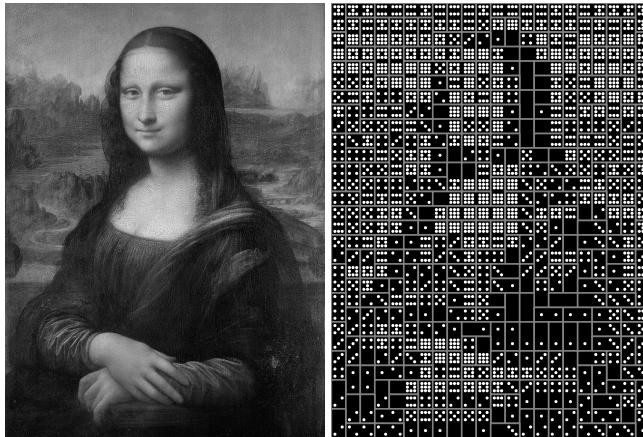


*Av. Καθηγητής Π. Λουρίδας
Τμήμα Διοικητικής Επιστήμης και Τεχνολογίας
Οικονομικό Πανεπιστήμιο Αθηνών*

Πορτραίτα Ντόμινο



Στην εργασία αυτή θα ξεκινήσετε από την επίλυση ενός γενικού προβλήματος βελτιστοποίησης και θα προχωρήσετε σε μία ίσως απρόσμενη εφαρμογή του, με την οποία θα παράγεται πορτραίτα όπως το επάνω δεξιά. Είναι καλύτερο να το δείτε από κάποια απόσταση.

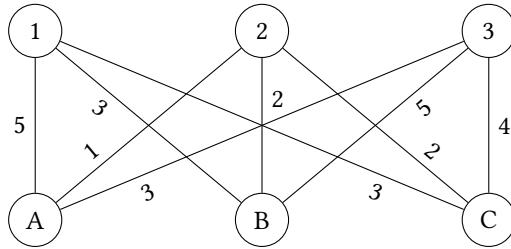
Τέλεια Ταιριάσματα

Έχουμε ένα σύνολο εργασιών και ένα σύνολο επαγγελματιών που μπορούν να τις διεκπεραιώσουν. Ο κάθε επαγγελματίας κοστολογεί διαφορετικά τις εργασίες και μπορεί να αναλάβει μόνο μία από αυτές. Εμείς θέλουμε να αναθέσουμε όλες τις εργασίες, μία σε κάθε επαγγελματία, ώστε να πετύχουμε το ελάχιστο συνολικό κόστος.

Για παράδειγμα, στον παρακάτω πίνακα έχουμε την Alice, τον Bob, και την Carol, οι οποίοι μας έχουν δώσει τις προσφορές τους για τον ελαιοχρωματισμό τριών κτιρίων.

| | Κτίριο 1 | Κτίριο 2 | Κτίριο 3 |
|-------|----------|----------|----------|
| Alice | 5000 | 1000 | 3000 |
| Bob | 3000 | 2000 | 5000 |
| Carol | 3000 | 2000 | 4000 |

Ποιος είναι ο οικονομικότερος συνδυασμός επαγγελματιών και εργασιών ώστε κάθε επαγγελματίας να αναλάβει ένα κτίριο, και κάθε κτίριο να ανατεθεί σε έναν επαγγελματία;



Εικόνα 1: Διμερής Γράφος

Το πρόβλημα αυτό είναι ένα παράδειγμα του λεγόμενου *Πρόβλημα της Ανάθεσης Γραμμικού Αθροίσματος* (linear sum assignment problem). Σε αυτό, έχουμε έναν πίνακα, ο οποίος ονομάζεται **πίνακας κόστους** (cost matrix), και θέλουμε να αντιστοιχίσουμε κάθε μία γραμμή σε μία και μόνο μία στήλη, ελαχιστοποιώντας το κόστος των κελλιών του πίνακα που ορίζονται από την αντιστοιχίση γραμμών και στηλών. Οι λέξεις του ονόματος προκύπτουν από τη φύση του προβλήματος. Θέλουμε να κάνουμε μια ανάθεση (από γραμμές σε στήλες), το κόστος κάθε επιλογής προστίθεται στο συνολικό κόστος, και το συνολικό κόστος ως απλό άθροισμα των επιμέρους μερών συνολικό κόστος είναι γραμμικός συνδυασμός τους.

Στη συνέχεια θα δούμε τον τρόπο με τον οποίο λύνουμε τέτοιου είδους προβλήματα. Στο συγκεκριμένο πάντως, βέλτιστη λύση είναι η Alice να αναλάβει το κτίριο 2, ο Bob το κτίριο 1, και η Carol το κτίριο 3, με συνολικό κόστος 8000.

Το πρόβλημα αυτό μπορεί να μοντελοποιηθεί χρησιμοποιώντας έναν **διμερής γράφο** (bipartite graph). Σε έναν διμερή γράφο $G = (U \cup V, E)$, οι κόμβοι χωρίζονται σε δύο σύνολα U, V , τα οποία δεν έχουν κοινά στοιχεία, και όλοι οι σύνδεσμοι συνδέουν έναν κόμβο από το ένα σύνολο με έναν κόμβο από τον άλλο. Δεν υπάρχουν σύνδεσμοι μεταξύ των κόμβων ενός συνόλου.

Το ένα σύνολο των κόμβων θα είναι οι επαγγελματίες και το άλλο θα είναι οι εργασίες. Οι σύνδεσμοι θα έχουν βάρη ίσα με το κόστος που απαιτεί ο κάθε επαγγελματίας για να διεκπεραιώσει κάθε εργασία. Τότε, για να βρούμε τη βέλτιστη λύση θα πρέπει να επιλέξουμε τους συνδέσμους εκείνους με τους οποίους μπορούμε να αντιστοιχίσουμε κάθε κόμβο από το σύνολο U με έναν και μόνο έναν κόμβο από το σύνολο V , και το άθροισμα των βαρών των επιλεγμένων συνδέσμων να είναι το ελάχιστο δυνατό. Η εικόνα 1 δείχνει έναν διμερή γράφο που αντιστοιχεί στον παραπάνω πίνακα κόστους (έχουμε διαιρέσει τις τιμές με το 1000). Οι γραμμές και οι στήλες του πίνακα κόστους είναι οι κόμβοι του γράφου, και οι τιμές των κελιών του πίνακα κόστους είναι τα βάρη των συνδέσμων του γράφου.

Σε έναν διμερή γράφο, ένα **ταίριασμα** (matching) είναι ένα σύνολο συνδέσμων του γράφου που δεν έχουν κοινούς κόμβους. Ένα ταίριασμα λοιπόν είναι μια αντιστοιχίση κόμβων από το σύνολο U σε κόμβους από το σύνολο V ώστε κάθε κόμβος να αντιστοιχεί σε έναν και μόνο έναν κόμβο. Ένα **τέλειο ταίριασμα** (perfect matching) είναι ένα ταίριασμα που περιλαμβάνει όλους τους κόμβους του γράφου, δηλαδή κάθε

κόμβος του συνόλου U αντιστοιχεί σε έναν και μόνο έναν κόμβο του συνόλου V .

Αν μοντελοποιήσουμε το πρόβλημά μας ως διμερή γράφο, το πρόβλημα της εύρεσης του οικονομικότερου συνδυασμού επαγγελματιών και εργασιών ανάγεται στο πρόβλημα της εύρεσης του τέλειου ταιριάσματος με το ελάχιστο δυνατό κόστος· δηλαδή, σε ένα ταίριασμα όπου κάθε κόμβος του συνόλου U αντιστοιχεί σε έναν και μόνο έναν κόμβο του συνόλου V και το άθροισμα των βαρών των συνδέσμων του ταιριάσματος είναι το μικρότερο δυνατό. Έτσι το πρόβλημα μπορούμε να το αποκαλέσουμε και *Πρόβλημα του Ελάχιστου Τέλειου Ταιριάσματος* ή *Ταίριασμα Ελαχίστου Κόστους σε Διμερή Γράφο*.

Για να βρούμε ένα ελάχιστο τέλειο ταίριασμα, βολεύει να εισάγουμε μια ακόμα έννοια. Μία *επισημείωση* (labeling) ενός γράφου με βάρη είναι μια αντιστοίχιση μιας τιμής (ετικέτα, label) σε κάθε κόμβο του γράφου, έτσι ώστε για κάθε σύνδεσμο (u, v) του γράφου, το άθροισμα της τιμής του u και της τιμής του v να είναι μικρότερο ή ίσο του βάρους του συνδέσμου:

$$l(u) + l(v) \leq \text{weight}(u, v)$$

Οι σύνδεσμοι εκείνοι για τους οποίους ισχύει η ισότητα, $l(u) + l(v) = \text{weight}(u, v)$, δηλαδή το κόστος του συνδέσμου είναι ίσο με το άθροισμα του κόστους των κόμβων του, ονομάζονται *σφιχτοί* (tight). αν ένας σύνδεσμος δεν είναι σφιχτός, ονομάζεται *χαλαρός* (slack).

