

Quadratic Assignement Problem

rapport d’optimisation discrète

Antoine Ganne | Optimisation discrète | 2019

# Instances de Taillard

Les instances de Taillard qui nous sont données se décomposent en 3 éléments :

* Dimension (un entier n compris entre 12 et 100)
* Matrice décrivant w la fonction de poids définie entre les équipements
* Matrice décrivant d la distance entre les emplacements

On a aussi la solution optimale pour certains jeu de données.

On considérera les matrices de poids et de distance comme étant symétriques, ce qui est le cas pour le jeu de donnée que l’on utilise.

# Choix de nommage et définitions

Dans ce projet j’appelle configuration ou solution, les différentes façons possibles d’affecter n équipements sur n emplacements. Je représente une configuration par un tableau d’entier. Par exemple {2,3,1} indique que la machine 2 est en position 1, que la machine 3 est en position 2 et que la machine 1 est en position 3.

Une configuration est correcte si et seulement si chaque machine y est présente une seule fois et le numéro de chaque machine est bien compris entre 1 et n (inclus) ;

Soit C l’ensemble des configurations correctes, soit

* \*

# Voisinages

On commence par prendre une transformation élémentaire simple : la permutation de l’emplacement de deux équipements.

Ce choix de voisinage me semble intéressant car il permet d’assurer que les voisins d’un configuration correcte sont aussi corrects.

On note que l’opération inverse d’une permutation correspond à la même permutation. Cela est utile pour la méthode Tabou.

Concernant la notation, « (i-j) » correspond à la permutation des éléments à la positions i avec la position j.

Le nombre de voisins est , il est possible que cela soit un nombre trop important lorsque n est grand : .

C’est pour cela que je testerai de limiter le nombre de voisins en introduisant les permutations limitées : une permutation entre i et i+1. Cela permettra de réduire le nombre de voisins à n.

# Paramètres initiaux des algorithmes

Les paramètre pour l’algorithme de recuit simulé sont les suivants :

* NB\_Steps : Nombre d’itérations (steps)
* T0 : Température initiale
* Mu  : valeur par laquelle la température est multipliée à chaque itération
* P  : probabilité servant à définir la température initiale

Je calcule T0 grâce a la valeur P donnée. Je sélectionne 3 configurations aléatoire, je prends la plus grosse différence de fitness entre une configuration et ses voisins. Enfin je calcule la température initiale de sorte a ce que la pire différence de fitness ait une probabilité p d’être acceptée par l’algorithme.

Les paramètres pour la méthode Tabou sont les suivants :

* NB\_Steps : Nombre d’itérations (steps)
* TabouLength : taille de la liste Tabou

La solution initiale est sélectionnée aléatoirement.

# Implémentation

J’ai décidé de coder le projet d’une manière à pouvoir facilement changer les opérations élémentaires et le landscape du modèle (grâce aux interfaces Landscape et ElementaryOperation).

J’ai implémenté la marche aléatoire, le recuit simulé et la méthode Tabou.

Vous pouvez exécuter les algorithmes en exécutant le jar a la racine du projet. La commande est :

java -jar Main.jar

## Vérification de l’implémentation

Pour vérifier le comportement correct, j’ai testé les algorithmes sur un problème très simple, de dimension 3

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Poids(weight)   |  |  |  | | --- | --- | --- | | 0 | 1 | 2 | | 1 | 0 | 1 | | 2 | 1 | 0 | | Distances (dist)   |  |  |  | | --- | --- | --- | | 0 | 3 | 1 | | 3 | 0 | 2 | | 1 | 2 | 0 | |

Ce qui correspond à ce schéma de l’ensemble des configurations et de leur voisinage :

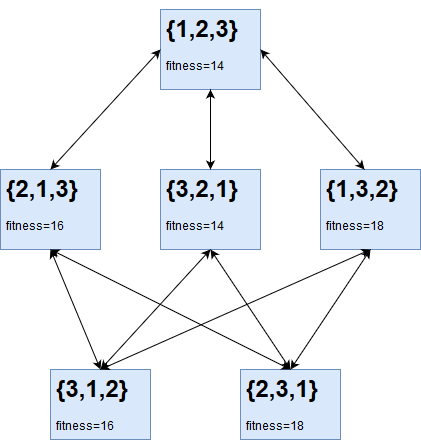


Figure : schéma de l'instance de test

Dans ces tests les algorithmes commenceront sur la configuration (1,2,3).

