Δημήτρης Κοκκίνης 03118896
Δεκέμβριος, 2024

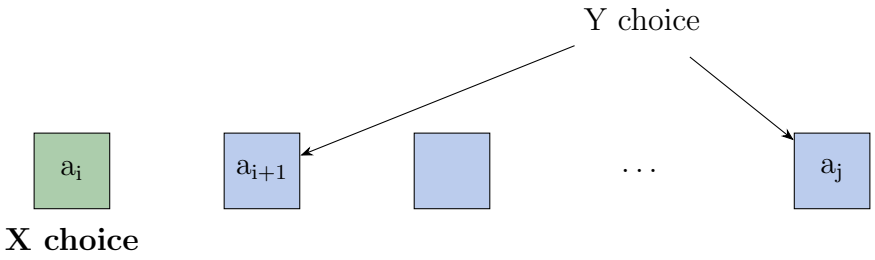
Αλγόριθμοι και Πολυπλοκότητα
2η Σειρά Γραπτών Ασκήσεων

Άσκηση 1: Παιχνίδι με Κάρτες

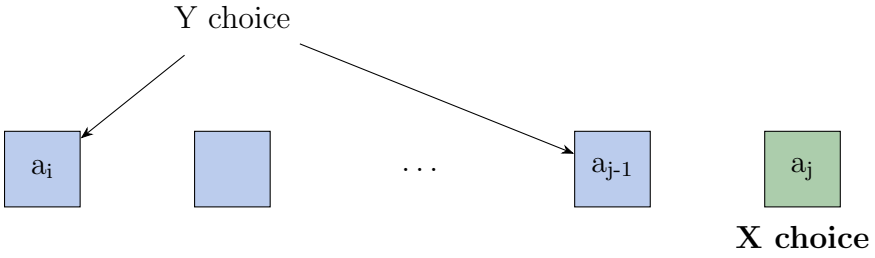
Όπως φαίνεται και απο το παράδειγμα που παρέχεται στο τέλος της άσκησης, το άπληστο κριτήριο της επιλογής της κάρτας με το μεγαλύτερο αριθμό ανάμεσα στις δύο, αποτυγχάνει να οδηγήσει σε μέγιστο κέρδος για τους παίκτες. Θα λύσουμε την άσκηση χρησιμοποιώντας δυναμικό προγραμματισμό.

Κάθε φορά που είναι η σειρά ενός παίκτη, αυτός έχει δύο επιλογές στη διάθεσή του: Να διαλέξει την πρώτη κάρτα ή να διαλέξει τη τελευταία κάρτα. Γνωρίζοντας βέβαια πως και ο άλλος παίκτης παίζει με σκοπό το μέγιστο κέρδος. Άρα μπορούμε να πούμε πως, κάθε παίκτης παίρνει την απόφασή του για να μεγιστοποιήσει το συνολικό του κέρδος ή απλά να ελαχιστοποιήσει το κέρδος του αντιπάλου του. Επομένως, ανάλογα με την επιλογή που κάνει ο κάθε παίκτης, για μια ακολουθία καρτών απο i έως j έχουμε τις εξής εκβάσεις:

1. Ο παίκτης (έστω X) διαλέγει τη πρώτη κάρτα i, με κέρδος a_i. Στη περίπτωση αυτή, ο αντίπαλός του θα πρέπει να διαλέξει ανάμεσα στην (i+1) κάρτα ή στην j κάρτα και προφανώς θα διαλέξει αυτή που θα αφήσει το ελάχιστο κέρδος για τον άλλον. Οπότε, ο παίκτης X με την απόφασή του αυτή θα έχει συνολικό κέρδος το a_i + το ελάχιστο ανάμεσα στο κέρδος που θα είχε αν ο αντίπαλος διαλέξει τη πρώτη κάρτα ή τη τελευταία κάρτα.



2. Ο παίκτης διαλέγει τη τελευταία κάρτα με κέρδος a_j. Ο αντίπαλος αυτή τη φορά έχει να διαλέξει ανάμεσα στη κάρτα i και στη κάρτα j-1, οπότε το κέρδος του παίκτη απο την επιλογή αυτή θα είναι το a_j + το ελάχιστο απο το κέρδος του για την ακολουθία καρτών (i+1, j-1) (στη περίπτωση που ο αντίπαλος διαλέξει τη πρώτη κάρτα) και (i, j-2) (στη περίπτωση που ο αντίπαλος διαλέξει τη τελευταία κάρτα).



Ορίζουμε έτσι την αναδρομική συνάρτηση $F[i, j]$, η οποία επιστρέφει το μέγιστο κέρδος που μπορεί να λάβει ο παίκτης για την ακολουθία καρτών απο i έως j. Σύμφωνα με τα παραπάνω, αν ο παίκτης διαλέξει τη πρώτη

κάρτα της ακολουθίας θα έχει κέρδος $F[i,j] = a_i + min(F[i+2,j], F[i+1,j-1]$ και αν διαλέξει τη τελευταία κάρτα θα έχει κέρδος $F[i,j] = a_j + min(F[i+1,j-1], F[i,j-2]$. Προφανώς, ο παίκτης θα διαλέξει τη κάρτα που θα του αποδώσει το μέγιστο κέρδος, έτσι το πρόβλημα αυτό μπορεί να εκφρασθεί μέσω της αναδρομικής σχέσεως ως:

$$F[i,j] = \begin{cases} a_i & \text{if} \quad i = j \\ max(a_i, a_j) & \text{if} \quad i+1 = j \\ max\{a_i + min(F[i+2,j], F[i+1,j-1], a_j + min(F[i+1,j-1], F[i,j-2]\} & otherwise \end{cases}$$

Υλοποιώντας τη σχέση αυτή με τη μέθοδο bottom-up θα γεμίσουμε έναν πίνακα $F[n][n]$. Το μέγιστο εφικτό κέρδος για τον παίκτη X θα είναι το τελευταίο στοιχείο του πίνακα και το μέγιστο κέρδος για τον παίκτη Y θα είναι η διαφορά του αθροίσματος της αρχικής ακολουθίας με το κέρδος του παίκτη X.

