Παρουσίαση Διπλωματικής Εργασίας

Υλοποίηση Αλγορίθμου για τον Υπολογισμό της Μικρότερης Γέφυρας Μεταξύ Δύο Ορθογωνίων Πολυγώνων

Νικόλαος Πουρλιάκας (ΑΜ: 3320)

Επιβλέπων: Λεωνίδας Παληός

Ιωάννινα, Ιούνιος 2024

Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πανεπιστήμιο Ιωαννίνων

ΠΕΡΙΛΗΨΗ (1)

Η εκπόνηση της παρούσας διπλωματικής εργασίας αφορά την υλοποίηση του **αλγορίθμου κατασκευής** μιας **βέλτιστης γέφυρας** μεταξύ δύο απλών ευθύγραμμων πολυγώνων, σε προγραμματιστικό περιβάλλον.

Μια τέτοια **γέφυρα** είναι ένα **ορθογώνιο μονοπάτι** που **συνδέει** ένα **σημείο στο ένα πολύγωνο** και ένα **σημείο στο άλλο** πολύγωνο και **ελαχιστοποιεί** τη **συνολική απόσταση** που προκύπτει από το **άθροισμα** του **μήκους** του **μονοπατιού** και της **απόστασης** του **μονοπατιού** των **άκρων** του στο κάθε πολύγωνο **από** τον **απώτερο γείτονα** τους στο ίδιο πολύγωνο.

Ο κώδικας υλοποιήθηκε στην προγραμματιστική γλώσσα **python**.

Τα βήματα του αλγορίθμου καθορίστηκαν με βάση την περιγραφή τους στην εργασία [1] του **D.P. Wang**.

Wang, D. P. (2001). An optimal algorithm for constructing an optimal bridge between two simple rectilinear polygons. Information processing letters, 79(5), 229-236.

ΠΕΡΙΛΗΨΗ (2)

Ο χρήστης παρέχει **δύο μη τεμνόμενα ορθογώνια πολύγωνα** στο επίπεδο, τα οποία αναπαρίστανται ως οι **συντεταγμένες** των **κόμβων** τους.

Θεωρούμε ότι βρίσκονται στην **Περίπτωση 1** εφόσον διαπιστώσουμε ότι υφίσταται **ευθύγραμμο γραμμικό τμήμα (η γέφυρα)** το οποίο **δεν διαπερνά** το **εσωτερικό** των **πολυγώνων**, και τα **ακριανά** του **σημεία** βρίσκονται το ένα **πάνω** σε **ακμή** του **ενός** πολυγώνου (άρα το ακριανό σημείο του τμήματος έχει ίδια τιμή στον ένα άξονα με δύο διαδοχικά σημεία του πολυγώνου) και το άλλο **πάνω** σε **ακμή** του **δεύτερου** πολυγώνου.

Ειδάλλως θεωρούμε ότι βρίσκονται στην **Περίπτωση 2**, και συνεπώς υφίστανται **δύο ευθύγραμμα τμήματα**, το **ένα οριζόντιο** και το **άλλο κάθετο**, τα οποία **δεν διαπερνούν** το **εσωτερικό** των **πολυγώνων** και έχουν **κοινό** το **ένα ακριανό** τους σημείο, και το **άλλο** ακριανό σημείο τους βρίσκεται πάνω σε **ακμή διαφορετικού πολυγώνου** από το άλλο.

ΜΕΡΟΣ 1:

ΕΙΣΑΓΩΓΗ

ΕΙΣΑΓΩΓΗ – ΒΑΣΙΚΟΙ ΟΡΙΣΜΟΙ (1)

- **<u>Πολύγωνο</u>**:

επίπεδο σχήμα που αποτελείται από μη τεμνόμενα ευθύγραμμα τμήματα (πλευρές) που ενώνονται ανά ζεύγη για να σχηματίσουν μια κλειστή διαδρομή.

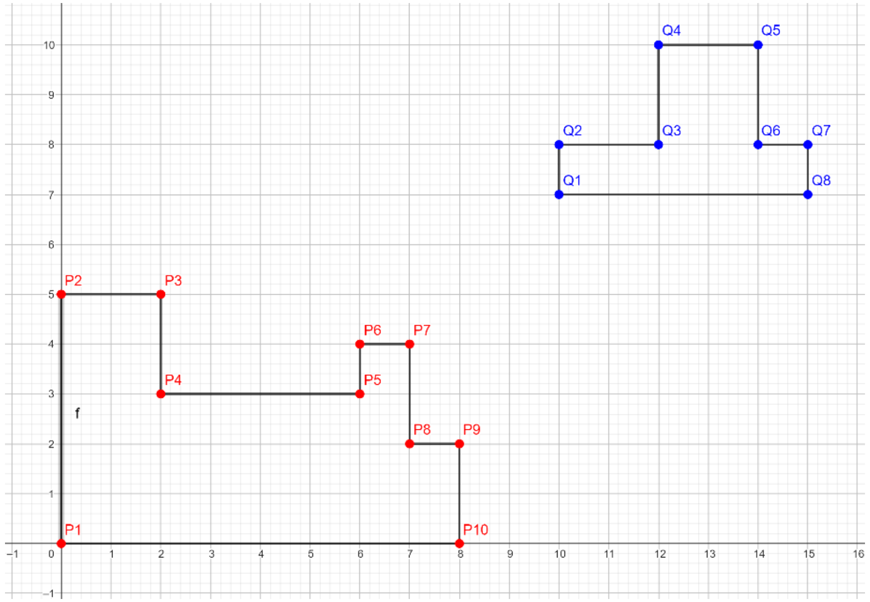
Στον κώδικα ορίζονται ως διαδοχικά σημεία, που αντιστοιχούν στους κόμβους τους.

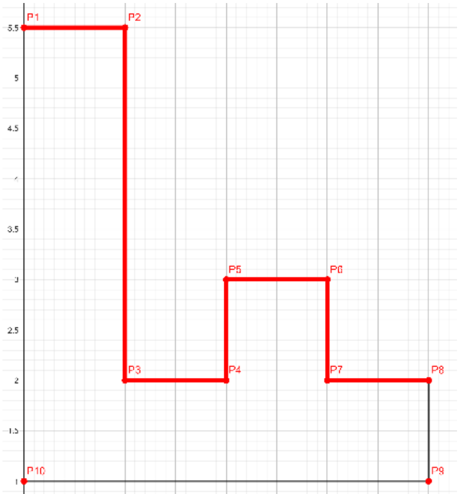
- **<u>Ορθογώνιο Πολύγωνο</u>**:

πολύγωνο που αποτελείται μόνο από ορθές γωνίες.

- **<u>Απόσταση</u>**:

Αναφερόμαστε συγκεκριμένα στην απόσταση του συντομότερου μονοπατιού που συνδέει 2 σημεία που ανήκουν στις ακμές ενός πολυγώνου.



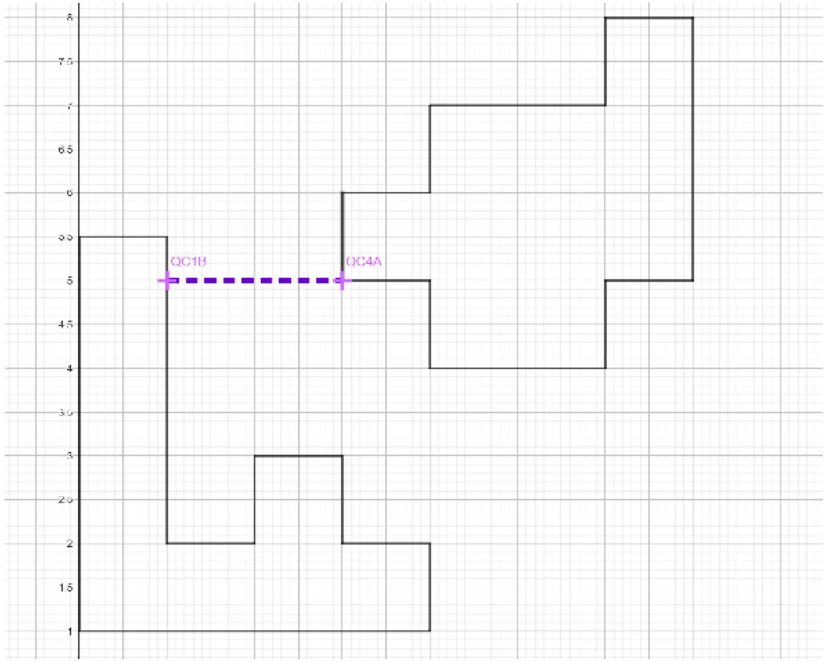


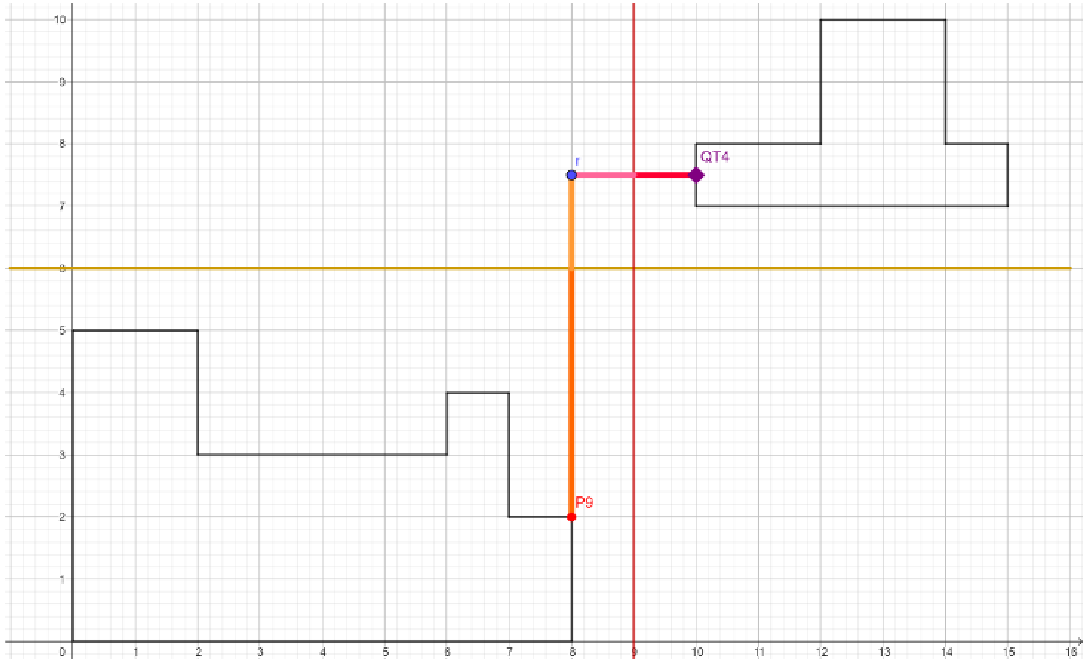
ΕΙΣΑΓΩΓΗ – ΒΑΣΙΚΟΙ ΟΡΙΣΜΟΙ (2)

- **Γέφυρα**:

Τα **δύο σημεία**, το **καθένα σε διαφορετικό πολύγωνο** (ασύνδετα μεταξύ τους) **αμοιβαία ορατά** από το ευθύγραμμο **τμήμα** που αποτελεί την **ευκλείδεια απόσταση** των σημείων, και **δεν διαπερνά** το **εσωτερικό** των **πολυγώνων**, και **ελαχιστοποιούν** το **άθροισμα** των **μονοπατιών** του καθενός τους **από** τον **απώτερο γείτονα τους** στο ίδιο πολύγωνο και της **ευκλείδειας απόστασης** των δύο σημείων.

Δεν είναι απαραίτητα μοναδική





ΕΙΣΑΓΩΓΗ – ΣΧΕΤΙΚΑ ΕΡΕΥΝΗΤΙΚΑ ΑΠΟΤΕΛΕΣΜΑΤΑ

- Στην εργασία [2] οι **S.K. Kim** και **C.S. Shin** εξέτασαν το πρόβλημα που αναφέραμε στην Περίληψη, με βάση αν τα πολύγωνα είναι κυρτά, και υπολόγισαν τις αντίστοιχες χρονικές πολυπλοκότητες που απαιτούνται για την επίλυση τους προβλήματος. Συγκεκριμένα καταλήγουν στο συμπέρασμα πως **για ζεύγος κυρτού με κυρτού ο χρόνος είναι O(n), για απλό με κυρτό είναι O(nlogn), και για απλό με απλό είναι $O(n^2)$**.

- Στην εργασία [1] ο **D.P. Wang** εξετάζει το πρόβλημα στην **περίπτωση** των **κυρτών ορθογωνίων πολυγώνων**, και παρουσιάζει **χρόνο επίλυσης O(n)**.

ΕΙΣΑΓΩΓΗ – ΘΕΩΡΗΤΙΚΟ ΥΠΟΒΑΘΡΟ (1)

Τα βήματα του αλγορίθμου στην εργασία 1 στην οποία βασίσαμε τον κώδικα που υλοποιήσαμε, είναι τα εξής:

Algorithm Optimal_Type_2_bridge (P, Q)

Input: Two polygons P and Q in Case 2

Output: An optimal Type 2 bridge $\overline{pr}$ + $\overline{rq}$

1: Find the sets *T(P)* and *T(Q)* of transition points and the sets $L_1P(P)$ and $L_1P(Q)$ of L_1-projection points.

2: Find $l_1(P)$, $l_2(P)$, $l_1(Q)$ and $l_2(Q)$.

3: Let

$\mu(p_1) = \min_{p \in l1(P)} \mu(p),$

$\mu(p2) = \min_{p \in l2(P)} \mu(p),$

$\mu(q1) = \min_{q \in l1(Q)} \mu(q),$ and

$\mu(q2) = \min_{q \in l2(Q)} \mu(q).$

4: If $F_2(p_1, q_2) \leqslant F2(p_2, q_1)$ then return $\overline{p_1 r}$ + $\overline{r_2 q}$ where $r = (x_{p1}, y_{q2})$; otherwise, return $\overline{p_2 r}$ + $\overline{r q_1}$ where $r = (x_{q1}, y_{p2})$.

ΕΙΣΑΓΩΓΗ – ΘΕΩΡΗΤΙΚΟ ΥΠΟΒΑΘΡΟ (2)

Τα βήματα του αλγορίθμου στο άρθρο «An optimal algorithm for constructing an optimal bridge between two simple rectilinear polygons», του D.P. Wang (2001), στον οποίο βασίσαμε τον κώδικα που υλοποιήσαμε, είναι τα εξής:

Algorithm Optimal_Type_1_bridge (P, Q)

Input: Two polygons P and Q in Case 1

Output: An optimal Type 1 bridge $\overline{pq}$

1: Candidate_bridge_set =∅.

2: Find the sets *T(P)* and *T(Q)* of transition points and the sets $L_1P(P)$ and $L_1P(Q)$ of L_1-projection points.

3: For every element *p* in $\{V(P) \cup T(P) \cup L_1P(P)\}$, find a set of points $S_q \subset \partial Q$, such that $\overline{pq}$ is a Type 1 bridge of polygons *P* and *Q* for $q \in S_q$.

Candidate_bridge_set = Candidate_bridge_set ∪ $\{\overline{pq}\}$ for $q \in S_q$.

4: For every element *q* in $\{V(Q) \cup T(Q) \cup L_1P(Q)\}$, find a set of points $S_p \subset \partial p$, such that $\overline{pq}$ is a Type 1 bridge of polygons *P* and *Q* for $p \in S_p$.

Candidate_bridge_set = Candidate_bridge_set ∪ $\{\overline{pq}\}$ for $p \in S_p$.

5: Evaluate the objective function $F_1(p,q)$ for every element $\overline{pq}$ in Candidate_bridge_set.

6: Output $\overline{pq}$ which is an element in Candidate_ bridge_set and has a minimum value of the objective function $F_1(p, q)$.

ΜΕΡΟΣ 2:

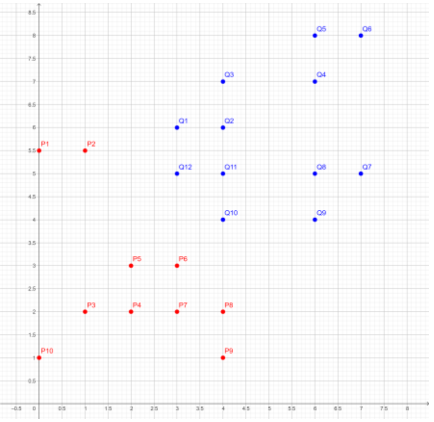
ΣΥΝΟΠΤΙΚΗ ΠΑΡΟΥΣΙΑΣΗ ΤΟΥ ΑΛΓΟΡΙΘΜΟΥ

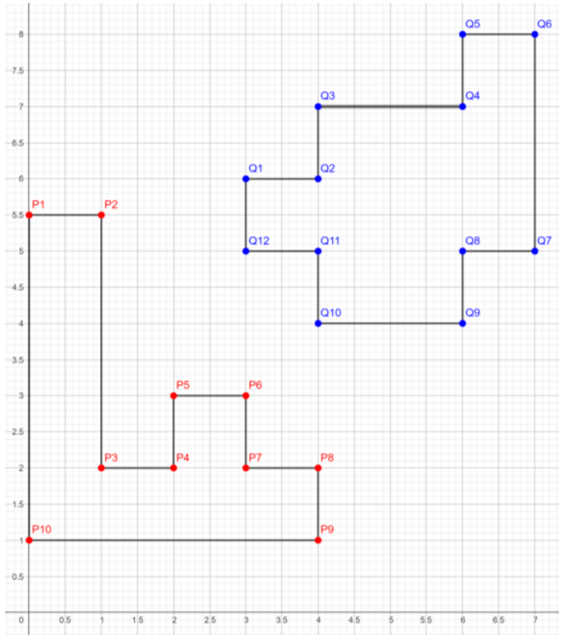
ΣΥΝΟΠΤΙΚΗ ΠΑΡΟΥΣΙΑΣΗ ΤΟΥ ΑΛΓΟΡΙΘΜΟΥ (1)

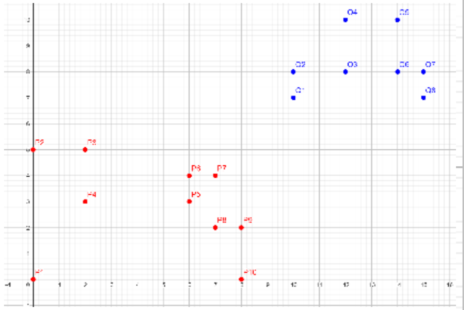
1) Αναγνωρίζουμε την **Περίπτωση** που βρίσκονται τα πολύγωνα, έστω P, Q, με **βάση** τους πιο **ακριανούς** τους **κόμβους** με βάση τους άξονες x και y.

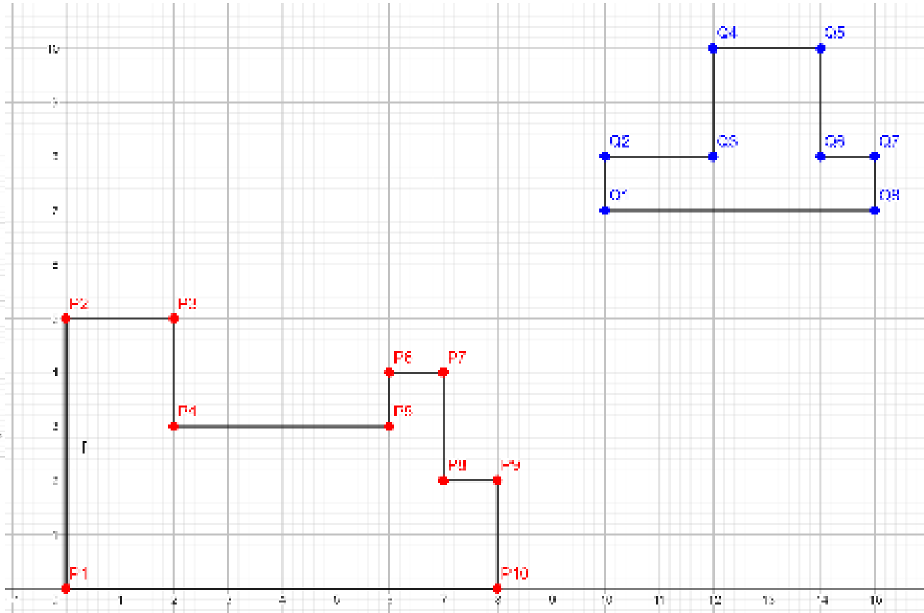
- Εφόσον **για 2 διαδοχικούς** ακριανούς κόμβους **του ενός** πολυγώνου **υπάρχει ακριανός** κόμβος που να έχει την **τιμή άξονα** που έχουν **κοινή** οι διαδοχικοί κόμβοι, **μικρότερη εκ του ενός**, και **μεγαλύτερη** από **του άλλου**, τότε ανήκει στην **Περίπτωση 1**.
- Ειδάλλως ανήκουν στην **Περίπτωση 2**.

Ουσιαστικά εξετάζουμε αν είναι εφικτό να ενωθούν 2 πλευρές, 1 από το κάθε πολύγωνο, με 1 κάθετη ευθεία, η οποία δεν διαπερνά τα πολύγωνα, ξεκινάει και καταλήγει σε σημεία που ανήκουν στις ακμές ή τους κόμβους των πολυγώνων.





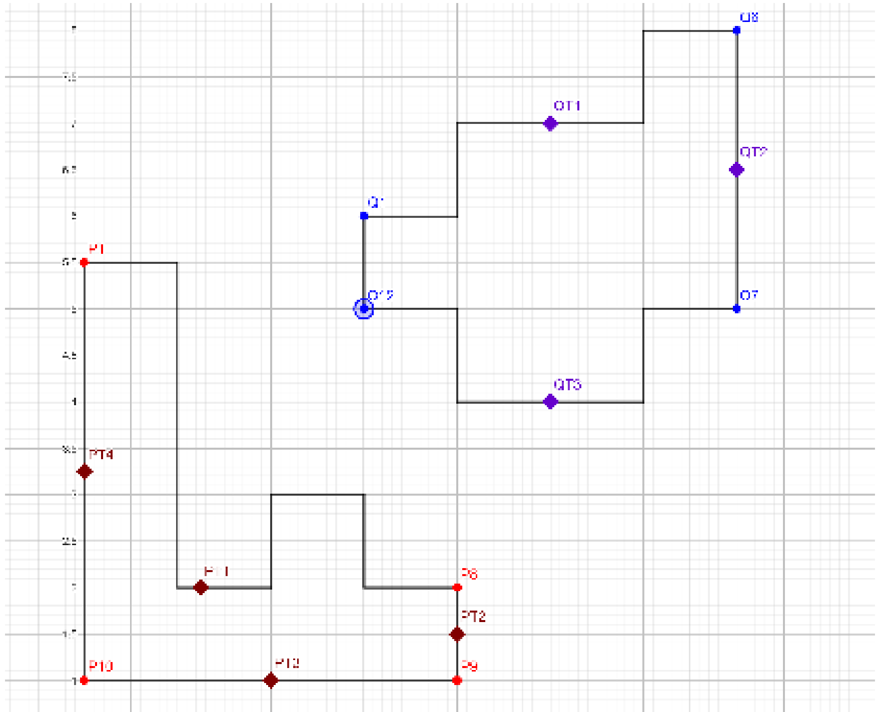


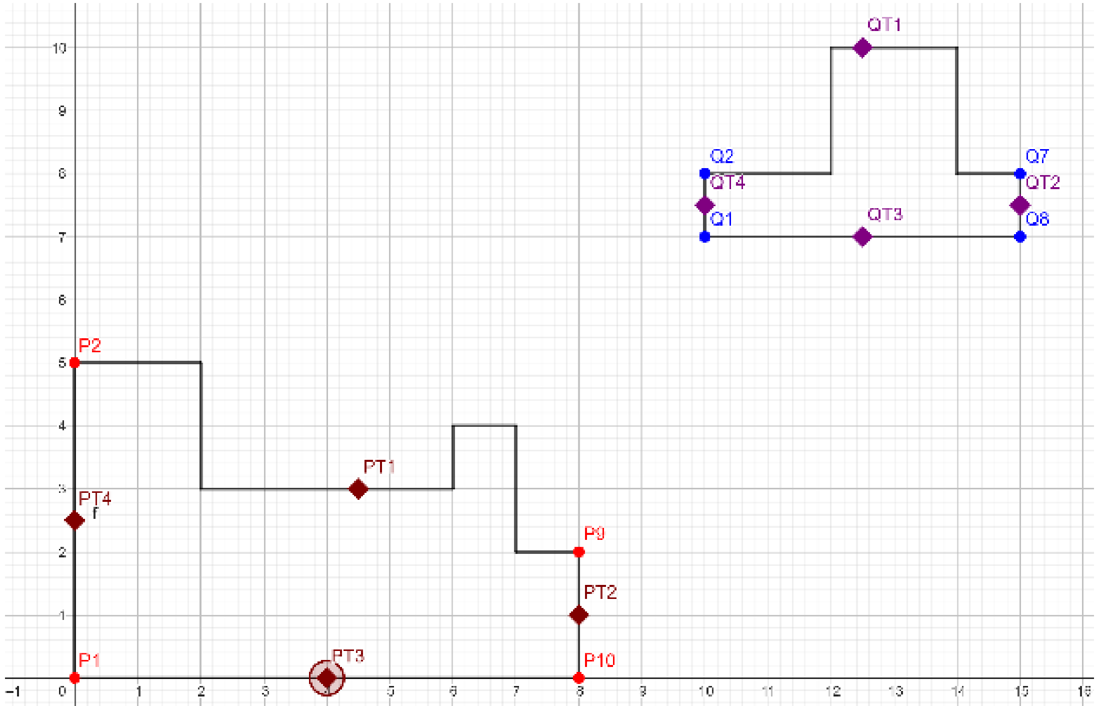


ΣΥΝΟΠΤΙΚΗ ΠΑΡΟΥΣΙΑΣΗ ΤΟΥ ΑΛΓΟΡΙΘΜΟΥ (2)

2) Βρίσκουμε **4 σημεία** στις ακμές του κάθε πολυγώνου, για τα οποία, **ανάμεσα σε διαδοχικά ζεύγη εξ αυτών**, τα **ενδιάμεσα** τους **σημεία**, (πάνω στις ακμές του πολυγώνου που ανήκουν), έχουν το **μεγαλύτερο μονοπάτι από τους 4 ακριανούς κόμβους** (του πολυγώνου που ανήκουν), οι οποίοι ορίζονται **πρωταρχικά** με τη **θέση** τους στον **x άξονα**,
και με **δευτερεύων κριτήριο** την **θέση** τους στον **y άξονα**.

Αναφερόμαστε στα σημεία ως **Σημεία Μετάβασης** (**Transition Points, T(P)**).



