Projet de Complexité et Algorithmique Appliquée

Couverture par sommets

M2 informatique

Christopher Cacciatore

Quentin Poussier

Table des matières

GÉNÉRATION ALÉATOIRE

Génération aléatoire d'arbres

Paramètres :

- n : nombre de sommets

Méthode :

- Pour chaque sommet S à ajouter
  - On ajoute S à la liste des sommets
  - Si le graphe contient déjà des sommets
    - On en tire un au hasard : S'!= S
    - On créé l'arête [S, S']

En C++ :

```
Tree::Tree(int nbNode)
{
    nbVerts = 0;
    for(int i=0; i<nbNode; i++)
    {
        Node n(i);
        addVert(n);

        int rdNodeId = rand()%nbVerts;
        while((nbVerts>1) && (rdNodeId==i))
            rdNodeId = rand()%nbVerts;

        ListAdj* Lrd = getListFromNode(Node(rdNodeId));
        ListAdj* Ln = getListFromNode(n);

        if((*Lrd).getNode() != n)
            addEdge(Lrd, Ln);

    }
}
```

Complexité :

Les opérations de création et d'ajout de sommets et d'arêtes s'effectuent en temps constant. Par conséquent, Cet algorithme a une complexité de O(n).

Génération aléatoire de graphes

Paramètres :

- n : nombre de sommets
- p : probabilité de tracer une arête entre deux sommets

Méthode :

- On ajoute n sommets au graphe
- Pour chaque couple de sommets (S, S')
  - On tire un nombre aléatoire entre 0 et 100 : rand
  - Si (S!=S') et (rand<p) et l'arête [S,S'] n'existe pas
    - on ajoute l'arête [S,S']

En C++ :

```
Graph::Graph(int nbNode, int prob)
{
    this->nbVerts = 0;
    this->nbEdges = 0;

    for(int i=0; i < nbNode; i++){
        Node n(i);
        addVert(n);
    }
    list<Node> done;
    for (list<ListAdj>::iterator it=listListAdj.begin(); it!=listListAdj.end(); ++it)
    {
        for (list<ListAdj>::iterator it2=listListAdj.begin(); it2!=listListAdj.end(); ++it2)
        {

            if(it != it2){
                if((rand() % 100) < prob){
                    Node n = (*it2).getNode();
                    if(!isInList<Node>(n, done)){
                        addEdge(&(*it), &(*it2));
                    }
                }
            }
            done.push_back((*it).getNode());
        }
    }

}
```

Complexité :

$O(n + n^2) = O(n^2)$

première boucle for — boucles imbriquées

Génération aléatoire de graphes bipartis

Paramètres :

- n : nombre de sommets
- p : probabilité de tracer une arête entre deux sommets

Méthode :

- On ajoute les n sommets en les répartissant dans deux ensembles distincts : partLeft et partRight non vides
- Pour chaque sommet S de partLeft
  - Pour chaque sommet S' de partRight
    - On tire un nombre aléatoire entre 0 et 100 : rand
    - Si (rand<p) et l'arête [S,S'] n'existe pas
      - on ajoute l'arête [S,S']

En C++ :

```
Bipart::Bipart(int nbNode, int prob)
{
    this->nbVerts = nbNode;
    partLeft.push_back(ListAdj((Node(0))));
    partRight.push_back(ListAdj((Node(1))));
    for(int i=2; i < nbVerts; i++){
        Node n(i);
        addVert(n);
    }
    for (list<ListAdj>::iterator it=partLeft.begin(); it!=partLeft.end(); ++it)
    {
        for (list<ListAdj>::iterator it2=partRight.begin(); it2!=partRight.end(); ++it2)
        {
            if(it != it2){
                if((rand() % 100) < prob){
                    addEdge(&(*it), &(*it2));
                }
            }
        }
    }

}
```

Complexité :

O(n + c*(n-c)) avec c étant le nombre d'éléments de partLeft.

