UE d'algorithmique (LU3IN003).
Licence d'informatique.

Examen du 8 janvier 2024.

Seule une feuille A4 portant sur les cours et les TD est autorisée, tout autre document est interdit. Téléphones portables éteints et rangés dans vos sacs. Le barème est indicatif et est susceptible d'être modifié. Toutes les réponses doivent être justifiées.

Exercice 1 (4 points)

Répondez aux questions à choix multiples suivantes en cochant la ou les cases correspondant aux bonnes réponses. Chaque bonne réponse aux questions 1 à 4 apportera 0.5 point, tandis que chaque mauvaise réponse retranchera 0.25 point à votre note. Chaque bonne réponse aux questions 5 et 6 apportera 1 point, tandis que chaque mauvaise réponse retranchera 0.5 point à votre note. La note de l'exercice sera le maximum entre 0 et la somme des points obtenus.

Questions	**Réponses**
1. Quel algorithme parmi les suivants est de type "diviser pour régner" ?	☐ algorithme de Huffman ■ tri fusion ☐ ordonnancement d'intervalles pondérés ☐ algorithme de Dijkstra
2. Parmi les algorithmes suivants, lequel n'est pas basé sur un parcours ?	☐ parcours en largeur ☐ algorithme de Prim ■ algorithme de Kruskal ☐ algorithme de Dijkstra
3. Quel algorithme pouvez-vous utiliser pour détecter un circuit dans un graphe orienté ?	☐ parcours en largeur ■ parcours en profondeur ☐ algorithme de Dijkstra ☐ algorithme de Bellman
4. Quel algorithme faut-il privilégier pour calculer un plus court chemin dans un graphe orienté *sans circuit* dans lequel tous les coûts sont *positifs* ?	■ algorithme de Bellman ☐ algorithme de Prim ☐ algorithme de Bellman-Ford ☐ algorithme de Dijkstra
5. Quelle est la complexité d'un algorithme du type "diviser pour régner" qui divise un problème de taille n en deux sous-problèmes de taille $n/2$ et dont l'opération de fusion est en $\Theta(n^2)$?	☐ $\Theta(n)$ ☐ $\Theta(n \log n)$ ■ $\Theta(n^2)$ ☐ $\Theta(n^3)$
	suite sur la page suivante...

Questions	Réponses
6. Exécuter l'algorithme de Dijkstra sur le graphe G_1 de la figure 1, afin de trouver une arborescence des plus courts chemins de racine 1. Dans quel ordre les sommets sont-ils examinés ? Vous *représenterez* également sur le graphe G_1 *l'arborescence des plus courts chemins* retournée par l'algorithme de Dijkstra (vous pouvez surligner les arcs sélectionnés).	☐ (1,2,3,4,5,6,7) ■ (1,4,2,5,3,6,7) ☐ (1,2,4,5,3,6,7) ☐ (1,4,2,5,6,3,7)

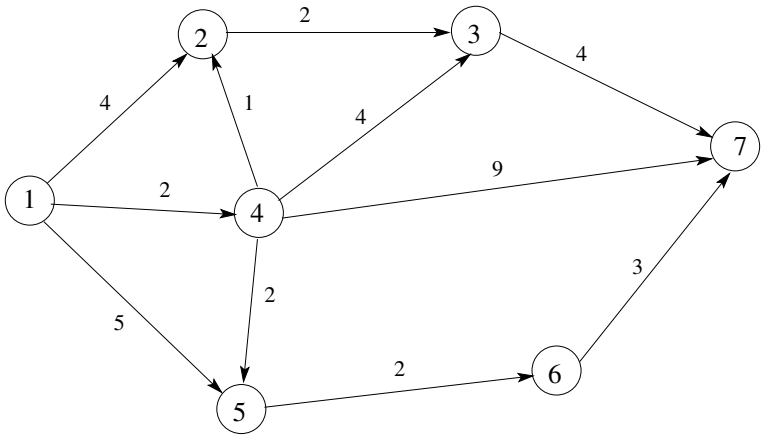

FIGURE 1 – Graphe G_1.

