INFO-F-413 : Implémentation de l'Algorithme de Karger

Thomas Chapeaux

September 30, 2012

1 Présentation

L'algorithme de Karger[1] est un algorithme probabiliste permettant d'estimer une coupe minimale d'un graphe, basé sur la contraction d'arêtes choisies aléatoirement.

Au cours, il a été vu que cet algorithme a une probabilité de succès (c'est-à-dire de trouver effectivement une coupe minimale) strictement supérieure à $\frac{2}{n(n-1)}$. Cette borne inférieure peut être améliorée jusqu'à $1 - \frac{1}{n^2}$ en exécutant $n(n-1)\log n$ itérations de l'algorithme et en gardant le meilleur résultat (la plus petite valeur trouvée).

Ce document présente une implémentation de l'algorithme en Python et l'utilise pour discuter les valeurs trouvées au cours.

2 Implémentation

2.1 Languages et bibliothèque

L'algorithme a été implémenté en Python 2.7 à l'aide de la bibliothèque igraph[2]. Cette bibliothèque permet de générer des graphes aléatoirement via une méthode géométrique[3] et propose une méthode pour trouver la coupé minimale d'un graphe, ce qui permettra de juger de la qualité des valeurs trouvées par notre algorithme.

Python a surtout été choisi par préférence personnelle, mais également pour son efficacité lors de projets de taille raisonnable comme celui-ci.

[1] Réf. [1]
[2] http://igraph.sourceforge.net
[3] Réf. [2] et Annexe 1

2.2 Implémentation de l'algorithme

2.2.1 Une itération

Lors de la contraction d'une arête A, chaque autre arête connectée à la destination de A est redirigée vers la source de A[4], et toutes les arêtes ayant la même source et la même destination que A (donc A également) sont supprimées.

```
# g is an instance of Graph()
while (g.vcount() > 2):
   edge_list = g.get_edgelist()
   rand_edge_id = random.randint(0,g.ecount()-1)
   chosen_edge = edge_list[rand_edge_id]
   chosen_edge_source = chosen_edge[0]
   chosen_edge_target = chosen_edge[1]
   for e in edge_list:
      if (e[0] == chosen_edge_target or e[1] == chosen_edge_target):
         if (e[0] == chosen_edge_target and e[1] != chosen_edge_source)
             :
            g.add_edge(chosen_edge_source, e[1])
         elif (e[1] == chosen_edge_target and e[0] !=
             chosen_edge_source):
            g.add_edge(e[0], chosen_edge_source)
         edge_id = graph.get_eid(e[0], e[1])
         g.delete_edges(edge_id)

   assert(g.degree(chosen_edge_target) == 0)
   g.delete_vertices(chosen_edge_target)
```

2.2.2 Algorithme complet

On a choisi de faire par défaut le nombre d'itérations trouvé en cours pour avoir une probabilité de succès de $1 - \frac{1}{n^2}$ mais celui-ci peut être contrôlé en modifiant le paramètre *CONFIDENCE_FACTOR* pour faire moins d'itérations (facteur de confiance > 1) ou plus (< 1).

```
number_of_vertices = graph.vcount()
best_mincut_value = graph.ecount() + 1 # borne superieure
required_nbr_iter = int(number_of_vertices*(number_of_vertices-1)*math.
    log(number_of_vertices))
nbr_iter = required_nbr_iter/CONFIDENCE_FACTOR
for i in range(nbr_iter):
   g_temp = g.copy()
   result = random_mincut_one_instance(g_temp)
   if (result < best_mincut_value):
      best_mincut_value = result
```

2.3 Outil d'analyse

2.3.1 Collecte de données

Le code va tester un certain nombre de graphes différents (générés aléatoirement). Lors de l'application de l'algorithme complet sur un graphe, certaines données sont collectées pour pouvoir en tirer des statistiques (voir plus loin).

