

Problème générique de Dijkstra [tr06] - Exercice

Karine Zampieri, Stéphane Rivière

Unisciel  algoprogram  Version 21 mai 2018

Table des matières

1	Problème de Dijkstra : Cas $k = 3$	2
2	Téléchargements – Utilitaires	3
3	Problème générique de Dijkstra / pgdijkstra	5
3.1	Représentation des données	5
3.2	Cas $k=2$	5
3.3	Cas $k=3$: Problème de Dijkstra	8
3.4	Cas $k=4$ et Cas générique	11
4	Références générales	14

C++ - Problème générique de Dijkstra (Solution)



Mots-Clés Algorithmes de tris et rangs ■

Requis Algorithmes de tris et rangs, Complexité des algorithmes ■

Difficulté ●●○ (2 h) ■



Objectif

Cet exercice décrit et analyse le problème générique du drapeau hollandais de E. DIJKSTRA.

1 Problème de Dijkstra : Cas $k = 3$

Le problème de Dijkstra

Vous disposez de cailloux rouges, blancs, bleus alignés dans un ordre quelconque. Par **échanges successifs** de cailloux (et non par constitution de paquets séparés en un autre endroit) et en ne testant qu'**une fois** la couleur de chaque caillou, vous devez mener à une situation finale où tous les cailloux bleus se trouvent avant tous les blancs, eux-mêmes avant tous les rouges. Le nombre des cailloux de chaque couleur est quelconque.

Objectif du problème

Le but à atteindre est la répartition des cailloux en trois ($k = 3$) classes, les seules opérations autorisées étant :

- Faire des échanges successifs.
- Tester la couleur une seule fois par caillou.
- Utiliser l'emplacement initial.

Première formalisation du problème

Elle consiste à considérer l'espace occupé par les cailloux comme une suite T de cases numérotées de 1 à n , une case contenant un seul caillou et étant identifié par $T[j]$ avec $1 \leq j \leq n$.

Problème générique de Dijkstra

Soit T un tableau contenant des cailloux colorés, la couleur étant identifiée à un entier de $[1..k]$ (k couleurs). C'est toujours possible : il suffit d'une fonction de couleur $c(v)$ qui associe à chaque élément de valeur v de T , un entier de $[1..k]$.

Objectif du problème générique

Le but à atteindre est la répartition des cailloux en k classes, les deux seules opérations autorisées étant :

- La permutation de deux éléments du tableau.
- Le test de la couleur d'un élément.

...(suite page suivante)...

2 Téléchargements – Utilitaires

Cet exercice utilise les opérations suivantes, toutes définies dans un bon nombre d'exercices de cet espace thématique :

- Fonction `saisirNombreElements`
- Procédure `afficherTri`
- Procédure `aleatoireTri`
- Procédure `permuterTab`

Elles ont été regroupées dans une bibliothèque.



Définitions C++

```
const int TMAX = ...;
using ITableau = int[TMAX];
```



Fixez la constante `TMAX=50` (nombre maximum d'éléments).



Téléchargez le fichier suivant et mettez-le dans votre dossier.

C++ @[UtilsTR.cpp]



Copiez/collez ensuite les lignes suivantes :

C++ **Au début** de votre programme :

```
#include "UtilsTR.cpp"
using namespace UtilsTR;
```



Soit la fonction `saisirNombreElements(nmax)` qui renvoie le nombre d'éléments, saisi par l'utilisateur, entier compris dans `[1..nmax]`. Elle affiche l'invite :

Nombre d'éléments dans `[1..[nmax]]?`

C++ @[saisirNombreElements] (dans UtilsTR.cpp)



Soit la procédure `afficherTri(t,n,g,h)` qui affiche, à la queue-leu-leu séparés par un espace le tout entre crochet, les `n` premières valeurs d'un `ITableau t`, les indices `g` et `h` indiquant le sous-intervalle du tri et représentés par une barre verticale. La barre de gauche est avant `g` et celle de droite est après `h`. Exemple :

```
afficherTri(t,10,4,9) ==> [1 2 3 |4 5 6 7 8 9 |10]
```

C++ @[afficherTri] (dans UtilsTR.cpp)



Soit la procédure `aleatoireTri(t,n,vmax)` qui initialise les `n` premiers éléments d'un `ITableau t` en utilisant `vmax` comme valeur maximale pour la fonction de génération d'un entier pseudo-aléatoire.