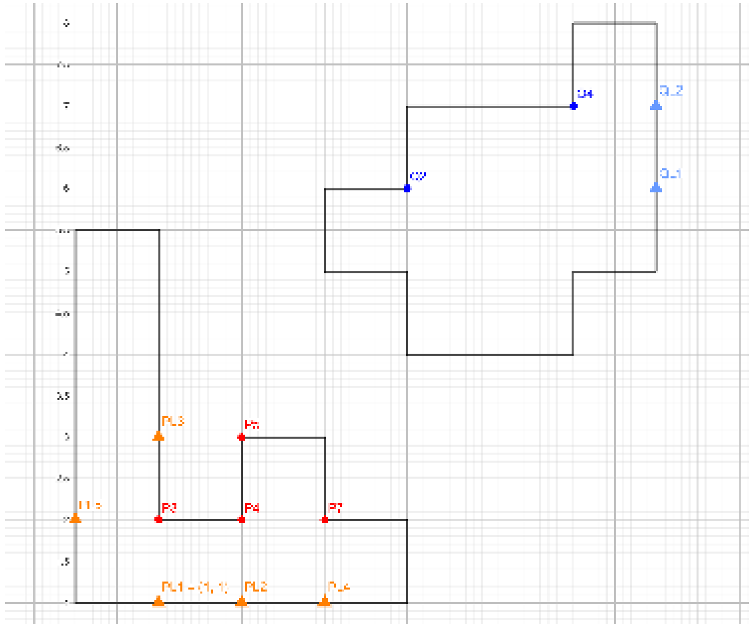


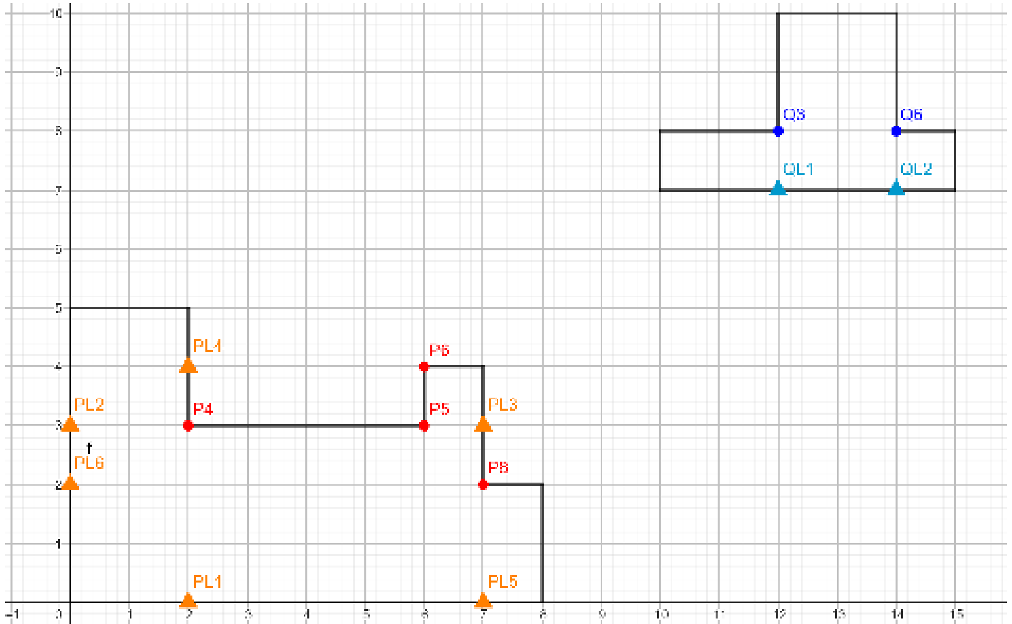
ΣΥΝΟΠΤΙΚΗ ΠΑΡΟΥΣΙΑΣΗ ΤΟΥ ΑΛΓΟΡΙΘΜΟΥ (3)

3) Υπολογίζουμε τα **Σημεία Προβολής** των **κόμβων** του κάθε πολυγώνου, καθώς και των **Σημείων Μετάβασης**, **δηλαδή** τα σημεία που **ανήκουν** στις **ακμές** του αντίστοιχου πολυγώνου, και έχουν **κοινή** τη **μια συντεταγμένη** του πολυγώνου, και την **άλλη διαφορετική**.

Είναι πιθανό να προκύψουν Σημεία Μετάβασης που ταυτίζονται με κόμβους καθώς τα πολύγωνα που εξετάζουμε είναι ορθογώνια, ωστόσο δεν τα συμπεριλαμβάνουμε, καθώς εξετάζονται στο σετ κόμβων, κατά τις διεργασίες που τα αφορούν.

Αναφερόμαστε στα σημεία ως **L_1-Προβολή Σημείου** (**L_1-Projection Point, L_1P(P)**).



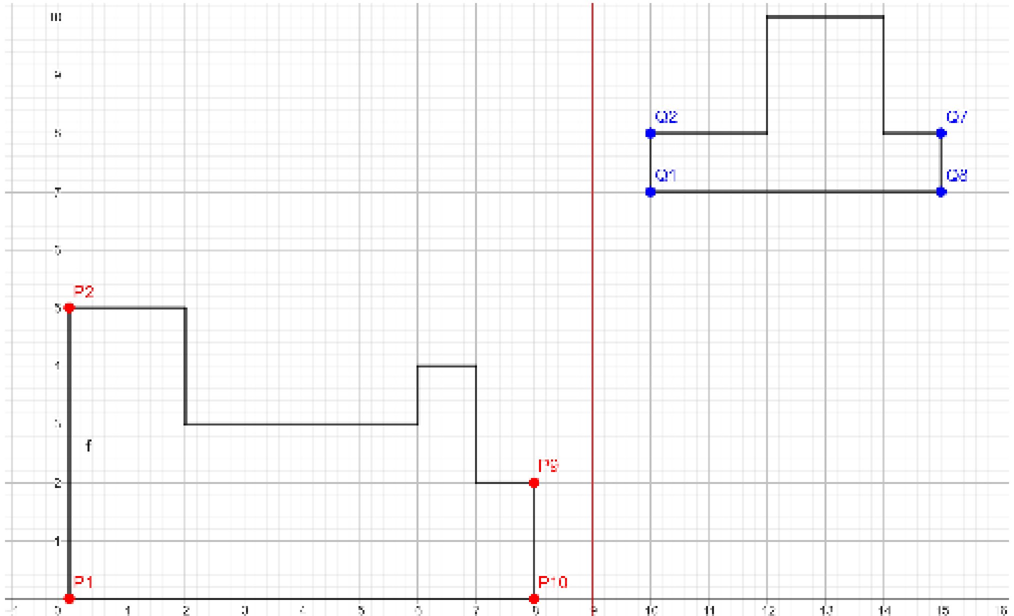


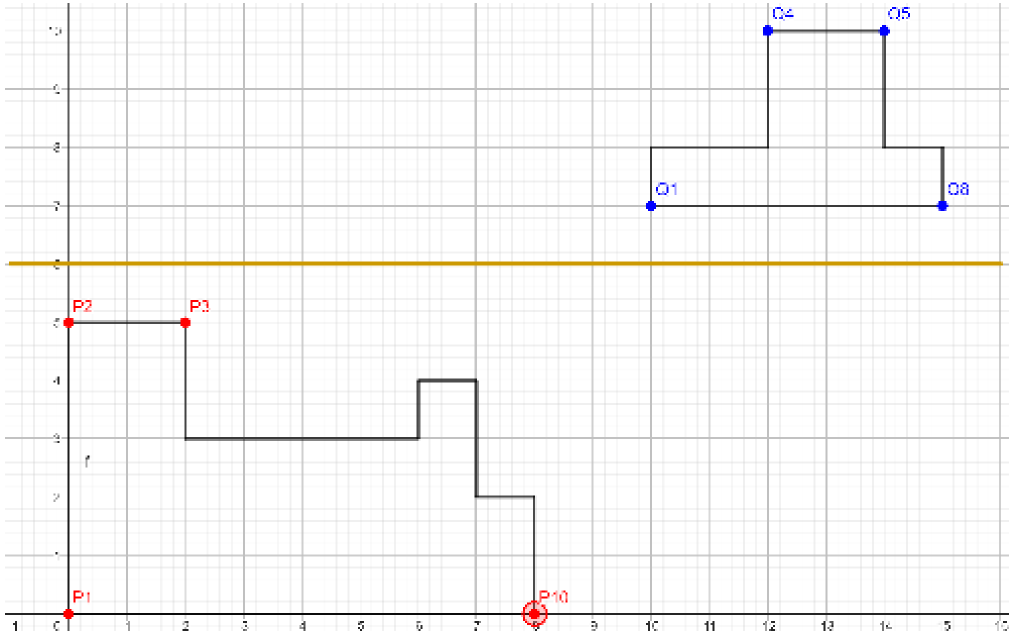
ΣΥΝΟΠΤΙΚΗ ΠΑΡΟΥΣΙΑΣΗ ΤΟΥ ΑΛΓΟΡΙΘΜΟΥ (4)

4) Επιλέγουμε **2 ευθύγραμμα τμήματα**, **1 κάθετο** και **1 οριζόντιο**, που **απέχουν ίση απόσταση** από τους πιο **ακριανούς** κόμβους των πολυγώνων, με βάση τον **x άξονα** για το **κάθετο**, και με βάση τον **y άξονα** για το **οριζόντιο**, δίχως το ευθύγραμμο τμήμα να έχει κοινή τιμή την μη σταθερή συντεταγμένη του με κάποιο από τα σημεία των πολυγώνων, εφόσον είναι εφικτό.

Συνεπώς επιλέγουμε τις ευθείες με βάση τους κοντινότερους ακριανούς κόμβους,
σε σχέση με τον άξονα που χρησιμοποιούμε ως κριτήριο για τη δημιουργία της κάθε ευθείας.

Αναφερόμαστε στο **κάθετο ευθύγραμμο τμήμα** ως $\mathbf{l_1}$ και στο **οριζόντιο** ως $\mathbf{l_2}$.



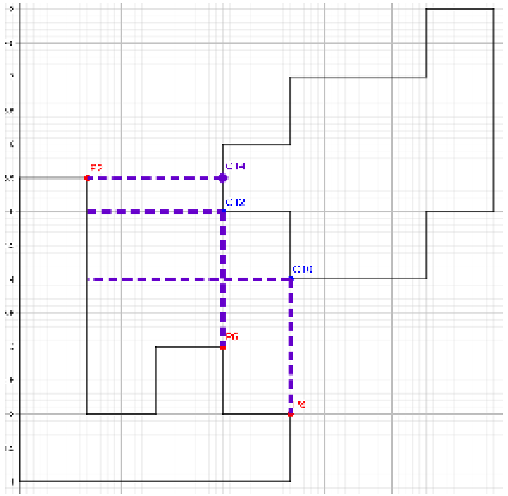


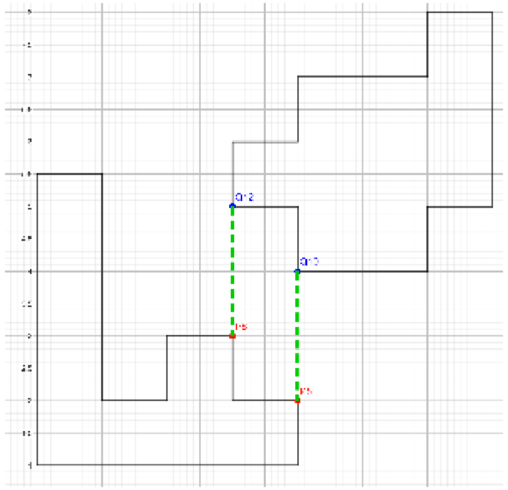
ΣΥΝΟΠΤΙΚΗ ΠΑΡΟΥΣΙΑΣΗ ΤΟΥ ΑΛΓΟΡΙΘΜΟΥ (5)

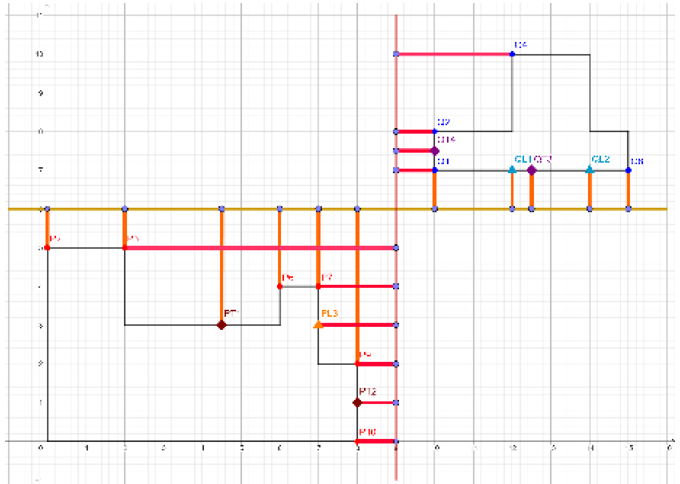
5) **Εντάσσουμε** σε νέες **λίστες**, 2 για το κάθε πολύγωνο, **1** για την **κάθε ευθεία**,
τα σημεία από τις λίστες των σημείων που καθορίσαμε στα προηγούμενα βήματα,
δηλαδή από τους **κόμβους**, τα **Σημεία Μετάβασης** και τα **Σημεία Προβολής**,
που είναι **αμοιβαία ορατά από** το **αντίστοιχο ευθύγραμμο τμήμα**.

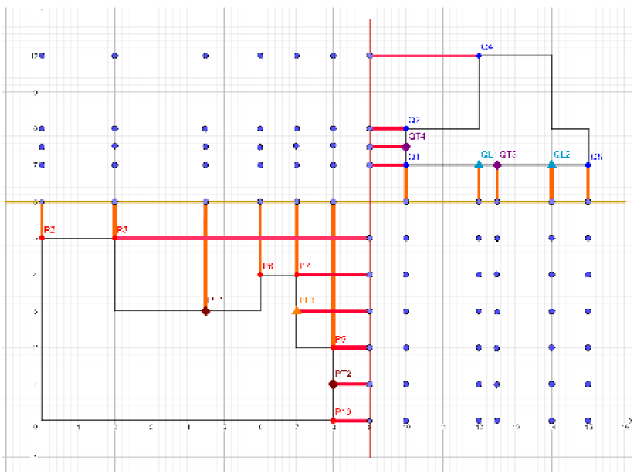
Δηλαδή ξεχωρίζουμε τα σημεία που μπορούν να συνδεθούν με ευθύγραμμο γραμμικό τμήμα,
κάθετο για το οριζόντιο, και αντίστροφα, δίχως το νέο ευθύγραμμο τμήμα που σχεδιάζουμε,
να διαπερνά τις ακμές των πολυγώνων, άρα να μην υπάρχουν σημεία στο ίδιο πολύγωνο,
με μικρότερη διαφορά τιμών στον άξονα που εξετάζουμε, συγκριτικά με αυτή του νέου ευθύγραμμου τμήματος.

Αναφερόμαστε στο **σύνολο** των σημείων του κάθε πολυγώνου,
που είναι ορατά από το **κάθετο** τμήμα ως **σετ** $\mathbf{l_1(P)}$, και από το **οριζόντιο** ως **σετ** $\mathbf{l_2(P)}$.







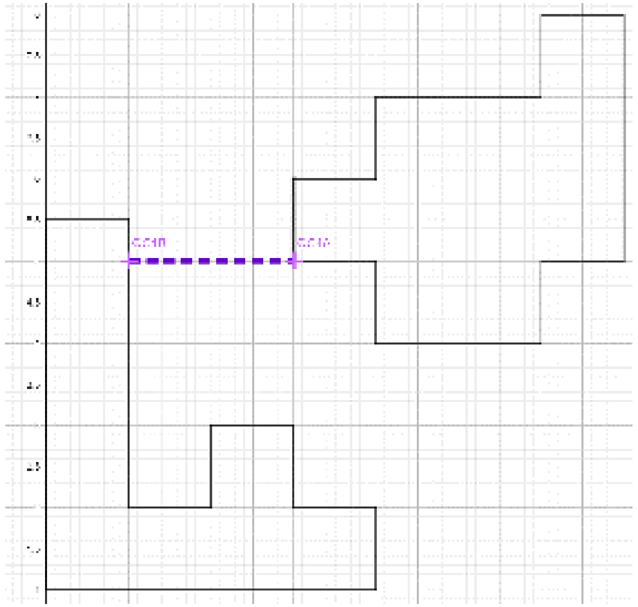


ΣΥΝΟΠΤΙΚΗ ΠΑΡΟΥΣΙΑΣΗ ΤΟΥ ΑΛΓΟΡΙΘΜΟΥ (6)

6) Υπολογίζουμε **για κάθε σημείο** των **σετ $l_1(P)$** και **$l_2(P)$** την **συνολική** του **απόσταση μονοπατιού** από τον απώτερο γείτονα του, σε συνδυασμό με την οριζόντια και την κάθετη απόσταση του από το αντίστοιχο ευθύγραμμο τμήμα.

Αποθηκεύουμε το σημείο του κάθε σετ του κάθε πολυγώνου με την **μικρότερη τιμή**.

Αναφερόμαστε στα σημεία αυτά ως **$\mu(p_1)$** και **$\mu(p_2)$**.



7) **Προσθέτουμε** τις τιμές του ενός **μ()** από το κάθε πολύγωνο για διαφορετικό ευθύγραμμο τμήμα, δηλαδή τα **$\mu(p_1)$** και **$\mu(q_2)$**, και αντίστροφα, **$\mu(p_2)$** και **$\mu(q_1)$**.

Αναφερόμαστε στα **αθροίσματα** αυτά ως **$F_2(p_1, q_2)$** και **$F_2(p_2, q_1)$** αντίστοιχα.

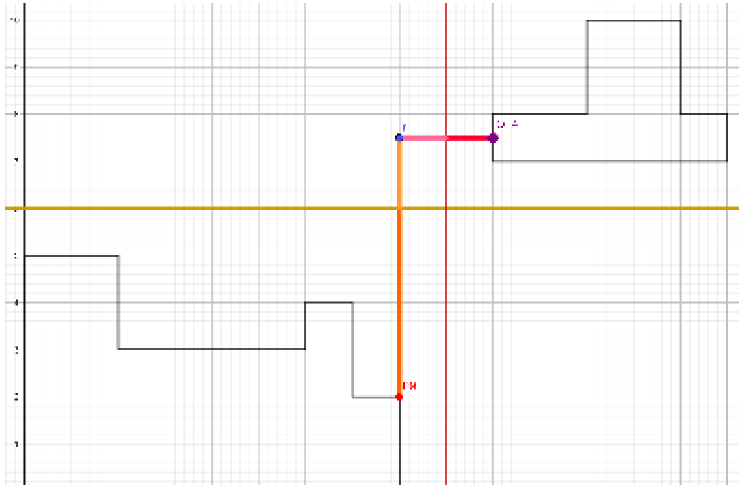
8) **Συγκρίνουμε** τα **$F_2(p_1, q_2)$** και **$F_2(p_2, q_1)$**, και με βάση το **μικρότερο** εξ αυτών, υπολογίζουμε τη **γέφυρα** που προκύπτει από τα σημεία που το συντελούν,

- <u>δηλαδή στην **Περίπτωση 1**</u>**:**

  τα **2 σημεία** που **ενώνονται** άμεσα με 1 ευθύγραμμο τμήμα,

- <u>και στην **Περίπτωση 2**</u>**:**

  τα **σημεία** από τα οποία **ξεκινάν** το **κάθετο** και το **οριζόντιο** ευθύγραμμο τμήμα, καθώς και το **σημείο** στο οποίο **συμπίπτουν**, στο οποίο αναφερόμαστε ως **σημείο r**.



ΜΕΡΟΣ 3:

ΑΝΑΛΥΤΙΚΗ ΠΑΡΟΥΣΙΑΣΗ ΤΟΥ ΑΛΓΟΡΙΘΜΟΥ

ΑΝΑΛΥΤΙΚΗ ΠΑΡΟΥΣΙΑΣΗ ΤΟΥ ΑΛΓΟΡΙΘΜΟΥ (1)

Θεωρούμε ότι τα 2 πολύγωνα είναι στην Περίπτωση εάν συνδέονται:

1. με **1 ευθύγραμμο γραμμικό τμήμα, κάθετο ή οριζόντιο**.

Ενδεχομένως, ανάλογα με την τοποθέτηση τους στο επίπεδο, να υφίστανται ταυτοχρόνως κάθετα και οριζόντια ευθύγραμμα γραμμικά τμήματα που τα συνδέουν.

2. με **2 ευθύγραμμα γραμμικά τμήματα, 1 κάθετο και 1 οριζόντιο**.

Να σημειωθεί ότι τα ευθύγραμμα τμήματα δεν πρέπει να εισέρχονται στις εσωτερικές περιοχές των πολυγώνων, συνεπώς δεν μπορούν να ανήκουν στις ακμές των πολυγώνων.

	Υπάρχουν τουλάχιστον 2 σημεία σε ακμή, καθένα σε διαφορετικού πολυγώνου, με	
ΣΕΝΑΡΙΟ	ίδια συντεταγμένη **x**	ίδια συντεταγμένη **y**
1	✓	×
2	×	✓

	Όλοι οι κόμβοι του πολυγώνου, βρίσκονται συγκριτικά με του άλλου (αρκεί να συγκρίνουμε τους κόμβους τους με τις μεγαλύτερες τιμές y και x)			
ΣΕΝΑΡΙΟ	Πιο πάνω	Πιο κάτω	Πιο δεξιά	Πιο αριστερά
1	✓	×	✓	×
2	✓	×	×	✓
3	×	✓	✓	×
4	×	✓	×	✓

ΑΝΑΛΥΤΙΚΗ ΠΑΡΟΥΣΙΑΣΗ ΤΟΥ ΑΛΓΟΡΙΘΜΟΥ (2)

Είσοδος:	2 πολύγωνα P και Q στην Περίπτωση 2
Έξοδος:	Βέλτιστη Τύπου 2 γέφυρα $\overline{pr} + \overline{rq}$.

ΒΗΜΑΤΑ Αλγορίθμου 2 - Optimal_Type_2_bridge(P ,Q)			
	Βρίσκουμε:		
1	T(P)	T(Q)	των Σημείων Μετάβασης
	$L_1P(P)$	$L_1P(Q)$	των L1-προβολή Σημείων
2	$l_1(P)$	$l_1(Q)$	l_1 : οριζόντια γραμμή ανάμεσα στα P, Q
	$l_2(P)$	$l_2(Q)$	l_2 : κάθετη γραμμή ανάμεσα στα P, Q
3	$\mu(p_1) = \min_{p \in l1(P)} \mu(p)$	$\mu(p_2) = \min_{p \in l2(P)} \mu(p)$	$\mu(k) = L_1(\, k, f(k)\,) + \lvert x_k \rvert + \lvert y_k \rvert$
	$\mu(q_1) = \min_{q \in l1(Q)} \mu(q)$	$\mu(q_2) = \min_{q \in l2(Q)} \mu(q)$	
4	$\overline{p1r} + \overline{rq2}$, όπου $r = (x_{p1}, y_{q2})$		Αν $F_2(p_1, q_2) <= F_2(p_2, q_1)$
	$\overline{p2r} + \overline{rq1}$, όπου $r = (x_{q1}, y_{p2})$		αλλιώς

ΑΝΑΛΥΤΙΚΗ ΠΑΡΟΥΣΙΑΣΗ ΤΟΥ ΑΛΓΟΡΙΘΜΟΥ (3)

➢ Εύρεση των **σετ T(P), T(Q) των Σημείων Μετάβασης**

Διευκρινίζουμε κάποιους από τους όρους που αναφέρουμε:

- **T(P)** $= \{\, t \mid t$ μεταβατικό σημείο του $B(v_i) \neq \emptyset$, αν $B(v_i) = \emptyset,\ 0 \leq i \leq |V(P)| - 1 \,\}$

- **B(v_i)** $= \{\, x \mid x \in \partial P$ και $f(x) = v_i \,\}$

  Το σύνολο σημείων στο όριο του πολυγώνου, με L_1-μακρύτεροι γείτονές v_i.

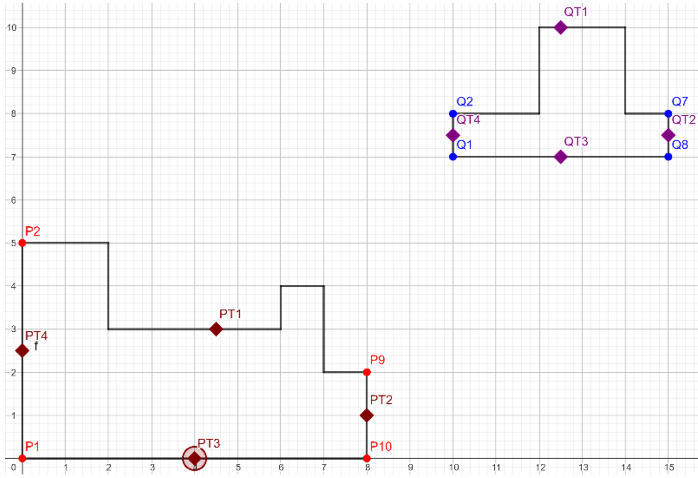
- **f(x)** : L_1-απώτατος γείτονας του x => κόμβος με μέγιστη L_1-απόσταση από x

- **L_1-απόσταση** : το μήκος συντομότερου ευθύγραμμου μονοπατιού,
  που συνδέει 2 σημεία του πολυγώνου.

ΑΝΑΛΥΤΙΚΗ ΠΑΡΟΥΣΙΑΣΗ ΤΟΥ ΑΛΓΟΡΙΘΜΟΥ (4)

➢ Διαδικασία Εύρεσης των **σετ T(P), T(Q) των Σημείων Μετάβασης**

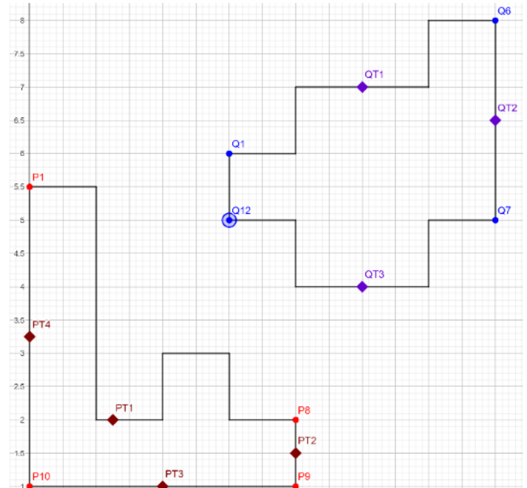
1) Επιλέγουμε τους **4 πιο ακρινούς κόμβους** του πολυγώνου,
με **βάση** την **x** συντεταγμένη τους (2 αριστερά, 2 δεξιά)
(επιλέγουμε το πρώτο κόμβο με την μεγαλύτερη συντεταγμένη x,
εξετάζοντας δεξιόστροφα).



2) Υπολογίζουμε την **συνολική απόσταση** για τα εξής **ζεύγη** ακρινών κόμβων:

i. πάνω_αριστερά - πάνω_δεξιά,
ii. πάνω_δεξιά - κάτω_δεξιά,
iii. κάτω_δεξιά - κάτω_αριστερά,
iv. κάτω_αριστερά - πάνω_αριστερά.

Για κάθε ζεύγος, για κάθε ενδιάμεση ακμή στο μονοπάτι τους, προσθέτουμε σε **καταμετρητή**, την **απόλυτη διαφορά** των τιμών, των **μη κοινών συντεταγμένων** των κόμβων που την αποτελούν.

