ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Παραλληλοποίηση Αλγορίθμου Branch and Bound σε Υπολογιστικά Συστήματα Μοιραζόμενης Μνήμης

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Αλέξανδρος Νικόλαος Β. Ζιώγας

Επιβλέπων : Γεώργιος Ι. Γκούμας

Λέκτορας Ε.Μ.Π.

Αθήνα, Φεβρουάριος 2016

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Παραλληλοποίηση Αλγορίθμου Branch and Bound σε Υπολογιστικά Συστήματα Μοιραζόμενης Μνήμης

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Αλέξανδρος Νικόλαος Β. Ζιώγας

Επιβλέπων : Γεώργιος Ι. Γκούμας

Λέκτορας Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 23η Μαρτίου 2016.

Αθήνα, Φεβρουάριος 2016

.....	.....	.....
Γεώργιος Ι. Γκούμας	Νεκτάριος Κοζύρης	Δημήτριος Σούντρης
Λέκτορας Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Αν. Καθηγητής Ε.Μ.Π.

.....

Αλέξανδρος Νικόλαος Β. Ζιώγας

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Στόχος της διπλωματικής εργασίας είναι η μελέτη του τρόπου παραλληλοποίησης του αλγορίθμου Branch and Bound σε υπολογιστικά συστήματα μοιραζόμενης μνήμης και η μέτρηση της επιτάχυνσης της εκτέλεσης που είναι δυνατό να επιτευχθεί. Για το σκοπό αυτό, αναπτύχθηκε προγραμματιστική βιβλιοθήκη σε γλώσσα C++ για την επίλυση προβλημάτων βελτιστοποίησης με τη μέθοδο Branch and Bound και υλοποιήθηκαν με βάση αυτή αλγόριθμοι για την επίλυση του διακριτού προβλήματος του σακιδίου και του προβλήματος του πλανόδιου πωλητή. Έγιναν μετρήσεις της απόδοσης του παράλληλου αλγορίθμου και της βιβλιοθήκης σε υπολογιστικό σύστημα με δύο επεξεργαστές Intel Xeon E5-2697 v3, με ιδιαίτερη έμφαση στον τρόπο υλοποίησης του δένδρου του αλγορίθμου Branch and Bound, με ομάδες ουρών προτεραιότητας διαφόρων χαρακτηριστικών. Προέκυψε ότι τεχνικές, όπως η χρήση ιδιωτικών ουρών με δυνατότητα κλοπής εργασιών, ιδιωτικά όρια, ανάρτηση νημάτων, καθώς και εκμετάλλευση της τεχνολογίας ταυτόχρονης πολυνηματοποίησης, μπορούν να βελτιώσουν σημαντικά την επιτάχυνση που μπορεί να επιτευχθεί.

Λέξεις Κλειδιά

Branch and Bound, Παραλληλοποίηση, Υπολογιστικά Συστήματα Μοιραζόμενης Μνήμης, Διακριτό Πρόβλημα του Σακιδίου, Πρόβλημα του Πλανόδιου Πωλητή, Ουρές Προτεραιότητας, Κλοπή Εργασιών, Ανάρτηση Νημάτων, Ταυτόχρονη Πολυνηματοποίηση

Abstract

The scope of this thesis is the parallelization of the Branch and Bound algorithm in computing systems with shared memory and the measurement of the speed-up that may be achieved. For this purpose, a programming library was developed in C++, in order to solve optimization problems with the Branch and Bound method. Using this library, more algorithms were developed that solve the discrete knapsack problem and the problem of the travelling salesman. Benchmarks were done in a computing system with two Intel Xeon E5-2697 v3 processors, where emphasis was given to the implementation of the Branch and Bound algorithm tree, with groups of priority queues of various characteristics. The results exhibited that techniques, such as the use of priority queues with work stealing capabilities, private bounds, thread pinning and taking advantage of simultaneous multi-threading, may improve the speed-up that can be achieved significantly.

Key Words

Branch and Bound, Parallelization, Computing Systems with Shared Memory, Discrete Knapsack Problem, Travelling Salesman Problem, Priority Queues, Work Stealing, Thread Pinning, SMT

Ευχαριστίες

Θα ήθελα να ευχαριστήσω θερμά τους Γεώργιο Γκούμα, Λέκτορα της σχολής και Αλέξανδρο Δημόπουλο, επιστημονικό συνεργάτη, για την πολύτιμη βοήθεια και καθοδήγηση τους, που δίχως αυτή δε θα μπορούσε να ολοκληρωθεί αυτή η εργασία.

Πίνακας περιεχομένων

Πίνακας Εικόνων

1 Αλγόριθμος Branch and Bound

1.1 Γενικά

Με τον όρο «**Branch and Bound**» (προτεινόμενη μετάφραση «**Διακλάδωσε και Φράξε**», συντομογραφικά **B&B**) περιγράφεται μια κατηγορία αλγορίθμων, που χρησιμοποιούνται για την επίλυση διακριτών και συνδυαστικών προβλημάτων βελτιστοποίησης. Η μέθοδος προτάθηκε από τους (Land & Doig, 1960) για την επίλυση προβλημάτων διακριτού προγραμματισμού. Η ονομασία «Branch and Bound» χρησιμοποιήθηκε για πρώτη φορά στην εργασία (Little, Murty, Sweeney, & Karel, 1963) για το «**Πρόβλημα του Πλανόδιου Πωλητή**» (**Traveling Salesman Problem**, συντομογραφικά **TSP**). Ο αλγόριθμος B&B απαριθμεί συστηματικά τις υποψήφιες λύσεις του προβλήματος, με αναζήτηση στο χώρο καταστάσεων σε δενδρική μορφή. Το σύνολο όλων των υποψήφιων λύσεων βρίσκεται αρχικά στη ρίζα του δένδρου και από εκεί διακλαδώνεται διαδοχικά σε υποσύνολα, έτσι ώστε τελικά κάθε φύλλο του δένδρου να περιέχει μία μόνο υποψήφια λύση του προβλήματος. Προτού όμως γίνει διακλάδωση οποιουδήποτε κόμβου του δένδρου σε επιμέρους υποσύνολα λύσεων, ο κόμβος-κλαδί συγκρίνεται με προσεγγιστικά υπολογισμένα άνω και κάτω όρια της βέλτιστης λύσης του προβλήματος. Αν από αυτή τη σύγκριση προκύψει πως ο κόμβος-κλαδί δε μπορεί να παραγάγει καλύτερη λύση από αυτή που έχει ήδη βρεθεί, τότε αυτός και το υποδένδρο κάτω από αυτόν απορρίπτονται.

1.2 Μαθηματική Διατύπωση

Έστω η συνάρτηση $f: \mathcal{X} \to \mathbb{R}$, φραγμένη στο πεδίο ορισμού της, ώστε να έχει μέγιστη ή ελάχιστη τιμή σε αυτό και ένα σύνολο περιορισμών $\boldsymbol{g}(\boldsymbol{x}) \leq \boldsymbol{b}, \boldsymbol{x} \in \mathcal{X}$. Αν η συνάρτηση έχει μέγιστη τιμή, τότε μπορεί να οριστεί το παρακάτω πρόβλημα βελτιστοποίησης, τηρουμένων των περιορισμών:

$$\max f(\boldsymbol{x})$$

$$\boldsymbol{g}(\boldsymbol{x}) \leq \boldsymbol{b}$$

$$\boldsymbol{x} \in \mathcal{X}$$

Εναλλακτικά, αν η συνάρτηση έχει ελάχιστη τιμή, μπορεί να οριστεί ένα δεύτερο πρόβλημα βελτιστοποίησης, τηρουμένων των περιορισμών:

$$\min f(\boldsymbol{x})$$

$$\boldsymbol{g}(\boldsymbol{x}) \leq \boldsymbol{b}$$

$$\boldsymbol{x} \in \mathcal{X}$$

Για λόγους σύμβασης όμως, θα χρησιμοποιηθεί μόνο ο πρώτος ορισμός. Στην περίπτωση που είναι επιθυμητή η αναζήτηση της ελάχιστης τιμής, τότε θα ορίζεται μία δεύτερη συνάρτηση $\hat{f}: \mathcal{X} \to \mathbb{R}$, τέτοια ώστε το πρόβλημα βελτιστοποίησης να δίνεται ως εξής:

$$\hat{f}(\boldsymbol{x}) = -f(\boldsymbol{x})$$

$$\max \hat{f}(x)$$

$$\boldsymbol{g}(\boldsymbol{x}) \leq \boldsymbol{b}$$

$$\boldsymbol{x} \in \mathcal{X}$$

Η εφαρμογή του αλγορίθμου B&B προϋποθέτει τη δυνατότητα δημιουργίας ενός δεύτερου προβλήματος βελτιστοποίησης, κατόπιν χαλάρωσης του αρχικού με μία σειρά από μεθόδους, όπως (Vizvári, 2011):

- Χαλάρωση των μη αλγεβρικών περιορισμών, δηλαδή $\boldsymbol{x} \in \mathcal{Y}, \mathcal{X} \subseteq \mathcal{Y}$
- Χαλάρωση των αλγεβρικών περιορισμών με τη δημιουργία των υποκατάστατων περιορισμών

$$\lambda_i \geq 0, i = 1, \dots, m$$

$$\sum_{i=1}^{m} \lambda_i \boldsymbol{g}_i(\boldsymbol{x}) \leq \sum_{i=1}^{m} \lambda_i b_i$$

- Χαλάρωση Lagrange με τη απαλοιφή των αλγεβρικών περιορισμών και ένταξη τους στη συνάρτηση

$$\lambda_i \geq 0, i = 1, \dots, m$$

$$\max f(\boldsymbol{x}) + \sum_{i=1}^{m} \lambda_\iota (b_i - \boldsymbol{g}_i(\boldsymbol{x}))$$

$$\boldsymbol{x} \in \mathcal{X}$$

Το παραγόμενο αυτό πρόβλημα ονομάζεται χαλαρωμένο πρόβλημα. Οποιαδήποτε μέθοδος χαλάρωσης και να χρησιμοποιηθεί, πρέπει να ισχύουν τα εξής:

- Όλες οι εφικτές λύσεις του αρχικού προβλήματος είναι εφικτές και στο χαλαρωμένο πρόβλημα. Σημειώνεται πως εφικτές είναι οι λύσεις που δεν παραβιάζουν τους περιορισμούς.
- Η βέλτιστη λύση του χαλαρωμένου προβλήματος είναι ένα άνω όριο της βέλτιστης λύσης του αρχικού προβλήματος.

1.3 Αλγόριθμος

Εφόσον ισχύουν τα παραπάνω, το χαλαρωμένο πρόβλημα μπορεί να χρησιμοποιηθεί για τον υπολογισμό του άνω ορίου της βέλτιστης λύσης που μπορεί να παραχθεί από οποιαδήποτε υποσύνολο λύσεων του δένδρου του αλγορίθμου B&B. Προαιρετικά, ορίζεται και ένα «κάτω όριο» ως η ελάχιστη τιμή που μπορεί να λάβει η βέλτιστη λύση και υπολογίζεται βρίσκοντας μία οποιαδήποτε εφικτή λύση από το υποσύνολο των λύσεων του αρχικού προβλήματος. Επίσης, πρέπει να ορίζεται σε οποιαδήποτε υποσύνολο λύσεων πράξη διακλάδωσης σε 2 ή περισσότερα

νέα υποσύνολα. Συνήθως, οι τομές αυτών των νέων υποσυνόλων είναι κενές, όμως αυτό δεν είναι απαραίτητο για τη σωστή λειτουργία του αλγορίθμου.

Αναλυτικά, τα βήματα του αλγορίθμου είναι τα εξής:

Έστω S το σύνολο όλων των υποψήφιων λύσεων και Q μία δομή στην οποία αποθηκεύονται τα υποσύνολα των λύσεων. Αρχικά $Q = \{S\}$. Έστω επίσης S^B η καλύτερη λύση που έχει βρεθεί μέχρι στιγμής. Αρχικά $S^B = -\infty$. Προαιρετικά, ορίζεται και το καλύτερο κάτω όριο που έχει βρεθεί, L^B.

1. Αν η Q δεν είναι άδεια, εξάγεται ένα υποσύνολο λύσεων S^C, διαφορετικά ο αλγόριθμος τερματίζει.
2. Εφαρμόζεται η πράξη της διακλάδωσης στο S^C και παράγονται τα υποσύνολα λύσεων $S_i^C, i = 1, \dots, n$. Για κάθε ένα από τα νέα υποσύνολα:
    a. Αν το υποσύνολο περιλαμβάνει μία μοναδική λύση, τότε εφόσον είναι εφικτή εφαρμόζεται η πράξη $S^B = \max\left(S^B, \left|S_i^C\right|\right)$.
    b. Διαφορετικά, υπολογίζονται τα άνω και κάτω όρια του υποσυνόλου, U_i^C και L_i^C. Αν $U_i^C > S^B$ και $U_i^C > L^B$ τότε εφαρμόζονται οι πράξεις $L^B = \max\left(L^B, L_i^C\right)$ και $Q = Q \cup S_i^C$. Αλλιώς, το υποσύνολο απορρίπτεται.
3. Επιστροφή στο βήμα 1.

Όταν ολοκληρωθεί η εκτέλεση του αλγορίθμου, η τιμή S^B είναι η βέλτιστη λύση του προβλήματος. Η παραπάνω μορφή του αλγορίθμου συχνά χαρακτηρίζεται ως «**πρόθυμη**» (**eager**), γιατί υπολογίζονται τα άνω και κάτω όρια των υποσυνόλων μόλις αυτά παραχθούν. Υπάρχει αντίστοιχα και δυνατότητα «**οκνηρού**» (**lazy**) υπολογισμού των ορίων ως εξής (Clausen, 1999):

1. Αν η Q δεν είναι άδεια, εξάγεται ένα υποσύνολο λύσεων S^C, διαφορετικά ο αλγόριθμος τερματίζει.
2. Αν το υποσύνολο περιλαμβάνει μία μοναδική λύση, τότε εφόσον είναι εφικτή εφαρμόζεται η πράξη $S^B = \max(S^B, |S^C|)$ και γίνεται επιστροφή στο βήμα 1.
3. Διαφορετικά, υπολογίζονται τα άνω και κάτω όρια του υποσυνόλου, U^C και L^C. Αν $U^C > S^B$ και $U^C > L^B$ τότε εφαρμόζεται οι πράξη $L^B = \max(L^B, L^C)$. Αλλιώς το υποσύνολο απορρίπτεται και γίνεται επιστροφή στο βήμα 1.
4. Εφαρμόζεται η πράξη της διακλάδωσης στο S^C και παράγονται τα υποσύνολα λύσεων $S_i^C, i = 1, \dots, n$. Για κάθε ένα από τα νέα υποσύνολα εφαρμόζεται η πράξη $Q = Q \cup S_i^C$.
5. Επιστροφή στο βήμα 1.

1.4 Διάσχιση του δένδρου του αλγορίθμου B&B

Η διάσχιση των κόμβων του δένδρου του αλγορίθμου B&B γίνεται είτε κατά βάθος (**Depth First Search**, συντομογραφικά **DFS**) ή με σειρά προτεραιότητας με βάση κάποιο κριτήριο (**Best First Search**, συντομογραφικά **BestFS**), π.χ. κάποιο εκ των ορίων. Κανένας από τους αλγορίθμους διάσχισης του δένδρου δεν παρουσιάζει εγγενώς καλύτερη απόδοση, όμως έχουν διαφορετικά πλεονεκτήματα και μειονεκτήματα (Mahapatra & Dutt, 1996). Ο BestFS, είναι πιθανό να οδηγήσει γρήγορα στη βέλτιστη λύση, εφόσον το επιλεγμένο κριτήριο έχει καλή απόδοση. Πολλές φορές όμως ο αλγόριθμος μετακινείται συνεχώς μεταξύ εσωτερικών κόμβων του δένδρου, χωρίς να φθάνει σε φύλλα. Τα τελευταία όμως είναι απαραίτητα για την εύρεση

εφικτών λύσεων, που μπορούν να χρησιμοποιηθούν ώστε να απορριφθούν υποσύνολα λύσεων με βάση το άνω όριο. Έτσι, μπορεί να αυξηθεί υπερβολικά το μέγεθος της δομής αποθήκευσης των υποσυνόλων και να προκληθούν τα σχετικά προβλήματα μνήμης. Ο DFS, αντίθετα, βρίσκει πολύ γρήγορα τελικές εφικτές λύσεις, αλλά με σχετικά τυχαίο τρόπο. Συνεπώς, είναι πιθανό να κάνει περιττές διακλαδώσεις. Η επιλογή του αλγορίθμου διάσχισης του δένδρου επηρεάζει και την υλοποίηση της δομής αποθήκευσης των υποσυνόλων λύσεων. Στην περίπτωση του DFS χρησιμοποιείται **στοίβα** (**stack**), ενώ στον BestFS η δομή υλοποιείται με **ουρά προτεραιότητας** (**priority queue**).

1.5 Παράδειγμα Αλγορίθμου Branch and Bound: Διακριτό Πρόβλημα Σακιδίου

Στο «**Διακριτό Πρόβλημα του Σακιδίου**» («**Discrete Knapsack Problem**», συντομογραφικά **DKP**) υπάρχει ένα σύνολο αντικειμένων, το καθένα με συγκεκριμένη αξία και βάρος, καθώς και ένα σακίδιο στο οποίο μπορούν να εισαχθούν αντικείμενα μέχρι ενός ορισμένου συνολικού βάρους. Ζητείται το υποσύνολο των αντικειμένων που μεγιστοποιούν την αξία του σακιδίου, χωρίς να παραβιάζουν τον περιορισμό του βάρους. Σε μαθηματική μορφή το πρόβλημα δίνεται ως εξής:

$$\max \sum_{i=1}^{n} v_i x_i$$

$$\sum_{i=1}^{n} w_i x_i \leq c, v_i > 0, w_i > 0, c > 0$$

$$x_i \in \{0, 1\}, i = 1, \dots, n$$

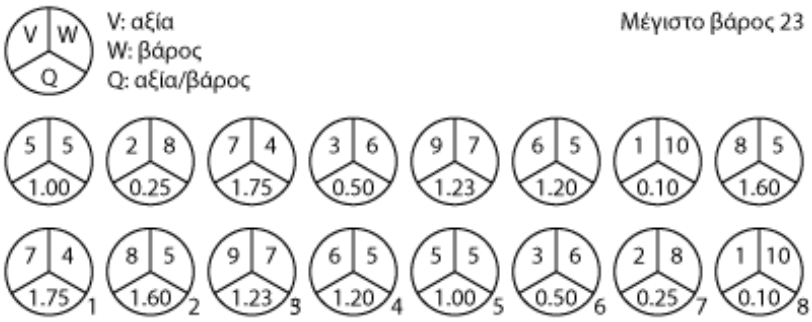
Παραπάνω, το πλήθος των αντικειμένων είναι n, v_i είναι η αξία του ι-οστού αντικειμένου, w_i είναι το βάρος του ι-οστού αντικειμένου και c ο περιορισμός για το συνολικό βάρος του σακιδίου. Το πρόβλημα ονομάζεται διακριτό, γιατί η μεταβλητή x_i, που δηλώνει το μέρος του ι-οστού αντικειμένου που εισάγεται στο σακίδιο, είναι διακριτή και μπορεί να λάβει μόνο τις τιμές 0 και 1. Δηλαδή, είτε το ι-οστό αντικείμενο εισάγεται στο σακίδιο ή αποκλείεται από αυτό.

Ένας τρόπος επίλυσης του DKP με αλγόριθμο B&B είναι η χαλάρωση των μη αλγεβρικών περιορισμών. Ορίζεται έτσι το εξής πρόβλημα, όπου η μεταβλητή x_i μπορεί να λάβει οποιαδήποτε πραγματική τιμή:

$$\max \sum_{i=1}^{n} v_i x_i$$

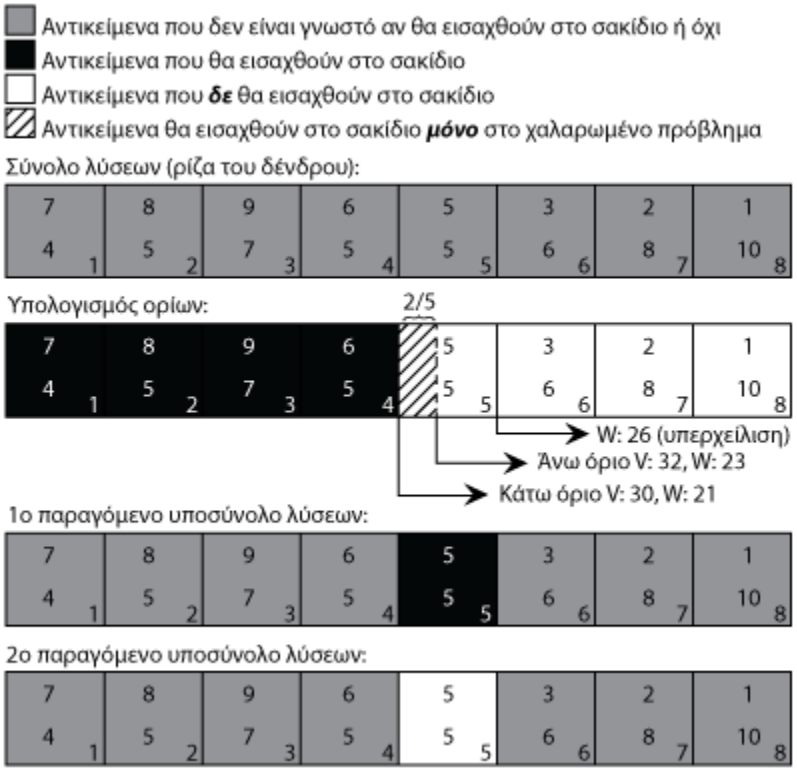
$$\sum_{i=1}^{n} w_i x_i \leq c, v_i > 0, w_i > 0, c > 0$$

$$x_i \in \mathbb{R}, i = 1, \dots, n$$



Εικόνα 1-1: Αριθμητικό παράδειγμα DKP

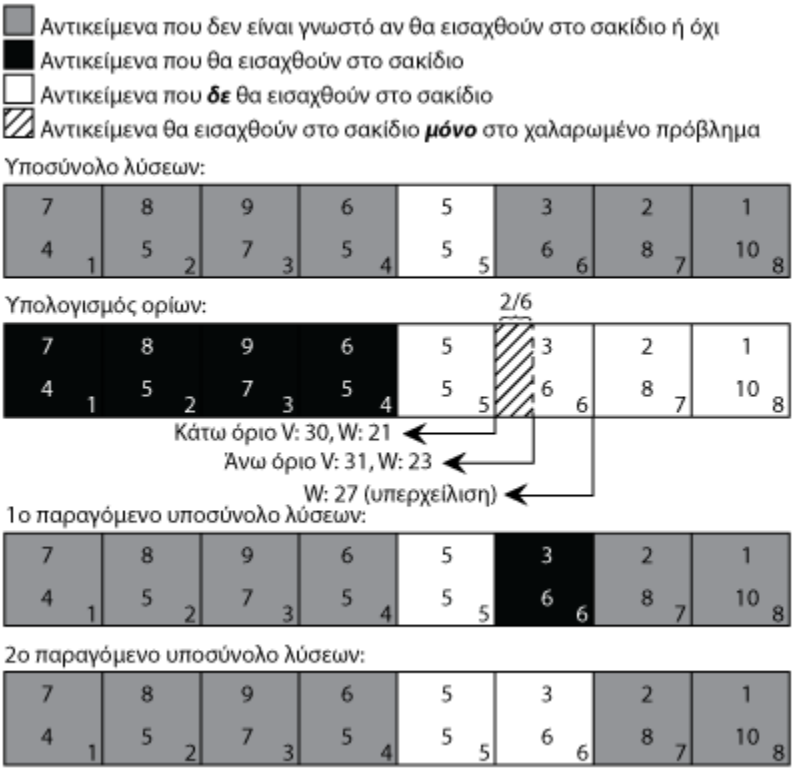
Η επίλυση του παραπάνω χαλαρωμένου προβλήματος είναι τετριμμένη. Αρχικά, γίνεται φθίνουσα ταξινόμηση των αντικειμένων ως προς την αξία τους ανά βάρος. Στη συνέχεια εφαρμόζεται «**άπληστος αλγόριθμος**» (**greedy algorithm**), όπου, ξεκινώντας από το πρώτο αντικείμενο του ταξινομημένου συνόλου, προστίθενται στο σακίδιο συνεχώς νέα αντικείμενα, μέχρι να παραβιαστεί το όριο βάρους (ή να τελειώσουν τα αντικείμενα). Το αντικείμενο, η προσθήκη του οποίου προκαλεί την υπερχείλιση βάρους, δεν εισάγεται τελικά στο σακίδιο ολόκληρο, παρά μόνο ένα κλάσμα του, τόσο ώστε το συνολικό βάρος του σακιδίου να γίνεται ίσο με τον περιορισμό. Αυτό είναι και το άνω όριο του αρχικού προβλήματος.



Εικόνα 1-2: Υπολογισμός ορίων και διακλάδωση της ρίζας

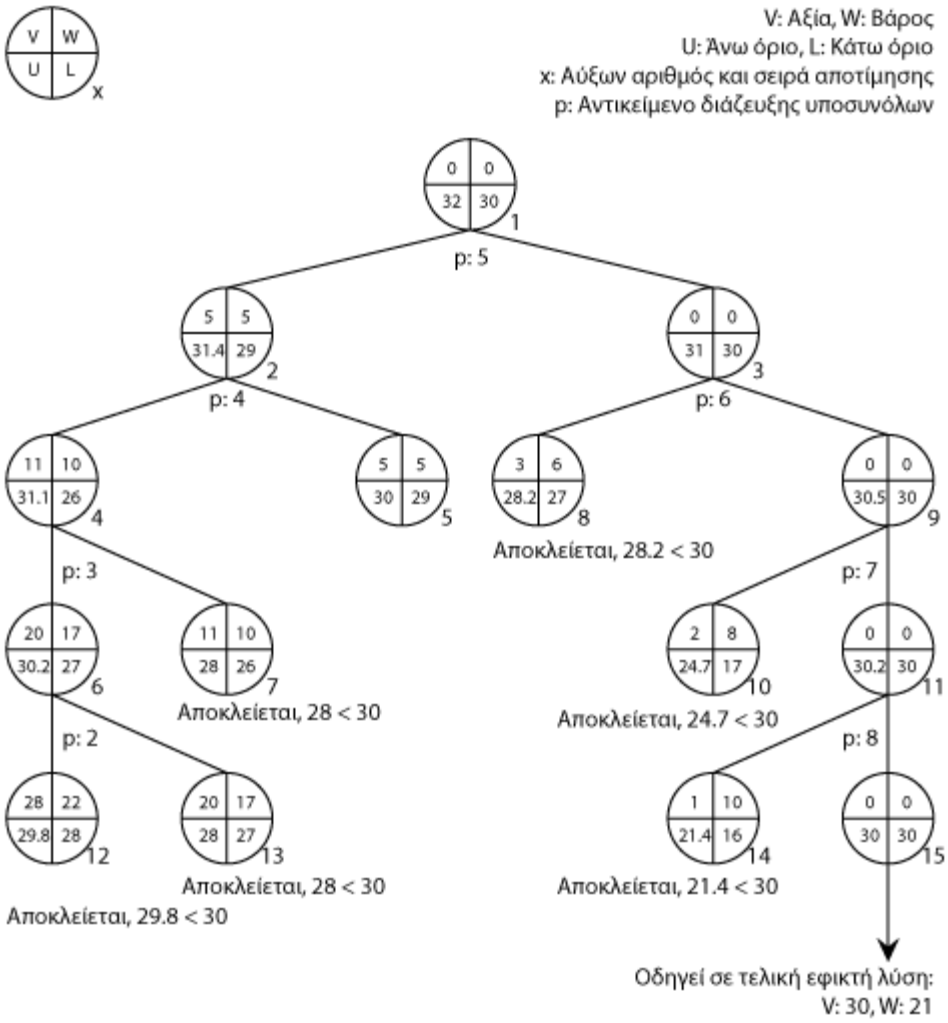
Το κάτω όριο βρίσκεται ακόμα πιο εύκολα, αφαιρώντας τελείως το τελευταίο αντικείμενο. Με αυτό τον τρόπο καταλήγουμε γρήγορα σε εφικτή λύση του DKP. Ως πράξη διακλάδωσης ορίζεται η διαίρεση του εκάστοτε υποσυνόλου λύσεων σε 2 με βάση το αντικείμενο που προκαλεί την

παραβίαση του ορίου βάρους. Δηλαδή, το πρώτο παραγόμενο υποσύνολο περιλαμβάνει όλες τις λύσεις στις οποίες το αντικείμενο εισάγεται τελικά στο σακίδιο, ενώ στις λύσεις του δευτέρου το αντικείμενο υποχρεωτικά αποκλείεται.



