Questions & réponses

*Vers le projet****: Algorithme linéaire pour la solution n-Queens Completion Problem***

***Grigoryan E.***

***1.Dans les commentaires sur le programme, on parle assez souvent de la matrice de décision, mais dans le programme, il n'y a pas une telle matrice. Quelle est la raison pour ça?***

Réponse. En fait, bien souvent, on peut trouver le terme «matrice de décision», qui est utilisé comme analogue d'un échiquier. Lors de la résolution d'un problème, tous les événements se déroulent sur la matrice de décision. C’est comme une scène de théâtre. En choisissant une ligne pour l'emplacement de la reine ou en recherchant une position libre dans n'importe quelle ligne, nous nous concentrons toujours sur la matrice de décision. Lorsque nous gardons une trace des indices des lignes libres restantes et du nombre de positions libres dans ces lignes, nous nous concentrons également sur la matrice de décision. De la même manière, nous nous concentrons sur la matrice de décision lorsque nous prenons en compte les contraintes diagonales qui sont imposées aux autres cellules libres lorsque la reine est placée dans une cellule. Tous les événements sur l'emplacement de la reine dans la position sélectionnée et les conséquences de ces événements sont enregistrés sur la matrice de décision. Cependant, dans la partie principale du programme, une matrice de décision n'est pas créée et n'est pas utilisée, car cela n'est pas nécessaire. De plus, pour de grandes valeurs de **n**, une grande quantité de RAM serait nécessaire pour créer une matrice de solution (par exemple, pour un échiquier de **106 x 106**, **1012** octets de mémoire devraient être alloués). Ce n'est qu'après avoir placé la plupart des reines sur l'échiquier que nous créons une matrice de solution plus petite, où nous réduisons «par projection» les lignes et colonnes libres restantes. (Par exemple, pour **n** = **106**, au dernier stade de la résolution du problème, une matrice de solution de taille **547 x 547** est créée uniquement après que les **999453** reines sont situées dans des positions arbitraires). Cette approche ne charge pas beaucoup de mémoire, car ce n'est qu'une petite partie de la matrice de solution. Cependant, cela nous donne un avantage dans la formation de l'algorithme et un gain en vitesse de résolution.

***2. Quelle est la signification du niveau de base et comment sont-ils calculés?***

Réponse. L'ensemble possible de branches pour trouver la solution de ce problème peut être divisé en deux sous-ensembles. L'un d'eux est un sous-ensemble des branches qui sont dans l'impasse. Par conséquent, lorsqu'un blocage se produit dans le processus de résolution du problème, nous devons revenir (c'est-à-dire effectuer la procédure Back Tracking (**BT**)) à l'un des niveaux précédents, puis reconstruire la solution, à partir de ce niveau. À cet égard, deux questions se posent:

 1) Combien de niveaux pour revenir en arrière?

 2) À quel niveau précédent devrions-nous revenir?

Ci-après, par niveau, nous entendons le nombre de reines correctement placées, quelle que soit la séquence de leur arrangement. Évidemment, l'exécution de la procédure **BT** est une charge supplémentaire pour le programme:

- pour tous les niveaux de retour, nous devons enregistrer des copies de tous les tableaux et variables nécessaires;

- sur la base des copies stockées, il est nécessaire de restaurer les valeurs des tableaux et variables actifs;

- il faut à nouveau continuer à former une nouvelle solution à partir de ce niveau, dans l'espoir que nous serons «chanceux cette fois».

De toute évidence, la présence de branches bloquées de la recherche est une propriété objective de cette tâche. Et puisque nous sommes obligés de revenir en arrière, nous devons choisir le niveau optimal, étant donné que:

- plus la procédure **BT** est effectuée et plus le niveau est éloigné pour revenir en arrière, plus il faudra de temps pour résoudre le problème;

- plus le niveau de retour est proche de **n**, moins il est probable que les décisions à partir de ce niveau seront productives.