Algorithm

1. Array a[n] with the cost of each card
2. Initialize array $F[n][n]$ with zeros
3. **if** $i = j$

   $F[i][j] = a[i]$
4. **if** $j = i + 1$

   $F[i][j] = max(a[i], a[j])$
5. **for** $iter = 0$ **to** n

   for $i = 0, j = iter$ **to** $j < n$

   $first = a[i] + min(F[i+2][j], F[i+1, j-1])$

   $last = a[j] + min(F[i+1][j-1], F[i][j-2])$

   $F[i][j] = max(first, last)$

Η χρονική πολυπλοκότητα του αλγόριθμου είναι $\mathcal{O}(n^2)$.

Άσκηση 2: Ταξιδεύοντας στον Κόσμο

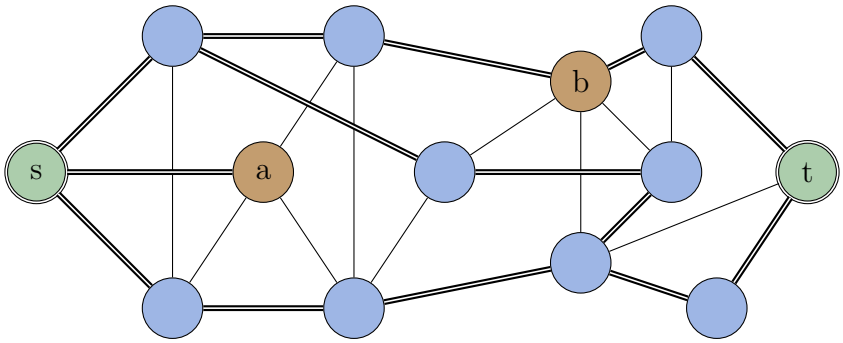
Για να λύσουμε το πρόβλημα, θα χρησιμοποιήσουμε τον αλγόριθμο Dijkstra, ο οποίος βρίσκει τα συντομότερα μονοπάτια απο έναν αρχικό κόμβο **s** προς όλους τους άλλους κόμβους.

Έχουμε ένα μη-κατευθυνόμενο γράφημα $G(V, E, \vec{c}, \vec{\tau})$, με κόμβους V, ακμές E και κόστη $\vec{c}$, $\vec{\tau}$. Εφόσον, υπάρχουν αρκετές ακμές οι οποίες οδηγούν σε συντομότερο μονοπάτι $s-t$ με βάση το κόστος ακμών c, ψάχνουμε ένα υποσύνολο των ακμών αυτών, οι οποίες ελαχιστοποιούν επίσης και το συντομότερο μονοπάτι $a-b$ με βάση το κόστος τ. Επομένως, θα εκτελέσουμε δύο φορές τον αλγόριθμο Dijkstra, μια για να βρούμε το συντομότερο $s-t$ μονοπάτι και μια για να βρούμε το συντομότερο $a-b$ μονοπάτι.

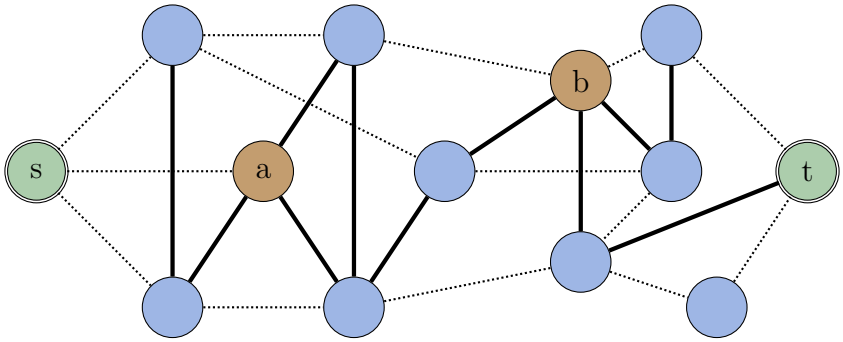
Στόχος του αλγόριθμου θα είναι αρχικά η εύρεση ενός υποσυνόλου ακμών, έστω G', οι οποίες επιτρέπουν απεριόριστες μετακινήσεις μεταξύ των πόλεων **s** και **t** με το ελάχιστο κόστος c. Στη συνέχεια, μεταξύ των ακμών αυτών G' θέλουμε να ελαχιστοποιήσουμε το επιπρόσθετο κόστος μετακίνησης τ μεταξύ των πόλεων $a-b$.

Έστω πως έχουμε το παρακάτω σιδηροδρομικό δίκτυο πόλεων, το οποίο αναπαριστάνεται απο κόμβους και

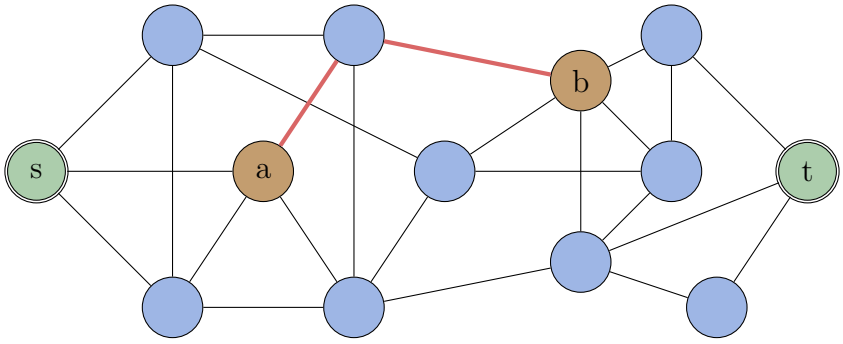
ακμές. Στο γράφημα αυτό, έστω G' έχουμε σημειώσει με διπλή γραμμή τις ακμές οι οποίες ανήκουν στο δένδρο συντομότερων μονοπατιών Dijkstra, με βάση το κόστος c. Όπως φαίνεται και απο το γράφο, υπάρχουν παραπάνω απο ένα συντομότερα μονοπάτια απο τη πόλη s στη πόλη t.