Αν έχουμε μια επισημείωση και ένα τέλειο ταίριασμα M , τότε από τον ορισμό της επισημείωσης προκύπτει:

$$\sum_{(u,v) \in M} l(u) + l(v) \leq \sum_{(u,v) \in M} \text{weight}(u, v) = \text{weight}(M)$$

Ταυτόχρονα, αφού το M είναι ένα τέλειο ταίριασμα, περιλαμβάνει όλους τους κόμβους του γράφου ακριβώς μία φορά, οπότε:

$$\sum_{(u,v) \in M} l(u) + l(v) = \sum_{x \in U \cup V} l(x)$$

Επομένως:

$$\sum_{x \in U \cup V} l(x) \leq \text{weight}(M)$$

Η παραπάνω ανισότητα σημαίνει ότι οποιοδήποτε τέλειο ταίριασμα έχει συνολικό κόστος μεγαλύτερο ή ίσο από το άθροισμα των τιμών των κόμβων οποιασδήποτε επισημείωσης.

Αν ένα ταίριασμα χρησιμοποιεί μόνο σφιχτούς συνδέσμους της επισημείωσης $l(\cdot)$, θα το ονομάζουμε σφιχτό ταίριασμα. Αν πάρουμε ένα τέλειο σφιχτό ταίριασμα M_l , θα ισχύει:

$$\text{weight}(M_l) = \sum_{(u,v) \in M_l} \text{weight}(u,v) = \sum_{(u,v) \in M_l} l(u) + l(v) = \sum_{x \in U \cup V} l(x)$$

Τότε από τις δύο τελευταίες σχέσεις προκύπτει ότι:

$$\text{weight}(M_l) \leq \text{weight}(M)$$

Δηλαδή, για κάθε επισημειώση, το κόστος ενός τέλειου ταιριάσματος είναι μεγαλύτερο ή ίσο του κόστους ενός σφιχτού τέλειου ταιριάσματος. Αν M_{min} είναι ένα τέλειο ταίριασμα με το ελάχιστο δυνατό κόστος, θα ισχύει πάλι:

$$\text{weight}(M_l) \leq \text{weight}(M_{min})$$

Αφού όμως το M_{min} έχει το ελάχιστο δυνατό κόστος, αναγκαστικά ισχύει και:

$$\text{weight}(M_{min}) \leq \text{weight}(M_l)$$

Συνεπώς:

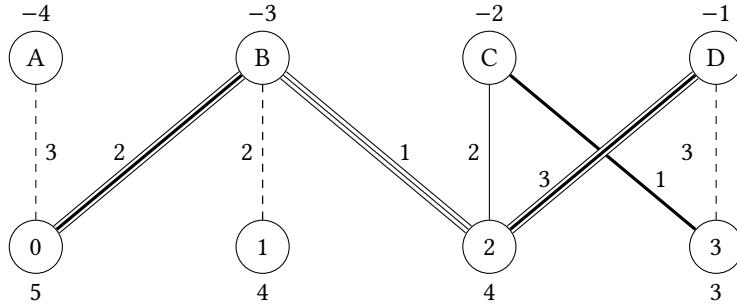
$$\text{weight}(M_{min}) = \text{weight}(M_l)$$

Επομένως το ελάχιστο δυνατό κόστος θα είναι ίσο με το κόστος του τέλειου σφιχτού ταιριάσματος.

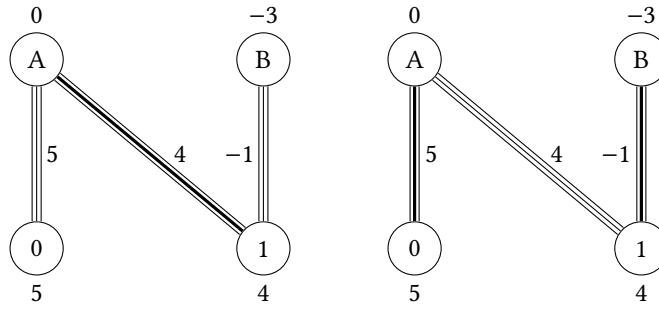
Από τα παραπάνω συνάγοντες ότι για να βρούμε ένα πλήρες ταίριασμα με το ελάχιστο δυνατό κόστος αρκεί να βρούμε μια επισημείωση και ένα πλήρες σφιχτό ταίριασμα για την επισημείωση αυτή: αυτό το ταίριασμα θα έχει και το ελάχιστο δυνατό κόστος.

Προϋπόθεση για τα παραπάνω είναι ότι το κόστος κάθε συνδέσμου δεν μπορεί να είναι αρνητικό. Αν τυχόν μπορούμε να έχουμε αρνητικά κόστη, τότε βρίσκουμε το πιο αρνητικό κόστος, έστω $-N$, και προσθέτουμε N σε όλους τους συνδέσμους. Με τον τρόπο αυτό όλοι οι σύνδεσμοι θα γίνουν θετικοί, ενώ προστίθεται συνολικά κόστος $|E| \times N$ σε κάθε πλήρες ταίριασμα του γράφου, επομένως δεν θα επηρεαστεί η λύση του προβλήματος.

Έστω τώρα ότι έχουμε ήδη βρει ένα σφιχτό ταίριασμα, όχι πλήρες, πάνω σε μία επισημείωση. Ένα μονοπάτι στο γράφο τέτοιο ώστε οι σύνδεσμοι από τους οποίους περνά να είναι όλοι σφιχτοί και εναλλάξ ανήκουν στο ταίριασμα που έχουμε βρει, ονομάζεται εναλλασσόμενο μονοπάτι (alternating path). Για παράδειγμα, στην εικόνα 2 με το ταίριασμα 0 – B, 2 – D, 3 – C, το μονοπάτι 0 – B – 2 – D είναι εναλλασσόμενο.



Εικόνα 2: Εναλλασσόμενο μονοπάτι.



Εικόνα 3: Αυξήσιμο μονοπάτι αριστερά, αυξημένο ταίριασμα δεξιά.

Οι διακεκομένες γραμμές αντιστοιχούν σε χαλαρούς συνδέσμους, οι απλές γραμμές σε σφιχτούς συνδέσμους, οι έντονες γραμμές αντιστοιχούν στους συνδέσμους του ταιριάσματος, και οι διπλές γραμμές αντιστοιχούν στους συνδέσμους που ανήκουν στο εναλλασσόμενο μονοπάτι.

Ένα εναλλασσόμενο μονοπάτι το οποίο ξεκινάει και τελειώνει σε κόμβους οι οποίοι δεν έχουν καλυφθεί από το τρέχον ταίριασμα, ονομάζεται *αυξήσιμο μονοπάτι* (augmenting path), επειδή μπορούμε να παράξουμε ένα μεγαλύτερο ταίριασμα το οποίο θα περιέχει τους συνδέσμους του μονοπατιού που δεν ανήκουν στο τρέχον ταίριασμα, ενώ δεν θα περιέχει τους συνδέσμους του μονοπατιού που ανήκουν στο τρέχον ταίριασμα. Με άλλα λόγια, ανατρέπουμε το ταίριασμα των συνδέσμων του μονοπατιού. Στην εικόνα 3 στα αριστερά έχουμε ένα αυξήσιμο μονοπάτι και το ταίριασμα $A - 1$. Στα δεξιά έχουμε το ταίριασμα $0 - A, 1 - B$ που προκύπτει μετά την ανατροπή των συνδέσμων (οπότε το μονοπάτι από αυξήσιμο γίνεται εναλλασσόμενο).

Αυτό λοιπόν μας δίνει ένα τρόπο να προχωράμε στην κατασκευή του ταιριάσματος που αναζητούμε.

- Ξεκινάμε με ένα κενό ταίριασμα, όπου κανείς κόμβος δεν αντιστοιχίζεται σε άλλον κόμβο, και μία επισημείωση όπου όλες οι τιμές των κόμβων είναι ίσες

με μηδέν. Στην κατάσταση αυτή, κανένας σύνδεσμος δεν είναι σφιχτός.

- Όσο δεν έχουμε βρει ένα πλήρες ταίριασμα:

- Αναζητούμε ένα αυξήσιμο μονοπάτι.
- Αν βρούμε ένα αυξήσιμο μονοπάτι, τότε το ανατρέπουμε και μεγαλώνουμε το ταίριασμά μας.
- Αν δεν βρούμε ένα αυξήσιμο μονοπάτι, τότε βελτιώνουμε την επισημείωση έτσι ώστε στην επόμενη επανάληψη να μπορέσουμε να βρούμε ένα αυξήσιμο μονοπάτι.

Με τον τρόπο αυτό, το ταίριασμα που κατασκευάζουμε είναι πάντοτε σφιχτό. Μόλις λοιπόν το ταίριασμα καλύψει όλους τους κόμβους του γράφου, θα είναι πλήρες και ταυτόχρονα θα έχει το ελάχιστο δυνατό κόστος.

Για να βάλουμε όμως τα παραπάνω βήματα στην πράξη, πρέπει να λύσουμε δύο θέματα. Πρώτον, πρέπει να βρούμε με ποιον τρόπο θα αναζητούμε κάθε φορά ένα αυξήσιμο μονοπάτι. Δεύτερον, πρέπει να βρούμε με ποιον τρόπο θα βελτιώνουμε την επισημείωση.