Εικόνα 1-3: Υπολογισμός ορίων και διακλάδωση υποσυνόλου

Στην **Error! Reference source not found.**εικόνα 1-1 δίνεται ένα αριθμητικό παράδειγμα με 8 αντικείμενα. Στη συνέχεια γίνεται ο υπολογισμός των ορίων για το σύνολο των λύσεων του προβλήματος (εικόνα 1-2). Αν εισαχθούν στο σακίδιο τα αντικείμενα 1 ως 4, η συνολική αξία γίνεται 30 και το βάρος 21. Αν εισαχθεί και το αντικείμενο 5, τότε το βάρος γίνεται 26 και παραβιάζει τον περιορισμό να έχει μέγιστη τιμή 23. Για να υπολογιστεί το άνω όριο, εισάγεται στο σακίδιο κλάσμα του αντικειμένου 5, τέτοιο ώστε το συνολικό βάρος να είναι 23. Έτσι, το κλάσμα του αντικειμένου 5 θα πρέπει να έχει βάρος 2, που αντιστοιχεί σε $\frac{2}{5}$. Συνεπώς, το άνω όριο της αξίας του σακιδίου είναι $30 + 5 * \frac{2}{5} = 32$. Το κάτω όριο είναι 30, δηλαδή η αξία του σακιδίου όταν δεν εισάγεται καθόλου το αντικείμενο 5. Στη συνέχεια, το σύνολο των λύσεων χωρίζεται σε 2 υποσύνολα. Το πρώτο περιλαμβάνει τις λύσεις, όπου το αντικείμενο 5 εισάγεται στο σακίδιο, ενώ στο δεύτερο υποσύνολο το αντικείμενο αποκλείεται. Στην **Error! Reference source not found.**εικόνα 1-3 δίνεται ο υπολογισμός των ορίων και η διακλάδωση του δευτέρου των υποσυνόλων λύσεων που περιεγράφηκαν αμέσως παραπάνω. Στην **Error! Reference source not found.**εικόνα 1-4 διαγράφεται η επίλυση του προβλήματος με eager αλγόριθμο B&B, με τη μέθοδο BestFS και κριτήριο το καλύτερο άνω όριο. Σημειώνεται ότι κατά την αποτίμηση του υποσυνόλου λύσεων με αύξων αριθμό 15, προκύπτει πως όλα τα υπολειπόμενα αντικείμενα εισάγονται στο σακίδιο δίχως παραβίαση των περιορισμών και συνεπώς τα 2 όρια είναι ίσα και καταλήγουν άμεσα σε τελική εφικτή λύση.



Εικόνα 1-4: Επίλυση αριθμητικού παραδείγματος με τη μέθοδο BestFS

1.6 Σκοπός

Στόχος της συγκεκριμένης διπλωματικής εργασίας είναι αρχικά η μελέτη του τρόπου παραλληλοποίησης του αλγορίθμου B&B σε υπολογιστικά συστήματα μοιραζόμενης μνήμης και κατόπιν η υλοποίηση σχετικών αλγορίθμων και μέτρηση της επιτάχυνσης της εκτέλεσης που είναι δυνατό να επιτευχθεί, δεδομένου των δυσκολιών (**bottlenecks**) που δύναται να παρουσιαστούν. Για το σκοπό αυτό, μελετήθηκε η σχετική βιβλιογραφία και αναπτύχθηκε προγραμματιστική βιβλιοθήκη σε γλώσσα C++ για την επίλυση προβλημάτων βελτιστοποίησης με τη μέθοδο B&B, η παρουσίαση των οποίων γίνεται στο επόμενο κεφάλαιο. Επιπλέον, για να μετρηθεί η απόδοση της βιβλιοθήκης, υλοποιήθηκαν με βάση αυτή αλγόριθμοι για την επίλυση του διακριτού προβλήματος του σακιδίου και του προβλήματος του πλανόδιου πωλητή. Τα αποτελέσματα παρουσιάζονται στο τρίτο κεφάλαιο.

2 Παραλληλοποίηση Αλγορίθμου Branch and Bound

2.1 Γενικά – Ορολογία

Για την καλύτερη κατανόηση της μελέτης απαιτούνται βασικές γνώσεις αρχιτεκτονικής υπολογιστών. Για λόγους πληρότητας και για δυνατότητα χρήσης της σχετικής ορολογίας παρακάτω, γίνεται συνοπτική περιγραφή ενός υπολογιστικού συστήματος μοιραζόμενης μνήμης. Το σύστημα περιλαμβάνει έναν ή περισσότερους (**μικρο**)**επεξεργαστές/κεντρικές μονάδες επεξεργασίας** ((**micro**)**processors/central processing units**, συντομογραφικά **CPUs**). Κάθε επεξεργαστής έχει έναν ή περισσότερους **πυρήνες** (**cores**), ενώ κάθε πυρήνας κανονικά μπορεί να εκτελεί σε κάθε χρονική στιγμή μόνο ένα/μία **νήμα/διεργασία** (**thread/process**). Οι σύγχρονοι επεξεργαστές συχνά διαθέτουν την τεχνολογία **ταυτόχρονης πολυνηματοποίησης** (**simultaneous multithreading**, συντομογραφικά **SMT**), που τους επιτρέπει να διατηρούν σε λειτουργία ταυτόχρονα περισσότερα του ενός νημάτων ανά πυρήνα. Τα λειτουργικά συστήματα που εκτελούνται σε υπολογιστικά συστήματα με SMT βλέπουν ως συνολικό αριθμό πυρήνων του συστήματος το πλήθος των διαφορετικών νημάτων που μπορούν να λειτουργούν παράλληλα. Για αυτό το λόγο έχουν προκύψει οι όροι των **φυσικών πυρήνων** (**physical cores**) και των **λογικών πυρήνων** (**logical cores**). Τα υπολογιστικά συστήματα περιλαμβάνουν επίσης μία ιεραρχία μνημών, που τυπικά αποτελείται από την κύρια **μνήμη τυχαίας προσπέλασης** (**random access memory**, συντομογραφικά **RAM**) και μία σειρά **κρυφών μνημών** (**cache**s). Η καθυστέρηση πρόσβασης σε κάθε επίπεδο μνήμης μπορεί να είναι ίδιο για όλους τους επεξεργαστές και πυρήνες ή μη (**NUMA** αρχιτεκτονικές). Τα λειτουργικά συστήματα διαθέτουν **χρονοδρομολογητές** (**task schedulers**), που με διάφορους αλγορίθμους και κριτήρια αποφασίζουν ποια νήματα/διεργασίες εκτελούνται σε κάθε χρονική στιγμή και σε ποιους πυρήνες. Για διάφορους λόγους, είναι δυνατό ο χρονοδρομολογητής να αλλάζει τον πυρήνα εκτέλεσης ενός νήματος. Η διαδικασία αυτή ονομάζεται «**context switch**» και ενέχει μη αμελητέο κόστος σε χρόνο και σε λειτουργίες μνήμης.

Η παραλληλοποίηση του αλγορίθμου B&B προκύπτει μοιράζοντας με οποιοδήποτε τρόπο τα υποσύνολα λύσεων του δένδρου σε διαφορετικά νήματα/διεργασίες. Για τα νήματα μπορεί να χρησιμοποιηθεί ο περισσότερο γενικός όρος «**εργάτης**» (**worker**), ενώ η επεξεργασία συγκεκριμένου υποσυνόλου λύσεων ονομάζεται «**εργασία**» (**task**). Ένα σημαντικό χαρακτηριστικό της παραλληλοποίησης του αλγορίθμου B&B είναι πως δεν είναι γνωστό εξ' αρχής το πλήθος των εργασιών. Μάλιστα, οι (Li & Wah, 1984) έδειξαν ότι αυτό μπορεί μεταβάλλεται στην παράλληλη μορφή του αλγορίθμου, είτε θετικά ή αρνητικά.

2.2 Μοντέλα Παραλληλοποίησης Αλγορίθμου B&B

Στη βιβλιογραφία έχει προκύψει σημαντικός αριθμός διαφορετικών μοντέλων παραλληλοποίησης του αλγορίθμου B&B. Συνοπτικά, αναφέρονται τα εξής (Gendron & Crainic, 1994):

Παράλληλο πολύ-παραμετρικό μοντέλο

Το μοντέλο εκτελεί ταυτόχρονα πολλαπλά στιγμιότυπα του σειριακού αλγορίθμου B&B με διαφορετικές παραμέτρους, έτσι ώστε κάθε ένα να επεξεργάζεται διαφορετικά υποσύνολα

λύσεων του προβλήματος. Οι τομές των υποσυνόλων αυτών, όμως, δεν είναι γενικά κενές, με αποτέλεσμα να σπαταλούνται πόροι για την επεξεργασία ορισμένων λύσεων πολλαπλές φορές.

Μοντέλο παράλληλης επεξεργασίας των παραγόμενων υποσυνόλων λύσεων

Κάθε φορά που ένα υποσύνολο λύσεων διακλαδώνεται σε επιμέρους υποσύνολα, τα όρια αυτών υπολογίζονται ταυτόχρονα. Ο βαθμός της παραλληλοποίησης εξαρτάται από το πλήθος των παραγόμενων υποσυνόλων, ενώ απαιτείται και ακριβός συγχρονισμός.

Μοντέλο παράλληλης αποτίμησης των ορίων ενός υποσυνόλου λύσεων

Οι υπολογισμοί των άνω και κάτω ορίων των υποσυνόλων λύσεων πραγματοποιούνται παράλληλα. Το μοντέλο αποδίδει εφόσον η αποτίμηση αυτή έχει υψηλή χρονική πολυπλοκότητα. Χρησιμοποιείται σε διάφορα προβλήματα μικτού ακέραιου γραμμικού προγραμματισμού. Το βασικό του μειονέκτημα είναι το γεγονός ότι, συνήθως, η πολυπλοκότητα της αποτίμησης των ορίων είναι αντιστρόφως ανάλογη του βάθους του υποσυνόλου λύσεων στο δένδρο του αλγορίθμου B&B.

Μοντέλο παράλληλης διάσχισης του δένδρου του αλγορίθμου B&B

Χρησιμοποιείται κοινή δομή (**pool**) κυρίως για την αποθήκευση των υποσυνόλων λύσεων. Γενικά, οι εργάτες που λειτουργούν παράλληλα εξάγουν υποσύνολα από την κοινή δομή, τα επεξεργάζονται και εισάγουν πίσω σε αυτή τα παραγόμενα νέα υποσύνολα λύσεων. Βασικό μειονέκτημα αυτού του μοντέλου είναι το χρονικό κόστος συγχρονισμού των εργατών με την κοινή δομή, τα κλειδώματα (**locks**) που απαιτούνται κλπ.

Στη συνέχεια θα γίνει μελέτη της παραλληλοποίησης του αλγορίθμου B&B με βάση το τελευταίο μοντέλο, αυτό της παράλληλης διάσχισης του δένδρου.

2.3 Bottlenecks στον Παράλληλο Αλγόριθμο Branch and Bound

Γενικά, στην παραλληλοποίηση οποιοδήποτε αλγόριθμου προκύπτουν μία σειρά από δυσκολίες, που σχετίζονται με τμήματα που πρέπει υποχρεωτικά να εκτελεστούν σειριακά, μειώνοντας έτσι την επιτάχυνση (Amdahl, 1967). Στον παράλληλο αλγόριθμο B&B δεν υπάρχουν μέρη που εκτελούνται εγγενώς σειριακά. Εντούτοις, υπάρχουν υπορουτίνες που μπορούν να εκτελούνται από ένα μόνο νήμα σε κάθε χρονική στιγμή. Συγκεκριμένα, πρέπει να περιμένει κάποιος να προκύψουν bottlenecks κατά το μοίρασμα των επιμέρους εργασιών στους/στα εργάτες/νήματα, δηλαδή στη δομή αποθήκευσης των υποσυνόλων λύσεων. Επιπλέον δυσκολίες μπορεί να παρουσιαστούν κατά τον έλεγχο των ορίων, όπου η ανάγκη για ασφαλή αλλαγή των τιμών τους απαιτεί αποκλειστική πρόσβαση από κάθε νήμα, επομένως δυνητικά σειριακή εκτέλεση. Τέλος, σε υπολογιστικά συστήματα μοιραζόμενης μνήμης οι διαφορετικοί εργάτες μοιράζονται τους πόρους του υποσυστήματος μνήμης και συνεπώς μπορούν να προκύψουν και εκεί εξαρτήσεις και καθυστερήσεις.

2.4 Υλοποίηση Δομής Αποθήκευσης Υποσυνόλων Λύσεων

Η υλοποίηση της δομής αποθήκευσης των υποσυνόλων λύσεων είναι μεγάλης σημασίας, καθώς καθορίζει σε σημαντικό βαθμό τον τρόπο που οι εργασίες μοιράζονται στους εργάτες (**load balancing**). Γενικά, υπάρχουν οι εξής επιλογές:

Χρήση κοινής δομής για όλους τους εργάτες

Χρησιμοποιείται κοινή δομή για όλους τους εργάτες. Το πλεονέκτημα είναι πως γίνεται αυτόματα ισοσκελές μοίρασμα των υποσυνόλων λύσεων. Δυνητικά, όμως, δημιουργούνται ουρές αναμονής από τους εργάτες που θέλουν να εκτελέσουν ταυτόχρονα πράξεις εισαγωγών και εξαγωγών προς και από τη δομή. Για την ελάττωση αυτών των φαινομένων έχουν προταθεί πολλές διαφορετικές υλοποιήσεις, κυρίως κάποιας ουράς με δυνατότητα συγχρονισμού είτε με κλειδώματα (**blocking**) ή χωρίς (**non-blocking**). Στη βιβλιογραφία μπορούν βρεθούν και προσεγγιστικές μεθόδους, με π.χ. σύνολο ουρών και τυχαία επιλογή της ουράς που θα εκτελέσει την πράξη εισαγωγής ή εξαγωγής των (Rihani, Sanders, & Dementiev, 2014).

Χρήση ιδιωτικής δομής για κάθε εργάτη

Χρησιμοποιείται πλήθος δομών, τόσες όσοι και οι εργάτες στον αριθμό, ενώ κάθε δομή χρησιμοποιείται αποκλειστικά από ένα μόνο εργάτη. Εξαλείφουν τις ουρές αναμονής που προκύπτουν στις κοινές δομές, όμως προκαλούν συχνά άνιση κατανομή των εργασιών (μη αποδοτικό load balancing).

Χρήση ιδιωτικών δομών με δυνατότητα «κλοπής» εργασιών

Η υλοποίηση είναι ίδια με πριν, με τη διαφορά ότι παρέχεται η δυνατότητα στους εργάτες με κάποιον τρόπο να «κλέψουν» εργασίες από τη δομή κάποιου άλλου εργάτη. Η δυνατότητα αυτή μπορεί να πάρει πολλές μορφές και έχουν προταθεί διάφορα σχήματα, π.χ. (Leroy, Mezmaz, Melab, & Tuyttens, 2014).

Σε αυτή τη μελέτη χρησιμοποιήθηκαν και οι τρεις παραπάνω υλοποιήσεις, ενώ ως βασική δομή χρησιμοποιήθηκε μία παράλληλη υλοποίηση ουράς προτεραιότητας από τη βιβλιοθήκη «**Threading Building Blocks**» της Intel (συντομογραφικά **Intel TBB** ή απλά **TBB**) (Wilmarth, 2010). Ως προς τη δυνατότητα «κλοπής», οι εργάτες προγραμματίστηκαν έτσι ώστε να μπορούν να λάβουν εργασίες από οποιαδήποτε άλλη ουρά προτεραιότητας, με την προϋπόθεση η δικιά τους να είναι άδεια.

2.5 Όρια στον Παράλληλο Αλγόριθμο Branch and Bound

Η παραλληλοποίηση του αλγορίθμου απαιτεί αναγκαστικά το συγχρονισμό της πρόσβασης των διαφορετικών εργατών στα άνω και κάτω όρια. Αυτά πρέπει να ελέγχονται πάντα μαζί και όχι ανεξάρτητα, συνεπώς απαιτείται η χρήση «**mutex**» για την επίτευξη αποκλειστικής πρόσβασης των ορίων από ένα μόνο εργάτη σε οποιαδήποτε χρονική στιγμή. Οι mutexes τυπικά αναστέλλουν τη λειτουργία ενός εργάτη/νήματος μέχρι να γίνει δυνατή η αποκλειστική πρόσβαση στη δομή που φυλάσσουν. Η αναστολή λειτουργίας επιτρέπει στο λειτουργικό σύστημα να εκτελέσει ένα άλλο διαφορετικό νήμα στη θέση του αρχικού (context switch) και έτσι να αυξήσει την αποδοτικότητα χρήσης των υπολογιστικών πόρων του συστήματος. Αν όμως ο χρόνος που πρέπει να ανασταλεί η λειτουργία του νήματος είναι αρκετά μικρός, είναι πιθανό να υπάρξει τελικά ελάττωση της απόδοσης λόγω του κόστους του context switch. Έτσι, στη συγκεκριμένη περίπτωση, επειδή ο χρόνος πρόσβασης των ορίων αναμένεται να είναι πολύ μικρός κάθε φορά, χρησιμοποιήθηκε «**spin mutex**», που αναπαραγάγει τη λειτουργία ενός απλού mutex, χωρίς όμως να αναστέλλει τη λειτουργία των νημάτων. Επιπλέον, δοκιμάστηκε και μία εναλλακτική υλοποίηση με χρήση ιδιωτικών ορίων ανά εργάτη. Σε αυτή την περίπτωση, κάθε

εργάτης καταλήγει σε μία διαφορετική τελική λύση, εφόσον μπορεί να βρεθεί μία, ενώ η βέλτιστη προκύπτει από τη σύγκριση αυτών. Σε αυτό το πλαίσιο, χρησιμοποιήθηκε και μία μικρή παραλλαγή, όπου τα ιδιωτικά όρια συγχρονίζονται μεταξύ τους, μέσω κάποιον κοινών μεταβλητών, σε προκαθορισμένα χρονικά διαστήματα.

2.6 Bottlenecks Μνήμης και Ανάρτηση Νημάτων

Σε υπολογιστικά συστήματα μοιραζόμενης μνήμης, η παράλληλη εκτέλεση πολλαπλών νημάτων και διεργασιών είναι πιθανό να προκαλέσει υπερφόρτωση του υποσυστήματος μνήμης. Ενδεικτικά αναφέρονται προβλήματα κορεσμού του εύρους ζώνης της μνήμης, ανταγωνισμός για την περιορισμένη χωρητικότητα των κρυφών μνημών κλπ., η ανάλυση των οποίων ξεπερνά τους σκοπούς αυτής της μελέτης. Ένα όμως πρόβλημα, που δυνητικά προκύπτει στα παράλληλα προγράμματα που χωρίζονται σε πολλές επιμέρους εργασίες και είναι άξιο προσοχής, είναι ο τρόπος μοιράσματος των νημάτων/διεργασιών στους επεξεργαστές και πυρήνες, εξαιτίας του κόστους του context switch. Μία τεχνική που συχνά εφαρμόζεται για την ελαχιστοποίηση αυτού το κόστους είναι η **ανάρτηση νημάτων** (**thread pinning**) ή η συγγένεια αυτών (**thread affinity**) σε/με συγκεκριμένους πυρήνες. Επιπλέον, η τεχνολογία SMT καθιστά το πρόβλημα ακόμα πιο περίπλοκο, καθώς ο χρονοδρομολογητής αναθέτει τα νήματα σε λογικούς πυρήνες που μπορεί να βρίσκονται στον ίδιο φυσικό πυρήνα ή σε διαφορετικούς. Στο επόμενο κεφάλαιο παρουσιάζονται μετρήσεις όπου πραγματοποιήθηκε ανάρτηση νημάτων τόσο με προτίμηση συγκέντρωσης των νημάτων σε λίγους φυσικούς πυρήνες όσο και με αποφυγή χρήσης της τεχνολογίας SMT όταν αυτό είναι δυνατό.

2.7 Παραλληλοποίηση Εργατών

Η παραλληλοποίηση των εργατών μπορεί να γίνει με τη χρήση διαφόρων βοηθητικών βιβλιοθηκών, όπως **OpenMP**, **MPI**, **Intel TBB**, **Intel Cilk** και άλλες, ανάλογα τη γλώσσα προγραμματισμού και την τοπολογία της μνήμης των παράλληλων υπολογιστικών συστημάτων (μοιραζόμενη/κατανεμημένη). Στην υλοποίηση που περιγράφεται εδώ, έγινε μελέτη της παραλληλοποίησης του αλγορίθμου B&B σε συστήματα μοιραζόμενης μνήμης μόνο, ενώ, καθώς χρησιμοποιήθηκε η γλώσσα C++, έγινε χρήση της **std::thread** από την καθιερωμένη βιβλιοθήκη (πρότυπο 2011 και έπειτα). Αλγοριθμικά, ο παράλληλος κώδικας είναι ίδιος με τον σειριακό, δηλαδή ο ψευδοκώδικας που περιγράφηκε στο προηγούμενο κεφάλαιο εκτελείται παράλληλα από πολλαπλά νήματα. Η διαφορά έγκειται στην υλοποίηση της δομής αποθήκευσης των υποσυνόλων λύσεων, καθώς και στη μέθοδο εξαγωγής υποσυνόλων από αυτή. Αναλυτικά:

Χρήση κοινής δομής για όλους τους εργάτες

Αν χρησιμοποιείται κοινή δομή για όλους τους εργάτες, τότε δεν υπάρχει καμία διαφορά, πέραν του ότι αλλάζει ελαφρώς η συνθήκη τερματισμού του προγράμματος. Υπενθυμίζεται ότι S είναι το σύνολο όλων των υποψήφιων λύσεων και Q η κοινή δομή στην οποία αυτά αποθηκεύονται. Αρχικά $Q = \{S\}$. S^B είναι η καλύτερη λύση που έχει βρεθεί μέχρι στιγμής. Αρχικά $S^B = -\infty$. Επιπλέον, ορίζεται και το πλήθος των υποσυνόλων που *πιθανόν* να υπάρχουν στην Q, N. Αρχικά $N = 1$.

1. Αν $N > 0$, εξάγεται ένα υποσύνολο λύσεων S^C, διαφορετικά ο αλγόριθμος τερματίζει.
2. Εφαρμόζεται η πράξη της διακλάδωσης στο S^C και παράγονται τα υποσύνολα λύσεων $S_i^C, i = 1, \dots, n$. Για κάθε ένα από τα νέα υποσύνολα:
    a. Αν το υποσύνολο περιλαμβάνει μία μοναδική λύση, τότε εφόσον είναι εφικτή εφαρμόζεται η πράξη $S^B = \max(S^B, |S_i^C|)$.
    b. Διαφορετικά, υπολογίζονται τα άνω και κάτω όρια του υποσυνόλου, U_i^C και L_i^C. Αν $U_i^C > S^B$ και $U_i^C > L^B$ τότε εφαρμόζονται οι πράξεις $L^B = \max(L^B, L_i^C)$, $Q = Q \cup S_i^C$ και $N = N + 1$. Αλλιώς, το υποσύνολο απορρίπτεται.
3. Εφαρμόζεται η πράξη $N = N - 1$.
4. Επιστροφή στο βήμα.

Χρήση ιδιωτικής δομής για κάθε εργάτη

Σε αυτή την περίπτωση, πρέπει πρώτα να τοποθετηθεί με κάποιο τρόπο ένα υποσύνολο σε κάθε ιδιωτική δομή. Αυτό συνήθως επιτυγχάνεται εκτελώντας το σειριακό αλγόριθμο, μέχρι το πλήθος των υποσυνόλων στη δομή να φτάσει το επιθυμητό μέγεθος. Έπειτα τα υποσύνολα μοιράζονται στις δομές και κάθε εργάτης εκτελεί το σειριακό αλγόριθμο.

Χρήση ιδιωτικών δομών με δυνατότητα «κλοπής» εργασιών

Εφαρμόζεται ο ίδιος αλγόριθμος, όπως στην περίπτωση χρήσης κοινής δομής για όλους τους εργάτες, όπου Q είναι κάθε φορά η ιδιωτική δομή του κάθε εργάτη. Η πράξη της εξαγωγής όμως παύει να είναι τετριμμένη και περιγράφεται παρακάτω. Ως $Q(i)$ ορίζεται η ιδιωτική δομή του ι-οστού εργάτη, W είναι το πλήθος των εργατών:

1. $counter = threadID$.
2. Όσο $Q(counter \ mod \ W)$ άδεια και $N > 0$ εφαρμόζεται η πράξη $counter = counter + 1$.
3. Αν $N = 0$ τέλος εκτέλεσης εργάτη. Διαφορετικά εξάγεται υποσύνολο την $Q(counter \ mod \ W)$.

3 Πειραματικές μετρήσεις

3.1 Αλγόριθμοι – Προβλήματα

Με βάση την υλοποίηση που περιγράφηκε στο προηγούμενο κεφάλαιο, έγιναν μετρήσεις της απόδοσης του παράλληλου αλγορίθμου B&B. Χρησιμοποιήθηκαν συνολικά τρεις διαφορετικοί αλγόριθμοι, ένας για το διακριτό πρόβλημα του σακιδίου και δύο για το πρόβλημα του πλανόδιου πωλητή. Ο αλγόριθμος για το διακριτό πρόβλημα του σακιδίου είναι αυτός που περιγράφηκε στο πρώτο κεφάλαιο. Οι υλοποιήσεις για το πρόβλημα του πλανόδιου πωλητή περιγράφονται παρακάτω.

3.1.1 Διακριτό πρόβλημα του σακιδίου: Δεδομένα προβλήματος

Από προηγούμενες μελέτες (Pisinger, 1995), έχει αποδειχθεί πως η δυσκολία του προβλήματος εξαρτάται από τη σχέση αξίας ανά βάρος των αντικειμένων. Μεγαλύτερη πολυπλοκότητα εμφανίζεται σε αυτά που η σχέση είναι γραμμική. Για τις μετρήσεις χρησιμοποιήθηκε η απλή γραμμική σχέση, όπου η αξία είναι ίση με το βάρος, με τις τιμές να δίνονται από τις συναρτήσεις $f(x) = x$ και $f(j) = 2^{\lfloor \log_2 j \rfloor + j + 1} + 2^{\lfloor \log_2 j \rfloor + j} + 1, j = 1, 2, \dots, n$.