Arbre retourné par l'algorithme : {1,3}, {4,2}, {4,5}, {2,3}, {5,6} et au choix {5,6} ou {6,7}.

Exercice 2 (7 points)

Vous souhaitez concevoir une projection de photos pour vos ami(e)s. Supposons que vous ayez déjà classé n photos $\{1, \ldots, n\}$ dans l'ordre dans lequel vous souhaitez les projeter. Vous avez maintenant la possibilité de mettre une ou deux photos par affichage lors de la projection. Mettre deux photos côté à côte peut en effet faire un joli effet, par exemple si les photos ont le même sujet. Pour $i \in \{1, \ldots, n-1\}$, vous notez $v_i \in \{0, 1\}$ la valeur de l'effet attendu si les photos i et $i+1$ sont projetées côte à côte : si $v_i = 1$, placer les photos i et $i+1$ côte à côte provoque un joli effet, et si $v_i = 0$, placer les photos i et $i+1$ côte à côte ne provoque pas de joli effet. Votre but est de maximiser le nombre de jolis effets pendant la projection de photos.

0	1	1	0	1	1	1

FIGURE 2 – Exemple de tableau V pour une projection de 8 photos, avec $V[i] = v_i$. Placer les photos 1 et 2 côte à côte ne crée pas de joli effet ($V[1] = 0$), alors que placer les photos 2 et 3 côte à côte crée un joli effet ($V[2] = 1$).

Votre petite sœur vous propose de considérer les photos dans l'ordre de 1 à n et de mettre côte à côte deux photos dès qu'elles créent un joli effet. Par exemple, dans le tableau V de la figure 2, vous placerez les photos 2 et 3 côte à côte puis les photos 5 et 6 côte à côte, suivies des photos 7 et 8 côte à côte.

Question 1 (2.5/7) — Prouver que cet algorithme maximise bien le nombre de jolis effets. Vous veillerez à être rigoureux/rigoureuse dans votre preuve.

Remarque : si lors de la projection, l'affichage des photos obtenu avec l'algorithme induit k jolis effets, on pourra noter $E = \{e_1, \ldots, e_k\}$ l'ensemble des photos placées avec une autre photo et à gauche, lors de la projection (i.e. $e_i = k$ si les photos k et $k+1$ sont projetées côte à côte). Dans l'exemple plus haut, on aurait eu $E = \{2, 5, 7\}$ car les jolis effets concernent les photos 2 et 3, présentées ensemble, puis les photos 5 et 6, puis les photos 7 et 8.

Notons $E = \{e_1, \ldots, e_k\}$ l'ensemble des jolis effets obtenus avec notre algorithme. Supposons par l'absurde que cet algorithme n'est pas optimal (i.e. ne maximise pas le nombre de jolis effets). Soit $O = \{o_1, \ldots, o_{k'}\}$ une solution optimale *telle que le nombre de jolis effets identiques entre E et O au début de la présentation est maximisé*. Autrement dit, s'il existe plusieurs solutions optimales, O est celle (ou l'une de celles) qui sélectionne les mêmes jolis effets que E le plus longtemps possible.
Comme E n'est pas un ensemble optimal, nous en déduisons que $E \neq O$. Soit p le plus petit indice auquel les jolis effets sélectionnés par notre algorithme glouton et l'algorithme optimal diffèrent : on a $e_i = o_i$ pour tout $i < p$, et $e_p \neq o_p$.
L'effet e_p se produit avant l'effet o_p, par construction (parmi les effets d'indice supérieur à $e_{(p-1)}$, l'algorithme glouton a sélectionné le premier). Supposons maintenant que l'on remplace dans O l'effet o_p par l'effet e_p. La solution est toujours réalisable (puisque $o_{p-1} = e_{p-1}$ est compatible avec l'effet e_p et l'effet e_p a lieu avant l'effet o_p et est donc compatible avec les effets de numéros o_i avec $o_i > o_p$). Ainsi, on a créé une solution $O' = O \setminus \{o_p\} \cup \{e_p\}$ qui contient le même nombre de jolis effets que O et qui est donc optimale. Dans cette solution, les p premiers jolis effets (au moins) sont les mêmes que les p premiers jolis effets de E. Comme O est une solution optimale maximisant le nombre de premiers jolis effets en commun avec E, nous arrivons à une contradiction. L'ensemble des jolis effets E est donc optimal.