Για το πρώτο θέμα, αρκεί μια παραλλαγή της κατά πλάτος αναζήτησης (breadth-first search). Στην παραλλαγή αυτή, ξεκινάμε την αναζήτηση από έναν κόμβο που δεν καλύπτεται από το ταίριασμα, και προχωράμε χρησιμοποιώντας μόνο τους σφιχτούς συνδέσμους του γράφου. Θυμηθείτε ότι στην κατά πλάτος αναζήτηση επισκεπτόμαστε τους γείτονες του κόμβου, μετά τους γείτονες των γειτόνων, τους γείτοντες των γειτόνων των γειτόνων, κ.ο.κ., εξερευνούμε δηλαδή τον γράφο σε επίπεδα απομακρυνόμενοι από την εκκίνηση. Επειδή ένα εναλλασσόμενο μονοπάτι πρέπει να αποτελείται εναλλάξ από συνδέσμους που ανήκουν ή δεν ανήκουν στο τρέχον ταίριασμα, κατά την εκτέλεση της κατά πλάτος αναζήτησης θα σημειώνουμε αν ένας κόμβος βρίσκεται σε άρτιο ή περιττό επίπεδο, ώστε να γνωρίζουμε αν οι σφιχτοί σύνδεσμοι του κόμβου αυτού θα πρέπει να ανήκουν ή να μην ανήκουν στο τρέχον ταίριασμα.

Μπορείτε να δείτε τον αλγόριθμο AugmentingPathBFS της κατά πλάτος αναζήτησης για την εύρεση αυξήσιμου μονοπατιού στην εικόνα 4· καλό είναι να την αντιπαραθέσετε με τον αλγόριθμο για την κατά πλάτος αναζήτηση που έχουμε δει στο μάθημα.

Στις γραμμές 1–3 του αλγορίθμου δημιουργούμε την ουρά Q που θα χρησιμοποιήσουμε κατά την εκτέλεση του αναζήτησης, τον πίνακα $visited$ που μας δείχνει αν έχουμε επισκεφτεί έναν κόμβο, και τον πίνακα $inqueue$ που μας δείχνει αν ένας κόμβος βρίσκεται στην ουρά.

Στη συνέχεια στις γραμμές 4–6 δημιουργούμε τις δομές που θα χρειαστούμε επιπλέον των δομών που χρειάζονται στην απλή κατά πλάτος αναζήτηση. Θα χρειαστούμε έναν πίνακα $pred$ που δείχνει για κάθε κόμβο τον κόμβο από τον οποίο φτάσαμε σε αυτόν, το σύνολο odd_nodes που περιέχει τους κόμβους που βρίσκονται περιττό αριθμό συνδέσμων μακριά από τον κόμβο εκκίνησης και το σύνολο $even_nodes$ που περιέχει τους κόμβους που βρίσκονται άρτιο αριθμό συνδέσμων από τον κόμβο εκκίνησης.

Algorithm 1: Augmenting Path Breadth-First Search

AugmentingPathBFS(G , $node$, $matching$, $costs$, $prices$) \rightarrow
 $augmenting_path$, odd_nodes , $even_nodes$

Input: $G = (U \cup V, E)$, a bipartite graph
 $node$, the starting vertex in G
 $matching$, the current matching
 $costs$, the costs associated with each edge
 $prices$, the prices associated with each vertex

Output: $augmenting_path$, an augmenting path or NULL
 odd_nodes , the odd visited nodes
 $even_nodes$, the even visited nodes

```
1   $Q \leftarrow CreateQueue()$ 
2   $visited \leftarrow CreateArray(|U| + |V|)$ 
3   $inqueue \leftarrow CreateArray(|U| + |V|)$ 
4   $pred \leftarrow CreateArray(|U| + |V|)$ 
5   $odd\_nodes \leftarrow CreateSet()$ 
6   $even\_nodes \leftarrow CreateSet()$ 
7  foreach  $u \in U \cup V$  do
8       $visited[u] \leftarrow \text{FALSE}$ 
9       $inqueue[u] \leftarrow \text{FALSE}$ 
10      $pred[u] \leftarrow -1$ 
11  Enqueue( $Q$ ,  $node$ )
12   $inqueue[node] \leftarrow \text{TRUE}$ 
13  Enqueue( $Q$ ,  $-1$ )
14   $level \leftarrow 0$ 
15  while  $\text{Size}(Q) > 1$  do
16       $c \leftarrow Dequeue(Q)$ 
17      if  $c = -1$  then
18           $level \leftarrow level + 1$ 
19           $c \leftarrow Dequeue(Q)$ 
20          Enqueue( $Q$ ,  $-1$ )
21           $inqueue[c] \leftarrow \text{FALSE}$ 
22           $visited[c] \leftarrow \text{TRUE}$ 
23          if  $level \bmod 2 = 0$  then
24               $next\_level\_odd \leftarrow \text{TRUE}$ 
25              AddToSet( $even\_nodes$ ,  $c$ )
26          else
27               $next\_level\_odd \leftarrow \text{FALSE}$ 
28              AddToSet( $odd\_nodes$ ,  $c$ )
29  foreach  $v$  in  $\text{AdjacencyList}(G, c)$  do
30      if not  $visited[v]$  and  $costs[c, v] = prices[c] + prices[v]$  then
31          if  $next\_level\_odd$  and  $matching[v] = \text{NULL}$  then
32               $pred[v] \leftarrow c$ 
33               $augmenting\_path \leftarrow GetPath(pred, v)$ 
34              return  $augmenting\_path$ ,  $odd\_nodes$ ,  $even\_nodes$ 
35          if (( $next\_level\_odd$  and  $matching[c] \neq v$ ) or
36              ( $not next\_level\_odd$  and  $matching[c] = v$ )) and
37              not  $inqueue[v]$  then
38              Enqueue( $Q$ ,  $v$ )
39               $inqueue[v] \leftarrow \text{TRUE}$ 
40               $pred[v] \leftarrow c$ 
41  return NULL,  $odd\_nodes$ ,  $even\_nodes$ 
```

Οι πίνακες *visited*, *inqueue*, και *pred* αρχικοποιούνται στις γραμμές 7–10, και προχωράμε στην εισαγωγή του κόμβου εκκίνησης και του -1 στην ουρά, στις γραμμές 11–13. Η τιμή -1 στην ουρά θα παίξει το ρόλο του φρουρού (sentinel): όταν τη συναντάμε, θα ξέρουμε ότι έχουμε επεξεργαστεί όλους τους κόμβους που βρίσκονται σε ένα συγκεκριμένο επίπεδο από τον κόμβο εκκίνησης. Αρχικά το επίπεδο είναι ίσο με 0 , όπως ορίζουμε στη γραμμή 14.

Η κατά πλάτος αναζήτηση διεξάγεται στην επανάληψη των γραμμών 15–40. Προχωράει όπως η κανονική κατά πλάτος αναζήτηση, βγάζοντας ένα στοιχείο από την ουρά (γραμμή 16), με κάποιες διαφορές. Κάθε φορά που βρίσκουμε τον φρουρό στην ουρά, αυξάνουμε το επίπεδο που βρισκόμαστε και ξαναβάζουμε τον φρουρό στο τέλος (γραμμές 17–20). Σημειώνουμε τον τρέχοντα κόμβο (γραμμές 21–22) και αναλόγως αν το επίπεδό μας είναι περιττός ή άρτιος αριθμός προσθέτουμε τον κόμβο είτε στο σύνολο *odd_nodes* ή *even_nodes* (γραμμές 23–28).

Επεξεργαζόμαστε τους γείτονες του τρέχοντα κόμβου στην επανάληψη των γραμμών 29–40. Μας ενδιαφέρουν μόνο οι γείτονες που δεν έχουμε επισκεφτεί και στους οποίους φτάνουμε δια μέσω ενός σφιχτού σύνδεσμου (γραμμή 30).

Αφού έχουμε ξεκινήσει από έναν κόμβο που δεν τον έχουμε ταιριάξει, αν ο γείτονας βρίσκεται σε περιττό επίπεδο και ούτε αυτόν τον έχουμε ταιριάξει, βρήκαμε ένα αυξήσιμο μονοπάτι (γραμμή 31). Το κατασκευάζουμε από τον πίνακα *pred* χρησιμοποιώντας τη βοηθητική συνάρτηση *GetPath* και επιστρέφουμε το μονοπάτι, τους κόμβους σε περιττή απόσταση και τους κόμβους σε άρτια απόσταση (γραμμές 32–24).

Διαφορετικά, προκειμένου να συνεχίσουμε την εξερεύνηση ώστε να έχουμε εναλλάξ συνδέσμους που ανήκουν ή δεν ανήκουν στο ταίριασμα, εξετάζουμε γείτονες κόμβους που δεν είναι ήδη στην ουρά και (γραμμές 35–37). Αφού ξεκινήσαμε από άρτιο επίπεδο χωρίς ταίριασμα, θέλουμε όταν βρισκόμαστε σε άρτιο επίπεδο να βρίσκουμε συνδέσμους που δεν ανήκουν στο τρέχον ταίριασμα. Αντίθετα, αν βρισκόμαστε σε περιττό επίπεδο θέλουμε να βρίσκουμε συνδέσμους που ανήκουν στο τρέχον ταίριασμα. Έτσι:

- Αν είμαστε σε άρτιο επίπεδο, ο γείτονας v βρίσκεται σε περιττό επίπεδο. Αν δεν τον έχουμε ταιριάξει με τον τρέχοντα κόμβο c , τον έχουμε ταιριάξει με άλλον κόμβο, επομένως ο σύνδεσμος (c, v) δεν ανήκει στο τρέχον ταίριασμα.
- Αν είμαστε σε περιττό επίπεδο, ο γείτονας v βρίσκεται σε άρτιο επίπεδο. Αν τον έχουμε ταιριάξει με τον τρέχοντα κόμβο c , ο σύνδεσμος (c, v) ανήκει στο τρέχον ταίριασμα.