3.1.2 Πρόβλημα πλανόδιου πωλητή: Γενικά

Ένας πλανόδιος πωλητής πρέπει να επισκεφτεί έναν αριθμό από πόλεις και αναζητά τη βέλτιστη διαδρομή που περνά από κάθε μία ακριβώς μία φορά, επιστρέφει στο σημείο εκκίνησης και έχει το μικρότερο δυνατό μήκος. Οι πόλεις και οι πιθανές διαδρομές του πωλητή περιγράφονται από ένα γράφο $G = (V, E)$, όπου V είναι το σύνολο των κορυφών/πόλεων με πλήθος N και E είναι το σύνολο των ακμών/δρόμων μεταξύ των πόλεων. Κάθε ακμή συνοδεύεται από θετικό βάρος που αντιπροσωπεύει το μήκος του αντίστοιχου δρόμου. Ο γράφος μπορεί να είναι κατευθυνόμενος (ασύμμετρο πρόβλημα) ή μη (συμμετρικό πρόβλημα). Αναζητείται ο απλός κύκλος βαθμού N του γράφου με το ελάχιστο δυνατό μήκος. Το πρόβλημα βελτιστοποίησης δίνεται ως εξής:

$$min \sum_{i=0}^{n} \sum_{j \neq i, j=0}^{n} c_{ij} x_{ij}$$

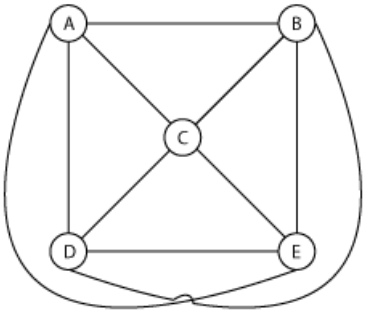
$$x_{ij} \in \{0, 1\}$$

$$\sum_{i \neq j, i=0}^{n} x_{ij} = 1$$

$$\sum_{j \neq i, j=0}^{n} x_{ij} = 1$$

$$u_i - u_j + n x_{ij} \leq n - 1, u_i \in \mathbb{Z}$$

$$i = 0, 1, \dots, n \; \kappa\alpha\iota \; j = 0, 1, \dots, n$$

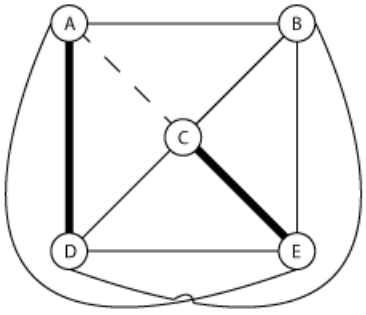


	A	B	C	D	E	
A	∞	633	257	91	412	
B	633	∞	390	661	227	
C	257	390	∞	228	189	
D	91	661	228	∞	363	
E	412	227	189	363	∞	
	91	227	189	91	189	
	257	390	228	228	227	
	348	617	417	319	416	2117

Ακμή του γράφου
Ακμή που περιλαμβάνεται υποχρεωτικά στη διαδρομή
Ακμή που αποκλείεται από τη διαδρομή

Εικόνα 3-1: Παράδειγμα συμμετρικού TSP και υπολογισμός ορίου στη ρίζα του δένδρου B&B

Η μεταβλητή x_{ij} είναι δυαδική και λαμβάνει την τιμή 1, αν η ακμή με αφετηρία την κορυφή i και προορισμό την κορυφή j είναι στη διαδρομή του πωλητή. Οι σταθερές c_{ij} είναι τα βάρη των αντίστοιχων ακμών, ενώ η u_i είναι βοηθητική μεταβλητή. Κάθε κορυφή στο ζητούμενο κύκλο γειτονεύει με δύο ακριβώς ακμές. Έτσι, ένας εύκολος τρόπος υπολογισμού ενός άνω ορίου (βλέπε §1.2 και §1.3) του μήκους της βέλτιστης διαδρομής προκύπτει από την εύρεση των δύο γειτονικών της ακμών κάθε κορυφής με το μικρότερο μήκος. Αποδεικνύεται ότι το άθροισμα των μηκών των $2N$ αυτών ακμών διαιρούμενο δια δύο είναι άνω όριο της βέλτιστης λύσης του συμμετρικού προβλήματος. Αν το πρόβλημα είναι ασύμμετρο, τότε μπορεί να βρεθεί για κάθε κορυφή η γειτονική ακμή με το μικρότερο μήκος με συγκεκριμένη κατεύθυνση (εισερχόμενη ή εξερχόμενη προς/από την κορυφή), ενώ το όριο είναι ίσο με το άθροισμα των μηκών των N ακμών. Στην εικόνα 3-1 δίνεται ένα παράδειγμα ασύμμετρου προβλήματος, καθώς και ο τρόπος υπολογισμού του άνω ορίου με τον παραπάνω αλγόριθμο. Στην εικόνα 3-2 περιγράφεται ο υπολογισμός του ορίου για μία μερική λύση του προβλήματος.



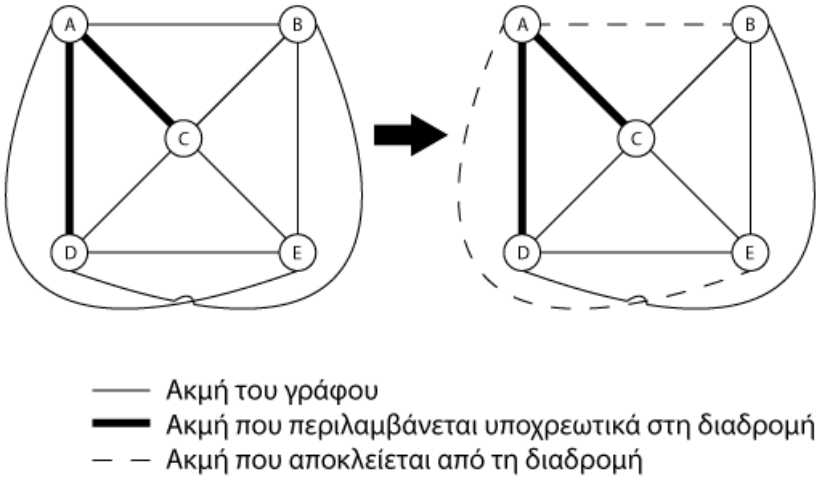
	A	B	C	D	E	
A	∞	633	~~257~~	**91**	412	
B	633	∞	390	661	227	
C	~~257~~	390	∞	228	**189**	
D	**91**	661	228	∞	363	
E	412	227	**189**	363	∞	
	91	227	189	91	189	
	412	390	228	228	227	
	503	617	417	319	416	2272

Ακμή του γράφου
Ακμή που περιλαμβάνεται υποχρεωτικά στη διαδρομή
Ακμή που αποκλείεται από τη διαδρομή

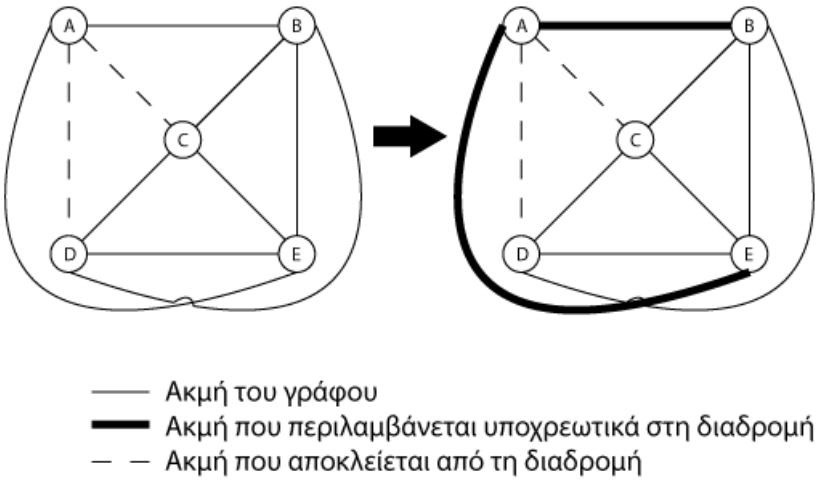
Εικόνα 3-2:Παράδειγμα συμμετρικού TSP και υπολογισμός ορίου για εσωτερικό κόμβο του δένδρου B&B

3.1.3 Πρόβλημα πλανόδιου πωλητή: Υλοποίηση με δυαδικό δένδρο

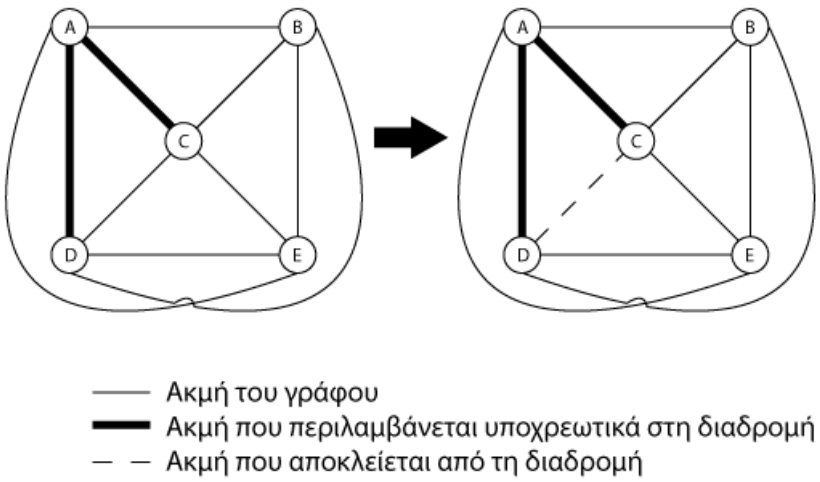
Σε αυτή την υλοποίηση, η ανάπτυξη του δένδρου του αλγορίθμου B&B γίνεται ως προς τις ακμές του γράφου του προβλήματος. Δηλαδή, σε κάθε κόμβο του δένδρου επιλέγεται μία ακμή και με βάση αυτή παράγονται δύο απόγονοι, ένας όπου η ακμή περιλαμβάνεται στη διαδρομή του πωλητή υποχρεωτικά και ένας όπου η συγκεκριμένη ακμή αποκλείεται. Σε κάθε διακλάδωση είναι απαραίτητο να γίνεται έλεγχος των περιορισμών, καθώς η συμπερίληψη ή ο αποκλεισμός μίας ακμής μπορεί να συνεπάγεται υποχρεωτική συμπερίληψη ή αποκλεισμό άλλων ακμών για την τήρηση των περιορισμών. Σχετικά παραδείγματα, όπως η αποφυγή σχηματισμού κύκλων και η γειτνίαση κάθε κορυφής με ακριβώς δύο ακμές, δίνονται στις εικόνες 3-3, 3-4, και 3-5.



Εικόνα 3-3: Υποχρεωτικός αποκλεισμός ακμών για την τήρηση του περιορισμού των δύο γειτονικών ακμών ανά κορυφή



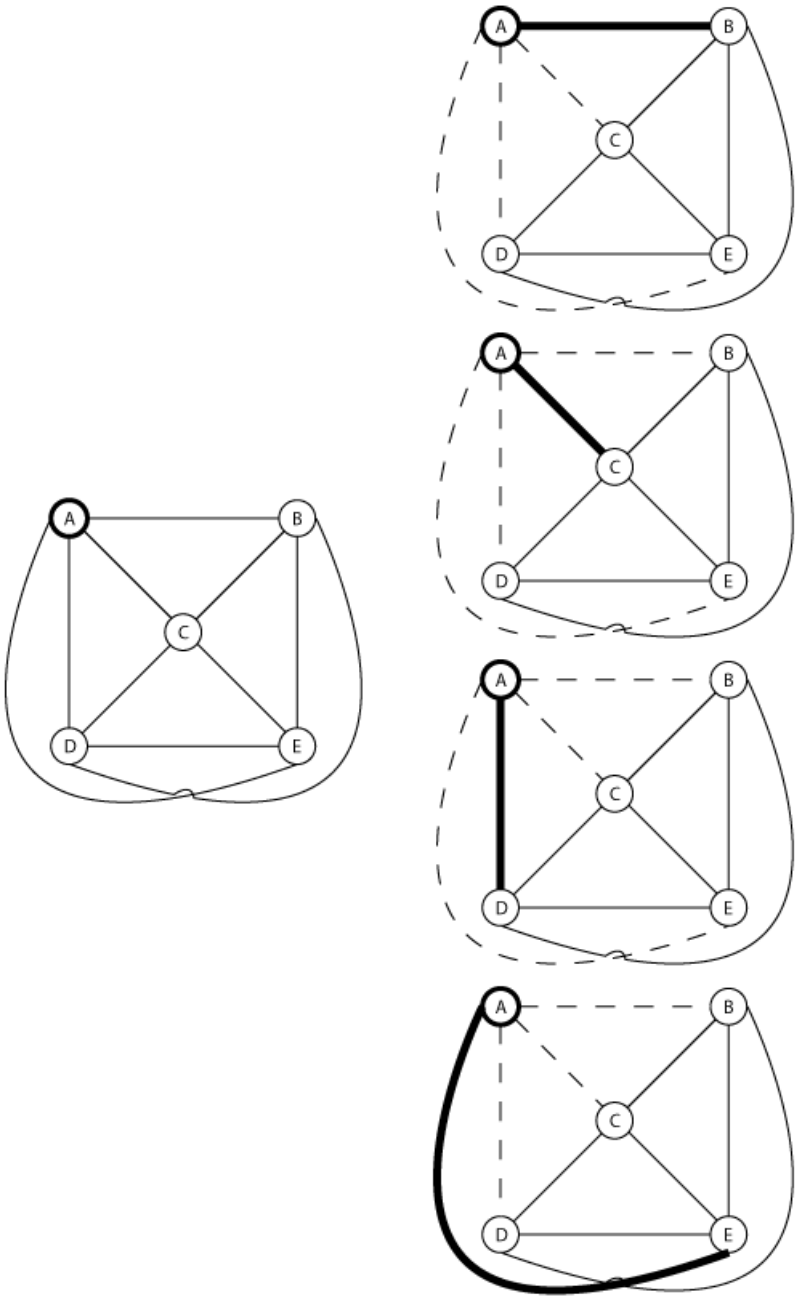
Εικόνα 3-4: Υποχρεωτική συμπερίληψη ακμών για την τήρηση του περιορισμού των δύο γειτονικών ακμών ανά κορυφή



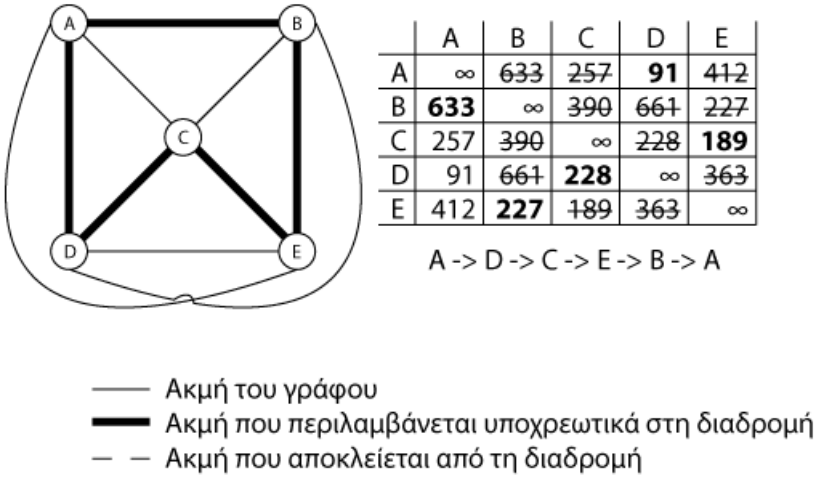
Εικόνα 3-5: Υποχρεωτικός αποκλεισμός ακμών για την αποφυγή δημιουργίας κύκλου

3.1.4 Πρόβλημα πλανόδιου πωλητή: Υλοποίηση με δένδρο βαθμού μεγαλύτερου του δύο

Εδώ, η ανάπτυξη του δένδρου του αλγορίθμου B&B γίνεται ως προς τις κορυφές του γράφου του προβλήματος. Στη ρίζα του δένδρου επιλέγεται μία συγκεκριμένη κορυφή ως η αρχή της διαδρομής του πωλητή. Παράγονται από τη ρίζα τόσοι απόγονοι, όσες και οι κορυφές με τις οποίες γειτνιάζει (μέσω μίας ακμής) η επιλεγμένη κορυφή. Σε κάθε έναν από τους απογόνους θεωρείται πως ο πωλητής έχει μετακινηθεί στην αντίστοιχη κορυφή (μέσω της αντίστοιχης ακμής) και ούτω καθ' εξής. Ο τρόπος αυτός ανάπτυξης του δένδρου έχει ως αποτέλεσμα η μερική διαδρομή που σχηματίζεται σε κάθε στιγμή να είναι απλό μονοπάτι. Επομένως, μπορεί σχετικά εύκολα να χρησιμοποιηθεί ο αλγόριθμος του «**Κοντινότερου Γείτονα**» (**Nearest Neighbor**, συντομογραφικά **NN**) προκειμένου να βρεθεί μία εφικτή λύση του προβλήματος ως κάτω όριο (εικόνα 3-7).



Εικόνα 3-6: Υλοποίηση λύσης TSP με δένδρο βαθμού μεγαλύτερου του δύο και διακλαδώσεις της ρίζας τους δένδρου



	A	B	C	D	E
A	∞	~~633~~	~~257~~	**91**	~~412~~
B	**633**	∞	~~390~~	~~661~~	~~227~~
C	257	~~390~~	∞	~~228~~	**189**
D	91	~~661~~	**228**	∞	~~363~~
E	412	**227**	~~189~~	~~363~~	∞

A -> D -> C -> E -> B -> A

Εικόνα 3-7: Υπολογισμός κάτω ορίου συμμετρικού TSP με τον αλγόριθμο NN

3.1.5 Πρόβλημα του πλανόδιου πωλητή: Δεδομένα προβλήματος

Για τις μετρήσεις χρησιμοποιήθηκε ένα συμμετρικό πρόβλημα με 12 πόλεις, που προέκυψε με αφαίρεση κορυφών και των αντίστοιχων ακμών από το πρόβλημα gr17 της βιβλιοθήκης TSPLIB (Office Research Group Discrete Optimization, Heidelberg University, n.d.).

3.2 Μετρήσεις

Έγιναν μετρήσεις στους παραπάνω αλγορίθμους σε υπολογιστικό σύστημα με δύο 14πύρηνους επεξεργαστές (2 x Intel Xeon E5-2697 v3), με τεχνολογία SMT και δύο λογικούς πυρήνες ανά έναν φυσικό και δυνατότητα ταυτόχρονης εκτέλεσης 56 νημάτων. Χρησιμοποιήθηκαν δύο αλγόριθμοι αναζήτησης δένδρου BestFS, ο ένας με κριτήριο το καλύτερο άνω όριο και ο δεύτερος με πρωτεύον κριτήριο το βάθος του δένδρου και δευτερεύον το άνω όριο. Για αποφυγή σύγχυσης, ο πρώτος αλγόριθμος θα αναφέρεται στη συνέχεια ως BestFS και ο δεύτερος ως (ελαφρώς καταχρηστικά) DFS. Για κάθε περίπτωση έγιναν δοκιμές με πέντε διαφορετικές υλοποιήσεις:

1. Απλή υλοποίηση
2. Υλοποίηση με ιδιωτικά όρια
3. Υλοποίηση με ιδιωτικά όρια και περιοδικό συγχρονισμό αυτών
4. Υλοποίηση νο.3 με ανάρτηση νημάτων και προτίμηση στη χρήση των λογικών πυρήνων
5. Υλοποίηση νο.3 με ανάρτηση νημάτων και αποφυγή χρήσης των λογικών πυρήνων

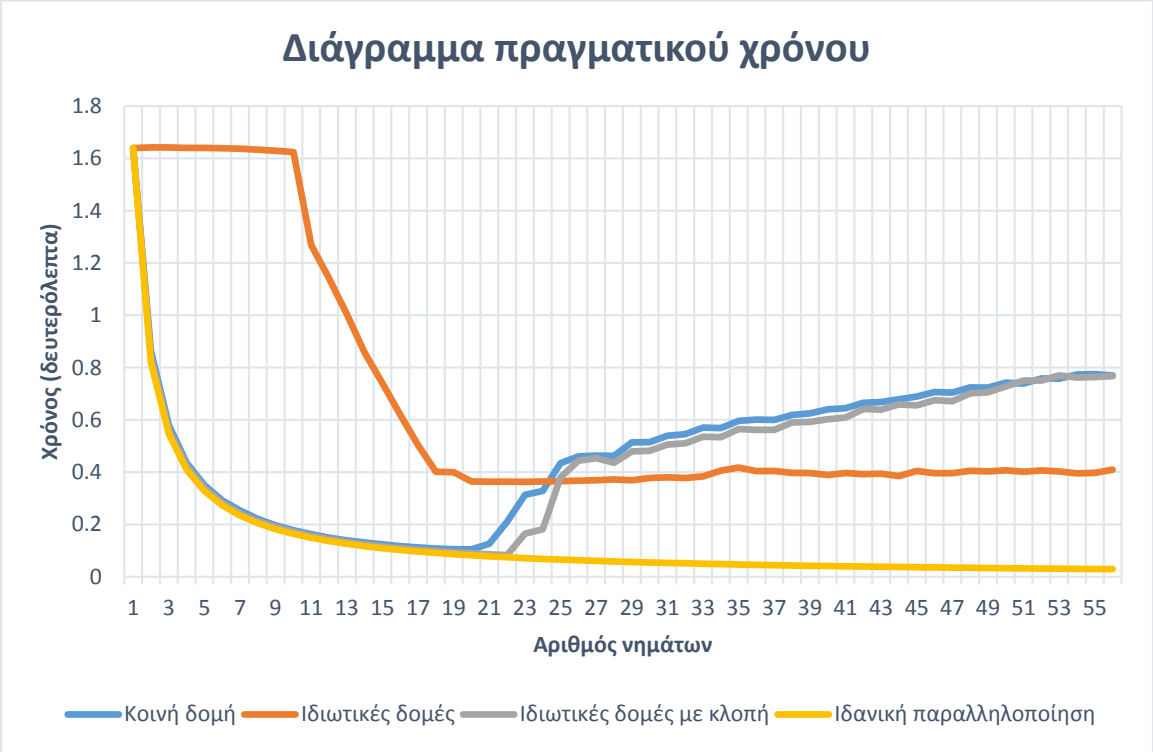
Παρακάτω παρουσιάζονται τα αποτελέσματα των μετρήσεων με παράθεση γραφημάτων του πραγματικού χρόνου εκτέλεσης των αλγορίθμων, της επιτάχυνσης του παράλληλου αλγορίθμου ως προς τον σειριακό, του αριθμού των κόμβων του δένδρου, αλλά και του χρόνου CPU. Στα τελευταία, έχει γίνει μέτρηση του συνολικού χρόνου εκτέλεσης όλων των νημάτων και στη συνέχεια παράσταση των χρόνων των επιμέρους μερών του αλγορίθμου επί τοις εκατό του συνολικού. Συγκεκριμένα, μετρήθηκαν:

1. Ο χρόνος που καταναλώθηκε για τις εισαγωγές και εξαγωγές από τη δομή αποθήκευσης των υποσυνόλων λύσεων
2. Ο χρόνος που καταναλώθηκε σε λειτουργίες μνήμης, κυρίως αντιγραφές
3. Ο χρόνος που απαιτήθηκε για των υπολογισμό των ορίων και τη διακλάδωση των κόμβων
4. Ο χρόνος που δαπανήθηκε για τον κύριο έλεγχο των ορίων
5. Ο χρόνος εκτέλεσης των λοιπών μερών του αλγορίθμου, όπως κάποιοι δευτερεύοντες έλεγχοι των ορίων

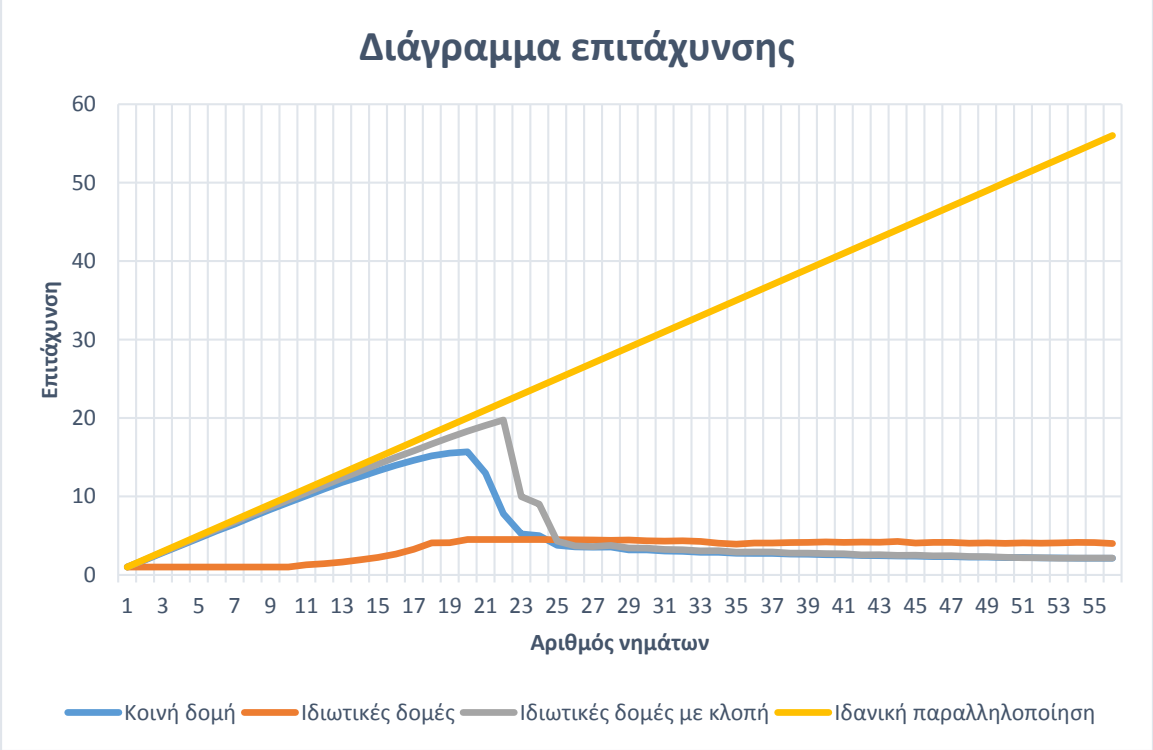
3.2.1 Μετρήσεις διακριτού προβλήματος του σακιδίου

Από τις μετρήσεις, δεν προκύπτει κάποια σημαντική διαφορά ως προς τα δεδομένα του προβλήματος και τον αλγόριθμο αναζήτησης. Δηλαδή, η απόδοση της παραλληλοποίησης φαίνεται να εξαρτάται κυρίως από τον τύπο της δομής αποθήκευσης των υποσυνόλων. Μάλιστα, καλύτερη απόδοση προκύπτει από τη χρήση κοινής δομής και ιδιωτικών δομών με δυνατότητα κλοπής. Αξίζει να σημειωθεί πως στο σειριακό αλγόριθμο (ένα μόνο νήμα) ο χρόνος υπολογισμού των ορίων και των διακλαδώσεων είναι μια τάξη μεγέθους μεγαλύτερος των υπολοίπων (γραφήματα χρόνου CPU 3-3, 3-4 και 3-5).