Dans cet algorithme, nous formons trois niveaux pour revenir en arrière si la branche formée de la solution est bloquée: **startEventBound**, **eventBound1** et **eventBound2**. Nous appelons ces niveaux niveaux **de base**. Ici, **startEventBound** est le niveau de base initial, qui correspond au début de la solution du problème. C'est le niveau où, après saisie des données, l'exactitude de la composition a été effectuée et les opérations nécessaires ont été effectuées pour commencer la recherche d'une solution.

*Comment les valeurs de base ont-elles été obtenues?*

Considérez une liste de valeurs de taille d'échiquier pour la modélisation: **n** =(*10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 200, 300, 500, 800, 1000, 3000, 5000, 10000, 30000, 50000, 80000, 105 , 3 ∗ 105 , 5 ∗ 105 , 106 , 3 ∗ 106 , 5 ∗ 106 , 107 , 3 ∗ 107 , 5 ∗ 107 , 8 ∗ 107 , 108* ), qui sera appelée «**liste de base de n valeurs pour les simulations informatiques**». Pour chaque valeur **n** de cette liste, nous effectuons les opérations suivantes:

1.Pour un échiquier de taille **n x n**, nous trouvons la solution du **problème n-Queens** en utilisant uniquement l'algorithme de recherche de solution **randSet & randSet**. En conséquence, avec une forte probabilité, une branche de blocage sera formée, où **k** reines seront correctement placées sur l'échiquier. Nous formerons un large échantillon de ces solutions. Définissez la valeur d'échantillon moyenne (**xMean**) et l'écart type (**xStd**). Définissez la valeur **baseLevel1 = xMean - 3 \* xStd**. Considérez la valeur obtenue de **baseLevel1** comme une valeur approximative de base level 2. Réalisons des calculs similaires pour la liste de base de **n** valeurs et stockons les résultats dans le tableau **baseLevelAr1**.

2. Nous effectuons des calculs similaires pour toute la liste de base de **n** valeurs, mais nous utilisons uniquement l'algorithme **rand & rand** pour résoudre le **n-Queens Problem**. Enregistrez les résultats dans le tableau **baseLevelAr2**. Ainsi, nous obtenons des valeurs approximatives de deux niveaux de base pour la liste de base de **n** valeurs.

3. En outre, il est nécessaire d'optimiser les valeurs obtenues des niveaux de base en changeant leurs valeurs avec un certain pas vers le haut ou vers le bas. Le critère d'optimalité de sélection sera de telles valeurs de **baseLevel1** et **baseLevel2**, dans lesquelles la solution au problème pour un grand échantillon de positif conpositions conduit à un nombre minimum de solutions de faux négatifs et à un temps de comptage moyen minimum. Nous sauvegardons les valeurs des niveaux de base ainsi optimisés pour la liste de base de **n** valeurs dans les tableaux **eventBoundAr1** et **eventBoundAr2**.

4. Effectuez une analyse de régression pour déterminer la dépendance de la valeur de **baseLevel1** par rapport à la valeur de **n**. De même, nous allons effectuer une analyse de régression et établir la dépendance de la valeur de **baseLevel2** sur la valeur de **n**. Ainsi, nous avons l'occasion de déterminer les valeurs des niveaux de base pour une valeur arbitraire de **n**.

Dans le programme de solution **n-Queens Completion Problem**, après avoir entré le tableau de données source, la taille **n** de l'échiquier initial est déterminée, et en fonction de celle-ci, à l'aide des régressions, les valeurs des niveaux de base **eventBound1** et **eventBound2** sont déterminées.