Le détail de ces tests se trouve sur classeursExcel/Verification\_implementation.xlsx

### Recuit simulé

Ainsi voici les résultats pour l’algorithme de recuit simulé avec les paramètres mu=0.8 , p=0.7 et NB\_Steps =30 :

|  |  |  |
| --- | --- | --- |
| Step | Configuration | Fitness |
| 0 | Config{ 1, 2, 3 ) | 14 |
| 1 | Config{ 1, 3, 2 ) | 18 |
| 2 | Config{ 3, 1, 2 ) | 16 |
| 3 | Config{ 2, 1, 3 ) | 16 |
| 4 | Config{ 2, 3, 1 ) | 18 |
| 5 | Config{ 1, 3, 2 ) | 18 |
| 6 | Config{ 1, 2, 3 ) | 14 |
| 7 | Config{ 2, 1, 3 ) | 16 |
| 8 | Config{ 3, 1, 2 ) | 16 |
| 9 | Config{ 3, 2, 1 ) | 14 |
| 10 | Config{ 3, 2, 1 ) | 14 |
| 11 | Config{ 3, 2, 1 ) | 14 |
| 12 | Config{ 3, 2, 1 ) | 14 |
| 13 | Config{ 3, 2, 1 ) | 14 |
| 14 | Config{ 3, 2, 1 ) | 14 |
|  | … |  |

On observe que l’on parcoure des configurations avec une mauvaise fitness lors des premières itérations et que rapidement l’algorithme se stabilise sur les configurations avec les meilleures fitness (1,2,3) et (3,2,1).

C’est le déroulement attendu du recuit simulé alors je considère que mon implémentation est correcte.

### Méthode Tabou

Voici les résultats de la méthode Tabou sur l’instance de test avec TabouLength=1.

|  |  |  |  |
| --- | --- | --- | --- |
| Step | Configuration | Fitness | Tabous |
| 0 | Config{ 1, 2, 3 ) | 14 |  |
| 1 | Config{ 3, 2, 1 ) | 14 | (0-2) |
| 2 | Config{ 3, 1, 2 ) | 16 | (1-2) |
| 3 | Config{ 2, 1, 3 ) | 16 | (0-2) |
| 4 | Config{ 1, 2, 3 ) | 14 | (0-2) |
| 5 | Config{ 2, 1, 3 ) | 16 | (0-1) |
| 6 | Config{ 3, 1, 2 ) | 16 | (0-2) |
| 7 | Config{ 3, 2, 1 ) | 14 | (0-2) |
| 8 | Config{ 3, 1, 2 ) | 16 | (1-2) |
| 9 | Config{ 2, 1, 3 ) | 16 | (0-2) |
| 10 | Config{ 1, 2, 3 ) | 14 | (0-2) |
| 11 | Config{ 2, 1, 3 ) | 16 | (0-1) |
| 12 | Config{ 3, 1, 2 ) | 16 | (0-2) |
| 13 | Config{ 3, 2, 1 ) | 14 | (0-2) |
| 14 | Config{ 3, 1, 2 ) | 16 | (1-2) |
| 15 | Config{ 2, 1, 3 ) | 16 | (0-2) |
| 16 | Config{ 1, 2, 3 ) | 14 | (0-2) |
|  | … |  |  |

On observe qu’à chaque itération l’algorithme se dirige bien au voisin non interdit par la liste Tabou qui as la meilleure fitness. On observe que l’algorithme se retrouve rapidement dans une boucle.

C’est le comportement attendu de la méthode Tabou donc je considère mon implémentation correcte.

### Marche aléatoire

Afin de pouvoir comparer les algorithmes avec un algorithme naïf, j’ai implémenté la marche aléatoire.

|  |  |  |
| --- | --- | --- |
| step | configuration | fitness |
| 0 | Config( 2, 1, 3 ) | 16 |
| 1 | Config( 1, 2, 3 ) | 14 |
| 2 | Config( 1, 3, 2 ) | 18 |
| 3 | Config( 3, 1, 2 ) | 16 |
| 4 | Config( 2, 1, 3 ) | 16 |
| 5 | Config( 2, 3, 1 ) | 18 |
| 6 | Config( 1, 3, 2 ) | 18 |
| 7 | Config( 2, 3, 1 ) | 18 |
| 8 | Config( 3, 2, 1 ) | 14 |
| 9 | Config( 2, 3, 1 ) | 18 |
| 10 | Config( 2, 1, 3 ) | 16 |

Ce fut le plus simple à implémenter et le résultat est cohérent.