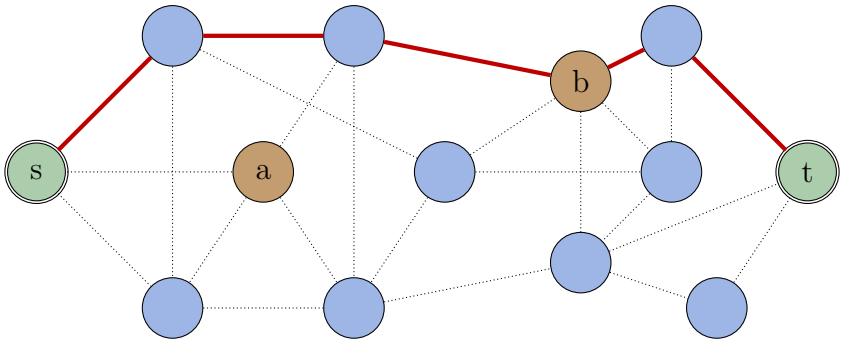
Η ιδέα του αλγόριθμου είναι η εξής: Εφόσον έχουμε βρεί το σύνολο ακμών G', ξανατρέχουμε Dijkstra αλλά αυτή τη φορά τροποποιούμε το αρχικό γράφημα, ώστε οι ακμές που ανήκουν στο G' να έχουν μηδενικό βάρος και όλες οι υπόλοιπες ακμές να έχουν βάρος τ, όπως φαίνεται και στο σχήμα, όπου με διακεκομμένες γραμμές φαίνονται οι ακμές μηδενικού βάρους. Έτσι ο αλγόριθμος Dijkstra θα βρεί το συντομότερο μονοπάτι $a - b$ το οποίο προφανώς θα περιέχει (εφόσον είναι συντομότερο) και ακμές μηδενικού βάρους (ή μονο μηδενικού βάρους αν κάτι τέτοιο είναι εφικτό).



Για παράδειγμα, ένα πιθανό συντομότερο μονοπάτι απο a μέχρι b θα μπορούσε να είναι



Το μονοπάτι αυτό περιέχει μια ακμή ελαχίστου κόστους c και μία μόνο μετάβαση κόστους τ. Επομένως, αν υποθέσουμε πως το συντομότερο μονοπάτι $a-b$ είναι αυτό, το συντομότερο $s-t$ με τον επιπρόσθετο αυτό προορισμό θα είναι.



Δηλαδή, απο το σύνολο των πιθανών καρτών που θα μπορούσαμε να είχαμε διαλέξει, διαλέξαμε τις κάρτες αυτές όπου ελαχιστοποιούν και το μονοπάτι $a-b$.

Algorithm

1. Initialize graph **G**
2. G'_{edges} = **Dijkstra(G, s, t, c)** *//Find shortest paths from s to t with Dijkstra*
3. **for each** edge e in G

   if e in G'_{edges}

   $e = 0$

   else

   $e = \tau(e)$
4. Find shortest a-b path in G' using **Dijkstra(G', a, b, τ)**
5. Find the shortest s-t path that contains the G'_{edges} in the shortest a-b path

Ο αλγόριθμος αυτός, αν υλοποιήσουμε τον αλγόριθμο Dijkstra με binary heap θα έχει πολυπλοκότητα $\mathcal{O}(mlogn)$ εφόσον τρέχουμε δύο φορές τον αλγόριθμο Dijkstra, αλλιώς αν υλοποιηθεί με Fibonacci heap θα είναι $\mathcal{O}(m+nlogn)$.

Άσκηση 3: Επίλυση Συστήματος Ανισοτήτων

(a)

Αρχικά θα προτείνω μια λύση η οποία είναι κάπως ανορθόδοξη και πηγάζει απο απλή παρατήρηση των σχέσεων των ανισοτήτων. Το σύστημα S αποτελείται απο το σύνολο m των ανισοτήτων της μορφής $x_i - x_j \leq b_{ij}$ όπου $i \geq 1 \quad \kappa\alpha\iota \quad j < n$. Οπότε για n μεταβλητές θα έχουμε $m = n^2$ πιθανές ανισότητες. Για παράδειγμα, για 3 μεταβλητές (x_1, x_2, x_3) το σύστημα S θα είναι:

$$\begin{matrix} & & \text{c} \\ \begin{pmatrix} (x_1 - x_1 \leq b_{11}) & (x_1 - x_2 \leq b_{12}) & (x_1 - x_3 \leq b_{13}) \\ (x_2 - x_1 \leq b_{21}) & (x_2 - x_2 \leq b_{22}) & (x_2 - x_3 \leq b_{23}) \\ (x_3 - x_1 \leq b_{31}) & (x_3 - x_2 \leq b_{32}) & (x_3 - x_3 \leq b_{33}) \end{pmatrix} \end{matrix}$$

Κάποιες παρατηρήσεις που μπορούμε να κάνουμε είναι οι εξής:

- Οι ανισότητες στη κύρια διαγώνιο, δεν μπορούν να είναι μικρότερες απο 0 καθώς είναι η διαφορά του αριθμού με τον εαυτό του.
- Πρέπει να ισχύει ότι $b_{ji} \geq -b_{ij}$ για $i \neq j$

Η δεύτερη παρατήρηση προκύπτει απο το γεγονός πως αν ισχύει $x_i - x_j \leq b_{ij}$, αυτό σημαίνει πως οι ακέραιοι αριθμοί x_i και x_j έχουν απόσταση μεγαλύτερη ή ίση του b_{ij} και ανάλογα με το πρόσημο του b_{ij}, είτε είναι ο x_i μεγαλύτερος είτε ο x_j. Έστω $b_{ij} < 0$, τότε ο x_j είναι μεγαλύτερος απο τον x_i κατα $|b_{ij}|$. Επομένως, εφόσον ο x_j είναι κατά $|b_{ij}|$ μεγαλύτερος του x_j, η διαφορά $x_j - x_i$ δεν μπορεί να είναι μικρότερη απο $|b_{ij}|$, και αφού άλλαξε η σειρά των αριθμών τώρα έχουμε θετικό πρόσημο.