C++ @[aleatoireTri] (dans UtilsTR.cpp)



Soit la procédure `permuterTab(t,j,k)` qui permute les éléments d'indice `j` et `k` d'un `ITableau t`. Les indices sont supposés valides.

C++ @[permuterTab] (dans UtilsTR.cpp)



Remarque

Si la fonction et les procédures n'ont pas été réalisées, il vous est conseillé de la(les) rédiger dans l'exercice @[Utilitaires Tris et Rangs].

...(suite page suivante)...

3 Problème générique de Dijkstra / pgdijkstra

Nous allons étudier les cas particuliers ($k = 2$, $k = 3$ et $k = 4$) puis le cas générique.

3.1 Représentation des données



Définissez les couleurs `CBLEU=1`, `CBLANC=2`, `CROUGE=3` et `CVERT=4`.



Définissez une fonction `couleur(t,k)` qui, pour un indice `k` d'un `Tableau t`, renvoie la couleur

- `CBLEU` si `t[k]=0`
- `CBLANC` si `t[k]=1`
- `CROUGE` si `t[k]=2`
- `CVERT` sinon (cas `t[k]>2`)



Validez vos définitions et votre fonction avec la solution.

Solution C++

@[pgdijkstra.cpp]

```
// Caillou Bleu
const int CBleu = 1;
// Caillou Blanc
const int CBlanc = 2;
// Caillou Rouge
const int CRouge = 3;
// Caillou Vert
const int CVert = 4;
// Nombre maximum de cailloux
const int TMAX = 100;
// Type Tableau de cailloux
using ITableau = int[TMAX];

/**
 * Couleur d'un caillou
 * @param[in] t - un ITableau
 * @param[in] k - indice valide
 * @return la Couleur de t[k]
 */
int couleur(const ITableau& t, int k)
{
    return (t[k] + 1);
}
```

3.2 Cas $k=2$

Il y a des cailloux de couleur `CBLEU` et `CBLANC` alignés dans un ordre quelconque. Le but est de placer les premiers à gauche et les seconds à droite.

La bonne méthode est d'élaborer un invariant qui va être maintenu tout au long de l'exécution. Plaçons-nous à une étape intermédiaire où la situation est la suivante :

1	g	j	h	n
... CBLEU ...	inconnu		... CBLANC ...	

Au départ : $j = 1$ et $h = n$. Lorsque $h < j$, la situation désirée est atteinte. Comment progresser vers ce but ?

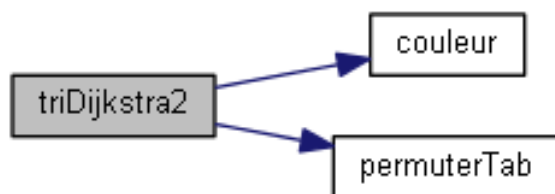
En regardant $T[j]$:

- S'il est de couleur **CBLEU** : incrémentez j . La quantité $(h - j)$ diminue.
- Sinon (il est de couleur **CBLANC**) : permuter les éléments $T[j]$ et $T[h]$ puis décrémente h . La quantité $(h - j)$ diminue.

La quantité $(h - j)$ est le *variant* de la boucle. Au départ elle vaut n et à chaque itération elle décroît. Au bout d'au plus n itérations elle sera négative ou nulle.



Écrivez une procédure `triDijkstra2(t,n)` qui effectue le réarrangement décrit ci-dessus pour n cailloux d'un **Tableau** t .



Validez votre procédure avec la solution.

Solution C++ @[pgdijkstra.cpp]

```

/**
 * Partition du drapeau avec k=2 couleurs
 * @param[in,out] t - un ITableau
 * @param[in] n - nombre de cailloux
 */
void triDijkstra2(ITableau& t, int n)
{
    int j = 0;
    int h = n - 1;
    while (j < h)
    {
        if (couleur(t,j) == CBleu)
        {
            ++j;
        }
        else
        {
            UtilsTR::permuterTab(t,j,h);
            --h;
        }
    }
}
  
```

```

    }
  }
}

```



Calculez la complexité au pire de votre solution.