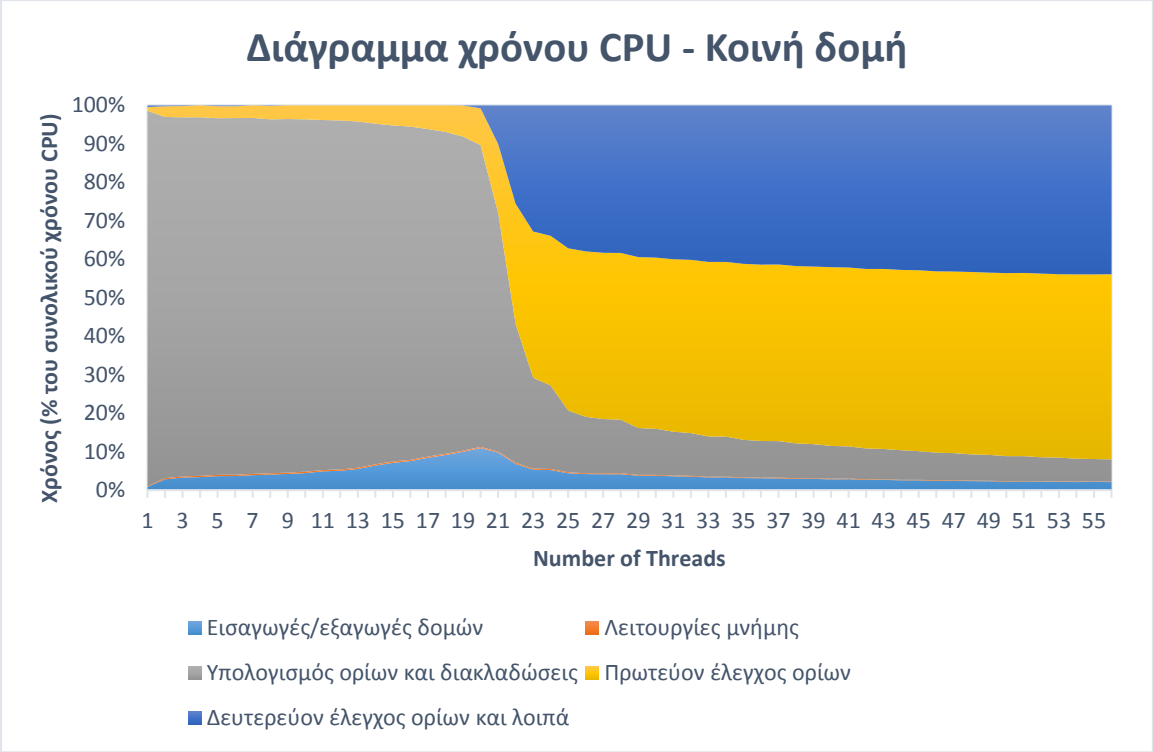
Πιο συγκεκριμένα, στην απλή υλοποίηση, καθώς αυξάνεται ο αριθμός των νημάτων παρατηρούμε απότομη πτώση της επιτάχυνσης (γραφήματα 3-1, 3-2, 3-7, 3-8 και 3-9). Από τα διαγράμματα χρόνου CPU (γραφήματα 3-3, 3-4 και 3-5) προκύπτει πως αυτή οφείλεται σε μεγάλη αύξηση του χρόνου που καταναλώνεται για τον έλεγχο των ορίων. Επειδή ο αριθμός των κόμβων δεν μεταβάλλεται (γράφημα 3-6), συμπεραίνει κανείς πως και το πλήθος των ελέγχων των ορίων παραμένει σταθερό. Συνεπώς, το πρόβλημα εντοπίζεται σε ραγδαία αύξηση του χρονικού κόστους του συγχρονισμού, αφού σε κάθε χρονική στιγμή, μόνο ένα νήμα επιτρέπεται να εκτελεί των κώδικα του ελέγχου των ορίων. Η χρήση απλών ιδιωτικών δομών έχει μη ικανοποιητικά αποτελέσματα καθώς, είτε δεν παράγονται αρκετοί κόμβοι ώστε να εκτελεστεί παράλληλα ο αλγόριθμος, ή, όταν γίνεται εφικτό να τροφοδοτηθούν όλα τα νήματα, γρήγορα αυτά ολοκληρώνουν τη λειτουργία τους και έτσι η τελική παραλληλία που επιτυγχάνεται είναι μικρή.



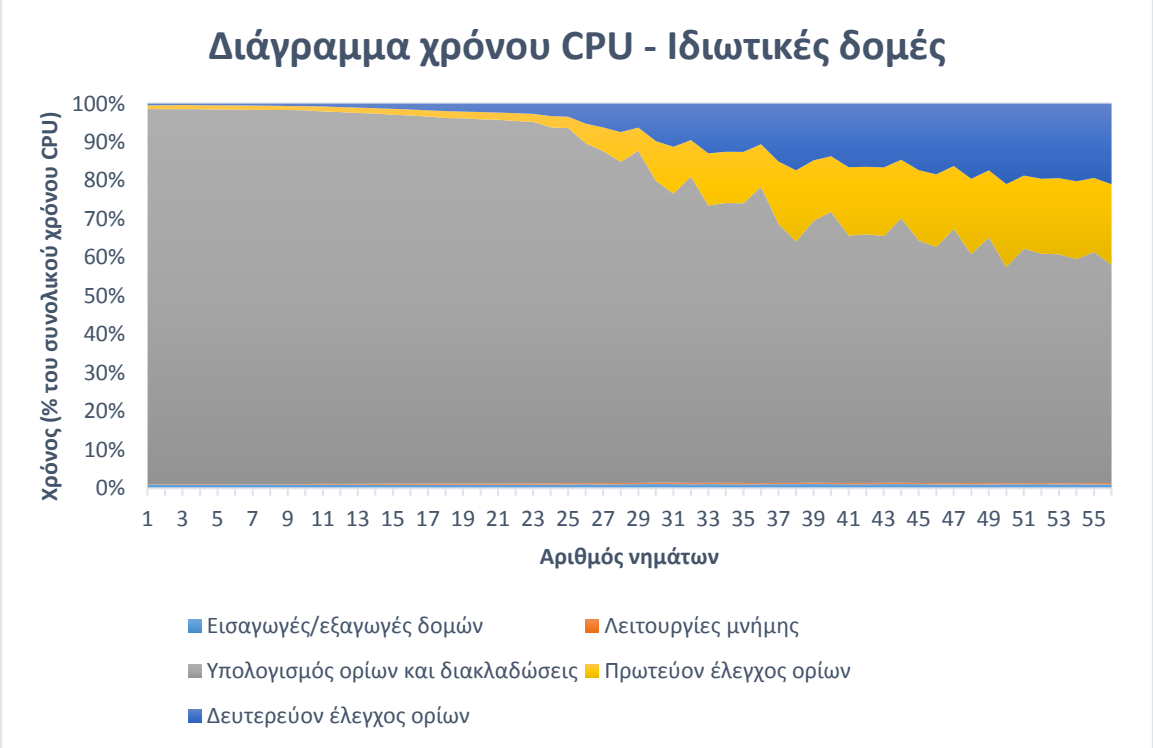
Γράφημα 3-1: DKP, f(x) = x, BestFS, απλή υλοποίηση, πραγματικός χρόνος



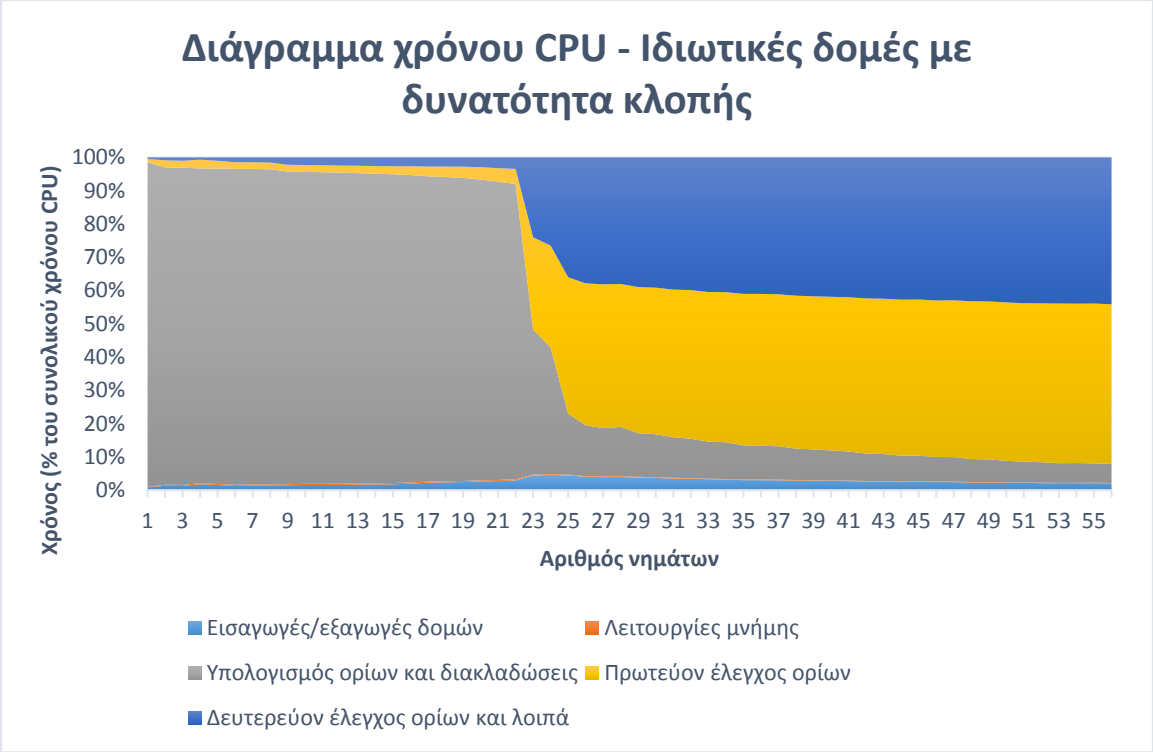
Γράφημα 3-2: DKP, f(x) = x, BestFS, απλή υλοποίηση, επιτάχυνση



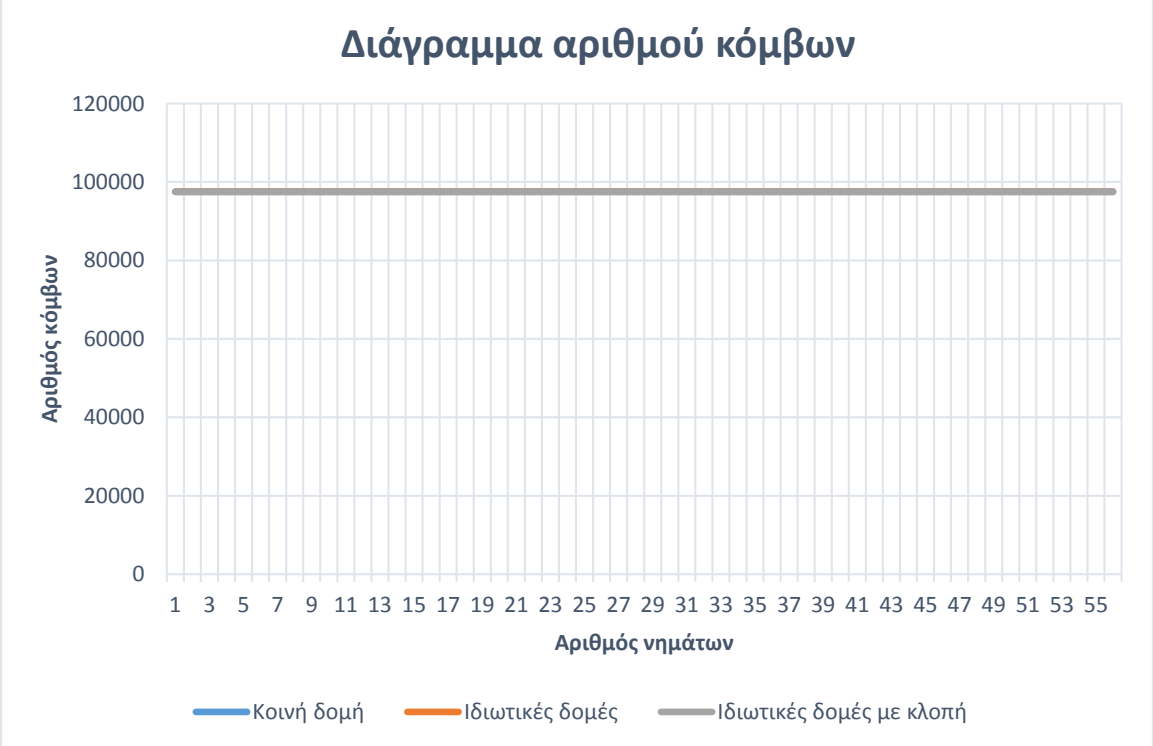
Γράφημα 3-3: DKP, f(x) = x, BestFS, απλή υλοποίηση, χρόνος CPU, κοινή δομή



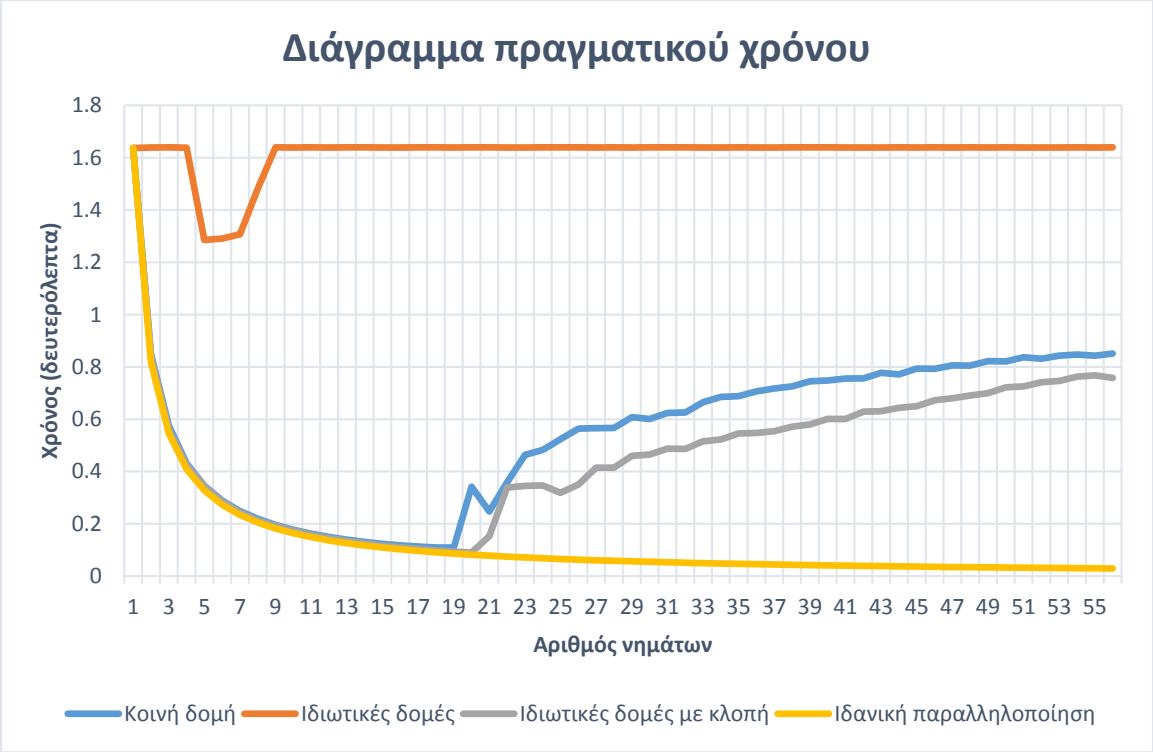
Γράφημα 3-4: DKP, f(x) = x, BestFS, απλή υλοποίηση, χρόνος CPU, ιδιωτικές δομές



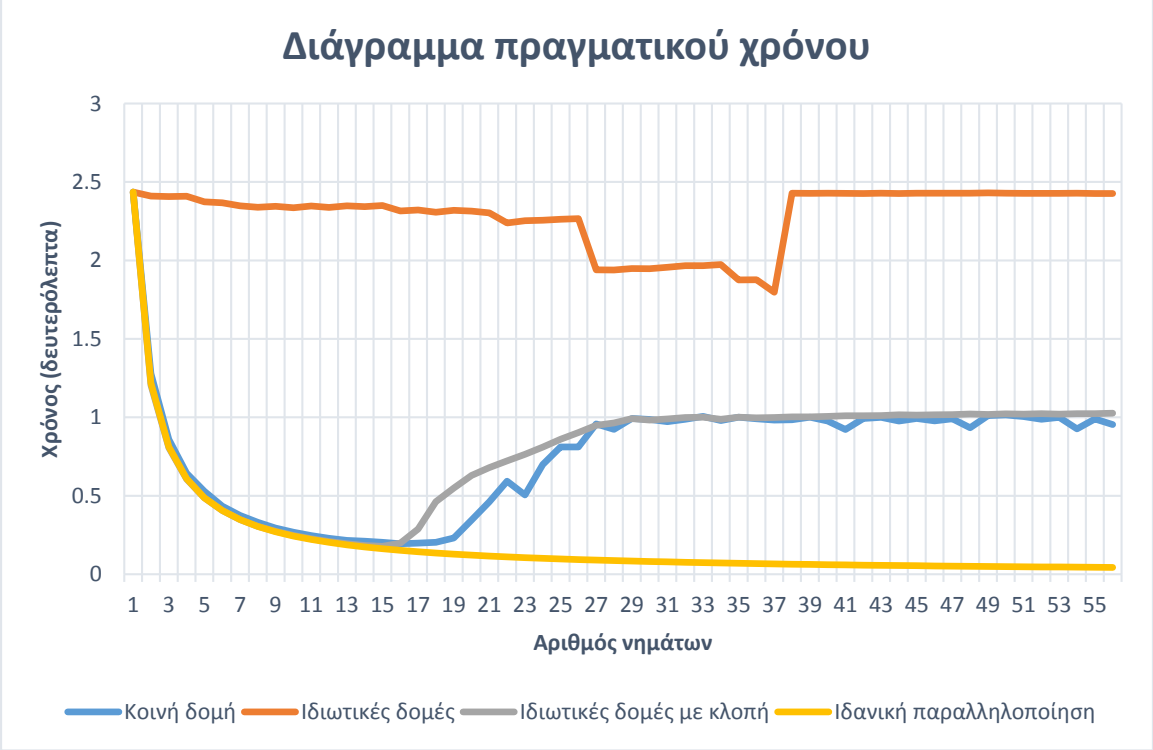
Γράφημα 3-5: DKP, f(x) = x, BestFS, απλή υλοποίηση, χρόνος CPU, ιδιωτικές δομές με δυνατότητα κλοπής



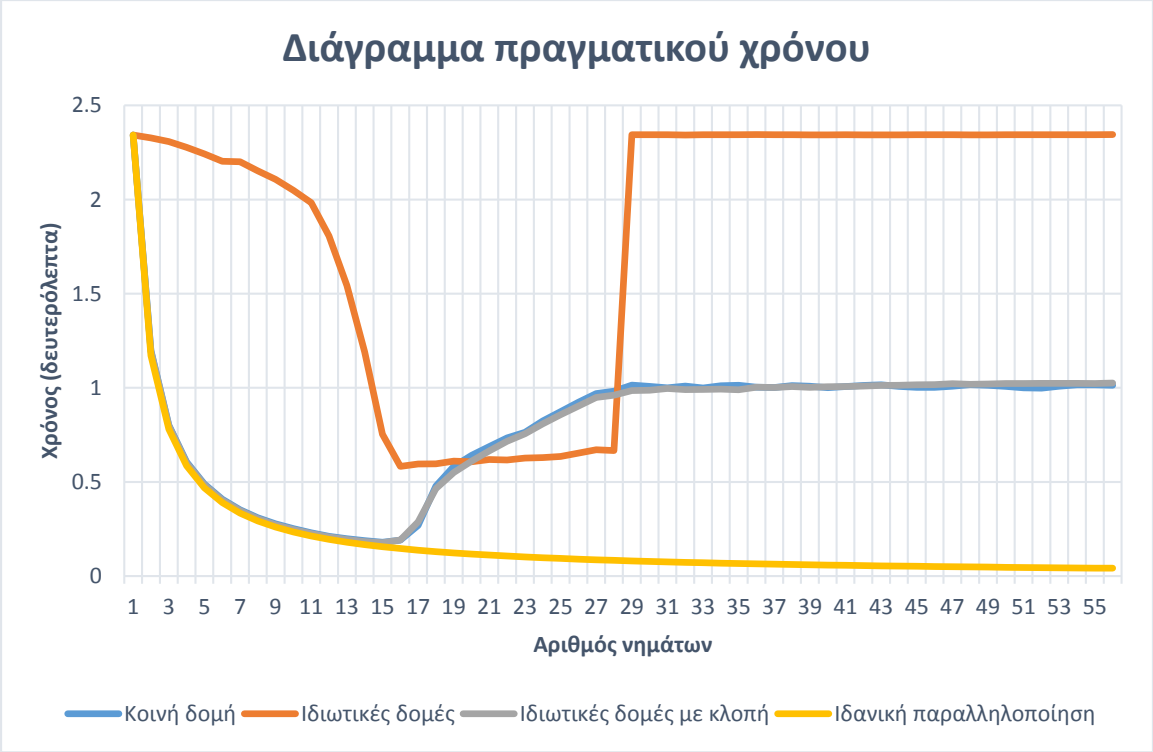
Γράφημα 3-6: DKP, f(x) = x, BestFS, απλή υλοποίηση, αριθμός κόμβων



Γράφημα 3-7: DKP, f(x) = x, DFS, απλή υλοποίηση, πραγματικός χρόνος

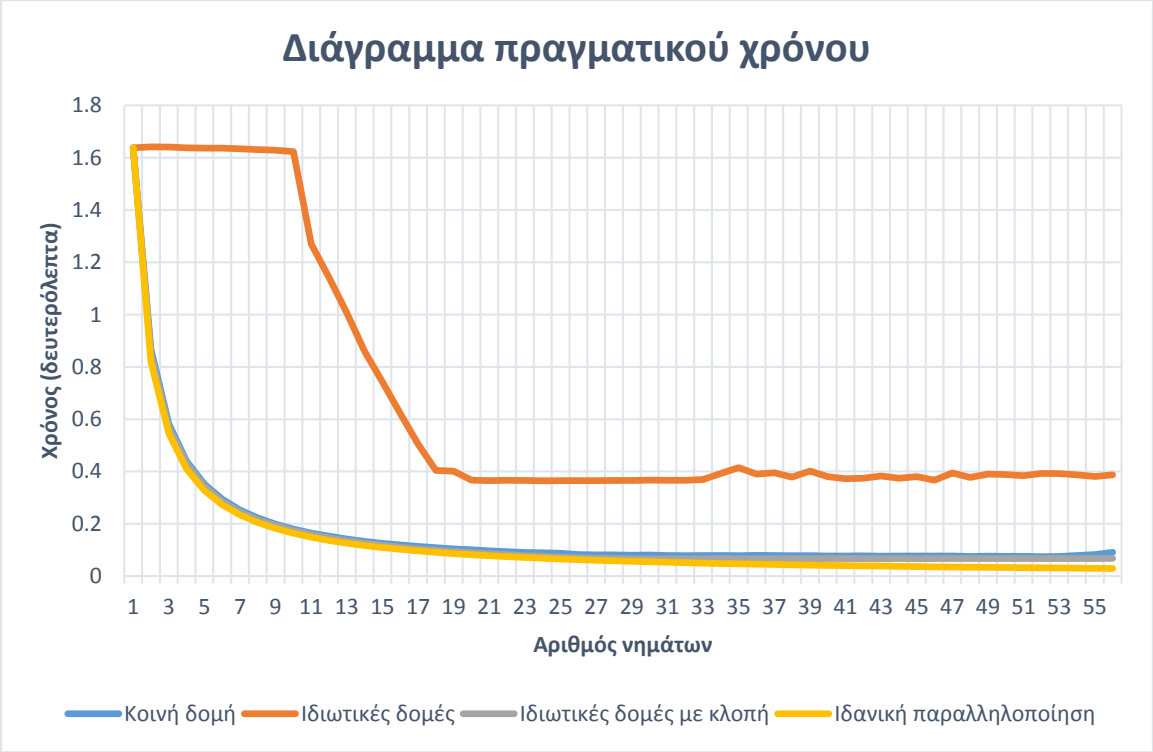


Γράφημα 3-8: DKP, f(x) = exp(x), BestFS, απλή υλοποίηση, πραγματικός χρόνος

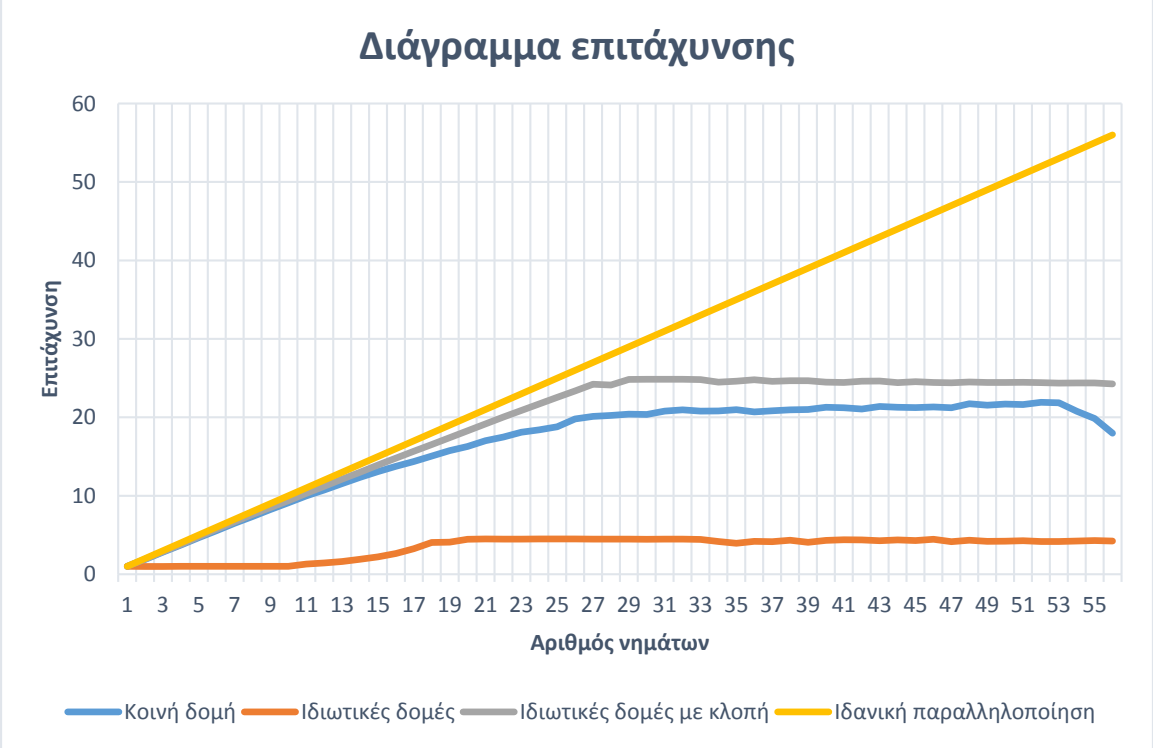


Γράφημα 3-9: DKP, f(x) = exp(x), DFS, απλή υλοποίηση, πραγματικός χρόνος

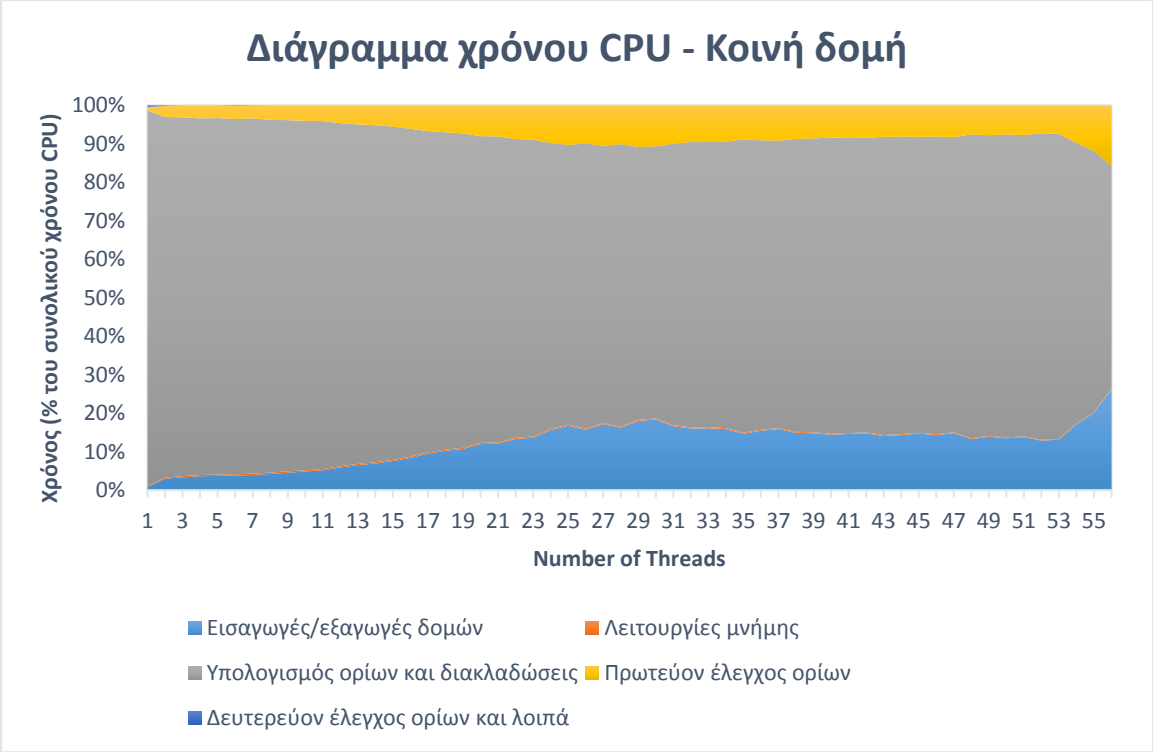
Η χρήση ιδιωτικών ορίων διορθώνει σημαντικά την απόδοση του παράλληλου αλγορίθμου όσον αφορά τη χρήση κοινής δομής και ιδιωτικών δομών με δυνατότητα κλοπής (γραφήματα 3-10 και 3-11), καθώς έλεγχος των ορίων παύει να αποτελεί bottleneck. Οι ιδιωτικές δομές με δυνατότητα κλοπής παρουσιάζουν καλύτερη επιτάχυνση, γιατί στη χρήση κοινής δομής παρατηρείται bottleneck στις εισαγωγές και εξαγωγές κόμβων (γραφήματα 3-12 και 3-13). Επιπλέον, ο περιοδικός συγχρονισμός των ορίων δε φαίνεται να αλλάζει το αποτέλεσμα, ενώ δεν υπάρχει καμία βελτίωση στη χρήση απλών ιδιωτικών δομών.