# Analyse de la complexité

Dans cette partie, je vais calculer la complexité temporelle des algorithmes. Cette analyse se fera selon les trois paramètres qui me semblent les plus intéressants à analyser, car leur valeur est susceptible de beaucoup évoluer :

* n : la dimension du problème
* NbSteps : le nombre d’itérations maximum
* V : le nombre de voisins de chaque configuration (dépend de n et du landscape choisit)

Je considère que les autres paramètres (tels que mu, TabouLength, etc…) ont une influence négligeable.

On s’intéressera a la complexité dans le pire des cas.

## Calcul de la fitness



Figure : code de calcul de la fitness

Le calcul de la fitness se fait par le biais de deux boucles imbriquées sur n. On note que j’utilise la symétrie des matrices de poids et de distance afin de réduire la complexité (en faisant commencer la seconde boucle à i+1).

On considère que le temps d’accès aux valeurs dans les tableaux d’entiers, la multiplication et l’addition d’entiers se fait en un temps constant selon n.

En posant la fonction f correspondant au temps d’exécution de la fonction de calcul de la fitness en fonction de n :

Je remarque que la complexité est assez médiocre car quadratique, cependant il ne semble pas possible de faire beaucoup mieux.

Je note donc que le calcul de la fitness est une opération couteuse et qu’il faudra limiter au mieux son appel dans les algorithmes.

## Marche aléatoire

La sélection d’une solution initiale aléatoire se fait en O(n). Cependant il n’est appelé qu’une fois donc son impact est négligeable par rapport aux traitements dans la boucle principale.

La boucle principale se fait sur le nombre d’itérations nbSteps. A l’intérieur on effectue les opérations :

* Sélection d’un voisin aléatoire : O(1).
* Calcul de la fitness : f(n)
* Mise à jour de la meilleure solution : O(1)

Donc la complexité de la marche aléatoire est O(nbSteps\*f(n))

## Recuit Simulé

La sélection de la solution initiale a une complexité de O(n).

La boucle principale se fait sur nbSteps. Elle se compose de :

* Sélection d’un voisin aléatoire : O(1)
* Calcul de la fitness du voisin : f(n)
  + Ce calcul ne se fait qu’une fois par itération.
* Traitement des conditions d’acceptation du voisin : O(1)
  + Car ces calculs ne sont pas affectés par n, v ou nbSteps

Ainsi la complexité du Recuit simulé est O(nbSteps\*f(n)), la même complexité que la marche aléatoire ce qui est rassurant.

On note que le calcul de la température initiale a une complexité de O(nbSelected\*f(n)) avec nbSelected le nombre de configuration sélectionnées. Ainsi la complexité y est quadratique mais nbSelected est négligeable face a nbSteps. Ainsi on ne considéra pas la complexité du calcul de la température initiale.

## Méthode Tabou

Pour se placer dans le pire dans cas, on considère que la liste Tabou est toujours vide, on ignore donc les calculs qui ne seront économisés grâce aux opérations interdites.

La sélection de la solution initiale a une complexité de O(n).

La boucle principale se fait sur nbSteps. Elle se compose de :

* Sélection du meilleur voisin : O(V\*f(n))
  + Pour chaque voisin on calcule sa fitness
  + A noter qu’on ne calcule pas la fitness pour les voisins interdits par la liste tabou
* Mise a jour de la liste tabou si besoin : O(1)

Ainsi la complexité temporelle de la méthode Tabou est O(nbSteps\*V\*f(n))

Avec notre voisinage de départ (permutations sur l’ensemble de n), la complexité de la méthode Tabou est alors O(nbSteps). Ce qui est bien pire que la complexité de la marche aléatoire ou du recuit simulé.

On note qu’avec les permutations limitées (entre i et i+1) la complexité s’améliore et passe à O(nbSteps).