O(n + n*c-c²) ≈ O(n)

Génération aléatoire de graphes ayant une petite couverture

Paramètres :

- n : nombre de sommets
- p : probabilité de tracer une arête entre deux sommets
- k : taille de la couverture

Méthode :

- On ajoute n sommets au graphe, et on définit les k premiers comme appartenant à la couverture
- Pour chaque sommet S de la couverture
  - Pour chaque sommet S' du graphe
    - Si S!=S' et rand<p et l'arête [S,S'] n'existe pas
      - on ajoute l'arête [S,S']

En C++ :

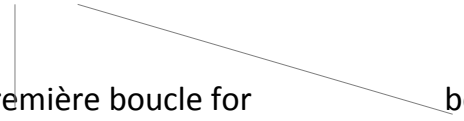
```
SmallCoverGraph::SmallCoverGraph(int nbNode, int prob, int coverSize)
{
    this->nbVerts=0;
    for(int i=0; i<nbNode; i++)
    {
        Node n(i);
        addVert(n);
        if(i<coverSize)
            cover.push_back(n);
    }

    std::list<Node> done;
    for (list<Node>::iterator it=cover.begin(); it != cover.end(); ++it)
    {
        for (list<ListAdj>::iterator it2=listListAdj.begin(); it2!=listListAdj.end(); ++it2)
        {
            if((*it).getId() != (*it2).getNode().getId())
            {
                if((rand() % 100) < prob)
                {
                    Node n = (*it2).getNode();
                    if(!isInList<Node>(n, done))
                    {
                        ListAdj* tmp = getListFromNode(*it);
                        addEdge(tmp, &(*it2));
                    }
                }
            }
            done.push_back((*it));
        }
    }

}
```

Complexité :

$O(n + k*n) = O(n*(k+1)) \approx O(n)$ (k étant supposé petit par rapport à n)



RECHERCHE D'UNE COUVERTURE

Algorithme de recherche pour les arbres

Paramètres :

- Entrée : un arbre
- Sortie : une liste de sommets

Méthode :

- On colore la racine de l'arbre en noir
- Pour chaque sommet S de l'arbre
  - Si S est noir, on colore ses enfants en blanc
  - Si S est blanc, on colore ses enfants en noir
- On retourne l'ensembe (blanc ou noir) le plus petit

En C++ :

```
std::list<Node> coverTree(Tree *tree)
{
    vertBlack.clear();
    vertWhite.clear();
    ListAdj current = (*tree).getRoot();
    vertBlack.push_back(current.getNode());

    sortTree(current, &vertWhite, &vertBlack, tree);

    if(vertWhite.size() >= vertBlack.size())
        return vertBlack;
    else
        return vertWhite;
}

void sortTree(ListAdj current, std::list<Node> *goodList, std::list<Node> *wrongList,
Tree *tree)
{
    if((!(current.getNeighbours().empty())) && (current.getNode() != Node(-1)))
    {
        list<Node> listNode = current.getNeighbours();
        for(list<Node>::iterator it=listNode.begin(); it != listNode.end(); ++it)
        {
            (*goodList).push_back((*it));
            ListAdj* next = (*tree).getListFromNode(*it);
            sortTree(*next, wrongList, goodList,tree);
        }
    }
}
```

Complexité :

La complexité de cette algorithme est la même que pour un algorithme de parcours en profondeur : O(|E|) avec |E| le nombre d'arêtes de l'arbre.

Algorithme de recherche pour les graphes bipartis

Paramètres :

- Entrée : un graphe biparti
- Sortie : une liste de sommets

Méthode :

- On retourne la partie du graphe de plus petite taille

En C++ :

```
std::list<Node> coverBipart(Bipart *bipart){
    if(bipart->getLeft().size() > bipart->getRight().size())
        return bipart->getRight();
    else
        return bipart->getLeft();
}
```

Complexité :

Notre implémentation des graphes bipartis permet de stocker les deux parties du graphe, partLeft et partRight, séparément. Il suffit donc de retourner celle qui a la plus petite taille. Cet algorithme a dans notre cas une complexité de O(1).

Algorithme 2-approché pour les graphes quelconques

Paramètres :

- Entrée : un graphe
- Sortie : une liste de sommets

Méthode :

- On trouve une forêt couvrante pour le graphe
- On retourne la concaténation des couvertures des arbres de cette forêt.

