ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

Τμήμα Μηχανικών Η/Υ Τηλεπικοινωνιών και Δικτύων

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΤΙΤΛΟΣ: ΥΛΟΠΟΙΗΣΗ ΚΑΙ ΕΚΤΙΜΗΣΗ ΑΛΓΟΡΙΘΜΩΝ ΣΤΟ ΠΕΡΙΒΑΛΛΟΝ LEDA

ΣΑΚΕΛΛΑΡΙΟΣ ΧΡΗΣΤΟΣ

ΕΠΙΒΛΕΠΩΝ ΚΑΘΗΓΗΤΗΣ: ΠΑΝΑΓΙΩΤΗΣ ΜΠΟΖΑΝΗΣ

ΑΚΑΔΗΜΑΙΚΟ ΕΤΟΣ: 2008-09

ΠΕΡΙΛΗΨΗ

Αντικείμενο της εργασίας είναι αφενός η υλοποίηση γνωστών αλγορίθμων, με τη γλώσσα προγραμματισμού C++ κάνοντας χρήση της βιβλιοθήκης LEDA, αφετέρου η εκτίμηση αυτών συγκρίνοντας τους με τις ήδη υπάρχουσες, έτοιμες συναρτήσεις του πακέτου LEDA με απώτερο σκοπό την ανάδειξη της αποτελεσματικότητά του. Συγκεκριμένα, υλοποιούνται πλήθος γνωστών αλγορίθμων και κατόπιν γίνονται χρονικές συγκρίσεις με τις αντίστοιχες συναρτήσεις της LEDA για διάφορα μεγέθη εισόδου. Οι εκτιμήσεις αυτές δε, αναδεικνύουν την καλύτερη ποιότητα – αποτελεσματικότητα των συναρτήσεων του πακέτου LEDA σε σχέση με άλλες υλοποιήσεις πολλών χρηστών, των ίδιων αλγορίθμων.

ΠΕΡΙΕΧΟΜΕΝΑ

ΕΙΣΑΓΩΓΗ

Στόχος της εργασίας είναι η υλοποίηση και εκτίμηση γνωστών αλγορίθμων, στο περιβάλλον LEDA. Τα αποτελέσματα δε των εκτιμήσεων αυτών, μπορούν να χρησιμοποιηθούν αφενός στην καταγραφή συμπεριφοράς κάθε αλγορίθμου ξεχωριστά, αφετέρου στην παραγωγή κρίσιμων πορισμάτων όσον αφορά την αποτελεσματικότητα της βιβλιοθήκης LEDA.

Η μεθοδολογία που ακολουθείται για κάθε αλγόριθμο είναι η εξής: Αρχικά, γίνεται η ανάλυση λειτουργίας του αλγορίθμου, δηλαδή εξηγείται τι ακριβώς κάνει ο αλγόριθμος ή ποιο πρόβλημα λύνει. Για μεγαλύτερη κατανόηση, δίνεται και ένα παράδειγμα – στιγμιότυπο τρεξίματος του αλγορίθμου. Έπειτα, παρουσιάζεται μια υλοποίηση του αλγορίθμου σε γλώσσα προγραμματισμού C++, με χρήση της βιβλιοθήκης LEDA. Τέλος, γίνεται η εκτίμηση του αλγορίθμου, παραθέτοντας διάγραμμα στο οποίο απεικονίζονται οι χρόνοι των διαφόρων υλοποιήσεων, αν τυχόν υπάρχουν. Ακολουθεί η επεξήγηση του διαγράμματος ώστε να γίνει σαφές γιατί η εκάστοτε υλοποίηση παρουσιάζει τα αντίστοιχα αποτελέσματα.

Στο **ΚΕΦΑΛΑΙΟ 1**, γίνεται μία γενική επισκόπηση της βιβλιοθήκης LEDA, παρουσιάζοντας τι ακριβώς είναι, που χρησιμοποιείται, οι τεχνικές - λειτουργικές προδιαγραφές και τα πλεονεκτήματά της. Τέλος, δίνονται παραδείγματα δομών και συναρτήσεων αυτών, ώστε ο αναγνώστης να πάρει μια γεύση από το περιβάλλον της LEDA.

Στο **ΚΕΦΑΛΑΙΟ 2**, ακολουθεί το βασικό μέρος της εργασίας, ήτοι η παρουσίαση κάθε αλγορίθμου με τη μεθοδολογία που αναφέρθηκε παραπάνω.

Στο **ΚΕΦΑΛΑΙΟ 3**, δίνονται τα τελικά συμπεράσματα της όλης διαδικασίας, τα οποία είναι αποτέλεσμα των υλοποιήσεων και των συγκρίσεων που έγιναν σ' αυτές. Δίνονται πλεονεκτήματα ή μειονεκτήματα δομών και αναφέρεται κατά πόσο εύκολη είναι η υλοποίηση αλγορίθμων σε περιβάλλον LEDA.

ΚΕΦΑΛΑΙΟ 1. ΓΕΝΙΚΗ ΕΠΙΣΚΟΠΗΣΗ ΤΗΣ LEDA

1.1. ΤΙ ΕΙΝΑΙ Η LEDA

Η LEDA είναι βιβλιοθήκη της γλώσσας προγραμματισμού C++ για αποτελεσματικούς τύπους δεδομένων και αλγορίθμους. Προσφέρει αλγόριθμους έχοντας μεγάλο βάθος γνώσεως στα πεδία των προβλημάτων γραφημάτων και δικτύων, γεωμετρικών υπολογισμών, συνδυαστικών προβλημάτων και άλλα. Είναι υλοποιημένη ακολουθώντας την αντικειμενοστρεφή προσέγγιση. Χρησιμοποιείται σε πεδία εφαρμογών όπως τηλεπικοινωνίες, GIS, VLSI σχεδιασμό, χρονοπρογραμματισμό, υπολογιστική βιολογία και σε οποιοδήποτε πεδίο σχετικό με σχεδιασμό υπολογιστικών προβλημάτων. Προσφέρει πλήθος αλγορίθμων δίνοντας τη δυνατότητα διαχείρισης με μεγάλη ευκολία αντικειμένων όπως γραφήματα, σειρές, λεξικά, δέντρα, ροές , συντομότερα μονοπάτια και πολλά άλλα. Έτσι, είναι εύκολο να εφαρμοστεί σε όλα τα software projects που χρησιμοποιούν αντικειμενοστρεφή προγραμματισμό. Μάλιστα, διατίθεται για διαφορετικά λειτουργικά συστήματα.

1.2. ΤΕΧΝΙΚΕΣ ΠΡΟΔΙΑΓΡΑΦΕΣ

Η LEDA παρέχει δομές δεδομένων και αλγόριθμους που επιτρέπουν είτε στους χρήστες είτε σε δεδομένα να τα χρησιμοποιούν δυναμικά, αλλάζοντάς τους παραμέτρους. Προσφέρει επαναληπτικές διαδικασίες τόσο για δομές δεδομένων όσο και για τύπους αριθμητικών δεδομένων. Η εγκυρότητα των αποτελεσμάτων είναι πάντοτε εξασφαλισμένη. Μάλιστα, αρκετοί αλγόριθμοι αποδεικνύουν την εγκυρότητα αυτή με απλές και κατανοητές ρουτίνες που στηρίζονται σε βασικά θεωρήματα μαθηματικών. Το error handling επιτυγχάνεται από C++ exceptions ή μέσω standard error output. Τέλος, η LEDA παρέχει τη δικιά της διαχείριση μνήμης (υπάρχει επιλογή ενεργοποίησης η όχι), η οποία αυξάνει την χρονική και χωρική (μνήμη) αποτελεσματικότητα των τύπων δεδομένων.

1.3. ΛΕΙΤΟΥΡΓΙΚΕΣ ΠΡΟΔΙΑΓΡΑΦΕΣ

Η LEDA περιέχει τύπους περιεχομένων δεδομένων όπως λίστες, πίνακες, χάρτες, λεξικά, ουρές προτεραιοτήτων αλλά και βασικούς τύπους δεδομένων όπως αλγόριθμους αναζήτησης ή ταξινόμησης και φυσικά γεωμετρικούς. Προσφέρει μια μεγάλη ποικιλία αλγορίθμων γραφημάτων και δικτύων όπως αναζήτηση κατά βάθος, αναζήτηση κατά πλάτος, συντομότερα μονοπάτια, ελάχιστα επικαλύπτοντα δέντρα, στοίχιση απλή ή βεβαρημένη, ροές και άλλους πολλούς. Επιπλέον, περιέχει εξειδικευμένα μοντέλα καλύπτοντας πεδία όπως η κρυπτογραφία, συμπίεση, ισομορφισμό γραφημάτων – υπογραφημάτων και στοίχιση συμβολοσειρών. Φυσικά όλες οι κλάσεις και αλγόριθμοι της LEDA είναι τεκμηριωμένοι με λεπτομέρεια, οι οποίες τεκμηριώσεις είναι online προσβάσιμες στο επίσημο site της LEDA.

1.4. ΠΛΕΟΝΕΚΤΗΜΑΤΑ ΤΗΣ LEDA

1.4.1 Ευκολία στη χρήση

Η LEDA είναι εύκολη στη χρήση ακόμα και σε καθημερινή διαδικασία ανάπτυξης. Έχει ενορατικό class interface διευκολύνοντας τον χρήστη. Ονόματα κλάσεων, συναρτήσεων και αλγορίθμων είναι καλά τεκμηριωμένα. Επίσης, η πολυπλοκότητα χρόνου και χώρου είναι σαφές ορισμένη.

1.4.2. Αποτελεσματικότητα

Παρέχει μεγάλο εύρος από αποτελεσματικούς αλγορίθμους, πολλούς από τους οποίους έχει αποδειχθεί να συγκλίνουν ασυμπτωτικά. Δίνει τα πιο αξιόπιστα αποτελέσματα σε σχέση με άλλες παρόμοιες βιβλιοθήκες. Η ενασχόλησή της με εξειδικευμένες αλγοριθμικές μεθοδολογίες και σωστά υλοποιημένους αλγορίθμους και δομές δεδομένων δημιουργεί πρόσθετη αξία και καινοτομίες.

1.4.3. Αξιοπιστία

Οι υλοποιημένοι αλγόριθμοι της LEDA είναι αξιόπιστοι καθώς τα φέρνουν εις πέρας παράγοντας σωστά αποτελέσματα ακόμα και με «κακές» εισόδους. Παρέχοντας διαφορετικό αριθμών πυρήνων, αποτρέπονται προβλήματα που δημιουργούνται από πεπερασμένης ακρίβειας υπολογισμούς. Η LEDA δοκιμάζεται επιτυχώς κατά τη διάρκεια μιας

αξιόπιστης αντικειμενοστραφούς ανάπτυξης καθώς και από το μεγάλο πλήθος χρηστών της. Τα αποτελέσματα των αλγορίθμων είναι πάντοτε εγγυημένα σωστά. Επιπροσθέτως, η LEDA εκτελεί και run-time error έλεγχο.

1.4.4. Κλιμάκωση

Η LEDA σα βιβλιοθήκη είναι εύκολα επεκτάσιμη και «τρέχει» σε διάφορα λειτουργικά συστήματα όπως Windows, Unix ή Linux με τους περισσότερους compilers της C++ διαθέσιμους. Είναι ανεξάρτητη από βιβλιοθήκες λογισμικού όπως η MFC ή STL και υπερτερεί αυτών σεβόμενο τη συνέχεια του προϊόντος.

1.5. ΠΑΡΑΔΕΙΓΜΑΤΑ ΣΥΝΑΡΤΗΣΕΩΝ – ΚΛΑΣΕΩΝ ΤΗΣ LEDA

Παρακάτω αναφέρονται επιγραμματικά οι κυριότεροι τύποι δεδομένων της βιβλιοθήκης καθώς και μερικές σημαντικές συναρτήσεις αυτών.

- **One Dimensional Arrays**

  1. array<E> A(int a, int b) : Δημιουργεί πίνακα A στοιχείων τύπου E, με όρια του πίνακα [a,…,b].
  2. E& A[int x] : Επιστρέφει το A(x).
  3. void A.copy(int x, int y) : Θέτει A(x)=A(y).
  4. void A.swap(int i, int j) : Ανταλλάζει τις τιμές των A(i) και A(j).
  5. void A.sort() : Εκτελεί τη γραμμική διάταξη των στοιχείων του A.

- **Stacks**

  1. stack<E> S : Δημιουργία στοίβας S στοιχείων τύπου Ελ
  2. top() : Επιστρέφει το κορυφαίο στοιχείο της στοίβας.
  3. push(E x) : Εισάγει το x στην κορυφή της στοίβας.

- **Queues**

  1. queue<E> Q : αρχικοποιεί μία κενή ουρά.
  2. pop() : διαγράφει και επιστρέφει το κορυφαίο στοιχείο.
  3. append(x) : εισάγει το x στο τέλος.

- **Bounded Stacks, Bounded Queues**

    1. b_stack<E> S(n) :αρχικοποιεί μία κενή στοίβα, μεγέθους μέχρι n στοιχείων.
    2. b_queue<E> Q(n): αρχικοποιεί μία κενή ουρά, μεγέθους μέχρι n στοιχείων.

- **Linear Lists**

    1. list<E> L:αρχικοποιεί μία κενή λίστα.
    2. head()/front() : το πρώτο στοιχείο τύπου E.
    3. succ()/pred() : το επόμενο/προηγούμενο list_item.
    4. L.sort() : ταξινομεί την λίστα βάσει E.
    5. **forall_items**(*it*, *L*) : iterator των list_item

- **Singly Linked Lists**

    1. slist<E> L : αρχικοποιεί μία κενή μονή λίστα.
    2. push(x)/append(x) : εισάγει στην αρχή/τέλος το στοιχείο x.
    3. cyclic_succ() : το κυκλικά επόμενο slist_item.

- **Sets**

    1. set<E>S: αρχικοποιεί ένα κενό σύνολο.
    2. insert(x): εισάγει το x στο σύνολο.
    3. member(x): επιστρέφει εάν το x είναι ή όχι μέλος του συνόλου.
    4. **forall**(*x*, *S*) : iterator των περιεχομένων του S.

- **Dynamic Trees**

1. dynamic_trees D**:** αρχικοποιεί ένα κενό δυναμικό βεβαρημένων ακμών δένδρο με ρίζα: οι κορυφές και οι ακμές μπορούν να έχουν και επιπλέον πληροφορία.
2. D.root(vertex v)*:* επιστρέφει την ρίζα του δένδρου όπου ανήκει η v.
3. D.cost(vertex v) *:* επιστρέφει το βάρος της ακμής v προς πατέρα της v.

- **Graphs**

1. graph G : δήλωση ενός γραφήματος
2. indeg(node v): επιστρέφει τον αριθμό των ακμών που καταλήγουν στη v.
3. target(edge e): επιστρέφει την κορυφή στην οποία καταλήγει η ακμή e.
4. number_of_nodes(): επιστρέφει τον αριθμό των κορυφών του γραφήματος.
5. adj_nodes(node v): επιστρέφει λίστα των κορυφών που συνδέονται μέσω ακμής με την κορυφή v.
6. is_directed(): επιστρέφει TRUE αν το γράφημα είναι κατευθυνόμενο.
7. new_edge(node v, node w): δημιουργεί μια νέα ακμή μεταξύ των κορυφών v, w.
8. **forall_nodes**(*v*, *G*): iterator των κορυφών του γραφήματος.
9. **forall_edges**(*e*, *G*): iterator των ακμών του γραφήματος.

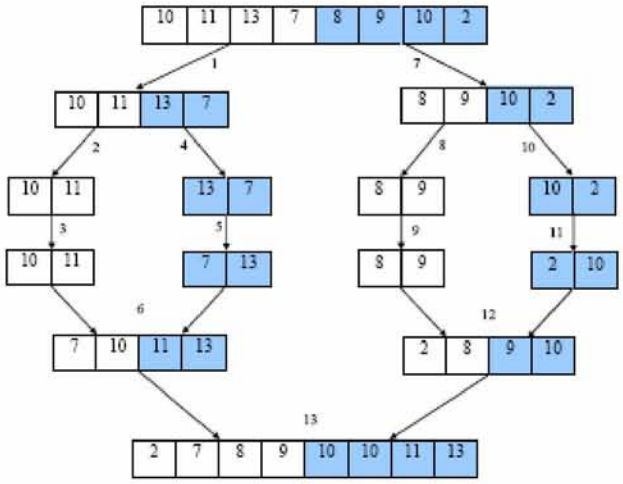
Και άλλα πολλά..

ΚΕΦΑΛΑΙΟ 2. ΑΝΑΛΥΣΗ ΚΑΙ ΣΥΓΚΡΙΣΗ ΑΛΓΟΡΙΘΜΩΝ

2.1. ΑΛΓΟΡΙΘΜΟΣ MERGESORT

2.1.1. Ανάλυση λειτουργίας αλγορίθμου

Ο αλγόριθμος mergesort (ταξινόμηση συγχωνεύσεως), παίρνει ως είσοδο στοιχεία ενός ολικά διατεταγμένου συνόλου και επιστρέφει ως έξοδο το σύνολο διατεταγμένο. Ακολουθεί τη γνωστή μέθοδο «διαίρει και βασίλευε» ως εξής: Στην αρχή χωρίζει την ακολουθία σε δύο ίσα τμήματα (διαίρει), έπειτα τα ταξινομεί ανεξάρτητα (βασίλευε) και τέλος τα συγχωνεύει (συνδύασε). Η διαδικασία έχει αναδρομικό χαρακτήρα. Ως προς εξήγηση αυτού, αν για παράδειγμα η είσοδος του αλγορίθμου είναι ένας πίνακας 8 ακεραίων (**βλ. Σχήμα 2.1.1)**, στην αρχή ο πίνακας θα χωριστεί στη μέση, σε δύο ίσα τμήματα 4 ακεραίων τα οποία θα ταξινομηθούν αναδρομικά (πρώτα το ένα και μετά το άλλο). Όταν τελειώσουν οι αναδρομικές κλήσεις, τα ταξινομημένα τμήματα συγχωνεύονται.



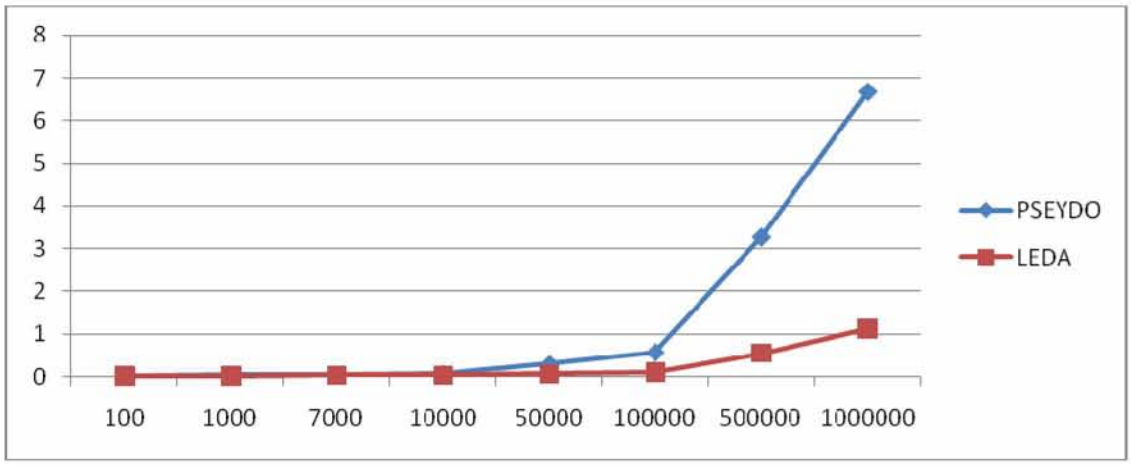
Σχήμα 2.1.1 : Στιγμιότυπο του mergesort.