## Vérification des temps d’exécution

J’ai exécuté les algorithmes sur toutes les instances de Taillard en mesurant leur temps d’exécution. J’ai pris la moyenne sur 3 tests pour chacune des valeurs.

Dans ces tests, NbSteps est fixé à mille et on utilise le voisinage des permutations complètes (en O()).

Pour le recuit simulé, on utilise mu=0.99 et p=0.9.

Pour la méthode tabou, on utilise TabouLength= n/2.

Les données sont sur le classeur Excel : classeursExcel/Temps\_execution.xlsx

|  |  |  |  |
| --- | --- | --- | --- |
| n \ méthode | Marche aléatoire | Recuit simulé | Méthode Tabou |
| 12 | 10 | 9 | 28 |
| 15 | 6 | 8 | 30 |
| 17 | 6 | 6 | 47 |
| 20 | 6 | 8 | 72 |
| 25 | 7 | 8 | 185 |
| 30 | 6 | 28 | 225 |
| 35 | 8 | 7 | 408 |
| 40 | 6 | 9 | 684 |
| 50 | 8 | 11 | 1484 |
| 60 | 9 | 12 | 2968 |
| 80 | 11 | 19 | 9531 |
| 100 | 15 | 25 | 22743 |

Tableau : temps d'exécution [ms] des algorithmes sur les instances de Taillard de taille n

Afin de mieux comprendre ces données j’ai produit deux graphiques :

D’abord, en s’intéressant aux petites dimensions, on observe que les temps d’exécution de la marche aléatoire et du recuit simulé sont très proches et faibles (inférieures ou égal à 10 ms). La méthode Tabou est quant à elle plus longue dès n=12 avec un temps d’exécution 3 fois supérieur aux autres algorithmes.

Afin de visualiser le temps d’exécution des algorithmes sur des grandes valeurs de n, j’utilise une échelle logarithmique en base 10.

On observe ainsi, sans surprise, que le recuit simulé prend légèrement plus de temps que la marche aléatoire mais que leur évolution selon n est similaire.

On voit aussi que le temps d’exécution de la méthode Tabou explose lorsque n augmente. Pour n=100 la méthode tabou prend 23 secondes alors que Recuit Simulé est à 25 ms, la différence est d’un facteur 1000.

On observe que l’évolution du temps d’exécution de la méthode Tabou est cohérente avec sa complexité en O(.

# Paramétrages du Recuit Simulé

Avant de comparer les deux algorithmes je vais chercher les paramètres optimaux de chaque algorithmes en commençant par le recuit simulé.

## Paramètre p

Pour rappel, le paramètre p sert au calcul de la température initiale. Il correspond a la probabilité d’accepter la pire différence de fitness constatée.

Je cherche le paramètre p qui minimise la fitness moyenne de la solution et le pas de découvert de cette fonction. Je limite ma recherche de p à l’intervalle .

Les autres paramètre sont fixés, On effectue l’analyse sur l’instance de Taillard 40.

Figure : fitness moyenne selon p

On voit que c’est la valeur 0.5 qui minimise la fitness moyenne.

Figure : pas moyen de découverte de la meilleure solution

On voit sur ce graphique que le pas ne dépasse jamais 10000 donc il semble ici que nbSteps pourrait être égal a seulement 10000 sans impact sur la qualité de la solution.

Pour p=0.5 , le pas est acceptable.

Ainsi je prendrai 0.5 pour valeur de p.

## Paramètre mu

De même on s’intéresse au paramètre mu qui correspond au facteur de décroissance de la température au cours des itérations. Mu .

On passe a 50 essais par valeur pour essayer d’avoir des résultats plus significatifs.

Figure : fitness moyenne selon mu

On observe que mu=0.9 est ici la valeur donnant la meilleure fitness.

Figure : pas moyen selon mu

On voit que pour p=0.9 le pas est correcte.

Ainsi on utilisera 0.9 comme valeur de mu.

## Le paramètre nbSteps

Pour paramétrer le paramètre nbSteps, j’ai mesuré le pas de découverte de la meilleure solution en me disant que nbSteps devrait être juste légèrement supérieur au pas maximum pour chaque valeur de n.