[4]Le graphe étant non dirigé, on parle de source et de destination seulement du point de vue de l'implémentation dans igraph.

```
class TestResult:
   def __init__(self, graph):
      #self.graph = graph # not used
      self.number_of_vertices = graph.vcount()
      self.number_of_edges = graph.ecount()
      self.number_of_iterations = None
      self.improving_iterations = []
      self.improved_value = []
      self.found_value = None
      self.correct_value = None

   def consistencyCheck(self):
      assert(len(self.improving_iterations)==len(self.improved_value))
      assert(self.found_value != None)
      assert(self.correct_value != None)
```

Une instance de cette classe par graphe testé est générée par le code pendant l'exécution de l'algorithme. On remarquera qu'on enregistre les itérations qui ont augmenté le résultat (*improving_iterations*) ainsi que la valeur correspondante (*improved_values*)

2.3.2 Analyse des données

Une fois tous les tests exécutés, le code tire les statistiques suivantes de tous les *TestResult* générés :

- Nombre de tests échoués (où une valeur trop grande a été trouvée pour la coupe minimale).
- Écart moyen avec la véritable valeur de coupe minimale.
- Proportions d'itérations "inutiles" (lorsque la valeur exacte avait déjà été trouvée)
- Itérations "utiles" les plus lointaines

Ces résultats sont ensuite affichés de la manière suivante :

```
confidence factor: 10
failures : 34 = 3.4% of tests
...with an average of 1.32 absolute error
(Failed tests were removed from the following stats)
Three latest useful iterations (relative to total #iter) : 0.91, 0.9,
    0.8
Proportion of useful work : 0.15 (0.02 without confidence factor)
```

3 Discussion

Nous avions démontré (au cours) qu'en effectuant $n(n-1)\log n$ itérations, on était certain d'avoir un taux d'échec inférieur à $\frac{1}{n^2}$. Voyons comment ce résultat se traduit dans les tests.

Après une série de 1000 tests avec ce nombre d'itérations sur des graphes générés aléatoirement ayant au moins 5 sommets, nous devrions observer au plus $\frac{1000}{n^2} = 40$ erreurs. Le résultat est le suivant :

```
confidence factor: 1
failures : 0 = 0.0% of tests
Three latest useful iterations (relative to total #iter) : 0.4, 0.22,
    0.19
Proportion of useful work : 0.02 (0.02 without confidence factor)
```

On peut donc observer que la borne supérieure que nous avons trouvé est non seulement correcte, mais aussi fort exagérée[5]. Sur les 1000 tests, aucun n'a échoué et l'itération trouvant la bonne valeur la plus lointaine n'était qu'à 40% de notre borne supérieure !

En fait, même en répétant plusieurs fois des séries de 1000 ou 10.000 tests, nous ne sommes arrivés que rarement à produire un test qui trouve une valeur erronée avec le nombre d'itérations. D'où l'ajout du paramètre *CONFIDENCE_FACTOR*, qui réduit le nombre d'itérations effectuées par le programme.

Série de 10.000 tests (donc 400 erreurs attendues si on respecte la borne supérieure), avec un facteur de confiance de 10 :

```
confidence factor: 10
failures : 474 = 4.7% of tests
...with an average of 1.22 absolute error
(Failed tests were removed from the following stats)
Three latest useful iterations (relative to total #iter) : 0.95, 0.93,
    0.85
Proportion of useful work : 0.15 (0.01 without confidence factor)
```

Cette fois, en faisant dix fois moins de calculs que prévus, on arrive quand même à respecter (ou presque) le taux de réussite qu'on s'était imposé. De plus, l'erreur commise n'est en moyenne pas très importante, ce qui pourrait être suffisant pour certaines applications nécessitant un calcul de coupe minimale.

On peut tout de même remarquer que, malgré notre facteur de confiance, 85% des itérations ont été calculées alors que la réponse avait déjà été trouvée. Ceci semble excessif mais est le prix à payer pour avoir une très grande majorité de résultats corrects (ceci peut se voir dans les trois itérations "gagnantes" les plus lointaines, qui sont toutes les trois arrivées vers la fin de l'algorithme).