On suppose maintenant que tous les jolis effets ne se valent pas : certains sont plus intéressants que d'autres. Pour $i \in \{1, \ldots, n-1\}$, on notera $v_i \in \mathbb{N}^+$ la valeur de l'effet attendu si les photos i et $i+1$ sont projetées côte à côte. Si $v(i) = 0$, il semble équivalent de passer les photos i et $i+1$ chacune sur une page, ou de les placer côte à côte. Si $v_i > 0$, alors placer les photos côte à côte provoque un joli effet, et plus $v(i)$ est important, plus l'effet est joli.

Vous souhaitez concevoir un algorithme de programmation dynamique permettant de déterminer quelles paires de photos présenter côte à côte (en respectant l'ordre initial choisi) de manière à maximiser la somme des effets attendus : $\sum_{i \in \{1,\ldots,n-1\} | i \text{ et } i+1 \text{ sont placées côte à côte}} v_i$.

Dans un premier temps, nous chercherons uniquement à calculer la somme maximale des effets attendus – on notera cette valeur OPT –, et pas à savoir quelles photos placer côte à côte.

Pour tout $i \in \{2, \ldots, n\}$, notons $S(i)$ la valeur maximale des effets que l'on peut obtenir en considérant les photos de 1 à i uniquement (ces photos étant placées dans l'ordre $1, 2, \ldots, i$). On fixera $S(1) = 0$ (il n'y a pas d'effet si l'on considère une photo uniquement).

Question 2 (0.5/7) — Exprimer OPT en fonction des valeurs $S(i)$.

$OPT = S(n)$.

Question 3 (1.5/7) —

a) Exprimer, en fonction d'une ou plusieurs valeur(s) $S(k)$, avec $k < i$ la valeur de $S(i)$ si l'on sait que la photo i est projetée seule.
b) Exprimer, en fonction d'une ou plusieurs valeur(s) $S(k)$, avec $k < i$ la valeur de $S(i)$ si l'on sait que la photo i est projetée côte à côte avec la photo $i-1$.
c) Donner une formule de récurrence permettant de calculer la valeur $S(i)$, pour tout $i \in \{3, \ldots, n\}$.

Considérons la photo i : soit on la projette seule, auquel cas la valeur maximale des effets correspond à $S(i-1)$, soit la projette côte à côte avec la photo $i-1$, auquel cas la valeur maximale des effets est $S(i-2) + v(i-1)$. Ainsi, pour tout $i > 2$:

$$S(i) = \max\{S(i-1); S(i-2) + v(i-1)\}.$$

Question 4 (1.5/7) — Écrire un algorithme de programmation dynamique calculant OPT. Quelle est la complexité de votre algorithme ?

Calcule_OPT(n, tableau v indiquant les valeurs $v(i)$)
Créer tableau S de n cases. //On pourrait utiliser deux valeurs seulement si on voulait optimiser...
$S[1] = 0$
$S[2] = \max(0, v[1])$
Pour i allant de 3 à n
    $S[i] = \max(S[i-1], S[i-2] + v[i-1])$
Retourner $S[n]$

Complexité : $O(n)$.

Question 5 (1/7) — Modifier votre algorithme pour qu'il affiche les pages côte à côte. Par exemple, il affichera "(2,3) (6,7)" si dans la solution optimale les photos 2 et 3 puis 6 et 7 sont affichées côte à côte. Vous pouvez pour cela ajouter une fonction supplémentaire.

On remplace l'instruction "Retourner $S[n]$" par FonctionAffiche(S, v, n), où la fonction Affiche est la suivante :

```
FonctionAffiche(S, v, i) :
si (i = 2 et v[1] > 0)
    afficher (1, 2)
si i > 2
    si S[i-1] < S[i-2] + v[i-1]
        FonctionAffiche(S,v, i-2)
        afficher("(i-1, i)") ;
    sinon
        FonctionAffiche(S,v, i-1)
```

Exercice 3 (11 points)

Étant donné un graphe $G = (S, A)$, un *approximant* (V, R) est un couple de parties disjointes de A tel qu'il existe un arbre couvrant de coût minimum OPT qui contient toutes les arêtes de V et aucune arête de R.