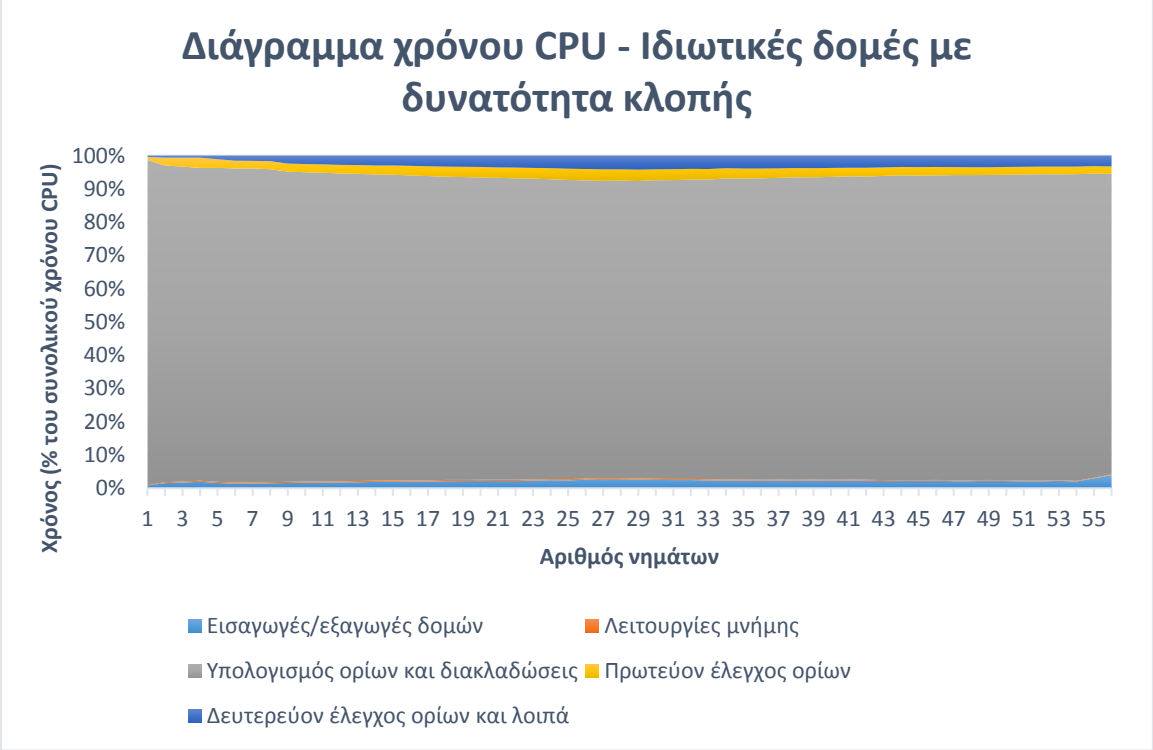
Γράφημα 3-10: DKP, f(x) = x, BestFS, ιδιωτικά όρια, πραγματικός χρόνος



Γράφημα 3-11: DKP, f(x) = x, BestFS, ιδιωτικά όρια, επιτάχυνση

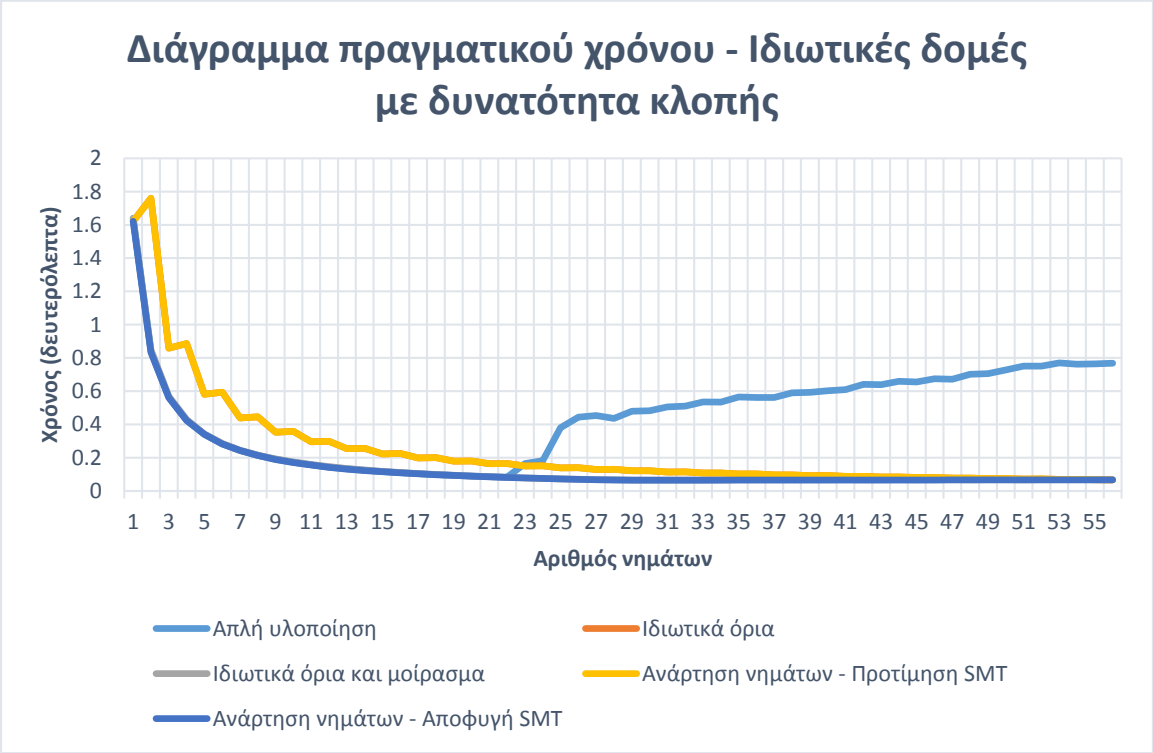


Γράφημα 3-12: DKP, f(x) = x, BestFS, ιδιωτικά όρια, χρόνος CPU, κοινή δομή

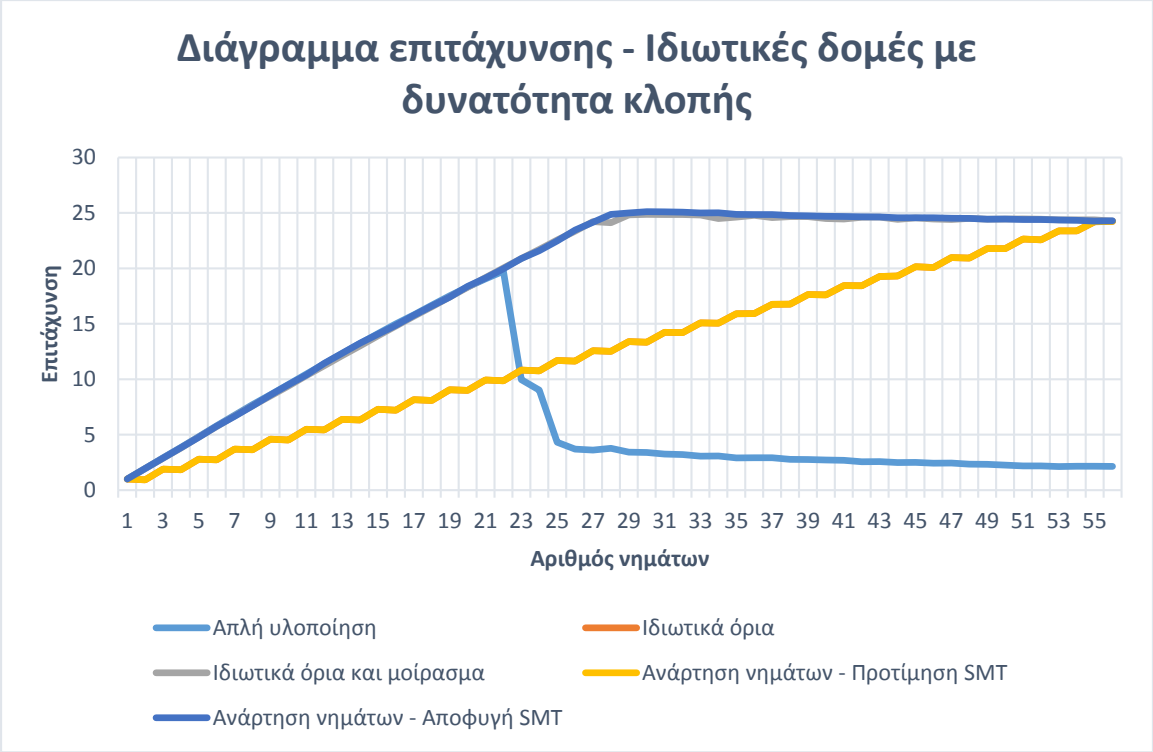


Γράφημα 3-13: DKP, f(x) = x, BestFS, ιδωτικά όρια, χρόνος CPU, ιδιωτικές δομές με δυνατότητα κλοπής

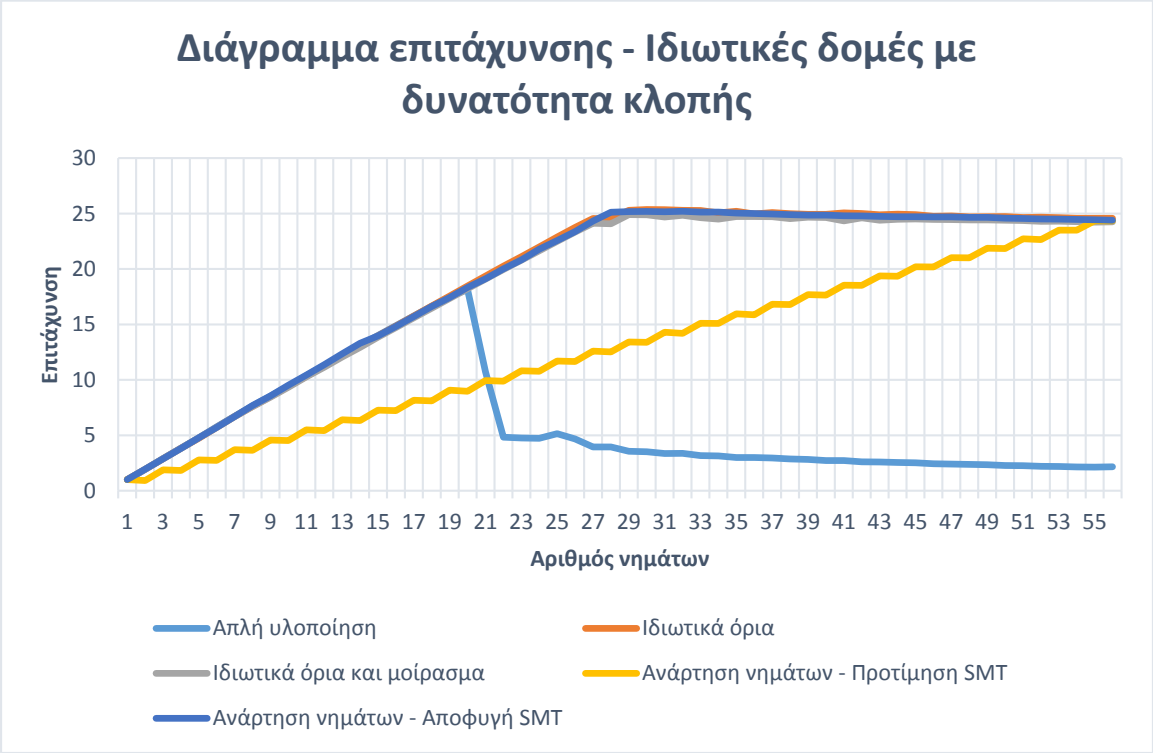
Η ανάρτηση νημάτων δείχνει με μεγάλη διαφορά πως, στο συγκεκριμένο αλγόριθμο, επιτυγχάνεται πολύ καλύτερη απόδοση όταν προτιμάται τα νήματα να εκτελούνται σε διαφορετικούς φυσικούς πυρήνες. Η επιτάχυνση παραμένει σταθερή όταν αρχίζει η χρήση της τεχνολογίας SMT, γεγονός που δείχνει πως, στο συγκεκριμένο αλγόριθμο, η χρήση της δεν αυξάνει την απόδοση. Το αποτέλεσμα που παρατηρείται είναι περίπου ίδιο με αυτό δίχως ανάρτηση νημάτων και επομένως, ο scheduler του λειτουργικού συστήματος φαίνεται πως προτιμά να αξιοποιεί διαφορετικούς φυσικούς πυρήνες όταν αυτό είναι εφικτό.



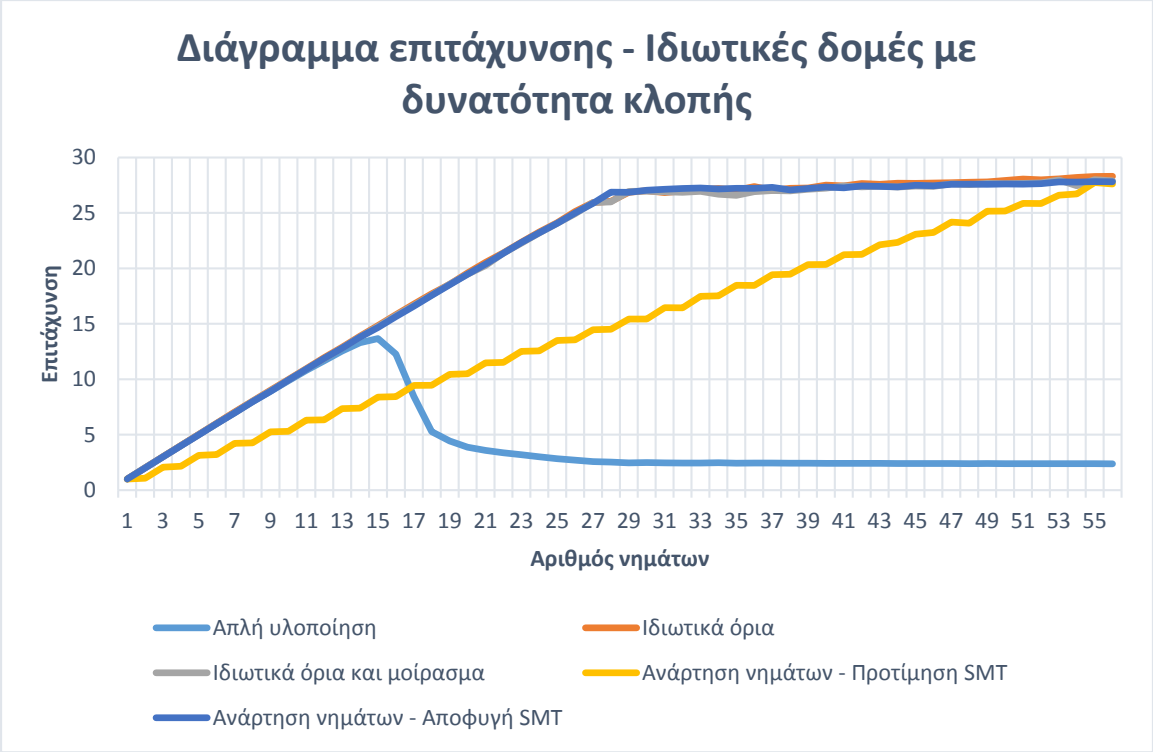
Γράφημα 3-14: DKP, f(x) = x, BestFS, πραγματικός χρόνος, ιδιωτικές δομές με δυνατότητα κλοπής



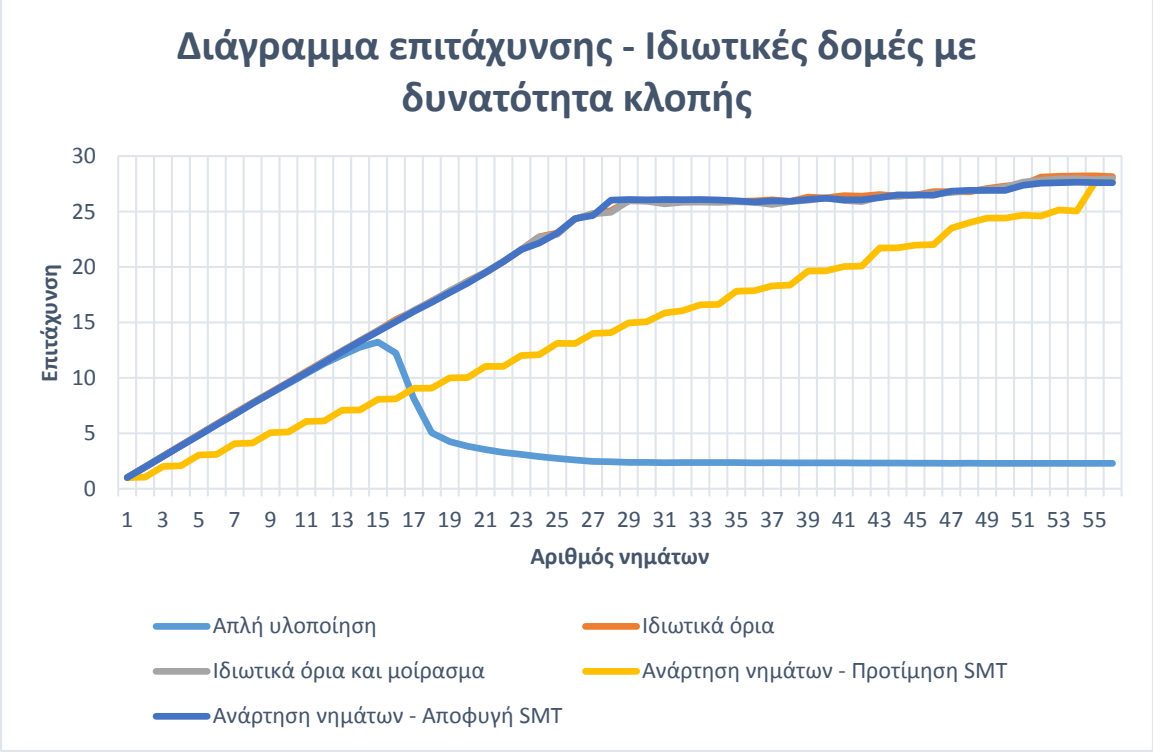
Γράφημα 3-15: DKP, f(x) = x, BestFS, επιτάχυνση, ιδιωτικές δομές με δυνατότητα κλοπής



Γράφημα 3-16: DKP, f(x) = x, DFS, επιτάχυνση, ιδιωτικές δομές με δυνατότητα κλοπής



Γράφημα 3-17: DKP, f(x) = exp(x), BestFS, επιτάχυνση, ιδιωτικές δομές με δυνατότητα κλοπής

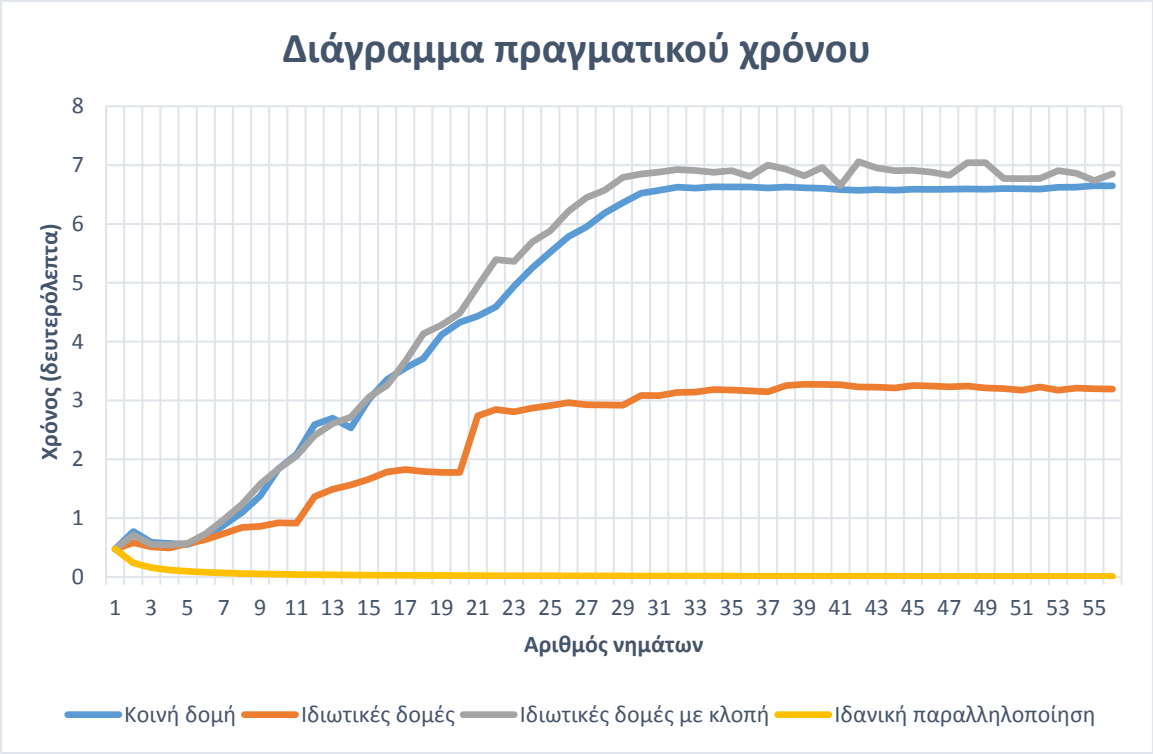


Γράφημα 3-18: DKP, f(x) = exp(x), επιτάχυνση, ιδιωτικές δομές με δυνατότητα κλοπής

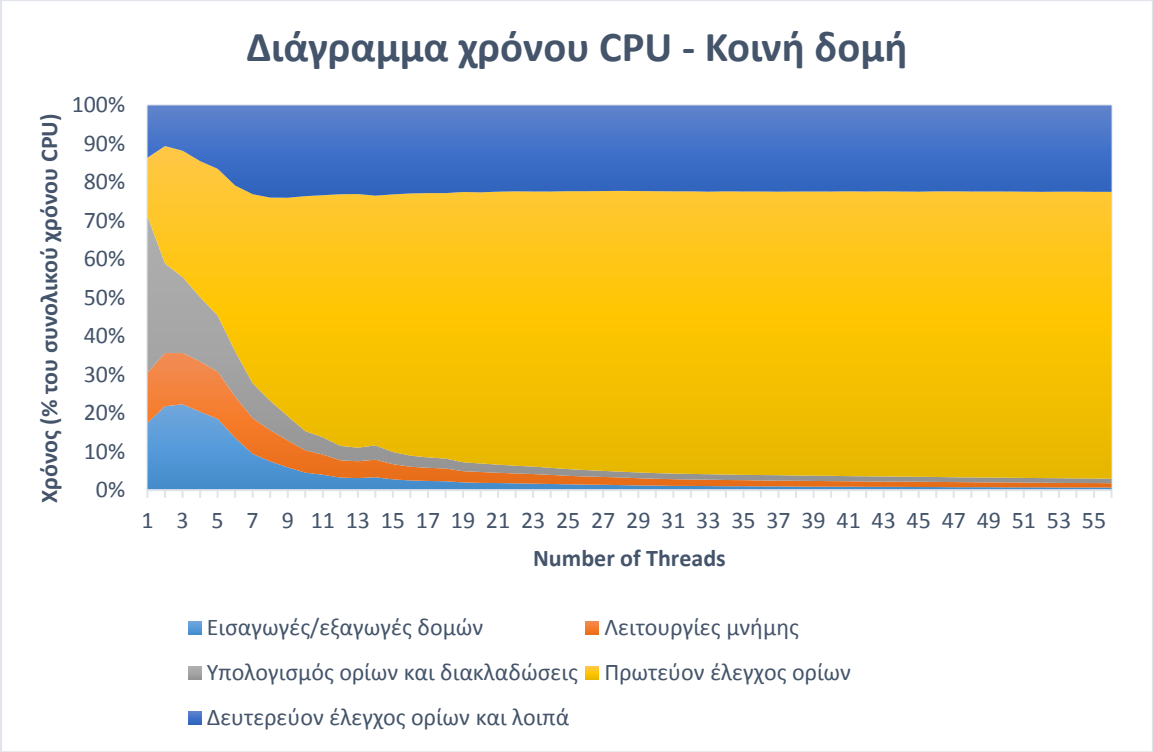
3.2.2 Μετρήσεις προβλήματος του πλανόδιου πωλητή

Σε αυτό το πρόβλημα, τα αποτελέσματα είναι αρκετά πιο ανάμικτα. Γενικά, η συμπεριφορά του αλγορίθμου είναι παρόμοια, ανεξαρτήτως του τρόπου επίλυσης και του αλγορίθμου αναζήτησης. Πρέπει να τονισθεί όμως, πως τελικά μεγαλύτερη επιτάχυνση επιτυγχάνεται με τη χρήση δυαδικού δένδρου. Στα παρακάτω γραφήματα, ο αλγόριθμος που χρησιμοποιεί δυαδικό δένδρο σημειώνεται ως «**E-alg**» και ο αλγόριθμος με δένδρου βαθμού μεγαλύτερο του δύο αναφέρεται ως «**V-alg**».

