TP 9

COUPLAGE DE CARDINAL MAXIMUM

On s'intéresse au problème de la détermination d'un couplage de cardinal maximum d'un graphe biparti, d'abord en utilisant l'algorithme vu en cours puis à l'aide d'une variante plus efficace, appelée algorithme de Hopcroft-Karp.

Dans tout le sujet :

- $G = (X \sqcup Y, E)$ désigne un graphe biparti – on note $V = X \sqcup Y$ son ensemble de sommets ;
- $X = \{0, \dots, n_1 - 1\}$, $Y = \{n_1, \dots, n - 1\}$ – toutes les arêtes relient donc un sommet d'indice strictement inférieur à n_1 à un sommet d'indice supérieur ou égal à n_1 ;
- on note $p = |E|$ le nombre d'arêtes du graphe ;
- on note $A \oplus B = (A \cup B) \setminus (A \cap B)$ la différence symétrique de deux ensembles A et B – on rappelle que cette opération est associative et commutative.

1 Implémentation de l'algorithme du cours

On représente un graphe biparti par le type suivant :

```
type bipartite = {
  n1 : int;
  adj : int list array
}
```

Un couplage est représenté par le type suivant :

```
type matching = int option array
```

Pour un couplage `m`, on aura :

- `m.(i) = None` si le sommet i est libre ;
- `m.(i) = Some j` et `m.(j) = Some i` si les sommets i et j sont appariés.

Un chemin $x_0, \dots, x_k$ sera simplement représenté par la liste `[x0; ...; xk]` :

```
type path = int list
```

Remarque

Mathématiquement, on verra parfois un chemin de longueur k comme une liste de k arêtes, parfois comme une liste de $k + 1$ sommets. En revanche, il sera toujours représenté informatiquement par une liste de $k + 1$ sommets.

► **Question 1** Écrire une fonction `is_augmenting` prenant en entrée un couplage `m` et un chemin et indiquant si le chemin est augmentant. On supposera sans le vérifier que le chemin est élémentaire et que les entiers sont bien entre 0 et $|m|$.

```
val is_augmenting : matching -> path -> bool
```

Remarque
Cette fonction n'est pas nécessaire pour la suite mais peut aider à identifier d'éventuelles erreurs dans le code. D'autres fonctions utilitaires (pour vérifier qu'un couplage est correct, l'afficher...) sont fournies.

▸ **Question 2** Écrire une fonction `delta` prenant en entrée un couplage M et un chemin p supposé augmentant pour M et effectuant l'opération $M \leftarrow M \oplus p$.

```
val delta : matching -> path -> unit
```

Remarque
Mathématiquement, on voit ici p et M comme des ensembles d'arêtes.

▸ **Question 3** Écrire une fonction `orient` prenant en entrée un graphe biparti G et un couplage M et renvoyant un graphe orienté G_M (sous forme d'un `int list array`) tel que :

- les sommets de G_M sont les mêmes que ceux de G ;
- G_M contient exactement un arc orienté pour chaque arête $\{x, y\}$ (avec $x \in X$ et $y \in Y$) de G :
  — si $xy \in M$, l'arc de G_M est (y, x) ;
  — si $xy \notin M$, l'arc de G_M est (x, y).

Cette orientation est celle présentée dans le cours – on ne rajoute en revanche pas de sommet source ni de sommet puits.

```
val orient : bipartite -> matching -> int list array
```

▸ **Question 4** Écrire une fonction `find_augmenting_path` prenant en entrée un graphe biparti G et un couplage M et renvoyant :

- `None` si M n'admet pas de chemin augmentant ;
- `Some p`, avec p un chemin augmentant, s'il en existe un.

On demande ici d'adapter et de transcrire l'algorithme présenté en cours. Il y a un peu de travail de programmation : en comptant les fonctions auxiliaires, on devrait avoir 20 à 30 lignes de code.