Ainsi, chaque arête de $e \in A$ peut soit appartenir à V, soit à R, soit ni à V ni à R. Nous dirons par la suite que les arêtes de V sont vertes, et celles de R sont rouges. Les arêtes de A qui ne sont ni dans V ni dans R seront dites libres, et considérées comme blanches. Un approximant est donc un sous-ensemble d'arêtes de A colorées en vert et en rouge, tel qu'il existe un arbre couvrant de coût minimum contenant l'ensemble des arêtes vertes et aucune arête rouge.

Exemple : sur la figure ci-dessous, (V, R) est un approximant avec $V = \{\{A, B\}, \{A, C\}\}$ (arêtes en gras plein) et $R = \{\{B, C\}\}$ (arête en pointillé). En effet, il existe un arbre couvrant de coût minimum $(\{\{A, B\}, \{A, C\}\}, \{C, D\})$ contenant les arêtes de V et pas celle de R.

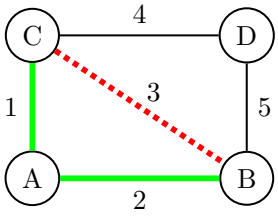

FIGURE 3 – Exemple d'approximant.

Question 1 (3/11) — Exécuter l'algorithme de Kruskal sur le graphe G_2 représenté dans la figure 4. Vous indiquerez, à chaque itération de votre algorithme, un approximant correspondant à l'exécution de l'algorithme. Vous surlignerez également l'arbre couvrant de coût minimum obtenu à la fin de l'exécution de l'algorithme.

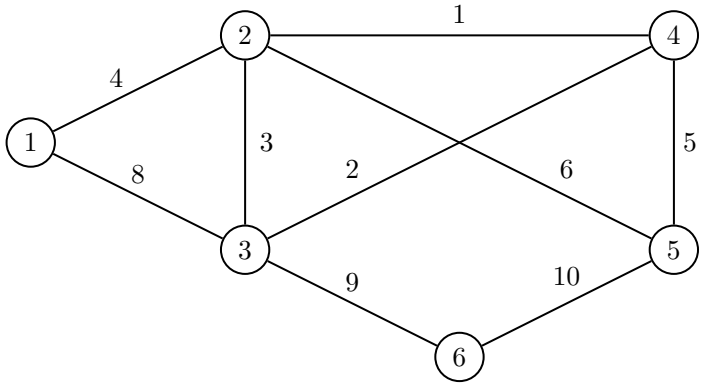

FIGURE 4 – Graphe G_2.

Les questions suivantes concernent *un graphe non orienté, connexe, et pondéré* $G = (S, A)$ *quelconque* (et non le graphe G_2 de la question précédente).

Question 2 (1/11) — Soit (V, R) un approximant de G.

a) Montrer que le graphe $G' = (S, V)$ est une forêt couvrante de G.

b) Que peut-on dire si $|V| = n - 1$? Justifiez votre réponse.

a) $G' = (S, V)$ est une forêt couvrante de G car tous les sommets de G sont dans G' et les arêtes de V sont un sous-ensemble d'arêtes de G qui appartiennent à un arbre couvrant et donc qui ne forment pas de cycle.

b) Puisqu'un arbre couvrant contient $n - 1$ arêtes, et puisqu'il y a n sommets, si $|V| = n - 1$ alors V est un arbre couvrant. Comme il est constitué uniquement d'arêtes qui appartiennent à un même arbre couvrant de coût minimum, c'est que c'est un arbre couvrant de coût minimum.

Deux règles permettent d'ajouter une arête (verte ou rouge) à un approximant : *la règle des cocycles*[1] et *la règle des cycles*. Ces règles permettent de créer un algorithme générique de construction d'arbre couvrant de coût minimum : en partant de l'approximant $(\emptyset, \emptyset)$ on ajoute peu à peu une arête rouge ou une arête verte jusqu'à obtenir un arbre couvrant de coût minimum. Nous verrons que les algorithmes de Prim et de Kruskal sont des cas particuliers de cet algorithme générique.