2.1.2. Υλοποίηση αλγορίθμου με χρήση βιβλιοθήκης LEDA

```
#include <iostream>
#include <cstdio>
#include <C:LEDA\core\array.h>


using namespace leda;
using std::cout;
using std::cin;
using std::endl;
int right,left;
void mergeSort(array<int>& A,int left, int right, array<int>& B);

int main() {

	return 0;
}

void mergeSort(array<int>& A,int left, int right, array<int>& B) {
	int middle=(right+left)/2;
	//diairei
	if(left<middle) mergeSort(A,left,middle,B);
	if((middle+1)<right) mergeSort(A,middle+1,right,B);
	//vasileue kai sindiaze
	for(int i=left;i<middle+1;i++)
	{
		B[i]=A[i];
	}
	int i=left;
	for(int r=middle;r<right;r++)
	{
		B[right+middle-r]=A[r+1];
	}
	int r=right;
	for(int k=left;k<=right;k++) //sigxwneusi diatetagmenwn
upopinakwn
		if(B[r]<B[i])
			A[k]=B[r--];
		else
			A[k]=B[i++];
}
```

2.1.3. Εκτίμηση αλγορίθμου

Διάγραμμα 1



Σχήμα 2.1.2 : Σύγκριση υλοποιημένου κώδικα MERGESORT vs κώδικα LEDA

2.1.3.1. Επεξήγηση διαγράμματος

Ο οριζόντιος άξονας του διαγράμματος δείχνει το εκάστοτε μέγεθος εισόδου του αλγορίθμου το οποίο είναι το μέγεθος των πινάκων που παίρνει ως όρισμα ο αλγόριθμος. Ο κάθετος δείχνει το χρόνο τρεξίματος του αλγορίθμου σε δευτερόλεπτα για το αντίστοιχο μέγεθος εισόδου στον αλγόριθμο. Είναι εμφανές η ανωτερότητα της υλοποίησης της LEDA (βλ. Κόκκινη γραμμή) έναντι της απλής υλοποίησης (βλ. Μπλε γραμμή). Η διαφορά χρόνου μεταξύ δύο συνεχόμενων διαφορετικών τιμών εισόδου , είναι πολύ μεγαλύτερη στην απλή υλοποίηση από αυτήν της LEDA δείχνοντας την ικανότητα αντοχής της σε μεγάλο μέγεθος εισόδου χωρίς ωστόσο τη μεγάλη αύξηση στο χρόνο τρεξίματος. Η διαφορά αυτή οφείλεται στην καλύτερη διαχείριση μνήμης της LEDA καθώς και στην απλούστερη υλοποίησή της η οποία δεν περιλαμβάνει αναδρομικές κλήσεις.

2.2. ΑΛΓΟΡΙΘΜΟΣ KNAPSACK

2.2.1. Ανάλυση αλγορίθμου

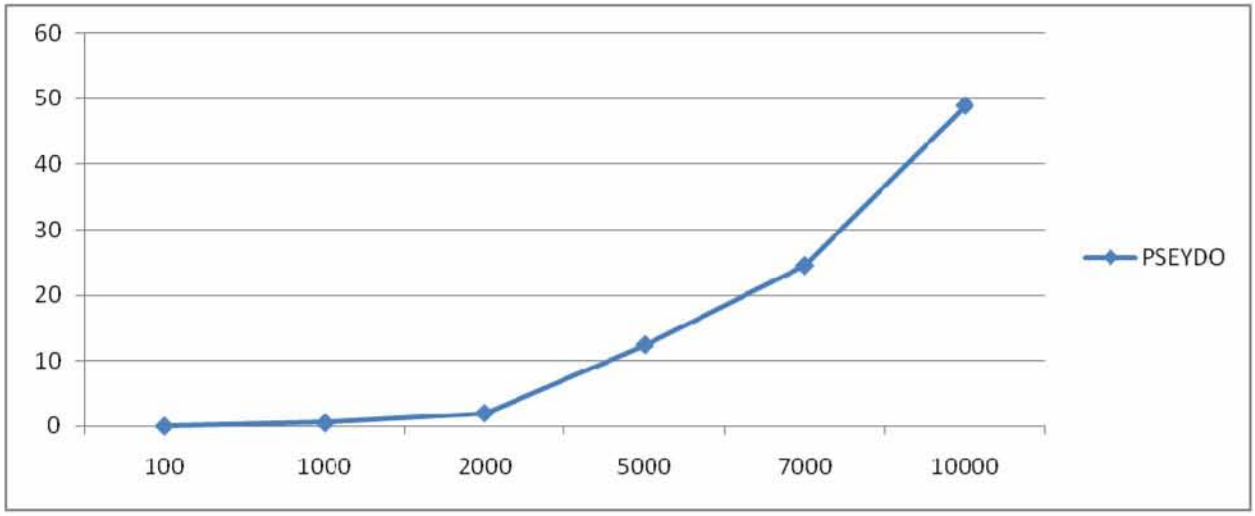
Ο αλγόριθμος KnapSack καλείται να λύσει το πρόβλημα του σακιδίου το οποίο είναι το εξής: Δοθέντος ενός συνόλου S = $\{a_1,a_2,\ldots,a_n\}$ n αντικειμένων, με κάθε αντικείμενο a_i να έχει ένα βάρος w_i και μία αξία v_i , καθώς και μίας τιμής W, ζητείται η εύρεση ενός υποσυνόλου S' του S, μέγιστης συνολικής αξίας Σv_j με a_j να ανήκει στο S', και με συνολικό βάρος Σw_j, a_i ανήκει στο S', να μην ξεπερνά την τιμή W. Για παράδειγμα, έστω 4 αντικείμενα a_1, a_2, a_3 και a_4 με βάρη και αξίες w_1=2, w_2=3, w_3=4, w_4=5 , v_1=4, v_2=3, v_3=2 και v_4=1. Αν η τιμή W ισούται με 7 τότε η μέγιστη συνολική αξία 7 επιλέγοντας ως S' = $\{a_1,a_2\}$.

2.2.2. Υλοποίηση αλγορίθμου με χρήση βιβλιοθήκης LEDA

```
#include <iostream>
#include <cstdio>
#include <C:LEDA\core\array.h>
#include <C:LEDA\core\array2.h>

using namespace leda;
using std::cout;
using std::cin;
using std::endl;

class item {
public:
	int w,v;


};

int knapSack(const array<item> S,int W,array2<int>& A);
int main() {
	return 0;
}

int knapSack(const array<item> S,int W,array2<int>& A) {
	array2<int> V(0,S.size()-1,0,W);
	int withk, withoutk;
	for(int j=0;j<=W;j++)
	{
		V(0,j)=0;
	}
	for(int k=1;k<S.size();k++){
		for(int w=0;w<=W;w++){
			withoutk=V(k-1,w); //lisi dixws to k-sto antikeimeno
			if(w>=S[k].w)
				withk=V(k-1,w-S[k].w)+S[k].v; //lisi me to k-
sto
			else
				withk=-MAXINT; //meion apeiro
			V(k,w)=max(withk,withoutk); //apofasi ti simferei
			A(k,w)=(withk<withoutk ? 0:1); //voithitikos pinakas
gia tin
			}                              //anaparagwgi tis
lisews
		}
	return V(S.size()-1,W);
	}
```

2.2.3. Εκτίμηση αλγορίθμου

Διάγραμμα 2



Σχήμα 2.2.1: Εκτίμηση υλοποιημένου κώδικα knapsack σε περιβάλλον LEDA

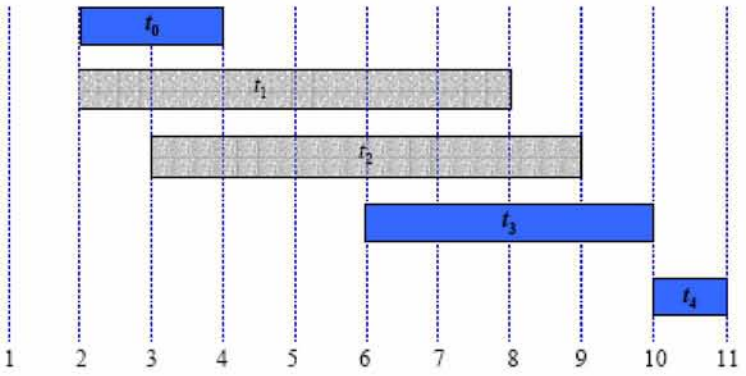
2.2.3.1. Επεξήγηση διαγράμματος

Ο οριζόντιος άξονας δείχνει το μέγεθος εισόδου στον αλγόριθμο, ήτοι ο αριθμός n στοιχείων που περνιούνται ως πίνακας n ακεραίων στη συνάρτηση KnapSack. Όπως φαίνεται και στο **σχήμα 2.2.1** παραπάνω, μετά από κάποιο ορισμένο μέγεθος εισόδου παρατηρείται μία σταθερή μεν αλλά έντονη δε αύξηση στο χρόνο τρεξίματος του αλγορίθμου κάτι που εξηγείται από τις πολλές συγκρίσεις και ενθέσεις με στοιχεία πινάκων.

2.3. ΑΛΓΟΡΙΘΜΟΣ TASKSCHEDULING

2.3.1. Ανάλυση αλγορίθμου

Ο αλγόριθμος καλείται να λύσει το πρόβλημα του προγραμματισμού εργασιών που ορίζεται ως εξής: Δοθέντος ενός συνόλου $T= \{t_0,t_1,\ldots,t_{n-1}\}$ n εργασιών τέτοιες ώστε κάθε t_j Ε Τ να χαρακτηρίζεται από το χρόνο ενάρξεώς της s_j και τον χρόνο περατώσεώς της $f_j>s_j$. Ένας μοναδικός πόρος r χρησιμοποιείται για να εκτελεστούν οι εργασίες και ανά πάσα χρονική στιγμή μπορεί να εκτελείται μία μόνο εργασία, μέχρι περατώσεώς της. Ζητείται λοιπόν, να βρεθεί το μέγιστο σύνολο εργασιών Τ' υποσύνολο του Τ που δύναται να εξυπηρετηθεί από τον πόρο r. Για παράδειγμα, στο **σχήμα 2.3.1**, έστω 5 εργασίες με τους αντίστοιχους χρόνους ενάρξεως και περατώσεώς τους: $t_0(2,4)$, $t_1(2,8)$, $t_2(3,9)$, $t_3(6,10)$ και $t_4(10,11)$. Η βέλτιστη χρονοδρομολόγηση που θα επιλέξει ο αλγόριθμος είναι να εκτελέσει πρώτα την t_0, έπειτα θ' απορριφθούν οι t_1 και t_2 διότι κατά το χρόνο ενάρξεώς τους ο πόρος είναι απασχολημένος, και έπειτα ακολουθεί η εκτέλεση των εργασιών t_3 και t_4.



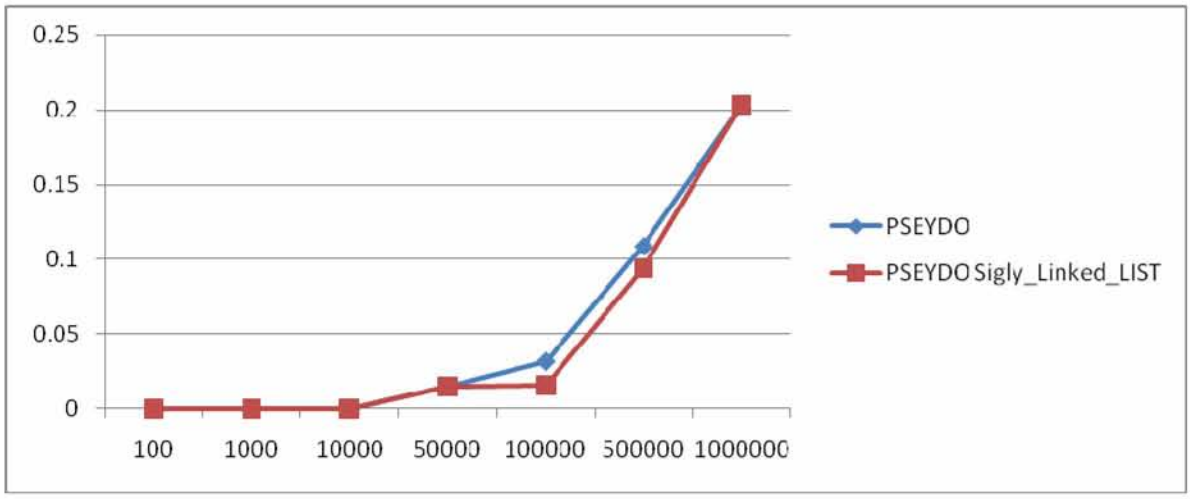
Σχήμα 2.3.1 : Παράδειγμα τρεξίματος TaskScheduling

2.3.2. Υλοποίηση αλγορίθμου με χρήση βιβλιοθήκης LEDA

```
#include <iostream>
#include <cstdio>
#include <C:LEDA\core\array.h>
#include <C:LEDA\core\list.h>
#include <C:LEDA\core\string.h>

class T {
public:
	//string name;
	int start;
	int end;
};
void taskScheduling(const array<T> task,list<T>& A);

int main() {
	return 0;
}

void taskScheduling(const array<T> task,list<T>& A) {
	A.push(task[0]);                                    //arxikopoiisi
tis listas me tin prwti ergasia
	int last=0;	//int k;                                        //i
teleutaia ergasia pou dromologeithike
	for(int k=1;k<task.size();k++){
		if(task[k].start>=task[last].end){  //den iparxei sigrousi
			A.push(task[k]);
			last=k;
		}
	}
}
```

2.3.3 Εκτίμηση αλγορίθμου

Διάγραμμα 3



Σχήμα 2.3.2 : Σύγκριση υλοποιημένου κώδικα σε περιβάλλον LEDA χρησιμοποιώντας διαφορετική δομή λίστας κάθε φορά.

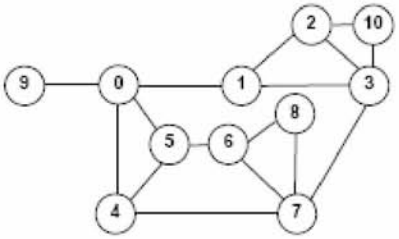
2.3.3.1. Επεξήγηση διαγράμματος

Ο οριζόντιος άξονας του παραπάνω **σχήματος 2.3.2** δείχνει τον αριθμό n των εργασιών ήτοι και το μέγεθος n του πίνακα είσοδο στον αλγόριθμο, ενώ ο κάθετος δείχνει τον αντίστοιχο χρόνο τρεξίματος του. Η πρώτη υλοποίηση γίνεται με χρήση Linear List (βλ. Μπλε γραμμή) ενώ η δεύτερη χρήση απλά συνδεδεμένης λίστας Singly Linked List (βλ. Κόκκινη γραμμή). Όπως φαίνεται και στο διάγραμμα, η χρήση της Singly Linked List αποδίδει ελαφρώς καλύτερα από αυτή της Linear List, κάτι που εξηγείται στη μικρότερη μνήμη που χρησιμοποιεί η πρώτη. Όσο όμως το μέγεθος εισόδου αυξάνει οι δύο υλοποιήσεις συγκλίνουν στους ίδιους χρόνους τρεξίματος.

2.4. ΑΛΓΟΡΙΘΜΟΣ DFS_MATRIX

2.4.1. Ανάλυση αλγορίθμου

Ο αλγόριθμος dfs Matrix εκτελεί διαπέραση ή διάβαση ενός γραφήματος και συγκεκριμένα εκτελεί αναζήτηση σε βάθος, εξού και το dfs (depth first searh). Χρησιμοποιεί πίνακα γειτνιάσεως ώστε να κρατάει τις ακμές του γραφήματος για κάθε κορυφή κι ένα πίνακα pre που δηλώνει τη σειρά προδιατάξεως μια κορυφής. Ξεκινώντας ο αλγόριθμος, επισκέπτεται αναδρομικά την πρώτη, κατά αύξουσα ονομασία, ανεξερεύνητη γειτονική κορυφή, συνδυάζοντας τις πληροφορίες διασυνδέσεως του πίνακα γειτνιάσεως και του πίνακα προδιατάξεως pre. Για παράδειγμα, με είσοδο στον αλγόριθμο το γράφημα του **σχήματος 2.4.1**, ο αλγόριθμος θα εξετάσει αρχικά την κορυφή 0 αυξάνοντας τον αριθμό προδιατάξεως του, έπειτα την ανεξερεύνητη κορυφή 1 μέσω του πίνακα γειτνιάσεως, κάνοντας τον αριθμό προδιατάξεως της κορυφής 1 κατά 1 μεγαλύτερο και ούτω καθεξής μέχρι να εξερευνηθούν όλες οι κορυφές του γραφήματος.



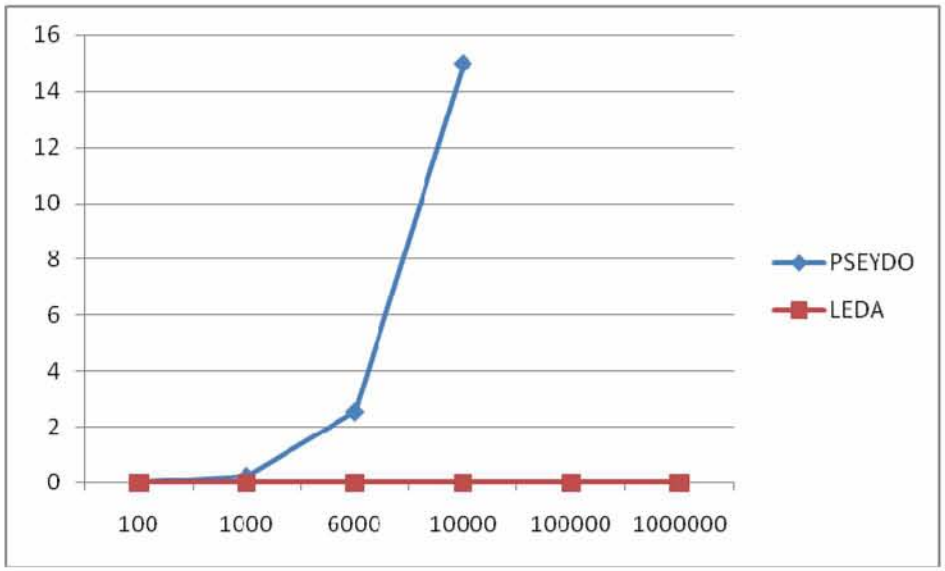
Σχήμα 2.4.1

2.4.2. Υλοποίηση αλγορίθμου με χρήση βιβλιοθήκης LEDA

```
#include <iostream>
#include <cstdio>
#include <C:LEDA\core\array.h>
#include <C:LEDA\graph\node_matrix.h>
#include <C:LEDA\graph\graph.h>


using namespace leda;
using std::cout;
using std::cin;
using std::endl;
static int order;

void dfsMatrix(const graph& G,node v,node_array<int>&
pre,node_matrix<int>& A);

int main() {

return 0;
}

void dfsMatrix(const graph& G,node v,node_array<int>&
pre,node_matrix<int>& A) {

      pre[v]=order++; //voithitikos pinakas prodiataksews
      node w;
      forall_nodes(w,G) {
            if(A(v,w)==1) //iparxei akmi
                  if(pre[w]==0) //i w den exei anakaliftei akomi
                        dfsMatrix(G,w,pre,A);
      }
      return;
}
```

2.4.3. Εκτίμηση αλγορίθμου

Διάγραμμα 4



Σχήμα 2.4.1 Σύγκριση υλοποιημένου κώδικα vs έτοιμου κώδικα LEDA

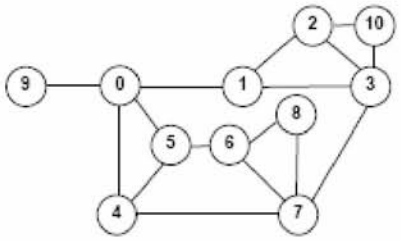
2.4.3.1. Επεξήγηση διαγράμματος

Ο οριζόντιος άξονας εκφράζει το μέγεθος εισόδου στον αλγόριθμο, δηλαδή την πολυπλοκότητα του γραφήματος G, και συγκεκριμένα τον αριθμό των κορυφών. Ο κάθετος άξονας δείχνει το χρόνο τρεξίματος για το αντίστοιχο μέγεθος εισόδου. Είναι πραγματικά τεράστια η διαφορά αποτελεσματικότητας των δύο αλγορίθμων. Ο dfs Matrix δείχνει να μην αντέχει καν την αύξηση του μεγέθους εισόδου καταγράφοντας υπερβολικούς χρόνους τρεξίματος, ενώ αντίθετα η έτοιμη συνάρτηση της LEDA " DFS(graph G, node s, bool reached), δείχνει να μην πτοείται καθόλου από το μέγεθος και πολυπλοκότητα του γραφήματος, τρέχοντας πάντα σε σχεδόν μηδενικό χρόνο. Η εξήγηση έγκειται στο ότι ο dfs Matrix εκτελεί πολλές συγκρίσεις σε στοιχεία πινάκων καθώς επίσης και αναδρομικές κλήσεις, ενώ η συνάρτηση της LEDA χρησιμοποιώντας καταλληλότερες δομές και διαχειριζόμενη καλύτερα τη μνήμη, επιφέρει εκπληκτικά αποτελέσματα.