Αυτή η παρατήρηση μας επιτρέπει να ελέγξουμε την ικανοποιησιμότητα των ανισοτήτων $x_j - x_i \leq b_{ji}$, έχοντας ως δεδομένο πως ικανοποιούνται οι ανισότητες $x_i - x_j \leq b_{ij}$. Η βασική ιδέα λοιπόν είναι, θεωρώντας ως δεδομένες κάποιες ανισότητες, να μπορούμε να βασιστούμε σε κάποιες συνθήκες οι οποίες επιτρέπουν να θεωρήσουμε και τις υπόλοιπες ανισότητες ως ικανοποιήσιμες. Τις υπόλοιπες αυτές συνθήκες θα τις λάβουμε, προσθέτοντας κατα μέλη ανισότητες για τις οποίες έχουμε βρει πως ικανοποιούνται.

Για παράδειγμα, έστω πως θέλουμε να ελέγξουμε αν ισχύει η ανισότητα $x_3 - x_2 \leq b_{32}$. Θεωρώντας ως δεδομένο πως ικανοποιείται η ανισότητα $x_3 - x_1 \leq b_{31}$ (απο τη παρατήρηση $\{2\}$ θα είναι $b_{31} \geq -b_{13}$ το οποίο λαμβάνουμε ως δεδομένο), και παίρνοντας επίσης ως δεδομένο την ανισότητα $x_1 - x_2 \leq b_{12}$, προσθέτουμε κατα μέλη και προκύπτει:

$$\begin{array}{rr} & x_3 - x_1 \leq b_{31} \\ + & x_1 - x_2 \leq b_{12} \\ \hline \\ & x_3 - x_2 \leq (b_{31} + b_{12}) \\ = & x_3 - x_2 \leq (b_{12} - b_{13}) \end{array}$$

Με αυτό το τρόπο, μπορούμε να θεωρήσουμε ως δεδομένες τις ανισότητες $x_1 - x_j \leq bj$ για $1 \leq j \leq n$ και να αναγάγουμε τις υπόλοιπες ανισότητες σύμφωνα με αυτές, ώστε να παρέχουμε μια βάση, η οποία θα αποτελεί συνθήκη για να ικανοποιούνται. Έτσι, μπορεί να φτιαχτεί ένα πινακάκι $n(n-1)$ το οποίο θα περιέχει τις τιμές τις οποίες θα πρέπει να ικανοποιούν οι ανισότητες.

Για παράδειγμα: Έστω για 3 μεταβλητές (x_1, x_2, x_3). Θεωρούμε πως $b_{11} = 0, b_{12} = b_1, b_{13} = b_2$. Σύμφωνα με τα παραπάνω, φτιάχνουμε το εξής πινακάκι.

$$\text{B} = \begin{bmatrix} \mathbf{-b_1} & \mathbf{-b_2} \\ \mathbf{0} & \mathbf{(b_1 - b_2)} \\ \mathbf{(b_2 - b_1)} & \mathbf{0} \end{bmatrix}$$

Ο πίνακας B περιέχει τις τιμές των ακεραίων b, για τις οποίες αν είναι μεγαλύτεροι, τότε θα ικανοποιούνται οι αντίστοιχες ανισότητες. Δηλαδή, αν ισχύει για κάθε b_{ij}

$$\begin{Bmatrix} b_{21} \geq -\mathbf{b_1} & b_{31} \geq -\mathbf{b_2} \\ b_{22} \geq \mathbf{0} & b_{32} \geq (\mathbf{b_1} - \mathbf{b_2}) \\ b_{23} \geq (\mathbf{b_2} - \mathbf{b_1}) & b_{33} \geq \mathbf{0} \end{Bmatrix}$$

, τότε το σύνολο S θα είναι ικανοποιήσιμο. Ακολουθώντας την ίδια διαδικασία για παραπάνω μεταβλητές, παρατήρησα ένα μοτίβο ως προς το πως φτιάχνεται το πινακάκι. Έστω για παράδειγμα, πως έχουμε 4 μεταβλητές (x_1, x_2, x_3, x_3) και ορίζουμε ως $b_{11} = 0 = b_0, b_{12} = b_1, b_{13} = b_2, b_{14} = b_3$. Τότε, το πινακάκι μπορεί να φτιαχτεί με τον εξής τρόπο:

$$\begin{bmatrix} \{0 - b_1\} \to -\mathbf{b_1} & \{0 - b_2\} \to -\mathbf{b_2} & \{0 - b_3\} \to -\mathbf{b_3} \\ \{b_1 - b_1\} \to \mathbf{0} & \{b_1 - b_2\} \to (\mathbf{b_1} - \mathbf{b_2}) & \{b_1 - b_3\} \to (\mathbf{b_1} - \mathbf{b_3}) \\ \{b_2 - b_1\} \to (\mathbf{b_2} - \mathbf{b_1}) & \{b_2 - b_2\} \to \mathbf{0} & \{b_2 - b_3\} \to (\mathbf{b_2} - \mathbf{b_3}) \\ \{b_3 - b_1\} \to (\mathbf{b_3} - \mathbf{b_1}) & \{b_3 - b_2\} \to (\mathbf{b_3} - \mathbf{b_2}) & \{b_3 - b_3\} \to \mathbf{0} \end{bmatrix}$$

Παρατηρούμε πως γενικεύεται για παραπάνω μεταβλητές: Ένα στοιχείο x_{ij} του πίνακα, ισούται με $b_i - b_j$, έχοντας ορίσει αρχικά κατάλληλα τα $b_0, .., b_n$.