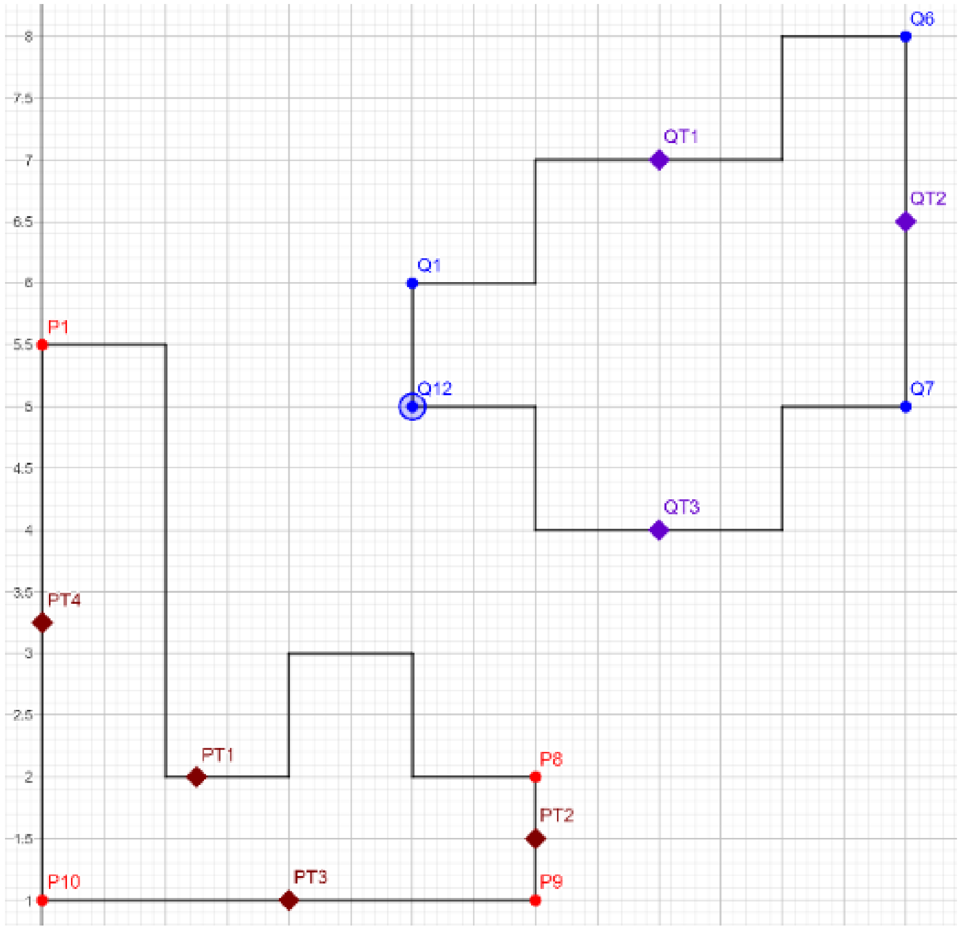


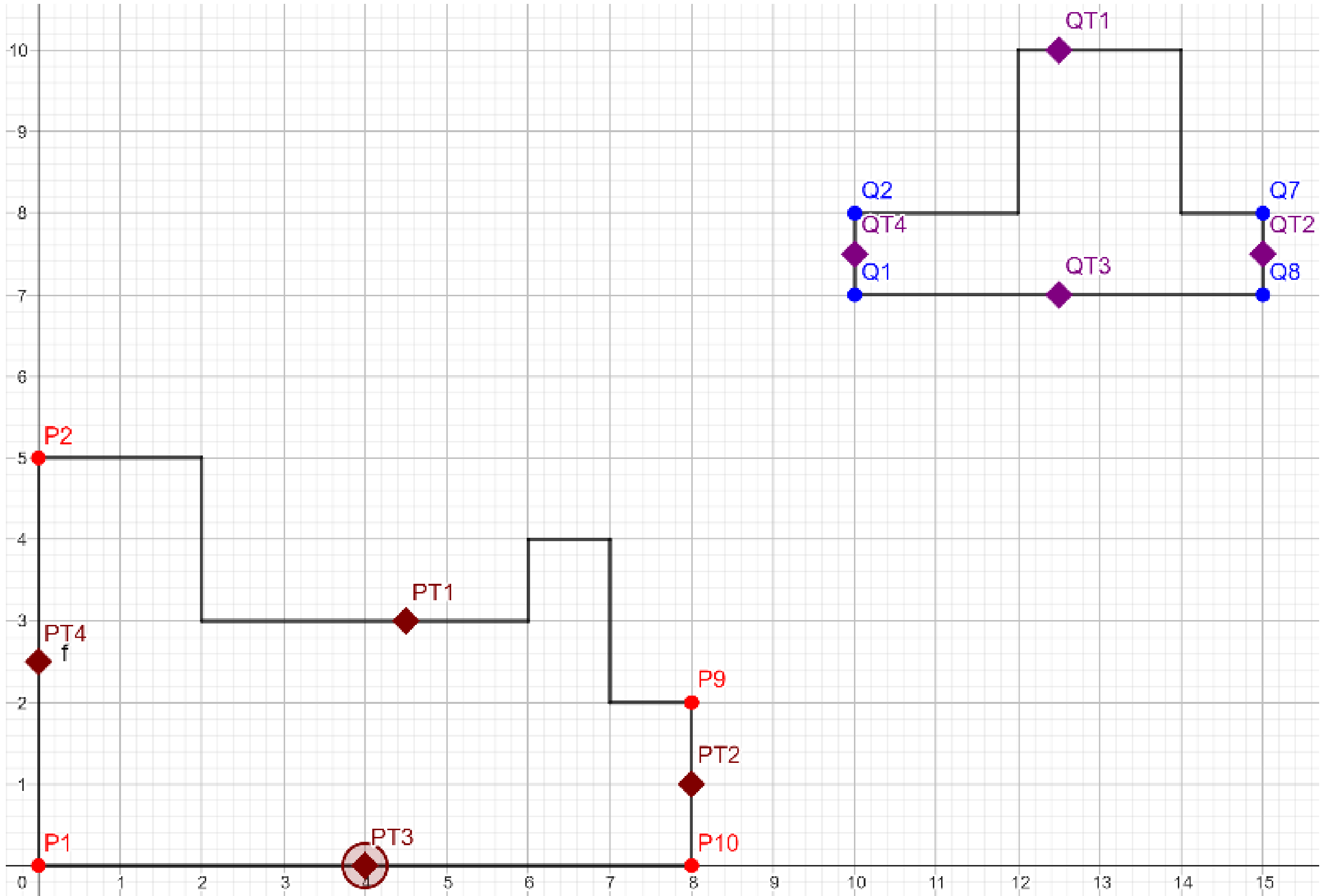
3) Υπολογίζουμε **για κάθε ζεύγος**, το **σημείο**, το οποίο ανήκει σε κάποια **από τις ακμές** του μονοπατιού, που βρίσκεται στη **μισή απόσταση**, διανύοντας τις ακμές, από την συνολική που υπολογίσαμε.

Τα 4 σημεία που θα προκύψουν, είναι τα **4 Σημεία Μετάβασης**.

ΑΝΑΛΥΤΙΚΗ ΠΑΡΟΥΣΙΑΣΗ ΤΟΥ ΑΛΓΟΡΙΘΜΟΥ (5)

➢ Εύρεση των **σετ T(P), T(Q) των Σημείων Μετάβασης**





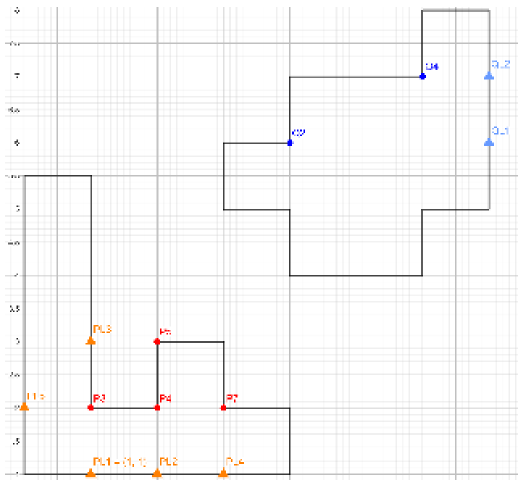
ΑΝΑΛΥΤΙΚΗ ΠΑΡΟΥΣΙΑΣΗ ΤΟΥ ΑΛΓΟΡΙΘΜΟΥ (6)

➢ Διαδικασία Εύρεσης των **σετ L_1P(P), L_1P(P) των L_1-προβολή Σημείων**

L_1 (v_i, t) = L_1 (v_i, e) : το **σημείο t προβάλλεται από** τον **κόμβο v_i στην ακμή e**.

Τα L_1-προβολή Σημεία είναι οι προβολές των κόμβων του πολυγώνου επάνω στις ακμές.

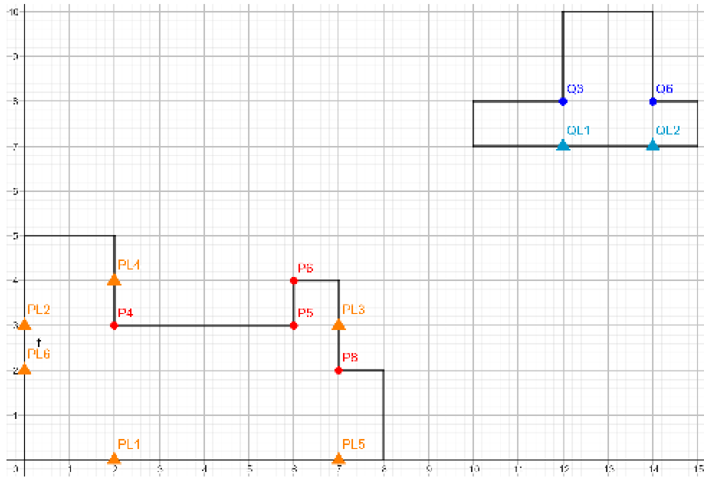
Δηλαδή, είναι σημεία, που δεν ταυτίζεται η θέση τους με άλλους κόμβους του πολυγώνου, ανήκουν σε μια ακμή, και έχουν κοινή τιμή μόνο στον έναν άξονα, με τον αντίστοιχο κόμβο τους. Προκειμένου να εντοπίσουμε το Σημείο Προβολής, εντοπίζουμε όλους τους διαδοχικούς κόμβους με διαφορετική τιμή από τον κόμβο που συγκρίνουμε στην μια συντεταγμένη, και με τιμή της άλλης συντεταγμένης, για τον έναν μεγαλύτερη, και για τον άλλον μικρότερη, ώστε να είναι ανάμεσα τους.



1) **Επιλέγουμε διαδοχικά** όλους τους **κόμβους** του πολυγώνου.

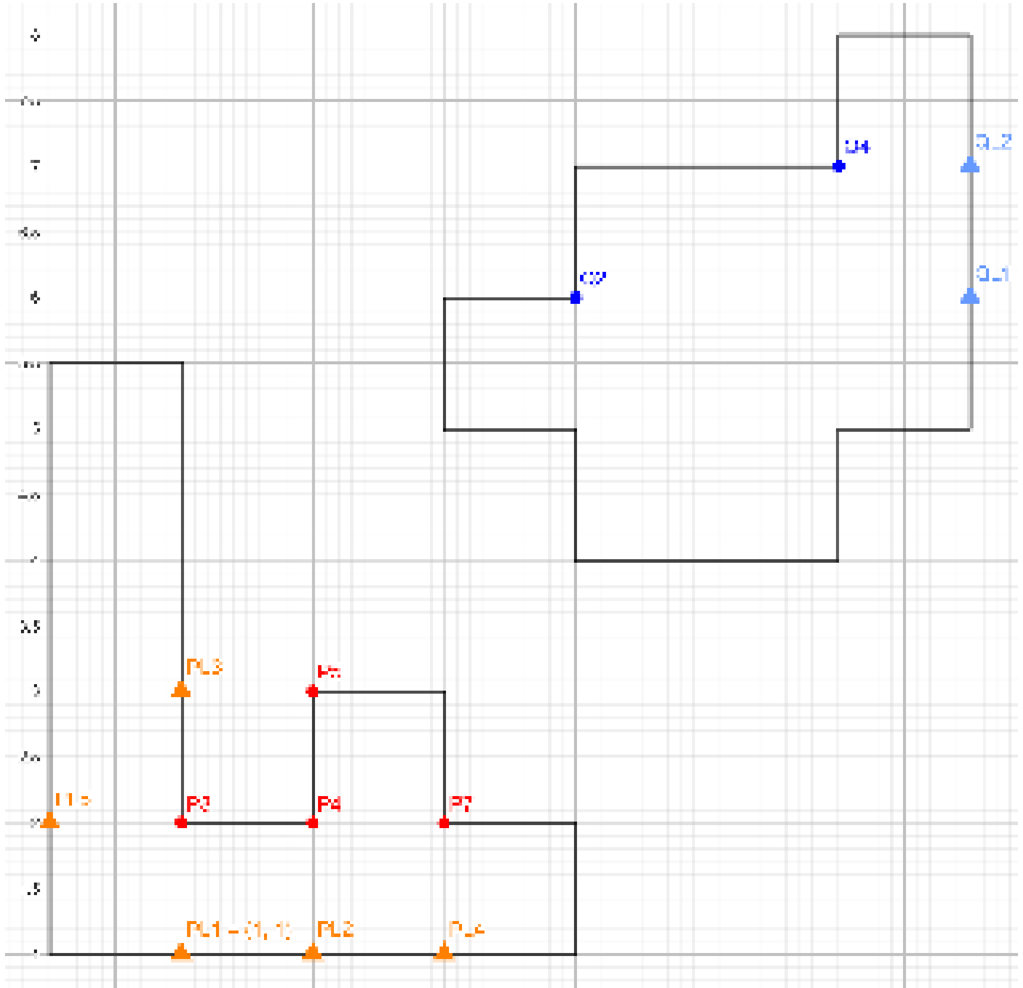
2) Τους **συγκρίνουμε** με όλα τα υπόλοιπα **ζεύγη** κόμβων, που δεν περιλαμβάνουν τον επιλεχθέντα, και **αποθηκεύουμε** τα **ζεύγη** που η **κοινή συντεταγμένη** τους είναι **διαφορετική του επιλεχθέντα**, και **η άλλη** τους είναι η **μια μεγαλύτερη** και η **άλλη μικρότερη** από του επιλεχθέντα, σε λίστα.

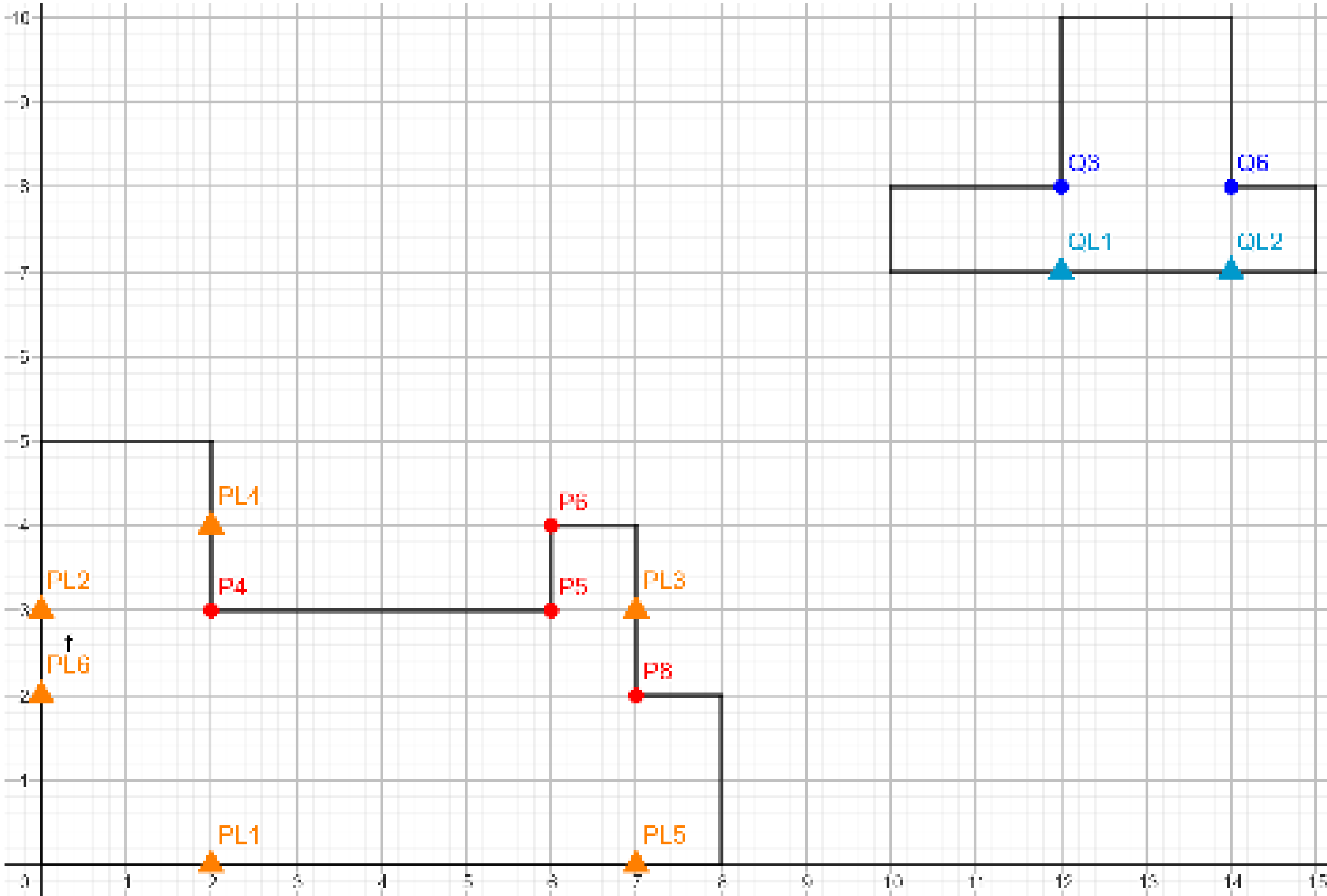
Στο τέλος η λίστα περιλαμβάνει όλα τα L_1-προβολή Σημεία των κόμβων του πολυγώνου.



ΑΝΑΛΥΤΙΚΗ ΠΑΡΟΥΣΙΑΣΗ ΤΟΥ ΑΛΓΟΡΙΘΜΟΥ (7)

➢ <u>Εύρεση των **σετ $L_1P(P)$, $L_1P(P)$ των L_1-προβολή Σημείων**</u>

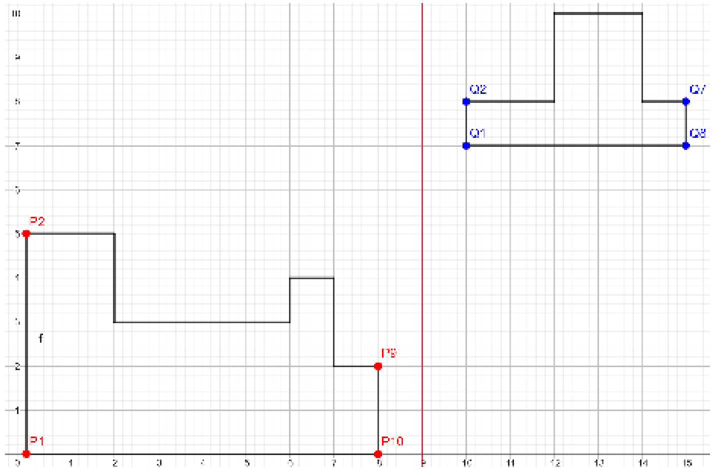




ΑΝΑΛΥΤΙΚΗ ΠΑΡΟΥΣΙΑΣΗ ΤΟΥ ΑΛΓΟΡΙΘΜΟΥ (8)

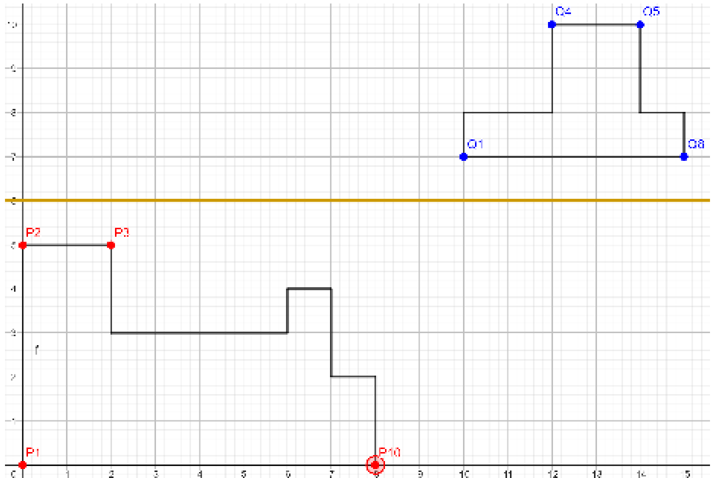
➢ Διαδικασία Εύρεσης του **οριζόντιου ευθύγραμμου τμήματος $\mathbf{l_1}$**

$\mathbf{l_1}$: **οριζόντιο ευθύγραμμο τμήμα ανάμεσα** στα πολύγωνα P, Q,
κάτω από το πιο πάνω, και πιο πάνω από το πιο κάτω.



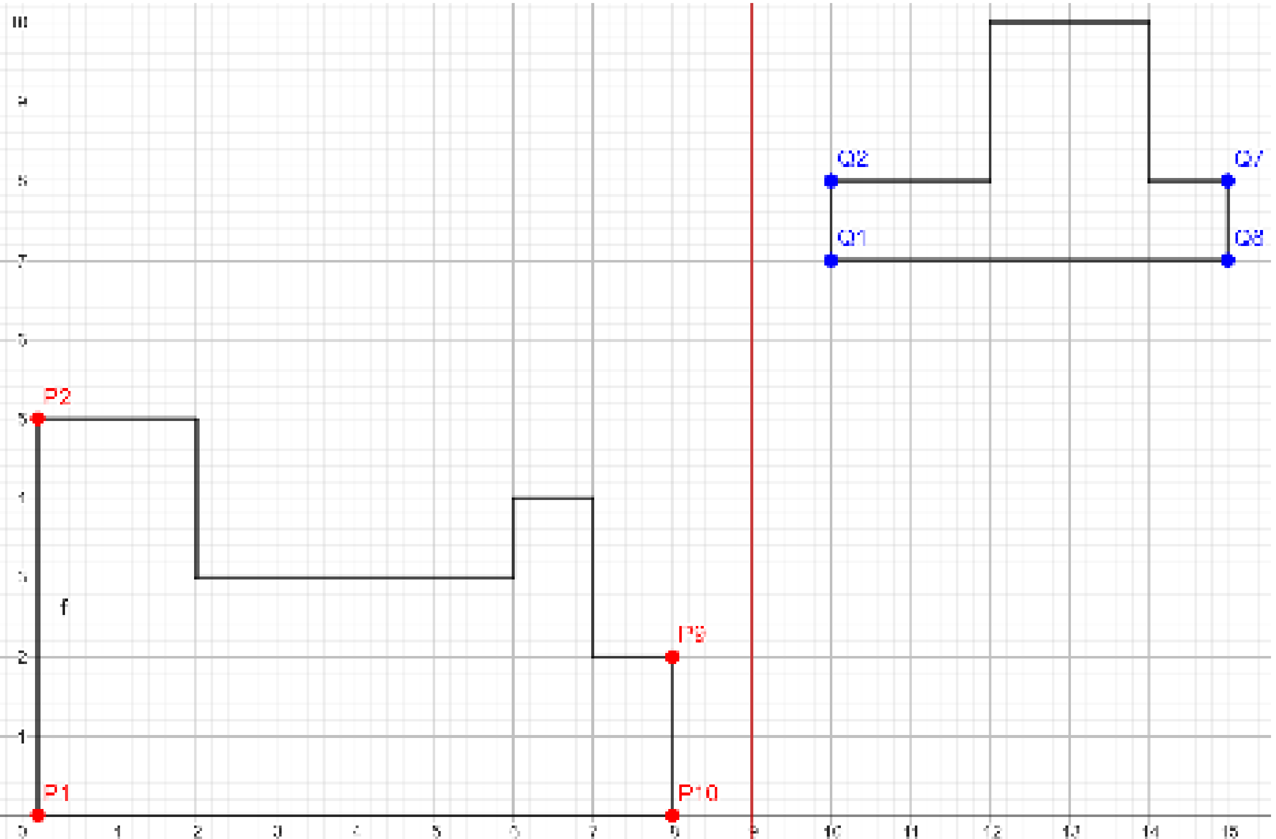
$\mathbf{l_2}$: **κάθετο ευθύγραμμο τμήμα ανάμεσα** στα πολύγωνα P, Q,
πιο αριστερά από το δεξιά, και αντίστροφα.

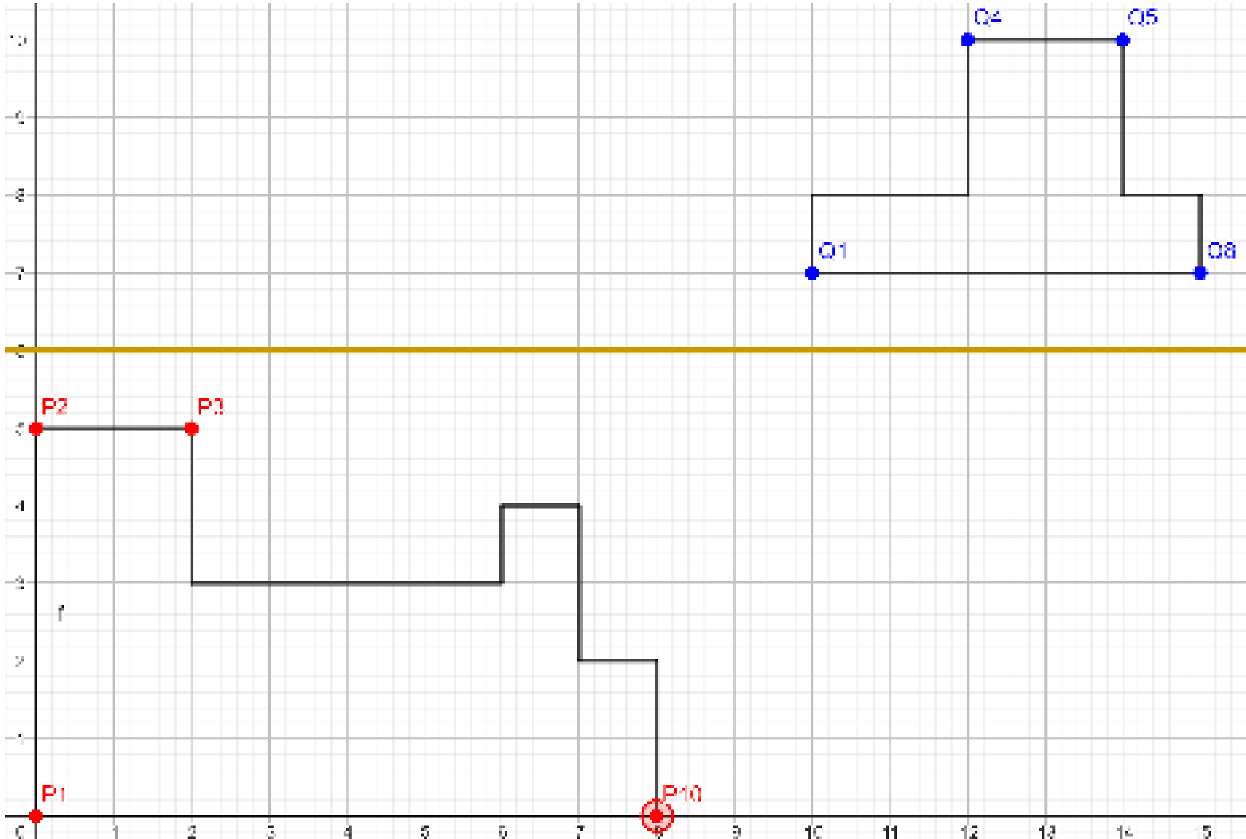
Κατά την επιλογή στον κώδικα, καθορίζουμε τις συντεταγμένες
όλων των σημείων της $\mathbf{l_1}$ ($\mathbf{l_2}$) με ίδια y (x) συντεταγμένη,
την ενδιάμεση από την απόσταση των πιο κοντινών ακρινών (y (x))
ανάμεσα στα 2 πολύγωνα, και x (y) συντεταγμένη κατά 1 μεγαλύτερη
στην κάθε άκρη από τους πιο ακρινούς αντίστοιχους κόμβους
με μεγαλύτερη απόσταση ανάμεσα στα πολύγωνα.