***3. Quelle est la signification des règles de «risque minimal» et de «dommage minimal» utilisées à la dernière étape de la résolution du problème.***

Réponse. Ces deux règles sont d'une importance fondamentale pour le fonctionnement de l'algorithme, en particulier la règle du «risque minimal». Les deux règles sont décrites en détail dans la publication: *Grigoryan E., Linear algorithm for solution n-Queens Completion problem, <https://arxiv.org/abs/1912.05935>*

*La règle du risque minimum*. Supposons que dans le processus de résolution, nous avons placé la plupart des reines sur l'échiquier et qu'il ne reste qu'un petit nombre de rangées libres sur l'échiquier. Si nous déterminons le nombre de positions libres dans chacune des rangées libres restantes, alors il peut s'avérer qu'il y a parmi elles une rangée dans laquelle il ne reste qu'une position libre et toutes les autres positions de cette rangée sont fermées. (Dans le processus de résolution, moins il y a de lignes libres, plus la probabilité d'une situation similaire est grande). Si, à cette étape de la solution, ne sélectionnez pas la ligne dans laquelle il n'y a qu'une seule position libre, mais une autre, alors un tel choix entraînera un grand risque. La raison en est que les restrictions diagonales associées au choix d'une telle position peuvent fermer la seule position libre dans la rangée des risques et conduire ainsi la solution à un blocage. Pour éliminer le risque d'une situation similaire, à chaque étape, nous trouvons une rangée avec un nombre minimum de positions libres, et c'est là que nous choisissons une position pour la reine. Si, dans la liste des lignes libres, il y a deux ou trois lignes avec la même valeur minimale du nombre de positions, alors nous sélectionnons au hasard l'une de ces deux (trois) lignes.

*La règle des dommages minimaux*. Après avoir sélectionné l'indice de ligne, selon la règle du risque minimum, nous devons choisir une position dans cette ligne pour l'emplacement de la reine. S'il ne reste qu'une position dans la ligne, nous la sélectionnons. S'il reste deux positions ou plus dans la ligne, nous sélectionnons alors la position qui, en raison des restrictions diagonales, cause un dommage minimal aux positions libres dans les lignes restantes. S'il s'avère que deux positions dans la ligne sélectionnée causent les mêmes dommages minimaux aux positions restantes, alors l'indice de l'une de ces deux positions est sélectionné au hasard.

***4. Pourquoi le programme tient-il compte du nombre total de cas lorsque la procédure BT est appliquée?***

*Réponse*. Compte tenu du nombre de cas où la procédure **BT** est utilisée (le nombre de cas où la branche de recherche conduit à une impasse), il est important que le programme fonctionne. Dans le processus de résolution de problèmes, tous les cas sont pris en compte lorsque la branche de recherche de la solution est interrompue et qu'un retour est effectué à l'un des niveaux de base. Lorsque la valeur totale (**totSimCount**) dépasse la valeur de seuil spécifiée (**totSimBound**), les calculs sont interrompus et l'achèvement est de nouveau répété. Le nombre maximum de ces répétitions est de **10**. Si, à la suite de ces actions, il n'est pas possible d'obtenir une solution, alors il est décidé que la composition est négative. C'est **totSimCount** qui est l'indicateur critique sur la base duquel une décision est prise d'interrompre les calculs et de les répéter.

Une caractéristique utile du compteur **totSimCount** est qu'il nous permet de déterminer l'efficacité de l'algorithme. Si, lors de l'achèvement, un blocage ne se produit jamais, cela signifie que l'algorithme est correct. Ce sont les solutions uniques dont nous pouvons parler, que l'algorithme «résout le problème non déterministe de manière déterministe». Par conséquent, il est évident que, toutes choses égales par ailleurs, plus le nombre de cas d'application de la procédure **BT** est petit, plus le fonctionnement de cet algorithme est efficace.

***5. Si la composition s'avère négative, le programme affiche un message: «Cette composition ne peut pas être terminée. La probabilité d'erreur d'une telle décision est inférieure à 0,0001. "Comment cette valeur a-t-elle été obtenue?***

*Réponse*. Au début, il est important de noter qu'un tel message pour les compositions négatives n'apparaît que si la taille de l'échiquier est **n < 100**. Dans cet intervalle, en fait, l'erreur de prendre une telle décision est inférieure à **0,0001**. Si **100 <= n < 800**, alors l'erreur de décision que la composition considérée est négative est inférieure à **0,00001**, et pour les valeurs **n > = 800**, l'erreur de décision correspondante est inférieure à **0,000001**.

