil}; not stop;
    REINE_COUR ← Indéfini;
    VAL_COUR ← -ϖ

Corps Algorithme
    Tant que (not stop) && (LISTE  nil) faire
        N ← Tete(LISTE);  // Liste  de cases imposées / interdites
        LISTE ← Queue(LISTE);
     Remplir OCCUPÉ, libre_col, ... via l'algo de déduction (prop de contraintes)

        Choisir une case (i0,j0) libre
        Généer les 2 noeuds
            n1 ← (+,i0,j0) · N; n2 ← (-,i0,j0);
        En utilisant une copie a OCCUPÉ, appliquer le precessus de déduction a la décsion (+i0,j
        Si succes, mois,aucune case de libre, on a une vraie solution et on cose sa valeur = vale
            Idem avec N2;
```

Principe : variable la plus contrainte

Question : "Mettre n1 ou n2 dans LISTE"

Est-ce qu'il y a un ordre pour les éléments de LISTE ?

Réponse : On va essayer de noter les noeuds, à l'aide d'un procédé d'estimation optimiste. (Banch / Bound) → LISTE sera alors ordonnée par notes décroissantes

Je calcule une valeur VAL en tenant compte de de certaines contraintes "faciles" et en delaissant les contraintes "difficiles" (lignes, diagonales)

VAL >= La valeur du meilleur placement compatible ? le noeud (Estimation optimiste)

Mecanismes de filtrage induits

- Règle destructive VAL <= VALCOUR |= Couper ~ j'élimine le noeud
- Règle constructive Si la solution associée à VAL satisfait les contraintes difficiles, aors elles se ??

Adaptation de l'algorithme REINE à l'utilisation de Estimation optimiste

LISTE devient une liste de couples (noeud, valeur) 1. Reste identique 2. Si n1 (n2) débouche sur un succès et des cases libres après propagation → je calcule la note VAL1 de n1 (idem pour N2) et j'applique le me filtrage : * Si v1 <= VALCOUR alors Exit n1 * si v1 > VALCOUR, ET le placement induit satisfait toutes les contraintes → Je mets à jour REINE_COUR et Exit n1; * Sinon j'insère (n1, val1) dans liste de façon à garder liste ordonné pour VAl1 décroissant (idem n2)

STOP ? quand la note cal du 1e element dans liste est <= VALCOUR

3 Problème orienté "simulation"Problème orienté simulation

A,B,C postes de travail

Produits (A), (B)

Produit (A), (B) arrivent "aléatoirement" → L_A, L_B, *lois d'arrivées*

On connait des durées de traitement pour (A), (B) sur A,B,C T_A, T_B, TˆC_A, TˆC_B aléatoires

→ On se fait une idée : - Taux de perte - durée d'attente - Taille des buffers à augmenter ?

NB : Ce n'est pas un poste d'assemblage, mais une machine de transformation

Questions à poser * Est-ce qu'on a observé des dysfonctionnements ? * Quels sont les enjeux économiques de ces dysfonctionnements ? * Données quatifiées * stock : S_A = 10 ; S_B = 7 ; S_C = 13 * Dans la simulation faut-il prévoir (Ici non pour tout car pas réel) * Maintenance préventive * set-up (màj de la machine chaque fois qu'elle change de tâche) * Relais liés aux changements d'activités humaines * *Quantifier* les lois L_A, L_B d'arrivées des objets A et B

Plusieurs options - Rythme d'arrivée presque deterministe… - Rythme d'arrivée avec grosses variations * L'intervalle de temps devient une *variable aléatoire* avec sa moyenne, variance, sa loi… - Il peut y avoir des correlations entre ces intervalles, et des correlations entre ces intervalles et les instants. (compliqués)

Avec une loi `L_A ~ Proba qu'un objet A arrive entre 2 instants t et t+1 ~ 0·3` `L_B ~ 0·2` * Idem pour les durées => on va supposier ici T_A ~ Proba que l'objet A étant strictement à l'instant t, il soit fini à l'instant t+1 = 0 · 4…