▸ **Question 5** Écrire une fonction `get_maximum_matching` prenant en entrée un graphe biparti G et renvoyant un couplage M de G de cardinal maximum.

```
val get_maximum_matching : bipartite -> matching
```

▸ **Question 6** Déterminer la complexité de la fonction précédente.

▸ **Question 7** Une série de fichiers nommés `graph_n1_n2_c.txt` est fournie : chaque fichier décrit un graphe biparti $G = (X \sqcup Y, E)$ avec $|X| = n_1$, $|Y| = n_2$ et c le cardinal maximal d'un couplage de G. Le format de fichier est très simple, mais il est inutile de s'y intéresser : une fonction `read_bipartite` est fournie. Elle prend en entrée un nom de fichier (ou plutôt un chemin d'accès) et renvoie un `bipartite`.

Écrire un programme, que l'on compilera et exécutera en ligne de commande, qui accepte en argument un nom de fichier et affiche le cardinal du couplage renvoyé par `get_maximum_matching` sur le graphe correspondant. Vérifier que vous obtenez bien les valeurs attendues (on laissera de côté les graphes avec $n_1 = n_2 = 10\,000$ pour l'instant : le temps de calcul risque d'être un peu long).

2 Algorithme de Hopcroft-Karp

2.1 Principe de l'algorithme

- Un ensemble $P_1, \dots, P_k$ de chemins est dit *sommet-disjoint* si aucun sommet n'appartient à plusieurs de ces chemins.
- Un ensemble $P_1, \dots, P_k$ de chemins est dit *bloquant* pour un couplage M si :
  - chacun de ces chemins est augmentant pour M ;
  - ces chemins sont sommet-disjoints ;
  - ces chemins sont tous de même longueur l, où l est la longueur minimale d'un chemin augmentant pour M ;
  - cet ensemble est maximal au sens de l'inclusion – il n'existe pas de chemin P de longueur l augmentant pour M et sommet-disjoint avec $P_1, \dots, P_k$.

On remarque que si $P_1, \dots, P_k$ sont des chemins augmentants sommet-disjoints pour un couplage M, alors P_i est augmentant pour $M \oplus P_1 \oplus \dots \oplus P_{i-1}$: on peut donc calculer $M \oplus P_1 \oplus \dots \oplus P_k$ et ajouter k arêtes au couplage. L'algorithme de Hopcroft-Karp exploite cette idée pour calculer un couplage de cardinal maximum avec une meilleure complexité :

Algorithme 1 – Algorithme de Hopcroft-Karp

Entrées : Un graphe biparti non orienté $G = (V, E)$
Sorties : Un couplage M de G de cardinal maximum.

$M \leftarrow \emptyset$
tant que M admet un chemin augmentant **faire**
  Soit $P_1, \dots, P_k$ un ensemble bloquant de chemins pour M.
  $M \leftarrow M \oplus P_1 \oplus \dots \oplus P_k$
renvoyer M

On appelle *phase* de l'algorithme un passage dans la boucle principale. On cherche à présent à :

- proposer une implémentation efficace d'une phase ;
- majorer le nombre de phases pour obtenir une borne sur la complexité totale de l'algorithme.

2.2 Implémentation d'une phase

On note X_f (respectivement Y_f) l'ensemble des sommets libres de X (respectivement Y) pour le couplage M. Pour $s \in V$, on définit $d_a(s)$ comme la longueur d'un plus court chemin *alternant* d'un sommet de X_f à s (qui vaut ∞ si s n'est pas accessible depuis X_f *via* un chemin alternant). On note l la longueur minimale d'un chemin augmentant pour M.

▸ **Question 8** Écrire une fonction `compute_distances` qui prend en entrée un graphe biparti G et un couplage M de G et renvoie :

- `None` si M n'admet pas de chemin augmentant ;
- `Some d` sinon, où `d` est un tableau de longueur n tel que :
  - si $d_a(i) \leqslant l$, alors `d.(i)`$= d_a(i)$;
  - sinon, `d.(i) = -1`.