L'algorithme est construit de telle manière que les solutions **False Positive**, c'est-à-dire que si une décision est reçue, elle ne peut pas être fausse, car le contrôle de l'exactitude de la position sélectionnée est vérifié à chaque étape. Cependant, l'algorithme n'exclut pas la possibilité de l'apparition de solutions **False Negative**, c'est-à-dire des compositions positives, mais le programme, après de nombreuses tentatives, prend une décision erronée et pense que la composition est négative.

*Quelle est la probabilité d'obtenir des solutions False Negative dans ce programme?*

Afin de répondre à cette question, une expérience de calcul assez volumineuse a été réalisée.

1. Considérez une liste de base de différentes tailles d'un échiquier pour les simulations de calcul.

2. Pour chaque valeur **n** de cette liste, nous allons former et enregistrer de grands échantillons de solutions au **n-Queens Problem**.

3. Sur la base de chaque échantillon de solutions aux **problèmes n-Queens**, nous formerons et enregistrerons un grand échantillon de compositions aléatoires. Pour *n = (20, 30, ..., 90, 100, 200, 300, 500, 800, 1000, 3000, 5000)*, la taille de l'échantillon des compositions était égale à **100 000**, et en outre, la taille de l'échantillon diminuait avec l'augmentation **n**.

Évidemment, chaque composition de ce type peut être complétée au moins d'une manière, puisque chaque composition fait partie d'une sorte de solution complète.

4. Examinez chaque échantillon de compositions et exécutez le programme pour terminer chaque composition. Dans le processus d'analyse de chaque échantillon, nous déterminons le nombre total de cas où le programme génère des solutions **faux négatifs**.

5. À la suite de l'analyse de toutes les données, il a été constaté:

a) Le programme résout avec succès le problème de la réalisation de presque toutes les compositions.

b) Dans la plage de valeurs **7 <= n < 100**, le programme n'a pas réussi à terminer certaines compositions et la part de ces solutions **faux négatifs** dans les échantillons considérés était inférieure à **0,0001**.

c) Dans la plage de valeurs **100 <= n < 800**, la part des solutions **faux négatifs** dans les échantillons considérés était inférieure à **0,00001**.

d) Dans la plage de valeurs **n > = 800**, toutes les compositions étaient terminées et il n'y avait pas une seule solution de **faux négatif**. Cependant, il est évident que cela n'exclut pas la possibilité d'apparition de solutions **faux négatifs** avec une augmentation multiple de la taille de l'échantillon. Dans tous les cas, dans la plage de valeurs *n = (800, 1000, 3000, 5000, 10000)*, la valeur de l'erreur de décision que la composition considérée est négative sera inférieure à **0,000001** et, avec une augmentation de la valeur de **n** cette erreur diminuera.

Dans le processus de simulations informatiques, nous avons considéré différentes valeurs limites du nombre total d'applications de la procédure **BT**. La valeur la plus optimale était **1000**. De plus, (ceci est important), si après avoir appliqué la procédure **BT** des milliers de fois la composition ne peut pas être achevée, alors une deuxième tentative est faite pour compléter la composition (depuis le début). Le nombre de ces tentatives est égal à dix. L'augmentation de cette valeur augmentera le temps après lequel le message indiquant que la composition est négative apparaîtra, et la proportion de solutions de **faux négatifs** diminuera légèrement. Si nous diminuons cette valeur, le temps de prise de décision diminuera en conséquence et la part des décisions **faussement négatives** augmentera légèrement.