ΑΝΑΛΥΤΙΚΗ ΠΑΡΟΥΣΙΑΣΗ ΤΟΥ ΑΛΓΟΡΙΘΜΟΥ (9)

➤ <u>Εύρεση του **οριζόντιου ευθύγραμμου τμήματος l_1**</u>





ΑΝΑΛΥΤΙΚΗ ΠΑΡΟΥΣΙΑΣΗ ΤΟΥ ΑΛΓΟΡΙΘΜΟΥ (10)

➢ Διαδικασία Εύρεσης των **σετ l_1(P), l_1(Q), l_2(P), l_2(Q)**

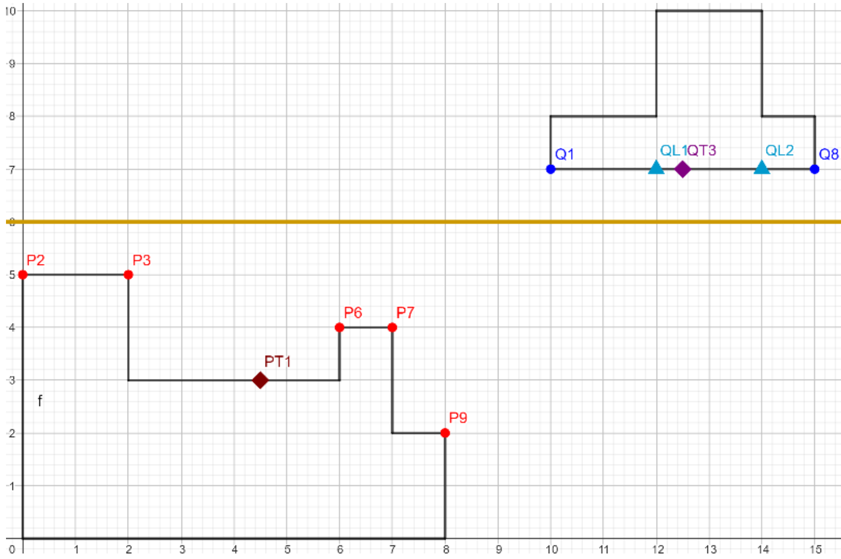
- **l_1(P) (l_2(P)) :**
  το **υποσύνολο** των σημείων του **V(P) ∪ T(P) ∪ L1 P(P)** ,
  που είναι αμοιβαία **ορατά απ' την l_1 (l_2)**, δηλαδή συνδέονται
  με κάθετο (οριζόντιο) ευθύγραμμο γραμμικό τμήμα μη εντός του P.

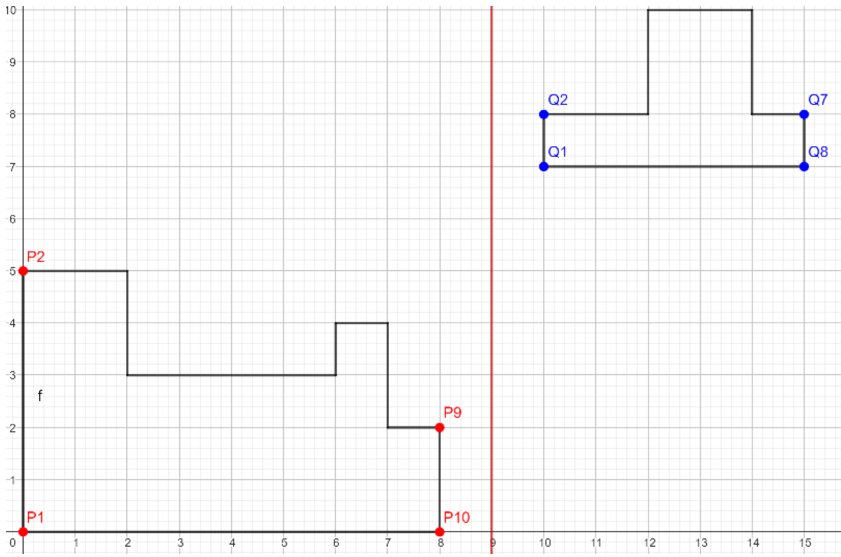
- **l_1(Q) (l_2(Q)) :**
  το **υποσύνολο** των σημείων του V(Q) ∪ T(Q) ∪ L1 P(Q) ,
  που είναι αμοιβαία **ορατά απ' την l_1 (l_2),** δηλαδή συνδέονται
  με κάθετο (οριζόντιο) ευθύγραμμο γραμμικό τμήμα μη εντός του Q.

- **V(polygon)**: το σύνολο των **κόμβων** που απαρτίζουν το πολύγωνο.
- **T(polygon)**: το σύνολο των **Σημείων Μετάβασης** του πολυγώνου.
- **L_1 P(polygon)**: το σύνολο των **L_1-προβολή Σημείων** του πολυγώνου.

1) Ορίζουμε το l_1 (l_2) ευθύγραμμο οριζόντιο τμήμα που βρίσκεται στη μέση της απόστασης του πιο δεξιού κόμβου του αριστερού πολυγώνου, και του πιο αριστερού κόμβου του δεξιού.

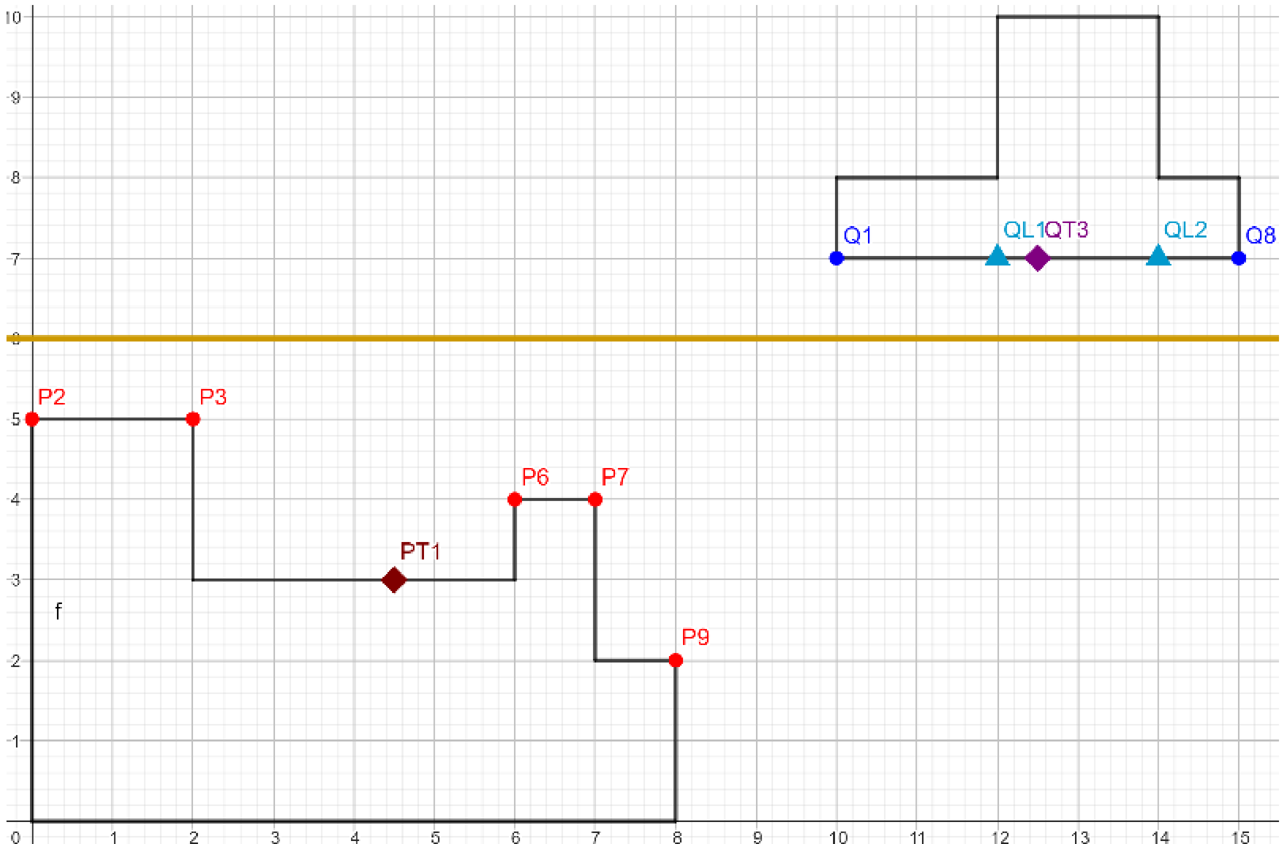
2) Εξετάζουμε αν το πολύγωνο είναι πάνω ή κάτω από την l_1 (l_2), και επιλέγουμε μόνο τα σημεία που δεν έχουν από κάτω ή από πάνω (αριστερά ή δεξιά) τους, αντίστοιχα, κάποια ακμή.

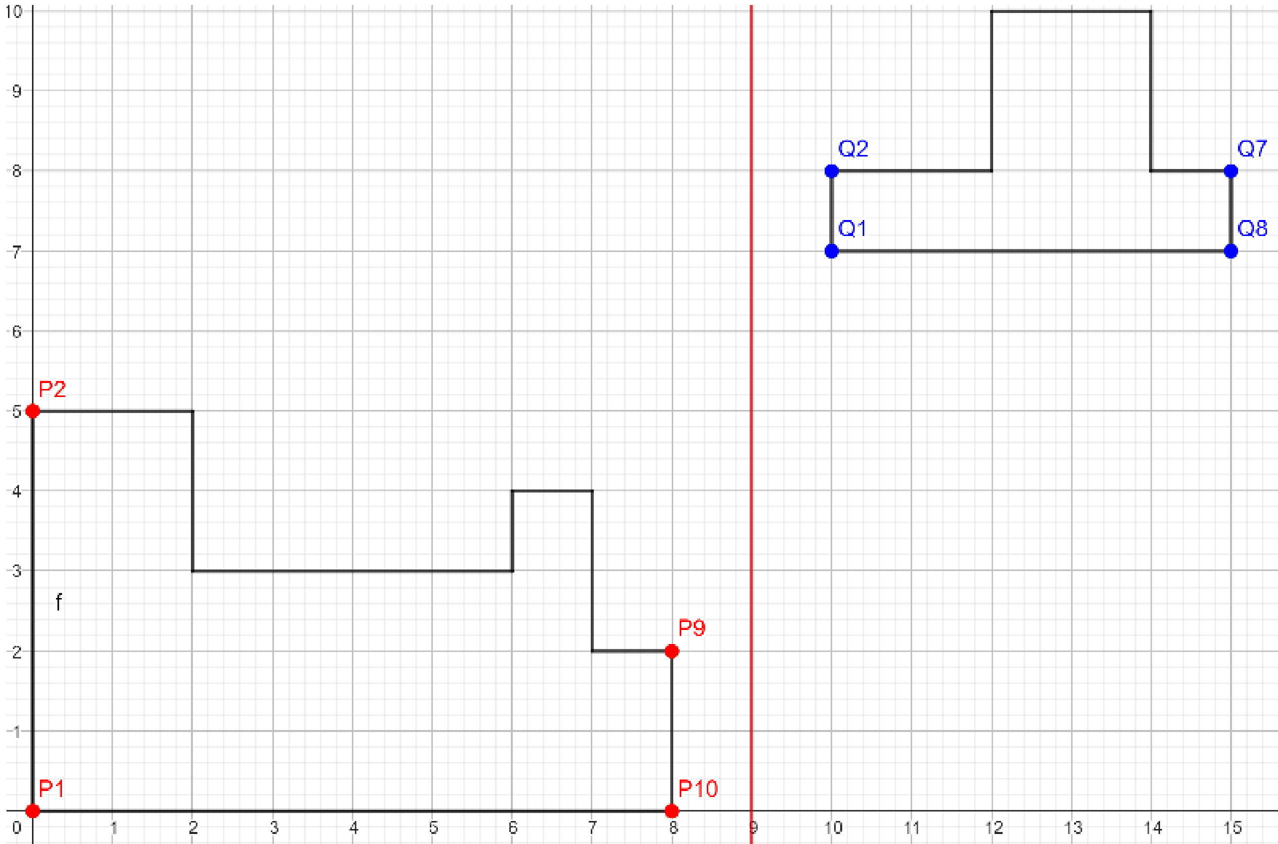




ΑΝΑΛΥΤΙΚΗ ΠΑΡΟΥΣΙΑΣΗ ΤΟΥ ΑΛΓΟΡΙΘΜΟΥ (11)

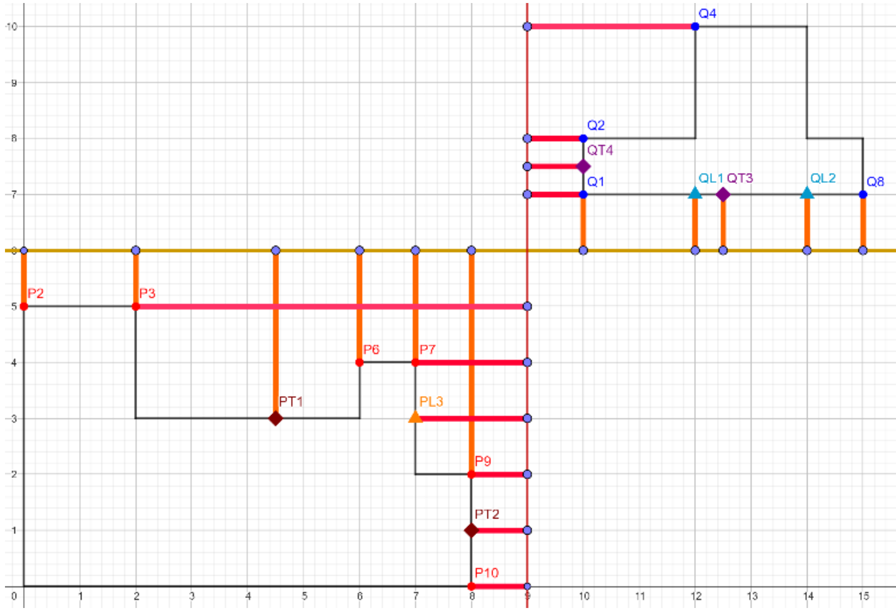
- <u>Διαδικασία Εύρεσης των **σετ $l_1(P)$, $l_1(Q)$, $l_2(P)$, $l_2(Q)$**</u>





ΑΝΑΛΥΤΙΚΗ ΠΑΡΟΥΣΙΑΣΗ ΤΟΥ ΑΛΓΟΡΙΘΜΟΥ (12)

➢ Υπολογισμός **$\mu(p1) = \min_{p \in l1(P)} \mu(p)$** και **$\mu(p1) = \min_{p \in l1(P)} \mu(p)$** και **$\mu(q1) = \min_{q \in l1(Q)} \mu(q)$** και **$\mu(q1) = \min_{q \in l1(Q)} \mu(q)$**



Γενικά: **$\mu(k)$** = $L_1(k, f(k)) + |x_k| + |y_k|$, όπου k κόμβος πολυγώνου,

- **$L_1(k, f(k))$:** η **απόσταση** του κόμβου **k** από τον **απώτατο γείτονα**
- **$|x_k|$:** η **απόσταση** του κόμβου **k** από την **l_2**
- **$|y_k|$:** η **απόσταση** του κόμβου **k** από την **l_1**

- **$\mu(p_1) = \min_{p \in l1(P)} \mu(p)$** : το σημείο του $l_1(P)$ με τη μικρότερη απόσταση $\mu(p)$
- **$\mu(p_2) = \min_{p \in l2(P)} \mu(p)$** : το σημείο του $l_2(P)$ με τη μικρότερη απόσταση $\mu(p)$
- **$\mu(q_1) = \min_{q \in l1(Q)} \mu(q)$** : το σημείο του $l_1(Q)$ με τη μικρότερη απόσταση $\mu(q)$
- **$\mu(q_2) = \min_{q \in l2(Q)} \mu(q)$** : το σημείο του $l_2(Q)$ με τη μικρότερη απόσταση $\mu(q)$

Βρίσκουμε όλες τις $\mu(k)$, όπου k όλα τα σημεία των σετ $l_1(P)$, $l_2(P)$, $l_1(Q)$, $l_2(Q)$,
και κρατάμε την μικρότερη τιμή $\mu(k)$, καθώς το αντίστοιχο σημείο για το κάθε σετ.

1) **Υπολογίζουμε** την **$L_1(k,f(k))$ απόσταση** του σημείου **k** από τον **απώτερο γείτονα** του **f(k)**, **προσθέτοντας** το **μήκος** των **ακμών** που τα συνδέουν.

2) **Υπολογίζουμε** την **$|x_k|$ απόσταση** του σημείου **k** από την **l_2** και την **$|y_k|$ απόσταση** του σημείου **k** από την **l_1**.

3) **Προσθέτουμε τις 3 τιμές** που βρήκαμε

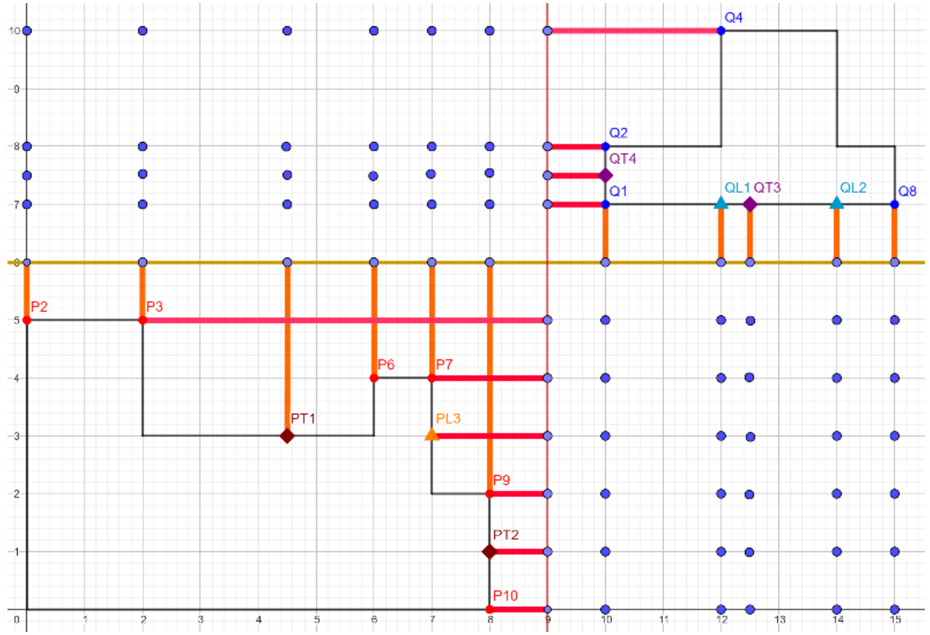
ΑΝΑΛΥΤΙΚΗ ΠΑΡΟΥΣΙΑΣΗ ΤΟΥ ΑΛΓΟΡΙΘΜΟΥ (13)

➢ <u>Υπολογισμός **$F_2(p_1, q_2)$** και **$F_2(p_2, q_1)$**</u>

<u>Αν</u> **$F_2(p_1, q_2) \leq F_2(p_2, q_1)$** <u>επέστρεψε</u> $\overline{p_1 r} + \overline{r q_2}$ <u>όπου</u> $r = (x_{p1}, y_{q2})$,

<u>αλλιώς επέστρεψε</u> $\overline{p_2 r} + \overline{r q_1}$ <u>όπου</u> $r = (x_{q1}, y_{p2})$

- **$F_2(p_1, q_2) = \mu(p_1) + \mu(q_2)$**
- **$F_2(p_2, q_1) = \mu(p_2) + \mu(q_1)$**



➢ <u>**Επιλογή Γέφυρας**</u>

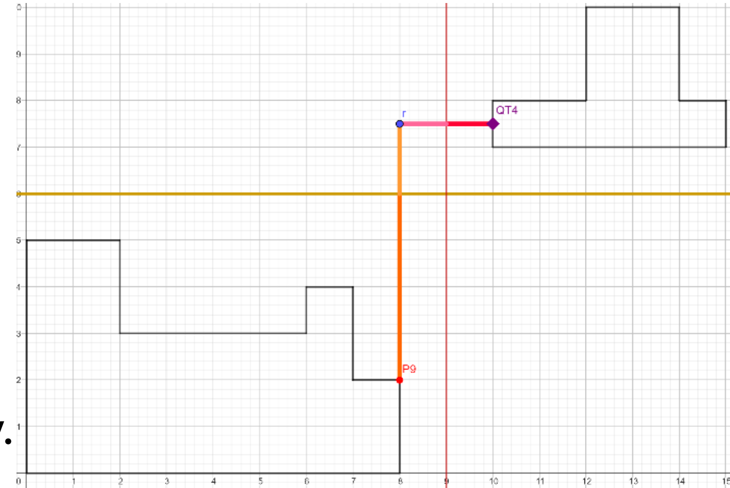
<u>Η διαδικασία εντοπισμού **$F_2(p_1, q_2) \leq F_2(p_2, q_1)$**, είναι</u>:

1) **Πρόσθεσε** τις **$\mu(p_1)$, $\mu(q_2)$** <u>και</u> τις **$\mu(p_2)$, $\mu(q_1)$**

2) Σύγκρινε τα 2 αθροίσματα, και επέστρεψε την κατάλληλη γέφυρα.

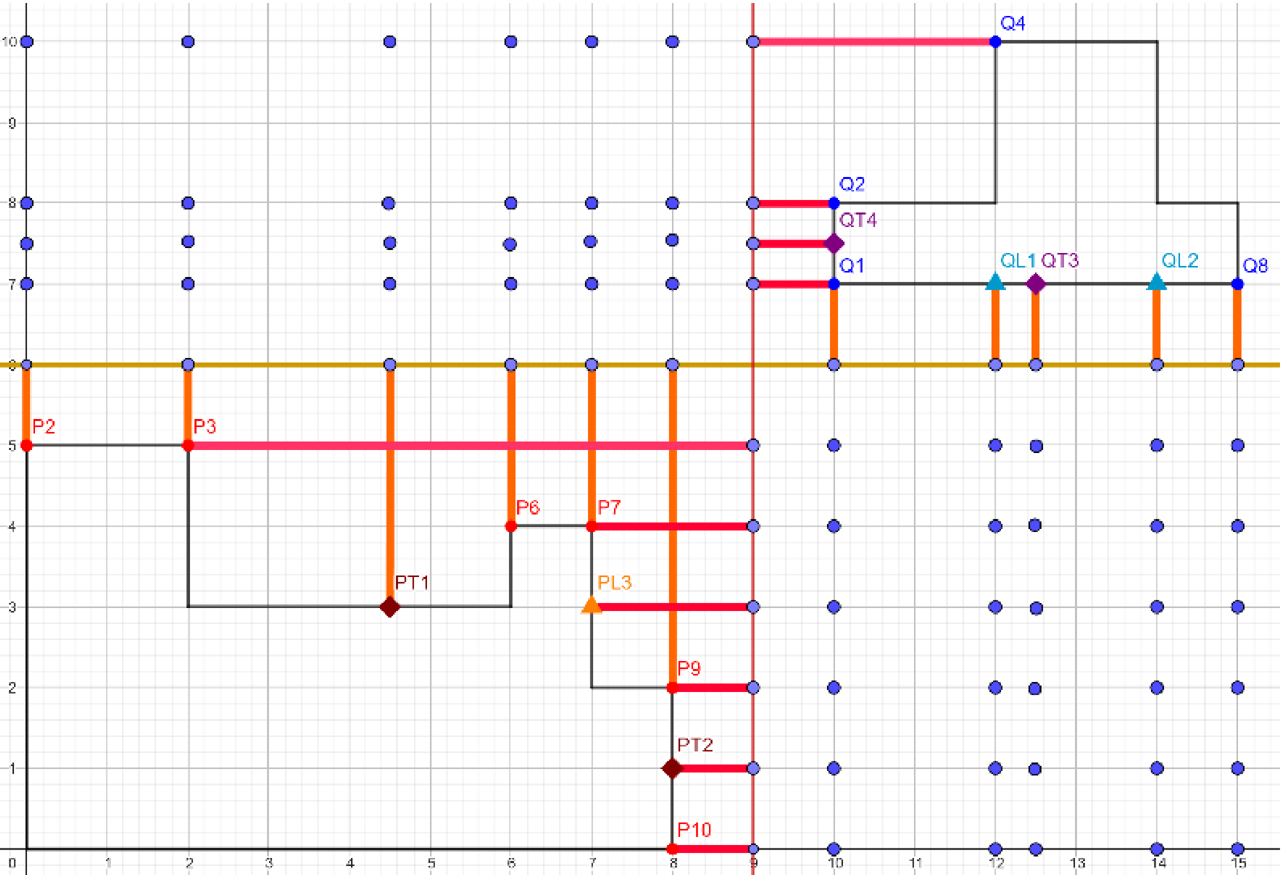
Την $\overline{\mathbf{p_1 r}} + \overline{\mathbf{r q_2}}$ **αν** το **άθροισμα $\mu(p_1)$, $\mu(q_2)$ μικρότερο**, <u>αλλιώς</u> την $\overline{\mathbf{p_2} r} + \overline{\mathbf{r q_1}}$.