Solution simple

A chaque itération, soit il n'y a aucun échange et le variant diminue de 1, soit il y a échange et le variant diminue de 2. Il y a donc au plus $\left\lceil \frac{n}{2} \right\rceil$ échanges. Le pire cas est atteint si le tableau de taille $n = 2N$ est composé de N cailloux de couleur 2 à gauche et de N cailloux de couleur 1 à droite. La complexité au pire est donc en $O(n)$.



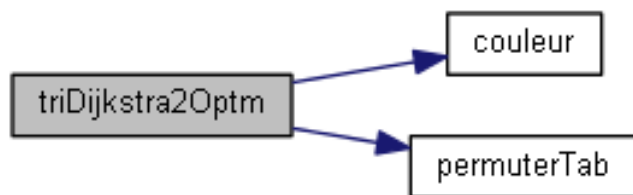
Pouvez-vous l'améliorer ?

Solution simple

Une version qui tient compte de la symétrie en (gauche, droite) est en $O(n/2)$ échanges.



Écrivez une procédure `triDijkstra2Optm(t,n)` qui effectue le réarrangement en tenant compte de la symétrie pour n cailloux d'un Tableau `t`.



Validez votre procédure avec la solution.

Solution C++

@[pgdijkstra.cpp]

```

/**
 * Partition du drapeau avec k=2 couleurs (optimal)
 * @param[in,out] t - un ITableau
 * @param[in] n - nombre de cailloux
 */
void triDijkstra2Optm(ITableau& t, int n)
{
    int j = 0;
    int h = n - 1;
    while (j <= h)
    {
        while (j <= h && couleur(t,j) == CBleu)
        {
            ++j;
        }
    }
}

```

```

while (j <= h && couleur(t,h) == CBlanc)
{
    --h;
}
if (j < h)
{
    UtilsTR::permuterTab(t,j,h);
    ++j;
    --h;
}
}
}

```



Écrivez un programme qui saisit le nombre de cailloux, déclare un **Tableau** et l'initialise de façon aléatoire en prenant 2 pour valeur maximale puis en effectue la partition en deux couleurs et l'affiche.



Testez.



Validez votre programme avec la solution.

Solution C++

@[pgdijkstra.cpp]

```

/**
 * @test
 */
void test1()
{
    ITableau tab;
    int nelems = UtilsTR::saisirNombreElements(TMAX);
    UtilsTR::aleatoireTri(tab,nelems,2);
    cout<<"Tri Dijkstra 2 couleurs"<<endl;
    triDijkstra2Optm(tab,nelems);
    cout<<"Tableau final"<<endl;
    UtilsTR::afficherTri(tab,nelems,0,nelems);
}

```

3.3 Cas k=3 : Problème de Dijkstra

Il y a des cailloux de couleur **CBLEU**, **CBLANC** et **CROUGE** alignés dans un ordre quelconque. Le but est de placer les premiers à gauche, les seconds au milieu et les troisièmes à droite. Reprenons la démarche par invariant et variant vue pour $k = 2$. La situation générique est donc la suivante :

1	g	j	h	n
...BLEUBLANC ...	inconnu	...ROUGE ...	

Au départ : $g = 0$, $j = 1$ et $h = n + 1$.

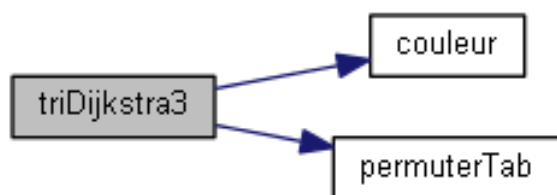
On cherche à réduire la zone inconnue de taille $h - j$.

Regardons $T[j]$:

- S'il est de couleur **CBLANC** : incrémentez j et le variant $(h - j)$ diminue.
- S'il est de couleur **CBLEU** : incrémentez g , permutez les éléments en j et g et incrémentez j . Le variant décroît aussi.
- Sinon (il est de couleur **CROUGE**) : décrémentez h puis permutez les éléments en j et h . Le variant décroît aussi.



Écrivez une procédure `triDijkstra3(t,n)` qui effectue le réarrangement décrit ci-dessus pour n cailloux d'un Tableau t .