Ainsi, sur la base de cette série d'expériences de calcul, la règle suivante a été établie: «si, lors de l'analyse d'une composition arbitraire, la procédure **BT** est utilisée **1000** fois et la composition ne peut être complétée, alors cette procédure est répétée à partir du le tout début. Le nombre total de ces calculs répétés est de **10**. Si, à la suite de ces actions, il n'est pas possible d'obtenir une solution, alors une telle composition est considérée comme négative, c'est-à-dire qu'une telle composition ne peut pas être complétée. La probabilité d'erreur lors de la classification d'une composition positive dans un groupe de compositions négatives, c'est-à-dire la probabilité d'une solution **faux négatif** dépend de la taille de l'échiquier **n**:

**7 < n < = 100**, alors la valeur de l'erreur de décision est inférieure à **0,0001**,

**100 < = n < 800**, la valeur de l'erreur de décision est inférieure à **0,00001**,

**n > = 800**, la valeur de l'erreur de décision est inférieure à **0,000001**

De toute évidence, cette erreur ne s'applique qu'aux compositions positives. Toute composition négative, le programme détermine avec une garantie à **100%**.

En sélectionnant **1000** comme nombre maximal autorisé de réapplications de la procédure **BT**, et en répétant cette procédure 10 fois, on forme le taux d'erreur pour les solutions faux négatifs (*0,0001, 0,00001, 0,000001*). Ceci est le résultat d'un compromis entre la vitesse du programme et les valeurs de l'erreur de décision. S'il est nécessaire de réduire la probabilité d'une erreur dans la formation de solutions faussement négatives dans la plage de valeurs **7 < n < = 800**, alors au lieu des valeurs indiquées, nous pouvons envisager un nombre beaucoup plus grand. Ensuite, lors du traitement d'un grand échantillon de compositions arbitraires, seul le temps de traitement des compositions négatives augmentera, et ces compositions positives que le programme classera à tort comme compositions négatives. La vitesse de traitement de toutes les autres compositions restera la même.

Il est important de noter que dans la version précédente du programme sur la base de laquelle la recherche a été menée et les résultats ont été publiés dans arhiv.org, le nombre de solutions répétées était de **3**. Ici, le nombre de re-décisions est **10**. La valeur limite du nombre de réutilisation de la procédure **BT** reste la même et est égale à **1000**.

***6. Comment l'indice d'événements est-il déterminé dans le programme?***

*Réponse*. L'index des événements dépend de plusieurs indicateurs:

- taille de l'échiquier (**n**),

- taille de la composition (**k**),

- du résultat de la comparaison de ces valeurs avec:

a) valeurs fixes **nFix1 = 50** et **nFix2 = 100**,

b) et les valeurs limites **eventBound1** et **eventBound2**, qui sont calculées sur la base de l'équation de régression pour une valeur donnée de **n**.

Pour plus de clarté de présentation, nous sélectionnons trois sections sur l'axe numérique de la série naturelle

**1**  **2** **3**

- -! - - - - - - - - - - - - !- - - - - - - - - - - - - - !- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

! ! !

**7** **50** **100**

1.  **7 < = n < 50**, **eventInd = 4**

Ici, les calculs partent du 4ème bloc préparatoire et sont ensuite effectués au bloc 5.

2. **50 < = n < 100**, **eventInd = 2**

Ici, les calculs commencent à partir du 2ème bloc préparatoire puis sont effectués dans les blocs 3, 4, 5

3. **n > = 100**,

Ici, le choix de la valeur de l'indice d'événement dépend du rapport entre la taille de la composition (k) et les valeurs limites des blocs d'événements eventBound1 et eventBound2:

a) si **k < eventBound1**, alors **eventInd = 1**;

b) si **eventBound1 < = k < eventBound2**, alors **eventInd = 2**;

c) si **k > eventBound2** alors **eventInd = 4**.

On peut noter que l'indice d'événement ne prend pas les valeurs **3** ou **5**. Cela est dû au fait que pour passer au bloc **3**, il faut d'abord effectuer un petit nombre d'opérations préparatoires au bloc **2**, puis , faites une transition vers le bloc **3**. Une situation similaire avec un tas de blocs **4** et **5**. Ici, comme dans le cas précédent, avant de passer au bloc **5**, il est nécessaire d'effectuer des opérations préparatoires au bloc **4**.