2.5. ΑΛΓΟΡΙΘΜΟΣ DFS_LIST

2.5.1. Ανάλυση αλγορίθμου

Παρόμοια με τον dfsMatrix , ο αλγόριθμος εκτελεί αναζήτηση σε βάθος ενός δοθέντος γραφήματος, είσοδο στον αλγόριθμο. Χρησιμοποιεί όμως λίστα γειτνιάσεως αντί για πίνακα, ώστε να κρατάει τις ακμές του γραφήματος για κάθε κορυφή και τον πίνακα pre που κρατάει τη σειρά προδιατάξεως κάθε κορυφής. Ο αλγόριθμος ελέγχει απ' ευθείας εάν οι γειτονικές κορυφές της εκάστοτε εξεταζόμενης κορυφής v έχουν ανακαλυφθεί, και μόνο αυτές, σαρώνοντας την αντίστοιχη λίστα γειτνιάσεως της v. Στο **σχήμα 2.5.1** εξερευνώντας αρχικά την κορυφή 0, ελέγχει τη λίστα γειτνιάσεως της η οποία περιλαμβάνει τις κορυφές {1, 4, 5, 9} και διαλέγοντας την κορυφή 1, εξετάζει αν έχει εξερευνηθεί, αν όχι καλεί αναδρομικά τον αλγόριθμο για την κορυφή 1.



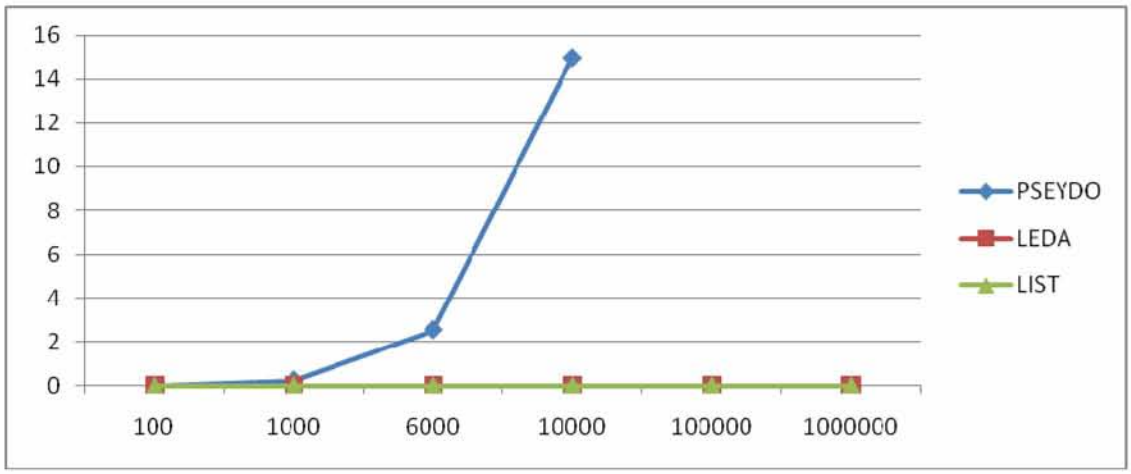
Σχήμα 2.5.1

2.5.2. Υλοποίηση αλγορίθμου με χρήση βιβλιοθήκης LEDA

```
#include <iostream>
#include <cstdio>
#include <C:LEDA\core\array.h>
#include <C:LEDA\graph\node_list.h>
#include <C:LEDA\graph\node_array.h>
#include <C:LEDA\graph\graph.h>
#include <C:LEDA\core\list.h>


using namespace leda;
using std::cout;
using std::cin;
using std::endl;
static int order;

void dfsList(const graph& G,node v,node_array<int>&
pre,node_array<list<node>>& L);

int main() {

	return 0;
}

void dfsList(const graph& G,node v,node_array<int>&
pre,node_array<list<node>>& L) {
	pre[v]=order++; //voithitikos pinakas prodiataksews
	node x;
	forall(x,L[v])
		if(pre[x]==0){ //i korifi den exei anakalifthei akoma
	dfsList(G,x,pre,L);}
}
```

2.5.3. Εκτίμηση αλγορίθμου

Διάγραμμα 5



Σχήμα 2.5.2 : Σύγκριση dfsMatrix vs dfsList vs LEDA dfs

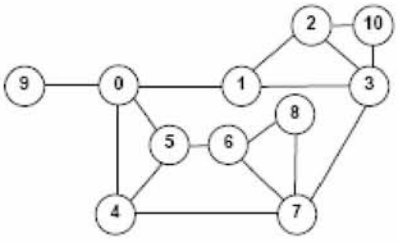
2.5.3.1. Επεξήγηση διαγράμματος

Ο οριζόντιος άξονας εκφράζει το μέγεθος εισόδου στον αλγόριθμο, δηλαδή την πολυπλοκότητα του γραφήματος G, και συγκεκριμένα τον αριθμό των κορυφών. Ο κάθετος άξονας δείχνει το χρόνο τρεξίματος για το αντίστοιχο μέγεθος εισόδου. Η μπλέ γραμμή δείχνει τους χρόνους τρεξίματος του αλγορίθμου dfsMatrix έναντι της πράσινης γραμμής του αλγορίθμου dfsList και της κόκκινης, της έτοιμης συνάρτησης της βιβλιοθήκης LEDA. Ο λόγος για τον οποίο και ο αλγόριθμος dfsList ακολουθεί τη γραμμή μηδενικού χρόνου της συνάρτησης της LEDA, είναι γιατί χρησιμοποιεί λίστα αντί του πίνακα όπως ο dfsMatrix. Αυτό είναι που του δίνει σημαντικό προβάδισμα στη λιγότερη χρησιμοποίηση μνήμης καθώς και ταχύτερες πράξεις σύγκρισης.

2.6. ΑΛΓΟΡΙΘΜΟΣ BFS_MATRIX

2.6.1. Ανάλυση αλγορίθμου

Ο αλγόριθμος bfsMatrix εκτελεί αναζήτηση κατά πλάτος, εξού και το BFS (breadth first search), σ' ένα δοθέν γράφημα. Χρησιμοποιεί μια FIFO ουρά, από την οποία οι κορυφές εισάγονται και εξάγονται κατά σειρά αύξουσας (ελάχιστης) αποστάσεως από την κορυφή αφετηρία. Ο πίνακας γειτνιάσεως υπάρχει για να κρατάει τις ακμές για κάθε κορυφή του γραφήματος. Για παράδειγμα, στο **σχήμα 2.6.1** αφού εξετάσει την κορυφή αφετηρία 0, ανακαλύπτει μία μία τις γειτονικές ανεξερεύνητες πάντα κορυφές εισάγοντάς τις, κάθε φορά στην ουρά, δηλαδή τις κορυφές {1, 4, 5, 9}. Αφού τελειώσει με αυτές, εξάγει την πρώτη από την ουρά , δηλαδή την κορυφή 1 και κάνει το ίδιο με αυτήν, έπειτα με την κορυφή 4 και ούτω καθεξής.



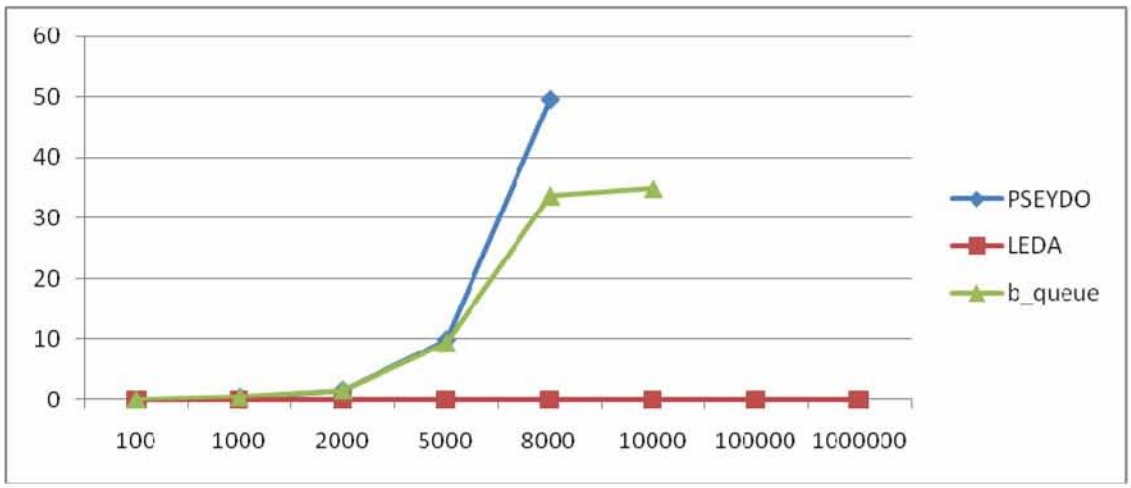
Σχήμα 2.6.1

2.6.2. Υλοποίηση αλγορίθμου με χρήση βιβλιοθήκης LEDA

```
#include <iostream>
#include <cstdio>
#include <C:LEDA\core\array.h>
#include <C:LEDA\graph\node_matrix.h>
#include <C:LEDA\graph\graph.h>
#include <C:LEDA\core\queue.h>


using namespace leda;
using std::cout;
using std::cin;
using std::endl;
static int order;

void bfsMatrix(const graph& G,node v,node_array<int>&
pre,node_matrix<int>& A,queue<node>& Q);

int main() {

	return 0;
}

void bfsMatrix(const graph& G,node v,node_array<int>&
pre,node_matrix<int>& A,queue<node>& Q) {
	Q.append(v);
	pre[v]=order++;
	while(!Q.empty()) {
		node w=Q.pop();
		node u;
		forall_nodes(u,G)
			if((A(w,u)==1) && (pre[u]==-1)) {
				Q.append(u);
				pre[u]=order++;
			}
	}
}
```

2.6.3 Εκτίμηση αλγορίθμου

Διάγραμμα 6



Σχήμα 2.6.2 : Σύγκριση bfsMatrix vs bfsMatrix (b_queue) vs LEDA bfs

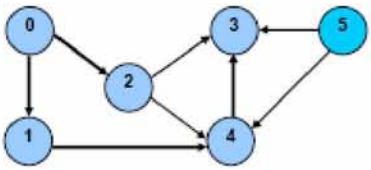
2.6.3.1 Επεξήγηση διαγράμματος

Ο οριζόντιος άξονας εκφράζει το μέγεθος εισόδου στον αλγόριθμο, δηλαδή την πολυπλοκότητα του γραφήματος G, και συγκεκριμένα τον αριθμό των κορυφών. Ο κάθετος άξονας δείχνει το χρόνο τρεξίματος για το αντίστοιχο μέγεθος εισόδου. Η μπλε γραμμή δείχνει το χρόνο τρεξίματος του υλοποιημένου αλγορίθμου bfsMatrix με απλή ουρά (queue), η πράσινή δείχνει το χρόνο του bfsMatrix με bounded queue (δεσμεύεται από την αρχή ο απαιτούμενος χώρος) και η κόκκινη γραμμή το χρόνο της έτοιμης συνάρτησης της LEDA. Όπως φαίνεται και στο διάγραμμα, η υλοποίησης της LEDA υπερτερεί όλων λόγω της καλύτερης διαχείρισης μνήμης απ' αυτή των άλλων δύο, δείχνοντας να μην πτοείται από την πολυπλοκότητα του γραφήματος εισόδου, όπως επίσης υπερτερεί και η υλοποίηση του αλγορίθμου bfsMatrix με bounded queue έναντι της υλοποίησης με απλή ουρά. Η δομή bounded queue η οποία υλοποιείται με κυκλικούς πίνακες, προσφέρει ταχύτερες πράξεις απ' αυτές της απλής δομής queue (υλοποίηση με απλά συνδεδεμένες λίστες), αλλά χρησιμοποιείται μόνο όταν είναι γνωστός ο μέγιστος αριθμός στοιχείων που μπορεί να πάρει.

2.7. ΑΛΓΟΡΙΘΜΟΣ TOPOLSORTLIST

2.7.1. Ανάλυση αλγορίθμου

Ο αλγόριθμος εκτελεί τοπολογική διάταξη (topological sort), στις κορυφές άκυκλου κατευθυνόμενου γραφήματος. Αιτείται δηλαδή, η διάταξη των κορυφών του άκυκλου κατευθυνόμενου γραφήματος, κατά τέτοιο τρόπο ώστε κάθε κορυφή v να προηγείται όλων των κορυφών w με (v,w) να ανήκει στο E. Κατά τη διαδικασία, επιλέγεται μία οποιαδήποτε πηγή s (κορυφή χωρίς εισερχόμενες ακμές), αναφέρεται ως επόμενη στη διάταξη κορυφή και εφαρμόζεται αναδρομικά η διαδικασία στο προκύπτον γράφημα. Συγκεκριμένα, ο αλγόριθμος χρησιμοποιεί λίστες γειτνιάσεως και με τη χρήση του βοηθητικού πίνακα aux, για τη δυναμική καταγραφή εισόδου των κορυφών καθώς και της ουράς FIFO, είναι δυνατή η τοπολογική διάταξη των κορυφών. Στην αρχή μπαίνουν όλες οι κορυφές πηγές στην ουρά. Έπειτα, επιλέγεται κάθε φορά η πρώτη απ' αυτές, καταγράφεται ως i-οστή κορυφή και μειώνεται ο αριθμός εισόδου των κορυφών που ανήκουν στη λίστα γειτνιάσεώς της. Όσες αυτών ισούνται με 0 , εισέρχονται στην ουρά. Για παράδειγμα, στο **σχήμα 2.7.1** που ακολουθεί, οι κορυφές 0 και 5 θα μπουν στην αρχή στην ουρά. Έπειτα, θα επιλεγεί πρώτα η κορυφή πηγή 0, μειώνοντας τον αριθμό εισερχόμενων ακμών των κορυφών 1 και 2, σε 0 προσθέτοντας τες στην ουρά, έπειτα θα επιλεγεί η επόμενη της ουράς, η κορυφή 5 και ούτω καθεξής.



Σχήμα 2.7.1 : Στιγμιότυπο τοπολογικής διατάξεως κορυφών ΚΑΓ

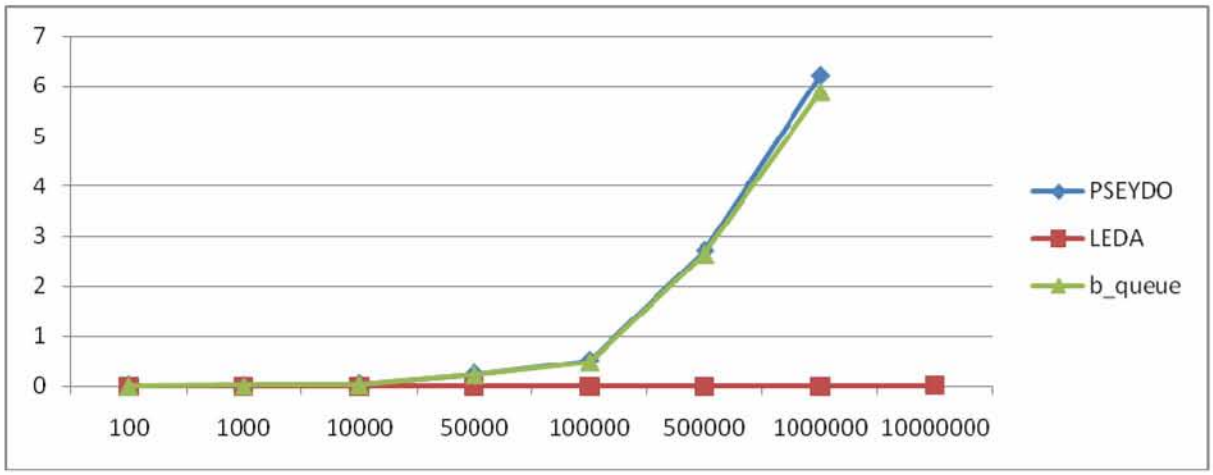
2.7.2. Υλοποίηση αλγορίθμου με χρήση βιβλιοθήκης LEDA

```
#include <iostream>
#include <cstdio>
#include <C:LEDA\core\array.h>
#include <C:LEDA\core\list.h>
#include <C:LEDA\core\queue.h>
#include <C:LEDA\graph\graph.h>
#include <C:LEDA\graph\node_array.h>
#include <C:LEDA\core\b_queue.h>

using namespace leda;
using std::cout;
using std::cin;
using std::endl;
static int seira;
void topolSortList(const graph& G,node_array<int>&
sort,node_array<int>& aux,node_array<list<node>>& List,node_array<int>&
order);

int main() {

	return 0;
}

void topolSortList(const graph& G,node_array<int>&
sort,node_array<int>& aux,node_array<list<node>>& List,node_array<int>&
order) {
	seira=0;
	//o aux xrhsimopoieitai gia ton entopismo pigwn
	sort.init(G,-1);// o pinakas diataksews sort
	node v,x,i;
	forall_nodes(v,G)
		forall(x,List[v]){
			aux[x]++;   //vrethike mia akomi prospiptousa akmh
pros tin x
 }
	b_queue<node> Q(G.number_of_nodes())  ;// oura me tis piges
	forall_nodes(v,G){
		//int k=aux[v];
		if(aux[v]==0) {
		Q.append(v);      //i v einai phgh
		}
	}
	forall(x,Q) {
		order[x]=seira++; //i v apotelei tin i-osti korifi
		forall(i,List[x])
			while(aux[i]>=0)
			if(--aux[i]==0) //afeiresh tis (v,w) apo tis
prospiptoyses
				Q.append(i);            //kai tuxon prosthiki
ths w sto synolo twn phgwn

	}
```

}

2.7.3 Εκτίμηση αλγορίθμου

Διάγραμμα 7



Σχήμα 2.7.2 Σύγκριση topolSortList vs topolSortList (b_queue) vs LEDA TOPSORT

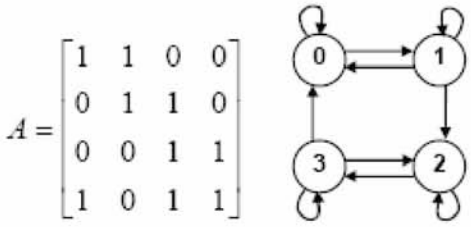
2.7.3.2. Επεξήγηση διαγράμματος

Ο οριζόντιος άξονας εκφράζει το μέγεθος εισόδου στον αλγόριθμο, δηλαδή την πολυπλοκότητα του άκυκλου κατευθυνόμενου γραφήματος G, και συγκεκριμένα τον αριθμό των κορυφών. Ο κάθετος άξονας δείχνει το χρόνο τρεξίματος για το αντίστοιχο μέγεθος εισόδου. Η κόκκινη γραμμή δείχνει το μηδενικό πάντοτε χρόνο της έτοιμης συνάρτησης της LEDA ενώ η μπλε και η πράσινη δείχνουν τις υλοποιήσεις του αλγόριθμου topolSortList, με χρήση απλής ουράς queue και bounded queue αντίστοιχα. Το διάγραμμα για ακόμη μια φορά αναδεικνύει την αποτελεσματικότητα της έτοιμης συνάρτησης της LEDA, η οποία εκτελείται σε μηδενικό χρόνο για τα διάφορα μεγέθη εισόδου, έναντι της υλοποίησης του αλγόριθμου topolSortList, ο οποίος ανεξαρτήτου επιλογής δομής ουράς, αφενός καταγράφει μεγάλους χρόνους τρεξίματος, αφετέρου αδυνατεί να παράγει αποτέλεσμα σε μεγάλα μεγέθη εισόδου, δείγμα της κακής διαχείρισης μνήμης.

2.8. ΑΛΓΟΡΙΘΜΟΣ TRANSCLOSURE

2.8.1. Ανάλυση αλγορίθμου

Ο αλγόριθμος βρίσκει τη μεταβατική κλειστότητα ενός γραφήματος, ήτοι όλες τις δυνατές διασυνδέσεις, άμεσες ή έμμεσες μέσω μονοπατιών, μεταξύ των κορυφών του γραφήματος. Χρησιμοποιεί λίστα γειτνιάσεως για να κρατά τις ακμές για κάθε κορυφή του γραφήματος και τον πίνακα TrClosure ο οποίος έχει αρχικά 1 στις θέσεις όπου υπάρχουν ακμές. Για κάθε ζεύγος κορυφών w και v που μεταξύ τους υπάρχει ακμή, ελέγχεται αν υπάρχει ακμή μεταξύ της w και με κάθε κορυφή x που ανήκει στη λίστα γειτνιάσεως της κορυφής v. Αν όχι, τότε προστίθεται μία ακμή (w,x) και καλείται ξανά η διαδικασία για τις κορυφές (w,x). Για παράδειγμα, στο **σχήμα 2.8.1**, για το ζεύγος κορυφών (0,1) θα ελεγχθεί αν υπάρχει 1 μεταξύ της κορυφής 0 και των κορυφών 0 και 2, δηλαδή τις κορυφές που ανήκουν στη λίστα γειτνιάσεως της 1. Επειδή δεν υπάρχει ακμή (0,2) A[0][2]=0 θα γίνει A[0][2]=1 και θα τρέξει η ίδια διαδικασία για το ζεύγος κορυφών 0 και 2.