Εάν συμβαίνει το ένα από τα δύο, βάζουμε τον γείτονα στην ουρά και ενημερώνουμε ότι θα τον επισκεφτούμε ερχόμενοι από τον τρέχοντα κόμβο (γραμμές 38–40). Αν μετά την ολοκλήρωση της αναζήτησης δεν έχουμε βρει αυξήσιμο μονοπάτι, επιστρέφουμε τους κόμβους που επισκεφτήκαμε σε περιττά και άρτια επίπεδα (γραμμή 41).

Έχοντας στη διάθεσή μας την παραλλαγή της κατά πλάτος αναζήτησης, μπορούμε να προχωρήσουμε στη σύνθεση του επονωμαζόμενου Ουγγαρέζικου Αλγόριθμου, ο οποίος λύνει το Πρόβλημα του Ελάχιστου Ταίριασματος, ή εναλλακτικά,

Algorithm 2: Hungarian Algorithm

```
Hungarian( $G, costs$ ) $\rightarrow r$ 
    Input:  $G = (U \cup V, E)$ , a bipartite graph
           $costs$ , a lookup table mapping edges to costs
    Output:  $matching$ , a minimum cost perfect matching

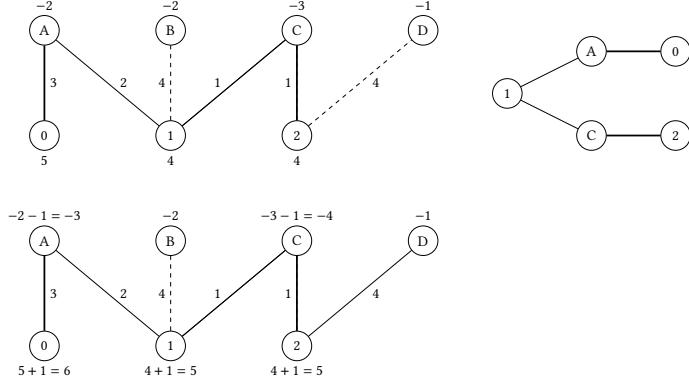
1    $matching \leftarrow \text{CreateArray}(|U| + |V|)$ 
2    $prices \leftarrow \text{CreateArray}(|U| + |V|)$ 
3   foreach  $u$  in  $U \cup V$  do
4        $prices[u] \leftarrow 0$ 
5   while not IsPerfectMatching( $G, matching$ ) do
6        $node \leftarrow \text{FindFirstUnmatched}(U, matching)$ 
7        $augmenting\_path, odd\_nodes, even\_nodes \leftarrow$ 
          AugmentingPathBFS( $G, node, matching, costs, prices$ )
8       if  $augmenting\_path \neq \text{NULL}$  then
9            $new\_matching \leftarrow \text{CopyArray}(matching)$ 
10          foreach  $(u \in U, v \in V)$  in  $augmenting\_path$  do
11               $new\_matching[u] \leftarrow v$ 
12               $new\_matching[v] \leftarrow u$ 
13           $matching \leftarrow new\_matching$ 
14      else
15           $\delta \leftarrow \text{ComputeDelta}(G, odd\_nodes, even\_nodes, costs, prices)$ 
16          foreach  $u$  in  $even\_nodes$  do
17               $prices[u] \leftarrow prices[u] + \delta$ 
18          foreach  $u$  in  $odd\_nodes$  do
19               $prices[u] \leftarrow prices[u] - \delta$ 
20      return  $matching$ 
```

Εικόνα 5: Ουγγαρέζικος Αλγόριθμος.

το Πρόβλημα της Ανάθεσης Γραμμικού Αθροίσματος, ή το Πρόβλημα Ταιριάσματος Ελαχίστου Κόστους σε Διμερή Γράφο, σε χρόνο $O(n^3)$. Σε μορφή ψευδοκώδικα ο αλγόριθμος είναι στην εικόνα 5.

Ο αλγόριθμος δημιουργεί έναν πίνακα $matching$ ο οποίος θα κρατάει το ταίριασμα που κατασκευάζουμε (γραμμή 1), και έναν πίνακα $prices$ ο οποίος θα έχει την τιμή που θα παίρνει κάθε κόμβος, δηλαδή την επισημείωση. (γραμμή 2). Οι αρχικές τιμές είναι όλες μηδέν (γραμμές 3-4).

Το ταίριασμα κατασκευάζεται σε επαναλήψεις (γραμμές 5-19). Σε κάθε επανάληψη, είτε βρίσκουμε ένα αυξήσιμο μονοπάτι οπότε μπορούμε να αυξήσουμε το ταίριασμα που κατασκευάζουμε, είτε, αν αυτό δεν είναι δυνατόν, αλλάζουμε τις τιμές των κόμβων ώστε να δημιουργήσουμε περισσότερους σφιχτούς συνδέσμους και να μπορέσουμε να βρούμε ένα αυξήσιμο μονοπάτι. Βρίσκουμε τον πρώτο (αλφαριθμητικά ή αριθμητικά) κόμβο ανάμεσα στους U ο οποίος δεν έχει ταίρι (γραμμή 6). Αφού τον



Εικόνα 6: Προσαρμογή τιμών.

βρούμε, καλούμε τον αλγόριθμο `AugmentingPathBFS` προκειμένου να βρούμε ένα αυξήσιμο μονοπάτι που να ξεκινάει από αυτόν (γραμμή 7).

Αν βρούμε ένα τέτοιο μονοπάτι, τότε το χρησιμοποιούμε για να αυξήσουμε το ταίριασμά μας, βάζοντας στο ταίριασμα τους συνδέσμους του μονοπατιού που δεν ανήκουν στο ταίριασμα, και βγάζοντας από το ταίριασμα τους συνδέσμους του μονοπατιού που ανήκουν (μέχρι τώρα) στο ταίριασμα (γραμμές 8–13). Αν δεν βρούμε ένα μονοπάτι, τότε θέλουμε να βελτιώσουμε τις τιμές που έχουμε δώσει στους κόμβους ώστε να προκύψουν και άλλοι σφιχτοί σύνδεσμοι που θα μπορούν να ενταχθούν σε αυξήσιμο μονοπάτι (γραμμές 14–19).

Για να το κάνουμε αυτό, βρίσκουμε την ελάχιστη διαφορά μεταξύ του βάρους του συνδέσμου και του αθροίσματος των τιμών των κόμβων του για κάθε σύνδεσμο (u, v) έτσι ώστε ο u να ανήκει στο σύνολο `even_nodes` και ο v να μην ανήκει στο σύνολο `odd_nodes`:

$$\delta = \min(\{ \text{weight}(u, v) - l(u) - l(v), \forall u \in \text{even_nodes}, v \notin \text{odd_nodes} \})$$

Με άλλα λόγια, ψάχνουμε να βρούμε τους λιγότερο χαλαρούς συνδέσμους που ξεκινούν από άρτιους κόμβους του εναλλασσόμενου μονοπατιού αφού μπορούμε πιο εύκολα να τους κάνουμε σφιχτούς. Στον αλγόριθμο, αυτό υλοποιείται με την κλήση στη συνάρτηση `ComputeDelta` (γραμμή 15).

Αφού βρούμε τη διαφορά δ , την προσθέτουμε στις τιμές των άρτιων κόμβων και την αφαιρούμε από τις τιμές των περιττών κόμβων (γραμμές 16–19). Ας δούμε τι ακριβώς σημαίνει αυτό στην εικόνα 6.

Στο τμήμα του γράφου που φαίνεται στο πάνω αριστερό κομμάτι της εικόνας, έχουμε προχωρήσει στην κατά πλάτος αναζήτηση ξεκινώντας από τον κόμβο 1, αλλά δεν έχουμε καταφέρει να βρούμε ένα αυξήσιμο μονοπάτι. Οι κόμβοι που έχουμε επισκεφτεί πάντως σχηματίζουν ένα δένδρο, όπως φαίνεται στο πάνω δεξιά κομμάτι της

εικόνας. Αν κάνουμε τους υπολογισμούς προκύπτει ότι $\delta = 1$ είναι η διαφορά μεταξύ του βάρους και των τιμών στον σύνδεσμο (2, D). Αν προσθέσουμε το δ στα άρτια επίπεδα του δένδρου και το αφαιρέσουμε από τα περιττά επίπεδα του δένδρου, παίρνουμε το γράφο όπως φαίνεται στο κάτω αριστερά κομμάτι της εικόνας. Παρατηρούμε ότι οι σύνδεσμοι που ήταν ήδη σφιχτοί δεν επηρεάστηκαν (αφού προσθέσαμε και αφαιρέσαμε τον ίδιο αριθμό στα δύο άκρα τους). Όμως, ο σύνδεσμος (2, D) έγινε σφιχτός—και επίσης ο σύνδεσμος (1, B) έγινε λιγότερο χαλαρός από ότι ήταν πριν.