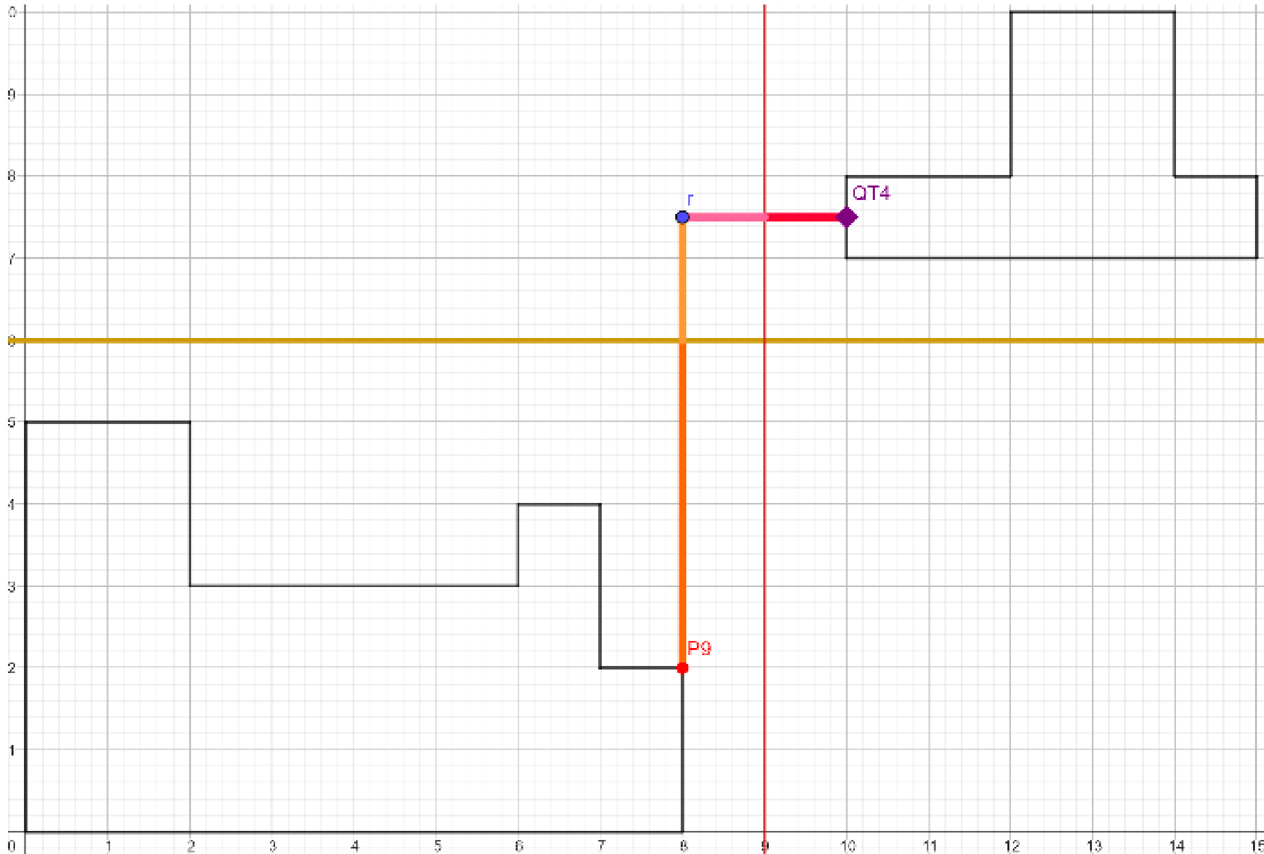
Η **γέφυρα** είναι το **αποτέλεσμα** της πράξης, από τα 3 σημεία που την αποτελούν.



ΑΝΑΛΥΤΙΚΗ ΠΑΡΟΥΣΙΑΣΗ ΤΟΥ ΑΛΓΟΡΙΘΜΟΥ (13)

➢ **Επιλογή Γέφυρας**





ΑΝΑΛΥΤΙΚΗ ΠΑΡΟΥΣΙΑΣΗ ΤΟΥ ΑΛΓΟΡΙΘΜΟΥ (14)

Είσοδος:	2 **πολύγωνα P** και **Q** στην Περίπτωση 1
Έξοδος:	Βέλτιστη Τύπου 1 ***γέφυρα*** $\overline{\mathbf{pq}}$

ΒΗΜΑΤΑ Αλγορίθμου 1 - Optimal_Type_1_bridge(P ,Q)

1	Θέτουμε:	Candidate_bridge_set = ∅.	
	Βρίσκουμε:		
2	T(P)	T(Q)	των Σημείων Μετάβασης
	$L_1P(P)$	$L_1P(Q)$	των L1-προβολή Σημείων
3	Candidate_bridge_set ∪ {pq}	∀ p ∈ {V(P) ∪ T(P) ∪ $L_1P(P)$}	q ∈ Sq ⊂ ∂Q, pq Τύπος 1 γέφυρα
4	Candidate_bridge_set ∪ {pq}	∀ q ∈ {V(Q) ∪ T(Q) ∪ $L_1P(Q)$}	p ∈ Sp ⊂ ∂P, pq Τύπος 1 γέφυρα
5	$F_1(p, q)$	∀ p,q ∈ Candidate_bridge_set	
6	$\overline{pq}$ ∈ Candidate_bridge_set	με min[$F_1(p, q)$]	

ΑΝΑΛΥΤΙΚΗ ΠΑΡΟΥΣΙΑΣΗ ΤΟΥ ΑΛΓΟΡΙΘΜΟΥ (15)

➢ Εύρεση των **σετ T(P), T(Q) των Σημείων Μετάβασης**

Διευκρινίζουμε κάποιους από τους όρους που αναφέρουμε:

- **T(P)** = $\{ t \mid t$ μεταβατικό σημείο του $B(v_i) \neq \emptyset$, αν $B(v_i) = \emptyset$, $0 \leq i \leq |V(P)| - 1 \}$

- **$B(v_i)$** = $\{ x \mid x \in \partial P$ και $f(x) = v_i \}$

  Το σύνολο σημείων στο όριο του πολυγώνου, με L_1-μακρύτεροι γείτονές v_i.

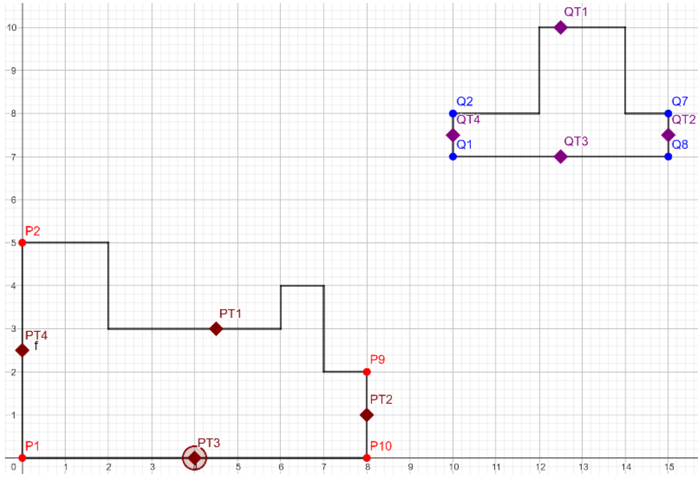
- **f(x)** : L_1-απώτατος γείτονας του x => κόμβος με μέγιστη L_1-απόσταση από x

- **L_1-απόσταση** : το μήκος συντομότερου ευθύγραμμου μονοπατιού, που συνδέει 2 σημεία του πολυγώνου.

ΑΝΑΛΥΤΙΚΗ ΠΑΡΟΥΣΙΑΣΗ ΤΟΥ ΑΛΓΟΡΙΘΜΟΥ (16)

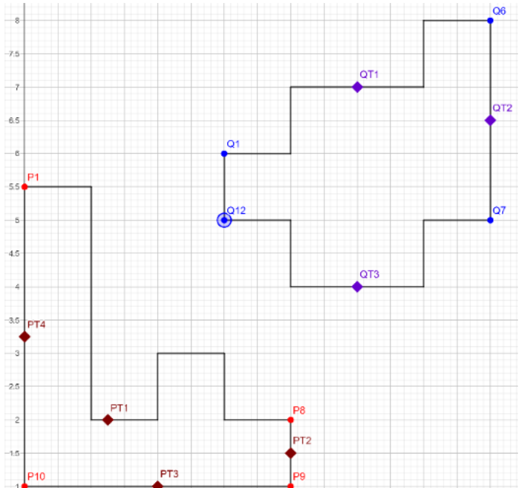
➢ Διαδικασία Εύρεσης των **σετ T(P), T(Q) των Σημείων Μετάβασης**

1) Επιλέγουμε τους **4 πιο ακρινούς κόμβους** του πολυγώνου,
με **βάση** την **x** συντεταγμένη τους (2 αριστερά, 2 δεξιά)
(επιλέγουμε το πρώτο κόμβο με την μεγαλύτερη συντεταγμένη x,
εξετάζοντας δεξιόστροφα).



2) Υπολογίζουμε την **συνολική απόσταση** για τα εξής **ζεύγη** ακρινών κόμβων:

i. πάνω_αριστερά - πάνω_δεξιά,
ii. πάνω_δεξιά - κάτω_δεξιά,
iii. κάτω_δεξιά - κάτω_αριστερά,
iv. κάτω_αριστερά - πάνω_αριστερά.



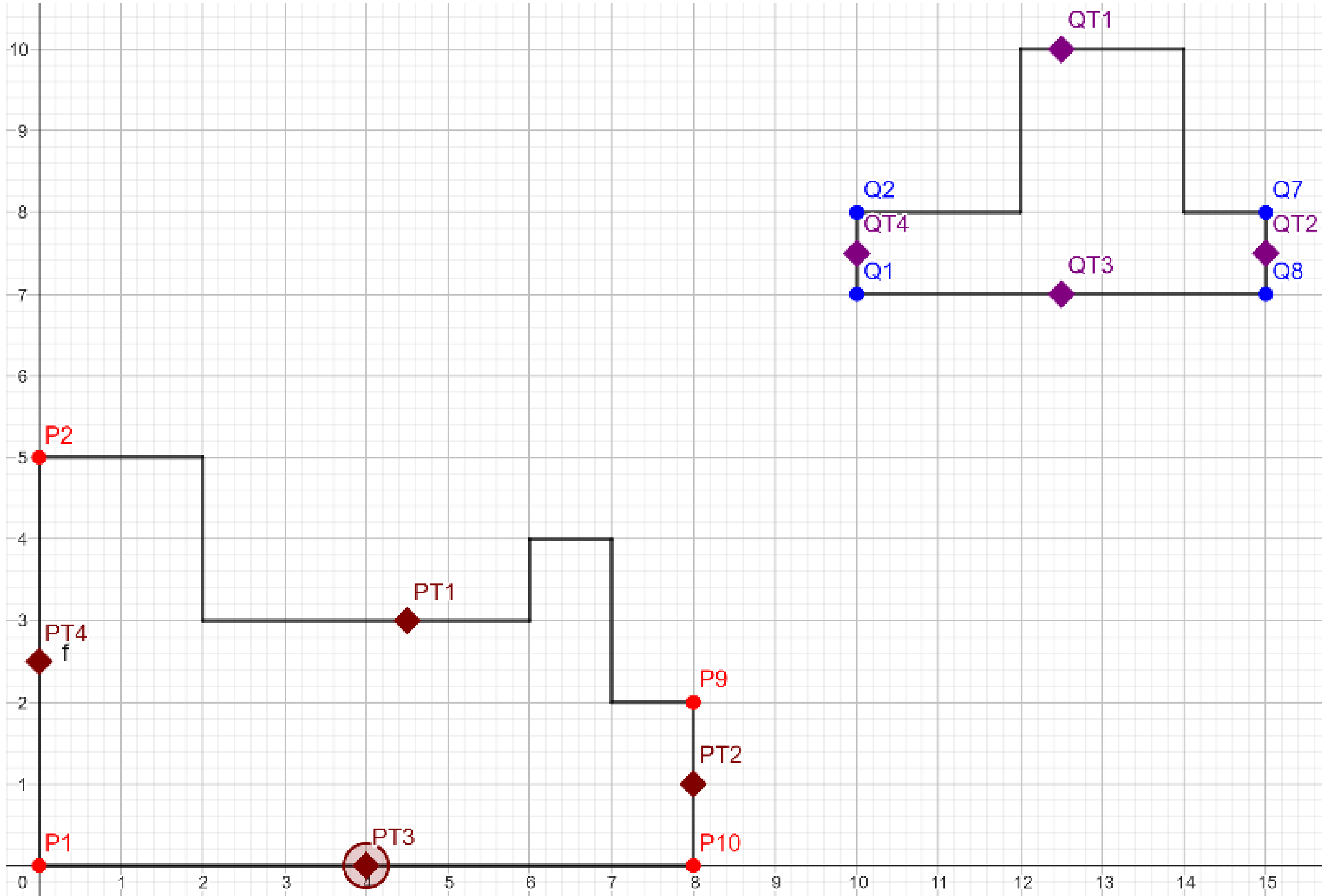
Για κάθε ζεύγος, για κάθε ενδιάμεση ακμή στο μονοπάτι τους,
προσθέτουμε σε **καταμετρητή**, την **απόλυτη διαφορά** των τιμών,
των **μη κοινών συντεταγμένων** των κόμβων που την αποτελούν.

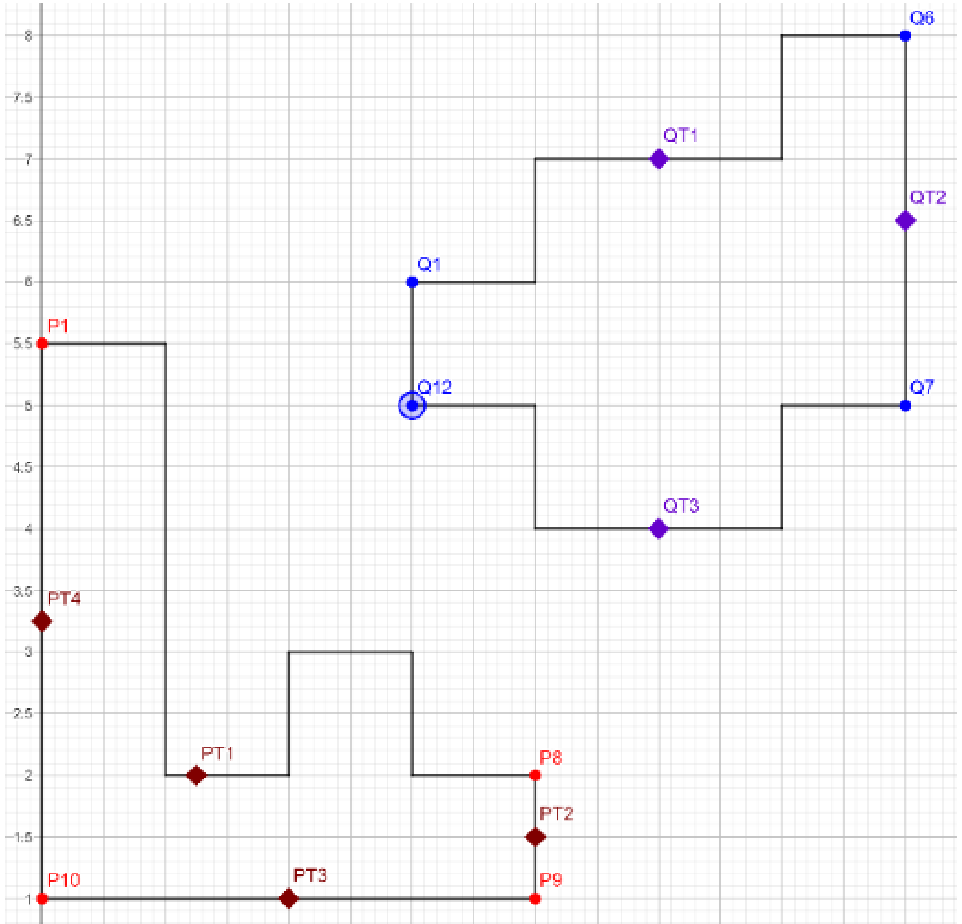
3) Υπολογίζουμε **για κάθε ζεύγος**, το **σημείο**, το οποίο ανήκει σε κάποια
από τις ακμές του μονοπατιού, που βρίσκεται στη **μισή απόσταση**,
διανύοντας τις ακμές, από την συνολική που υπολογίσαμε.

Τα 4 σημεία που θα προκύψουν, είναι τα **4 Σημεία Μετάβασης**.

ΑΝΑΛΥΤΙΚΗ ΠΑΡΟΥΣΙΑΣΗ ΤΟΥ ΑΛΓΟΡΙΘΜΟΥ (17)

- Εύρεση των **σετ T(P), T(Q) των Σημείων Μετάβασης**





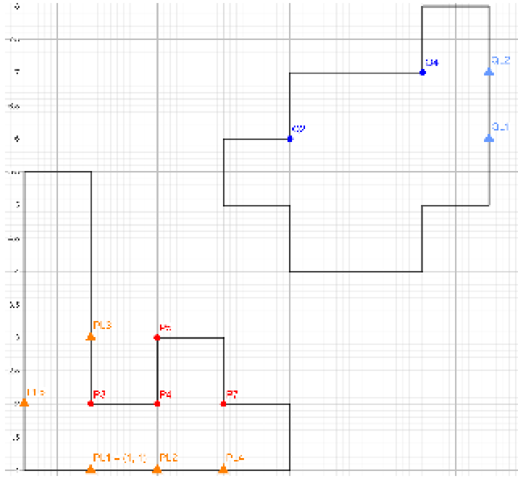
ΑΝΑΛΥΤΙΚΗ ΠΑΡΟΥΣΙΑΣΗ ΤΟΥ ΑΛΓΟΡΙΘΜΟΥ (18)

➢ Διαδικασία Εύρεσης των **σετ L_1P(P), L_1P(P) των L_1-προβολή Σημείων**

L_1 (v_i, t) = L_1 (v_i, e) : το **σημείο t προβάλλεται από** τον **κόμβο v_i στην ακμή e**.

Τα L_1-προβολή Σημεία είναι οι προβολές των κόμβων του πολυγώνου επάνω στις ακμές.

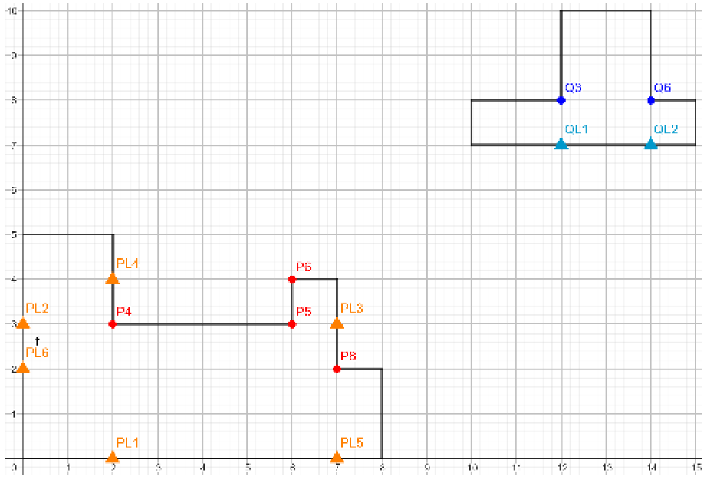
Δηλαδή, είναι σημεία, που δεν ταυτίζεται η θέση τους με άλλους κόμβους του πολυγώνου, ανήκουν σε μια ακμή, και έχουν κοινή τιμή μόνο στον έναν άξονα, με τον αντίστοιχο κόμβο τους. Προκειμένου να εντοπίσουμε το Σημείο Προβολής, εντοπίζουμε όλους τους διαδοχικούς κόμβους με διαφορετική τιμή από τον κόμβο που συγκρίνουμε στην μια συντεταγμένη, και με τιμή της άλλης συντεταγμένης, για τον έναν μεγαλύτερη, και για τον άλλον μικρότερη, ώστε να είναι ανάμεσα τους.



1) **Επιλέγουμε διαδοχικά** όλους τους **κόμβους** του πολυγώνου.

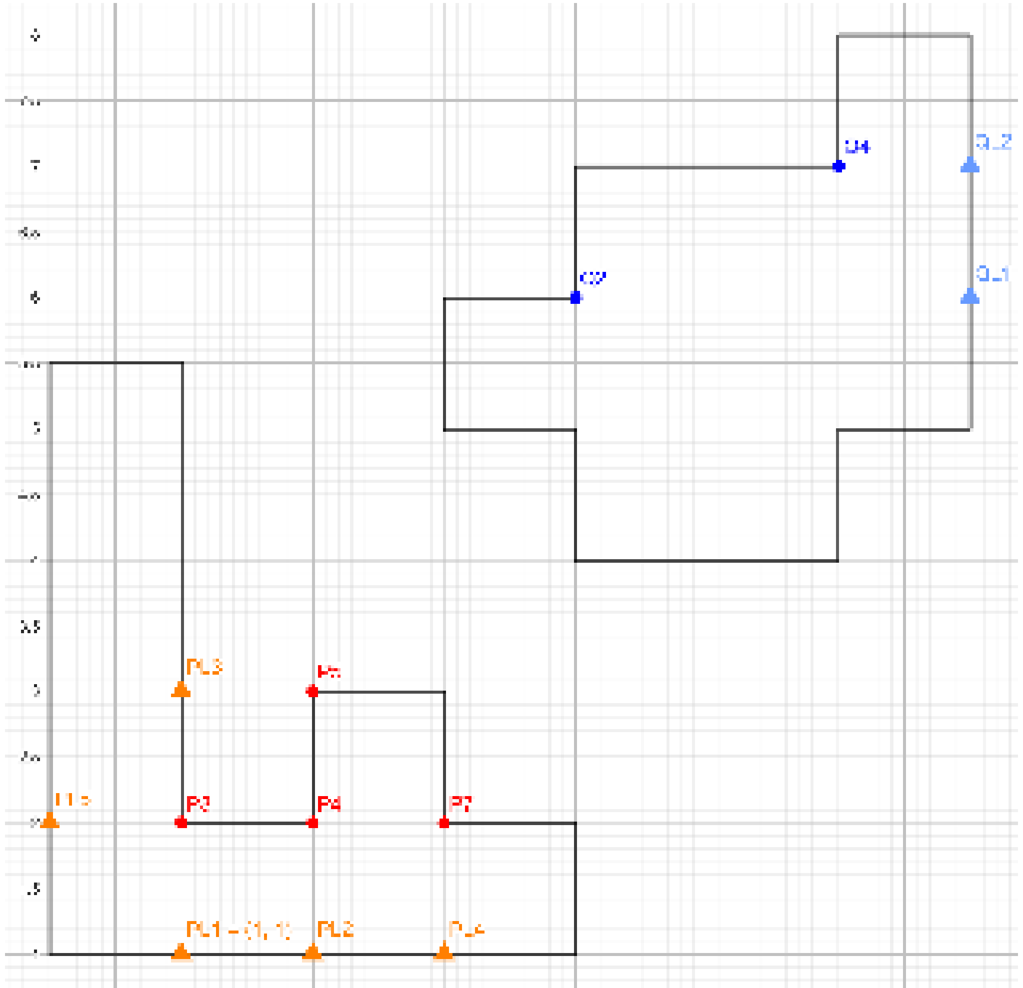
2) Τους **συγκρίνουμε** με όλα τα υπόλοιπα **ζεύγη** κόμβων, που δεν περιλαμβάνουν τον επιλεχθέντα, και **αποθηκεύουμε** τα **ζεύγη** που η **κοινή συντεταγμένη** τους είναι **διαφορετική του επιλεχθέντα**, και **η άλλη** τους είναι η **μια μεγαλύτερη** και η **άλλη μικρότερη** από του επιλεχθέντα, σε λίστα.

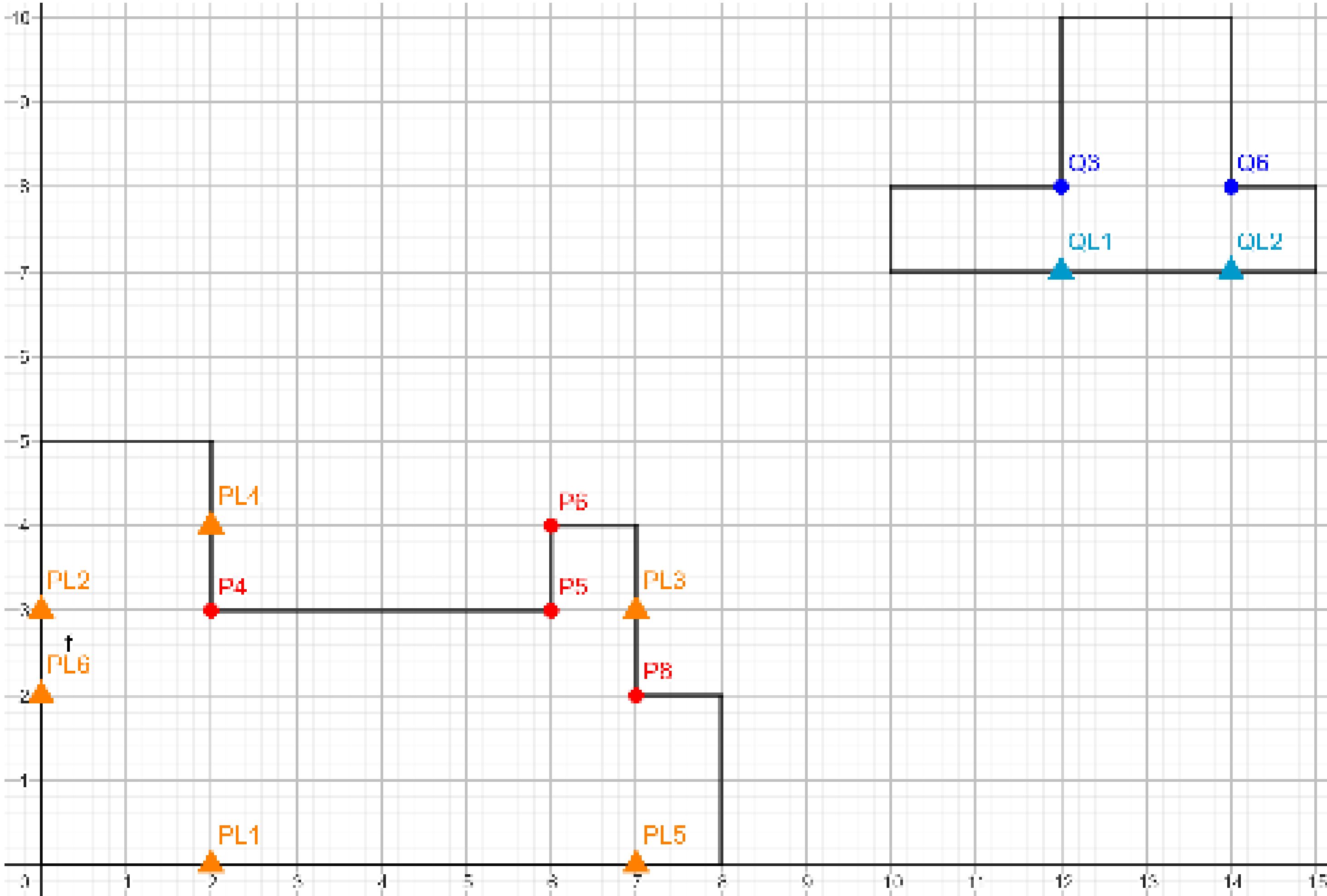
Στο τέλος η λίστα περιλαμβάνει όλα τα L_1-προβολή Σημεία των κόμβων του πολυγώνου.