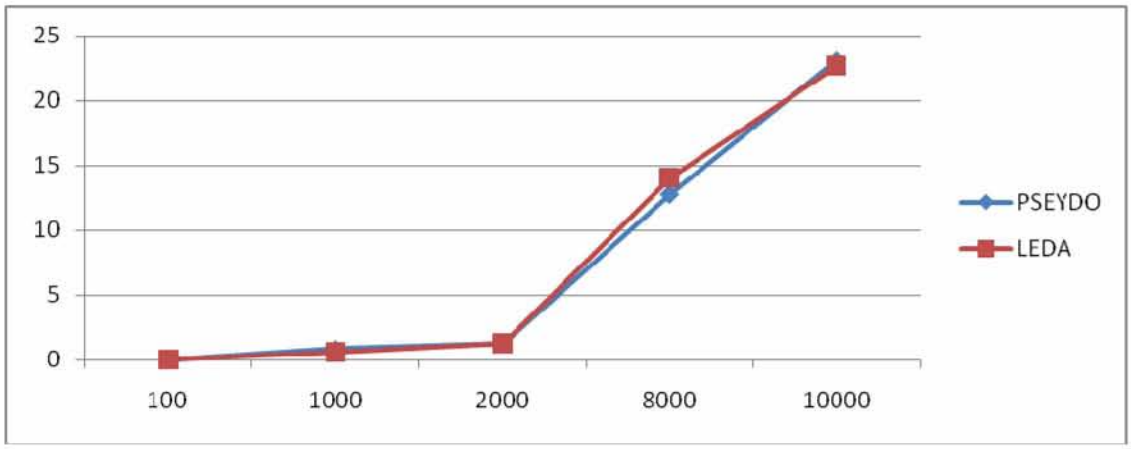
Σχήμα 2.8.1

2.8.2. Υλοποίηση αλγορίθμου με χρήση βιβλιοθήκης LEDA

```
#include <iostream>
#include <cstdio>
#include <C:LEDA\core\array.h>
#include <C:LEDA\graph\node_list.h>
#include <C:LEDA\graph\node_array.h>
#include <C:LEDA\graph\graph.h>
#include <C:LEDA\core\list.h>
#include <C:LEDA\graph\node_matrix.h>


using namespace leda;
using std::cout;
using std::cin;
using std::endl;

static void dfsListTC(const graph& G,graph& t,node w,node
v,node_matrix<int>& TrClosure,node_array<list<node>>&
List_g,node_array<list<node>>& List_t);
void transClosure(const graph& G,graph& t,node_matrix<int>&
TrClosure,node_array<list<node>>& List_g,node_array<list<node>>&
List_t);

int main() {

	return 0;
}

void transClosure(const graph& G,graph& t,node_matrix<int>&
TrClosure,node_array<list<node>>& List_g,node_array<list<node>>&
List_t) {
	node v;
	forall_nodes(v,G)
		dfsListTC(G,t,v,v,TrClosure,List_g,List_t);
}

static void dfsListTC(const graph& G,graph& t,node w,node
v,node_matrix<int>& TrClosure,node_array<list<node>>&
List_g,node_array<list<node>>& List_t) {
	List_t[w].append(v);
	node x;
	forall(x,List_g[v]){
		if(TrClosure(w,x)==0) {
			TrClosure(w,x)=1;
			dfsListTC(G,t,w,x,TrClosure,List_g,List_t);
			}
	}
}
```

2.8.3. Εκτίμηση αλγορίθμου

Διάγραμμα 8



Σχήμα 2.8.1 : Σύγκριση υλοποιημένου TransClosure vs έτοιμης συνάρτησης LEDA

2.8.3.1. Επεξήγηση διαγράμματος

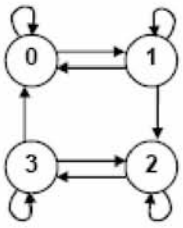
Ο οριζόντιος άξονας εκφράζει το μέγεθος εισόδου στον αλγόριθμο, δηλαδή την πολυπλοκότητα του κατευθυνόμενου γραφήματος G, και συγκεκριμένα τον αριθμό των κορυφών. Ο κάθετος άξονας δείχνει το χρόνο τρεξίματος για το αντίστοιχο μέγεθος εισόδου. Όπως φαίνεται από το **διάγραμμα 8**, οι δύο διαφορετικές υλοποιήσεις τρέχουν σε παραπλήσιους χρόνους για τα αντίστοιχα μεγέθη εισόδου, κάτι που σημαίνει ότι λύνουν το πρόβλημα της μεταβατικής κλειστότητας με παρόμοιο τρόπο. Συμπεραίνεται λοιπόν, ότι οι πράξεις πινάκων, πόσο μάλλον πολυδιάστατων, κοστίζουν σε χρόνο και μνήμη αν παρατηρήσει κανείς τους μεγάλους χρόνους που καταγράφουν οι δύο αλγόριθμοι σε σχετικά μικρά μεγέθη εισόδου.

2.9. ΑΛΓΟΡΙΘΜΟΣ WARSHALL

2.9.1. Ανάλυση αλγορίθμου

Ο αλγόριθμος warshallTC, βρίσκει και αυτός τη μεταβατική κλειστότητα ενός κατευθυνόμενου γραφήματος, με τη χρήση όμως του δυναμικού προγραμματισμού. Σύμφωνα μ' αυτόν, κατασκευάζεται μια σειρά γραφημάτων G_1, G_2, …, G_v, τέτοια ώστε το G_i, $1 \le i \le V$ να περιέχει πρώτον, όλες τις ακμές του G_{i-1}, και δεύτερον, κάθε ακμή της μορφής (v_k, v_j), εφόσον οι ακμές (v_k, v_i), και (v_i, v_j), ανήκουν αμφότερες στο G_{i-1}. Το τελευταίο γράφημα G_v, αποτελεί τη μεταβατική κλειστότητα του G. Συγκεκριμενα, κάθε φορά επιλέγεται μία κορυφή v, και εντοπίζεται κάθε μονοπάτι της μορφής (x-v-u), όπου (x-v) υπάρχον ακμή και (v-u) ήδη υπάρχον μονοπάτι. Αν δεν υπάρχει ακμή (x-u) τότε αυτή προστίθεται. Για παράδειγμα στο **σχήμα 2.9.1**, θα επιλεγεί πρώτα η κορυφή 0 και θα βρεθούν τα μονοπάτια (3-0-x) και (1-0-x). Κρίσιμα είναι τα (3-0-1), (3-0-1-2) και (3-0-1-2-3). Επειδή δεν υπάρχει ακμή (3-1) προστίθεται. Ο αλγόριθμος μετά συνεχίζει με τον ίδιο τρόπο και για τις άλλες κορυφές.

$$A = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{bmatrix}$$



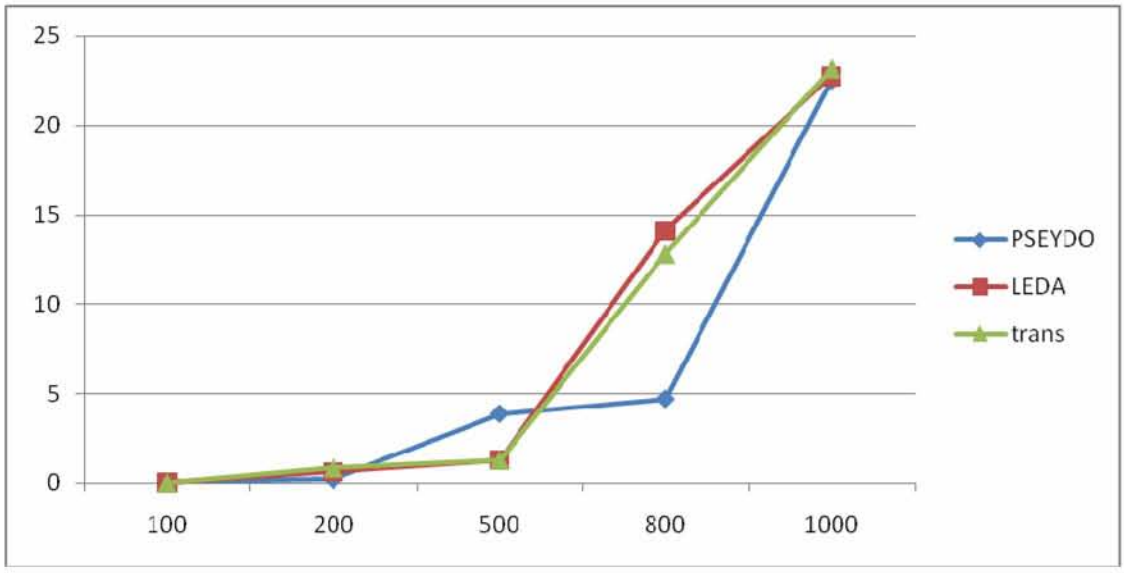
Σχήμα 2.9.1 : Στιγμιότυπο μεταβατικής κλειστότητας κατά Warshall

2.9.2. Υλοποίηση αλγορίθμου με χρήση βιβλιοθήκης LEDA

```
#include <iostream>
#include <cstdio>
#include <C:LEDA\core\array.h>
#include <C:LEDA\graph\node_list.h>
#include <C:LEDA\graph\node_array.h>
#include <C:LEDA\graph\graph.h>
#include <C:LEDA\core\list.h>
#include <C:LEDA\graph\node_matrix.h>

using namespace leda;
using std::cout;
using std::cin;
using std::endl;

void warshallTC(const graph& G,node_matrix<int>&
TrClosure,node_matrix<int>& A);

int main() {

	return 0;
}

void warshallTC(const graph& G,node_matrix<int>&
TrClosure,node_matrix<int>& A) {
	node v,w,i;
	forall_nodes(v,G) {			//arxikopoiisi tou pinaka
kleistotitas
		forall_nodes(w,G)
			TrClosure(v,w)=A(v,w);
	}
	forall_nodes(v,G)
		TrClosure(v,v)=1;
	forall_nodes(i,G) {			//upologismos tou i-stou
grafimatos
		forall_nodes(v,G)	//upologismos metavasews mesw
		if(TrClosure(v,i)==1)	//tis korifis i
			forall_nodes(w,G)
			if(TrClosure(i,w)==1)
				TrClosure(v,w)=1;
	}

}
```

2.9.3. Εκτίμηση αλγορίθμου

Διάγραμμα 9



Σχήμα 2.9.2 : Σύγκριση υλοποιημένου warshallTC vs έτοιμης συνάρτησης LEDA

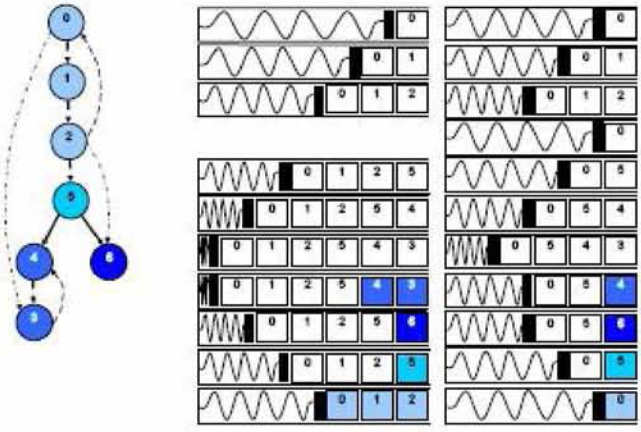
2.9.3.1 Επεξήγηση διαγράμματος

Ο οριζόντιος άξονας εκφράζει το μέγεθος εισόδου στον αλγόριθμο, δηλαδή την πολυπλοκότητα του κατευθυνόμενου γραφήματος G, και συγκεκριμένα τον αριθμό των κορυφών. Ο κάθετος άξονας δείχνει το χρόνο τρεξίματος για το αντίστοιχο μέγεθος εισόδου. Στο διάγραμμα φαίνεται με μπλε γραμμή ο χρόνος του warshall ενώ με πράσινη και κόκκινη, οι χρόνοι του transClosure και της έτοιμης συνάρτησης της LEDA αντίστοιχα. Φαίνεται ότι η υλοποίηση του warshall προς στιγμή παράγει καλύτερα αποτελέσματα, κάτι που οφείλεται στη μέθοδο δυναμικού προγραμματισμού που χρησιμοποιεί και άρα στην καλύτερη διαχείριση μνήμης, στο τέλος όμως και οι τρεις αλγόριθμοι συγκλίνουν στον ίδιο χρόνο τρεξίματος, μιας και όπως αναφέρθηκε παραπάνω οι πράξεις πολυδιάστατων πινάκων κοστίζουν σε χρόνο και μνήμη.

2.10. ΑΛΓΟΡΙΘΜΟΣ GABOW

2.10.1. Ανάλυση αλγορίθμου

Ο αλγόριθμος εντοπίζει ισχυρά συνεκτικές συνιστώσες στο δοθέν γράφημα είσοδο. Για την επίτευξη του σκοπού αυτού, χρησιμοποιεί δύο στοίβες ως εξής: Η πρώτη στοίβα αποθηκεύει το τρέχον μονοπάτι. Η δεύτερη στοίβα σκοπό έχει να εντοπίσει ποιές κορυφές είναι σημεία εισόδου σε ισχυρά συνεκτικές συνιστώσες. Ακολουθώντας την εξερεύνηση αναζήτησης σε βάθος, όταν εντοπίζεται μία κορυφή v, εισάγεται και στις δύο στοίβες. Εάν βρεθεί μία οπισθοακμή – άρα βρισκόμαστε εντός συνιστώσας – με διαδοχικά pop στη δεύτερη στοίβα, παραμένει μόνο η κορυφή απόληξη της οπισθοακμής. Στο τέλος της επεξεργασίας της v, ελέγχεται μήπως είναι κορυφή της δεύτερης στοίβας. Αν ναι, η v είναι σημείο εισόδου και οι κορυφές στην πρώτη στοίβα, από την κεφαλή μέχρι τη v, αποτελούν ισχυρά συνεκτική συνιστώσα. Για παράδειγμα στο **σχήμα 2.10.1**, εκτελώντας AσB, εισάγονται και στις δύο στοίβες οι κορυφές {0, 1, 2}, η κορυφές 1 και 2 κατόπιν αφαιρούνται από τη δεύτερη στοίβα λόγος της οπισθοακμής (2,0). Αφού παρόμοια εισαχθούν οι κορυφές {3, 4, 5} και στις δύο στοίβες και αφαιρεθεί η 3 λόγω οπισθοακμής (3,4), η κορυφή 4 κατά το τέλος της επεξεργασίας της αποτελεί την κεφαλή της δεύτερης ουράς, άρα και σημείο εισόδου. Έτσι με διαδοχικά pop, αφαιρούνται από την πρώτη στοίβα οι κορυφές {3, 4} οι οποίες αποτελούν και την πρώτη ισχυρά συνεκτική συνιστώσα του γραφήματος. Ο αλγόριθμος συνεχίζει ανάλογα.



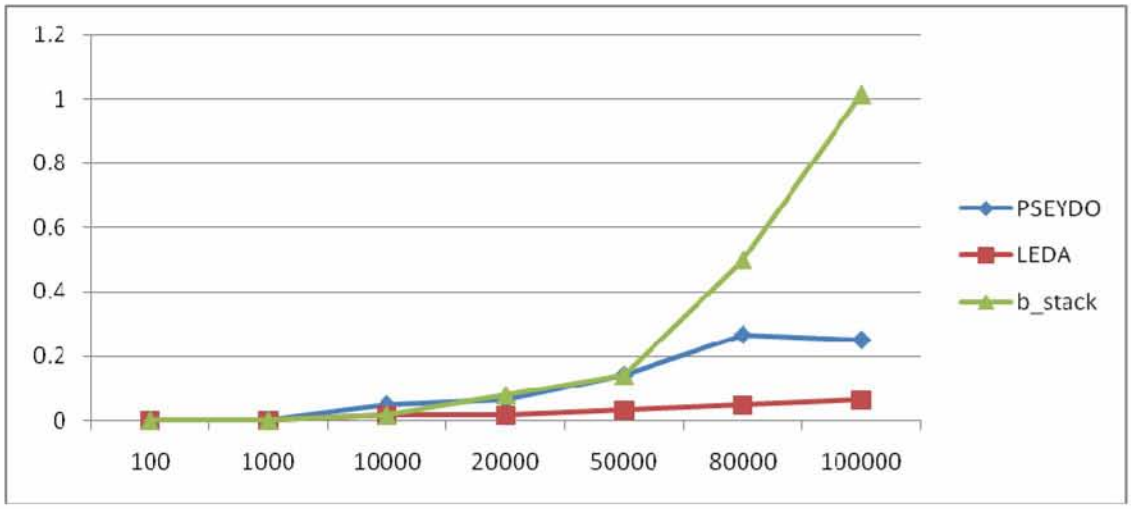
Σχήμα 2.10.1 : Παράδειγμα ευρέσεως ισχυρά συνεκτικών συνιστωσών βάσει Gabow

2.10.2. Υλοποίηση αλγορίθμου με χρήση βιβλιοθήκης LEDA

```
#include <iostream>
#include <cstdio>
#include <C:LEDA\core\array.h>
#include <C:LEDA\core\stack.h>
#include <C:LEDA\graph\node_list.h>
#include <C:LEDA\graph\node_array.h>
#include <C:LEDA\graph\graph.h>
#include <C:LEDA\core\list.h>
#include <C:LEDA\graph\node_matrix.h>

using namespace leda;
using std::cout;
using std::cin;
using std::endl;

static int order,strcompnum;

void gabowSC(const graph& G,node w,node_array<int>&
pre,node_array<int>& sc,stack<node>& S,stack<node>&
pS,node_array<list<node>> List);

int main() {

      return 0;
}

void gabowSC(const graph& G,node w,node_array<int>&
pre,node_array<int>& sc,stack<node>& S,stack<node>&
pS,node_array<list<node>> List) {
      pre[w]=order++;                        //arithmos prodiataksis
      S.push(w);                             //enthesi stis stoives
      pS.push(w);
      node x;
      forall(x,List[w])
            if(pre[x]==-1)                   //aneksereunitos apogonos
                  gabowSC(G,x,pre,sc,S,pS,List);
            else if(sc[x]==-1)               //aprosdioristos arithmos
sunistwsas
                  while(pre[pS.top()]>pre[x])
                        pS.pop();
      if(pS.top()!=w)                        //den apotelei "korifi
eisodou"
            return;
      else
            pS.pop();                        //apotelei "korifi eisodou"
se sunistwsa
      node t;
      do                                          //anakthsh korifwn
sunistwsas
      sc[(t=S.pop())]=strcompnum;
      while(t!=w);
      strcompnum++;                          //enhmerwsh trexontos
arithmou sunistwsas
}
```

2.10.3. Εκτίμηση αλγορίθμου

Διάγραμμα 10



Σχήμα 2.10.2 : Σύγκριση υλοποιημένου Gabow με χρήση απλής stack και με bounded stack vs έτοιμης συνάρτησης LEDA

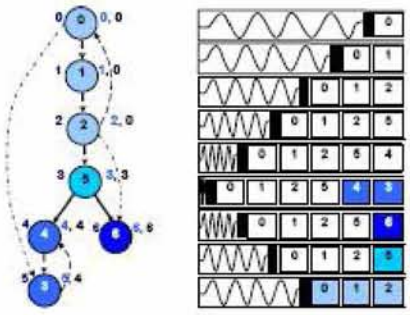
2.10.3.1 Επεξήγηση διαγράμματος

Ο οριζόντιος άξονας εκφράζει το μέγεθος εισόδου στον αλγόριθμο, δηλαδή την πολυπλοκότητα του κατευθυνόμενου γραφήματος G, και συγκεκριμένα τον αριθμό των κορυφών. Ο κάθετος άξονας δείχνει το χρόνο τρεξίματος για το αντίστοιχο μέγεθος εισόδου. Είναι προφανή η ανωτερότητα της έτοιμης συνάρτησης της LEDA , STRONG_COMPONENTS (graph G, int compnum) κόκκινη γραμμή, έναντι των υλοποιήσεων του Gabow με απλή stack (μπλε γραμμή) και με bounded stack (πράσινη γραμμή). Οι πολύ καλύτεροι χρόνοι που καταγράφει η συνάρτηση της LEDA οφείλεται στην καλύτερη υλοποίηση που έχει, χρησιμοποιώντας μάλιστα λίστες. Μάλιστα φαίνεται στο διάγραμμα, ότι η υλοποίηση του Gabow με bounded stack υπερέχει για μικρότερα μεγέθη εισόδου αυτή της απλής stack, ξεφεύγει όμως και αυξάνει σχεδόν εκθετικά το χρόνο τρεξίματος, καθώς αυξάνει το μέγεθος εισόδου. Αυτό είναι λογικό, αφού γίνεται κακή διαχείριση της μνήμης, μιας και για την bounded stack δεσμεύεται εξ' αρχής η απαιτούμενη μνήμη. Επίσης ένας άλλος λόγος είναι η διαφορά υλοποίησης μεταξύ αυτών, αφού η bounded stack υλοποιείται με C++ vectors ενώ η απλή stack με απλά συνδεδεμένες λίστες.