Testons maintenant une valeur extrême de confiance (10.000 tests, facteur de confiance 100) :

```
confidence factor: 100
failures : 5139 = 51.39% of tests
...with an average of 2.7 absolute error
Three latest useful iterations (relative to total #iter) : 0.5, 0.0,
    0.0
Proportion of useful work : 0.03 (0.0 without confidence factor)
```

(L'algorithme ne testant qu'une ou deux itérations par graphes, les deux dernières lignes moins significatives)

[5]En tout cas pour l'ensemble des graphes accessibles par notre fonction de génération aléatoire, qui n'est bien sûr qu'un sous-ensemble qu'on espère représentatif de tous les graphes possibles.

On a maintenant une majorité de résultats erronés (ce qui était attendu vu le facteur de confiance très élevé). Cependant, on peut quand même remarquer que l'erreur moyenne n'est pas si grande que ça (elle est significative, mais pas extravagante). On pourrait imaginer des applications nécessitant souvent d'avoir très rapidement une petite coupe d'un graphe, sans contrainte de minimalité, où l'algorithme de Karger avec un grand facteur de confiance serait parfaitement adapté.

4 Références

- [1] Karger, David (1993). "Global Min-cuts in RNC and Other Ramifications of a Simple Mincut Algorithm". Proc. 4th Annual ACM-SIAM Symposium on Discrete Algorithms.
- [2] "n points are chosen randomly and uniformly inside the unit square and pairs of points closer to each other than a predefined distance d are connected by an edge" `http://hal.elte.hu/~nepusz/development/igraph/tutorial/tutorial.html`

5 Annexes

5.1 Génération aléatoire de graphes

Pour avoir des résultats significatifs, il nous a semblé préférable de tester l'algorithme avec des graphes générés aléatoirement plutôt qu'avec des "cas particuliers" codés en dur dans le code.

L'algorithme utilisé, proposé par la bibliothèque igraph, prend comme paramètres :

n : le nombre de sommets du graphe

r : un nombre entre 0 et 1

Il génère le graphe en posant les n sommets aléatoirement dans un carré de côté 1, puis en reliant d'une arête les points situés à une distance inférieure à r.

L'algorithme peut donc générer des graphes non connexes, qui sont des cas triviaux jugés non intéressants car chaque itération de l'algorithme (modifié pour accepter ce cas) renverra la bonne valeur avec une probabilité de 1. Il a donc été choisi de détecter les graphes connexes renvoyés par l'algorithme et de demander un nouveau graphe aléatoire dans ce cas.

Au cours, on a calculé que l'algorithme de Karger avait une complexité en $O(n^4)$, il convient donc de choisir un n d'une taille raisonnable. Sur notre matériel et pour des séries de 1000 tests, 10 sommets semblent être une borne maximale pour avoir un temps de calcul viable.
Le choix de r, qui modifiera surtout le nombre d'arêtes du graphe, semblait avoir moins d'importance sur le temps de calcul. Mais il nous semblait préférable d'éviter les valeurs extrêmes (0 : aucunes arêtes, 1 : tous les sommets sont

reliés).
Au final, par plaisir de randomisation, il a été décidé de générer aléatoirement pour chaque graphe un n entre 5 et 10, et un r entre 0,4 et 1,0 (ces valeurs étant des constantes facilement modifiables).

5.2 Améliorations possibles

- Fonction de génération couvrant plus de types de graphes
- Graphique de la position relative des itérations "gagnantes" pour chaque test (information déjà présente dans les instances de *TestResult*, il reste "seulement" à l'afficher).
- En plus des valeurs moyennes, calculer les variances et les valeurs maximales des grandeurs mesurées.