On exige une complexité en $O(n + p)$.

```
val compute_distances : bipartite -> matching -> int array option
```

On considère alors le sous-graphe de H de G_M obtenu en ne conservant que :

- les sommets s tels que $d_a(s) \leqslant l$;
- les arcs (u, v) tels que $d_a(v) = d_a(u) + 1$.

▶ **Question 9** On considère l'algorithme suivant :

$\mathrm{chemins} \leftarrow \emptyset$
pour $x \in X_f$ **faire**
  Chercher un chemin dans H de x vers Y_f à l'aide d'un parcours (quelconque) de graphe.
  si un tel chemin P existe **alors**
    $\mathrm{chemins} \leftarrow P, \mathrm{chemins}$
    Retirer à H les sommets de P.
renvoyer $\mathrm{chemins}$

Justifier que cet algorithme calcule un ensemble bloquant de chemins pour M. Quelle est sa complexité ?

▶ **Question 10** Modifier cet algorithme pour obtenir une complexité en $O(n+p)$ (en justifiant qu'il reste correct). *Le type de parcours utilisé pour chercher un chemin sera crucial ici.*

▶ **Question 11** Écrire une fonction `compute_blocking_set` prenant en entrée un graphe G et un couplage M et un tableau d tel que celui renvoyé par `compute_distances`, et renvoyant un ensemble bloquant de chemins pour M, sous forme d'une liste de chemins.

```
val compute_blocking_set : bipartite -> matching -> int array -> path list
```

▶ **Question 12** Écrire une fonction `hopcroft_karp` ayant la même spécification que `maximum_matching`, mais utilisant l'algorithme de Hopcroft-Karp.

```
val hopcroft_karp : bipartite -> matching
```

2.3 Majoration du nombre de phases

▶ **Question 13** Soient M et M' deux couplages tels que $|M'| - |M| = k \geqslant 0$. Montrer que $M \oplus M'$ contient au moins k chemins augmentants sommet-disjoints pour M.

Soient M un couplage, $P_1, \dots, P_k$ un ensemble bloquant de chemins pour M et $M' = M \oplus P_1 \oplus \cdots \oplus P_k$. On note l la longueur commune des chemins $P_1, \dots, P_k$, et l'on suppose que M' admet un chemin augmentant P.

▶ **Question 14** Montrer que $M \oplus M' \oplus P$ contient au moins $l(k+1)$ arêtes.

▶ **Question 15** Exprimer $M \oplus M' \oplus P$ en fonction de $P_1 \cup \cdots \cup P_k$ et de P.

▶ **Question 16** En déduire que P est de longueur au moins $l+1$.

▶ **Question 17** Que peut-on en déduire sur la longueur des chemins augmentants trouvés lors des différentes phases de l'algorithme ?

▶ **Question 18** Soit M un couplage dont un plus court chemin augmentant est de longueur l. Montrer que si M' est un couplage, alors $|M'| \leqslant |M| + \frac{|V|}{l+1}$.

▶ **Question 19** Montrer que le nombre d'itérations de la boucle principale de l'algorithme de Hopcroft-Karp est majoré par $2\sqrt{n}$.

▶ **Question 20** En déduire la complexité de l'algorithme de Hopcroft-Karp.

Solutions

Implémentation de l'algorithme du cours

► **Question 1** Le code est simple si (et seulement si !) on réfléchit bien avant de l'écrire. Un chemin $x_0, x_1, \ldots, x_k$ est augmentant si et seulement si :

- $k > 0$;
- x_0 et x_k sont libres ;
- toutes les arêtes $x_{2i+1}x_{2i+2}$ sont dans le couplage (ce qui implique que $x_{2i}x_{2i+1}$ n'y est pas !).