Validez votre procédure avec la solution.

Solution C++ @[pgdijkstra.cpp]

```

/**
 * Partition du drapeau avec k=3 couleurs
 * @param[in,out] t - un ITableau
 * @param[in] n - nombre de cailloux
 */
void triDijkstra3(ITableau& t, int n)
{
    int j = 0;
    int g = -1;
    int h = n;
    while (j < h)
    {
        if (couleur(t,j) == CBlanc)
        {
            ++j;
        }
        else if (couleur(t,j) == CBleu)
        {
            ++g;
            UtilsTR::permuterTab(t,j,g);
            ++j;
        }
        else
        {
            --h;
            UtilsTR::permuterTab(t,j,h);
        }
    }
}
  
```

```
}
}
```



Modifiez votre programme pour qu'il saisisse en plus le nombre de couleurs dans `ncouleurs` (entier), initialise aléatoirement les cailloux d'un `Tableau` dans `[0..ncouleurs]` puis effectue la partition en deux ou trois couleurs selon la valeur de `ncouleurs` et l'affiche.



Testez.



Quelle est la complexité de votre solution ?

Solution simple

A chaque itération le variant diminue de 1 et il y a au plus un échange. L'état désiré est atteint au bout d'au plus n échanges. Ce pire cas est atteint s'il n'y a que des boules de couleur `CROUGE`. La complexité au pire est donc en $O(n)$.



Qu'en pensez-vous ?
Pouvez-vous l'améliorer ?

Solution simple

Il n'y a pas d'amélioration possible : c'est le cas optimal.



Il est possible de trouver une solution indirecte.
Expliquez comment.

Solution simple

Une autre solution consiste à considérer d'abord les couleurs `CBLEU` et `CBLANC` comme une unique couleur. On applique alors l'algorithme précédent $k = 2$ pour placer les cailloux de couleur `CBLEU` et `CBLANC` à gauche et ceux de couleur `CROUGE` à droite. Soit alors h le rang du premier caillou de couleur `CROUGE`. On réapplique ensuite l'algorithme $k = 2$ à la partie gauche `[1..h - 1]` pour placer les boules de couleur `CBLEU` à gauche et la couleur `CBLANC` à droite. D'où la répartition souhaitée.



Quel est le coût de votre solution indirecte ?

Solution simple

Ceci coûte au plus $\frac{1}{2}(n + (n - h))$ itérations. (Le lecteur pourra aisément trouver un exemple où ce pire cas est atteint.)

3.4 Cas k=4 et Cas générique

Il y a des boules de couleur **CBLEU**, **CBLANC**, **CROUGE** et **CVERT**. Le but est de placer les cailloux de couleur **CBLEU** à gauche, **CBLANC** puis **CROUGE** et **CVERT** à droite. La situation générique est la suivante :

1	g	j	h	v	n
...BLEUBLANC ...	inconnu	...ROUGEVERT ...	

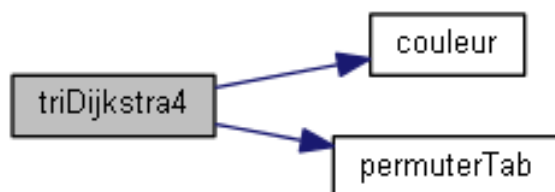
Au départ : $g = 0$, $j = 1$ et $v = h = n + 1$. Nous cherchons à réduire la zone inconnue de taille $h - j$.

Regardons $T[j]$:

- S'il est de couleur **CBLANC** : incrémentez j .
- S'il est de couleur **CBLEU** : incrémentez g , permutez les éléments en j et g et incrémentez j (comme précédemment).
- S'il est de couleur **CROUGE** : décrétez h puis permutez les éléments en j et h .
- Sinon (il est de couleur **CVERT**) : décrétez v puis permutez les éléments en j et v ; puis vérifiez si $h \leq v$ auquel cas il faut également décrétez h et permuter les éléments.



Écrivez une procédure `triDijkstra4(t,n)` qui effectue le réarrangement décrit ci-dessus pour n cailloux d'un Tableau t .



Validez votre procédure avec la solution.