2.11. ΑΛΓΟΡΙΘΜΟΣ TARJAN

2.11.1. Ανάλυση αλγορίθμου

Ο αλγόριθμος εντοπίζει ισχυρά συνεκτικές συνιστώσες ενός γραφήματος G, με μια απλή σάρωση και τη χρήση μιας στοίβας. Απαιτεί μία απλή σάρωση του γραφήματος, εκμεταλλευόμενος το γεγονός πως η κατά σειρά μεταδιατάξεως επεξεργασία των κορυφών του γραφήματος ,σε συνδυασμό με τις τυχόν οπισθοακμές που θα συναντηθούν, ανακαλύπτει τις συνιστώσες κατά αντίστροφη τοπολογική διάταξη. Πιο συγκεκριμένα, βάση ΑσΒ, κάθε φορά που συναντάται μια ανεξέταστη κορυφή v εισάγεται στη στοίβα. Όταν περατωθεί η επεξεργασία της, τελειώσουν δηλαδή όλες οι αναδρομικές κλήσεις που αυτή προκαλεί, ελέγχεται αν είναι σημείο εισόδου (pre[v]=low[v]). Αν ναι, τότε οι κορυφές που περιέχονται στη στοίβα, από την αρχή της μέχρι τη v, αποτελούν μια ισχυρή συνιστώσα. Για παράδειγμα, στο **σχήμα 2.11.1**, σαρώνοντας με ΑσΒ, εισάγονται στη στοίβα διαδοχικά οι κορυφές {0, 1, 2, 5, 4, 3}. Τελειώνοντας με την κορυφή 3, αφού δεν αποτελεί σημείο εισόδου (pre[3] ≠ low[3]), επιστρέφει στην 4 η οποία είναι σημείο εισόδου (pre[4]=low[4]), οπότε και εξάγονται από τη στοίβα οι κορυφές {3, 4} ως η πρώτη ανακαλυφθείσα ισχυρά συνιστώσα. Ο αλγόριθμος συνεχίζει ανάλογα και με τις άλλες κορυφές.



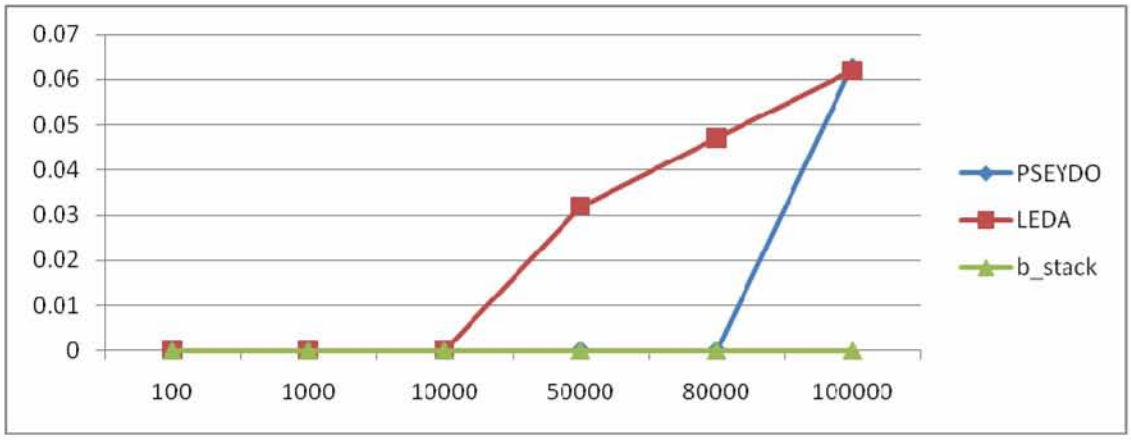
Σχήμα 2.11.1 : Παράδειγμα ευρέσεως ισχυρά συνεκτικών συνιστωσών κατά Tarjan

2.11.2. Υλοποίηση αλγορίθμου με χρήση βιβλιοθήκης LEDA

```
#include <iostream>
#include <cstdio>
#include <C:LEDA\core\array.h>
#include <C:LEDA\core\stack.h>
#include <C:LEDA\graph\node_list.h>
#include <C:LEDA\graph\node_array.h>
#include <C:LEDA\graph\graph.h>
#include <C:LEDA\core\list.h>
#include <C:LEDA\graph\node_matrix.h>

using namespace leda;
using std::cout;
using std::cin;
using std::endl;
static int order,strcompnum;
void tarjanSC(const graph& G,node w,node_array<int>&
pre,node_array<int>& low,stack<node>& S,node_array<list<node>>&
List,node_array<int>& sc);
int main() {

	return 0;
}

void tarjanSC(const graph& G,node w,node_array<int>&
pre,node_array<int>& low,stack<node>& S,node_array<list<node>>&
List,node_array<int>& sc) {
	int minimum;
	pre[w]=order++;		//arithmos prodiataksews ths w
	low[w]=pre[w];		//o mikroteros arithmos prodiatakews pou
prokiptei
	minimum=low[w];			//apo opisthoakmi pros progono
	S.push(w);		//topothetisi sti voithitiki stoiva
	node x;
	forall(x,List[w]) {
		if(pre[x]==-1)	//den exei anakalifthei
			tarjanSC(G,x,pre,low,S,List,sc);
		if(low[x]<minimum)	//vrethike opisthoakmi tou apogonou
x ths w
			minimum=low[x];	//pros progono ths
	}
	if(minimum<low[w]) {	//enhmerwsh tou elaxistou arithmou
prodiataksews
		low[w]=minimum;		//apogonou ths w pros apogono ths
		return;
	}
	node v;
	do { // einai arxh sunistwsas i opoia anaktatai me diadoxika pop
		sc[v=S.pop()]=strcompnum;
		low[v]=G.number_of_nodes();	//telos ths epeksergasias ths
	}
	while(v!=w);
	strcompnum++;				//auksisi tou arithmou
sunistwswn
		}
```

2.11.3. Εκτίμηση αλγορίθμου

Διάγραμμα 11



Σχήμα 2.11.2 : Σύγκριση υλοποιημένου Tarzan με χρήση απλής stack και με bounded stack vs έτοιμης συνάρτησης LEDA

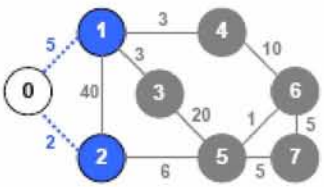
2.11.3.1. Επεξήγηση διαγράμματος

Ο οριζόντιος άξονας εκφράζει το μέγεθος εισόδου στον αλγόριθμο, δηλαδή την πολυπλοκότητα του κατευθυνόμενου γραφήματος G, και συγκεκριμένα τον αριθμό των κορυφών. Ο κάθετος άξονας δείχνει το χρόνο τρεξίματος για το αντίστοιχο μέγεθος εισόδου. Παρότι στο διάγραμμα φαίνεται οι χρόνοι των διαφορετικών υλοποιήσεων να διαφέρουν κατά πολύ, αυτό δεν ισχύει. Η διαφορά χρόνων που καταγράφουν οι δύο διαφορετικές υλοποιήσεις του Tarzan και η έτοιμη συνάρτηση της LEDA, είναι της τάξης εκατοστών του δευτερολέπτου, που σε συνδυασμό με τους μηδενικούς χρόνους τρεξίματός τους, αποφαίνεται ότι και όλες οι υλοποιήσεις λειτουργούν αποτελεσματικά χάρη στη σωστή χρήση των δομών που χρησιμοποιούν (στοίβα για τον Tarzan και λίστες για τη LEDA).

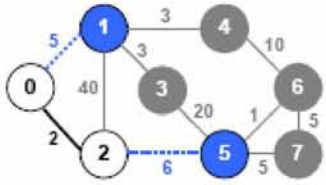
2.12. ΑΛΓΟΡΙΘΜΟΣ PRIM

2.12.1. Ανάλυση αλγορίθμου

Ο αλγόριθμος PRIM, εντοπίζει ελάχιστο επικαλύπτον δέντρο σ' ένα βεβαρημένο συνεκτικό γράφημα. Ξεκινώντας από το στοιχειώδες δένδρο της μίας κορυφής, κάθε φορά επιλέγει να συμπεριλάβει στο υπό διαμόρφωση ΕΕΔ Τ, την ακμή (v, w) με το μικρότερο βάρος, με v v' ανήκει στο Τ και w v' ανήκει στο V-Τ. Τια κάθε κορυφή του V-Τ, διατηρείται μόνο η ελαφρύτερη από τις συνοριακές ακμές που την ενώνουν με κορυφές του Τ, εάν βέβαια υπάρχουν. Για παράδειγμα, στο **σχήμα 2.12.1**, ξεκινώντας από την κορυφή 0, το σύνολο V-Τ, ή αλλιώς σύνολο παρυφής, αποτελείται από τις κορυφές 1 και 2. Θα επιλεγεί η ακμή (0, 2) ως η ελαφρύτερη αυξάνοντας το ΕΕΔ κατά μία κορυφή. Το σύνολο παρυφής έπειτα θα είναι όπως φαίνεται από το **σχήμα 2.12.2,** οι κορυφές 1 και 5 και θα επιλέγει η 1 ως η ελαφρύτερη ακμή μεταξύ των κορυφών του συνόλου V-Τ και Τ. Ο αλγόριθμος συνεχίζει ανάλογα.



Σχήμα 2.12.1



Σχήμα 2.12.2

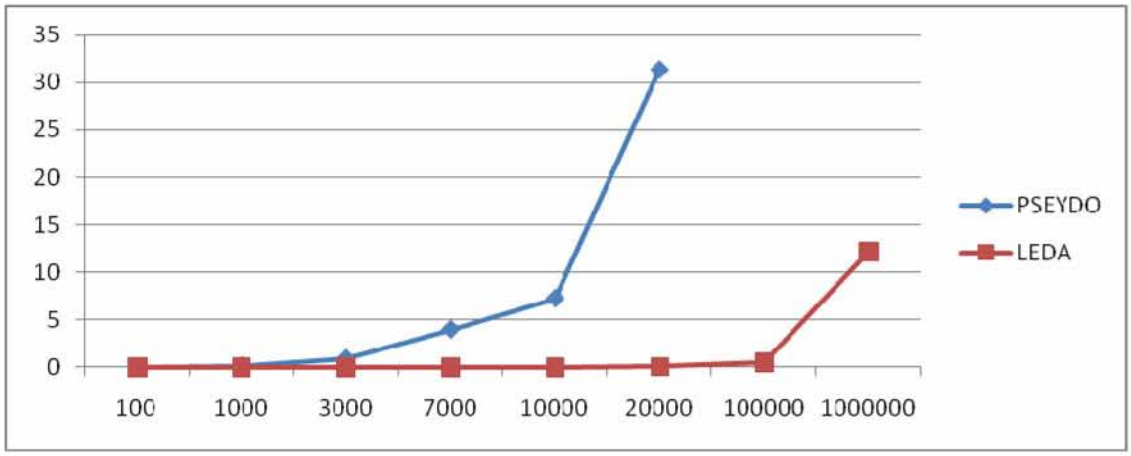
2.12.2. Υλοποίηση αλγορίθμου με χρήση βιβλιοθήκης LEDA

```
#include <iostream>
#include <cstdio>
#include <C:LEDA\core\list.h>
#include <C:LEDA\core\array.h>
#include <C:LEDA\graph\node_array.h>

using namespace leda;
using std::cout;
using std::cin;
using std::endl;

const int TRUE = 1;
const int FALSE = 0;

void Prim(UGRAPH<char,int> MyGraph);
bool Reachable(UGRAPH<char,int> MyGraph,node LookForNode,node
ParentNode,node CurrentNode);

int main() {

      return 0;
}

bool Reachable(UGRAPH<char,int> MyGraph,node LookForNode,node
ParentNode,
node CurrentNode) {

node AdjacentNode;

//----See of found original source
if (LookForNode == CurrentNode)
    return(TRUE);
//----Otheriwse look through all successors
else forall_adj_nodes(AdjacentNode,CurrentNode) {
        if (AdjacentNode != ParentNode &&
Reachable(MyGraph,LookForNode,
CurrentNode,AdjacentNode))
            return(TRUE);
        }
return(FALSE);
}

// PRIM
void Prim(UGRAPH<char,int> MyGraph) {

edge Edge;
//----Lista na krataei tis akmes
list<edge> SortedEdges;
//----Arxikopoiisi xwris korifes sto dentro
node_array<bool> InTree(MyGraph,FALSE);
list_item ListItem;


bool FoundOne;
```

```
//----Arxikopoiisi prwtis koryfis sto dentro
InTree[MyGraph.first_node()] = TRUE;
//----Taxinomisi twn akmwn tou grafhmatos
MyGraph.sort_edges();
//----Apokripsi olwn twn akmwn
forall_edges(Edge,MyGraph) {
    MyGraph.hide_edge(Edge);
    }
//----Lista olwn twn krifwn akmwn
SortedEdges = MyGraph.hidden_edges();

//----Loop oso to dentro den exei oloklirwthei
while (MyGraph.number_of_edges() < MyGraph.number_of_nodes() - 1) {

    FoundOne = FALSE;

    forall_items(ListItem,SortedEdges) {

        Edge = SortedEdges.contents(ListItem);

        if ((InTree[MyGraph.source(Edge)] &&
!InTree[MyGraph.target(Edge)]) ||
(!InTree[MyGraph.source(Edge)] && InTree[MyGraph.target(Edge)])) {
//----Diegrapse to item apo ti lista
            SortedEdges.del_item(ListItem);
//----Epanefere tin akmi sto grafima
            MyGraph.restore_edge(Edge);

            InTree[MyGraph.source(Edge)] = TRUE;
            InTree[MyGraph.target(Edge)] = TRUE;

            FoundOne = TRUE;
            break;
            }
        }
//----An den iparxei katallhlh akmh diekopse
    if (!FoundOne)
        break;
    }

//----An to grafhma den einai epikalupton tipwse minima lathous
if (MyGraph.number_of_edges() < MyGraph.number_of_nodes() - 1)
    cout << "No spanning tree" << endl;
//----Diaforetika tupwse to dentro
else MyGraph.print();
}
```

2.12.3. Εκτίμηση αλγορίθμου

Διάγραμμα 12



Σχήμα 2.12.3 : Σύγκριση υλοποιημένου PRIM vs έτοιμης συνάρτησης LEDA

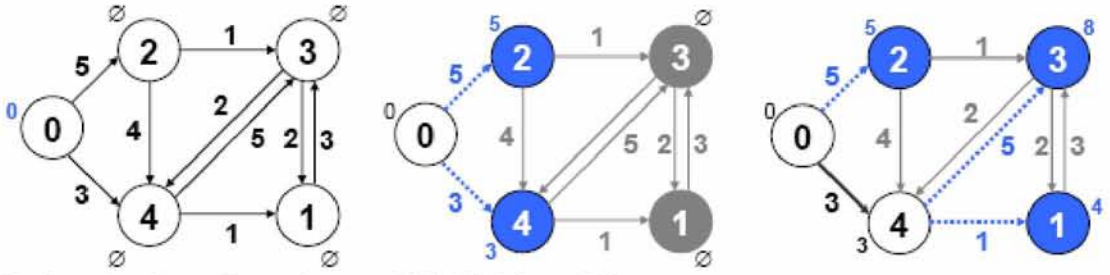
2.12.3.1. Επεξήγηση διαγράμματος

Ο οριζόντιος άξονας εκφράζει το μέγεθος εισόδου στον αλγόριθμο, δηλαδή την πολυπλοκότητα του κατευθυνόμενου γραφήματος G, και συγκεκριμένα τον αριθμό των κορυφών. Ο κάθετος άξονας δείχνει το χρόνο τρεξίματος για το αντίστοιχο μέγεθος εισόδου. Η μπλε γραμμή δείχνει το χρόνο του υλοποιημένου PRIM ενώ η κόκκινη της έτοιμης συνάρτησης της LEDA για την εύρεση ελάχιστων επικαλύπτων δέντρα, MIN_SPANNING_TREE(graph G, edge_array<int> cost). Το πρώτο σημαντικό πόρισμα του διαγράμματος είναι η μεγαλύτερη αντοχή της LEDA έναντι της απλής υλοποίησης, σε μεγαλύτερα μεγέθη εισόδου, δείγμα της καλύτερης διαχείρισης μνήμης που διαθέτει. Το δεύτερο πόρισμα είναι οι κατά πολύ καλύτεροι χρόνοι που καταγράφει η συνάρτηση MIN_SPANNING_TREE , αποτέλεσμα της καλύτερης υλοποίησης, με καταλληλότερες δομές.

2.13. ΑΛΓΟΡΙΘΜΟΣ DIJSTRA

2.13.1. Ανάλυση αλγορίθμου

Ο αλγόριθμος λύνει το πρόβλημα Συντομότερων Μονοπατιών Μοναδικής Πηγής (ΣΜΜΠ), μόνο όμως για βεβαρημένα γραφήματα θετικού κόστους. Σε γενικές γραμμές, ο αλγόριθμος χωρίζει τις κορυφές σε δύο σύνολα: το πρώτο S, στο οποίο ανήκουν οι κορυφές για τις οποίες έχει βρεθεί το συντομότερο μονοπάτι από την πηγή και το δεύτερο, V-S (σύνολο παρυφής), στο οποίο ανήκουν οι κορυφές για τις οποίες είναι γνωστή μια προσέγγιση της τελικής λύσεως. Οι προσεγγίσεις αυτές βελτιώνονται και βελτιστοποιούνται, κατά τι διάρκεια μιας βεβαρημένης αναζήτησης κατά πλάτος. Στο **σχήμα 2.13.1**, παρουσιάζονται 3 βήματα του αλγορίθμου σ' ένα βεβαρημένο γράφημα. Στην αρχή όλες οι κορυφές εκτός της αρχικής-πηγή, 0, έχουν άπειρη απόσταση από την πηγή. Το σύνολο κορυφής αποτελείται από τις κορυφές {2, 4} με άπειρες αποστάσεις. Επιλέγεται η 4 ως εγγύτερη, χαλαρώνοντας την απόστασή της σε 3 από ∞. Το νέο σύνολο παρυφής αποτελείται τώρα από τις κορυφές {2, 1, 3} από τις οποίες επιλέγεται η 1 ως εγγύτερη από την πηγή με απόσταση 4. Ο αλγόριθμος συνεχίζει ανάλογα μέχρι να βρεθούν όλα τα συντομότερα μονοπάτια από την πηγή 0.