Ο Ουγγαρέζικος Αλγόριθμος με τη μορφή που τον παρουσιάσαμε βρίσκει τέλειο ταίριασμα με το ελάχιστο κόστος. Αν θέλουμε να μεγιστοποιήσουμε το κόστος, αρκεί να πολλαπλασιάσουμε με -1 τα βάρη των συνδέσμων του γράφου πριν ξεκινήσουμε. Προσέξτε ότι εφόσον αυτό δημιουργεί αρνητικά βάρη, όπως παρατηρήσαμε στην αρχή, αν $-N$ είναι το πιο αρνητικό κόστος μεταξύ όλων των συνδέσμων, πρέπει να προσθέσουμε N σε όλους τους συνδέσμους ώστε ο αλγόριθμος να λειτουργήσει στη συνέχεια σωστά.

Στην εργασία αυτή θα πρέπει κατ' αρχήν να υλοποιήσετε τον Ουγγαρέζικο Αλγόριθμο για την επίλυση προβλημάτων εύρεσης του βέλτιστου τέλειου ταιριάσματος, αν σας δίνει κάποιος τον πίνακα κόστους. Κατόπιν, θα χρησιμοποιήσετε την υλοποίησή σας για μια εφαρμογή που ίσως έχει κάποιο αισθητικό ενδιαφέρον.

Κατασκευή Πορτραίτων

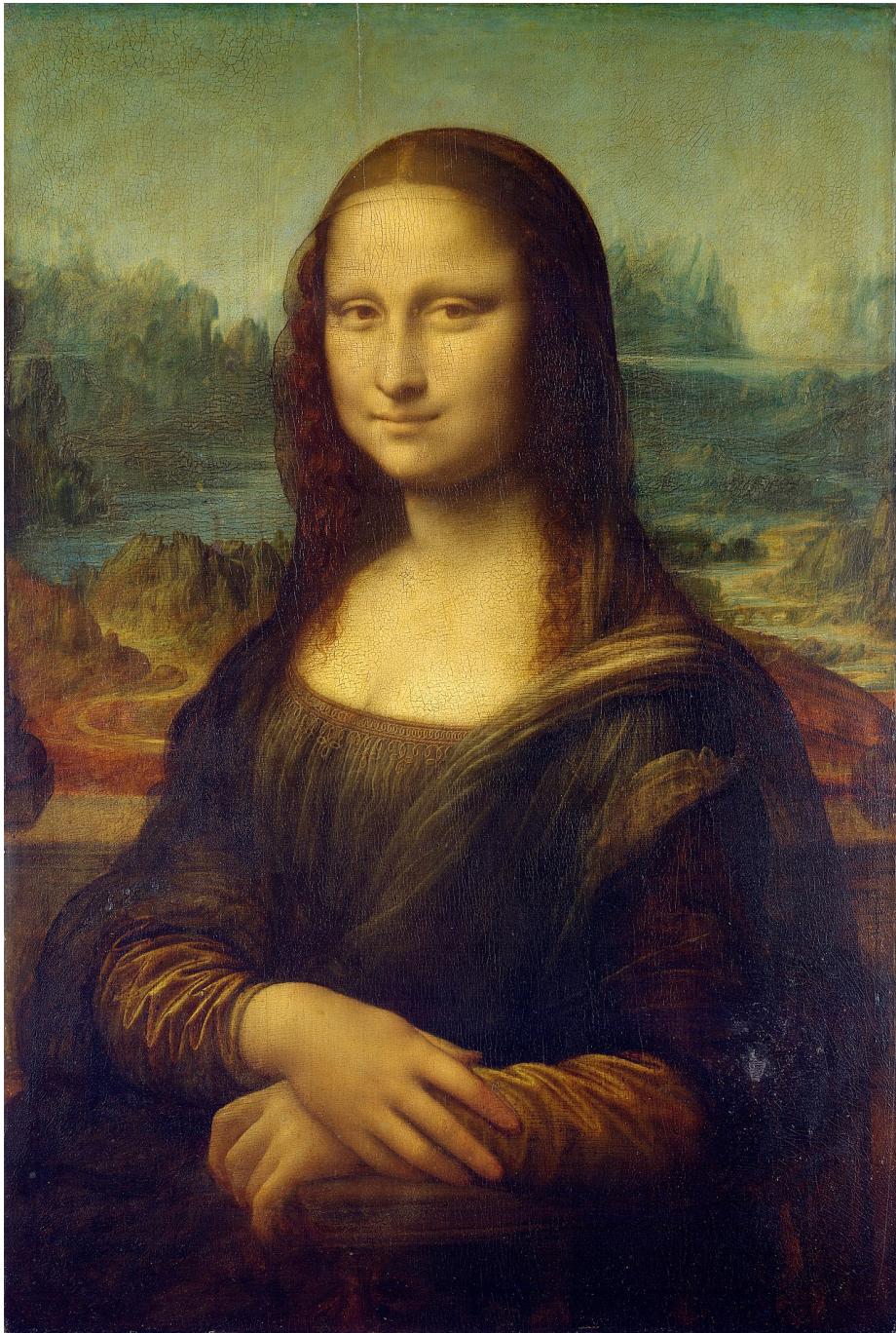
Τι σχέση έχει το ταίριασμα στους διμερείς γράφους με την κατασκευή πορτραίτων από ντόμινο, όπως αυτό στην αρχή της εκφώνησης; Ξεκινάμε από ένα πορτραίτο, όπως για παράδειγμα η Μόνα Λίζα, στην εικόνα 7.

Κάνουμε την εικόνα μονοχρωματική και κόβουμε τα άκρα της ώστε να μπορεί να διαιρεθεί σε έναν αριθμό ψηφίδων που ορίζουμε, συγκεκριμένα σε 30 γραμμές και 22 στήλες, όπως φαίνεται στην εικόνα 8.

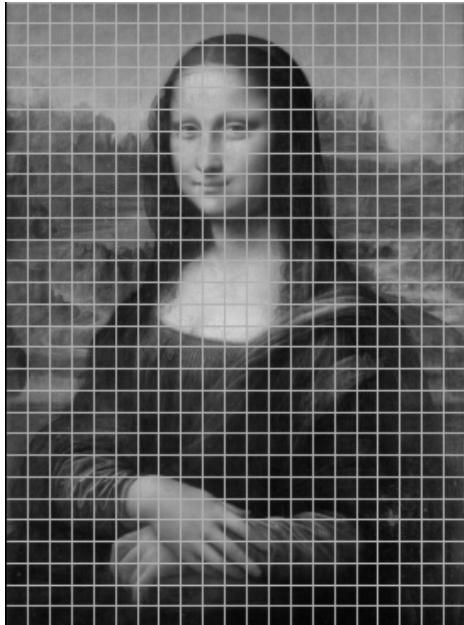
Στη συνέχεια, αντικαθιστούμε κάθε μία από τις ψηφίδες που ορίσαμε στον πίνακα με το πιο αντιπροσωπευτικό τόνο από 10 τόνους του γκρίζου, όπως φαίνεται στην εικόνα 9.

Δεδομένου ότι κάθε ντόμινο περιέχει δύο τετράγωνα, θα πρέπει οι ψηφίδες που φτιάχαμε να συνδυαστούν σε δυάδες ώστε να μπορούμε να βάλουμε ένα ντόμινο σε κάθε μία δυάδα. Αναλόγως με το πού βρίσκεται μια ψηφίδα, μπορεί να συνδυαστεί με άλλες δύο ψηφίδες (αν βρίσκεται στη γωνία), με άλλες τρεις (αν βρίσκεται στην άκρη της εικόνας αλλά όχι στη γωνία), ή με άλλες τέσσερεις (αν βρίσκεται στο εσωτερικό), όπως φαίνεται στην εικόνα 10.

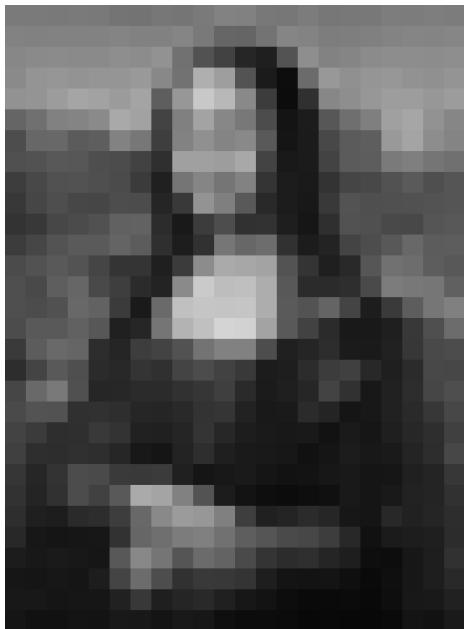
Θα μετράμε τις ψηφίδες από πάνω προς τα κάτω και από τα αριστερά προς τα δεξιά, ώστε (x, y) να είναι η ψηφίδα στην x γραμμή από την κορυφή και στην y στήλη από τα αριστερά. Τότε, για παράδειγμα, η ψηφίδα $(0, 0)$ είναι στην πάνω αριστερά μεριά της εικόνας και έχει γείτονες τις ψηφίδες $(0, 1)$ (δεξιά) και $(1, 0)$ (κάτω). Το πρόβλημα που έχουμε είναι η αντιστοίχιση κάθε ψηφίδας που το άθροισμα των συντεταγμένων της είναι άρτιος αριθμός με μία από τις γειτονικές της ψηφίδες, η κάθε μία από τις οποίες έχει συντεταγμένες που αθροίζουν σε περιττό αριθμό. Με άλλα λόγια, πρέπει



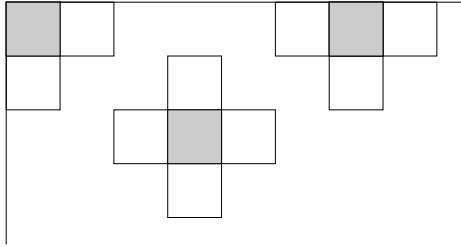
Εικόνα 7: Μόνα Λίζα (Πηγή: Wikipedia).