On peut donc vérifier que l'élément en tête est libre, puis traiter les autres sommets deux par deux et vérifier qu'il reste un sommet, libre, à la fin.

```
let is_augmenting m path =
  let rec aux = function
    | x :: y :: xs -> m.(x) = Some y && aux xs
    | [x] -> m.(x) = None
    | [] -> false in
  match path with
  | x :: xs -> m.(x) = None && aux xs
  | _ -> false
```

► **Question 2** Après différence symétrique avec un chemin $x_0, \ldots, x_{2k+1}$, chaque sommet x_{2i} sera apparié avec le sommet x_{2i+1}.

```
let rec delta m path =
  match path with
  | x :: y :: tl ->
    m.(x) <- Some y;
    m.(y) <- Some x;
    delta m tl
  | [] -> ()
  | [x] -> failwith "illformed path"
```

► **Question 3** On suit simplement la définition.

```
let orient g m =
  let n = Array.length g.adj in
  let g' = Array.make n [] in
  for i = 0 to g.n1 - 1 do
    let f j =
      if m.(i) = Some j then g'.(j) <- i :: g'.(j)
      else g'.(i) <- j :: g'.(i) in
    List.iter f g.adj.(i)
  done;
  g'
```

▸ **Question 4** On effectue un parcours en profondeur en stockant l'arbre de parcours dans un tableau (`tree.(i)` indique le parent de `i` dans cet arbre) pour pouvoir reconstruire le chemin. Ce tableau sert aussi ici à identifier les sommets déjà vus.

```
let rec reconstruct tree x =
  if tree.(x) = x then [x]
  else x :: reconstruct tree tree.(x)

exception Found of int

let find_augmenting_path g m =
  let tree = Array.make (Array.length m) (-1) in
  let g' = orient g m in
  let rec explore x =
    if x >= g.n1 && m.(x) = None then raise (Found x);
    let f y =
      if tree.(y) = -1 then (
        tree.(y) <- x;
        explore y
      ) in
    List.iter f g'.(x) in
  try
    for i = 0 to g.n1 - 1 do
      if m.(i) = None then (tree.(i) <- i; explore i)
    done;
    None
  with
  | Found x -> Some (reconstruct tree x)
```

▸ **Question 5** Tout le travail a déjà été fait :

```
let get_maximum_matching g =
  let n = Array.length g.adj in
  let m = Array.make n None in
  let rec loop () =
    match find_augmenting_path g m with
    | None -> m
    | Some p -> delta m p; loop () in
  loop ()
```

▸ **Question 6** Chaque appel à `find_augmenting_path` effectue :

- un appel à `orient`, en temps $O(n+p)$;
- un parcours en profondeur de G_M, à nouveau en temps $O(n+p)$;
- éventuellement un appel à `delta`, en temps proportionnel à la longueur du chemin et donc en $O(n)$.

On a donc du $O(n+p)$ au total pour `find_augmenting_path`, et l'on effectue au plus $n/2$ appels puisque le nombre de sommets couverts augment de deux à chaque fois. La fonction `get_maximum_matching` est donc en $O(n(n+p))$.

▸ **Question 7** Le programme tient en quelques lignes :

```
let () =
  let filename = Sys.argv.(1) in
  let g = read_bipartite filename in
  let c = cardinal (get_maximum_matching g) in
  Printf.printf "Cardinal : %d\n" c
```

Implémentation d'une phase

▶ **Question 8** On effectue un parcours en largeur en initialisant la file avec tous les sommets de X_f, et en interrompant le parcours dès que l'on extrait un sommet de Y_f (tous les sommets à distance l sont alors nécessairement déjà présents dans la file, et l'on a donc déjà écrit les valeurs correspondantes dans le tableau).

```
let compute_distances g m =
  let n = Array.length g.adj in
  let g' = orient g m in
  let d = Array.make n (-1) in
  let q = Queue.create () in
  for i = 0 to g.n1 - 1 do
    if m.(i) = None then (
      Queue.push i q;
      d.(i) <- 0
    )
  done;
  let rec loop () =
    if Queue.is_empty q then None
    else (
      let x = Queue.pop q in
      if x >= g.n1 && m.(x) = None then Some d
      else
        let f y =
          if d.(y) = -1 then (
            d.(y) <- d.(x) + 1;
            Queue.add y q
          ) in
        List.iter f g'.(x);
        loop ()
    ) in
  loop ()
```