Algorithm

1. Initialize array S[n][n] with integers b_{ij}
2. Initialize array $B = (b_0, b_1, ..., b_n)$ where $b_0 = b_{11}$, $b_1 = b_{12}$, $b_2 = b_{13}$,.., $b_n = b_{1n}$
3. **for** $i = 0$ **to** n

   for $j = 2$ **to** n

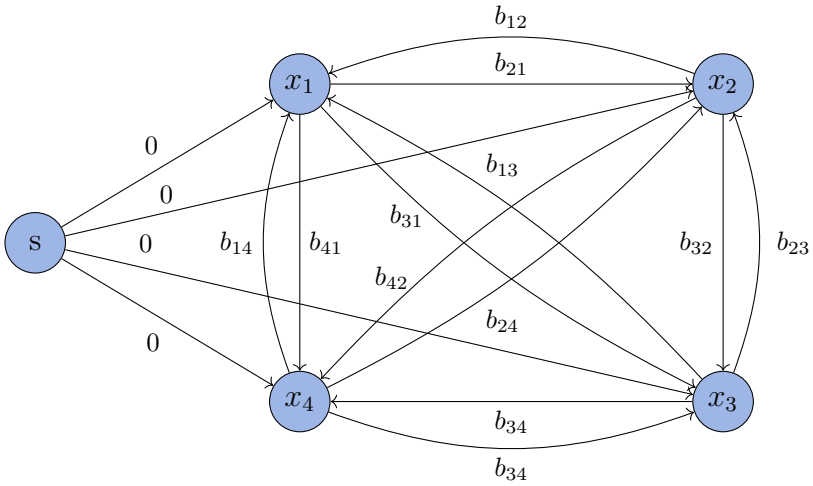
   if $S[i][j] < B[i] - B[j]$

   break;
4. **return True**

Ο παραπάνω αλγόριθμος έχει χρόνο εκτέλεσης $\mathcal{O}(n^2)$, ωστόσο απλά ελέγχει την ικανοποιησιμότητα του συστήματος S και δεν βρίσκει αποδεκτές τιμές για τις μεταβλητές. Επίσης, αναφέρεται στη περίπτωση όπου έχουμε όλες τις πιθανές ανισότητες($x_1 - x_j \leq b_{1j}$ στις οποίες θα βασιστούμε.

Μια άλλη παρατήρηση που οδηγεί σε ενα διαφορετικό τρόπο επίλυσης είναι πως η ανισότητα του συστήματος S μοιάζει αρκετά με τη τριγωνική ανισότητα που ικανοποιούν οι ακμές ενός γράφου $d(s, u) \leq d(s, v) + w(v, u)$. Επομένως, θα μπορούσαμε να αναπαριστάνουμε το σύστημα S με ένα γράφο με $n + 1$ κόμβους, όσες και οι μεταβλητές μαζί με έναν αρχικό βοηθητικό κόμβο s. Οι ακέραιες μεταβλητές x_i θα ορίζουν τη συντομότερη διαδρομή απο τον s στον x_i κόμβο, οπότε θα θεωρούμε μεταβλητή x_i ως τη συντομότερη διαδρομή $d(s, x_i)$ στο γράφο. Αντίστοιχα, τους ακέραιους b_{ij} θα τους θεωρήσουμε ως ακμές ανάμεσα στους κόμβους.

Για παράδειγμα, για 4 μεταβλητές και για 12 ανισότητες έχουμε το εξής γράφημα:



Ορισμός του G

i Οι κορυφές του αντιστοιχούν στις μεταβλητές $(x1, ..., x_n)$

ii Για κάθε ανισότητα του συστήματος S, $x_i - x_j \leq b_{ij}$, ο γράφος περιέχει μια ακμή απο το κόμβο x_j προς τον κόμβο x_i

Ο τρόπος που ορίσαμε τις ακμές, είναι προφανής αν αναλογιστούμε την τριγωνική ανισότητα.

$$\mathbf{x_i - x_j \leq b_{ij} \implies x_i \leq x_j + b_{ij}}$$

Το οποίο, σύμφωνα με το γράφημα μπορεί να γραφεί ως:

$$\mathbf{d(s, x_i) \leq d(s, x_j) + w(x_j, x_i)}$$

Άρα η ακμή για το b_{ij} έχει κατεύθυνση απο το x_j στο x_i

Μια ικανή και αναγκαία συνθήκη για είναι το σύστημα S ικανοποιήσιμο, είναι το συνεπαγόμενο κατευθυνόμενο γράφημα G να μην περιέχει αρνητικό κύκλο. Όπως είπαμε, οποιαδήποτε λύση x_i ορίζει μια εκτίμηση της ελάχιστης απόστασης απο τον αρχικό κόμβο μέχρι το κόμβο s στο γράφημα G. Αν υπάρχει αρνητικός κύκλος, οι αποστάσεις αυτές θα ήταν ασυνεπείς, οπότε το σύστημα δεν θα ήταν ικανοποιήσιμο. Για να ελέγχξουμε την ύπαρξη αρνητικού κύκλου αλλά και να βρούμε έγκυρες τιμές για τις μεταβλητές x_i, μπορούμε να τρέξουμε τον αλγόριθμο Bellman-Ford για τα συντομότερα μονοπάτια απο το κόμβο s προς όλους τους άλλους κόμβους.