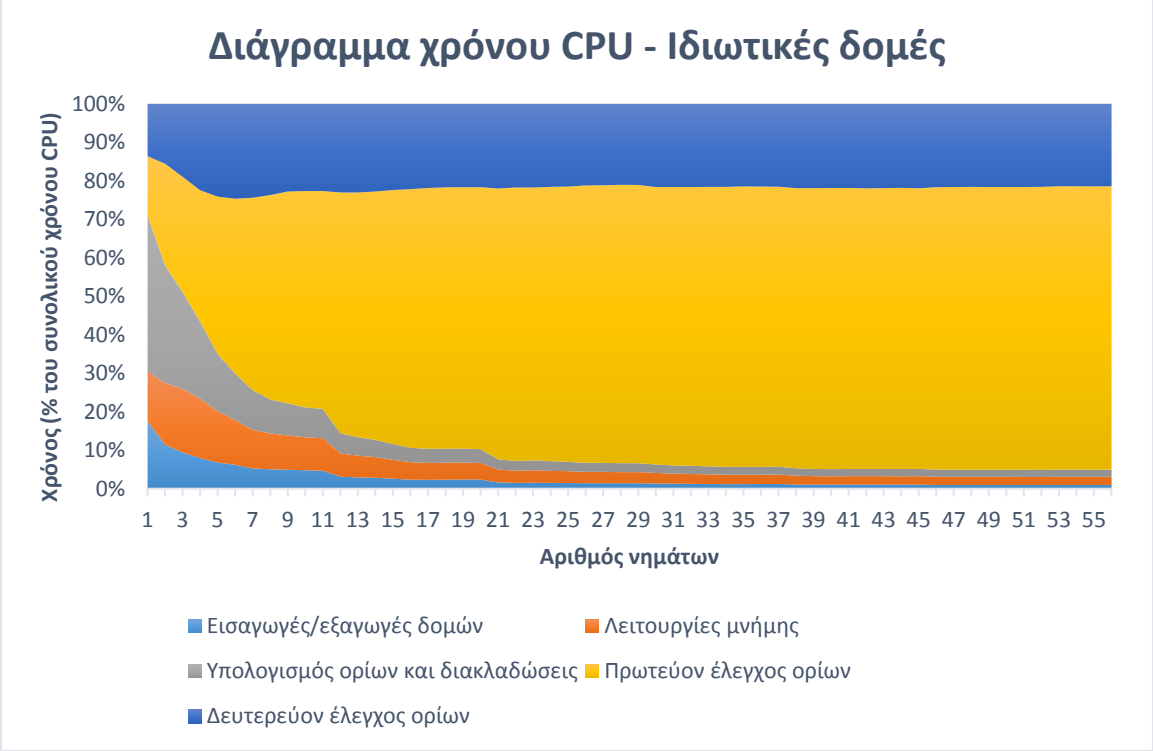
Στην απλή υλοποίηση, ουσιαστικά δεν επιτυγχάνεται καθόλου επιτάχυνση. Για την ακρίβεια, όσο αυξάνουν τα νήματα (γραφήματα 3-19, 3-23, 3-24 και 3-25), χειροτερεύει ο χρόνος εκτέλεσης. Τα διαγράμματα χρόνου CPU δείχνουν ότι, ενώ στο σειριακό αλγόριθμο οι διάφοροι χρόνοι που μετρούνται είναι ίδιας τάξης μεγέθους, καθώς αυξάνουν τα νήματα επικρατούν συντριπτικά οι χρόνοι ελέγχου των ορίων (γραφήματα 3-20, 3-21 και 3-22).



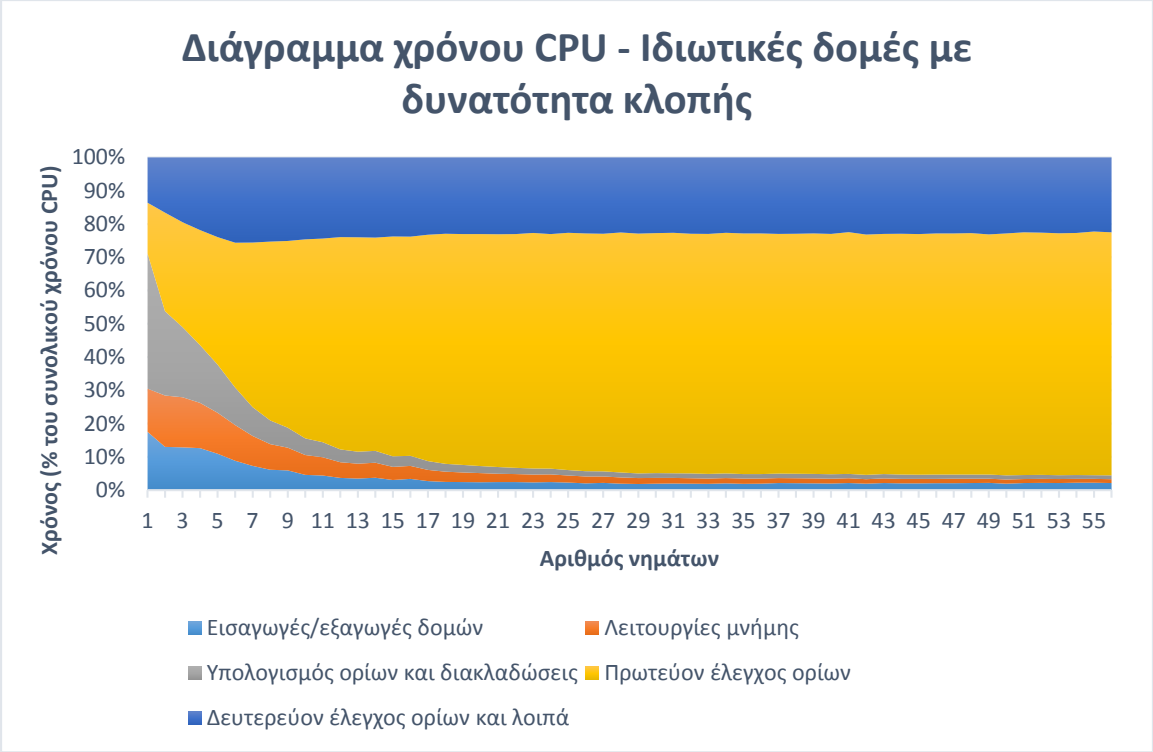
Γράφημα 3-19: TSP, V-alg, BestFS, πραγματικός χρόνος, απλή υλοποίηση



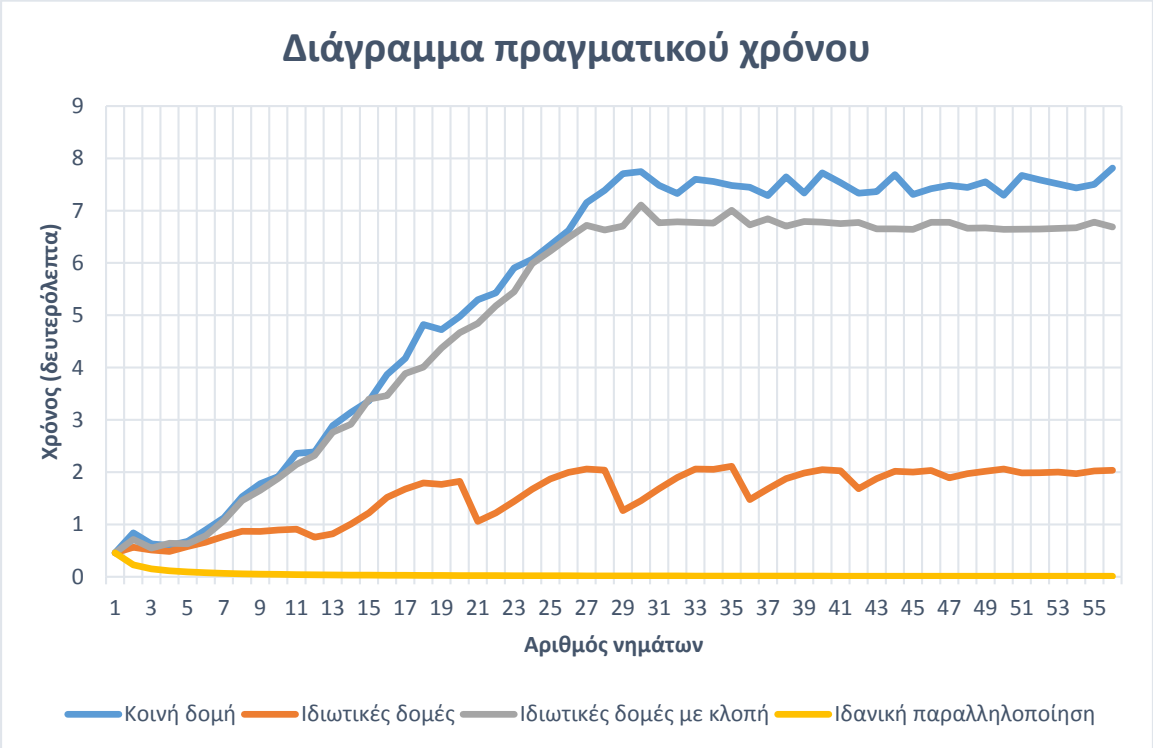
Γράφημα 3-20: TSP, V-alg, BestFS, απλή υλοποίηση, πραγματικός χρόνος, κοινή δομή



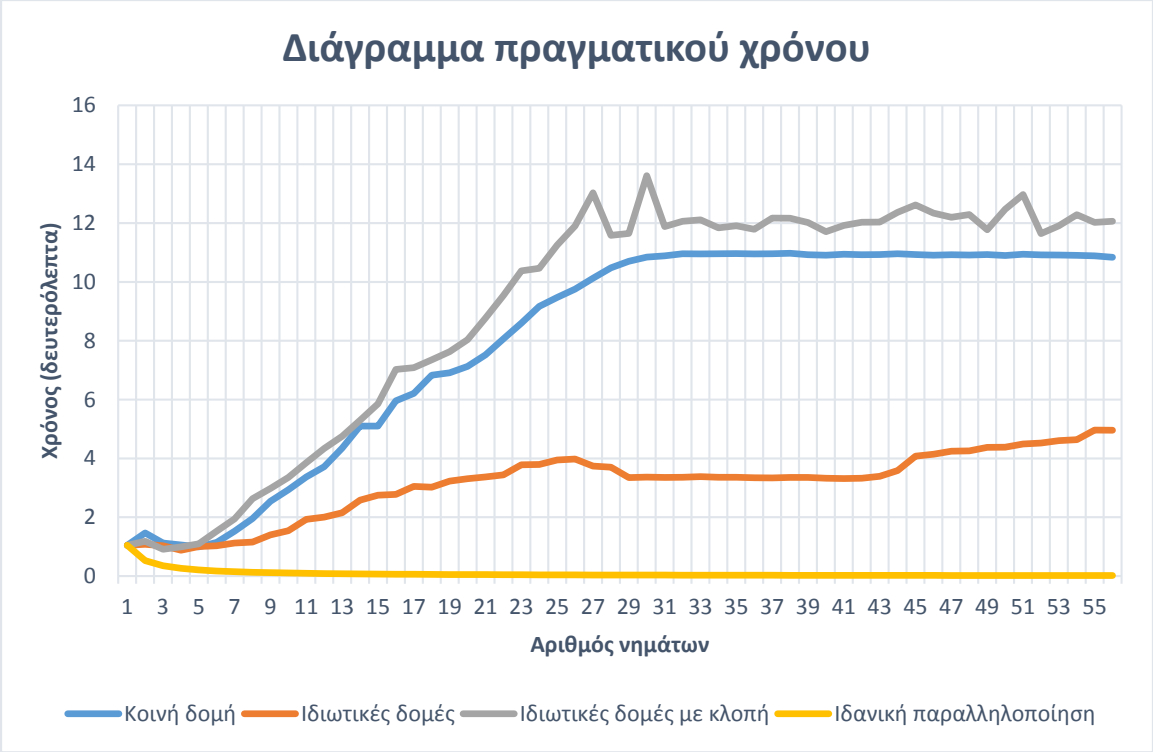
Γράφημα 3-21: TSP, V-alg, BestFS, απλή υλοποίηση, χρόνος CPU, ιδιωτικές δομές



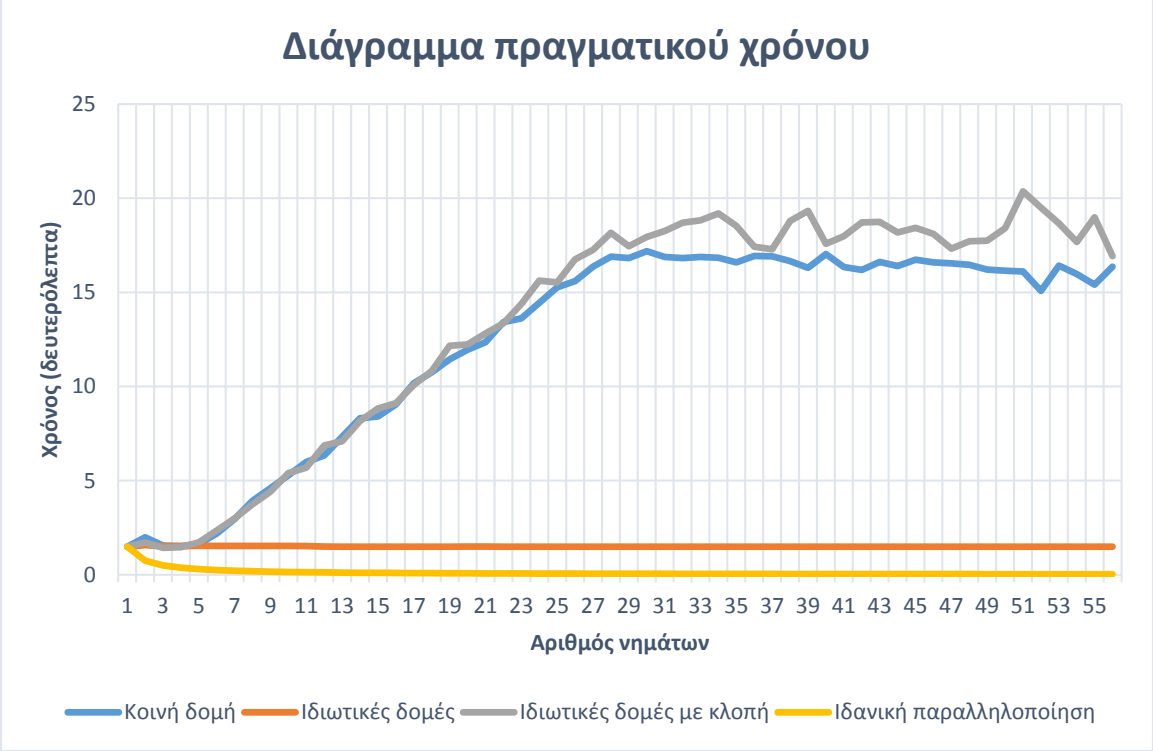
Γράφημα 3-22: TSP, V-alg, BestFS, απλή υλοποίηση, χρόνος CPU, ιδιωτικές δομές με δυνατότητα κλοπής



Γράφημα 3-23: TSP, V-alg, DFS, απλή υλοποίηση, πραγματικός χρόνος

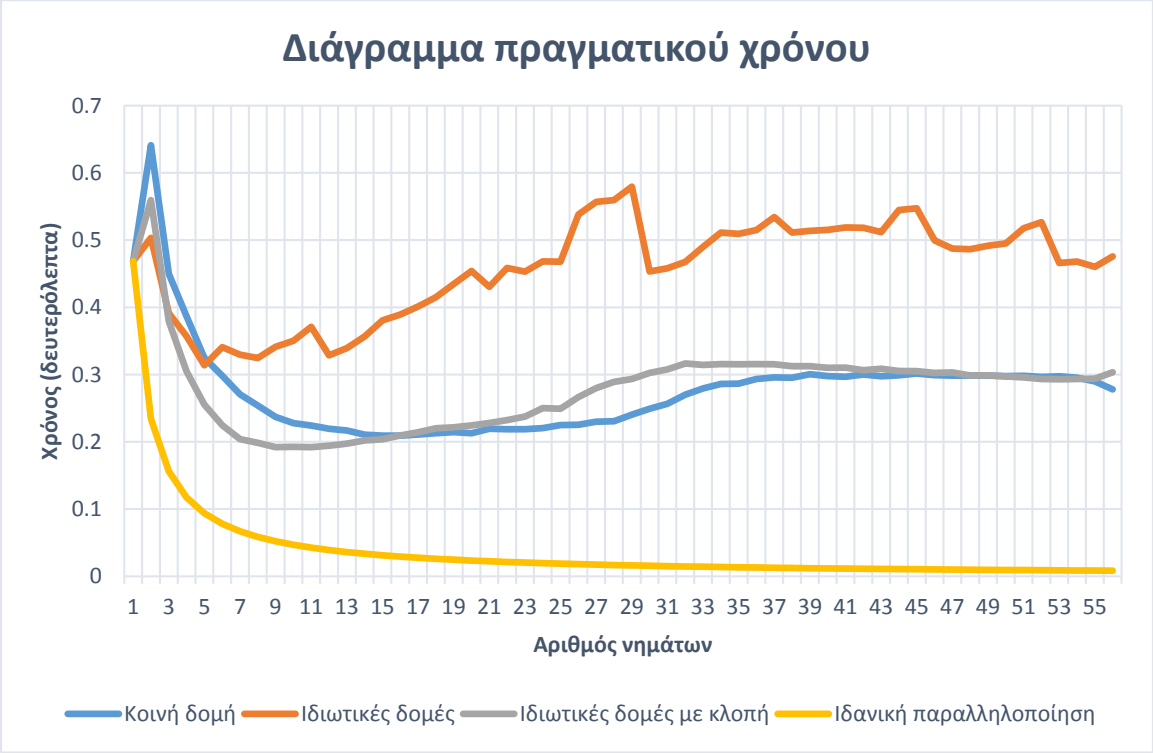


Γράφημα 3-24: TSP, E-alg, BestFS, απλή υλοποίηση, πραγματικός χρόνος

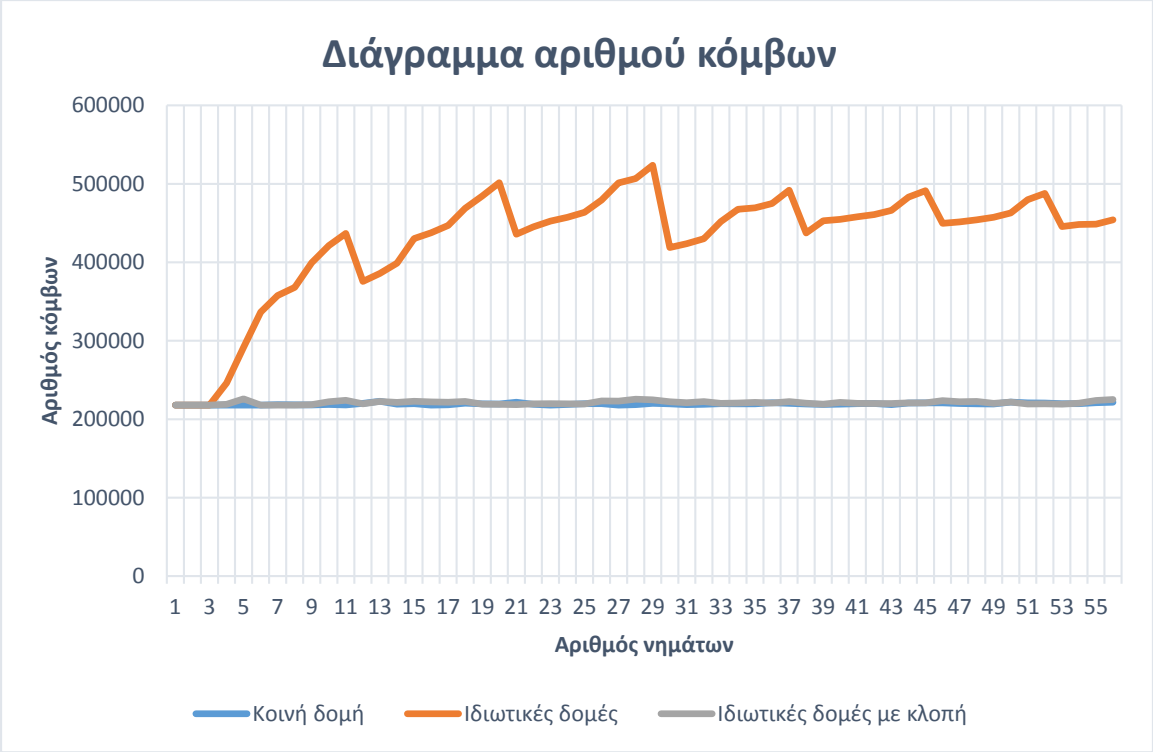


Γράφημα 3-25: TSP, E-alg, DFS, απλή υλοποίηση, πραγματικός χρόνος

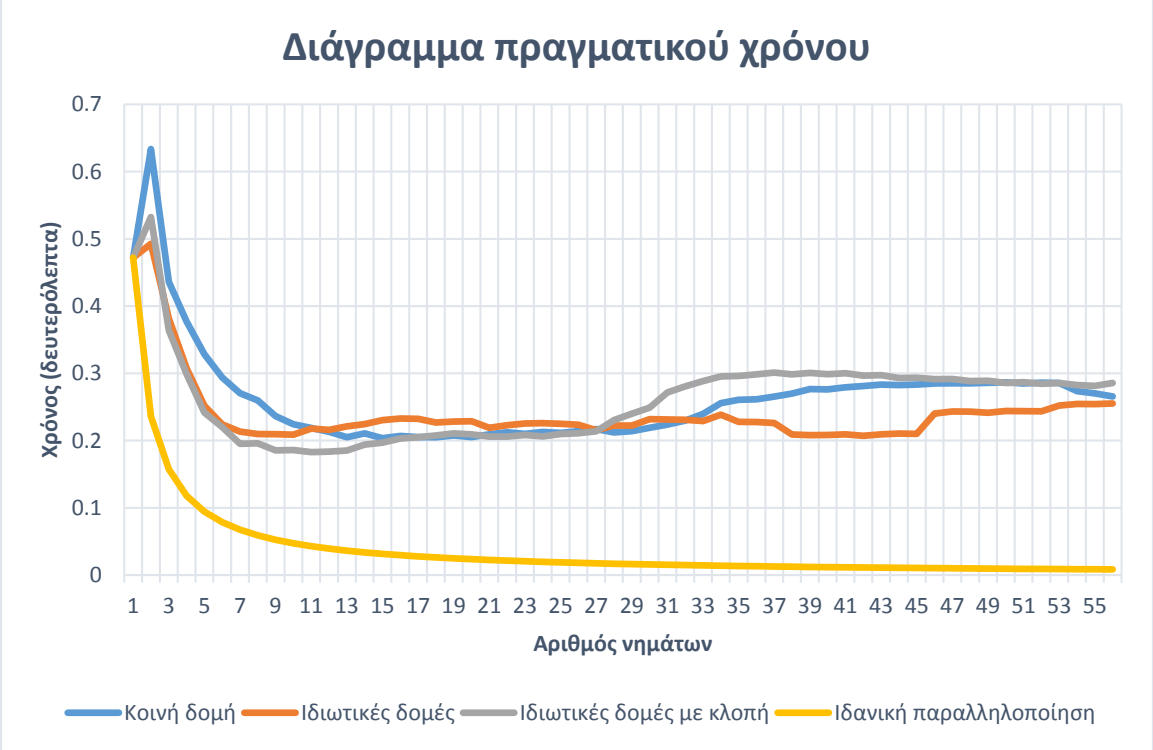
Η χρήση ιδιωτικών ορίων, όπως και πριν, βελτιώνει σημαντικά και τους τρεις διαφορετικούς τύπους δομής αποθήκευσης των υποσυνόλων λύσεων. Όμως, παρατηρείται πως οι απλές ιδιωτικές δομές υστερούν ελαφρώς (γράφημα 3-26). Αυτό δικαιολογείται από το γεγονός πως σε αυτή την περίπτωση αυξάνει σημαντικά ο αριθμός των κόμβων (γράφημα 3-27). Ο περιοδικός συγχρονισμός των ορίων διορθώνει το πρόβλημα και όλοι οι τύποι της δομής αποκτούν πανομοιότυπη συμπεριφορά (γράφημα 3-28).