5.3 Code source complet

Le code a normalement été fourni avec le présent rapport. Il est aussi disponible sur GitHub :
`https://github.com/BafAltom/info-f413-random-mincut`

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

from igraph import *
# igraph: http://igraph.sourceforge.net/
import random
import math

random.seed()

# Nombre de graphes qui vont  tre  test s
NUMBER_OF_TESTS = 1000
# Proportion d'it rations qui ne vont pas  tre  effectu es (par
    rapport    la borne minimale vue au cours)
CONFIDENCE_FACTOR = 1 # def: 5
# Bornes min/max du nombre de sommets de chaque graphe
VERTICES_NB_MIN = 5 # def: 5
VERTICES_NB_MAX = 10 # def: 10
# Borne min/max du radius GRG de chaque graphe (une plus grande valeur
    signifie plus d'ar tes)
GRG_RADIUS_MIN = 0.4 # def: 0.4
GRG_RADIUS_MAX = 1.0 # def: 0.9
# Utilise (True) ou non (False) les graphes pour lesquels la coupe
    minimale n'a pas   t   trouv e dans les statistiques
REMOVE_FAILED_TESTS_IN_STATS = True
# Affiche (True) ou non (False) un  r  s u m  des tests pour lesquels la
    coupe minimale n'a pas   t   trouv e
DISPLAY_FAILURES = False
# Recommence l'ex cution si aucune faute n'a   t   trouv e (peut
    prendre beaucoup de temps...)
CONTINUE_ON_PERFECT_TEST = False

class TestResult: # Classe stockant un  r  s u m  de l'ex cution d'un
    test
   def __init__(self, graph):
      #self.graph = graph
      self.number_of_vertices = graph.vcount()
```

```
32      self.number_of_edges = graph.ecount()
33      self.number_of_iterations = None
34      self.improving_iterations = []
35      self.improved_values = []
36      self.found_value = None
37      self.correct_value = None
38
39   def consistencyCheck(self):
40      assert(len(self.improving_iterations) == len(self.improved_values
            ))
41      assert(self.found_value != None)
42      assert(self.correct_value != None)
43
44   def __str__(self):
45      self.consistencyCheck()
46      s = "Test with a graph of " + str(self.number_of_vertices) + "
            verticles and " + str(self.number_of_edges) + " edges.\n"
47      s += "Used " + str(self.number_of_iterations) + " iterations.\n"
48      for i in range(len(self.improving_iterations)):
49         s += "- Found value : " + str(self.improved_values[i]) + " at
               iteration " + str(self.improving_iterations[i]) + "\n"
50      s += "Final value : " + str(self.found_value) + "\n"
51      s += "Correct value : " + str(self.correct_value) + "\n"
52      return s
53
54 def random_mincut_one_instance(g):
55   while (g.vcount() > 2 and g.ecount() > 0):
56      edge_list = g.get_edgelist()
57      if (g.ecount() == 1): rand_edge_id = 0 # hotfix pour des graphes
            non connexes
58      else: rand_edge_id = random.randint(0,g.ecount()-1)
59      chosen_edge = edge_list[rand_edge_id]
60      chosen_edge_source = chosen_edge[0]
61      chosen_edge_target = chosen_edge[1]
62      for e in edge_list:
63         if (e[0] == chosen_edge_target or e[1] == chosen_edge_target):
64            if (e[0] == chosen_edge_target and e[1] !=
                  chosen_edge_source):
65               g.add_edge(chosen_edge_source, e[1])
66            elif (e[1] == chosen_edge_target and e[0] !=
                  chosen_edge_source):
67               g.add_edge(e[0], chosen_edge_source)
68            edge_id = g.get_eid(e[0], e[1])
69            g.delete_edges(edge_id)
70
71      assert(g.degree(chosen_edge_target) == 0)
72      g.delete_vertices(chosen_edge_target)
73   return g.ecount()
74
75 def random_mincut_all(graph, myTestResult):
76   number_of_vertices = graph.vcount()
77   best_mincut_value = 999999
78   required_nbr_iter = int(number_of_vertices*(number_of_vertices-1)*
        math.log(number_of_vertices))
79   nbr_iter = max(1,required_nbr_iter/CONFIDENCE_FACTOR)
80   myTestResult.number_of_iterations = nbr_iter
81   for i in range(nbr_iter):
82      g_temp = graph.copy()
83      result = random_mincut_one_instance(g_temp)
84      if (result < best_mincut_value):
85         best_mincut_value = result
86         myTestResult.improving_iterations.append(i)
```