▶ **Question 9** Remarquons d'abord que les chemins d'un sommet $x \in X_f$ à un sommet $y \in Y_f$ dans H sont exactement les plus courts chemins augmentants pour M. En effet :

- il s'agit bien d'un chemin augmentant (cas particulier d'un chemin de X_f vers Y_f dans G_M), et sa longueur vaut nécessairement l (puisque $d_a(s)$ augmente de une unité à chaque arc suivi, et que les sommets de Y_f présents dans H vérifient $d_a(y) = l$) ;
- inversement, dans un chemin augmentant *de longueur minimale*, la distance à X_f augmente nécessairement de un à chaque arc suivi, et un tel chemin n'utilise donc que des arcs présents dans H.

Il reste à prouver que l'ensemble de chemins renvoyé est maximal. Pour cela, on considère un plus court chemin augmentant $P = x, \dots, y$ qui ne fait pas partie de l'ensemble renvoyé $\mathcal{S}$:

- si l'un des chemins de $\mathcal{S}$ commence en x, alors P n'est pas sommet-disjoint avec $\mathcal{S}$;
- sinon, cela signifie que Y_f n'était pas accessible depuis x dans H au moment où on a effectué le parcours depuis x. Or les seuls sommets retirés de H sont ceux des chemins déjà choisis : à nouveau, P n'est pas sommet-disjoint avec $\mathcal{S}$.

Pour la complexité, on fait jusqu'à n_1 parcours en profondeur indépendants, chacun s'effectuant en temps $O(n + p)$. On obtient donc du $O(n(n + p))$.

▶ **Question 10** L'idée cruciale est que, à condition d'utiliser un parcours en profondeur, on peut retirer du graphe *tous les sommets explorés* à chaque étape (et pas seulement ceux faisant partie d'un chemin choisi). Plus précisément, pour chaque $x \in X_f$:

- on effectue un parcours en profondeur depuis x, en supprimant (ou en marquant) tous les sommets explorés ;

- dès que l'on atteint un sommet de Y_f, on interrompt le parcours et l'on ajoute le chemin trouvé à l'ensemble (si Y_f n'est pas accessible depuis x, le parcours ira jusqu'au bout et l'on marquera tous les sommets accessibles depuis X_f).

On maintient ainsi l'invariant suivant : quand on marque un sommet v, soit il fera partie d'un chemin sélectionné, soit Y_f n'est pas accessible depuis v par un chemin sommet-disjoint avec ceux déjà choisis. En effet, c'est clairement vrai pour le premier sommet visité (aucun sommet n'est alors marqué), et ensuite :

- si l'on marque le sommet sans l'ajouter à un chemin, c'est que le parcours depuis v a échoué, et donc les éventuels chemins de v à Y_f passent tous par un sommet marqué ;
- supposons alors qu'il existe un chemin de v à Y_f ne passant par aucun sommet choisi, et soit w un sommet marqué de ce chemin. w a été marqué avant v, donc d'après l'invariant il n'existe pas de chemin de w à Y_f n'empruntant pas de sommet choisi : contradiction.

Puisqu'un sommet marqué lors d'un parcours le reste pour les suivants, on ne fait au total qu'un parcours en profondeur du graphe (chaque arc est suivi au plus une fois). On a donc bien une complexité en $O(n + p)$.

► **Question 11** De loin le plus simple est d'utiliser une exception que l'on lève quand on trouve un chemin. On pourrait calculer explicitement le graphe H, mais il est plus simple de travailler avec G_M en ignorant les arcs et sommets absents (le test `d.(y) = d.(x) + 1` garantit également que $d_a(y) \leqslant l$ puisque les autres sommets ont une valeur `-1` dans le tableau `d`).

```
exception Path of int list

let compute_blocking_set g m d =
  let n = Array.length g.adj in
  let seen = Array.make n false in
  let g' = orient g m in
  let rec explore x path =
    if not seen.(x) then (
      seen.(x) <- true;
      if x >= g.n1 && m.(x) = None then raise (Path path);
      let f y = if d.(y) = d.(x) + 1 then explore y (y :: path) in
      List.iter f g'.(x)
    ) in
  let set = ref [] in
  for i = 0 to g.n1 - 1 do
    if m.(i) = None then
      try
        explore i [i]
      with
      | Path p -> set := p :: !set
  done;
  !set
```