ΑΝΑΛΥΤΙΚΗ ΠΑΡΟΥΣΙΑΣΗ ΤΟΥ ΑΛΓΟΡΙΘΜΟΥ (19)

- Εύρεση των **σετ $L_1P(P)$, $L_1P(P)$ των L_1-προβολή Σημείων**

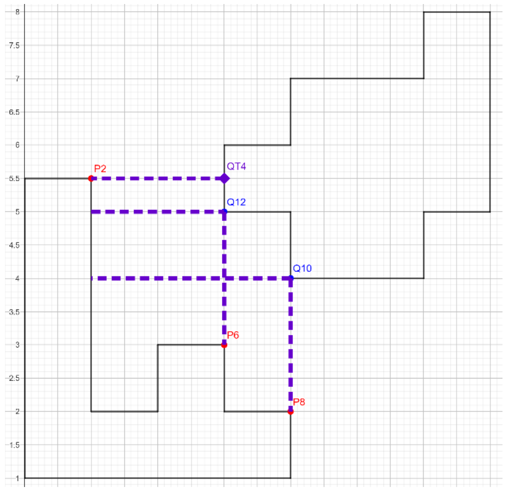




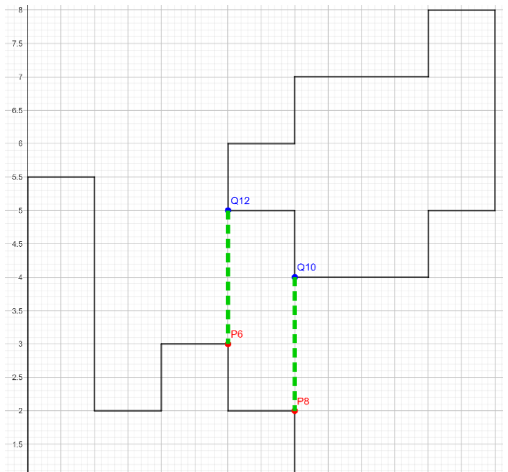
ΑΝΑΛΥΤΙΚΗ ΠΑΡΟΥΣΙΑΣΗ ΤΟΥ ΑΛΓΟΡΙΘΜΟΥ (20)

➢ Εύρεση των σημείων **(p, q) διασύνδεσης γέφυρας** P, Q

Συγκρίνουμε κάθε σημείο **p ∈ {V(P) ∪ T(P) ∪ L1P(P)}**
με τις **ακμές** του **Q**, προκειμένου να διαπιστώσουμε
εάν υπάρχει ακμή που περιέχει **σημείο προβολής** του **p**,
δηλαδή εάν μπορεί να συνδεθεί το p με το σημείο που αναζητούμε,
με 1 κάθετο ή 1 οριζόντιο ευθύγραμμο τμήμα, που δεν εισέρχεται
στις εσωτερικές περιοχές των πολυγώνων.



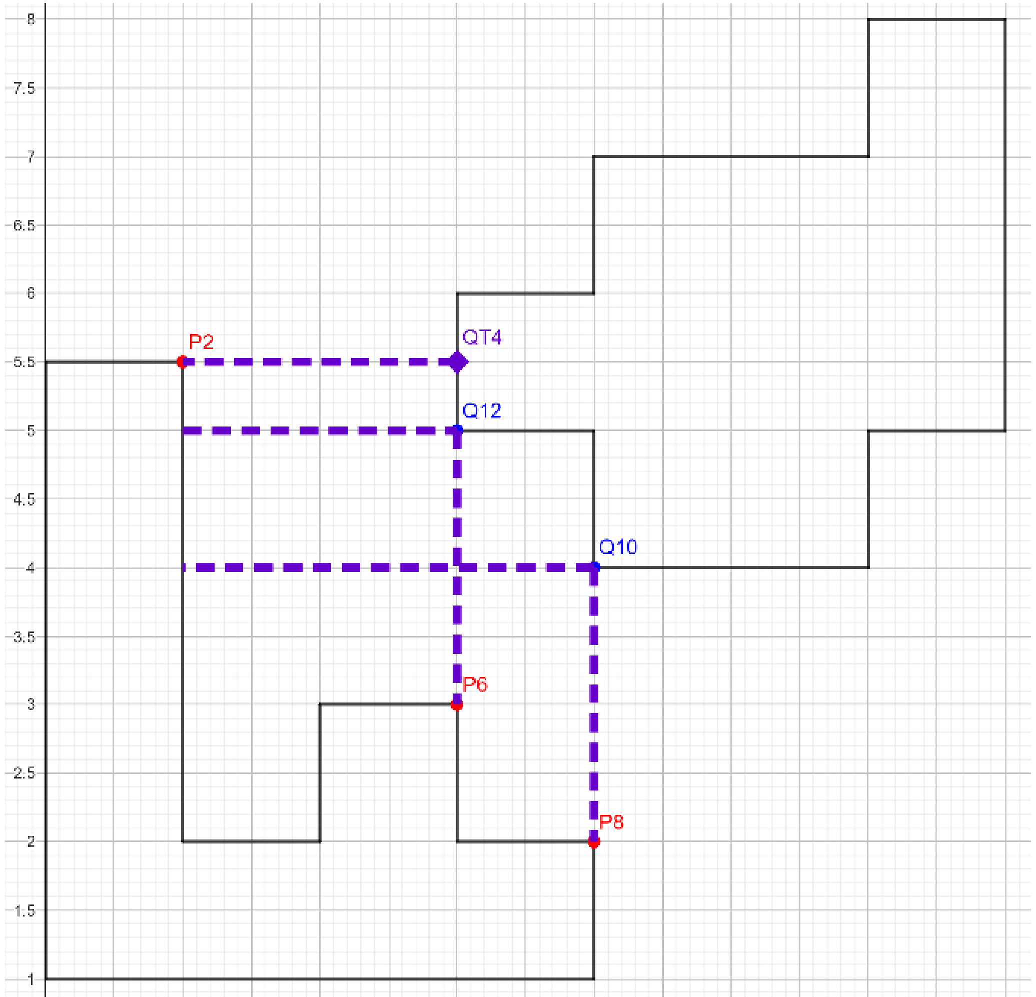
Αντίστοιχα για κάθε σημείο **q ∈ {V(Q) ∪ T(Q) ∪ L_1P(Q)}** και πολύγωνο **P**.

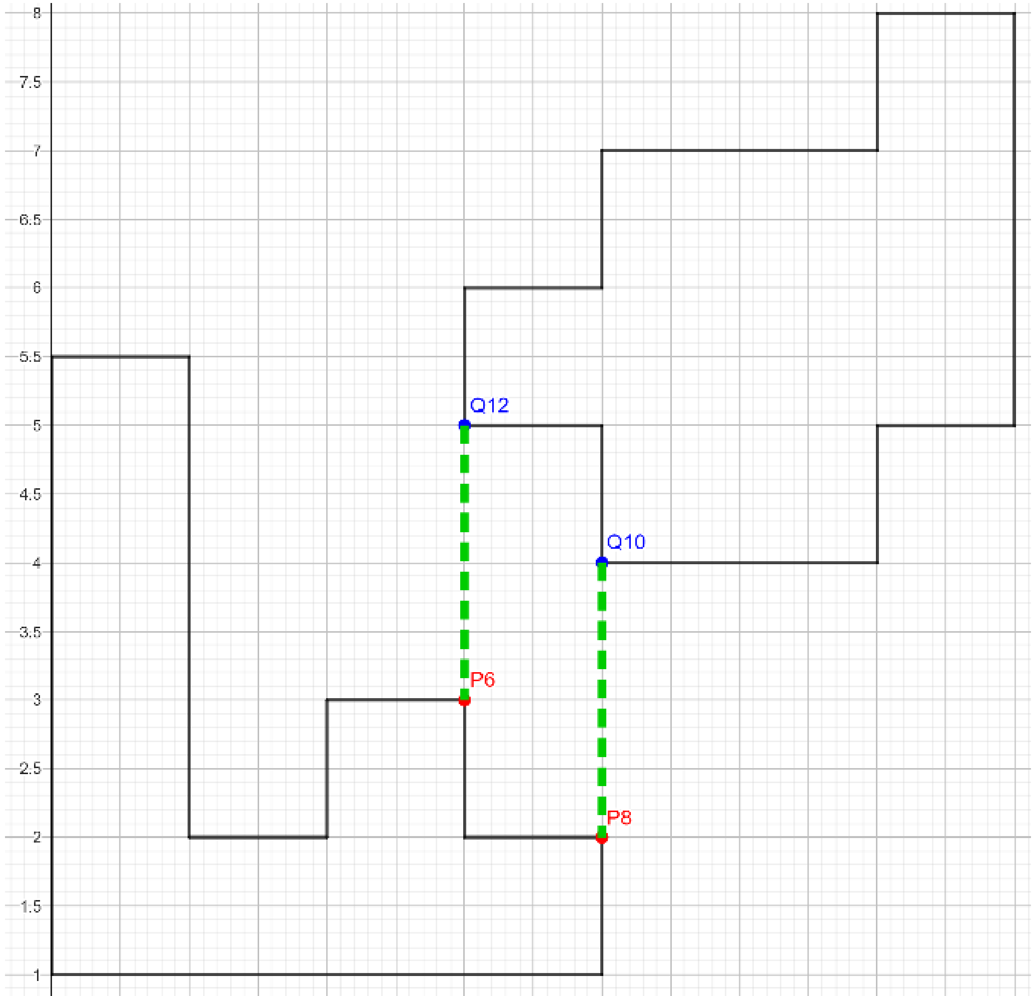


Προσθέτουμε στη **λίστα Candidate_bridge_set** όλα τα σημεία που
ικανοποιούν την παραπάνω συνθήκη, **ως ζεύγη** με το **σημείο του
άλλου πολυγώνου** καθώς σχηματίζουν επιτυχώς γέφυρα, και
μια εξ αυτών θα επιλεχθεί σε επόμενο βήμα ως η βέλτιστη.

ΑΝΑΛΥΤΙΚΗ ΠΑΡΟΥΣΙΑΣΗ ΤΟΥ ΑΛΓΟΡΙΘΜΟΥ (21)

- Εύρεση των σημείων **(p, q) διασύνδεσης γέφυρας** P, Q





ΑΝΑΛΥΤΙΚΗ ΠΑΡΟΥΣΙΑΣΗ ΤΟΥ ΑΛΓΟΡΙΘΜΟΥ (22)

➢ Υπολογισμός F_1(p, q) για γέφυρα $\overline{pq}$ ∊ Candidate_bridge_set

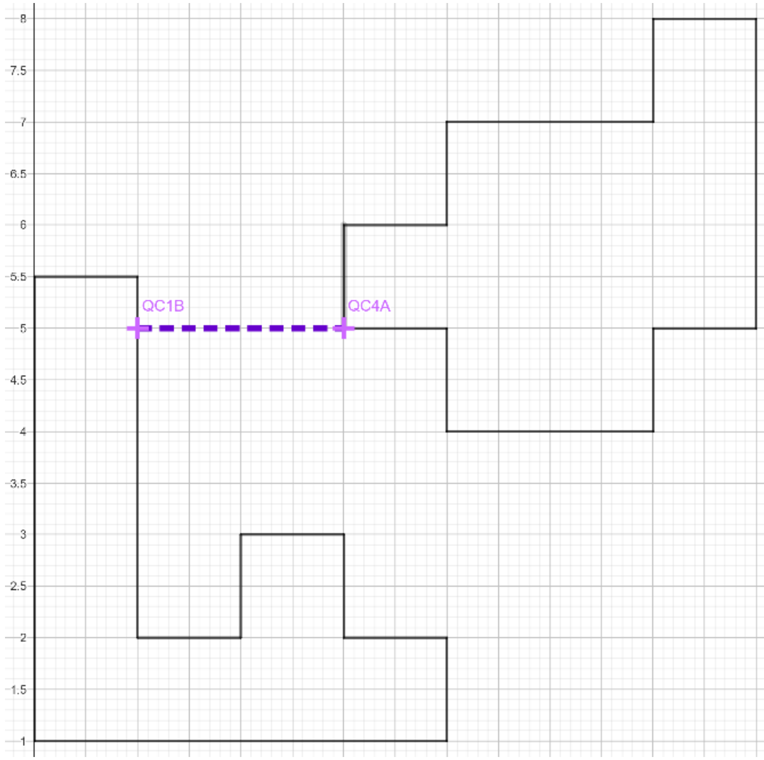
Ισχύει: $F_1(p, q) = L_1(p, f(p)) + L_1(q, f(q)) + |x_{distance}| + |y_{distance}|$.

- **L_1(k, f(k))** : η απόσταση του κόμβου k από τον απώτατο γείτονα του
- **$|x_{distance}|$** : η απόσταση του κόμβου p από το σημείο q στις x συντεταγμένες τους
- **$|y_{distance}|$** : η απόσταση του κόμβου p από το σημείο q στις y συντεταγμένες τους

1) **Υπολογίζουμε** τις αποστάσεις **L_1(p,f(p)),** του p από τον f(p),
   και **L_1(q,f(q)),** του q από τον f(q),
   προσθέτοντας το μήκος των ακμών που τα συνδέουν.

2) **Υπολογίζουμε** τις **| $x_{distance}$ |** και **| $y_{distance}$ | αποστάσεις** των **p** και **q**.

3) **Προσθέτουμε** τις 4 τιμές που βρήκαμε.



➢ Επιλογή Γέφυρας

Εντοπίζουμε την **ελάχιστη** τιμή **F_1(p, q)** και **επιστρέφουμε** ως **βέλτιστη γέφυρα** το **ζεύγος p** και **q**.

ΜΕΡΟΣ 4:

ΥΛΟΠΟΙΗΣΗ ΤΟΥ ΑΛΓΟΡΙΘΜΟΥ ΣΕ ΚΩΔΙΚΑ

ΥΛΟΠΟΙΗΣΗ ΤΟΥ ΑΛΓΟΡΙΘΜΟΥ ΣΕ ΚΩΔΙΚΑ (1)

➢ **Είσοδος – Έξοδος**

Ο κώδικας που υλοποιήσαμε, δέχεται ως είσοδο 2 πολύγωνα, με τη μορφή διαδοχικών σημείων, και επιστρέφει ως έξοδο τα σημεία όπου οι 2 ευθείες (ή η μια ευθεία) που αποτελούν την γέφυρα ξεκινούν (στις ακμές των πολυγώνων) και εκεί που τέμνονται.

Ορίζουμε τα πολύγωνα ως poly_P, poly_Q και τα δίνουμε ως ορίσματα στην find_Bridge(poly_P, poly_Q).

```
poly_P = [(0,5.5),(1,5.5),(1,2),(2,2),(2,3),(3,3),(3,2),(4,2),(4,1),(0,1)]
poly_Q = [(3,6),(4,6),(4,7),(6,7),(6,8),(7,8),(7,5),(6,5),(6,4),(4,4),(4,5),(3,5)]
find_Bridge(poly_P, poly_Q)
```

➢ Μέθοδοι για την **Αναγνώριση Περίπτωσης**

```
1. find_Bridge(poly_P, poly_Q)
```

```
2. check_polygon(poly)
```

```
3. which_Case(poly_P, poly_Q)
```

ΥΛΟΠΟΙΗΣΗ ΤΟΥ ΑΛΓΟΡΙΘΜΟΥ ΣΕ ΚΩΔΙΚΑ (2)

- <u>Μέθοδοι για τον **Algorithm Optimal_Type_2_bridge(P,Q)**</u>

```
1.  optimal_type_2_bridge(poly_P, poly_Q)
2.  find_transition_points(polygon)
3.  find_farthest_left_vertices(polygon)
4.  find_farthest_right_vertices(polygon)
5.  find_total_distance(polygon, start_vertex, end_vertex)
6.  find_point_for_half_distance(polygon, start_point, total_distance)
7.  get_L1_projection_points(poly)
8.  find_horizontal_line(poly_P, poly_Q)
9.  find_vertical_line(poly_P, poly_Q)
10. find_l1_set(poly, transition_points, L1_projection_points, horizontal_line)
11. find_l2_set(poly, transition_points, L1_projection_points, vertical_line)
12. generate_candidates(point_set, neighbor_set, horizontal_line, vertical_line)
13. find_most_distant_neighbor_path(point, neighbor_set)
14. find_F2(mu_p1, mu_p2, mu_q1, mu_q2)
15. euclidean_distance(a, b)
```

ΥΛΟΠΟΙΗΣΗ ΤΟΥ ΑΛΓΟΡΙΘΜΟΥ ΣΕ ΚΩΔΙΚΑ (3)

➢ Μέθοδοι για **Algorithm Optimal_Type_1_bridge(P,Q)**

```
1.  optimal_type_1_bridge(poly_P, poly_Q)
2.  find_transition_points(polygon)
3. get_L1_projection_points(poly)
4. find_add_connectable_points(poly_1, poly_2, transition_points, L1_projection_points)
5. generate_candidates(point_set, neighbor_set, horizontal_line, vertical_line)
6. find_F1(points_pairs_set, neighbor_set)
7. remove_duplicates(Bridge_Alg1)
8. sort_list(list)
9. find_F1_for_list(poly, connectable_points)
10.F1_min(F1_list_P, F1_list_Q, connectable_points_P, connectable_points_Q)
```

ΥΛΟΠΟΙΗΣΗ ΤΟΥ ΑΛΓΟΡΙΘΜΟΥ ΣΕ ΚΩΔΙΚΑ (4)

- **def find_Bridge (poly_P, poly_Q)**

Καλεί τις μεθόδους που υλοποιούν τους αλγορίθμους.

Αρχικά καλεί την check_polygon(poly) για τους ελέγχους λειτουργικότητας, και στη συνέχεια καλεί την which_Case(poly_P, poly_Q) ώστε να αναγνωρίσει σε ποια περίπτωση ανήκουν τα πολύγωνα.

Ανάλογα με την κατάσταση που έλαβε,

καλεί την optimal_type_1_bridge(poly_P, poly_Q), ή την optimal_type_2_bridge(poly_P, poly_Q),

και τυπώνει (print) το αποτέλεσμα που θα του επιστραφεί, στην εκάστοτε περίπτωση.

- **def check_polygon (poly)**

Ελέγχει αν το πολύγωνο αποτελείται από τουλάχιστον 4 κόμβους (4≤len(poly)), και

αν διαδοχικοί κόμβοι σχηματίζουν ακμή που αντιστοιχεί σε ευθύγραμμο κάθετο ή οριζόντιο γραμμικό τμήμα,

(μόνο 1 συντεταγμένη με ίδια τιμή) (if (poly[i][0] == poly[i+1][0]) or (poly[i][1] == poly[i+1][1])).

- **def which_Case (poly_P, poly_Q)**

Εντοπίζει τους πιο ακριανούς κόμβους, του κάθε πολυγώνου, με βάση τις x, y συντεταγμένες, και εφόσον διαπιστώσει ότι τουλάχιστον 1 συντεταγμένη του ενός, ανήκει στο διάστημα των συντεταγμένων ακριανών κόμβων με κοινή τη μια τους συντεταγμένη, του άλλου πολυγώνου, ότι είναι στην Περίπτωση 1.
Ειδάλλως, θεωρείται ότι είναι στην Περίπτωση 2.

```python
def find_Bridge(poly_P, poly_Q):

    check_polygon(poly_P)
    check_polygon(poly_Q)
    # recognise between Cases 1 and 2
    case = which_Case(poly_P, poly_Q)

    # Case 1: the polygons are connected by 1 rectilinear line segment
    if (case == 1):
        case_1_result = optimal_type_1_bridge(poly_P, poly_Q)
        # bridge price = euclidean(p,q)
        case_1_bridge_price = case_1_result[0]
        # bridge connects points p and q
        case_1_p = case_1_result[1]
        case_1_q = case_1_result[2]
        print("CASE 1:")
        print("bridge price:", case_1_bridge_price)
        print("p:", case_1_p)
        print("q:", case_1_q)
        print("\n")

    # Case 2: the polygons are connected by 2 rectilinear line segments
    if (case == 2):
        case_2_result = optimal_type_2_bridge(poly_P, poly_Q)
        # bridge price = euclidean(p,r) + euclidean(r,q)
        case_2_bridge_price = case_2_result[0]
        # bridge connects points p,r and r,q
        case_2_p = case_2_result[1]
        case_2_q = case_2_result[2]
        case_2_r = case_2_result[3]
        print("CASE 2:")
        print("bridge price:", case_2_bridge_price)
        print("p:", case_2_p)
        print("q:", case_2_q)
        print("r:", case_2_r)
        print("\n")
```

```python
def check_polygon(poly):

    # Check if the polygon has at least three vertices

    if len(poly) < 3:
        print("Error: Polygon must have at least three vertices.")
        return

    # Iterate through the vertices
    for i in range(len(poly)-1):
        # Check if x or y coordinates are equal, but not both
        if (poly[i][0] == poly[i+1][0]) or (poly[i][1] == poly[i+1][1]):
            if not ((poly[i][0] == poly[i+1][0]) and (poly[i][1] == poly[i+1][1])):
                continue
        print(f"Error: Vertices {poly[i]} and {poly[i+1]} do not satisfy the condition.")
        return

    # Check the condition for the first and last vertices
    if (poly[0][0] == poly[-1][0]) or (poly[0][1] == poly[-1][1]):
        if not ((poly[0][0] == poly[-1][0]) or (poly[0][1] == poly[-1][1])):
            print(f"Error: Vertices {poly[0]} and {poly[-1]} do not satisfy the condition.")
            return
```

```python
def which_Case(poly_P, poly_Q):

    case = 1
    min_y_P = min(p[1] for p in poly_P)
    max_y_P = max(p[1] for p in poly_P)
    min_y_Q = min(p[1] for p in poly_Q)
    max_y_Q = max(p[1] for p in poly_Q)
    between_y = 0
    min_x_P = min(p[0] for p in poly_P)
    max_x_P = max(p[0] for p in poly_P)
    min_x_Q = min(p[0] for p in poly_Q)
    max_x_Q = max(p[0] for p in poly_Q)
    between_x = 0
    condition_1_y = (min_y_P <= min_y_Q <= max_y_P)
    condition_2_y = (min_y_P <= max_y_Q <= max_y_P)
    condition_3_y = (min_y_Q <= min_y_P <= max_y_Q)
    condition_4_y = (min_y_Q <= max_y_P <= max_y_Q)

    if (condition_1_y) or (condition_2_y) or (condition_3_y) or (condition_4_y):
        between_y = 1
    condition_1_x = (min_x_P <= min_x_Q <= max_x_P)
    condition_2_x = (min_x_P <= max_x_Q <= max_x_P)
    condition_3_x = (min_x_Q <= min_x_P <= max_x_Q)
    condition_4_x = (min_x_Q <= max_x_P <= max_x_Q)
    if (condition_1_x) or (condition_2_x) or (condition_3_x) or (condition_4_x):
        between_x = 1
    if (between_y) == 0 and (between_x) == 0:
        case = 2

    return case
```

ΥΛΟΠΟΙΗΣΗ ΤΟΥ ΑΛΓΟΡΙΘΜΟΥ ΣΕ ΚΩΔΙΚΑ (5)

- **def optimal_type_2_bridge (poly_P, poly_Q)**

Η παρούσα μέθοδος καλεί όλες τις μεθόδους για τα βήματα του αλγορίθμου, διαδοχικά, δίνοντας ως κάποια από τα ορίσματα τους, κάποια από τα αποτελέσματα που επέστρεψαν προηγούμενες μέθοδοι. Στο τέλος επιστρέφει την γέφυρα, ως την τιμή της όπως προκύπτει από τον τύπο, και τα 3 σημεία από τα οποία υπολογίζεται.

- **def find_transition_points (polygon)**

Αφότου καλέσει την find_farthest_left_vertices ώστε να αναγνωρίσει τους 4 πιο ακρινούς κόμβους του πολυγώνου, καλεί την find_total_distance για να υπολογίσει την μέγιστη απόσταση, και στην συνέχεια την find_point_for_half_distance για να υπολογίσει το Σημείο Μετάβασης.

- **def find_farthest_left_vertices (polygon)**

Αναγνωρίζει αρχικά τους αριστερότερους κόμβους, με βάση την x συντεταγμένη τους, και στη συνέχεια αναγνωρίζει με βάση την y συντεταγμένη τους ποιος είναι πάνω.

- **def find_farthest_right_vertices (polygon)**

Αναγνωρίζει αρχικά τους δεξιότερους κόμβους, με βάση την x συντεταγμένη τους, και στη συνέχεια αναγνωρίζει με βάση την y συντεταγμένη τους ποιος είναι πάνω.

```python
def optimal_type_2_bridge(poly_P, poly_Q):
    # poly_P and poly_Q are lists of points representing the polygons
    # STEP 1
    # 1.1 : Find Sets T(P), T(Q) of Transition Points
    transition_points_P = find_transition_points(poly_P)
    transition_points_Q = find_transition_points(poly_Q)
    # 1.2 : Find Sets L1P(P), L1P(Q) of L1-Projection Points
    L1_projection_points_P = get_L1_projection_points(poly_P)
    L1_projection_points_Q = get_L1_projection_points(poly_Q)
    # STEP 2
    # 2.1 : Find lines that are between the polygons
    horizontal_line = find_horizontal_line(poly_P, poly_Q)
    vertical_line = find_vertical_line(poly_P, poly_Q)
    # 2.2 : Find l1(P), l2(P), l1(Q), l2(Q)
    # the vertices, transition and L1-projection points of the polygon (P/Q), visible by
    # l1_set_ : the horizontal line , l2_set_ : the vertical line
    l1_set_P = find_l1_set(poly_P, transition_points_P, L1_projection_points_P, horizontal_line)
    l2_set_P = find_l2_set(poly_P, transition_points_P, L1_projection_points_P, vertical_line)
    l1_set_Q = find_l1_set(poly_Q, transition_points_Q, L1_projection_points_Q, horizontal_line)
    l2_set_Q = find_l2_set(poly_Q, transition_points_Q, L1_projection_points_Q, vertical_line)
    # STEP 3 :
    # m(p1) = min[(p belongs in I1) l1(P)] m(p),
    # m(p2) = min[(p belongs in l2) l2(P)] m(p),
    # m(q1) = min[(q belongs in I1) l1(Q)] m(q),
    # m(q2) = min[(q belongs in l2) l2(Q)] m(q),
    # m(x) is the mu_x_min_price, but we need the point for STEP 4
    mu_p1_min = generate_candidates(l1_set_P, poly_P, horizontal_line, vertical_line)
    mu_p2_min = generate_candidates(l2_set_P, poly_P, horizontal_line, vertical_line)
    mu_q1_min = generate_candidates(l1_set_Q, poly_Q, horizontal_line, vertical_line)
    mu_q2_min = generate_candidates(l2_set_Q, poly_Q, horizontal_line, vertical_line)
    # STEP 4
    # Find F2(p1,q2) = m(p1) + m(q2), F2(p2,q1) = m(p2) + m(q1)
    result = find_F2(mu_p1_min, mu_p2_min, mu_q1_min, mu_q2_min)
    return result
```

```python
def find_transition_points(polygon):

    sorted_left_vertices = find_farthest_left_vertices(polygon)
    left_below = sorted_left_vertices[0]
    left_up = sorted_left_vertices[1]
    sorted_right_vertices = find_farthest_right_vertices(polygon)
    right_below = sorted_right_vertices[0]
    right_up = sorted_right_vertices[1]

# Calculate the total distance for each of the 4 pairs of vertices, for the polygon's 4 sides
    total_distance_left_to_right = find_total_distance(polygon, left_up, right_up)
    total_distance_top_to_bottom = find_total_distance(polygon, right_up, right_below)
    total_distance_right_to_left = find_total_distance(polygon, right_below, left_below)
    total_distance_bottom_to_top = find_total_distance(polygon, left_below, left_up)

# Find the transtion points for each of the 4 pairs of vertices, the half of the total distance
    left_point = find_point_for_half_distance(polygon,left_below, total_distance_bottom_to_top)
    right_point = find_point_for_half_distance(polygon, right_up, total_distance_top_to_bottom)
    top_point = find_point_for_half_distance(polygon, left_up, total_distance_left_to_right)
    bottom_point = find_point_for_half_distance(polygon, right_below,
total_distance_right_to_left)
    return [top_point, right_point, bottom_point, left_point]
```

```python
def find_farthest_left_vertices(polygon):

    # Define a lambda function to extract the x-coordinate from each point
    x_coordinate = lambda point: point[0]
    # Sort the polygon based on x-coordinate using the defined lambda function
    sorted_polygon_left = sorted(polygon, key=x_coordinate)
    # Get the first two points from the sorted_polygon_left
    left_vertices = sorted_polygon_left[:2]
    # Define a lambda function to extract the y-coordinate from each point
    y_coordinate = lambda point: point[1]
    # Sort the left_vertices based on y-coordinate and get the two points
    sorted_left_vertices = sorted(left_vertices, key=y_coordinate)
    # Assign the first value to left_below and the second value to left_up
    left_below = sorted_left_vertices[0]
    left_up = sorted_left_vertices[1]
    return left_below, left_up


def find_farthest_right_vertices(polygon):

    # Define a lambda function to extract the y-coordinate from each point
    y_coordinate = lambda point: point[0]
    # Sort the polygon based on y-coordinate in reverse order
    sorted_polygon_right = sorted(polygon, key=y_coordinate, reverse=True)
    # Get the first two points from the sorted_polygon_right
    right_vertices = sorted_polygon_right[:2]
    # Define a lambda function to extract the y-coordinate from each point
    y_coordinate = lambda point: point[1]
    # Sort the right_vertices based on y-coordinate and get the two points
    sorted_right_vertices = sorted(right_vertices, key=y_coordinate)
    # Assign the first value to right_below and the second value to right_up
    right_below = sorted_right_vertices[0]
    right_up = sorted_right_vertices[1]
    return right_below, right_up
```

ΥΛΟΠΟΙΗΣΗ ΤΟΥ ΑΛΓΟΡΙΘΜΟΥ ΣΕ ΚΩΔΙΚΑ (6)

- **def find_total_distance (polygon, start_vertex, end_vertex)**

Δέχεται ως είσοδο όλους του κόμβους του πολυγώνου, και ξεχωριστά τους 2 ακρινούς κόμβους, των οποίων την απόσταση υπολογίζει. Αναζητά στο σύνολο των κόμβων τον αρχικό, και αφού τον εντοπίσει, προσθέτει την απόσταση των ενδιάμεσων ακμών, ώσπου να καταλήξει στον τελικό.