Dans cet exercice, vous prouverez la règle des cocycles (questions 3 à 6), et admettrez la règle des cycles. En utilisant ces règles, vous prouverez dans les questions 7 et 8 que les algorithmes de Prim et de Kruskal retournent bien des arbres couvrants de coût minimum.

La règle des cocycles.

Étant donné un approximant (V, R), la règle des cocycles va nous permettre d'ajouter une arête verte à V en colorant en vert l'arête blanche la moins chère du cocycle :

> **Règle des cocycles :** Soit $G = (S, A)$ un graphe connexe non orienté et pondéré. Soit (V, R) un approximant de G, et soit ω un cocycle dont chaque arête est soit rouge soit blanche. Soit $e \in A$ une arête blanche de ω de coût minimum. $(V \cup \{e\}, R)$ est un approximant.

Les questions suivantes concernent un graphe $G = (S, A)$ connexe non orienté et pondéré quelconque. Soit (V, R) un approximant de G et soit OPT un arbre couvrant de coût minimum de G tel que OPT contient les arêtes de V mais pas celles de R. Supposons qu'il existe dans G un cocycle ω formé d'arêtes rouges (i.e. de R) ou blanches (i.e. de $A \setminus \{V \cup R\}$).

Question 3 (0.5/11) — Pourquoi peut-on être sûr qu'il existe nécessairement une arête blanche dans ω ?

> Par l'absurde : si ce n'était pas le cas toutes les arêtes de ω seraient rouges et donc OPT ne serait pas connexe. Or G est connexe et OPT est un arbre couvrant de G et est donc forcément connexe.

Soit $e = \{x, y\}$ une arête blanche de ω de coût minimum.

Question 4 (0.5/11) — La taille d'un approximant (V, R) est $|V \cup R|$: le nombre d'arêtes colorées en vert ou en rouge. Donner un approximant de taille $|V| + |R| + 1$ si $e \in OPT$. On ne demande pas de justification.

> Si $e \in OPT$, alors $V \cup \{e\} \subset OPT$. Ainsi, $(V \cup \{e\}, R)$ est un approximant.

Supposons maintenant que $e \notin OPT$.

Question 5 (1/11) — Soit γ la chaîne qui relie x à y, les deux extrémités de e, dans l'arbre couvrant OPT. Montrer que γ contient nécessairement une arête f qui est blanche et telle que $c(f) \geq c(e)$.

> Puisque $\gamma \cup \{e\}$ forme un cycle, γ contient nécessairement au moins une arête f de ω distincte de e. Cette arête est blanche car elle appartient en même temps à ω (qui ne contient pas d'arête verte) et à OPT (qui ne contient pas d'arête rouge). On a de plus $c(f) \geq c(e)$ car e est l'arête blanche de coût minimum de ω.

1. rappel : Soit $S' \subset S$ un sous ensemble des sommets d'un graphe $G = (S, A)$. Le *cocycle* $\omega(S')$ associé à S' est l'ensemble des arêtes qui ont une extrémité dans S' et une extrémité qui n'est pas dans S'.

Question 6 (2/11) — Donner un approximant de taille $|V|+|R|+1$ si $e \notin OPT$. Justifiez votre réponse.

Indice : on pourra considérer le graphe partiel $X = (OPT \setminus \{f\}) \cup \{e\}$.

Considérons maintenant le graphe partiel OPT' issu de OPT dans lequel on remplace l'arête f par l'arête e : $OPT' = (OPT \setminus \{f\}) \cup \{e\}$. On sait que OPT' est connexe (car on a obtenu OPT' en retirant une arête à un cycle de $OPT \cup \{e\}$, un graphe connexe), et que OPT' possède $n-1$ arêtes : OPT' est donc un arbre couvrant de G. De plus, OPT' a un coût égal à $c(OPT) - c(f) + c(e)$. Comme $c(f) \geq c(e)$ et comme OPT est un arbre couvrant de coût minimum, on en déduit que $c(f) = c(e)$ et OPT' est également un arbre couvrant de coût minimum. On sait de plus que les arêtes de V, ainsi que e appartiennent à OPT', et les arêtes de R n'appartiennent pas à OPT'. Ainsi, $(V \cup \{e\}, R)$ est un approximant.