Σχήμα 2.13.1 : Στιγμιότυπα ΣΜΜΠ βάσει Dijstra

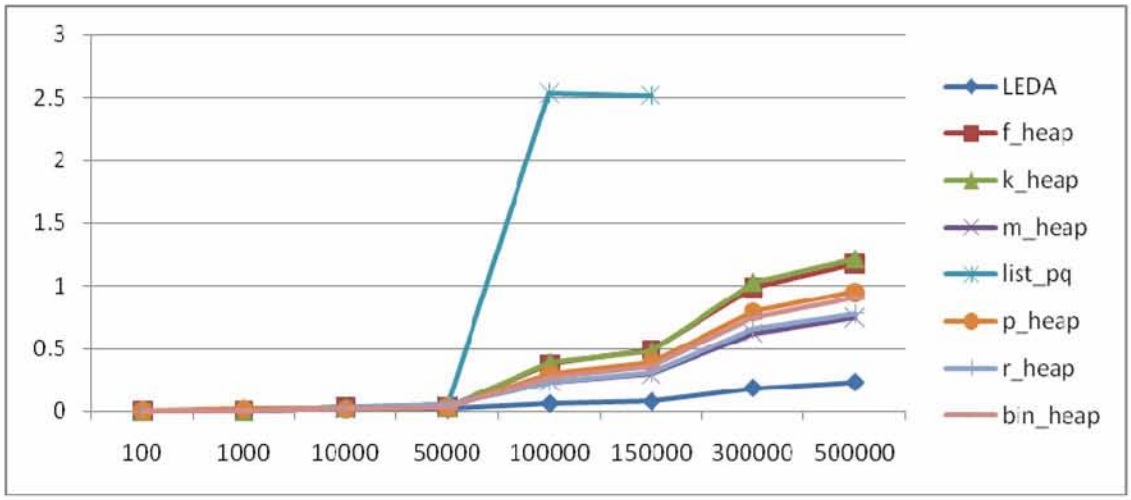
2.13.2. Υλοποίηση αλγορίθμου με χρήση βιβλιοθήκης LEDA

```
#include <iostream>
#include <cstdio>
#include <C:LEDA\graph\graph.h>
#include <C:LEDA\core\_p_queue.h>
#include <C:LEDA\graph\graph_alg.h>
#include <C:LEDA\core\impl\f_heap.h>
#include <C:LEDA\core\impl\k_heap.h>
#include <C:LEDA\core\impl\bin_heap.h>
#include <C:LEDA\core\impl\m_heap.h>
#include <C:LEDA\core\impl\p_heap.h>
#include <C:LEDA\core\impl\r_heap.h>
#include <C:LEDA\core\impl\list_pq.h>


using namespace leda;
using std::cout;
using std::cin;
using std::endl;

template<class pq_impl>
void dijkstra(graph& G,
              node s,
              edge_array<int>&  cost,
              node_array<int>&  dist,
              node_array<edge>& pred,
              p_queue<int,node,pq_impl>& PQ)     // prosthetes
parametroi
{                                                // PQ ylopoiisi...
  typedef typename p_queue<int,node,pq_impl>::item pq_item;
  node_array<pq_item> I(G);
  node v;

  forall_nodes(v,G)                              // Fash arxikopoiisis
  { pred[v] = nil;
    dist[v] = MAXINT;
   }
  dist[s] = 0;                                   //Xtisimo twn priority queues
  I[s] = PQ.insert(0,s);

  while (! PQ.empty())                           // Kirio loop..
  { pq_item it = PQ.find_min();                  // Euresi tis korifis me to
min label
    node u = PQ.inf(it);
    int du = dist[u];
    edge e;
    forall_adj_edges(e,u)                        // Enhmerwsh twn labels
    { v = G.target(e);
      int c = du + cost[e];
      if (c < dist[v])
      { if (dist[v] == MAXINT)
          I[v] = PQ.insert(c,v);
        else
          PQ.decrease_p(I[v],c);                 // Meiwsh toy kleidiou
        dist[v] = c;
```

```
            pred[v] = e;
          }
        }
       PQ.del_item(it);                    // Diagrafh tis korifis apo PQ
      }
  }
```

2.13.3. Εκτίμηση αλγορίθμου

Διάγραμμα 13



Σχήμα 2.13.2 : Σύγκριση υλοποιημένου Dijstra με διαφορετικές δομές vs έτοιμης συνάρτησης LEDA

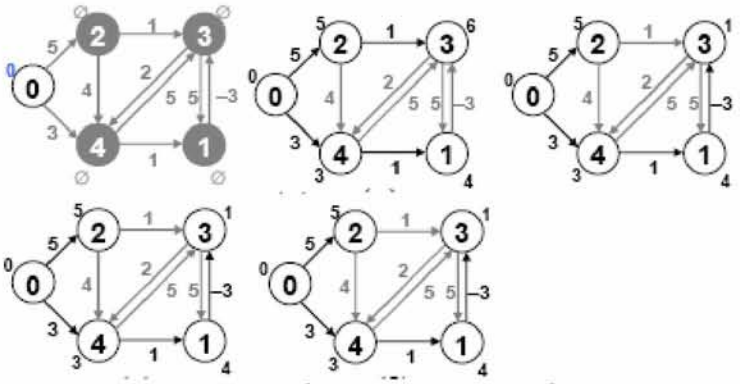
2.13.3.1. Επεξήγηση διαγράμματος

Ο οριζόντιος άξονας εκφράζει το μέγεθος εισόδου στον αλγόριθμο, δηλαδή την πολυπλοκότητα του βεβαρημένου γραφήματος G, και συγκεκριμένα τον αριθμό των κορυφών. Ο κάθετος άξονας δείχνει το χρόνο τρεξίματος για το αντίστοιχο μέγεθος εισόδου. Το διάγραμμα παρουσιάζει αποτελέσματα πολλών υλοποιήσεων του αλγορίθμου Dijstra, με διαφορετική δομή ουράς προτεραιότητας κάθε φορά, καθώς και της έτοιμης συνάρτησης της LEDA. Κατά αύξουσα αποτελεσματικότητα υλοποιήσεων, τ' αποτελέσματα είναι: list_pq, k_heap, f_heap, p_heap, bin_heap, r_heap, m_heap και τελευταία και καλύτερη η έτοιμη συνάρτηση της LEDA.

2.14. ΑΛΓΟΡΙΘΜΟΣ BELLLMAN - FORD

2.14.1. Ανάλυση αλγορίθμου

Ο αλγόριθμος λύνει το πρόβλημα Συντομότερων Μονοπατιών Μοναδικής Πηγής (ΣΜΜΠ), για βεβαρημένο γράφημα, ακόμα και όταν αυτό περιέχει αρνητικά βάρη. Στηρίζεται στην αρχή των χαλαρώσεων ακμών εκτελώντας V-1, διαφορετικά στάδια. Σε κάθε ένα από αυτά, δοκιμάζονται για χαλάρωση όλες οι ακμές του γραφήματος και αν ακόμα και στο τέλος υπάρχει ακμή που επιδέχεται χαλάρωση, τότε το γράφημα διαθέτει αρνητικό κύκλο. Διαφορετικά, έχουν εντοπιστεί τα συντομότερα μονοπάτια. Στο **σχήμα 2.14.1**, φαίνονται το αρχικό γράφημα με τις άπειρες αποστάσεις των κορυφών από την πηγή 0, και μετά, τα V-1 γραφήματα – στάδια χαλαρώσεως.



Σχήμα 2.14.1 : Στιγμιότυπα ΣΜΜΠ κατά Bellman – Ford.

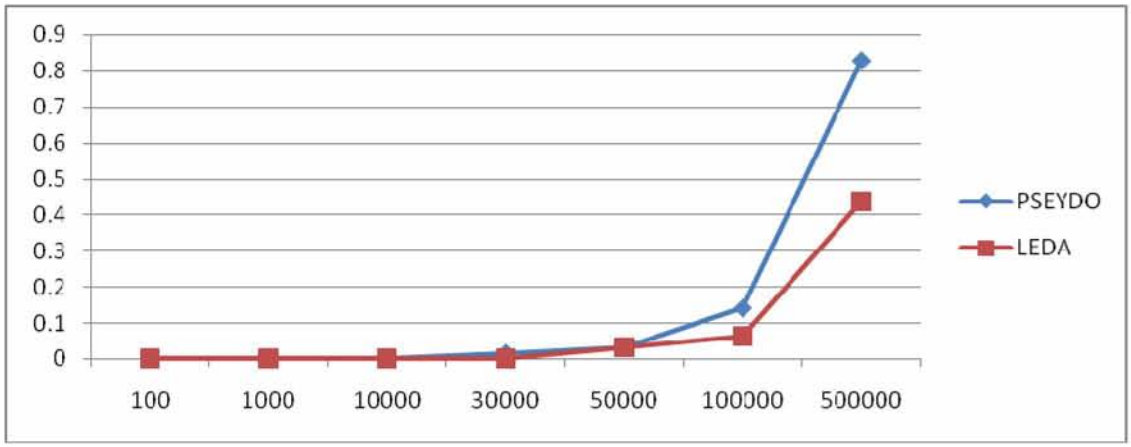
2.14.2. Υλοποίηση αλγορίθμου με χρήση βιβλιοθήκης LEDA

```
#include <iostream>
#include <cstdio>
#include <C:LEDA\graph\graph.h>
#include <C:LEDA\graph\graph_alg.h>
#include <C:LEDA\graph\edge_array.h>
#include <C:LEDA\graph\node_array.h>
#include <C:LEDA\core\b_queue.h>


using namespace leda;
using std::cout;
using std::cin;
using std::endl;

inline void Update_pred(const graph& G,node v,const node_array<bool>&
in_R,
                                        node_array<bool>&
reached_from_node_in_U,node_array<edge>& pred) {

      reached_from_node_in_U[v]=true;
      edge e;
      forall_adj_edges(e,v)
                  {node w=G.target(e);

      if(!reached_from_node_in_U[w])
                        {if(in_R[w]) pred[w]=e;

      Update_pred(G,w,in_R,reached_from_node_in_U,pred);
                        }
                  }
}


bool Bellman_Ford_B_T(const graph& G,node s,const edge_array<int>&
cost,node_array<int>& dist,node_array<edge>& pred);

int main() {

      return 0;
}

bool Bellman_Ford_B_T(const graph& G,node s,const edge_array<int>&
cost,node_array<int>& dist,node_array<edge>& pred) {
      int n=G.number_of_nodes();
      int phase_count=0;
      b_queue<node> Q(n+1);
      node_array<bool> in_Q(G,false);
      node u,v;
      edge e;
      forall_nodes(v,G) pred[v]=nil;
      dist[s]=0;
      Q.append(s); in_Q[s]=true;
      Q.append((node)nil);
      while(phase_count<n)
```

```
{u=Q.pop();
if(u==nil)
{phase_count++;
if(Q.empty()) return true;
Q.append((node)nil);
continue;
}
else in_Q[u]=false;
int du=dist[u];
forall_adj_edges(e,u)
{v=G.opposite(u,e); //Gia na douleuei kai gia mh kateuthinomena
int d=du+cost[e];
if((pred[v]==nil && v!=s) || d<dist[v])
{dist[v]=d; pred[v]=e;
if(!in_Q[v]) { Q.append(v); in_Q[v]=true; }
                                        }
                        }
                    }
if(pred[s]!=nil) return false;
node_array<bool> in_R(G,false);
forall_edges(e,G)
      if(e!=pred[G.target(e)]) ((graph*)&G)->hide_edge(e);
DFS(G,s,in_R);
((graph*)&G)->restore_all_edges();
node_array<bool> reached_from_node_in_U(G,false);
forall_nodes(v,G)
      if(in_Q[v] && !reached_from_node_in_U[v])
            Update_pred(G,v,in_R,reached_from_node_in_U,pred);
return false;
      }
```

2.14.3. Εκτίμηση αλγορίθμου

Διάγραμμα 14



Σχήμα 2.14.2 : Σύγκριση υλοποιημένου Bellman Ford vs έτοιμης συνάρτησης LEDA

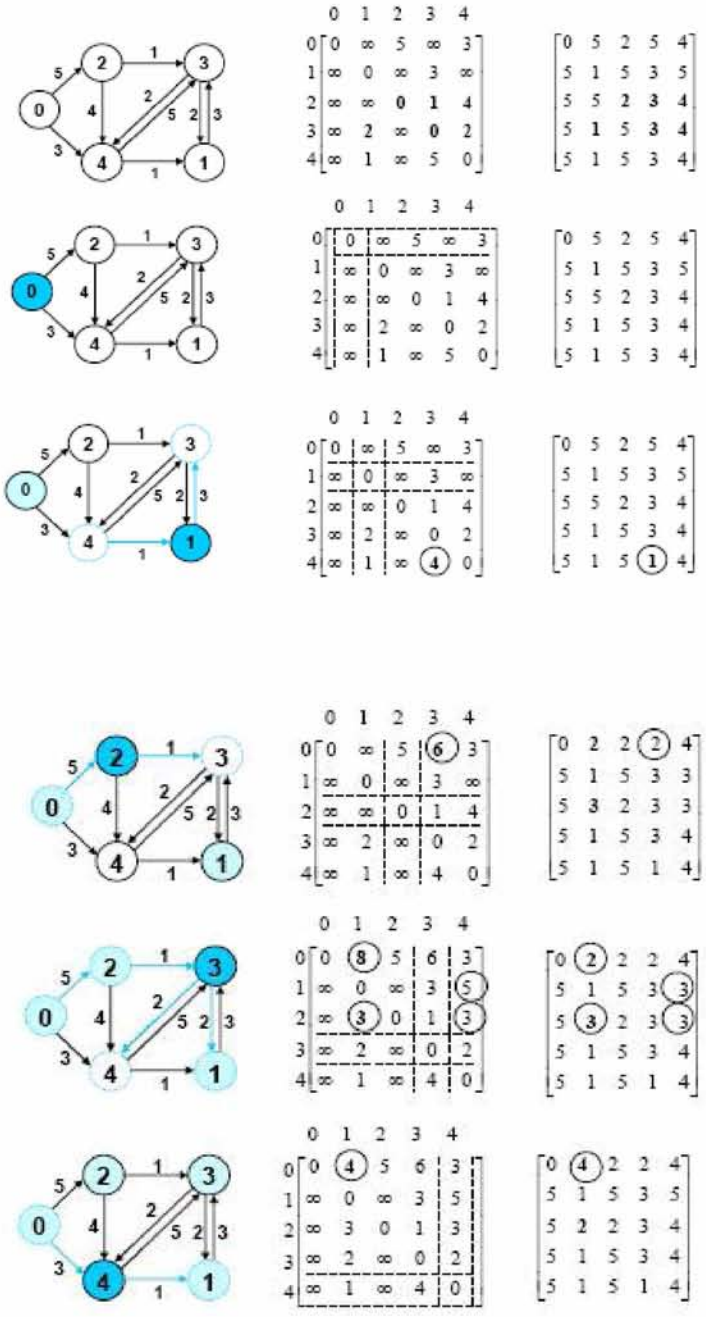
2.14.3.1. Επεξήγηση διαγράμματος

Ο οριζόντιος άξονας εκφράζει το μέγεθος εισόδου στον αλγόριθμο, δηλαδή την πολυπλοκότητα του βεβαρημένου γραφήματος G, και συγκεκριμένα τον αριθμό των κορυφών. Ο κάθετος άξονας δείχνει το χρόνο τρεξίματος για το αντίστοιχο μέγεθος εισόδου. Ο λόγος που η έτοιμη συνάρτηση της LEDA (κόκκινη γραμμή), καταγράφει καλύτερους χρόνους απ' τον υλοποιημένο αλγόριθμο του Bellman Ford (μπλε γραμμή), είναι η απλούστερη υλοποίηση της πρώτης έναντι της δεύτερης. Ο υλοποιημένος αλγόριθμος Bellman Ford περιλαμβάνει πολλές πράξεις, καθώς και αναδρομικές κλήσεις αυξάνοντας έτσι τη μνήμη που χρησιμοποιεί.

2.15. ΑΛΓΟΡΙΘΜΟΣ FLOYD

2.15.1. Ανάλυση αλγορίθμου

Ο αλγόριθμος επιλύει το πρόβλημα των συντομότερων μονοπατιών όλων των ζευγών (ΣΜΟΖ) ενός γραφήματος, στηριζόμενος στην τεχνική του δυναμικού προγραμματισμού. Αφού διαταχθούν οι κορυφές με μια τυχαία σειρά διατάξεως, εκτελούνται V βήματα υπολογισμού. Κατά το L-ιοστό βήμα, κατασκευάζονται τα συντομότερα μονοπάτια $\pi_{i,j}$, κόστους $d_{i,j}$, μεταξύ όλων των ζευγών κορυφών i,j, τα οποία αποτελούνται αποκλειστικά από κορυφές που ανήκουν στο σύνολο των L πρώτων κορυφών, εφαρμόζοντας τη χαλάρωση μονοπατιού $d^{l}_{i,j} = \min\{d^{l-1}_{i,j}, d^{l-1}_{i,l} + d^{l-1}_{l,j}\}$. Στο παρακάτω σχήμα **2.15.1**, φαίνονται τα V διαδοχικά βήματα του αλγορίθμου.



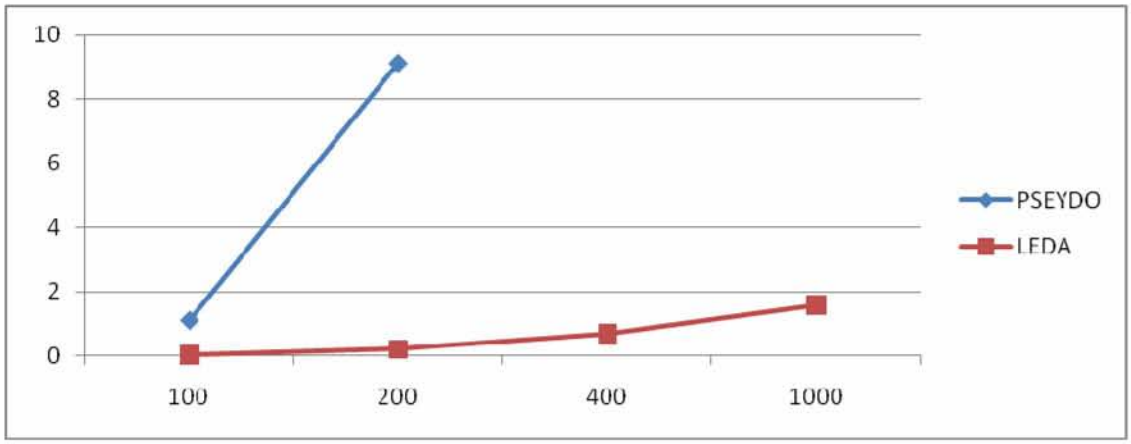
Σχήμα 2.15.1 : Παράδειγμα υπολογισμού ΣΜΟΠ κατά Floyd

2.15.2. Υλοποίηση αλγορίθμου με χρήση βιβλιοθήκης LEDA

```
#include <iostream>
#include <cstdio>
#include <C:LEDA\core\array.h>
#include <C:LEDA\graph\node_matrix.h>
#include <C:LEDA\graph\graph.h>
#include <C:LEDA\numbers\integer.h>


using namespace leda;
using std::cout;
using std::cin;
using std::endl;

void FloydAPSP(const graph& G,node_matrix<int>& d,node_matrix<node>&
path,node_matrix<int>& A);

int main() {

	return 0;
}

void FloydAPSP(const graph& G,node_matrix<int>& d,node_matrix<node>&
path,node_matrix<int>& A) {
	node i,j,l;
	forall_nodes(i,G)
		forall_nodes(j,G) {
			d[i][j]=MAXINT;
			path[i][j]=G.last_node();
	}
	forall_nodes(i,G)
		forall_nodes(j,G)
		if((d[i][j]=A[i][j])<MAXINT)
			path[i][j]=j;
	forall_nodes(l,G)
		forall_nodes(i,G)
		if(d[i][l]<MAXINT)
			forall_nodes(j,G)
			if(d[i][j]>d[i][l]+d[l][i]) {
				path[i][j]=path[i][l];
				d[i][j]=d[i][l]+d[l][j];
			}
		}
```

2.15.3. Εκτίμηση αλγορίθμου

Διάγραμμα 15



Σχήμα 2.15.2 : Σύγκριση υλοποιημένου FLOYD vs έτοιμης συνάρτησης LEDA

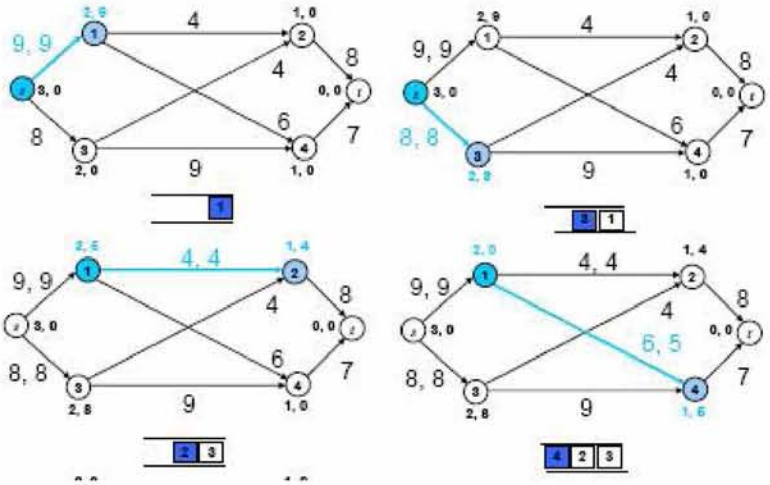
2.15.3.1. Επεξήγηση διαγράμματος

Ο οριζόντιος άξονας εκφράζει το μέγεθος εισόδου στον αλγόριθμο, δηλαδή την πολυπλοκότητα του βεβαρημένου γραφήματος G, και συγκεκριμένα τον αριθμό των κορυφών. Ο κάθετος άξονας δείχνει το χρόνο τρεξίματος για το αντίστοιχο μέγεθος εισόδου. Η παραπάνω υλοποίηση του FLOYD (μπλε γραμμή), δεν αποδίδει καθόλου , καταγράφοντας μεγάλους χρόνους σε μικρά μεγέθη εισόδου , χωρίς μάλιστα ν' αντέχει σε μεγαλύτερα όπως αντέχει η έτοιμη συνάρτηση της LEDA (κόκκινη γραμμή), ALL_PAIRS_SHORTEST_PATHS_T. Είναι λογικό να μην αποδίδει ο απλός FLOYD, δεδομένου ότι παίρνει ως ορίσματα - παραμέτρους, δισδιάστατους πίνακες που αφενός καταναλώνουν αρκετή μνήμη κάνοντας μη εφικτή την εκτέλεση σε μεγαλύτερα μεγέθη εισόδου, αφετέρου κοστίζουν σε χρόνο, μιας και ο αλγόριθμος εκτελεί επαναληπτικά πολλές πράξεις σε στοιχεία των πινάκων αυτών. Απ' την άλλη, η LEDA κάνοντας καλύτερη διαχείριση μνήμης, τρέχει σε μικρότερους πάντα χρόνους , αντέχοντας και σε μεγαλύτερα μεγέθη εισόδου.