Επιστρέφει την συνολική απόσταση που καταμέτρησε, η οποία αντιστοιχεί στην συνολική απόσταση.

- **def find_point_for_half_distance (polygon, start_point, total_distance)**

Δέχεται ως είσοδο όλους του κόμβους του πολυγώνου, τον αρχικό ακρινό κόμβο, και την συνολική απόσταση που επέστρεψε η προηγούμενη μέθοδος. Ομοίως, εντοπίζει το αρχικό σημείο, και υπολογίζει την απόσταση που διανύει, καθώς κινείται στις ακμές.

Ελέγχει κάθε φορά στην επόμενη ακμή, αν η προσθήκη του μήκους της ξεπερνάει την μισή της συνολικής, κι εφόσον δεν την ξεπερνά, την προσθέτει και περνά στην επόμενη, ειδάλλως προσθέτει στην συντεταγμένη του αρχικού σημείου της ακμής, που δεν έχει κοινή τιμή με τον δεύτερο κόμβο του ζεύγους, το εναπομείναντος μήκος.

Επιστρέφει το σημείο που κατέληξε, το οποίο αντιστοιχεί στο σημείο μετάβασης.

```python
# This method calculates the total distance of
# the combination of edges connecting the pair of vertices

def find_total_distance(polygon, start_vertex, end_vertex):

    n = len(polygon)
    total_distance = 0
    # Find the indices of the start and end vertices in the polygon
    # in order to know when to start, and when to stop
    start_index = polygon.index(start_vertex)
    end_index = polygon.index(end_vertex)
    # set the current vertice as the starting vertice
    current_index = start_index

    # Check all the vertices from the start to the end of the side
    while (current_index) != (end_index):

        # For consecutive vertices, add their edge's distance to the total
        x1 = polygon[current_index][0]
        y1 = polygon[current_index][1]
        x2 = polygon[(current_index + 1) % n][0]        # % n to stay in range
        y2 = polygon[(current_index + 1) % n][1]

        distance_x = abs(x1 - x2)
        distance_y = abs(y1 - y2)
        total_distance += distance_x + distance_y
        current_index = (current_index + 1) % n

    return total_distance
```

```python
# This method calculates half of the total distance of the pair of vertices

def find_point_for_half_distance(polygon, start_point, total_distance):

    n = len(polygon)
    current_distance = 0
    half_distance = total_distance / 2

    # Check every vertice of the polygon, until you find the starting one
    for i in range(n):
        # (x1,y1) the first vertice of the pair, and (x2,y2) the second
        x1 = polygon[i][0]
        y1 = polygon[i][1]
        x2 = polygon[(i + 1) % n][0]
        y2 = polygon[(i + 1) % n][1]

        if (x1, y1) == (start_point):
            for j in range(i,n):
                x1 = polygon[j][0]
                y1 = polygon[j][1]
                x2 = polygon[(j + 1) % n][0]
                y2 = polygon[(j + 1) % n][1]
                distance_x = abs(x1 - x2)
                distance_y = abs(y1 - y2)

                edge_distance = distance_x + distance_y
                # If you cover the half distance with the currrent edge,
                # the transition point is located on it
                if (current_distance + edge_distance) >= (half_distance):
                    remaining_distance = half_distance - current_distance

                    # If the consecutive vertices create a horizontal edge
                    # calculate the y price, differentiate the upper and lower side
                    if (x1) == (x2):
                        if (y1) > (y2):
                            transition_point = (x1, y1 - remaining_distance)
                        else:
                            transition_point = (x1, y1 + remaining_distance)

                    # If the consecutive vertices create a vertical edge
                    # calculate the x price, differentiate the left and right side
                    else:
                        if (x1) > (x2):
                            transition_point = (x1 - remaining_distance, y1)
                        else:
                            transition_point = (x1 + remaining_distance, y1)
                    return transition_point

                # If you haven't covered the half distance with the currrent edge,
                # add it to the total distance covered, and move to the next edge
                current_distance += edge_distance
```

ΥΛΟΠΟΙΗΣΗ ΤΟΥ ΑΛΓΟΡΙΘΜΟΥ ΣΕ ΚΩΔΙΚΑ (7)

➢ **def get_L1_projection_points (poly)**

Επιλέγει διαδοχικά όλους τους κόμβους του πολυγώνου, και τους συγκρίνει με τα ζεύγη των κόμβων που δεν περιλαμβάνουν τον επιλεχθέντα.

Εξετάζει αρχικά αν η ακμή που σχηματίζει το εκάστοτε ζεύγος είναι κάθετη ή οριζόντια, προκειμένου να διαπιστωθεί εάν είναι πιθανό να εντοπιστεί σημείο προβολής, με βάση την κοινή τους συντεταγμένη.

Εφόσον είναι εφικτό, συγκρίνουμε τη θέση του επιλεχθέντα κόμβου με του ζεύγους, ώστε να διαπιστωθεί ποια είναι η σχετική του τοποθέτηση (πάνω – κάτω / δεξιά - αριστερά), και στη συνέχεια προσθέτουμε στη λίστα των Σημείων Προβολής σημείο με τη μια συντεταγμένη, την αντίστοιχη με τιμή την κοινή του ζεύγους, και την άλλη με την αντίστοιχη τιμή του επιλεχθέντα.

Επιστρέφει την λίστα των σημείων, που περιλαμβάνει όλα τα L_1 -προβολή Σημεία του πολυγώνου.

➢ **def find_l1_set (poly, transition_points, L1_projection_points, horizontal_line)**

Δέχεται τους κόμβους, τα Σημεία Μετάβασης, τα Σημεία Προβολής, και την οριζόντια γραμμή που βρίσκεται ανάμεσα στα πολύγωνα.

Εξετάζουμε όλα τα σημεία από τα σετ που έλαβε η μέθοδος ως εισόδους και αφαιρούμε αυτά για τα οποία υπάρχει ζεύγος διαδοχικών κόμβων, με την κοινή τους συντεταγμένη πιο κοντά στην οριζόντια γραμμή από ότι η αντίστοιχη συντεταγμένη του σημείου, και την άλλη τους συντεταγμένη, του ενός μεγαλύτερη από το εξεταζόμενο σημείο, και του άλλου μικρότερη.

```python
# STEP 1.2 : L1 projection points are the points projected by a vertice on an edge of the polygon
def get_L1_projection_points(poly):
    n = len(poly)
    projection_points = []
    for i in range(n):
        current_vertex = poly[i]
        x_current = current_vertex[0]
        y_current = current_vertex[1]
        # Initialize variables to store the closest pair and its projection point
        closest_projection = None
        # Iterate over pairs of consecutive vertices
        for j in range(n):
            # Set the pair of vertices you compare to the one you check
            first_next_index = (j + 1) % n
            first_other_vertex = poly[first_next_index]
            first_x_other = first_other_vertex[0]
            first_y_other = first_other_vertex[1]
            second_next_index = (j + 2) % n
            second_other_vertex = poly[second_next_index]
            second_x_other = second_other_vertex[0]
            second_y_other = second_other_vertex[1]
            # Check if the conditions are met for x-axis projection
            is_same_x_axis = (first_x_other == second_x_other)
            is_y_current_between = (first_y_other > y_current and second_y_other < y_current)
            is_y_current_between_reverse = (first_y_other < y_current and second_y_other > y_current)
            # if the pair is on a vertical edge, and are the one above and the other below the vertice
            # add the point projected on the y axis price of the vertice on their edge
            if (is_same_x_axis) and ((is_y_current_between) or (is_y_current_between_reverse)):
                projection = (first_x_other, y_current)
                distance_condition = lambda cv, p, cp: (euclidean_distance(cv, p)) < (euclidean_distance(cv, cp))
                if (closest_projection is None) or (distance_condition(current_vertex, projection, closest_projection)):
                    closest_projection = projection
            # Check if the conditions are met for y-axis projection
            is_same_y_axis = (first_y_other == second_y_other)
            is_x_current_between = (first_x_other > x_current > second_x_other)
            is_x_current_between_reverse = (first_x_other < x_current < second_x_other)
            # if the pair is on a horizontal edge, and are the one left and the other right of the vertice
            # add the point projected on the x-axis price of the vertice on their edge
            if (is_same_y_axis) and ((is_x_current_between) or (is_x_current_between_reverse)):
                projection = (x_current, first_y_other)
                distance_condition = lambda cv, p, cp: euclidean_distance(cv, p) < euclidean_distance(cv, cp)
                if (closest_projection is None) or (distance_condition(current_vertex, projection, closest_projection)):
                    closest_projection = projection
        # Append the closest projection point to the list if it is not None and not already in the list
        if (closest_projection is not None) and (closest_projection not in projection_points):
            projection_points.append(closest_projection)
    return projection_points
```

```python
# STEP 2.2 : the vertices, transition and L1-projection points of the polygon
#            visible by the horizontal line, meaning connected by 1 vertical line,
#            that doesn't enter the interior areas of the polygon

def find_l1_set(poly, transition_points, L1_projection_points, horizontal_line):

    l1_set = set()
    for vertex in poly:
        l1_set.add(vertex)
    for point in transition_points:
        l1_set.add(point)
    for projection_point in L1_projection_points:
        l1_set.add(projection_point)
    # Remove all occurrences of None from the l1_set in place
    l1_set.difference_update({None})

    # If the polygon is below the horizontal line, exclude vertices that have an edge above them,
    # meaning a pair of consecutive vertices that have greater y-coordinate, and lesser-greater x-coordinate
    if (min(p[1] for p in poly)) < (horizontal_line[0][1]):
        tail_poly = poly[1:]
        first_vertex = poly[0]
        extended_poly = tail_poly + [first_vertex]
        paired_vertices = zip(poly, extended_poly)

        for v1, v2 in paired_vertices:
            if (v1[1] == v2[1]):
                excluded_vertices_condition = lambda v3: (v3[1] < v1[1]) and ( (v1[0] >= v3[0] >= v2[0]) or (v1[0] <= v3[0] <= 
v2[0]) )
                excluded_vertices = {v3 for v3 in l1_set if excluded_vertices_condition(v3)}
                l1_set -= excluded_vertices

    # If the polygon is above the horizontal line, exclude vertices that have an edge below them,
    # meaning a pair of consecutive vertices that have lesser y-coordinate, and lesser-greater x-coordinate
    elif (max(p[1] for p in poly) > horizontal_line[0][1]):
        tail_poly = poly[1:]
        first_vertex = poly[0]
        extended_poly = tail_poly + [first_vertex]
        paired_vertices = zip(poly, extended_poly)

        for v1, v2 in paired_vertices:
            if (v1[1] == v2[1]):
                excluded_vertices_condition = lambda v3: (v3[1] > v1[1]) and ( (v1[0] >= v3[0] >= v2[0]) or (v1[0] <= v3[0] <= 
v2[0]) )
                excluded_vertices = {v3 for v3 in l1_set if excluded_vertices_condition(v3)}
                l1_set -= excluded_vertices
    return l1_set
```

ΥΛΟΠΟΙΗΣΗ ΤΟΥ ΑΛΓΟΡΙΘΜΟΥ ΣΕ ΚΩΔΙΚΑ (8)

- **def find_horizontal_line (poly_P, poly_Q)**

Δέχεται ως είσοδο τους κόμβους των πολυγώνων, επιλέγει τους υψηλότερους και χαμηλότερους κόμβους του κάθε πολυγώνου, διαπιστώνει ποιο είναι από πάνω, και υπολογίζει την μισή απόσταση ανάμεσα στον χαμηλότερο κόμβο του υψηλότερου, και τον χαμηλότερο του υψηλότερου.

Η οριζόντια γραμμή προκύπτει από την y συντεταγμένη της μισής απόστασης, και η x συντεταγμένη από τους 2 πιο συνολικά ακρινούς κόμβους τους.   Δηλαδή τον αριστερότερο κόμβο του αριστερά πολυγώνου, και του δεξιότερου κόμβου του δεξιού.

- **def find_vertical_line (poly_P, poly_Q)**

Δέχεται ως είσοδο τους κόμβους των πολυγώνων, επιλέγει τους δεξιότερους και αριστερότερους κόμβους του κάθε πολυγώνου, διαπιστώνει ποιο είναι δεξιότερα, και υπολογίζει την μισή απόσταση ανάμεσα στον δεξιότερο κόμβο του αριστερότερου, και τον αριστερότερο του δεξιότερου.

Η κάθετη γραμμή προκύπτει από την x συντεταγμένη της μισής απόστασης, και η y συντεταγμένη από τους 2 πιο συνολικά ακρινούς κόμβους τους. Δηλαδή τον υψηλότερο κόμβο του από πάνω πολυγώνου, και του χαμηλότερου κόμβου του από κάτω.

```python
# STEP 2.1 : The horizontal line is above the lower and below the upper polygon,
#       so we use the midpoint of the distance between their top and bottom vertice respectively

def find_horizontal_line(poly_P, poly_Q):

    # Find the vertexes with the minimum and the max y-coordinate in poly_P
    bottom_vertex_P = min(poly_P, key=lambda p: p[1])
    top_vertex_P = max(poly_P, key=lambda p: p[1])
    # Find the vertexes with the minimum and the max y-coordinate in poly_Q
    bottom_vertex_Q = min(poly_Q, key=lambda q: q[1])
    top_vertex_Q = max(poly_Q, key=lambda q: q[1])

    # Determine which polygon is upper and which is lower
    if (bottom_vertex_P[1]) < (bottom_vertex_Q[1]):
        upper_polygon = poly_Q
        lower_polygon = poly_P
        # Midpoint between bottom of the upper and top of the lower
        y_coordinate = (bottom_vertex_Q[1] - top_vertex_P[1]) / 2 + top_vertex_P[1]
    else:
        upper_polygon = poly_P
        lower_polygon = poly_Q
        # Midpoint between bottom of the upper and top of the lower
        y_coordinate = (bottom_vertex_P[1] - top_vertex_Q[1]) / 2 + top_vertex_Q[1]

    # Find the farthest left and right points on the x-axis for both polygons
    min_x_upper = min(p[0] for p in upper_polygon)
    min_x_lower = min(p[0] for p in lower_polygon)
    max_x_upper = max(p[0] for p in upper_polygon)
    max_x_lower = max(p[0] for p in lower_polygon)
    left_point = min(min_x_upper, min_x_lower)
    right_point = max(max_x_upper, max_x_lower)
    # Create a horizontal line using the calculated coordinates
    # (the -1 is added in order to always include the max vertices)
    horizontal_line = [(left_point-1, y_coordinate), (right_point+1, y_coordinate)]
    return horizontal_line
```

```python
# The vertical line is right of the left and left of the right polygon,
# so we use the midpoint of the distance between their max left and right vertice respectively

def find_vertical_line(poly_P, poly_Q):

    # Find the vertex with the maximum x-coordinate in poly_P
    rightmost_vertex_P = max(poly_P, key=lambda p: p[0])
    leftmost_vertex_P = min(poly_P, key=lambda p: p[0])
    # Find the vertex with the minimum x-coordinate in poly_Q
    rightmost_vertex_Q = max(poly_Q, key=lambda q: q[0])
    leftmost_vertex_Q = min(poly_Q, key=lambda q: q[0])

    # Determine which polygon is on the left and which is on the right
    if rightmost_vertex_P[0] < leftmost_vertex_Q[0]:
        left_polygon = poly_P
        right_polygon = poly_Q
        # Midpoint between the rightmost of the left and leftmost of the right
        x_distance = (leftmost_vertex_Q[0] - rightmost_vertex_P[0]) / 2
        x_coordinate = x_distance + rightmost_vertex_P[0]
    else:
        left_polygon = poly_Q
        right_polygon = poly_P
        # Midpoint between the rightmost of the left and leftmost of the right
        y_distance = (leftmost_vertex_P[0] - rightmost_vertex_Q[0]) / 2
        x_coordinate = y_distance + leftmost_vertex_P[0]

    # Find the farthest top and bottom points on the y-axis for both polygons
    max_y_left_polygon = max(p[1] for p in left_polygon)
    min_y_left_polygon = min(p[1] for p in left_polygon)
    max_y_right_polygon = max(p[1] for p in right_polygon)
    min_y_right_polygon = min(p[1] for p in right_polygon)
    topmost_point = max(max_y_left_polygon, max_y_right_polygon)
    bottommost_point = min(min_y_left_polygon, min_y_right_polygon)
    # Create a vertical line using the calculated coordinates
    # (the -1 is added in order to always include the max vertices)
    top_point = topmost_point + 1
    bottom_point = bottommost_point - 1
    vertical_line = [(x_coordinate, top_point), (x_coordinate, bottom_point)]
    return vertical_line
```

ΥΛΟΠΟΙΗΣΗ ΤΟΥ ΑΛΓΟΡΙΘΜΟΥ ΣΕ ΚΩΔΙΚΑ (9)

- **def find_l2_set (poly, transition_points, L1_projection_points, vertical_line)**

Δέχεται τους κόμβους, τα Σημεία Μετάβασης, τα Σημεία Προβολής, και την κάθετη που βρίσκεται ανάμεσα στα πολύγωνα.

Εξετάζει όλα τα σημεία από τα σετ που έλαβε η μέθοδος ως εισόδους και αφαιρούμε αυτά για τα οποία υπάρχει ζεύγος διαδοχικών κόμβων, με την κοινή τους συντεταγμένη πιο κοντά στην κάθετη γραμμή από ότι η αντίστοιχη συντεταγμένη του σημείου, και την άλλη τους συντεταγμένη, του ενός μεγαλύτερη από το εξεταζόμενο σημείο, και του άλλου μικρότερη.

- **def generate_candidates (point_set, neighbor_set, horizontal_line, vertical_line)**

Δέχεται το σετ $l_1(P)$ (ή $l_2(P)$, ή $l_1(Q)$, ή $l_2(Q)$), τους κόμβους του πολυγώνου, το οριζόντιο και το κάθετο ευθύγραμμο τμήμα που χωρίζουν τα πολύγωνα P,Q.

Για κάθε σημείο του εκάστοτε σετ, υπολογίζει την διαφορά της x συντεταγμένης του από την κάθετη γραμμή, και την διαφορά της y συντεταγμένης του από την οριζόντια γραμμή.

Στη συνέχεια υπολογίζει την απόσταση του μονοπατιού που συνδέει το εκάστοτε σημείο, με κάθε ζεύγος σημείων του σετ κόμβων του πολυγώνου, ώστε να βρει τον απώτατο γείτονα.

```
#            the vertices, transition and L1-projection points of the polygon
#            visible by the vertical line, meaning connected by 1 horizontal line,
#            that doesn't enter the interior areas of the polygon

def find_l2_set(poly, transition_points, L1_projection_points, vertical_line):

    l2_set = set()
    for vertex in poly:
        l2_set.add(vertex)
    for point in transition_points:
        l2_set.add(point)
    for projection_point in L1_projection_points:
        l2_set.add(projection_point)
    # Remove all occurrences of None from the l2_set in place
    l2_set.difference_update({None})

    # If the polygon is right of the vertical line, exclude vertices that have an edge left of them,
    # meaning a pair of consecutive vertices that have lesser x-coordinate, and lesser-greater y-coordinate
    if (max(p[0] for p in poly) > vertical_line[0][0]):
        tail_poly = poly[1:]
        first_vertex = poly[0]
        extended_poly = tail_poly + [first_vertex]
        paired_vertices = zip(poly, extended_poly)
        for v1, v2 in paired_vertices:
            if (v1[0] == v2[0]):
             excluded_vertices_condition = lambda v3: (v3[0] > v1[0]) and ( (v1[1] >= v3[1] >= v2[1]) or (v1[1] <= v3[1] <= v2[1]) )
               excluded_vertices = {v3 for v3 in l2_set if excluded_vertices_condition(v3)}
               l2_set -= excluded_vertices

    # If the polygon is left of the vertical line, exclude vertices that have an edge right of them,
    # meaning a pair of consecutive vertices that have greater y-coordinate, and lesser-greater x-coordinate
    elif (min(p[0] for p in poly) < vertical_line[0][0]):
        tail_poly = poly[1:]
        first_vertex = poly[0]
        extended_poly = tail_poly + [first_vertex]
        paired_vertices = zip(poly, extended_poly)

        for v1, v2 in paired_vertices:
            if (v1[0] == v2[0]):
                excluded_vertices_condition = lambda v3: (v3[0] < v1[0]) and ( (v1[1] >= v3[1] >= v2[1]) or (v1[1] <= v3[1] <=
v2[1]) )
                excluded_vertices = {v3 for v3 in l2_set if excluded_vertices_condition(v3)}
                l2_set -= excluded_vertices

    return l2_set
```

```python
# STEP 3

#   m(x) = min[(x belongs in I1/2) l(P/Q)] m(x)
#   m(p) = L1(p,f(p)) + |xp| + |yp|
#   L1 the distance of the path between p and its farthest neighbor f(p)
#   |xp|,|yp| the distance on the x-axis and y-axis respectively
#   We find the point with the minimum m price of the set

def generate_candidates(point_set, neighbor_set, horizontal_line, vertical_line):

    # the variants that hold the point with the mu price and its corresponding point
    min_price = float('inf')
    min_price_point = None
    # we check all the points of the given set, to find every mu, in order to choose the min

    for point in point_set:
        # first find the distance from the dividing lines
        distance_horizontal = abs(point[1] - horizontal_line[0][1])
        distance_vertical = abs(point[0] - vertical_line[0][0])
        max_path = find_most_distant_neighbor_path(point, neighbor_set)
        # add the 3 values you found
        total_distance = distance_horizontal + distance_vertical + max_path
        # compare to the current min price, and replace it if the new is lesser

        if (total_distance < min_price):
            min_price = total_distance
            min_price_point = point
        # return poth the point and the price

    return min_price_point, min_price
```

ΥΛΟΠΟΙΗΣΗ ΤΟΥ ΑΛΓΟΡΙΘΜΟΥ ΣΕ ΚΩΔΙΚΑ (10)

- **def find_most_distant_neighbor_path (point, neighbor_set)**

Εξετάζει όλα τα ζεύγη των κόμβων του πολυγώνου, και μετράει τις αποστάσεις από τον κάθε ακριανό κόμβο. Ξεκινάμε να μετράμε την απόσταση μόνο αφότου εντοπίσουμε το σημείο που μας δόθηκε, και καταμετρούμε τους κόμβους, ώστε να υπολογίσουμε ξανά από την αρχή τις ακμές του πολυγώνου που προσπεράσαμε, και περιλαμβάνονται στην απόσταση του μονοπατιού για τους άλλους κόμβους.

Αφότου έχουν υπολογιστεί και οι 4 αποστάσεις του σημείου, τόσο δεξιόστροφα, όσο και αριστερόστροφα (δηλαδή συνολικό μήκος – δεξιόστροφη απόσταση) από τους 4 ακριανούς κόμβους, συγκρίνουμε τη διαφορά της δεξιόστροφης και αριστερόστροφης απόστασης για τον κάθε ακριανό κόμβο, και κρατάμε την μικρότερη, καθώς υποδεικνύει ότι ανήκει στον απώτερο κόμβο, αφού οι άλλοι 3 θα έχουν την 1 απόσταση σχετικά πιο κοντά, και την άλλη σχετικά πιο μακριά, ενώ ο πιο μακρινός, θα είναι σχετικά εξίσου μακριά και από τις δυο κατευθύνσεις.