Algorithm

1. Construct graph $G(S, n, m)$
2. $has\_negative\_cycle, distances$ = **Bellman_Ford(G)**
3. **if** $has\_negative\_cycle$
   return False
4. **else**
   $x_i = distances$

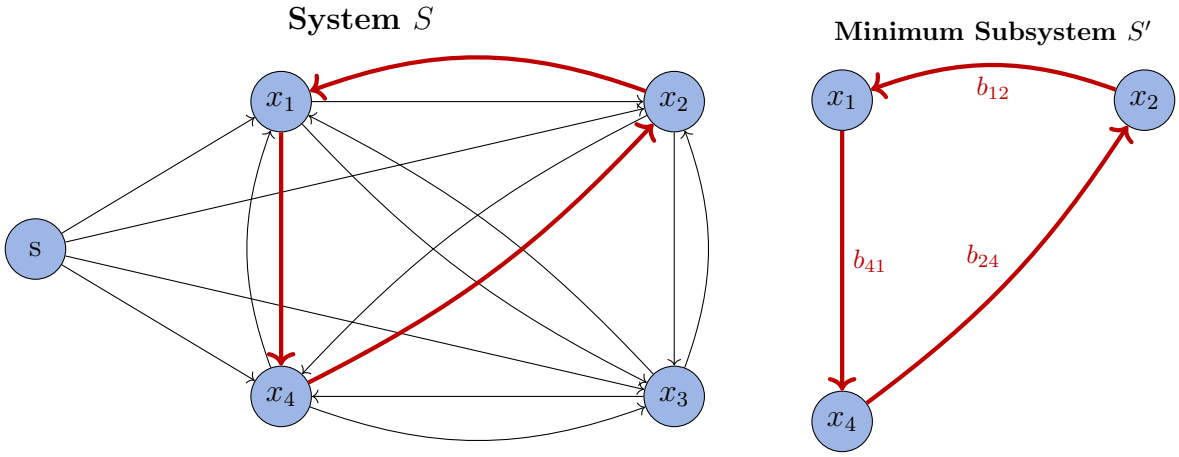
Bellman-Ford

1. **for** each vertex in G
   $distance[V] \leftarrow infinite$
   $previous[V] \leftarrow NULL$
2. $distance[S] < -0$
3. cycles = *False*
4. **for** each vertex V in G
   for each edge (U,V) in G
   $tempDistance \leftarrow distance[U] + edge\_weight(U, V)$
   if $tempDistance < distance[V]$
   $distance[V] \leftarrow tempDistance$
   $previous[V] \leftarrow U$
5. **for** each edge (U,V) in G
   if $distance[U] + edge\_weight(U, V) < distance[V]$
   cycles = *True*
6. **return cycles, distance[]**

Η πολυπλοκότητα του αλγόριθμου θα είναι $\mathcal{O}(nm)$ στη χειρότερη περίπτωση

(b)

Στη περίπτωση που το σύστημα S δεν είναι ικανοποιήσιμο, έτσι όπως ορίσαμε το πρόβλημα, αυτό σημαίνει πως υπάρχει τουλάχιστον ένας αρνητικός κύκλος στο γράφημα, ο οποίος συμβολίζει ένα υποσύνολο ανισοτήτων που χαλάνε τη συνθήκη ικανοποιησιμότητας. Επομένως, το ελάχιστο, ως προς το πλήθος ανισοτήτων, υποσύστημα S' που δεν είναι ικανοποιήσιμο, θα είναι ο μικρότερος, ως το πλήθος των ακμών του, αρνητικός κύκλος στο γράφο G που ορίστηκε προηγουμένως.



Οπότε, αλλάζουμε τον αλγόριθμο του ερωτήματος (a), έτσι ώστε αν βρεθεί αρνητικός κύκλος, τότε να βρίσκει μέσω του βοηθητικού πίνακα των προκατόχων κάθε κόμβου, τον μικρότερο αρνητικό κύκλο στο γράφο. Τροποποιούμε, λοιπόν, τον αλγόριθμο Bellman-Ford, έτσι ώστε στη τελευταία επανάληψη, αν βρεθεί ακμή που ανήκει σε αρνητικό κύκλο, να ανακατασκευάζει τον κύκλο με τον πίνακα των προκατόχων και να επιστρέφει τον μικρότερο κύκλο, ο οποίος είναι και το ζητούμενο σύστημα.

Algorithm

1. $smallest\_cycle = None$
2. $smallest\_cycle\_length = \infty$
3. **for** each edge (U, V)

   if $distances[U] + w < distances[V]$ *//Negative cycle*

   $cycle =$ **reconstruct_cycle**(**predecessors**, **V**)

   if $len(cycle) < smallest\_cycle\_length$

   $smallest\_cycle = cycle$

   $smallest\_cycle\_length = len(cycle)$
4. $S = smallest\_cycle$
5. **return S**

όπου η συνάρτηση $reconstruct\_cycle(predecessors, V)$, επιστρέφει τον αρνητικό κύκλο που ξεκινάει απο τον κόμβο V. Η πολυπλοκότητα του Bellman-Ford δεν αλλάζει, όμως τρέχουμε για κάθε πιθανό κόμβο τη συνάρτηση

ανακατασκευής κύκλου, οπότε η πολυπλοκότητα θα είναι $\mathcal{O}(n^2m)$.

Ένας διαφορετικός τρόπος είναι να εκμεταλλευτούμε τον αλγόριθμο Bellman-Ford ως δυναμικό προγραμματισμό. Ορίζοντας την αναδρομική συνάρτηση

$$D(i, z) = \text{lenght of shortest path from starting node s to z using at most i edges,}$$

η αναδρομική σχέση για το πρόβλημα των συντομότερων μονοπατιών ορίζεται ως:

$$D(i, s) = min(D(i-1, z), min(D(i-1, y) + w(y, z)) \quad \text{for all y with edges leading to z)}$$

Με αυτό το τρόπο, φτιάχνουμε ένα πινακάκι $B[n+1][n]$, με τόσες γραμμές όσες επαναλήψεις εκτελεί και ο αλγόριθμος Bellman-Ford. Για να βρούμε τον υπογράφο με αρνητικό κύκλο, μπορούμε, ξεκινώντας απο τη τελευταία γραμμή του πίνακα, να ελέγχουμε ποιός αριθμός αλλάζει απο τη προηγούμενη επανάληψη/γραμμή. Αυτός ο κόμβος τότε θα ανήκει στον αρνητικό κύκλο και θα ακολουθούμε την ίδια διαδικασία και για τον προκάτοχό του που θα έχουμε αποθηκεύσει σε ξεχωριστό πίνακα.