2.16. ΑΛΓΟΡΙΘΜΟΣ GOLDBERG - TARJAN

2.16.1. Ανάλυση αλγορίθμου

Ο αλγόριθμος βρίσκει τη μέγιστη ροή σ' ένα δίκτυο ροής, με την τεχνική Προροής – Προωθήσεως. Σύμφωνα με τον Goldberg – Tarzan, όταν επιλεγεί μία ενεργή κορυφή v, κορυφή δηλαδή που έχει περίσσευμα ροής, συνεχίζουμε να προωθούμε δια μέσου αυτής ροή, μέσω νόμιμων γειτονικών ακμών, μέχρι η v να καταστεί ανενεργή ή να τελειώσουν οι νόμιμες ακμές, οπότε και προσαρμόζουμε το ύψος της. Στο **σχήμα 2.16.1**, βλέπουμε ένα στιγμιότυπο τρεξίματος του αλγορίθμου για ένα δίκτυο ροής. Στην αρχή οδηγούνται σε κορεσμό οι ακμές που οδηγούν στις κορυφές 1 και 3. Αποθηκεύεται δε, η σειρά ενεργοποίησης κορυφών σε μία ουρά Fifo, για τη μετέπειτα σειρά επεξεργασίας κορυφών. Έτσι, επεξεργαζόμενη την 1, οι 9 μονάδες που διοχετεύτηκαν σ' αυτήν, διοχετεύονται στις κορυφές 2 και 4 μέσω νόμιμων ακμών, προσθέτοντας τις στην ουρά Fifo. Ο αλγόριθμος συνεχίζει με την κορυφή 2 ανάλογα κ.ο.κ.



Σχήμα 2.16.1 : Στιγμιότυπο της μεθόδου προροής – προωθήσεως με χρήση FIFO.

2.16.2. Υλοποίηση αλγορίθμου με χρήση βιβλιοθήκης LEDA

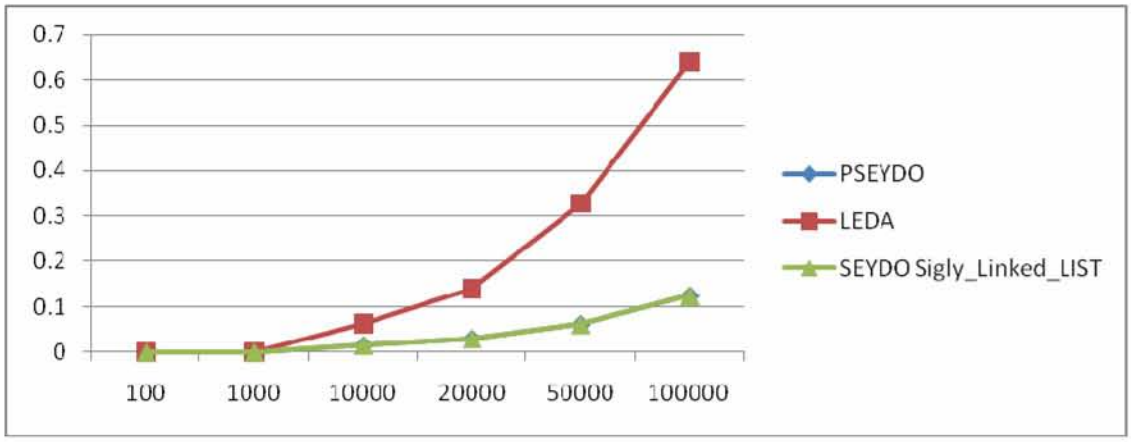
```
#include <iostream>
#include <cstdio>
#include <C:LEDA\core\array.h>
#include <C:LEDA\graph\node_list.h>
#include <C:LEDA\graph\node_array.h>
#include <C:LEDA\graph\edge_array.h>
#include <C:LEDA\graph\graph.h>
#include <C:LEDA\core\list.h>
#include <C:LEDA\graph\node_matrix.h>

using namespace leda;
using std::cout;
using std::cin;
using std::endl;

class fifo_set {
      list<node> L;
public:
      fifo_set(){}
      node del() {if(!L.empty()) return L.pop(); else return nil; }
      void insert(node v,int d) {L.append(v);}
      void insert0(node v,int d) {L.append(v);}
      bool empty() {return L.empty();}
      void clear() {L.clear();}
      ~fifo_set() {}
};
static int num_pushes,num_edge_inspections,num_relabels;
int Max_Flow_Basic(const graph& G,node s,node t,const edge_array<int>&
cap,edge_array<int>& flow,
                           fifo_set& U,int& num_pushes,int&
num_edge_inspections,int& num_relabels);

int main() {

      return 0;
}

int Max_Flow_Basic(const graph& G,node s,node t,const edge_array<int>&
cap,edge_array<int>& flow,
                           fifo_set& U,int& num_pushes,int&
num_edge_inspections,int& num_relabels) {
                              //MF arxikopoiisi flow kai excess kai
koresmos akmwn pou feugoun apo thn s
flow.init(G,0);
if(G.outdeg(s)==0) return 0;
                                 int n=G.number_of_nodes();  int
max_level=2*n-1;
                                 int m=G.number_of_edges();
                                 node_array<int> excess(G,0);
                           // koresmos akmwn pou feugoun apo thn s
                                 edge e;
                                 forall_out_edges(e,s)
                                 {
```

```
                                    int c=cap[e];
                                    if(c==0) continue;
                                    node v=target(e);
                                    flow[e]=c;
                                    excess[s]-=c;
                                    excess[v]+=c;
                              }
                        //MF arxikopoiisi dist kai U
                              node_array<int> dist(G,0); dist[s]=n;
                              node v;
                              forall_nodes(v,G)
                                    if(excess[v]>0)
U.insert(v,dist[v]);
                        //MF arxikopoiisi metrhtwn

num_relabels=num_pushes=num_edge_inspections=0;

                        //MF_BASIC main loop
                              for(;;)
                              {
                                    node v=U.del();
                                    if(v==nil) break;
                                    if(v==t) continue;
                                    int ev=excess[v];    //excess
tou v
                                    int dv=dist[v];
      //epipedo tou  v
                                    edge e;

                                    for(e=G.first_adj_edge(v);
e;e=G.adj_succ(e))
                                    {num_edge_inspections++;
                                    int& fe=flow[e];
                                    int rc=cap[e]-fe;
                                    if(rc==0) continue;
                                    node w=target(e);
                                    int dw=dist[w];
                                    if(dw<dv)
      //isodinamo me (dw==dv-1)
                                    {num_pushes++;
                                    int& ew=excess[w];
                                    if(ew==0) U.insert0(w,dw);
                                    if(ev<=rc)
                                    {ew+=ev; fe+=ev;
                                    ev=0;
      //stop: excess[v] koresmeno
                                    break;
                                    }
                                    else
                                    {ew+=rc; fe+=rc;
                                    ev-=rc;
                                    }
                                  }
                              }
                                    if(ev>0) {
```

```
for(e=G.first_in_edge(v);e;e=G.in_succ(e))
                                                {num_edge_inspections++;
                                                int& fe=flow[e];
                                                if(fe==0) continue;
                                                node w=source(e);
                                                int dw=dist[w];
                                                if(dw<dv)
      //isodinamo tou(dw==dv-1)
                                                {num_pushes++;
                                                int& ew=excess[w];
                                                if(ew==0) U.insert0(w,dw);
                                                if(ev<=fe)
                                                {fe-=ev; ew+=ev;
                                                ev=0;
      //stop: excess[v] koresmeno
                                                break;
                                                }
                                                else
                                                {ew+=fe; ev-=fe;
                                                fe=0;
                                               }
                                        }
                                    }
                                }
                            }
                        }
```

2.16.3. Εκτίμηση αλγορίθμου

Διάγραμμα 16



Σχήμα 2.16.2 : Σύγκριση υλοποιημένου GOLDBERG - TARJAN vs έτοιμης συνάρτησης LEDA

2.16.3.1. Επεξήγηση διαγράμματος

Ο οριζόντιος άξονας εκφράζει το μέγεθος εισόδου στον αλγόριθμο, δηλαδή την πολυπλοκότητα του κατευθυνόμενου γραφήματος G, και συγκεκριμένα τον αριθμό των κορυφών. Ο κάθετος άξονας δείχνει το χρόνο τρεξίματος για το αντίστοιχο μέγεθος εισόδου. Όπως φαίνεται από το διάγραμμα, ο υλοποιημένος αλγόριθμος GOLDBERG – TARJAN (μπλε και πράσινη γραμμή), αποδίδει χρονικά καλύτερα από την έτοιμη συνάρτηση της LEDA, MAX_ FLOW_T (κόκκινη γραμμή). Ο λόγος είναι, ότι η συνάρτηση της LEDA εκτελεί περισσότερους υπολογισμούς απ' ότι ο αλγόριθμος GOLDBERG – TARJAN, υπολογισμούς όπως για το αν το αποτέλεσμα είναι όντως το σωστό κ.α. Άρα, η LEDA μπορεί καταγράφει μεγαλύτερους χρόνους στη συγκεκριμένη περίπτωση, παρ' όλα αυτά όμως, εγγυάται ότι το αποτέλεσμα που δίνει είναι το σωστό.

2.17. ΑΛΓΟΡΙΘΜΟΣ LARGESTCOMMONSUBSEQ

2.17.1. Ανάλυση αλγορίθμου

Ο αλγόριθμος μετράει την ομοιότητα δύο συμβολοσειρών βρίσκοντας τη μέγιστη κοινή υπακολουθίας τους (η μεγαλύτερου μήκους υπακολουθία που εμφανίζεται και στις δύο), ως εξής: Αν $x=x_{i1}, x_{i2}, \ldots x_{ik}$ με $i_1<i_2<\ldots<i_k\leq n$ καλείται υπακολουθία μήκους k της x.. Έστω λοιπόν, $x=x_1x_2, \ldots. x_n$ και $y=y_1y_2\ldots y_m$ οι δύο συμβολοσειρές εισόδου. Η μέγιστη κοινή υπακολουθία τους z, ορίζεται ως εξής:
$Z=\mu\kappa\upsilon(x,y)=x_{i1} x_{i2} \ldots. x_{ik} =y_{j1} y_{j2} \ldots y_{jk}$ με $1\leq i_1<i_2<\ldots<i_k\leq n$ και $1\leq j1<j2<\ldots<jk\leq n$ και το k έχει τη μέγιστη δυνατή τιμή. Στο **σχήμα 2.17.1**, φαίνεται ο τρόπος που τρέχει ο αλγόριθμος για να βρει τη μέγιστη ακολουθία των συμβολοσειρών ΒΑΒΕΛ ΚΑΙ ΑΒΕΛ. Αφού γεμίσει την πρώτη στήλη και γραμμή του πίνακα με 0, κάθε θέση (i,j) συμπληρώνεται ως εξής: Αν υπάρχει ταίριασμα των χαρακτήρων x_i, y_j, τότε στο διαγώνιο κουτί (i-1, j-1), που αφορά τα προθέματα x_{i-1}, y_{j-1}, θα έπρεπε να προστεθεί 1 στον πληθάριθμο και να επικολληθεί ο κοινός χαρακτήρας xi, yj. Διαφορετικά, γίνεται σύγκριση των θέσεων (i-1,j) και (i, j-1) και προτιμάται το πρώτο αν είναι μεγαλύτερο ή ίσο από το δεύτερο, αλλιώς το δεύτερο.

		0	A 1	B 2	E 3	Λ 4
	0	0	←0	←0	←0	←0
B	1	↑0	↑0	↖1	←1	←1
A	2	↑0	↖1	←1	←1	←1
B	3	↑0	↑1	↖2	←2	←2
E	4	↑0	↑1	↑2	↖3	←3
Λ	5	↑0	↑1	↑2	↑3	↖4

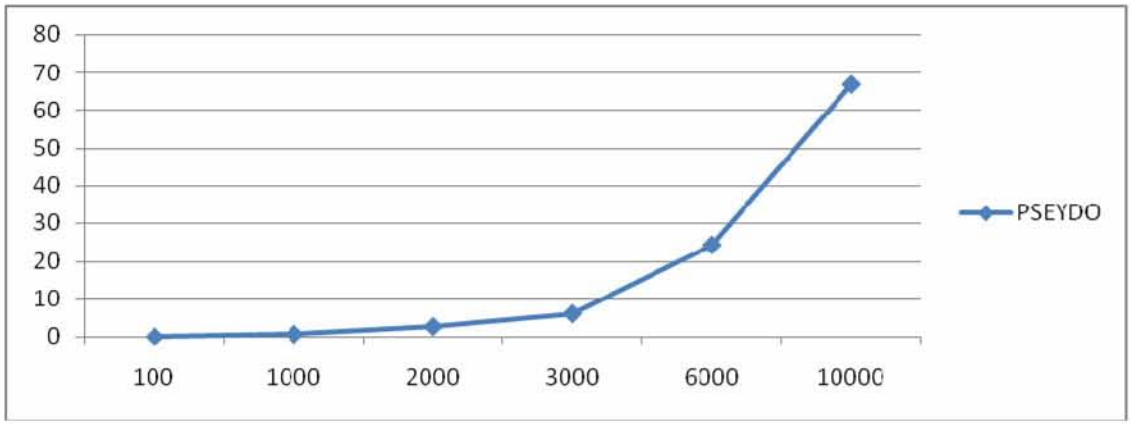
Σχήμα 2.17.1 : Στιγμιότυπο υπολογισμού μέγιστης κοινής υπακολουθίας

2.17.2. Υλοποίηση αλγορίθμου με χρήση βιβλιοθήκης LEDA

```
#include <iostream>
#include <cstdio>
#include <C:LEDA\core\array.h>
#include <C:LEDA\core\array2.h>


using namespace leda;
using std::cout;
using std::cin;
using std::endl;
int NOT_X,NOT_Y,XY;
void largestCommonSubSeq(array<char>& x,array<char>& y,array2<char>&
lCSS);

int main() {

	return 0;
}

void largestCommonSubSeq(array<char>& x,array<char>& y,array2<char>&
lCSS) {
	int n=x.size();
	int m=y.size();
	array2<int> b(n+1,m+1);
	array2<int> c(n+1,m+1);			//pinakes kostous
	array2<int> aux(n+1,m+1);		//voithitikos pinakas
katagrafhs
	for(int i=0;i<=n;i++) {
		c(i,0)=0;
		b(i,0)=NOT_X;
	}
	for(int i=0;i<=m;i++) {
		c(0,i)=0;
		b(0,i)=NOT_Y;
	}
	for(int i=1;i<=n;i++)
		for(int j=1;j<=m;j++) {
			if(x[i]==y[j]) {		//tairiasma
				c(i,j)=c(i-1,j-1)+1;
				aux(i,j)=XY;
			}
			else if(c(i-1,j)>=c(i,j-1)) {
				c(i,j)=c(i-1,j);	//to x(i) den anhkei
				aux(i,j)=NOT_X;
			}
			else {
				c(i,j)=c(i,j-1);	//to y(j) den anhkei
				aux(i,j)=NOT_Y;
			}
		}
		for(int i=0;i<=n;i++)
			for(int j=0;j<=m;j++)
				lCSS(i,j)=c(i,j);
		}
```

2.17.3. Εκτίμηση αλγορίθμου

Διάγραμμα 17



Σχήμα 2.17.2 : Εκτίμηση υλοποιημένου LARGESTCOMMONSUBSEQ

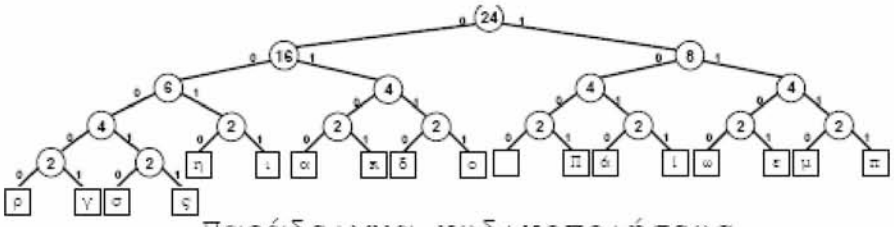
2.17.3.1. Επεξήγηση διαγράμματος

Ο οριζόντιος άξονας δείχνει το μέγεθος εισόδου, ήτοι το μέγεθος των πινάκων συμβολοσειρών, των οποίων θα βρεθεί η μέγιστη κοινή υπακολουθία. Ο κάθετος άξονας δείχνει το χρόνο τρεξίματος του αλγορίθμου για το αντίστοιχο μέγεθος εισόδου. Απ' το διάγραμμα είναι προφανές και λογικό ο αλγόριθμος να τρέχει σε υψηλούς σχετικά χρόνους και για μικρά μεγέθη εισόδου, δεδομένου οτι οι πολλές επαναληπτικές πράξεις σε στοιχεία δισδιάστατων πινάκων κοστίζουν σημαντικά.

2.18. ΑΛΓΟΡΙΘΜΟΣ HUFFMAN

2.18.1 Ανάλυση αλγορίθμου

Ο αλγόριθμος χρησιμοποιείται για συμπίεση κειμένων στηρίζοντας στην αρχή πως χαρακτήρες με μεγαλύτερη συχνότητα εμφανίσεως κωδικοποιούνται με λιγότερα bit απ' ότι οι σπανιότεροι. Με την ιδιότητα των κωδικών προθεμάτων (δυαδικές συμβολοσειρές, με την κρίσιμη ιδιότητα καμία να μην αποτελεί πρόθεμα της άλλης), καθιστά δυνατή την περιγραφή των κωδικών με ένα δυαδικό δένδρο Τ, όπου κάθε αριστερή διακλάδωση αντιστοιχεί στο ψηφίο '0', κάθε δεξιά στο ψηφίο '1', ενώ κάθε φύλλο του δέντρου απεικονίζει και ένα διακριτό κωδικό. Στο **σχήμα 2.18.1**, βλέπουμε το τελικό στάδιο κωδικοποίησης της φράσης «Παράδειγμα κωδικοποιήσεως» με τον εν λόγω αλγόριθμο.