Γράφημα 3-26: TSP, V-alg, BestFS, ιδιωτικά όρια, πραγματικός χρόνος

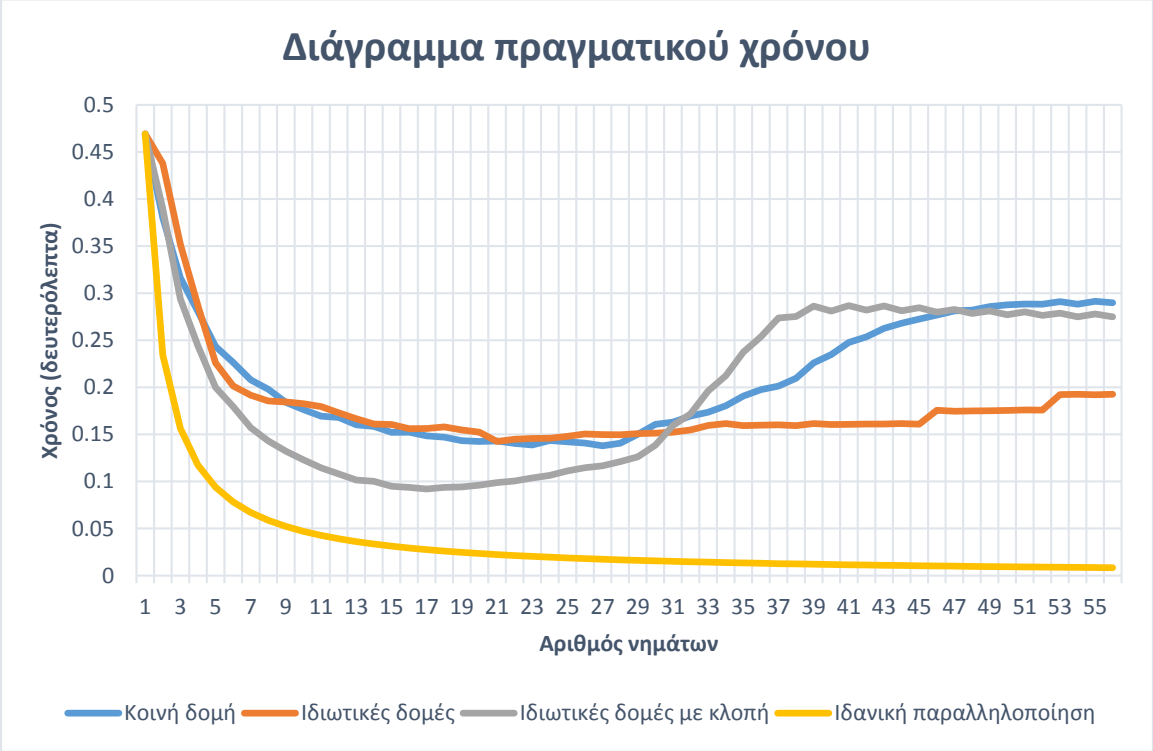


Γράφημα 3-27: TSP, V-alg, BestFS, ιδιωτικά όρια, αριθμός κόμβων

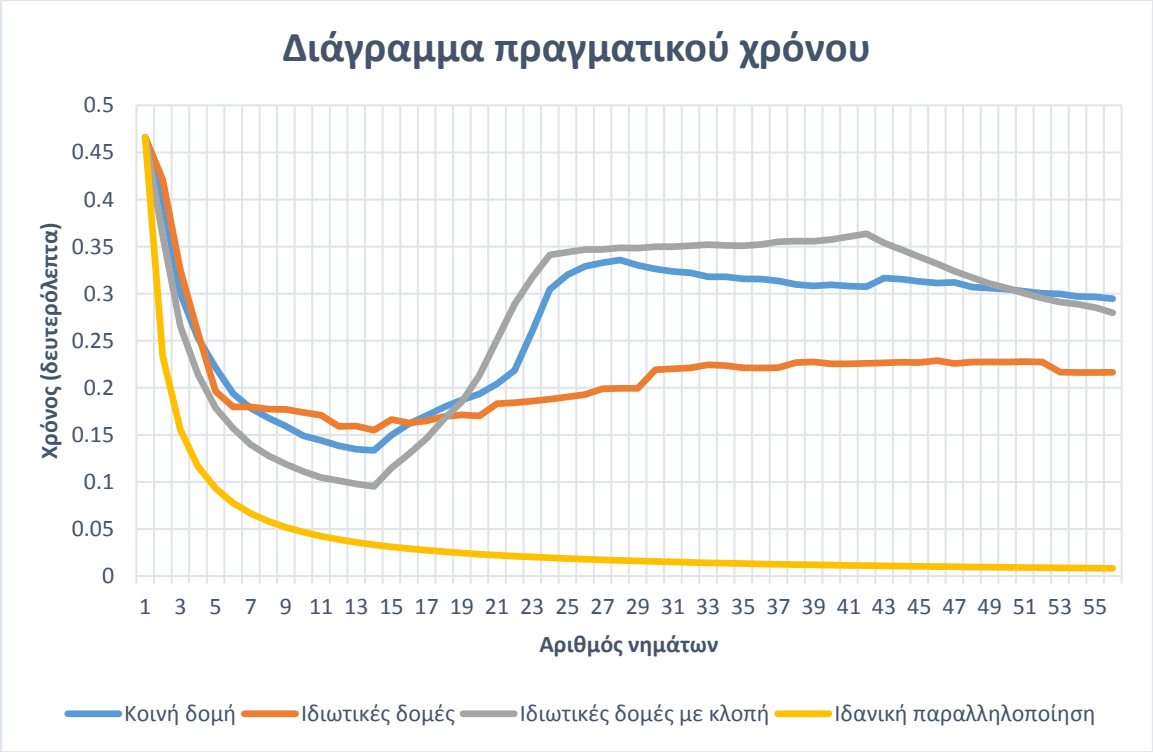


Γράφημα 3-28: TSP, V-alg, BestFS, ιδιωτικά όρια και συγχρονισμός, πραγματικός χρόνος

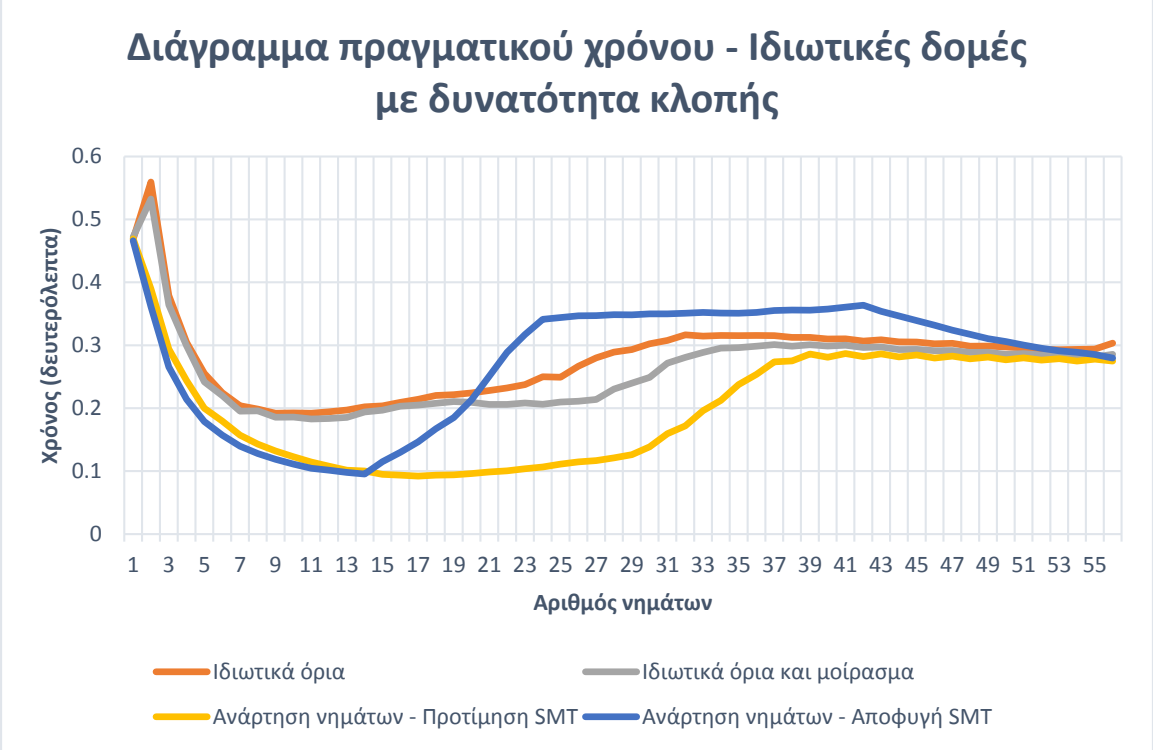
Η ανάρτηση νημάτων αλλάζει πάλι τα τελικά αποτελέσματα, καθώς βελτιώνει τις περιπτώσεις χρήσεις κοινής δομής και ιδιωτικών δομών με δυνατότητα κλοπής (γραφήματα 3-29 και 3-30). Φαίνεται πως μειώνει τα bottlenecks μνήμης που προκύπτουν από την ανταλλαγή δεδομένων μεταξύ των νημάτων. Η χρήση απλών ιδιωτικών δομών δε βελτιώνεται, καθώς κάθε νήμα εργάζεται σε αποκλειστικά δικά του δεδομένα. Ενδιαφέρον έχει επίσης, πως τελικά μεγαλύτερη επιτάχυνση παρατηρείται όταν προτιμάται η χρήση λιγότερων φυσικών πυρήνων (γραφήματα 3-34 και 3-35), όπως και επίσης ότι υπάρχει κατά περίπτωση σχετικά μικρός αριθμός νημάτων που είναι βέλτιστος, καθώς η απόδοση μειώνεται σημαντικά πέρα από αυτόν.



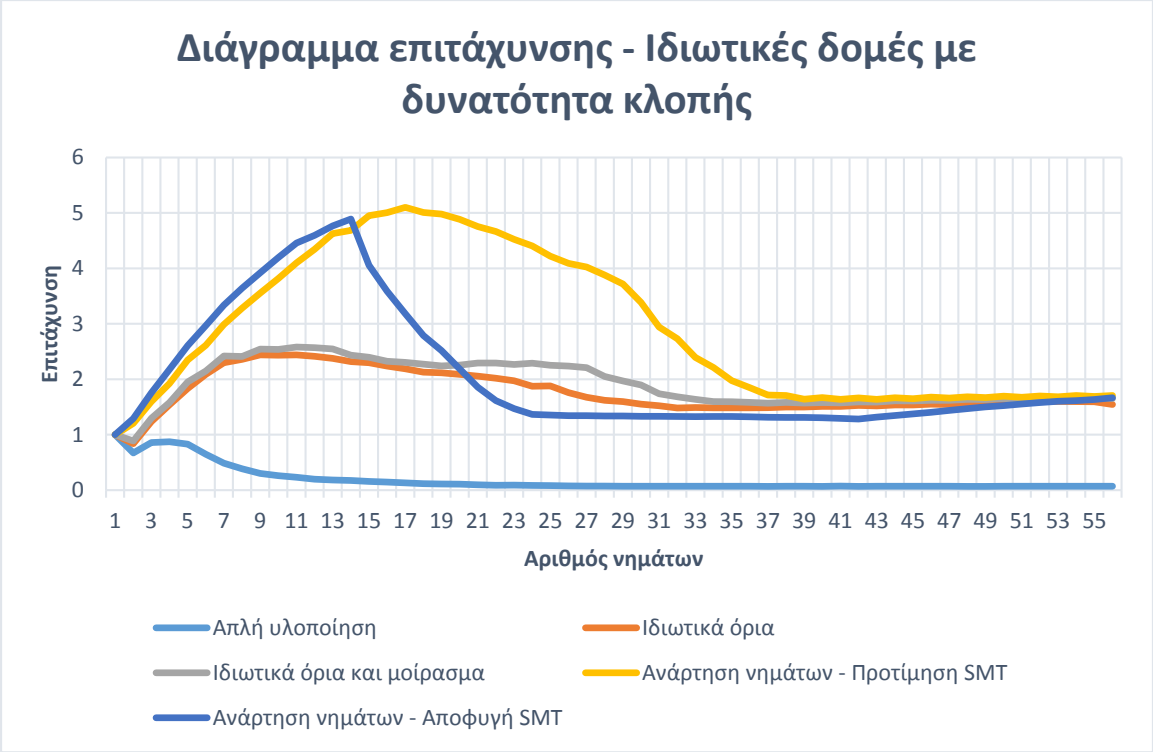
Γράφημα 3-29: TSP, V-alg, BestFS, ανάρτηση νημάτων - προτίμηση SMT, πραγματικός χρόνος



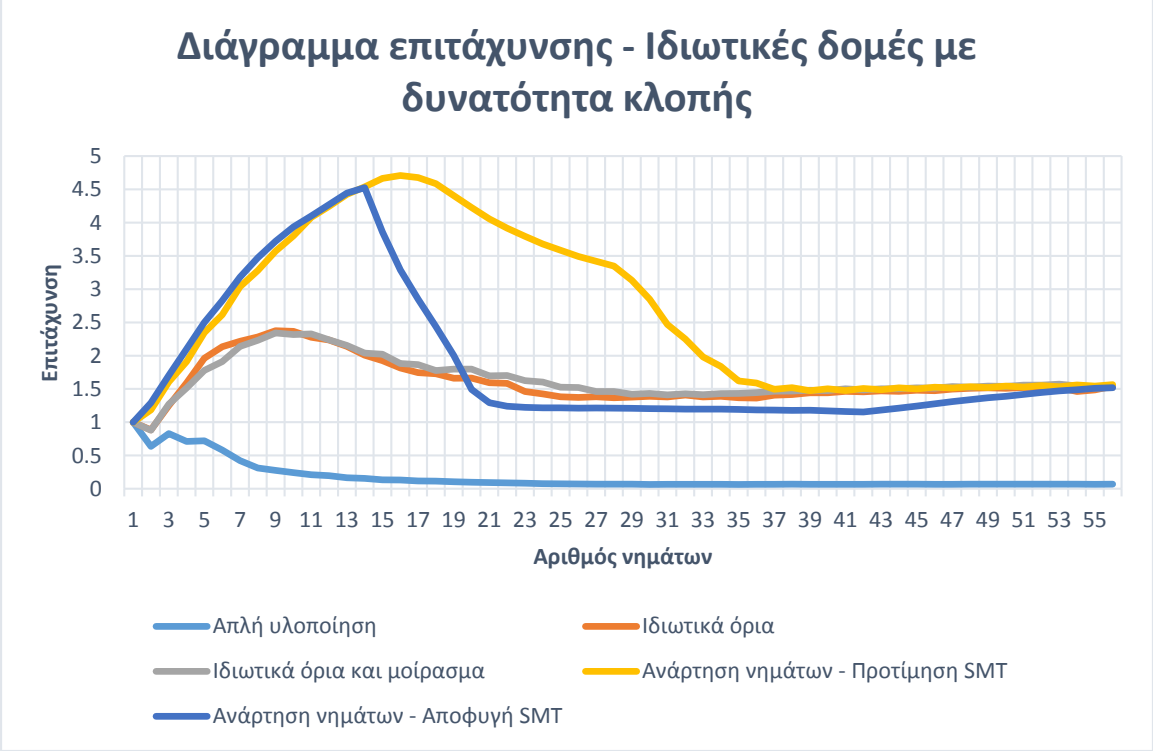
Γράφημα 3-30: TSP, V-alg, BestFS, ανάρτηση νημάτων - αποφυγή SMT, πραγματικός χρόνος



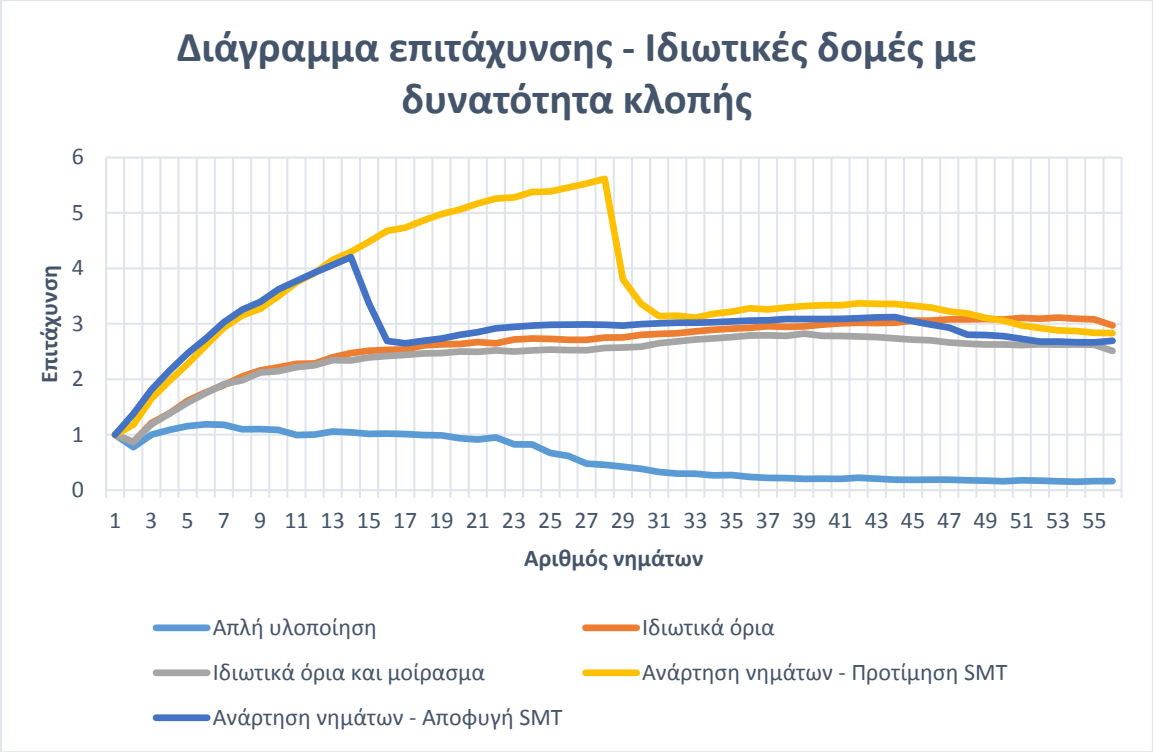
Γράφημα 3-31: TSP, V-alg, BestFS, πραγματικός χρόνος, ιδιωτικές δομές με δυνατότητα κλοπής



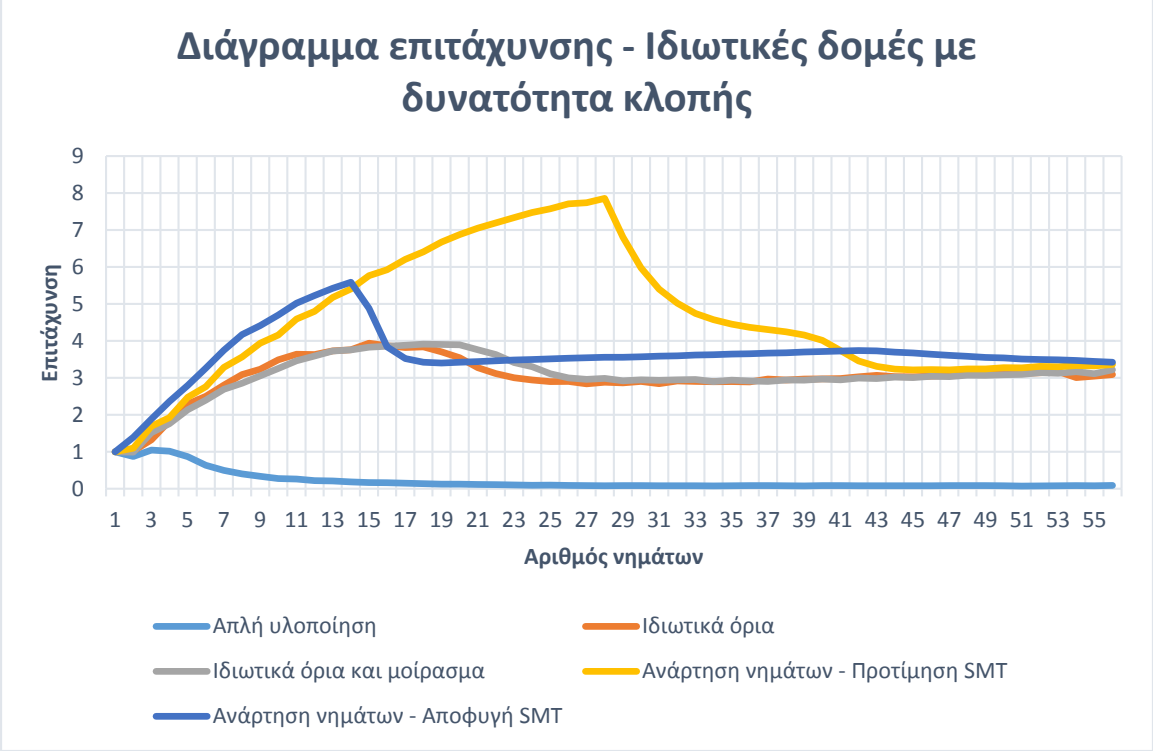
Γράφημα 3-32: TSP, V-alg, BestFS, επιτάχυνση, ιδιωτικές δομές με δυνατότητα κλοπής



Γράφημα 3-33: TSP, V-alg, DFS, επιτάχυνση, ιδιωτικές δομές με δυνατότητα κλοπής



Γράφημα 3-34: TSP, E-alg, BestFS, επιτάχυνση, ιδιωτικές δομές με δυνατότητα κλοπής



Γράφημα 3-35: TSP, E-alg, DFS, επιτάχυνση, ιδιωτικές δομές με δυνατότητα κλοπής

4 Συμπεράσματα

Σε αυτή τη μελέτη περιγράφηκε ο βασικός αλγόριθμος B&B, με σειριακή και παράλληλη εκτέλεση. Κατασκευάστηκε σχετική προγραμματιστική βιβλιοθήκη σε γλώσσα C++ για υπολογιστικά συστήματα μοιραζόμενης μνήμης και με βάση αυτή έγιναν υλοποιήσεις αλγορίθμων B&B για το διακριτό πρόβλημα του σακιδίου και το πρόβλημα του πλανόδιου πωλητή. Πραγματοποιήθηκαν μετρήσεις σε υπολογιστικό σύστημα με δυνατότητα παράλληλης εκτέλεσης μέχρι και 56 διαφορετικών νημάτων, που έδειξαν ότι είναι εφικτό να επιταχυνθεί σημαντικά η εκτέλεση του αλγορίθμου B&B, εφόσον αντιμετωπιστούν τα bottlenecks.

Συγκεκριμένα, φάνηκε ότι, ενώ ο παράλληλος αλγόριθμος B&B δεν έχει τμήματα εγγενώς σειριακά, η απαίτηση για συγχρονισμό των νημάτων κατά την πρόσβαση στη δομή αποθήκευσης των υποσυνόλων λύσεων και των έλεγχο των ορίων προκαλεί συχνά αναστολή της λειτουργίας τους. Όσον αφορά τη δομή αποθήκευσης των υποσυνόλων λύσεων, δοκιμάστηκαν τρεις υλοποιήσεις, χρησιμοποιώντας μία μοιραζόμενη δομή, ιδιωτικές δομές για κάθε νήμα και ιδιωτικές δομές με δυνατότητα κλοπής υποσυνόλων. Από τα αποτελέσματα προέκυψε πως καλύτερη απόδοση συνολικά επιτυγχάνεται με τις τελευταίες, καθώς εξαλείφουν τον ανταγωνισμό των νημάτων κατά την πρόσβαση στα υποσύνολα λύσεων, αλλά και αναδιανέμουν δυναμικά το φόρτο τους. Επιπλέον, προέκυψε πως τεχνικές, όπως η χρήση ιδιωτικών ορίων, μειώνουν σημαντικά τις καθυστερήσεις κατά τον έλεγχο τους, ενώ απαιτείται μόνο περιοδικός συγχρονισμός τους, προκειμένου να εξασφαλίζεται ότι οι εργάτες δεν αναλύουν άχρηστα υποσύνολα λύσεων.

Ακόμα, όπως φάνηκε από τις μετρήσεις (§3.2.1 και §3.2.2), η ανάρτηση νημάτων επηρεάζει και βελτιώνει τα αποτελέσματα, αλλά χρειάζεται προσοχή στο κατά πόσο θα προτιμηθεί η χρήση των διαφορετικών φυσικών πυρήνων ή όχι, εφόσον διατίθεται δυνατότητα SMT. Όταν στο χρόνο εκτέλεσης του αλγορίθμου επικρατεί ο χρόνος υπολογισμού των ορίων και των διακλαδώσεων, τότε φαίνεται πως πρέπει να προτιμάται η χρήση διαφορετικών φυσικών πυρήνων για τα νήματα. Όμως, καθώς οι χρόνοι εκτέλεσης των διαφόρων μερών του αλγορίθμου B&B γίνονται ίδιας τάξης μεγέθους, ενδεχομένως να επιτυγχάνεται καλύτερη απόδοση με προτίμηση χρήσης ενός φυσικού πυρήνα για περισσότερα του ενός νήματα.

Ως μελλοντική επέκταση μπορεί να μελετηθούν περαιτέρω αλγόριθμοι B&B, όπως αυτοί που παρουσιάστηκαν για το πρόβλημα του πλανόδιου πωλητή. Σε αυτούς τους αλγορίθμους, παρόλο που επιτυγχάνεται επιτάχυνση της εκτέλεσης, αυτή είναι σχετικά μικρή και μάλιστα μειώνεται από έναν αριθμό νημάτων και πέρα. Μία πιθανή εξήγηση είναι ότι η ανταλλαγή δεδομένων μεταξύ νημάτων που εκτελούνται σε διαφορετικούς επεξεργαστές είναι εξαιρετικά ακριβή χρονικά. Σε αυτό το πλαίσιο, θα μπορούσε να προστεθεί στη βιβλιοθήκη που αναπτύχθηκε ένα επιπλέον επίπεδο διαμοίρασης των εργασιών, που θα λαμβάνει υπόψιν τους διαφορετικούς επεξεργαστές και θα ελαχιστοποιεί ή και θα εξαλείφει πλήρως την ανταλλαγή δεδομένων μεταξύ αυτών. Επίσης, θα είχε ενδιαφέρον να γίνει επέκταση της βιβλιοθήκης ώστε να υποστηρίζει υπολογιστικά συστήματα κατανεμημένης μνήμης και να μελετηθεί το μέγεθος του χρόνου υπολογισμού των ορίων και των διακλαδώσεων που απαιτείται ως προς τους υπολοίπους, προκειμένου να είναι εφικτή η επιτάχυνση του αλγορίθμου B&B.

5 Βιβλιογραφία

Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. *AFIPS '67 (Spring) Proceedings of the April 18-20, 1967, spring joint computer conference* (σσ. 483-485). ACM. doi:10.1145/1465482.1465560

Chvátal, V. (1980, December 1). Hard Knapsack Problems. *Operations Research, 28*(6), σσ. 1402-14011. Ανάκτηση από http://dx.doi.org/10.1287/opre.28.6.1402

Clausen, J. (1999, 12 March). Branch and Bound Algorithms—Principles and Examples. Copenhagen, Denmark. Ανάκτηση από http://www.diku.dk/OLD/undervisning/2003e/datV-optimer/JensClausenNoter.pdf

Gendron, B., & Crainic, T. G. (1994, November-December). Parallel Branch-And-Bound Algorithms: Survey and Synthesis. *Operations Research, 42*(6), 1042-1066.

Land, A. H., & Doig, A. G. (1960, July). An automatic method of solving discrete programming problems. *Econometrica, 28*(3), 497-520. doi:10.2307/1910129

Leroy, R., Mezmaz, M., Melab, N., & Tuyttens, D. (2014). Work Stealing Strategies For Multi-Core Parallel Branch-and-Bound Algorithm Using Factorial Number System. *Proceedings of Programming Models and Applications on Multicores and Manycores (PMAM'14).* ACM. doi:10.1145/2560683.2560694

Li, G.-j., & Wah, B. W. (1984). How to cope with anomalies in parallel approximate branch-and-bound algorithms. *Proceedings of the Fourth National Conference on Artificial Intelligence (AAAI-84)*, (σσ. 212-215). Austin, Texas.

Little, J. D., Murty, K. G., Sweeney, D. W., & Karel, C. (1963, December 1). An Algorithm for the Traveling Salesman Problem. *Operations Research, 11*(6), 972-989. doi:10.1287/opre.11.6.972

Mahapatra, N. R., & Dutt, S. (1996). Sequential and parallel branch-and-bound search under limited-memory constraints. *Proc. Parallel Optimization Colloquium (POC 1996)*, (σσ. 147-165). Versailles, France.

Office Research Group Discrete Optimization, Heidelberg University. (n.d.). *TSPLIB*. Ανάκτηση από Discrete and Combinatorial Optimization: http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/

Pisinger, D. (1995). *Algorithms for knapsack problems.* University of Copenhagen, Department of Computer Science, Copenhagen, Denmark. Ανάκτηση από http://www.diku.dk/~pisinger/95-1.pdf

Rihani, H., Sanders, P., & Dementiev, R. (2014). *MultiQueues: Simpler, Faster, and Better Relaxed Concurrent Priority Queues.* Université Joseph Fourier Grenoble, France, Karlsruhe Institute of Technology, Germany, Intel GmbH Munich, Germany.

Vizvári, B. (2011). The Branch and Bound Method. Στο *Algorithm of Informatics* (Τόμ. 3, σσ. 1251-1304). Budapest, Hungary: AnTonCom Infokommunikációs Kft.