► **Question 12** Pas de difficulté majeure :

```
let hopcroft_karp g =
  let n = Array.length g.adj in
  let m = Array.make n None in
  let rec loop () =
    match compute_distances g m with
    | None -> m
    | Some d ->
      let paths = compute_blocking_set g m d in
      List.iter (delta m) paths;
      loop () in
  loop ()
```

Majoration du nombre de phases

▶ **Question 13** On reprend la démonstration du cours :

- le graphe $H = (V, M \oplus M')$ est de degré maximum inférieur ou égal à 2, donc ses composantes connexes sont des sommets isolés, des chemins et des cycles ;
- les cycles sont forcément de longueur paire puisque les arêtes successives doivent alterner entre M et M' – ils contiennent donc autant d'arêtes de M que de M'
- les sommets isolés et les chemins de longueur paire contiennent également autant d'arêtes de chaque couplage ;
- les seules composantes contenant plus d'arêtes de M' que de M sont donc les chemins commençant et se terminant par une arête de M' ;
- chacun de ces chemins est augmentant pour M (on vérifie facilement que les sommets extrêmes sont nécessairement libres pour M) ;
- d'autre part, chacun de ces chemins contient exactement une arête de plus issue de M' que de M – il y en a donc au moins k au total ;
- pour finir, ces chemins étant des composantes connexes de H, ils sont nécessairement sommet-disjoints.

En conclusion, on a donc au moins k chemins augmentants pour M sommet-disjoints, et ces chemins (vus comme des ensembles d'arêtes) sont tous inclus dans $M \oplus M'$.

▶ **Question 14** On a M' est un couplage de cardinal $|M| + k$ et P est augmentant pour M', donc $M' \oplus P$ est un couplage de cardinal $|M| + k + 1$. D'après la question précédente, $M \oplus M' \oplus P = M \oplus (M' \oplus P)$ contient donc un ensemble de $k + 1$ chemins augmentants pour M sommet-disjoints (et donc *a fortiori* disjoints en tant qu'ensembles d'arêtes). Par définition de l, chacun de ces chemins contient au moins l arêtes : on en déduit que $|M \oplus M' \oplus P| \geqslant l(k+1)$.

▶ **Question 15** On a :

$$\begin{aligned}
M \odot M' \oplus P &= M \oplus M \oplus P_1 \oplus \dots P_k \oplus P \\
&= (P_1 \oplus \dots \oplus P_k) \oplus P \\
&= (P_1 \cup \dots \cup P_k) \oplus P && P_1, \dots, P_k \text{ sommet-disjoints}
\end{aligned}$$

▶ **Question 16** Si P est sommet-disjoint avec $P_1, \dots, P_k$, alors P est un chemin augmentant pour M et comme $P_1, \dots, P_k$ est bloquant, on en déduit $|P| \geqslant l + 1$.

On suppose donc à présent que P a un sommet x en commun avec un certain P_i. Les sommets de P_i sont tous couverts par M', donc x n'est pas une extrémité de P, et donc P contient l'arête xy incidente à x dans M' : cette arête appartient à P_i. On a donc $|(P_1 \cup \dots P_k) \cap M'| \geqslant 1$, d'où $|M \odot M' \oplus P| = |(P_1 \cup \dots P_k) \odot P| \leqslant |P_1 \cup \dots P_k| + |P| - 1 = lk + |P| - 1$. En combinant avec $|M \odot M' \oplus P| \geqslant l(k+1)$, on obtient à nouveau $|P| \geqslant l + 1$.