Solution C++ @[pgdijkstra.cpp]

```

/**
 * Partition du drapeau avec k=4 couleurs
 * @param[in,out] t - un ITableau
 * @param[in] n - nombre de cailloux
 */
void triDijkstra4(ITableau& t, int n)
{
    int j = 0;
    int g = -1;
    int v = n;
    int h = n;
    while (j < h)
  
```

```

{
    if (couleur(t,j) == CBlanc)
    {
        ++j;
    }
    else if (couleur(t,j) == CBleu)
    {
        ++g;
        UtilsTR::permuterTab(t,j,g);
        ++j;
    }
    else if (couleur(t,j) == CRouge)
    {
        --h;
        UtilsTR::permuterTab(t,j,h);
    }
    // Quatrième zone: décrémentation de v, éventuellement h
    else
    {
        --v;
        UtilsTR::permuterTab(t,j,v);
        if (v < h)
        {
            --h;
        }
    }
}
}
}

```



Complétez votre programme afin d'effectuer la partition en quatre couleurs si `ncouleurs` vaut 4.



Testez.



Validez votre programme avec la solution.

Solution C++ @[pgdijkstra.cpp]

```

/**
 * @test
 */
void test2()
{
    ITableau tab;
    int nelems = UtilsTR::saisirNombreElements(TMAX);
    int ncouleurs;
    cout<<"Nombre de couleurs? ";
    cin>>ncouleurs;
    UtilsTR::aleatoireTri(tab,nelems,ncouleurs);
    cout<<"Tableau initial"<<endl;
    UtilsTR::afficherTri(tab,nelems,0,nelems);
    cout<<"Tri Dijkstra "<<ncouleurs<<" couleurs"<<endl;
}

```

```

switch (ncouleurs)
{
    case 2:
        triDijkstra2(tab,nelems);
        break;
    case 3:
        triDijkstra3(tab,nelems);
        break;
    case 4:
        triDijkstra4(tab,nelems);
        break;
}
cout<<"Tableau final"<<endl;
UtilsTR::afficherTri(tab,nelems,0,nelems);
}

```



Quelle est la complexité de votre solution ?

Solution simple

A chaque itération le variant diminue de 1 au moins et il y a au plus deux échanges. L'ensemble coûte donc au plus $2n$ échanges. La complexité est donc encore en $O(n)$.



Ici aussi, il est possible de trouver une solution indirecte. Expliquez comment.

Solution simple

Il est possible de confondre les trois premières couleurs en une pour se ramener au cas précédent $k = 3$. Ceci conduit à trois applications successives de l'algorithme à des tableaux de taille n , $n - h$, $n - v$ donc $\frac{1}{2}(3n - (v + h))$ échanges.

Une autre solution est de grouper les couleurs par deux : couleurs **CBLEU** et **CBLANC** d'une part et couleurs **CROUGE** et **CVERT** d'autre part. La séparation des deux groupes coûte $\frac{n}{2}$ échanges au plus. Soit r la taille du groupe de gauche. La séparation interne à ce groupe coûte $\frac{r}{2}$ et celle de l'autre $n - \frac{r}{2}$ donc en tout $\frac{2n}{2} = n$ exactement.



Connaissant les solutions pour $k = 1, \dots, m - 1$,

Comment construire une solution pour $k = m$?

Solution simple

Si k est de la forme 2^p le traitement dichotomique ci-dessus peut se généraliser. Il y a p étapes avec $\frac{n}{2}$ échanges au plus à chaque fois. Globalement le nombre maximal d'échanges est donc $\frac{n}{2} \lceil \lg k \rceil$. En procédant par un groupage 3 par 3 nous obtenons :

$$\frac{n}{2} \lceil \log_3 k \rceil = \frac{n}{2} \left\lceil \frac{\lg k}{\lg 3} \right\rceil$$

Comme $\lg 3 < 2$, le groupage par 2 est plus avantageux que celui par 3.

4 Références générales

Ce problème a servi de banc d'essai à de nombreux travaux sur les méthodes de conception et de validation de programmes. On trouvera une étude dans le livre de D. GRIES, *The science of programming*. Une application dans le cas de deux couleurs est le calcul de sommes sans faux dépassements.

Comprend [Bouge-AL1 :c03] ■