Découvrons maintenant la règle des cycles, qui permet d'ajouter une arête rouge à un approximant.

La règle des cycles.

La règle des cycles consiste à dire qu'étant donné un approximant et un cycle ne contenant pas d'arête rouge, on peut colorer en rouge l'arête blanche la plus chère du cycle et garder un approximant. Plus formellement :

Règle des cycles : Soit $G = (S, A)$ un graphe connexe non orienté et pondéré. Soit (V, R) un approximant de G, et soit $\mathcal{C}$ un cycle dont chaque arête est soit verte soit blanche. Soit $e \in A$ une arête blanche de $\mathcal{C}$ de coût maximum. $(V, R \cup \{e\})$ est un approximant.

Par manque de temps, on admettra cette règle sans la prouver. Vous pourrez donc supposer par la suite que les règles des cycles et des cocycles ont toutes les deux été prouvées.

Un algorithme générique.

Les algorithmes de Prim et de Kruskal, comme d'autres, sont des algorithmes gloutons qui reposent sur l'application de la règle des cycles et des cocycles. En effet, tout algorithme glouton qui procède par itération, en colorant à chaque étape une arête en rouge ou en vert, en appliquant la règle des cycles ou des cocycles, va, au bout d'un certain nombre d'itérations, retourner un arbre couvrant de coût minimum.

Question 7 (1/11) — Rappeler en quelques mots le principe de l'algorithme de Prim en expliquant en quoi cet algorithme utilise la règle des cocycles et/ou la règle des cycles. En déduire la validité de l'algorithme de Prim.

L'algorithme de Prim consiste tout simplement à construire un arbre T de manière incrémentale, en commençant par un arbre réduit à seul sommet $s \in S$. Il applique $n-1$ fois la règle des cocycles sur le cocycle des sommets de T : à chaque étape est ajoutée à T l'arête de plus petit coût de ce cocycle. À chaque étape, on a donc un approximant (V, R), où V sont les arêtes de T et $R = \emptyset$. Cet algorithme retourne donc bien un arbre couvrant de coût minimum.

Question 8 (2/11) — Rappeler en quelques mots le principe de l'algorithme de Kruskal en expliquant en quoi cet algorithme utilise la règle des cocycles et/ou la règle des cycles. En déduire la validité de l'algorithme de Kruskal.

L'algorithme de Kruskal applique la règle des cycles et la règle des cocycles, en partant de l'approximant $(\emptyset, \emptyset)$. Supposons qu'à l'étape k l'algorithme examine l'arête $\{x, y\}$.

— Si x et y appartiennent à la même composante connexe de la forêt couvrante en construction – que l'on appellera K –, alors il existe un cycle, formé des arêtes de la chaîne entre x et y dans $K(= V)$, et de l'arête $\{x, y\}$. Toutes les arêtes de ce cycle sont vertes sauf $\{x, y\}$ qui est blanche. L'algorithme de Kruskal ne sélectionne pas cette arête, qu'il ne considérera alors plus : il applique donc la règle des cycles en ajoutant $\{x, y\}$ à R.

— Si x et y n'appartiennent pas à la même composante connexe de K : appelons C_x l'ensemble des sommets de la composante connexe dans laquelle se trouve x. Le cocycle de C_x est constitué d'arêtes blanches (pas encore considérées par l'algorithme) ou rouges (déjà examinées mais pas sélectionnées par l'algorithme). L'arête $\{x, y\}$ de ce cocycle est blanche et de coût minimum car, par construction, c'est la première arête blanche de la liste triée. Cette arête est sélectionnée (i.e. ajoutée à V). Ainsi, l'algorithme applique ici la règle des cycles.

L'algorithme de Kruskal applique donc m fois soit la règle des cycles soit la règle des cocycles à partir de l'approximant $(\emptyset, \emptyset)$: l'ensemble d'arêtes V retourné est bien un arbre couvrant de coût minimum de G.