- **def find_F2 (mu_p1, mu_p2, mu_q1, mu_q2)**

Υπολογίζει με βάση τον τύπο $F_2 (p_1, q_2) = \mu(p_1) + \mu(q_2) \leq F_2 (p_2, q_1) = \mu(p_2) + \mu(q_1)$, ποια γέφυρα, δηλαδή ποια τιμή και ποια σημεία θα επιστρέψει, από τα 4 ορίσματα.

```python
def find_most_distant_neighbor_path(point, neighbor_set):
# find the 4 most distant vertices of the polygon from the neighbor_set
    # for the left
    sorted_left_vertices = sorted(neighbor_set, key=lambda point: point[0])
    left_vertices = sorted_left_vertices[:2]
    sorted_left_vertices_by_y = sorted(left_vertices, key=lambda point: point[1])
    left_below = sorted_left_vertices_by_y[0]
    left_up = sorted_left_vertices_by_y[1]
    # for the right
    sorted_right_vertices = sorted(neighbor_set, key=lambda point: point[0], reverse=True)
    right_vertices = sorted_right_vertices[:2]
    sorted_right_vertices_by_y = sorted(right_vertices, key=lambda point: point[1])
    right_below = sorted_right_vertices_by_y[0]
    right_up = sorted_right_vertices_by_y[1]
# set the variants
    total_edge_length = 0
    tail_neighbor_set = neighbor_set[1:]
    first_vertex = neighbor_set[0]
    extended_neighbor_set = tail_neighbor_set + [first_vertex]
    paired_vertices = zip(neighbor_set, extended_neighbor_set)
    countVertices=0
    FoundPoint=0
    missedVertices=0
    countDistant=0
    foundFirst=0
    distanceToFirst=0
    distanceToSecond=0
    distanceToThird=0
    distanceToFourth=0
    second_distanceToFirst=0
    second_distanceToSecond=0
    second_distanceToThird=0
    second_distanceToFourth=0
    max_path=0
    max_distanceToFirst=0
    max_distanceToSecond=0
    max_distanceToThird=0
    max_distanceToFourth=0
    # compare the chosen point to every couple of corresponding vertices of the polygon
    # count the vertices and the total distanace you pass along the edges,
    # until you find the vertices that define the edge that contains the chosen point
    for v1, v2 in paired_vertices:
        countVertices += 1
        # when you have not found the vertices that define the edge
        # that contains the chosen point, check if you are on it
        if (FoundPoint == 0):
            if (v1[1] == v2[1]):
                condition_1 = (v1[0] <= point[0] <= v2[0])
                condition_2 = (v2[0] <= point[0] <= v1[0])
                if (condition_1) or (condition_2):
                    total_edge_length += abs(point[0] - v1[0])
            elif (v1[0] == v2[0]):
                condition_1 = (v1[1] <= point[1] <= v2[1])
                condition_2 = (v2[1] <= point[1] <= v1[1])
                if (condition_1) or (condition_2):
                    total_edge_length += abs(point[1] - v1[1])
            if (v1[0] == v2[0]):
                if (point[0] == v1[0]):
                    condition_1 = (v1[1] < point[1] < v2[1])
                    condition_2 = (v2[1] < point[1] < v1[1])
                    if (condition_1) or (condition_2):
                        FoundPoint=1
                        missedVertices = countVertices
            if (v1[1] == v2[1]):
                if (point[1] == v1[1]):
                    condition_1 = (v1[0] < point[0] < v2[0])
                    condition_2 = (v2[0] < point[1] < v1[0])
                    if (condition_1) or (condition_2):
                        FoundPoint=1
                        missedVertices = countVertices
```

```
# when you already found the vertices that define the edge that contains the chosen point,
        # when you find a most distand vertice, since one of those 4 is the farthest neighbor,
        # if it is the first you find, save it, in order to know which vertices you missed
        if (FoundPoint == 1):
            if (v1[1] == v2[1]):
                condition_1 = (v1[0] <= point[0] <= v2[0])
                condition_2 = (v2[0] <= point[0] <= v1[0])
                if (condition_1) or (condition_2):
                    total_edge_length += abs(point[0] - v1[0])
                if (v1==left_up):
                    countDistant+=1
                    distanceToFirst=total_edge_length
                    if (foundFirst!=0):
                        foundFirst = 1
                if (v1==right_up):
                    countDistant+=1
                    distanceToSecond=total_edge_length
                    if (foundFirst!=0):
                        foundFirst = 2
                if (v1==right_below):
                    countDistant+=1
                    distanceToThird=total_edge_length
                    if (foundFirst!=0):
                        foundFirst = 3
                if (v1==left_below):
                    countDistant+=1
                    distanceToFourth=total_edge_length
                    if (foundFirst!=0):
                        foundFirst = 4
            elif (v1[0] == v2[0]):
                condition_1 = (v1[1] <= point[1] <= v2[1])
                condition_2 = (v2[1] <= point[1] <= v1[1])
                if (condition_1) or (condition_2):
                    total_edge_length += abs(point[1] - v1[1])
                if (v1==left_up):
                    countDistant+=1
                    distanceToFirst=total_edge_length
                    if (foundFirst!=0):
                        foundFirst = 1
                if (v1==right_up):
                    countDistant+=1
                    distanceToSecond=total_edge_length
                    if (foundFirst!=0):
                        foundFirst = 2
                if (v1==right_below):
                    countDistant+=1
                    distanceToThird=total_edge_length
                    if (foundFirst!=0):
                        foundFirst = 3
                if (v1==left_below):
                    countDistant+=1
                    distanceToFourth=total_edge_length
                    if (foundFirst!=0):
                        foundFirst = 4
```

```python
# if you have not seen all the vertices, start again
        # until you have counted the distances to each of them
        if (countDistant < 4):
            tail_neighbor_set2 = neighbor_set[missedVertices:]
            first_vertex2 = neighbor_set[countDistant]
            extended_neighbor_set2 = tail_neighbor_set2 + [first_vertex2]
            paired_vertices2 = zip(neighbor_set, extended_neighbor_set2)
            for v1, v2 in paired_vertices2:
                if (v1[1] == v2[1]):
                    condition_1 = (v1[0] <= point[0] <= v2[0])
                    condition_2 = (v2[0] <= point[0] <= v1[0])
                    if (condition_1) or (condition_2):
                        total_edge_length += abs(point[0] - v1[0])
                    if (v1==left_up):
                        countDistant+=1
                        distanceToFirst=total_edge_length
                        if (foundFirst!=0):
                            foundFirst = 1
                    if (v1==right_up):
                        countDistant+=1
                        distanceToSecond=total_edge_length
                        if (foundFirst!=0):
                            foundFirst = 2
                    if (v1==right_below):
                        countDistant+=1
                        distanceToThird=total_edge_length
                        if (foundFirst!=0):
                            foundFirst = 3
                    if (v1==left_below):
                        countDistant+=1
                        distanceToFourth=total_edge_length
                        if (foundFirst!=0):
                            foundFirst = 4
                elif (v1[0] == v2[0]):
                    condition_1 = (v1[1] <= point[1] <= v2[1])
                    condition_2 = (v2[1] <= point[1] <= v1[1])
                    if (condition_1) or (condition_2):
                        total_edge_length += abs(point[1] - v1[1])
                    if (v1==left_up):
                        countDistant+=1
                        distanceToFirst=total_edge_length
                        if (foundFirst!=0):
                            foundFirst = 1
                    if (v1==right_up):
                        countDistant+=1
                        distanceToSecond=total_edge_length
                        if (foundFirst!=0):
                            foundFirst = 2
                    if (v1==right_below):
                        countDistant+=1
                        distanceToThird=total_edge_length
                        if (foundFirst!=0):
                            foundFirst = 3
                    if (v1==left_below):
                        countDistant+=1
                        distanceToFourth=total_edge_length
                        if (foundFirst!=0):
                            foundFirst = 4
                # for safety an additional check
            if (countDistant == 4):
                break;
    # now count the actual distance, of all the edges of the polygon
    total_edge_length=0
    for v1, v2 in paired_vertices:
        if (v1[1] == v2[1]):
            condition_1 = (v1[0] <= point[0] <= v2[0])
            condition_2 = (v2[0] <= point[0] <= v1[0])
            if (condition_1) or (condition_2):
                total_edge_length += abs(point[0] - v1[0])
        elif (v1[0] == v2[0]):
            condition_1 = (v1[1] <= point[1] <= v2[1])
            condition_2 = (v2[1] <= point[1] <= v1[1])
            if (condition_1) or (condition_2):
                total_edge_length += abs(point[1] - v1[1])
    # the distanceTo_ is the path from left to right
    # the second_distanceTo_ is the path from right to left
    second_distanceToFirst = total_edge_length - distanceToFirst
    second_distanceToSecond = total_edge_length - distanceToSecond
    second_distanceToThird = total_edge_length - distanceToThird
    second_distanceToFourth = total_edge_length - distanceToFourth
    # choose the max path, of the 2
    max_distanceToFirst = max(distanceToFirst, second_distanceToFirst)
    max_distanceToSecond = max(distanceToSecond, second_distanceToSecond)
    max_distanceToThird = max(distanceToThird, second_distanceToThird)
    max_distanceToFourth = max(distanceToFourth, second_distanceToFourth)
    # choose the min path, of the 4, since it is the farthest neighbor
    max_path = min(max_distanceToFirst, max_distanceToSecond, max_distanceToThird, max_distanceToFourth)
    return max_path
```

```python
# STEP 4

def find_F2(mu_p1, mu_p2, mu_q1, mu_q2):

    mu_p1_min_price_point = mu_p1[0]
    mu_p1_min_price = mu_p1[1]
    mu_q2_min_price_point = mu_q2[0]
    mu_q2_min_price = mu_q2[1]
    mu_p2_min_price_point = mu_p2[0]
    mu_p2_min_price = mu_p2[1]
    mu_q1_min_price_point = mu_q1[0]
    mu_q1_min_price = mu_q1[1]
    F2_1 = mu_p1_min_price + mu_q2_min_price
    F2_2 = mu_p2_min_price + mu_q1_min_price

    # If [F2(p1,q2) less than or equal to F2 (p2,q1)] :
      return euclidean(p1,r) + euclidean(r,p2), where r=(xp1,yq2)
    # else : return euclidean(p2,r) + euclidean(r,q1), where r=(xq1,yq2)
    if (F2_1 <= F2_2):
        r = (mu_p1_min_price_point[0], mu_q2_min_price_point[1])
        distance_p1_to_r = euclidean_distance(mu_p1_min_price_point, r)
        distance_r_to_q2 = euclidean_distance(r, mu_q2_min_price_point)
        eucl_result = distance_p1_to_r + distance_r_to_q2
        final_p = mu_p1_min_price_point
        final_q = mu_q2_min_price_point
    else:
        r = (mu_q1_min_price_point[0], mu_p2_min_price_point[1])
        distance_p2_to_r = euclidean_distance(mu_p2_min_price_point, r)
        distance_r_to_q1 = euclidean_distance(r, mu_q1_min_price_point)
        eucl_result = distance_p2_to_r + distance_r_to_q1
        final_p = mu_q1_min_price_point
        final_q = mu_p2_min_price_point
    result = eucl_result, final_p, final_q, r

    return result
```

ΥΛΟΠΟΙΗΣΗ ΤΟΥ ΑΛΓΟΡΙΘΜΟΥ ΣΕ ΚΩΔΙΚΑ (11)

- **<u>def euclidean_distance (a, b)</u>**

Δέχεται 2 σημεία, και υπολογίζει την ευκλείδεια απόσταση τους.
Ευκλείδεια Απόσταση = [(x_a - x_b)^2 + (y_a - y_b)^2]^2 , όπου τα σημεία a = (x_a, y_a) , b = (x_b, y_b).

- **<u>def optimal_type_1_bridge(poly_P, poly_Q)</u>**

Καλεί όλες τις μεθόδους για τα βήματα του αλγορίθμου, διαδοχικά, δίνοντας ως κάποια από τα ορίσματα τους, κάποια από τα αποτελέσματα που επέστρεψαν προηγούμενες μέθοδοι.

Στο τέλος επιστρέφει την γέφυρα, ως την τιμή της όπως προκύπτει από τον τύπο, και τα 2 σημεία από τα οποία υπολογίζεται.

- **<u>def find_add_connectable_points (poly_1, poly_2, transition_points, L1_projection_points)</u>**

Διαχωρίζουμε τις ακμές των 2 πολυγώνων, δηλαδή ταξινομούμε σε αντίστοιχες λίστες τα ζεύγη των σημείων που τις αποτελούν, ανάλογα με το αν είναι παράλληλα στον y ή x άξονα, δηλαδή εξετάζουμε ποια συντεταγμένη έχουν κοινή τα ζεύγη, και τα τοποθετούμε στην αντίστοιχη λίστα και τα συγκρίνουμε όλα τα σημεία που ανήκουν στα σετ κόμβοι, Σημεία Μετάβασης, και Σημεία Προβολής.

Εφόσον διαπιστώσουμε ότι το σημείο που εξετάζουμε, έχει τιμή στην συντεταγμένη του σετ που δεν ανήκει, μικρότερη τιμή και μεγαλύτερη από 2 σημεία του ίδιου ζεύγους στην ίδια λίστα, και αφότου επιβεβαιώσουμε ότι δεν διακόπτονται από κάποια ακμή των πολυγώνων, το προσθέτουμε στη λίστα των υποψήφιων ζευγών για γέφυρα μαζί με το σημείο στην λίστα του άλλου πολυγώνου, που έχει ίδια συντεταγμένη με την ίδια του ζεύγους, και την άλλη ίδια με το σημείο που εξετάζουμε.

```python
# the euclidean distance is shown on the variables as an upper line
def euclidean_distance(a, b):
    return ((a[0] - b[0]) ** 2 + (a[1] - b[1]) ** 2) ** 0.5

def optimal_type_1_bridge(poly_P, poly_Q):
    # poly_P and poly_Q are lists of points representing the polygons
    # STEP 1
    # Initialize an empty list for points
    Bridge_Alg1 = []

    # STEP 2
    # 2.1 : Find Sets T(P), T(Q) of Transition Points
    transition_points_P = find_transition_points(poly_P)
    transition_points_Q = find_transition_points(poly_Q)
    # 2.2 : Find Sets L1P(P), L1P(Q) of L1-Projection Points
    L1_projection_points_P = get_L1_projection_points(poly_P)
    L1_projection_points_Q = get_L1_projection_points(poly_Q)

    # STEP 3
    # Call find_add_connectable_points of P and add the result to Bridge_Alg1
    connectable_points_P = find_add_connectable_points(poly_P, poly_Q, transition_points_P, L1_projection_points_P)
    Bridge_Alg1.extend(connectable_points_P)

    # STEP 4
    # Call find_and_add_connectable_points of Q and add the result to Bridge_Alg1
    connectable_points_Q = find_add_connectable_points(poly_Q, poly_P, transition_points_Q, L1_projection_points_Q)
    Bridge_Alg1.extend(connectable_points_Q)

    # STEP 5
    # remove the duplicates from Bridge_Alg1 for cases like vertice to vertice
    Bridge_Alg1 = remove_duplicates(Bridge_Alg1)
    # Calculate prices for each pair
    F1_list_P = find_F1_for_list(poly_P, connectable_points_P)
    F1_list_Q = find_F1_for_list(poly_Q, connectable_points_Q)

    # STEP 6
    result = F1_min(F1_list_P, F1_list_Q, connectable_points_P, connectable_points_Q)

    return result
```

```python
def find_add_connectable_points(poly_1, poly_2, transition_points, L1_projection_points):

    connectable_points = set()
    horizontal_points_poly_1 = []
    vertical_points_poly_1 = []
    # Find pairs of consecutive vertices in poly_1 forming horizontal lines
    poly_1_list = list(poly_1)
    paired_points = zip(poly_1_list, poly_1_list[1:])
    horizontal_lines_poly_1 = {(p1, p2) for p1, p2 in paired_points if (p1[1] == p2[1])}
    # Find pairs of consecutive vertices in poly_1 forming vertical lines
    poly_1_list = list(poly_1)
    paired_points_vertical = zip(poly_1_list, poly_1_list[1:])
    vertical_lines_poly_1 = {(p1, p2) for p1, p2 in paired_points_vertical if (p1[0] == p2[0])}
    # Add the first and last points of poly_1 to horizontal_lines_poly_1
    poly_1_list = list(poly_1)
    first_point_1 = poly_1_list[0]
    poly_1_list = list(poly_1)
    last_point_1 = poly_1_list[-1]
    # Add the tuples to horizontal_lines_poly_1
    vertical_lines_poly_1.add((last_point_1, first_point_1))
    # Find pairs of consecutive vertices in poly_2 forming horizontal lines
    paired_points_horizontal = zip(poly_2, poly_2[1:])
    horizontal_lines_poly_2 = {(p1, p2) for p1, p2 in paired_points_horizontal if (p1[1] == p2[1])}
    # Find pairs of consecutive vertices in poly_2 forming vertical lines
    paired_points_vertical = zip(poly_2, poly_2[1:])
    vertical_lines_poly_2 = {(p1, p2) for p1, p2 in paired_points_vertical if (p1[0] == p2[0])}
    # Add the first and last points of poly_1 to horizontal_lines_poly_1
    first_point_2 = list(poly_2)[0]
    last_point_2 = list(poly_2)[-1]
    # Add the tuples to horizontal_lines_poly_1
    vertical_lines_poly_2.add((last_point_2, first_point_2))

    for point in horizontal_lines_poly_1:
        horizontal_points_poly_1.append(point[0])
        horizontal_points_poly_1.append(point[1])

    for point in vertical_lines_poly_1:
        vertical_points_poly_1.append(point[0])
        vertical_points_poly_1.append(point[1])
    # Check which set transition_points belong to and add points to the respective list

    for point in transition_points:
        x = point[0]
        y = point[1]
        for (p1, p2) in horizontal_lines_poly_1:
            x1 = p1[0]
            y1 = p1[1]
            x2 = p2[0]
            y2 = p2[1]
            condition_1 = (y == y1)
            condition_2 = (x1 < x < x2) or (x1 > x > x2)
            if (condition_1) and (condition_2):
                horizontal_points_poly_1.append(point)
                break
        for (k1, k2) in vertical_lines_poly_1:
            x1 = k1[0]
            y1 = k1[1]
            x2 = k2[0]
            y2 = k2[1]
            condition_1 = (x == x1)
            condition_2 = (y1 < y < y2) or (y1 > y > y2)
            if (condition_1) and (condition_2):
                vertical_points_poly_1.append(point)
                break

    # Check which set L1_projection_points belong to and add points to the respective list
    for point in L1_projection_points:
        x = point[0]
        y = point[1]
        for (p1, p2) in horizontal_lines_poly_1:
            x1 = p1[0]
            y1 = p1[1]
            x2 = p2[0]
            y2 = p2[1]
            condition_1 = (y == y1)
            condition_2 = (x1 < x < x2) or (x1 > x > x2)
            if (condition_1) and (condition_2):
                horizontal_points_poly_1.append(point)
                break
        for (k1, k2) in vertical_lines_poly_1:
            x1 = k1[0]
            y1 = k1[1]
            x2 = k2[0]
            y2 = k2[1]
            condition_1 = (x == x1)
            condition_2 = (y1 < y < y2) or (y1 > y > y2)
            if (condition_1) and (condition_2):
                vertical_points_poly_1.append(point)
                break
```

```
#print(horizontal_points_poly_1)
    #print(vertical_points_poly_1,"\n")
    # Iterate through each point in horizontal_points_poly_1
    for point1 in horizontal_points_poly_1:
        x1 = point1[0]
        y1 = point1[1]
        check = 0
        # Check if there exists a pair in horizontal_lines_poly_2
        matching_pairs = {(p1, p2) for (p1, p2) in horizontal_lines_poly_2 if ((p1[0] <= x1 <= p2[0]) or (p1[0] >= x1 >= p2[0]))}

        # Check conditions for matching_point
        for p1, p2 in matching_pairs:
            check = 0
            matching_point = (x1, p1[1])
            x_matching = matching_point[0]
            y_matching = matching_point[1]

            # Check conditions for horizontal_lines_poly_2
            for (x_p1, y_p1), (x_p2, y_p2) in horizontal_lines_poly_2:
                condition_1 = (x_p1 < x_matching < x_p2) or (x_p1 > x_matching > x_p2) or (x_p1 == x_matching) or (x_matching == x_p2)
                condition_2 = (y1 < y_p1 < y_matching) or (y1 > y_p1 > y_matching)
                if (condition_1) and (condition_2):
                    check=1
            # Check conditions for horizontal_lines_poly_1
            for (x_p1, y_p1), (x_p2, y_p2) in horizontal_lines_poly_1:
                condition_1 = (x_p1 < x_matching < x_p2) or (x_p1 > x_matching > x_p2) or (x_p1 == x_matching) or (x_matching == x_p2)
                condition_2 = (y1 < y_p1 < y_matching) or (y1 > y_p1 > y_matching)
                if (condition_1) and (condition_2):
                    check=1
            if (check==0):
                connectable_points.add((point1, matching_point))
        #print("h cp: ",connectable_points,"\n")
    # Iterate through each pair in vertical_lines_poly_1

    for point1 in vertical_points_poly_1:
        x1 = point1[0]
        y1 = point1[1]
        check = 0
        # Check if there exists a pair in vertical_lines_poly_2
        matching_pairs = {(p1, p2) for (p1, p2) in vertical_lines_poly_2 if (p1[1] <= y1 <= p2[1]) or (p1[1] >= y1 >= p2[1])}

        # Check conditions for matching_point
        for p1, p2 in matching_pairs:
            matching_point = (p1[0], y1)
            x_matching = matching_point[0]
            y_matching = matching_point[1]

            # Check conditions for vertical_lines_poly_2
            for (x_p1, y_p1), (x_p2, y_p2) in vertical_lines_poly_2:
                condition_1 = (y_p1 < y_matching < y_p2) or (y_p1 > y_matching > y_p2) or (y_p1 == y_matching) or (y_matching == y_p2)
                condition_2 = (x1 < x_p1 < x_matching) or (x1 > x_p1 > x_matching)
                if (condition_1) and (condition_2):
                    check=1
            # Check conditions for vertical_lines_poly_1
            for (x_p1, y_p1), (x_p2, y_p2) in vertical_lines_poly_1:
                condition_1 = (y_p1 < y_matching < y_p2) or (y_p1 > y_matching > y_p2) or (y_p1 == y_matching) or (y_matching == y_p2)
                condition_2 = (x1 < x_p1 < x_matching) or (x1 > x_p1 > x_matching)
                if (condition_1) and (condition_2):
                    check=1
            if (check == 0):
                connectable_points.add((point1, matching_point))
        #print("v cp: ",connectable_points,"\n")

    return connectable_points
```

ΥΛΟΠΟΙΗΣΗ ΤΟΥ ΑΛΓΟΡΙΘΜΟΥ ΣΕ ΚΩΔΙΚΑ (12)

- **def find_F1(points_pairs_set, neighbor_set)**

Υπολογισμό τον τύπο F_1. Δηλαδή, προσθέτει την συνολική απόσταση των σημείων, με βάση την απόλυτη διαφορά των συντεταγμένων τους, και την L_1 -απόσταση του καθενός, από τον απώτερο γείτονα.

- **def remove_duplicates(Bridge_Alg1)**

Θέτει την λίστα Bridge_Alg1 ως περιεχόμενο άλλης μεταβλητής, και αμέσως μετά την επαναπροσδιορίζει, ώστε να αφαιρεθούν κατά την διαδικασία, τα διπλά στοιχεία.

- **def sort_list(list)**

Ταξινομεί τα ζευγάρια, αντιγράφοντας τα σε, και στη συνέχεια από, μια προσωρινή λίστα.

- **def find_F1_for_list(poly, connectable_points)**

Δημιουργεί μια λίστα για τα F_1 και καλεί την μέθοδο που τα υπολογίζει, και τα εντάσσει.

- **def F1_min(F1_list_P, F1_list_Q, connectable_points_P, connectable_points_Q)**

Βρίσκει και επιστρέφει την μικρότερη τιμή στην λίστα των F_1 και την επιστρέφει σε συνδυασμό με το αντίστοιχο ζεύγος σημείων, σε μια μεταβλητή πίνακα 3 θέσεων.

```python
def find_F1(points_pairs_set, neighbor_set):

    point1 = points_pairs_set[0]
    point2 = points_pairs_set[1]
    # the variants that hold the point with the mu price and its corresponding point
    F1_list_prices = set()
    # first find the distance of the 2 points
    distance_horizontal = abs(point1[0] - point2[0])
    distance_vertical = abs(point1[1] - point2[1])
    max_path = find_most_distant_neighbor_path(point1, neighbor_set)
    # add the 3 values you found
    total_distance = distance_horizontal + distance_vertical + max_path
    # add to the list
    F1_list_prices.add(total_distance)
    # return the price, since we have the points on a list
    return F1_list_prices


def remove_duplicates(Bridge_Alg1):

    # Convert Bridge_Alg1 back to a list (removing duplicates)
    unique_bridge_set = sort_list(Bridge_Alg1)
    Bridge_Alg1 = sort_list(unique_bridge_set)
    return Bridge_Alg1


def sort_list(list):

     # Create an empty list to store the sorted pairs
    sorted_pairs = []
    # Iterate through each pair in Bridge_Alg1 and append the sorted pair to the list
    for pair in list:
        sorted_pair = tuple(sorted(pair))
        sorted_pairs.append(sorted_pair)
    # Create a set from the list to remove duplicates
    unique_list_set = set(sorted_pairs)
    return unique_list_set
```

```python
def find_F1_for_list(poly, connectable_points):

    # Create an empty list to store the results
    F1_list = []
    # Iterate through each pair in connectable_points_P and append the result to the list
    for pair in connectable_points:
        result = find_F1(pair, poly)
        F1_list.append(result)
    return F1_list


def F1_min(F1_list_P, F1_list_Q, connectable_points_P, connectable_points_Q):

    # Find the minimum price and its corresponding pair
    zipped_values = zip(F1_list_P, connectable_points_P)
    min_value_pair = min(zipped_values, key=lambda x: x[0])
    F1_min_P_price = min_value_pair[0]
    F1_min_P_point = min_value_pair[1]
    zipped_values_Q = zip(F1_list_Q, connectable_points_Q)
    min_value_pair_Q = min(zipped_values_Q, key=lambda x: x[0])
    F1_min_Q_price = min_value_pair_Q[0]
    F1_min_Q_point = min_value_pair_Q[1]

    # Compare the minimum prices and select the overall minimum
    if (F1_min_P_price < F1_min_Q_price):
        F1_min_price = F1_min_P_price
        F1_min_point = F1_min_P_point
    else:
        F1_min_price = F1_min_Q_price
        F1_min_point = F1_min_Q_point

    # Extract the bridge price from the result
    F1_min_price = F1_min_price.pop()
    result = (F1_min_price, F1_min_point[0], F1_min_point[1])
    return result
```

ΠΑΡΑΔΕΙΓΜΑ ΕΚΤΕΛΕΣΗΣ ΤΟΥ ΚΩΔΙΚΑ

```
# CASE 1:
poly_P = [(0,5.5),(1,5.5), (1,2), (2,2), (2,3),
(3,3), (3,2), (4,2), (4,1), (0,1)]
poly_Q = [(3,6), (4,6), (4,7), (6,7), (6,8),
(7,8), (7,5), (6,5), (6,4), (4,4), (4,5), (3,5)]
find_Bridge(poly_P, poly_Q)
```

CASE 1:

bridge price: 7

p: (3, 5)

q: (1, 5)

```
# CASE 2:
poly_P = [(0, 0),(0, 5),(2, 5),(2, 3),(6, 3),(6,
4),(7, 4),(7, 2),(8, 2),(8, 0)]
poly_Q = [(10, 7), (10, 8), (12, 8), (12, 10),
(14, 10), (14, 8), (15, 8), (15, 7)]
find_Bridge(poly_P, poly_Q)
```

CASE 2:

bridge price: 7.5

p: (8, 2)

q: (10, 7.5)

r: (8, 7.5

ΕΠΙΛΟΓΟΣ

Υλοποιήσαμε σε κώδικα στη γλώσσα προγραμματισμού python, έναν αλγόριθμο για την αναζήτηση της μικρότερης γέφυρας ανάμεσα σε 2 ορθογώνια πολύγωνα, τα οποία μπορούν να συνδεθούν από σημεία στις ακμές του καθενός, είτε με μια κάθετη και μια οριζόντια ευθεία, που τέμνονται κάθετα, είτε μόνο μια εξ αυτών, ανάλογα με την περίπτωση στην οποία υπόκεινται τα πολύγωνα με βάση τη θέση τους στο πεδίο, με βάση τα βήματα του αλγορίθμου στο άρθρο «An optimal algorithm for constructing an optimal bridge between two simple rectilinear polygons», του D.P. Wang (2001).

ΒΙΒΛΙΟΓΡΑΦΙΑ

[1]	D.P. Wang. An optimal algorithm for constructing an optimal bridge between two simple rectilinear polygons. Information Processing Letters, Volume 79 (Issue 5), pp. 229-236, 2001
[2]	Kim, S., Shin, CS. Computing the Optimal Bridge between Two Polygons. Theory Comput. Systems 34, 337–352 (2001)