Παράδειγμα κωδικοποιήσεως

Huffman

1001**0100**00000**1010**0110**1101**0011**00001**111**01**
001000**0101**1100**0110**0011**0101**0111**1111**011**10**
110010**0010**0010**00011**

Σχήμα 2.18.1 : Κωδικοποίηση Huffman του «Παράδειγμα κωδικοποίησης

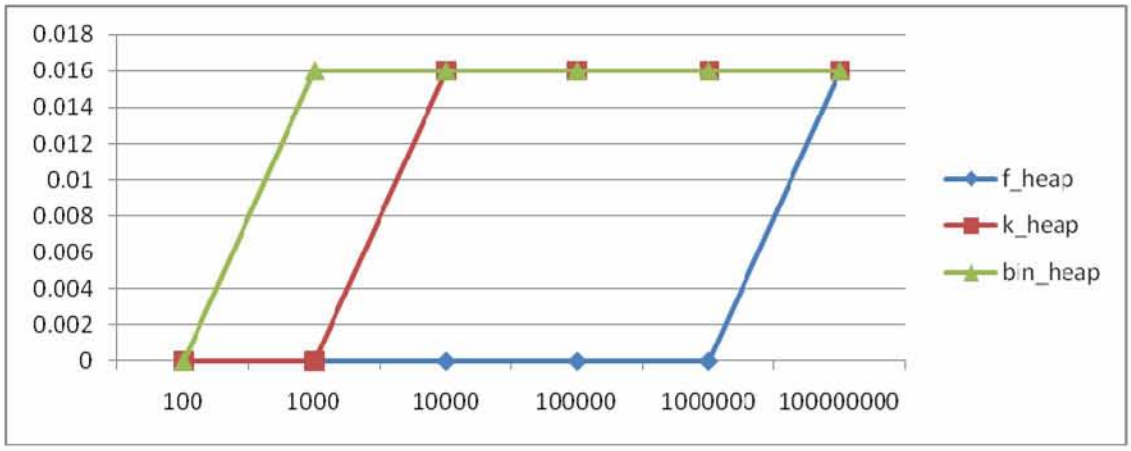
2.18.2. Υλοποίηση αλγορίθμου με χρήση βιβλιοθήκης LEDA

```
#include <iostream>
#include <cstdio>
#include <C:LEDA\core\d_array.h>
#include <C:LEDA\core\p_queue.h>
#include <C:LEDA\core\impl\k_heap.h>
#include <C:LEDA\core\impl\bin_heap.h>
#include <C:LEDA\core\impl\m_heap.h>
#include <C:LEDA\core\impl\p_heap.h>
#include <C:LEDA\core\impl\r_heap.h>
#include <C:LEDA\core\impl\list_pq.h>
#include <C:LEDA\core\string.h>

using namespace leda;
using std::cout;
using std::cin;
using std::endl;

class huffman_node {
public:
  huffman_node *parent;
  int priority;
  bool is_left_child;

  huffman_node() : parent(nil) {};
};


int main()
            {
d_array<char,int> H;
char c;
p_queue<int, huffman_node*> PQ;
  d_array<char, huffman_node*> leaf_belonging_to_char;
  float start_Time=used_time();
  // gia kathe xarakthra eisage ena huffman node stin priority queue
  // autos o node tha einai fullo tou dentrou pou tha kataskeuastei
  forall_defined(c,H) {
    //cout << c << ": " << H[c] << std::endl;
    huffman_node *new_leaf = new huffman_node;
    new_leaf->priority = H[c];
    PQ.insert(H[c], new_leaf);
    leaf_belonging_to_char[c] = new_leaf;
  }

  // kataskeui tou Huffman code
 // cout << "Constructing the Huffman code for all read characters.\n";

  while(PQ.size() > 1) {

    // eksagwgh dio "paidiwn" me elaxisti proteraiotita
    huffman_node *left_child = PQ.inf(PQ.find_min());
    PQ.del_min();
    huffman_node *right_child = PQ.inf(PQ.find_min());
    PQ.del_min();
```

```
    // kataskeui tou neou parent node
    huffman_node *new_parent = new huffman_node;
    left_child->parent = right_child->parent = new_parent;
    left_child->is_left_child = true;
    right_child->is_left_child = false;
    new_parent->priority = left_child->priority + right_child-
>priority;

    // eisagwgh tou neou parent stin priority queue
    PQ.insert(new_parent->priority, new_parent);

    // control output
   /* cout << "New parent created with priority ";
    cout << new_parent->priority << " = ";
    cout << left_child->priority << " + " << right_child->priority <<
"\n";
  }

  cout << "Oi Huffman kwdikes einai:\n";   */
  }

  forall_defined(c, H) {
   // cout << c << ": ";
    huffman_node *climber = leaf_belonging_to_char[c];
    string huffman_code;
    while(climber->parent != nil) { // min kaneis tipota gia ti riza
      if(climber->is_left_child)
        huffman_code = "0" + huffman_code;
      else
        huffman_code = "1" + huffman_code;
      climber = climber->parent;
    }
   // cout << huffman_code << std::endl;
  }
}
```

2.18.3. Εκτίμηση αλγορίθμου

Διάγραμμα 18



Σχήμα 2.18.2 : Σύγκριση υλοποιημένου Huffman με 3 διαφορετικές δομές

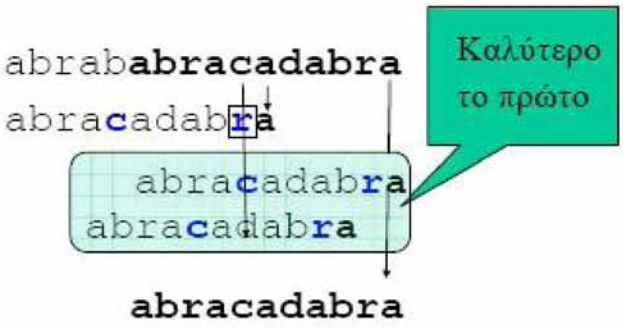
2.18.3.1. Επεξήγηση διαγράμματος

Ο οριζόντιος άξονας δείχνει το μέγεθος εισόδου, τον αριθμό δηλαδή χαρακτήρων του κειμένου , και ο κάθετος άξονας το χρόνο τρεξίματος του αλγορίθμου για το αντίστοιχο μέγεθος εισόδου. Απ' τους χρόνους που καταγράφει ο αλγόριθμος και με τις τρεις διαφορετικές δομές σωρού, ήτοι Fibonacci heap, k-nary heap και binary heap, είναι προφανές ότι οι διαφορές μεταξύ αυτών είναι αμελητέες και θα ήταν άστοχο να γίνει οποιαδήποτε κατάταξη καταλληλότητας. Και οι τρεις υλοποιήσεις λύνουν το πρόβλημα της συμπίεσης κειμένου αποτελεσματικά και είναι αδύνατο μέσω του Huffman, έστω μία εξ' αυτών να τεθεί ως καλύτερη, από τη στιγμή που οι διαφορές χρόνων τρεξίματος αγγίζουν την τάξη των χιλιοστών του δευτερολέπτου.

2.19. ΑΛΓΟΡΙΘΜΟΣ BOYER - MOORE

2.19.1. Ανάλυση αλγορίθμου

Ο αλγόριθμος λύνει το πρόβλημα του ακριβούς ταιριάσματος προτύπου που ορίζεται ως εξής: Δοθέντος, μιας συμβολοσειράς T, μήκους n, και μιας συμβολοσειράς – πρότυπο P, ζητείται ο εντοπισμός όλων των εμφανίσεων του προτύπου στο κείμενο. Κατά τη διαδικασία, ένα παράθυρο μήκους m, τοποθετείται επί του κειμένου. Ο έλεγχος ταιριάσματος ή όχι των χαρακτήρων ξεκινά από το αριστερό άκρο του προτύπου και όσο υπάρχει επιτυχία, μετακινείται προς τα δεξιά. Όταν σημειωθεί αποτυχία, η διαδικασία επανεκινείται, αφού το παράθυρο ολισθήσει κατά έναν αριθμό θέσεων προς τα δεξιά, το μέγιστο από τις δύο τιμές που υπαγορεύουν δύο συγκεκριμένα ευρετικά κριτήρια. Στο **σχήμα 2.19.1**, φαίνεται ένα στιγμιότυπου τρεξίματος του αλγορίθμου, κατά το οποίο αναζητείται η εμφάνιση του abracadabra στο κείμενο abrababracadabra. Μετά από ένα ταίριασμα κατά την αρχική τοποθέτηση του παραθύρου, σημειώνεται η πρώτη αποτυχία. Έπειτα, βάση πρώτου κριτηρίου η ολίσθηση είναι 6 ενώ κατά το δεύτερο 4, άρα και επιλέγεται το πρώτο. Ο αλγόριθμος συνεχίζει ανάλογα.



Σχήμα 2.19.1 : Παράδειγμα εύρεσης ακριβούς ταιριάσματος προτύπου κατά BOYER - MOORE

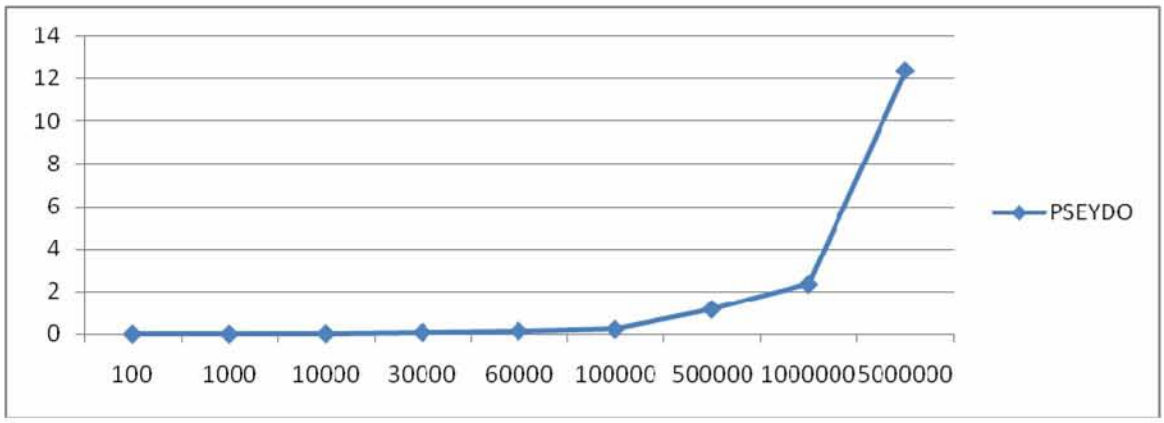
2.19.2. Υλοποίηση αλγορίθμου με χρήση βιβλιοθήκης LEDA

```
#include <iostream>
#include <cstdio>
#include <C:LEDA\core\array.h>
#include <C:LEDA\core\list.h>


using namespace leda;
using std::cout;
using std::cin;
using std::endl;

int bmPM(array<char>& T,array<char>& P,array<char>& S);
void occurBM(array<char>& P,array<char>& S,array<int>& d);
void matchMB(array<char>& P,array<int>& match);
int c2i(char c,array<char>& S);


int main() {
	return 0;
}

int bmPM(array<char>& T,array<char>& P,array<char>& S) {
	int n=T.size();
	int m=P.size();
	array<int> d(S.size());
	occurBM(P,S,d);                                    //sunarthsh
emfanisews
	array<int> match(1,m);
	matchMB(P,match);                             //sunarthsh
tairiasmatos
	int i=m;
	while(i<=n) {
		int j=m;
		while((j>0) && (P[j]=T[i])) { //tairiasma xarakthrwn
			j--;
			i--;
		}
		if(j==0)                                   //vrethike
			return i+1;
		else                                       //olisthisi tou
parathirou kata to megisto
			i+=max(match[j],d[T[i]]);
	}
	return -1;
}

void occurBM(array<char>& P,array<char>& S,array<int>& d) {
	int m=P.size();
	for(int k=0;k<S.size();k++)
		d[k]=0;
	for(int k=1;k<=P.size();k++)
		d[c2i(P[k],S)]=m-k;                        //h c2i
metatrepei ton xarakthra
```

```
}                                                                    //ston
arithmo diataksews tou sto S

void matchMB(array<char>& P,array<int>& match) {
      int m=P.size();
      array<int> fsuf(P.size()+1);
      for(int k=1;k<=P.size();k++)          //arxikopoiisi
            match[k]=m+1;                                  //me akiri timi
olisthisews
      fsuf[m]=m+1;
      for(int k=m-1;k>=0;k--) {                  //upologismos
sunarthshs apotuxias
            int j=fsuf[k+1];
            while(j<=P.size()) {
                  if(P[k+1]==P[j])
                        break;
                  match[j]=min(match[j],j-(k+1));
                  j=fsuf[j];
            }
            fsuf[k]=j-1;
      }
      int l=1; int shft=fsuf[0];
      while(shft<=P.size()) {                    //olisthisi vasei
megistou tairiasmatos
            for(int k=l;k<=shft;k++)             //prothematos se
prosfuma ean den
                  match[k]=min(match[k],shft);//vrethike endiamesh
uposumboloseira
            l=shft+1;
            shft=fsuf[shft];
      }
      for(int j=1;j<=P.size();j++)          //prosthesh sthn olisthisi
tou plhthous
            match[j]+=P.size()-j;                //twn xarakthrwn pou
exoun tairiastei
}

int c2i(char c,array<char>& S) {
      return S.binary_search(c);
}
```

2.19.3. Εκτίμηση αλγορίθμου

Διάγραμμα 19



Σχήμα 2.19.2 : Εκτίμηση υλοποιημένου BOYER - MOORE

2.19.3.1. Επεξήγηση διαγράμματος

Ο οριζόντιος άξονας δείχνει το μέγεθος εισόδου στον αλγόριθμο, δηλαδή το μέγεθος του κειμένου συμβολοσειρών. Ο κάθετος άξονας δείχνει το χρόνο τρεξίματος του αλγορίθμου για το αντίστοιχο μέγεθος εισόδου. Απ' το σχήμα μπορούμε να συμπεραίνουμε ότι η υλοποίηση δεν καταναλώνει πολύ μνήμη, παράγοντας το επιθυμητό αποτέλεσμα ακόμα και για μεγάλα μεγέθη, απ' την άλλη δε, εκτελεί πολλές πράξεις (ενθέσεις, συγκρίσεις), που καταναλώνουν πολύ χρόνο.

ΚΕΦΑΛΑΙΟ 3. ΣΥΜΠΕΡΑΣΜΑΤΑ

Οι αλγόριθμοι που υλοποιήθηκαν, είναι αλγόριθμοι γνωστοί, οι περισσότεροι από τους οποίους έχουν υλοποιηθεί σε πολλές γλώσσες προγραμματισμού, από διαφορετικούς προγραμματιστές. Καλύπτουν ευρύ φάσμα εφαρμογών σε πολλά πεδία (τηλεπικοινωνίες, ανάλυση δικτύων κ.α.) καθώς επιλύουν σημαντικά προβλήματα όπως εύρεση ελάχιστων επικαλύπτων δέντρων, συντομότερων μονοπατιών, μέγιστης ροής δικτύων, ομοιότητα συμβολοσειρών και πολλά άλλα. Η υλοποίηση των αλγορίθμων αυτών απαιτεί πλήθος δομών, ώστε να φέρουν εις πέρας τα ζητούμενα αποτελέσματα, πολλές εξ' αυτών είναι πολύπλοκες και δύσκολο να υλοποιηθούν από τις διάφορες γλώσσες προγραμματισμού. Το περιβάλλον της βιβλιοθήκης LEDA ωστόσο, όχι μόνο διαθέτει έτοιμες υλοποιημένες τις περισσότερες από τις παραπάνω δομές, δίνοντας στο χρήστη τη δυνατότητα να τις χρησιμοποιήσει με ευκολία σε οποιαδήποτε υλοποίηση αυτός θέλει, αλλά παρέχει επίσης, πλήθος σημαντικών αλγορίθμων, εγγυώμενο για την αποτελεσματικότητα και πιστότητα αυτών.
Η LEDA παρέχει δομές όπως πίνακες μονοδιάστατους ή δισδιάστατους, λίστες γραμμικά ή κυκλικά συνδεδεμένες, σύνολα, λεξικά , ουρές απλές, ουρές προτεραιοτήτων, πολλών ειδών γραφήματα και άλλες πολλές οι οποίες είναι δύσκολο να υλοποιηθούν και να διαχειριστούν σε μία γλώσσα προγραμματισμού. Οι αλγόριθμοι που υλοποιήθηκαν κατά την εργασία, έκαναν ευρεία χρήση των δομών αυτών και απέδειξαν ότι περίπλοκες εφαρμογές – αλγόριθμοι , είναι δυνατόν να αναπτυχθούν εύκολα και εγγυημένα σωστά, στο περιβάλλον της LEDA. Μάλιστα, στους αλγορίθμους αυτούς, οι δομές που χρησιμοποιήθηκαν δοκιμάστηκαν με διάφορες παραμέτρους κάθε φορά, δεδομένου ότι η LEDA δίνει τη δυνατότητα αυτή στους χρήστες της, με στόχο τον έλεγχο και εκτίμηση της αποτελεσματικότητάς τους. Τα δεδομένα έδειξαν ότι η χρήση διαφορετικών παραμέτρων, είτε στις δομές είτε στις έτοιμες συναρτήσεις, επιφέρουν διαφορετικά αποτελέσματα, κάτι που σημαίνει ότι ο χρήστης κάθε φορά μπορεί να χρησιμοποιεί τη δομή ή τον αλγόριθμο που θέλει, με τον τρόπο που του επιφέρει τα μέγιστα δυνατά αποτελέσματα.
Όσον αφορά την εκτίμηση των προκειμένων, υλοποιημένων σε περιβάλλον LEDA, αλγορίθμων, τα συμπεράσματα είναι εμφανή σχεδόν σε κάθε διάγραμμα που παρουσιάστηκε. Στις συγκρίσεις μεταξύ των απλά υλοποιημένων αλγορίθμων και αυτών των έτοιμων συναρτήσεων της LEDA, διαπιστώνεται εύκολα η τεράστια υπεροχή των δεύτερων. Ο λόγος είναι σαφής: η LEDA πέρα από την καλύτερη διαχείριση μνήμης που διαθέτει, χρησιμοποιεί τις καταλληλότερες κάθε φορά δομές για την υλοποίηση του κάθε αλγορίθμου. Έτσι, αφενός πετυχαίνει καλύτερους χρόνους τρεξίματος από τους αντίστοιχους απλά υλοποιημένους αλγόριθμους, αφετέρου μπορεί να παράγει αποτελέσματα για πολύ μεγαλύτερα μεγέθη εισόδου. Δείγμα της ορθής υλοποίησης των έτοιμων συναρτήσεων της LEDA, είναι οι σταθερές αλλά μικρές διαφορές χρόνου τρεξίματος στις αυξήσεις μεγέθους εισόδου.
Οι δομές που χρησιμοποιούν κάθε φορά οι διάφορες υλοποιήσεις, επηρεάζουν αντίστοιχα και τους χρόνους που καταγράφουν. Οι πίνακες και μάλιστα οι δισδιάστατοι, επιφέρουν τα χειρότερα αποτελέσματα σε χρόνους και μνήμη, ενώ

οι λίστες, οι ουρές ή στοίβες, προσφέρουν γρηγορότερες πράξεις και αντοχή σε μεγαλύτερα μεγέθη εισόδου. Επίσης, οι διάφορες υλοποιήσεις των δομών που υπάρχουν, συνήθως δίνουν καλύτερα αποτελέσματα για τα περισσότερα μεγέθη εισόδου, για παράδειγμα οι bounded queue έναντι της απλής queue, ή η s_list έναντι της απλής list, στο τέλος όμως, υπάρχει περίπτωση να συγκλίνουν στους ίδιους χρόνους.
Γενικό συμπέρασμα, είναι ότι η χρήση της βιβλιοθήκης LEDA, διευκολύνει κατά πολύ την ανάπτυξη ακόμα και των πιο δύσκολων αλγορίθμων, προσφέροντας εγγύηση για την αποτελεσματικότητα των συναρτήσεών της, σε θέματα χρόνου και μνήμης καθώς και για την πιστότητα αυτών.

ΒΙΒΛΙΟΓΡΑΦΙΑ

1. Παναγιώτης Δ. Μποζάνης, (2003) , *ΑΛΓΟΡΙΘΜΟΙ – Σχεδιασμός και Ανάλυση,* Θεσσαλονίκη , Εκδόσεις Τζιόλα.
2. Kurt Mehlhorn, Stefan Naher, (1999), *LEDA, A Platform for Combinatorial and Geometric Computing, Cambridge University Press.*
3. Φ. Λούκος, Π. Μποζάνης, *LEDA INTRO.ppt*
4. www.algorithmic-solutions.com
5. www.umich.edu/~me558/manuals/LEDA/MANUAL.html
6. www.comp.nus.edu.sg
7. www.leda-tutorial.org
8. www.cs.miami.edu