(c)

Όπως και στις προηγούμενες περιπτώσεις, έτσι και τώρα αναπαριστάνουμε το σύστημα ως ένα γράφο με κορυφές $V = 1, ..., n$ και ακμές που αντιστοιχούν στις ανισότητες. Στη συγκεκριμένη περίπτωση όμως, κάθε ακμή (j, i) έχει βάρος w_{ij} και ετικέτα b_{ij}. Όπως βρήκαμε και προηγουμένως, ένα υποσύστημα ανισώσεων που δεν είναι ικανοποιήσιμο, αντιστοιχεί σε έναν αρνητικό κύκλο στο γράφημα. Το πρόβλημα αυτή τη φορά είναι η εύρεση ενός αρνητικού κύκλου με το ελάχιστο βάρος των ακμών του.

Για αυτό το σκοπό, θα τροποποιήσουμε τον αλγόριθμο Bellman-Ford με δυναμικό προγραμματισμό, προσθέτοντας παράμετρο, έτσι ώστε να αποθηκεύεται το ελάχιστο βάρος που θα έχει συναντήσει μέχρι στιγμής για κάθε κορυφή. Ορίζουμε έτσι τη συνάρτηση:

$$dist(i, k, w)$$

Η οποία επιστρέφει *true* αν υπάρχει μονοπάτι απο την αρχική κορυφή **s** μέχρι τη κορυφή **i** με ακριβώς **k** ακμές και συνολικό βάρος **w**.

Algorithm

1. Initialize graph $G = (V, E)$
2. Initialize dist[n][m][m] as False except dist[s][0][0] = True
3. **for** i = 0 **to** n: *//n is the number of nodes*

   for each edge (j, i)

   for w = 0 **to** W *//W is sum of all weights*

   if $dist[j][k][w] ==$ **True**

   if $(x_i - x_j \leq b_{ij})$: $dist[i][k+1][w + w_{ij}] =$ **True**
4. Check if there are i and w such that $dist[i][m][w] ==$ **True**

Αν βρεθεί αρνητικός κύκλος, μπορούμε να τον ανακατασκευάσουμε με backtracking απο το πίνακα dist. Για τη κατασκευή του γραφήματος χρειαζόμαστε χρόνο $\mathcal{O}(m)$ όπου m ο αριθμός των ακμών. Ο DP Bellman-Ford έχει πολυπλοκότητα $\mathcal{O}(n * m * W)$. Αν θεωρήσουμε ότι το συνολικό βάρος W είναι πολυωνυμικό ως προς το μέγεθος της εισόδου, τότε ο αλγόριθμος είναι ψευδοπολυωνυμικός. Στη χειρότερη περίπτωση ωστόσο, ο αλγόριθμος θα έχει εκθετική πολυπλοκότητα ως προς το μέγεθος της εισόδου. Ήταν αναμενόμενο να μην έχουμε καλό χρόνο, διότι γενικά η εύρεση αρνητικού κύκλου ελάχιστου βάρους σε γράφους είναι ένα NP-hard πρόβλημα.

Άσκηση 4: Μετατροπή Ροής Ακμών σε Ροή Μονοπατιών

Η σχέση ανάμεσα στη ροή ακμών και στη ροή μονοπατιών μας λέει πως για κάθε ακμή e του γράφου, η συνολική ροή που περνά από την ακμή e, πρέπει να είναι ίση με το άθροισμα της ροής όλων των μονοπατιών **p** που περιλαμβάνουν την ακμή e. Οπότε η ροή **f** μπορεί να προκύψει απο κατάλληλη κατανομή της ροής **g** στα διάφορα **s-t** μονοπάτια. Η βασική ιδέα είναι να βρίσκουμε επαναληπτικά έγκυρα μονοπάτια στον υπολειμματικό γράφο. Η επανάληψη του αλγόριθμου θα σταματάει μόλις $f(e) = \Sigma_{p \in P: e \in p} g(p)$ για όλες τις ακμές του γράφου, δηλαδή όταν ο υπολειμματικό γράφος δεν έχει s-t μονοπάτια, καθώς κάθε αποσύνθεση θα μειώνει την υπολειμματική χωρητικότητα κάποιας ακμής και έχουμε το πολύ **m** ακμές.

Algorithm

1. Initialize residual graph $\mathbf{G_{residual}}$

   for each edge $e \in E$

   $c_r(u, v) \leftarrow c(u, v) - f(u, v)$ *// forward edge*

   $c_r(v, u) \leftarrow f(u, v)$ *// backward edge*

2. Initialize path flow $g(p) \leftarrow 0$ **for** all s-t paths p
3. Initialize set P' to store paths with positive flow
4. **while** s-t path exists in $G_{residual}$

   Find s-t path p *//BFS*

   Compute $\Delta f \leftarrow min(c_r(e))$ **for** all edges $e \in p$ *//bottleneck capacity*

   Update path flow

   $g(p) \leftarrow g(p) + \delta f$

   Update residual graph

   for each edge $e = (u, v) \in p$ in the initial graph

   $c_r(u, v) \leftarrow c_r(u, v) - \Delta f$

   if $capacity = 0$

   remove edge from $G_{residual}$

   $c_r(v, u) \leftarrow c_r(v, u) + \Delta f$

5. **return** g and P'

Σε κάθε επανάληψη, βρίσκουμε ένα νέο μονοπάτι στο υπολειμματικό δίκτυο και επειδή αφαιρούμε τουλάχιστον μια ακμή απο το υπολειμματικό δίκτυο, χρειαζόμαστε το πολύ m επαναλήψεις για να βρούμε όλα τα μονοπάτι και να τελειώσει ο αλγόριθμος, όπου m είναι ο αριθμός των ακμών στο αρχικό δίκτυο. Επομένως, το σύνολο P' θα περιέχει το πολύ m μονοπάτια με θετική ροή.