***7. Cet algorithme peut-il être utilisé pour des valeurs n> 100 000 000?***

*Réponse*. Au cours de la recherche, l'algorithme a été développé pour une large gamme de valeurs de la taille de l'échiquier (**n**): de **7** à **100 millions**. En fait, la limite supérieure de l'intervalle indiqué ici n'a pas d'importance fondamentale. Il a été choisi uniquement pour la commodité de la modélisation, en tenant compte des coûts de temps associés à la formation et à l'analyse d'échantillons pour de grandes valeurs de **n**. Pour chaque utilisateur, la valeur de la limite supérieure ne peut être limitée que par la taille de la **RAM** de l'ordinateur. Sur un ordinateur avec **32** Go de **RAM**, les calculs ont été effectués pour un échiquier de **n = 800 millions** ou plus.

Sur la limite inférieure de l'intervalle. Bien que la valeur 7 soit indiquée comme la limite inférieure de l'intervalle, ceux qui aiment «appuyer sur le bouton rouge» peuvent effectuer des calculs pour les valeurs **n = (5, 6)**, le programme autorise de telles valeurs.

***8. Comment évaluer l'efficacité de l'algorithme obtenu?***

*Réponse*. Voici quelques indicateurs importants utilisés pour évaluer l'efficacité de tout algorithme:

1. L'algorithme doit fonctionner rapidement, c'est-à-dire que le temps de résolution du problème doit être minimal.

2. Il est important que l'algorithme soit linéaire en temps **O (n)** (ou proche de cela).

3. L'algorithme doit fonctionner de manière fiable dans un large éventail de modifications des paramètres de base du problème à résoudre.

4. L'algorithme doit être conçu de manière à fonctionner «librement» dans une quantité raisonnable de RAM (par exemple, 32 Go), c'est-à-dire qu'aucune opération supplémentaire ne doit être associée à un manque de RAM.

5. Tout ce qui est nécessaire pour résoudre le problème, l'algorithme "doit se trouver", sans nécessiter d'actions supplémentaires de la part de l'utilisateur.

Ce sont des caractéristiques qui sont évidentes pour chaque programmeur et qui peuvent être complétées par autre chose. Cependant, ce sont des attributs externes du programme. Il est important pour nous de déterminer le critère interne de fonctionnement de l'algorithme qui le rend efficace. Dans les problèmes non déterministes, l'un des critères importants est le nombre de cas d'utilisation de la procédure **Back Tracking** (**BT**) dans le processus de résolution de problèmes. Moins il y a de procédures **BT**, mieux c'est. Ce serait idéal si la procédure **BT** n'était jamais utilisée du tout pendant le processus de résolution. Cela signifie qu'en choisissant séquentiellement une ligne libre et une position libre dans cette ligne, nous atteignons l'objectif sans jamais faire d'erreur critique - toutes les étapes effectuées sont correctes. En ce sens, l'algorithme proposé est assez efficace. Par exemple, pour **n = 1000**, dans l'expérience où un million de compositions ont été assemblées, dans **60,5 0**% des cas, la procédure **BT** n'a jamais été utilisée, et dans les solutions restantes - dans **22,72**% des cas, la procédure **BT** n'a été utilisée qu'une seule fois, dans **9,21**% des cas, la procédure **BT** a été utilisée deux fois et dans **3,95**% des cas, la procédure **BT** a été utilisée trois fois. Ensemble, cela représente **96,38**%. Le reste, soit **3,62**%, a utilisé un nombre différent de procédures **BT**.

Les performances décrites de l'algorithme ne sont pas uniquement liées à la valeur **n = 1000**. Un modèle similaire est typique pour une large plage de **n** valeurs. Pour cette raison, la vitesse de cet algorithme est assez élevée. Par exemple, pour la valeur considérée **n = 1000**, le temps d'exécution moyen pour une composition arbitraire est de **0,035 seconde**. Et comme mentionné ci-dessus, sur un ordinateur avec **32** Go de RAM, l'algorithme permet des calculs pour un échiquier de **800 millions** ou plus. Par conséquent, en général, selon les critères ci-dessus, l'algorithme peut être considéré comme assez efficace.