Remarque

Dans les deux cas, on peut en fait conclure que $|P| \geqslant l + 2$ puisqu'un chemin augmentant est toujours de longueur impaire. On pourrait ainsi améliorer la borne prouvée plus loin sur le nombre de phases, mais cela ne change rien à la complexité asymptotique.

▶ **Question 17** Si l'on effectue une phase avec une longueur l de plus court chemin augmentant, on vient de prouver que le nouveau couplage n'admet aucun chemin augmentant de longueur inférieure ou égale à l. Ainsi, l augmente strictement à chaque phase.

▶ **Question 18** Notons $k = |M'| - |M|$. D'après la question 13, on a au moins k chemins augmentants sommet-disjoints pour M, et chacun de ces chemins contient au moins $l + 1$ sommets. On a donc $k(l + 1) \leqslant |V|$, ce qui est précisément le résultat demandé.

► **Question 19** Après i phases, la longueur minimale l d'un chemin augmentant vaut au moins $i+1$ d'après la question 17 (ou alors l'algorithme a déjà terminé). En prenant $i = \lfloor\sqrt{n}\rfloor$, on a donc $l \geqslant \sqrt{n}$. Si l'on considère un couplage de cardinal maximum M^*, on a donc $|M^*| - |M| \leqslant \frac{n}{1+\sqrt{n}} \leqslant \sqrt{n}$ d'après la question précédente. Or chacune des phases suivantes ajoutera au moins une arête au couplage : on terminera donc après au plus $\sqrt{n}$ phases supplémentaires. Au total, le nombre de phases est donc majoré par $2\sqrt{n}$.

► **Question 20** On a vu qu'une phase pouvait être implémentée en temps $O(n+p)$, et l'on vient de montrer qu'il y a $O(\sqrt{n})$ phases. On obtient donc une complexité totale en $O((n+p)\sqrt{n})$, que l'on peut simplifier en $O(p\sqrt{n})$ si l'on suppose G connexe. On gagne un facteur $\sqrt{n}$ par rapport à l'algorithme « naïf ».

En pratique, on observe bien une amélioration nette (sachant que nous n'avons pourtant pas été très soigneux dans notre implémentation, en calculant par exemple deux fois le graphe G_M à chaque étape) :

```
$ ./test.native "data/graph_1000_1000_975.txt"
Cardinal: 975 in 0.30s (naive algorithm)
Cardinal: 975 in 0.00s (Hopcroft-Karp)
$ ./test.native "data/graph_10000_10000_10000.txt"
Cardinal: 10000 in 95.71s (naive algorithm)
Cardinal: 10000 in 0.14s (Hopcroft-Karp)
```

On constate même que l'on gagne bien plus qu'un facteur $\sqrt{n}$ sur ces exemples. Dans le cas de l'algorithme naïf, la complexité trouvée dans le pire cas est en fait essentiellement la complexité moyenne. Pour Hopcroft-Karp en revanche, on [1] peut montrer sous des conditions assez générales que la complexité moyenne est en fait en $O(p \log n)$. Plus modestement, on peut constater que sur les deux exemples fournis de graphes à 10 000 + 10 000 sommets, qui ont été générés de manière à admettre un couplage parfait avec probabilité proche de 1/2, le nombre de phases est très largement inférieur à $2\sqrt{n}$:

```
$ ./test.native "data/graph_10000_10000_9946.txt"
8724 paths of length 1
836 paths of length 3
236 paths of length 5
75 paths of length 7
42 paths of length 9
24 paths of length 11
6 paths of length 13
1 paths of length 15
1 paths of length 17
1 paths of length 19
Cardinal: 9946 in 0.14s (Hopcroft-Karp)
$ ./test.native "data/graph_10000_10000_10000.txt"
9317 paths of length 1
586 paths of length 3
81 paths of length 5
12 paths of length 7
3 paths of length 9
1 paths of length 11
Cardinal: 10000 in 0.15s (Hopcroft-Karp)
```

1. « on », notez bien : moi je ne sais pas faire. . .