En C++ :

```
std::list<Tree> dfs(Graph *graph){
    std::list<Node> visited;
    std::list<Tree> result;
    std::list<ListAdj> lists = graph->getLists();
    for (std::list<ListAdj>::iterator it=lists.begin(); it != lists.end(); ++it)
    {
        Node n = (*it).getNode();
        if(!isInList<Node>(n, visited))
        {
            Tree treeRes;
            treeRes.addVert((*it).getNode());
            visited.push_back((*it).getNode());
            sdfs(graph,(*it), &visited, &treeRes);
            result.push_back(treeRes);
        }
    }
    return result;
}

void sdfs(Graph *graph, ListAdj neighbours,std::list<Node> *visited, Tree
*result ){
    std::list<Node> nbr = neighbours.getNeighbours();
    for (std::list<Node>::iterator it=nbr.begin(); it != nbr.end(); ++it){
        if(!isInList((*it), *visited)){
            (*result).addVert(*it);

            ListAdj *n1 = (*result).getListFromNode(neighbours.getNode());
            ListAdj *n2 = (*result).getListFromNode(*it);
            (*result).addEdge(n1, n2);

            (*visited).push_back(*it);

            ListAdj *next = (*graph).getListFromNode(*it);
            sdfs(graph, *next, visited, result);
        }
    }
}
```

```
std::list<Node> coverGraph(Graph *graph){
    std::list<Tree> lt = dfs(graph);
    std::list<Node> res;

    for(std::list<Tree>::iterator tree=lt.begin(); tree!=lt.end(); ++tree){
        std::list<Node> tmpRes = coverTree(&(*tree));
        res.insert(res.end(), tmpRes.begin(), tmpRes.end());
    }

    return res;
}
```

Complexité :

Dfs (Deep-First Search : parcours en profondeur) a une complexité en O(|E|), ainsi que coverTree. La concaténation de deux listes se fait ici en temps constant. La complexité de cet algorithme est donc O(|E|).

UTILISATION DE MINISAT

Transformation d'une instance de VC en instance de SAT

Paramètres :

- Un graphe
- Un fichier de sortie

Méthode :

- Les arêtes du graphe représentent les clauses de SAT
- Pour une arête/clause, on a donc deux sommets/variables

En C++ :

```
void coverToSAT(Graph *graph, std::string outputFile){
    int nbVert = graph->getNbVerts();
    int nbEdge = graph->getNbEdges();
    std::ofstream file(outputFile.c_str(), std::ios::out);
    if(file){
        file << "p cnf " << nbVert << " " << nbEdge;
        file << "\n";
        std::list<Edge> listEdge = graph->getListEdges();
        for(list<Edge>::iterator current=listEdge.begin(); current!=listEdge.end(); ++current){
            file << (*current).getFirstNode().getId();
            file << " " << (*current).getSecondNode().getId();
            file << "\n";
        }
        file.close();
    }else{
        std::cerr << "file not found" << std::endl;
    }

}
```

Complexité :

Ici un simple parcours des arêtes du graphe suffit à générer un fichier acceptable par minisat : la complexité de cet algorithme est donc O(|E|).

Transformation d'une solution fournie par minisat en couverture pour l'instance initiale de VC

Paramètres :

- Entrée : un fichier contenant un résultat fourni par minisat
- Sortie : une liste de sommets

Méthode :

- On parse le fichier, et on ajoute à la couverture les sommets correspondant aux variables dont la valeur est « vrai » dans la solution.

En C++ :

```
std::list<Node> satToCover(std::string inputFile){
    std::list<Node> resultCover;
    std::ifstream flux(inputFile.c_str(), std::ios::in);
    if(flux){
        std::string line;
        std::getline(flux, line);
        if(line == "SAT"){
            std::getline(flux, line);
            std::vector<std::string> vNode = split(line, ' ');
            for(vector<std::string>::iterator idNode=vNode.begin();idNode!=vNode.end(); ++idNode)
            {
                if(atoi((*idNode).c_str()) >= 0){
                    Node n(atoi((*idNode).c_str()));
                    resultCover.push_back(n);
                }
            }
        }else{
            std::cerr << "no solution found" << std::endl;
        }
        flux.close();
    }else{
        std::cerr << "file not found" << std::endl;
    }
    return resultCover;
}
```

Complexité :

La solution contient autant de variables qu'il y a de sommets dans le graphe de départ, la complexité est donc O(n).