```
87          myTestResult.improved_values.append(best_mincut_value)
88
89      myTestResult.found_value = best_mincut_value
90      myTestResult.correct_value = int(graph.mincut().value)
91      return myTestResult
92
93  def generate_random_connex_graph(number_of_vertices, GRG_radius):
94      g = Graph.GRG(number_of_vertices, GRG_radius)
95      while (g.cohesion() == 0):
96          g = Graph.GRG(number_of_vertices,GRG_radius)
97      return g
98
99  def strRound2(aFloat):
100     return str(round(aFloat,2))
101
102 if __name__ == "__main__":
103     anotherRound = True
104     while (anotherRound):
105         tests_results = []
106         for i in range(NUMBER_OF_TESTS):
107             if (NUMBER_OF_TESTS < 1000): print "test " + str(i+1) + " of "
                    + str(NUMBER_OF_TESTS) + "..."
108             elif(((i+1)%(NUMBER_OF_TESTS/100)) == 0): print "test " + str(
                    i+1) + " of " + str(NUMBER_OF_TESTS) + "..."
109
110             number_of_vertices = random.randint(VERTICES_NB_MIN,
                    VERTICES_NB_MAX)
111             GRG_radius = random.randint(GRG_RADIUS_MIN*100,GRG_RADIUS_MAX
                    *100)/float(100)
112             g = generate_random_connex_graph(number_of_vertices,
                    GRG_radius)
113             result = TestResult(g)
114             random_mincut_all(g, result)
115             tests_results.append(result)
116
117         # compute some statistics on tests_results
118         current_prop_sum = 0
119         current_failure_sum = 0
120         current_failure_value = 0
121         current_max_useful = 0
122         current_max_useful_2 = 0
123         current_max_useful_3 = 0
124         for tr in tests_results:
125             tr.consistencyCheck()
126             #failure count and value
127             if(tr.found_value != tr.correct_value):
128                 if (DISPLAY_FAILURES):
129                     print "_____________________________"
130                     print "Failure :"
131                     print tr
132                 current_failure_sum += 1
133                 current_failure_value += tr.found_value - tr.correct_value
134                 if (REMOVE_FAILED_TESTS_IN_STATS):
135                     continue
136             # max useful
137             number_of_correction = len(tr.improving_iterations)
138             if (number_of_correction == 0):
139                 print tr
140             last_useful_iteration_abs = tr.improving_iterations[
                    number_of_correction-1]
141             last_useful_iteration_rel = last_useful_iteration_abs / float(
                    tr.number_of_iterations)
```

```
142         if (last_useful_iteration_rel > current_max_useful):
143            current_max_useful_3 = current_max_useful_2
144            current_max_useful_2 = current_max_useful
145            current_max_useful = last_useful_iteration_rel
146         # proportion of useful work
147         current_prop_sum += last_useful_iteration_rel
148
149      prop_average = None
150      if(REMOVE_FAILED_TESTS_IN_STATS):
151         prop_average = current_prop_sum/(len(tests_results) -
               current_failure_sum)
152      else:
153         prop_average = current_prop_sum/len(tests_results)
154
155      if (CONTINUE_ON_PERFECT_TEST and current_failure_sum == 0):
            anotherRound = True
156      else: anotherRound = False
157
158   print "_____"
159   print "confidence factor: " + str(CONFIDENCE_FACTOR)
160   print "failures : " + str(current_failure_sum) + " = " + strRound2
         (100*current_failure_sum/float(NUMBER_OF_TESTS)) + "% of tests"
161   if (current_failure_sum > 0): print "...with an average of " +
         strRound2(current_failure_value/float(current_failure_sum)) + "
         absolute error"
162   if (REMOVE_FAILED_TESTS_IN_STATS): print "(Failed tests were removed
         from the following stats)"
163   print "Three latest useful iterations (relative to total #iter) : "
         + strRound2(current_max_useful) + ", " + strRound2(
         current_max_useful_2) + ", " + strRound2(current_max_useful_3)
164   print "Proportion of useful work : " + strRound2(prop_average) + " (
         " + strRound2(prop_average/CONFIDENCE_FACTOR) + " without
         confidence factor)"
```