J’ai effectué des tests avec nbSteps fixé à cinq-cents mille, un chiffre suffisamment grand pour être sûr que cela n’ait pas d’impact. Les autres paramètres sont fixés.

Tableau : pas moyen et maximum sur 5 essais selon n

|  |  |  |
| --- | --- | --- |
| N | Pas moyen | Pas maximum |
| 12 | 215 | 299 |
| 15 | 552 | 1142 |
| 17 | 287 | 388 |
| 20 | 576 | 1100 |
| 25 | 2405 | 3260 |
| 30 | 3398 | 7565 |
| 35 | 3448 | 5240 |
| 40 | 4424 | 6152 |
| 50 | 9002 | 12346 |
| 60 | 25487 | 46072 |
| 80 | 35171 | 42578 |
| 100 | 76821 | 102120 |

Tableau : Pas moyen et pas maximum de découverte de la meilleure solution sur 5 essais selon la dimension n.

Déjà on observe que le pas maximum est toujours bien en dessous de 500000, notre valeur fixée de nbSteps. Donc on peut considérer que cela n’a pas d’impact sur les résultats.

Je vois que le pas maximum n’évolue pas linéairement. Il semble évoluer quadratiquement. Ce qui serait inquiétant pour des valeurs importantes de n. En effet pour n=100, le pas maximum est déjà très important.

Au vu des résultats, je considère que nbSteps=10 000 est suffisant pour n<=40, que nbSteps=100 000 sera nécessaire pour n compris entre 50 et 80 et que j’utiliserai nbSteps=150 000 pour n=100.

# Paramétrage de la méthode Tabou

Le paramétrage de la méthode Tabou s’annonce plus compliqué à cause du temps d’exécution important sur des grandes valeurs de n. Afin d’avoir les tests les plus rapides possible je vais commencer a chercher la plus petite valeur de nbSteps que je puis utiliser.

## Paramètre nbSteps

J’ai effectué mes tests sur nbSteps de la même manière que pour le recuit simulé cependant par soucis de performance j’ai d’abord effectué un test avant seulement un essai par valeur de n et avec un nbSteps moins grand égal à 100 000. On se limite a un essai car l’algorithme est déterministe, a la seule exception de la sélection aléatoire de la solution initiale.

Figure : pas de découverte de la meilleure solution selon n.

On observe que parfois l’algorithme trouve rapidement sa meilleure solution tandis que parfois cela lui prend bien plus d’itérations (notamment pour n=35 et n=80). Les très faibles valeurs du pas de découverte montrent une certaine corrélation avec la dimension n. Cependant l’étonnante valeur du pas de découverte pour n=100 semble montrer que la valeur de n n’est pas le seul facteur a prendre en compte pour choisir le paramètre nbSteps.

J’ai aussi effectué ces mêmes tests sur 5 essais sur des petites valeurs de n. Je veux ainsi savoir si la solution initiale (choisie aléatoirement) à un impact important.

Figure : pas de découverte moyen et maximum de la meilleure solution selon n. Sur 5 essais

Les résultats sont assez surprenants, notamment pour n=35 : la plus grand valeur de pas maximum mais aussi une valeur moyenne assez faible. J’en conclus que la solution initiale sélectionnée a un impact non négligeable.

Concernant le paramétrage de nbSteps, j’en conclus que 100 000 est une valeur acceptable de nbSteps et que, en moyenne, une valeur de 50 000 à 70 000 peut être utilisée.

## Paramétrage de TabouLength

On commence par rappeler que TabouLength doit être compris entre 0 et le nombre d’opération élémentaires. Dans notre cas des permutation, ce nombre est n(n-1) /2 , une valeur assez importante.

J’ai commencé par tester l’effet de différentes valeurs de Taboulength pour n=20.

Figure : fitness moyenne de la meilleure solution selon TabouLength sur 3 essais.

Sans surprise une valeur de TabouLength a zéro donne les pire résultats. Ensuite augmenter la valeur de TabouLength donne de meilleurs résultats mais l’amélioration de la fitness est moins importante pour tabouLength >=20.

Figure : pas moyen de découverte selon TabouLength sur 3 essais.

Concernant le pas moyen de découverte de la meilleure solution. TabouLength ne semble pas avoir un effet particulier. Mis a part que pour des petites valeurs de TabouLength le pas de découverte est petit, ce qui peut s’expliquer par le fait que l’algo se bloque plus rapidement dans une boucle lorsque la taille de la liste Tabou est petite.