Εικόνα 8: Μόνα Λίζα μονοχρωματική 30×22 .



Εικόνα 9: Ψηφιδοποιημένη Μόνα Λίζα 30×22 .



Εικόνα 10: Δυνατά ντόμινο ανά θέση ψηφίδας.

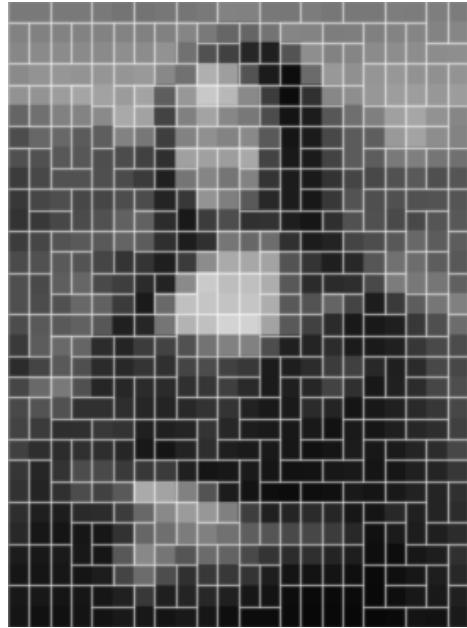
να αντιστοιχίσουμε τις μισές, άρτιες ψηφίδες της εικόνας $((30 \times 22)/2 = 330 \text{ ψηφίδες})$, με τις άλλες μισές, περιττές ψηφίδες. Μπορούμε λοιπόν να κατασκευάσουμε έναν διμερή γράφο $G = (U \cup V, E)$ όπου U είναι οι 330 άρτιες ψηφίδες, V είναι οι 330 περιττές ψηφίδες, και E είναι οι σύνδεσμοι (από δύο έως τέσσερεις) μεταξύ μιας άρτιας ψηφίδας και των γειτονικών της περιττών ψηφίδων. Επιπλέον, κάθε σύνδεσμος θα έχει ένα βάρος, το οποίο θα αντιπροσωπεύει πόσο επιθυμητό είναι να ταιριάζουν οι δύο ψηφίδες που συνδέει. Αν επανέλθουμε στην εικόνα με τις ψηφίδες, το βάρος κάθε συνδέσμου θα το συσχετίσουμε με τη διαφορά στη φωτεινότητα μεταξύ των δύο γειτονικών ψηφίδων. Συγκεκριμένα, για δύο ψηφίδες u και v το βάρος του συνδέσμου που θα τις συνδέει θα είναι ίσο με το τετράγωνο της διαφοράς της φωτεινότητας των δύο ψηφίδων. Δηλαδή, αν $b(u)$ και $b(v)$ είναι οι φωτεινότητες των γειτονικών ψηφίδων u και v αντίστοιχα, το βάρος $weight(u, v)$ θα είναι:

$$weight(u, v) = (b(u) - b(v))^2$$

Επομένως, για να βρούμε τον βέλτιστο τρόπο να διαμερίσουμε την εικόνα μας σε θέσεις ντόμινο, θα βρούμε το τέλειο ταίριασμα μεταξύ των άρτιων και των περιττών ψηφίδων έτσι ώστε το όθροισμα των βαρών των ακμών του διμερούς γράφου που προκύπτει να μεγιστοποιηθεί. Θέλουμε, δηλαδή, οι θέσεις ντόμινο που θα προκύψουν να καλύπτουν συνολικά όσο πιο ετερογενείς ψηφίδες γίνεται. Αν λύσουμε αυτό το πρόβλημα ταιριάσματος, θα πάρουμε τις θέσεις ντόμινο στις οποίες θα πρέπει να διαιρέσουμε την εικόνα μας· στην περίπτωση της Μόνα Λίζα, το αποτέλεσμα φαίνεται στην εικόνα 11.

Το επόμενο, και τελικό βήμα, είναι να τοποθετήσουμε κομμάτια ντόμινο πάνω στις θέσεις ντόμινο που βρήκαμε. Αυτό ανάγεται και πάλι σε ένα πρόβλημα ταιριάσματος. Αφού έχουμε συνολικά $22 \times 30 = 660 \text{ ψηφίδες}$, έχουμε $660/2 = 330 \text{ θέσεις ντόμινο}$, άρα θα πρέπει να χρησιμοποιήσουμε 330 κομμάτια ντόμινο για να τις καλύψουμε. Αυτός είναι και ο λόγος που επιλέξαμε αρχικά να χωρίσουμε την εικόμα σε 30 γραμμές και 22 στήλες. Τόσα είναι ακριβώς έξι πλήρη σύνολα κομματιών ντόμινο από το $(0, 0)$ μέχρι το $(9, 9)$. ένα τέτοιο σύνολο μπορείτε να δείτε στην εικόνα 12. Οι τιμές πάνω σε κάθε τετραγωνάκι το ντόμινο κυμαίνονται από το 0 έως και το 9, όσοι δηλαδή είναι οι τόνοι του γκρίζου που έχει η εικόνα με τις ψηφίδες.

Για να βρούμε ποιο ντόμινο πρέπει να τοποθετήσουμε σε ποια θέση, από αυτές που



Εικόνα 11: Οι θέσεις ντόμινο της Μόνα Λίζα.



Εικόνα 12: Ντόμινο 9×9 .



Εικόνα 13: Εναλλακτικές λύσεις τοποθέτησης ντόμινο.

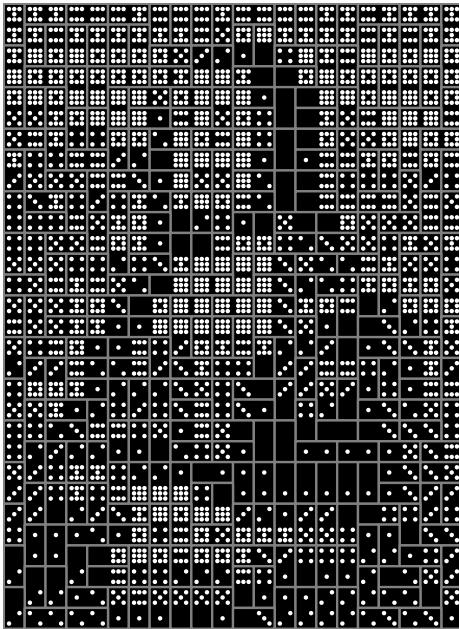
βρήκαμε στο προηγούμενο βήμα, κατασκευάζουμε ένα νέο διμερή γράφο $G = (U \cup V, E)$ όπου U είναι τα 330 ντόμινο και V είναι οι 330 θέσεις. Κάθε ντόμινο μπορεί να μπει σε οποιαδήποτε από τις 330 θέσεις, οπότε συνολικά έχουμε 330×330 συνδέσμους. Το βάρος κάθε συνδέσμου τώρα θα δείχνει πόσο απέχει η φωτεινότητα του ντόμινο από τη φωτεινότητα της θέσης του ντόμινο. Εδώ όμως υπάρχουν δύο ενδεχόμενα, αναλόγως με τον προσανατολισμό του ντόμινο. Για παράδειγμα, στις θέσεις ((6, 7), (6, 8)) της ψηφιδοποιημένης Μόνα Λίζα η φωτεινότητα είναι (2, 5). Εκεί μπορούμε να τοποθετήσουμε, ας πούμε, το ντόμινο (2, 9) είτε ως έχει, είτε ανάποδα, ως (9, 2), όπως μπορείτε να δείτε στην εικόνα 13. Η τοποθέτηση ως (2, 9) αντιστοιχεί καλύτερα στη φωτεινότητα της εικόνας. Ορίζουμε λοιπόν το βάρος του συνδέσμου μεταξύ μιας θέσης και κάθε ντόμινο ως το ελάχιστο των δύο αθροισμάτων (ένα για κάθε δυνατό προσανατολισμό του ντόμινο) της διαφοράς των τετραγώνων της αξίας του ντόμινο και της φωτεινότητας της θέσης. Στο παράδειγμά μας της εικόνας 13, έχουμε στα αριστερά $(2 - 2)^2 + (5 - 9)^2 = 16$ ενώ στα δεξιά είναι $(2 - 9)^2 + (5 - 2)^2 = 58$, συνεπώς θα επιλέξουμε τον προσανατολισμό στα αριστερά, με βάρος 16. Γενικότερα, αν (b_1, b_2) είναι οι φωτεινότητες της θέσης που εξετάζουμε και (d_1, d_2) είναι οι τιμές πάνω στο ντόμινο, ως κόστος του συνδέσμου μεταξύ της θέσης και του ντόμινο ορίζουμε:

$$c = \min((b_1 - d_1)^2 + (b_2 - d_2)^2), (b_1 - d_2)^2 + (b_2 - d_1)^2)$$

Συνοψίζοντας λοιπόν, στο δεύτερο στάδιο υπολογίζουμε το κόστος της τοποθέτησης του κάθε ντόμινο σε κάθε δυνατή θέση, για κάθε έναν από τους δύο δυνατούς προσανατολισμούς του. Κρατάμε τον προσανατολισμό με το μικρότερο κόστος, αφού έτσι το ντόμινο θα απέχει λιγότερο οπτικά από τη θέση την οποία εξετάζουμε. Το κόστος αυτό θα είναι το βάρος του συνδέσμου στο διμερή γράφο μεταξύ της θέσης και αυτού του ντόμινο.