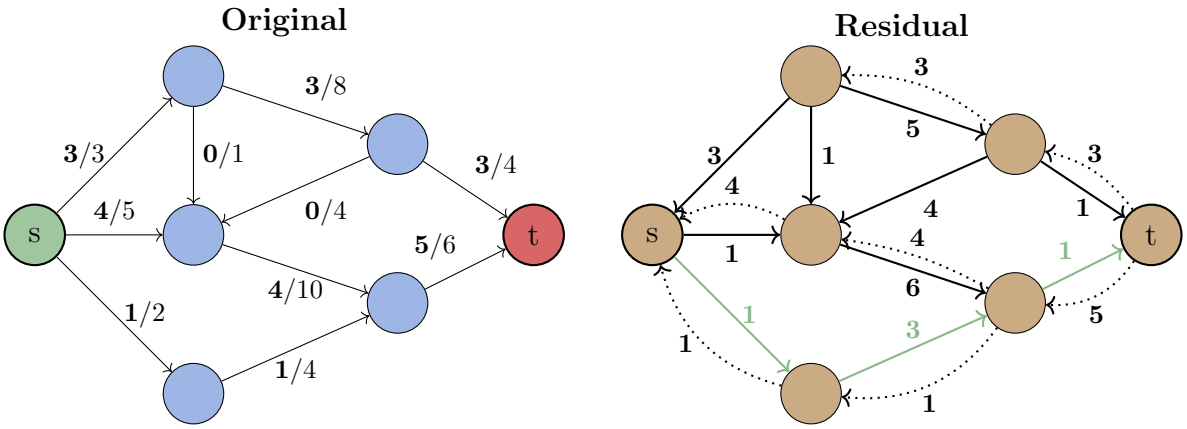
Για τη κατασκευή του υπολειμματικού δικτύου χρειαζόμαστε $\mathcal{O}(m)$, για την εύρεση του μονοπατιού s-t μπορούμε να χρησιμοποιήσουμε είτε BFS είτε DFS σε χρόνο $\mathcal{O}(m+n)$ χρόνο και επειδή εκτελούμε m επαναλήψεις, συνολικά ο αλγόριθμός μας θα έχει πολυπλοκότητα $\mathcal{O}(m(m+n))$.

Άσκηση 5: Επιβεβαίωση και Αναπροσαρμογή Μέγιστης Ροής

(a)

Για να ελέγξουμε αν η **f** είναι μέγιστη **s-t** ροή στο δίκτυο **G(V,E,c)**, θα βασιστούμε στο θεώρημα μέγιστης ροής-ελάχιστης τομής, σύμφωνα με το οποίο, μια **s-t** ροή **f** είναι μέγιστη αν και μόνο αν δεν υπάρχει **s-t** μονοπάτι με θετική υπολειπόμενη χωρητικότητα στο υπολοιπόμενο γράφημα.

Επομένως, μπορούμε να υπολογίζουμε την υπολειπόμενη χωρητικότητα για κάθε ακμή $e = (u, v)$ στο δίκτυο και να αναζητούμε αν υπάρχει **s-t** μονοπάτι στο υπόλοιπο γράφημα $G_{residual}$ με θετική υπολειπόμενη χωρητικότητα. Αν δεν βρεθεί τέτοιο μονοπάτι, τότε η δοσμένη ροή είναι όντως η μέγιστη, αλλιώς υπάρχει καλύτερο μονοπάτι με περιθώριο, μέσω του οποίου μπορούμε να στείλουμε περισσότερη ροή.



Όπως φαίνεται και στο απλό παράδειγμα, η δοσμένη ροή δεν είναι μέγιστη καθώς εξακολουθούν να υπάρχουν μονοπάτια απο τον **s** έως τον **t** στον υπολειπόμενο γράφο.

Algorithm

1. Initialize $G_{residual} = empty$
2. **for each** edge $e = (u, v)$ in G

   add edge (u, v) to $G_{residual}$ with capacity $c_{residual}(e) = c(e) - f(e)$

   add reverse edge (u, v) to $G_{residual}$ with capacity $c_{residual}(e_{reverse}) = f(e)$
3. BFS **from** s in $G_{residual}$
4. **if** there exists a path **from** s **to** t **in** $G_{residual}$

   return False
5. **else**

   return True

Η υπολογιστική πολυπλοκότητα της κατασκευής του γραφήματος υπολοίπων είναι $\mathcal{O}(m)$ καθώς εξετάζουμε κάθε ακμή του G, ο BFS παίρνει $\mathcal{O}(m+n)$, οπότε η συνολική πολυπλοκότητα του γραφήματος είναι $\mathcal{O}(m+n)$, που είναι γραμμική.

(b)

Αρχικά θα ενημερώσουμε την υπολειμματική χωρητικότητα της ακμής $e = (u, v)$ για να προκύψει το γράφημα G'. Αν η ροή της ακμής ξεπερνάει τη χωρητικότητά της, προφανώς η **f** δεν είναι πλέον εφικτή και πρέπει να μειώσουμε τη ροή στην ακμή e και να αναπροσαρμόσουμε τη ροή στο δίκτυο.

Αυτό που θα κάνουμε ουσιαστικά, θα είναι να διαγράψουμε την επιπλέον ροή απο την ακμή e και να την αναδιανέμουμε στο δίκτυο. Μετά την κατασκευή του γράφου G', θα αναζητήσουμε μονοπάτια **s-t** με θετική υπολειμματική χωρητικότητα στο υπόλοιπο γράφημα $\mathbf{G'_{residual}}$, και θα αυξάνουμε την ροή χρησιμοποιώντας τη διαδικασία αύξησης όπως στον Ford-Fulkerson.

Algorithm

1. Update capacity of edge e

   $c'(e) = c(e) - k$

2. **if** $f(e) > c'(e)$

   $\Delta f = f(e) - c'(e)$

   reduce f(e) to c'(e)

   distribute Δf into the residual graph

   add Δf to $f(v, u)$ *//reverse flow*

3. BFS in the residual graph G'

4. **while** an augmenting path exists

   augment the flow along the path

   update the residual capacities

5. **return** updated flow f

Συνολικά, ο αλγόριθμος έχει πολυπλοκότητα $\mathcal{O}(m + n)$.