Etant donné que TabouLength accepte des grandes valeurs, j’ai voulu tester les effets d’une valeur importante de TabouLength. J’ai fait le test sur n=30, ce qui nous donne une valeur max de TabouLength de 435.

Figure : fitness moyenne de la meilleure solution selon TabouLength (n=30)

On observe clairement que des valeurs très importantes de TabouLength ne permettent pas de trouver de meilleures solutions.

Figure : pas moyen selon TabouLength (n=30)

Concernant le pas moyen, on voit qu’avec une valeur importante de TabouLength, l’algorithme se bloque rapidement dans des boucles avec des solutions médiocres.

Je retiens de ces tests qu’on trouve de meilleures fitness lorsque TabouLength est proche de n. J’utiliserai donc cette valeur.

# Comparaison des algorithmes

Nous arrivons finalement à la comparaison des résultats des différents algorithmes. Pour allons exécuter chaque algorithme 3 fois sur chaque instance de Taillard. On mesure la meilleure fitness trouvée sur les 3 essais, le pas de découverte de cette solution, la solution en elle-même et la fitness moyenne des meilleures solutions trouvées. Afin de se faire une idée de la pertinence des fitness trouvée, je les compare a la fitness des meilleurs solutions connues qui sont disponibles sur [http://anjos.mgi.polymtl.ca/qaplib/inst.html#Ta](http://anjos.mgi.polymtl.ca/qaplib/inst.html%23Ta)

J’applique les paramètres déterminés dans les parties précédentes :

* Marche aléatoire : nbSteps=100 000
* Recuit simulé : mu= 0.9, p=0.5, nbSteps =20 000 pour n<= 50 et nbSteps=100 000 sinon
* Méthode Tabou : TabouLength=n, nbSteps=50 000 pour n<=50 et nbSteps=100 000 . En raison des problèmes de performances, j’ai du limiter nbSteps à 10 000 pour n=100.

Les résultats sont disponibles sur /classeursExcel/ComparaisonAlgorithmes.xlsx

De ces données, le graphique qui me semble le plus pertinent : la fitness proportionnelle à la meilleure fitness connue des différents algorithmes.

Figure : fitness de la meilleure solution trouvée en proportion de différence avec la meilleure fitness connue, selon n et les algorithmes testés.

On voit déjà les pauvres performances de la marche aléatoire, les résultats assez corrects du recuit simulé et les bons résultats de la méthode Tabou. En effet on note que la méthode Tabou trouve la meilleure solution connue pour n=12,15,20 et 30.

On remarque aussi que l’écart entre le recuit simulé et la méthode Tabou se resserre lorsque n augmente. Cela alors que le temps d’exécution de la méthode Tabou est de l’ordre des minutes voire dizaine de minutes.

# Conclusion

Nous arrivons donc à la conclusion de mon analyse des algorithmes de recuit simulé et de la méthode Tabou.

Pour pousser l’analyse il aurait fallu tester d’autres voisinages pour la méthode Tabou, notamment en limitant le nombre de permutation possible. De même j’aurais aimé pousser l’analyse du nombre d’itérations (nbSteps) qui maximise la fitness trouvée tout en minimisant le temps d’exécution, car il semble que passé un certain nombre d’itération le gain de fitness est négligeable comparée au temps d’exécution. J’aurais aussi pu essayer de détecter le moment ou la méthode Tabou se bloque dans un cycle ou le moment où, dans le cadre du recuit simulé, l’algorithme se bloque dans un minimum local.

Cependant j’ai produit une implémentation correcte des différents algorithmes et mon analyse amène tout de même à des conclusions. J’ai pu proposer des paramétrages des algorithmes qui semblent pertinents au vu des résultats.

J’ai finalement pu comparer les algorithmes. Au vu des résultats je peux dire que la méthode Tabou est assurément l’algorithme de choix pour des petites valeurs de n car il est capable de trouver l’optimum en un temps correct. Cependant lorsque n augmente le recuit simulé devient envisageable pour ses bonnes performances (même complexité que la marche aléatoire) et parce qu’il parvient à trouver des résultats corrects, notamment sur des grandes valeurs de n.