Αφού κατασκευάσουμε τον δεύτερο γράφο και δώσουμε τα βάρη στους συνδέσμους, λύνουμε το πρόβλημα του ταιριάσματος μεταξύ ντόμινο και θέσεων, προσπαθώντας αυτή τη φορά να ελαχιστοποιήσουμε το συνολικό κόστος. Όταν το κάνουμε αυτό, παίρνουμε τη Μόνα Λίζα ζωγραφισμένη με ντόμινο, όπως φαίνεται στην εικόνα 14.

Στην εργασία θα πρέπει, αφού έχετε υλοποίησε τον Ουγγαρέζικο Αλγόριθμο, να τον εφαρμόσετε αλγόριθμο για την κατασκευή πορτραίτων από ντόμινο. Η εργασία σας δεν θα περιλαμβάνει τα στάδια της επεξεργασίας εικόνας, δηλαδή της μετατροπής πορτραίτου σε μονοχρωματικό και την αντιστοίχιση τόνων γκρίζου.



Εικόνα 14: Μόνα Λίζα μωσαϊκό ντόμινο.

Απαιτήσεις Προγράμματος

Κάθε φοιτητής θα εργαστεί σε αποθετήριο στο GitHub. Για να αξιολογηθεί μια εργασία θα πρέπει να πληροί τις παρακάτω προϋποθέσεις:

- Για την υποβολή της εργασίας θα χρησιμοποιηθεί το ιδιωτικό αποθετήριο του φοιτητή που δημιουργήθηκε για τις ανάγκες του μαθήματος και του έχει αποδοθεί. Το αποθετήριο αυτό έχει όνομα του τύπου `username-algo-assignments`, όπου `username` είναι το όνομα του φοιτητή στο GitHub. Για παράδειγμα, το σχετικό αποθετήριο του διδάσκοντα θα ονομαζόταν `louridas-algo-assignments` και θα ήταν προσβάσιμο στο <https://github.com/dmst-algorithms-course/louridas-algo-assignments>. Τυχόν άλλα αποθετήρια απλώς θα αγνοηθούν.
- Μέσα στο αποθετήριο αυτό θα πρέπει να δημιουργηθεί ένας κατάλογος `assignment-2020-4`.
- Μέσα στον παραπάνω κατάλογο το πρόγραμμα θα πρέπει να αποθηκευτεί με το όνομα `domino_portrait.py`.
- Δεν επιτρέπεται η χρήση έτοιμων βιβλιοθηκών γράφων ή τυχόν έτοιμων υλοποιήσεων των αλγορίθμων, ή τμημάτων αυτών, εκτός αν αναφέρεται ρητά ότι επιτρέπεται.
- Επιτρέπεται η χρήση δομών δεδομένων της Python όπως στοίβες, λεξικά, σύ-

νολα, κ.λπ.

- Επιτρέπεται η χρήση της βιβλιοθήκης `collections.deque` και της βιβλιοθήκης `collections.defaultdict`.
- Επιτρέπεται η χρήση της βιβλιοθήκης `argparse` ή της βιβλιοθήκης `sys` (συγκεκριμένα, της λίστας `sys.argv`) προκειμένου να διαβάσει το πρόγραμμα τις παραμέτρους εισόδου.
- Το πρόγραμμα θα πρέπει να είναι γραμμένο σε Python 3.

Το πρόγραμμα θα καλείται ως `εξής` (όπου `python` η κατάλληλη εντολή στο εκάστοτε σύστημα):

```
python domino_portrait.py input_file
                      [-m, --maximize]
                      [-a, --assignment]
                      [tiling_file]
                      [dominoes_file]
```

Οι παράμετροι εμφανίζονται σε διαφορετικές γραμμές προς χάρη της τυπογραφίας. Η σημασία των παραμέτρων είναι η εξής:

- `input_file`: το αρχείο που περιγράφει τον πίνακα κόστους ή τους τόνους του γκρίζου της εικόνας.
- `-m, --maximize`: αν δίνεται, τότε συνδυάζεται με την παράμετρο `-a, --assignment` που εξηγείται παρακάτω, προκειμένου να βρει το ταίριασμα με το μέγιστο κόστος.
- `-a, --assignment`: αν δίνεται, θα είναι το όνομα ενός αρχείου που θα περιέχει έναν πίνακα κόστους.
- `tiling_file`: αν δίνεται, θα είναι το όνομα του αρχείου που θα δημιουργήσει το πρόγραμμα και θα περιέχει την κατάτμηση της εικόνας `input_file` σε θέσεις ντόμινο.
- `dominoes_file`: αν δίνεται, θα είναι το όνομα του αρχείου που θα δημιουργήσει το πρόγραμμα και θα περιέχει την αντιστοίχιση των ντόμινο στις θέσεις του αρχείου `tiling_file`.

Για την οπτικοποίηση των αποτελεσμάτων, μπορείτε να χρησιμοποιήσετε το παρακάτω προγράμμα:

- `draw_dominoes.py` το οποίο μπορείτε να καλείτε ως:

```
python draw_dominoes.py input_file output_file
```

όπου `input_file` είναι το αρχείο των αποτελεσμάτων σας (το `dominoes_file`) και `output_file` είναι η τελική εικόνα με ντόμινο που θέλετε να δημιουργήσετε. Για να λειτουργήσει, θα πρέπει να εγκαταστήσετε τη βιβλιοθήκη `matplotlib`.

Παραδείγματα

Παράδειγμα 1

Αν καλέσετε το πρόγραμμα με:

```
python domino_portrait.py -a test_1.txt
```

τότε το πρόγραμμα θα διαβάσει το αρχείο `test_1.txt` το οποίο περιέχει έναν πίνακα κόστους, θα τρέξει τον Ουγγαρέζικο Αλγόριθμο, και θα τυπώσει στην οθόνη ακριβώς τα παρακάτω:

```
2 1 0  
6
```

δηλαδή, η βέλτιστη αντιστοίχιση είναι μεταξύ της γραμμής 0 και της στήλης 2, της γραμμής 1 και της στήλης 1, της γραμμής 2 και της στήλης 0, με συνολικό κόστος 6.

Παράδειγμα 2

Αν καλέσετε το πρόγραμμα με:

```
python domino_portrait.py -a test_1.txt -m
```

τότε το πρόγραμμα θα τρέξει τον Ουγγαρέζικο Αλγόριθμο για το ίδιο αρχείο όπως και του προηγούμενου παραδείγματος, αλλά αυτή τη φορά θα μεγιστοποιήσει το κόστος και θα τυπώσει στην οθόνη ακριβώς τα παρακάτω:

```
0 2 1  
12
```

δηλαδή, η βέλτιστη αντιστοίχιση είναι μεταξύ της γραμμής 0 και της στήλης 0, της γραμμής 1 και της στήλης 2, και της γραμμής 2 με τη στήλη 1, με συνολικό κόστος 12.

Παράδειγμα 3

Αν καλέσετε το πρόγραμμα με:

```
python domino_portrait.py -a test_2.txt
```

τότε το πρόγραμμα θα διαβάσει το αρχείο `test_2.txt` το οποίο περιέχει έναν πίνακα κόστους, θα τρέξει τον Ουγγαρέζικο Αλγόριθμο, και θα τυπώσει στην οθόνη ακριβώς τα παρακάτω:

```
1 4 0 3 2  
8
```

Παράδειγμα 4

```
python domino_portrait.py -a test_2.txt -m
```

τότε το πρόγραμμα θα τρέξει τον Ουγγαρέζικο Αλγόριθμο για το ίδιο αρχείο όπως και του προηγούμενου παραδείγματος, αλλά αυτή τη φορά θα μεγιστοποιήσει το κόστος και θα τυπώσει στην οθόνη ακριβώς τα παρακάτω:

3 1 0 2 4

35

Παράδειγμα 5

```
python domino_portrait.py -a test_3.txt
```

τότε το πρόγραμμα θα διαβάσει το αρχείο [test_3.txt](#) το οποίο περιέχει έναν πίνακα κόστους, θα τρέξει τον Ουγγαρέζικο Αλγόριθμο, και θα τυπώσει στην οθόνη ακριβώς τα παρακάτω:

6 1 9 2 8 3 0 5 7 4

6

Παράδειγμα 6

```
python domino_portrait.py -a test_3.txt -m
```

τότε το πρόγραμμα θα τρέξει τον Ουγγαρέζικο Αλγόριθμο για το ίδιο αρχείο όπως και του προηγούμενου παραδείγματος, αλλά αυτή τη φορά θα μεγιστοποιήσει το κόστος και θα τυπώσει στην οθόνη ακριβώς τα παρακάτω:

5 0 1 7 2 6 3 8 4 9

78

Παράδειγμα 7

```
python domino_portraits.py mona_lisa.txt mona_lisa_tiles.txt \
    mona_lisa_dominoes.txt
```

τότε το πρόγραμμα θα διαβάσει το αρχείο [mona_lisa.txt](#) το οποίο περιέχει την εικόνα της Μόνα Λίζα σε 10 διαβαθμίσεις του γκρίζου, στην οθόνη θα εμφανίσει:

1088

3244

που είναι το κόστος της επίλυσης του πρώτου και του δεύτερου ταιριάσματος, και θα δημιουργήσει το αρχείο [mona_lisa_tiles.txt](#) και το αρχείο [mona_lisa_dominoes.txt](#)

Το αρχείο [mona_lisa_tiles.txt](#) περιέχει την κατάτμηση του πορτραίτου της Μόνα Λίζα σε θέσεις ντόμινο. Επομένως περιέχει την επίλυση του πρώτου ταιριάσματος στη διαδικασία κατασκευής πορτραίτων. Κάθε γραμμή περιέχει μία αντιστοίχιση ψηφίδων, δηλαδή μία θέση στην οποία μπορεί να μπει ντόμιμο. Για παράδειγμα, η γραμμή:

(0, 0) (1, 0)

σημαίνει ότι μια θέση ντόμινο καλύπτει την ψηφίδα (0, 0) (πάνω αριστερά) και την ψηφίδα (1, 0), που είναι αμέσως κάτω της.

Η γραμμή:

(3, 4) (4, 4)



Εικόνα 15: Μπόρις Κάρλοφ, ως Τέρας του Φρανκενστάιν (πηγή: Wikipedia).

σημαίνει ότι μια θέση ντόμινο καλύπτει την ψηφίδα $(3, 4)$ (γραμμή 3, μετρώντας από το μηδέν, από πάνω, στήλη 4, μετρώντας από το μηδέν, από αριστερά) και την ψηφίδα $(4, 4)$.

Το αρχείο `mona_lisa_dominoes.txt` περιέχει την αντιστοίχιση ντόμινο στις θέσεις ντόμινο του αρχείου `mona_lisa_tiles.txt`. Επομένως περιέχει την επίλυση του δεύτερου (και τελικού) ταιριάσματος στη διαδικασία κατασκευής πορτραίτων. Κάθε γραμμή περιγράφει μια αντιστοίχιση. Για παράδειγμα, η γραμμή

$((0, 0), (1, 0)) : (7, 7)$

δείχνει ότι οι ψηφίδες $(0, 0)$ και $(1, 0)$ καλύπτονται από το ντόμινο $(7, 7)$.

Για να δείτε το αποτέλεσμα, μπορείτε να τρέξετε:

```
python draw_dominoes.py mona_lisa_dominoes.txt mona_lisa_dominoes.pdf
```

Παράδειγμα 8

Αν καλέσουμε το πρόγραμμα ως:

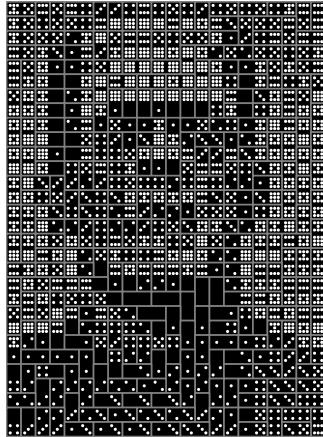
```
python domino_portraits.py frankensteins_monster.txt \
    frankensteins_monster_tiles.txt \
    frankensteins_monster_dominoes.txt
```

τότε το πρόγραμμα θα διαβάσει το αρχείο `frankensteins_monster.txt` το οποίο περιέχει την φωτογραφία του Μπόρις Κάρλοφ στο ρόλο του τέρατος του Φρανκενστάιν (βλ. εικόνα 15) σε 10 διαβαθμίσεις του γκρίζου, στην οθόνη θα εμφανίσει:

2970

1318

που είναι το κόστος της επίλυσης του πρώτου και του δεύτερου ταιριάσματος, και θα δημιουργήσει το αρχείο `frankensteins_monster_tiles.txt` και το αρχείο `frankensteins_monster_dominoes.txt`, που περιγράφει την εικόνα 16.



Εικόνα 16: Μπόρις Κάρλοφ, Τέρας του Φρανκενστάιν με ντόμινο.

Για να δείτε το αποτέλεσμα, μπορείτε να τρέξετε:

```
python draw_dominoes.py frankenstein_monster_dominoes.txt \
    frankenstein_monster_dominoes.pdf
```

Καλή Επιτυχία.

Για Περισσότερες Πληροφορίες.

Ο Ουγγαρέζικος αλγόριθμος παρουσιάστηκε από τον Harold Kuhn (1955, 1956). Ο Kuhn ήταν αμερικάνος· ονόμασε τον αλγόριθμο Ουγγαρέζικο γιατί βασίστηκε σε πρότερη δουλειά δύο ούγγρων μαθηματικών, του Dénes König και του Jenő Egerváry. Πολύ αργότερα, το 2006, ανακαλύφθηκε ότι το πρόβλημα της ανάθεσης το είχε λύσει τον 19ο αιώνα ο Carl Gustav Jacobi, ενώ η λύση του είχε δημοσιευθεί μετά θάνατον, στα λατινικά (1865), για αγγλική μετάφραση, δείτε το Ollivier (2009). Μια παραλλαγή του αλγορίθμου παρουσιάστηκε από τον (1957), οπότε ο αλγόριθμος είναι γνωστός και ως αλγόριθμος Kuhn-Munkres. Τόσο ο αρχικός αλγόριθμος όσο και ο αλγόριθμος Kuhn-Munkres έχουν πολυπλοκότητα $O(n^4)$. Το Πρόβλημα της Ανάθεσης Γραμμικού Κόστους είναι ένα είδος προβλήματος ανάθεσης. Το πεδίο είναι πολύ ευρύτερο, βλ. για παράδειγμα (Burkard, Dell'Amico, and Martello 2012). Η παρουσίαση του αλγορίθμου εδώ βασίζεται στις σημειώσεις του Tim Roughgarden (2016) και του Alex Grinman (2015).

Η κατασκευή πορτραίτων από ντόμινο αρχικά προτάθηκε από τον Knowlton (1983). Ο Donald Knuth (1993) προσέγγισε την κατασκευή ως εφαρμογή δύο προβλημάτων ταιριάσματος σε διμερή γράφο και έδειξε πώς μπορεί να εφαρμοστεί στη Μόνα Λίζα· κατά συνέπεια, η μέθοδος που παρουσιάσαμε συνήθως αναφέρεται ως μέθοδος Knuth-Knowlton. Περισσότερες λεπτομέρειες, και βελτιώσεις στη μέθοδο Knuth-Knowlton μπορείτε να δείτε στο βιβλίο του Robert Bosch (2019), ο οποίος χρησιμοποιεί ως παράδειγμα το πορτραίτο του Μπόρις Καρλόφ ως τέρας του Φρανκενστάιν.

- Bosch, Robert. 2019. *Opt Art: From Mathematical Optimization to Visual Design*. Princeton, NJ: Princeton University Press.
- Burkard, Rainer, Mauro Dell'Amico, and Silvano Martello. 2012. *Assignment Problems*. Revised reprint. Philadelphia, PA: Society for Industrial and Applied Mathematics.
- Grinman, Alex. 2015. “The Hungarian Algorithm for Weighted Bipartite Graphs.” https://www.cc.gatech.edu/~rpeng/18434_S15/hungarianAlgorithm.pdf.
- Jacob, Carl Gustav. 1865. “De investigando ordine systematis æquationum differentialium vulgarium cuiuscunque.” Edited by August Leopold Crelle. *Journal Für Die Reine Und Angewandte Mathematik* 64 (4): 297–320.
- Knowlton, Kenneth C. 1983. Representation of designs. 4,398,890. US patent, issued 1983.
- Knuth, Donald E. 1993. *The Stanford Graphbase: A Platform for Combinatorial Computing*. New York, NY: ACM Press.
- Kuhn, H. W. 1955. “The Hungarian Method for the assignment problem.” *Naval Research Logistics Quarterly* 2 (1–2): 83–97.
- . 1956. “Variants of the Hungarian Method for the assignment problem.” *Naval Research Logistics Quarterly* 3 (4): 253–58.
- Munkres, James. 1957. “Algorithms for the Assignment and Transportation Problems.” *Journal of the Society for Industrial and Applied Mathematics* 5 (1): 32–38.
- Ollivier, François. 2009. “Looking for the Order of a System of Arbitrary Ordinary Differential Equations.” *Applicable Algebra in Engineering, Communication and Computing* 20 (1): 7–32.
- Roughgarden, Tim. 2016. “CS261: A Second Course in Algorithms Lecture #5: Minimum-Cost Bipartite Matching.” <http://timroughgarden.org/w16/l/l5.pdf>.