Wilmarth, T. (2010, December 2). *Intel Threading Building Blocks Version 3.0 Update 4 showcases its first Community Preview feature: Concurrent Priority Queue*. Ανάκτηση από Intel Corporation Web site: https://software.intel.com/en-us/blogs/2010/12/02/intel-threading-building-blocks-version-30-update-4-showcases-its-first-community-preview-feature-concurrent-priority-queue

Παράρτημα: Documentation της Βιβλιοθήκης

Table of Contents

1 Hierarchical Index

1.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

2 Class Index

2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

3 File Index

3.1 File List

Here is a list of all files with brief descriptions:

4 Class Documentation

4.1 benchmark Class Reference

A benchmarking class.

`#include <benchmark.hpp>`

4.1.1 Public Member Functions

- **benchmark** ()
  The default constructor.
- **benchmark** (const **benchmark** &**benchmark**)
  The copy constructor.
- void **setFathomedNodesNum** (size_t value)
  Sets the number of fathomed nodes.
- void **setSolutionSwapsNum** (size_t value)
  Sets the number of solution swaps.
- void **setQueueMaxSize** (size_t value)
  Sets the maximum observed size of the queue.
- void **setQueuePushesNum** (size_t value)
  Sets the number of push operations.
- void **setQueuePopsNum** (size_t value)
  Sets the number of pop operations.
- void **setNodesCreatedNum** (size_t value)
  Sets the number of generated node objects.
- void **setSpareNodesUsedNum** (size_t value)
  Sets the number of recycled node objects.
- void **setNodesStolenNum** (size_t value)
  Set the number of "stolen" nodes.
- void **setComplexity** (size_t value)
  Sets the complexity.
- void **setTotalTime** (tbb::tick_count::interval_t interval)
  Sets the actual total time.
- void **setThreadTime** (tbb::tick_count::interval_t interval)
  Sets the thread CPU time.
- void **setQueuePushesTime** (tbb::tick_count::interval_t interval)
  Sets the queue insertions time.
- void **setQueuePopsTime** (tbb::tick_count::interval_t interval)
  Sets the queue extractions time.
- void **setMemoryTime** (tbb::tick_count::interval_t interval)
  Sets the memory operations time.
- void **setBranchTime** (tbb::tick_count::interval_t interval)
  Sets the branching time.
- void **setBoundTime** (tbb::tick_count::interval_t interval)
  Sets the bounding time.

- void **setSolutionChecksTime** (tbb::tick_count::interval_t interval)
  Sets the bound checking time.
- void **incFathomedNodesNum** ()
  Increases the fathomed nodes by one.
- void **incSolutionSwapsNum** ()
  Increases the solution swaps by one.
- void **updateQueueMaxSize** (size_t size)
  Updates the maximum observed queue size.
- void **incQueuePushesNum** ()
  Increases the number of push operations by one.
- void **incQueuePopsNum** ()
  Increases the number of pop operations by one.
- void **incNodesCreatedNum** ()
  Increases the number of generated node objects by one.
- void **incSpareNodesUsedNum** ()
  Increases the number of recycled node objects by one.
- void **incNodesStolenNum** ()
  Increases the number of "stolen" nodes by one.
- void **incComplexity** (size_t value)
  Increases the complexity by a set value.
- void **incTotalTime** (tbb::tick_count::interval_t interval)
  Increases the actual total time by a set interval.
- void **incThreadTime** (tbb::tick_count::interval_t interval)
  Increases the thread CPU time by a set interval.
- void **incQueuePushesTime** (tbb::tick_count::interval_t interval)
  Increases the queue insertions time by a set interval.
- void **incQueuePopsTime** (tbb::tick_count::interval_t interval)
  Increases the queue extractions time by a set interval.
- void **incMemoryTime** (tbb::tick_count::interval_t interval)
  Increases the memory operations time by a set interval.
- void **incBranchTime** (tbb::tick_count::interval_t interval)
  Increases the branching time by a set interval.
- void **incBoundTime** (tbb::tick_count::interval_t interval)
  Increases the bounding time by a set interval.
- void **incSolutionChecksTime** (tbb::tick_count::interval_t interval)
  Increases the bound checking time by a set interval.
- size_t **getFathomedNodes** ()
  Returns the number of fathomed nodes.
- size_t **getSolutionSwapsNum** ()
  Returns the number of solution swaps.
- size_t **getQueueMaxSize** ()
  Returns the maximum observed queue size.
- size_t **getQueuePushesNum** ()
  Returns the number of push operations.
- size_t **getQueuePopsNum** ()
  Returns the number of pop operations.

- size_t **getNodesCreatedNum** ()
  Returns the number of generated node objects.
- size_t **getSpareNodesUsedNum** ()
  Returns the number of recycled node objects.
- size_t **getNodesStolenNum** ()
  Returns the number of "stolen" nodes.
- size_t **getComplexity** ()
  Returns the complexity.
- tbb::tick_count::interval_t **getTotalTime** ()
  Returns the actual total time.
- tbb::tick_count::interval_t **getThreadTime** ()
  Returns the thread CPU time.
- tbb::tick_count::interval_t **getQueuePushesTime** ()
  Returns the queue insertions time.
- tbb::tick_count::interval_t **getQueuePopsTime** ()
  Returns the queue extractions time.
- tbb::tick_count::interval_t **getMemoryTime** ()
  Returns the memory operations time.
- tbb::tick_count::interval_t **getBranchTime** ()
  Returns the branching time.
- tbb::tick_count::interval_t **getBoundTime** ()
  Returns the bounding time.
- tbb::tick_count::interval_t **getSolutionChecksTime** ()
  Returns the bound checking time.
- **benchmark** & **operator=** (const **benchmark** &**benchmark**)
  The assignment operator.
- **benchmark operator+** (const **benchmark** &bench) const
  The addition operator.
- **benchmark** & **operator+=** (const **benchmark** &bench)
- void **print** ()
  Prints the measurements.

4.1.2 Static Public Member Functions

- static bool **lessThan** (const tbb::tick_count::interval_t &lhs, const tbb::tick_count::interval_t &rhs)
  Less-than operator for tbb::tick_count::interval_t.
- static **benchmark min** (const **benchmark** &xBench, const **benchmark** &yBench)
  Compares two benchmark objects and returns a new one with the minimum values for each counter and timer.
- static **benchmark max** (const **benchmark** &xBench, const **benchmark** &yBench)
  Compares two benchmark objects and returns a new one with the maximum values for each counter and timer.
- static std::vector< **benchmark** > **reduction** (const std::vector< **benchmark** > &xBench, const std::vector< **benchmark** > &yBench)
  Compares two benchmark objects and returns three new ones with the minimum, total and maximum values for each counter and timer.
- static std::vector< **benchmark** > **reduction** (const std::vector< **benchmark** > &benchVector)
  Reduces a vector of any number of benchmark objects to three objects with the minimum, total and maximum values for each counter and timer.

- static void **write** (std::ofstream &fileStream, const std::vector< **benchmark** > &benchVector)
  Write the measurements in a vector of benchmark objects to a file stream.

4.1.3 Detailed Description

A benchmarking class.

Implements a number of counters and timers in order to measure the performance of the branch and bound algorithm.

Definition at line 10 of file benchmark.hpp.

4.1.4 Constructor & Destructor Documentation

4.1.4.1 benchmark::benchmark () [inline]

The default constructor.

Definition at line 38 of file benchmark.hpp.

4.1.4.2 benchmark::benchmark (const benchmark & benchmark) [inline]

The copy constructor.

4.1.4.2.1 Parameters:

benchmark	The benchmark object to be copied.

Definition at line 60 of file benchmark.hpp.

4.1.5 Member Function Documentation

4.1.5.1 tbb::tick_count::interval_t benchmark::getBoundTime () [inline]

Returns the bounding time.

Definition at line 333 of file benchmark.hpp.

4.1.5.2 tbb::tick_count::interval_t benchmark::getBranchTime () [inline]

Returns the branching time.

Definition at line 328 of file benchmark.hpp.

4.1.5.3 size_t benchmark::getComplexity () [inline]

Returns the complexity.

Definition at line 297 of file benchmark.hpp.

4.1.5.4 size_t benchmark::getFathomedNodes () [inline]

Returns the number of fathomed nodes.

Definition at line 257 of file benchmark.hpp.

4.1.5.5 tbb::tick_count::interval_t benchmark::getMemoryTime () [inline]

Returns the memory operations time.
Definition at line 323 of file benchmark.hpp.

4.1.5.6 size_t benchmark::getNodesCreatedNum () [inline]

Returns the number of generated node objects.
Definition at line 282 of file benchmark.hpp.

4.1.5.7 size_t benchmark::getNodesStolenNum () [inline]

Returns the number of "stolen" nodes.
Definition at line 292 of file benchmark.hpp.

4.1.5.8 size_t benchmark::getQueueMaxSize () [inline]

Returns the maximum observed queue size.
Definition at line 267 of file benchmark.hpp.

4.1.5.9 size_t benchmark::getQueuePopsNum () [inline]

Returns the number of pop operations.
Definition at line 277 of file benchmark.hpp.

4.1.5.10 tbb::tick_count::interval_t benchmark::getQueuePopsTime () [inline]

Returns the queue extractions time.
Definition at line 318 of file benchmark.hpp.

4.1.5.11 size_t benchmark::getQueuePushesNum () [inline]

Returns the number of push operations.
Definition at line 272 of file benchmark.hpp.

4.1.5.12 tbb::tick_count::interval_t benchmark::getQueuePushesTime () [inline]

Returns the queue insertions time.
Definition at line 313 of file benchmark.hpp.

4.1.5.13 tbb::tick_count::interval_t benchmark::getSolutionChecksTime () [inline]

Returns the bound checking time.
Definition at line 338 of file benchmark.hpp.

4.1.5.14 size_t benchmark::getSolutionSwapsNum () [inline]

Returns the number of solution swaps.
Definition at line 262 of file benchmark.hpp.

4.1.5.15 size_t benchmark::getSpareNodesUsedNum () [inline]

Returns the number of recycled node objects.

Definition at line 287 of file benchmark.hpp.

4.1.5.16 tbb::tick_count::interval_t benchmark::getThreadTime () [inline]

Returns the thread CPU time.

Definition at line 308 of file benchmark.hpp.

4.1.5.17 tbb::tick_count::interval_t benchmark::getTotalTime () [inline]

Returns the actual total time.

Definition at line 303 of file benchmark.hpp.

4.1.5.18 void benchmark::incBoundTime (tbb::tick_count::interval_t interval) [inline]

Increases the bounding time by a set interval.

Definition at line 246 of file benchmark.hpp.

4.1.5.19 void benchmark::incBranchTime (tbb::tick_count::interval_t interval) [inline]

Increases the branching time by a set interval.

Definition at line 241 of file benchmark.hpp.

4.1.5.20 void benchmark::incComplexity (size_t value) [inline]

Increases the complexity by a set value.

Definition at line 210 of file benchmark.hpp.

4.1.5.21 void benchmark::incFathomedNodesNum () [inline]

Increases the fathomed nodes by one.

Definition at line 170 of file benchmark.hpp.

4.1.5.22 void benchmark::incMemoryTime (tbb::tick_count::interval_t interval) [inline]

Increases the memory operations time by a set interval.

Definition at line 236 of file benchmark.hpp.

4.1.5.23 void benchmark::incNodesCreatedNum () [inline]

Increases the number of generated node objects by one.

Definition at line 195 of file benchmark.hpp.

4.1.5.24 void benchmark::incNodesStolenNum () [inline]

Increases the number of "stolen" nodes by one.

Definition at line 205 of file benchmark.hpp.

4.1.5.25 void benchmark::incQueuePopsNum () [inline]

Increases the number of pop operations by one.

Definition at line 190 of file benchmark.hpp.

4.1.5.26 void benchmark::incQueuePopsTime (tbb::tick_count::interval_t interval) [inline]

Increases the queue extractions time by a set interval.

Definition at line 231 of file benchmark.hpp.

4.1.5.27 void benchmark::incQueuePushesNum () [inline]

Increases the number of push operations by one.

Definition at line 185 of file benchmark.hpp.

4.1.5.28 void benchmark::incQueuePushesTime (tbb::tick_count::interval_t interval) [inline]

Increases the queue insertions time by a set interval.

Definition at line 226 of file benchmark.hpp.

4.1.5.29 void benchmark::incSolutionChecksTime (tbb::tick_count::interval_t interval) [inline]

Increases the bound checking time by a set interval.

Definition at line 251 of file benchmark.hpp.

4.1.5.30 void benchmark::incSolutionSwapsNum () [inline]

Increases the solution swaps by one.

Definition at line 175 of file benchmark.hpp.

4.1.5.31 void benchmark::incSpareNodesUsedNum () [inline]

Increases the number of recycled node objects by one.

Definition at line 200 of file benchmark.hpp.

4.1.5.32 void benchmark::incThreadTime (tbb::tick_count::interval_t interval) [inline]

Increases the thread CPU time by a set interval.

Definition at line 221 of file benchmark.hpp.

4.1.5.33 void benchmark::incTotalTime (tbb::tick_count::interval_t interval) [inline]

Increases the actual total time by a set interval.

Definition at line 216 of file benchmark.hpp.

4.1.5.34 static bool benchmark::lessThan (const tbb::tick_count::interval_t & lhs, const tbb::tick_count::interval_t & rhs) [inline], [static]

Less-than operator for tbb::tick_count::interval_t.

Definition at line 448 of file benchmark.hpp.

4.1.5.35 static benchmark benchmark::max (const benchmark & xBench, const benchmark & yBench) [inline], [static]

Compares two benchmark objects and returns a new one with the maximum values for each counter and timer.

Definition at line 481 of file benchmark.hpp.

4.1.5.36 static benchmark benchmark::min (const benchmark & xBench, const benchmark & yBench) [inline], [static]

Compares two benchmark objects and returns a new one with the minimum values for each counter and timer.

Definition at line 454 of file benchmark.hpp.

4.1.5.37 benchmark benchmark::operator+ (const benchmark & bench) const [inline]

The addition operator.

4.1.5.37.1 Parameters:

bench	The rhs of the operator.

Definition at line 371 of file benchmark.hpp.

4.1.5.38 benchmark& benchmark::operator+= (const benchmark & bench) [inline]

4.1.5.38.1 Parameters:

bench	The rhs of the operator.

Definition at line 397 of file benchmark.hpp.

4.1.5.39 benchmark& benchmark::operator= (const benchmark & benchmark) [inline]

The assignment operator.

4.1.5.39.1 Parameters:

benchmark	The benchmark object to be copied.

Definition at line 344 of file benchmark.hpp.

4.1.5.40 void benchmark::print () [inline]

Prints the measurements.

Definition at line 423 of file benchmark.hpp.

4.1.5.41 static std::vector<benchmark> benchmark::reduction (const std::vector< benchmark > & xBench, const std::vector< benchmark > & yBench) [inline], [static]

Compares two benchmark objects and returns three new ones with the minimum, total and maximum values for each counter and timer.

Definition at line 508 of file benchmark.hpp.

4.1.5.42 static std::vector<benchmark> benchmark::reduction (const std::vector< benchmark > & benchVector) [inline], [static]

Reduces a vector of any number of benchmark objects to three objects with the minimum, total and maximum values for each counter and timer.

Definition at line 522 of file benchmark.hpp.

4.1.5.43 void benchmark::setBoundTime (tbb::tick_count::interval_t interval) [inline]

Sets the bounding time.

Definition at line 159 of file benchmark.hpp.

4.1.5.44 void benchmark::setBranchTime (tbb::tick_count::interval_t interval) [inline]

Sets the branching time.

Definition at line 154 of file benchmark.hpp.

4.1.5.45 void benchmark::setComplexity (size_t value) [inline]

Sets the complexity.

Definition at line 123 of file benchmark.hpp.

4.1.5.46 void benchmark::setFathomedNodesNum (size_t value) [inline]

Sets the number of fathomed nodes.

Definition at line 83 of file benchmark.hpp.

4.1.5.47 void benchmark::setMemoryTime (tbb::tick_count::interval_t interval) [inline]

Sets the memory operations time.

Definition at line 149 of file benchmark.hpp.

4.1.5.48 void benchmark::setNodesCreatedNum (size_t value) [inline]

Sets the number of generated node objects.

Definition at line 108 of file benchmark.hpp.

4.1.5.49 void benchmark::setNodesStolenNum (size_t value) [inline]

Set the number of "stolen" nodes.

Definition at line 118 of file benchmark.hpp.

4.1.5.50 void benchmark::setQueueMaxSize (size_t value) [inline]

Sets the maximum observed size of the queue.

Definition at line 93 of file benchmark.hpp.

4.1.5.51 void benchmark::setQueuePopsNum (size_t value) [inline]

Sets the number of pop operations.

Definition at line 103 of file benchmark.hpp.

4.1.5.52 void benchmark::setQueuePopsTime (tbb::tick_count::interval_t interval) [inline]

Sets the queue extractions time.

Definition at line 144 of file benchmark.hpp.

4.1.5.53 void benchmark::setQueuePushesNum (size_t value) [inline]

Sets the number of push operations.

Definition at line 98 of file benchmark.hpp.

4.1.5.54 void benchmark::setQueuePushesTime (tbb::tick_count::interval_t interval) [inline]

Sets the queue insertions time.

Definition at line 139 of file benchmark.hpp.

4.1.5.55 void benchmark::setSolutionChecksTime (tbb::tick_count::interval_t interval) [inline]

Sets the bound checking time.

Definition at line 164 of file benchmark.hpp.

4.1.5.56 void benchmark::setSolutionSwapsNum (size_t value) [inline]

Sets the number of solution swaps.

Definition at line 88 of file benchmark.hpp.

4.1.5.57 void benchmark::setSpareNodesUsedNum (size_t value) [inline]

Sets the number of recycled node objects.

Definition at line 113 of file benchmark.hpp.

4.1.5.58 void benchmark::setThreadTime (tbb::tick_count::interval_t interval) [inline]

Sets the thread CPU time.

Definition at line 134 of file benchmark.hpp.

4.1.5.59 void benchmark::setTotalTime (tbb::tick_count::interval_t interval) [inline]

Sets the actual total time.

Definition at line 129 of file benchmark.hpp.

4.1.5.60 void benchmark::updateQueueMaxSize (size_t size) [inline]

Updates the maximum observed queue size.

Definition at line 180 of file benchmark.hpp.

4.1.5.61 static void benchmark::write (std::ofstream & fileStream, const std::vector< benchmark > & benchVector) [inline], [static]

Write the measurements in a vector of benchmark objects to a file stream.

Definition at line 537 of file benchmark.hpp.

4.1.5.62 The documentation for this class was generated from the following file:

- **benchmark.hpp**

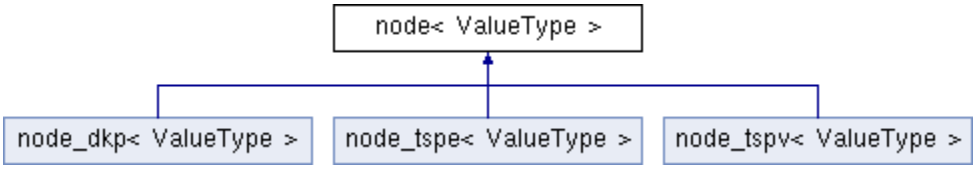
4.1.5.63

4.2 node< ValueType > Class Template Reference

The basic tree-node class.

`#include <node.hpp>`

Inheritance diagram for node< ValueType >:



4.2.1 Public Member Functions

- virtual ValueType **getUpperBound** ()=0
  Returns the upper bound value.
- virtual ValueType **getLowerBound** ()=0
  Returns the lower bound value.
- virtual size_t **getDepth** ()=0
  Returns the depth of the branch and bound tree.
- virtual bool **isLeaf** ()=0
  Returns true if the node is a leaf (solution), otherise false.
- virtual size_t **getChildNodesNum** ()=0
  Returns the number of child-nodes.
- virtual void **branch** (size_t childIndex)=0
  Branches the node to one or more child-nodes.
- virtual size_t **calcBounds** ()=0
  Calculates the upper and lower bounds.
- virtual void **printSolution** ()=0
  Prints the solution.

4.2.2 Detailed Description

4.2.2.1 template<class ValueType>

4.2.2.2 class node< ValueType >

The basic tree-node class.

A pure virtual class that describes the basic template for nodes implementing the branch and bound algorihtm tree.

4.2.2.2.1 Template Parameters:

ValueType	The numeric type of the upper and lower bounds.

Definition at line 12 of file node.hpp.

4.2.3 Member Function Documentation

4.2.3.1 template<class ValueType > virtual void node< ValueType >::branch (size_t childIndex) [pure virtual]

Branches the node to one or more child-nodes.

Implemented in **node_tspe< ValueType >** (*p.79*), **node_tspv< ValueType >** (*p.82*), and **node_dkp< ValueType >** (*p.74*).

4.2.3.2 template<class ValueType > virtual size_t node< ValueType >::calcBounds () [pure virtual]

Calculates the upper and lower bounds.

Implemented in **node_tspe< ValueType >** (*p.79*), **node_tspv< ValueType >** (*p.83*), and **node_dkp< ValueType >** (*p.75*).

4.2.3.3 template<class ValueType > virtual size_t node< ValueType >::getChildNodesNum () [pure virtual]

Returns the number of child-nodes.

Implemented in **node_tspe< ValueType >** (*p.79*), **node_tspv< ValueType >** (*p.83*), and **node_dkp< ValueType >** (*p.75*).

4.2.3.4 template<class ValueType > virtual size_t node< ValueType >::getDepth () [pure virtual]

Returns the depth of the branch and bound tree.

Implemented in **node_tspe< ValueType >** (*p.79*), **node_tspv< ValueType >** (*p.83*), and **node_dkp< ValueType >** (*p.75*).

4.2.3.5 template<class ValueType > virtual ValueType node< ValueType >::getLowerBound () [pure virtual]

Returns the lower bound value.

Implemented in **node_tspe< ValueType >** (*p.79*), **node_tspv< ValueType >** (*p.83*), and **node_dkp< ValueType >** (*p.75*).

4.2.3.6 template<class ValueType > virtual ValueType node< ValueType >::getUpperBound () [pure virtual]

Returns the upper bound value.

Implemented in **node_tspe< ValueType >** (*p.79*), **node_tspv< ValueType >** (*p.83*), and **node_dkp< ValueType >** (*p.75*).

4.2.3.7 template<class ValueType > virtual bool node< ValueType >::isLeaf () [pure virtual]

Returns true if the node is a leaf (solution), otherise false.

Implemented in **node_tspe< ValueType >** (*p.79*), **node_tspv< ValueType >** (*p.83*), and **node_dkp< ValueType >** (*p.75*).

4.2.3.8 template<class ValueType > virtual void node< ValueType >::printSolution () [pure virtual]

Prints the solution.

Implemented in **node_tspe< ValueType >** (*p.80*), **node_dkp< ValueType >** (*p.76*), and **node_tspv< ValueType >** (*p.84*).

4.2.3.9 The documentation for this class was generated from the following file:

- **node.hpp**

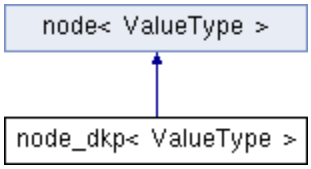
4.2.3.10

4.3 node_dkp< ValueType > Class Template Reference

The tree-node class for the DKP.

`#include <node_dkp.hpp>`

Inheritance diagram for node_dkp< ValueType >:



4.3.1 Public Member Functions

- **node_dkp** ()
  The default constructor.
- **node_dkp** (std::shared_ptr< **problem_dkp**< ValueType >> **problem**)
  The main constructor.
- **node_dkp** (const **node_dkp** &**node**)
  The copy constructor.
- **~node_dkp** ()
  The destructor.
- **node_dkp** & **operator=** (const **node_dkp** &**node**)
  The assignment operator.
- ValueType **getUpperBound** ()
  Returns the upper bound value.
- ValueType **getLowerBound** ()
  Returns the lower bound value.
- size_t **getDepth** ()
  Returns the depth of the branch and bound tree.
- bool **isLeaf** ()
  Returns true if the node is a leaf (solution), otherise false.
- size_t **getChildNodesNum** ()
  Returns the number of child-nodes.
- void **branch** (size_t childIndex)
  Branches the node to one or more child-nodes.
- size_t **calcBounds** ()
  Calculates the upper and lower bounds.
- void **printSolution** ()
  Prints the solution.

4.3.2 Detailed Description

4.3.2.1 template<class ValueType>

4.3.2.2 class node_dkp< ValueType >

The tree-node class for the DKP.

Implements the basic tree-node class "node" for the discrete knapsack problem.

4.3.2.2.1 Template Parameters:

ValueType	The numeric type of the upper and lower bounds.

Definition at line 13 of file node_dkp.hpp.

4.3.3 Constructor & Destructor Documentation

4.3.3.1 template<class ValueType> node_dkp< ValueType >::node_dkp () [inline]

The default constructor.

Definition at line 52 of file node_dkp.hpp.

4.3.3.2 template<class ValueType> node_dkp< ValueType >::node_dkp (std::shared_ptr< problem_dkp< ValueType >> problem) [inline]

The main constructor.

4.3.3.2.1 Parameters:

problem	Pointer to the DKP problem.

Definition at line 67 of file node_dkp.hpp.

4.3.3.3 template<class ValueType> node_dkp< ValueType >::node_dkp (const node_dkp< ValueType > & node) [inline]

The copy constructor.

4.3.3.3.1 Parameters:

node	The tree-node object to be copied.

Definition at line 83 of file node_dkp.hpp.

4.3.3.4 template<class ValueType> node_dkp< ValueType >::~node_dkp () [inline]

The destructor.

Definition at line 98 of file node_dkp.hpp.

4.3.4 Member Function Documentation

4.3.4.1 template<class ValueType> void node_dkp< ValueType >::branch (size_t childIndex) [inline], [virtual]

Branches the node to one or more child-nodes.

Implements **node< ValueType >** (*p.71*).

Definition at line 153 of file node_dkp.hpp.

4.3.4.2 template<class ValueType> size_t node_dkp< ValueType >::calcBounds () [inline], [virtual]

Calculates the upper and lower bounds.

Implements **node< ValueType >** (*p.71*).

Definition at line 170 of file node_dkp.hpp.

4.3.4.3 template<class ValueType> size_t node_dkp< ValueType >::getChildNodesNum () [inline], [virtual]

Returns the number of child-nodes.

Implements **node< ValueType >** (*p.71*).

Definition at line 145 of file node_dkp.hpp.

4.3.4.4 template<class ValueType> size_t node_dkp< ValueType >::getDepth () [inline], [virtual]

Returns the depth of the branch and bound tree.

Implements **node< ValueType >** (*p.71*).

Definition at line 134 of file node_dkp.hpp.

4.3.4.5 template<class ValueType> ValueType node_dkp< ValueType >::getLowerBound () [inline], [virtual]

Returns the lower bound value.

Implements **node< ValueType >** (*p.71*).

Definition at line 128 of file node_dkp.hpp.

4.3.4.6 template<class ValueType> ValueType node_dkp< ValueType >::getUpperBound () [inline], [virtual]

Returns the upper bound value.

Implements **node< ValueType >** (*p.71*).

Definition at line 123 of file node_dkp.hpp.

4.3.4.7 template<class ValueType> bool node_dkp< ValueType >::isLeaf () [inline], [virtual]

Returns true if the node is a leaf (solution), otherise false.

Implements **node< ValueType >** (*p.71*).

Definition at line 139 of file node_dkp.hpp.

4.3.4.8 template<class ValueType> node_dkp& node_dkp< ValueType >::operator= (const node_dkp< ValueType > & node) [inline]

The assignment operator.

4.3.4.8.1 Parameters:

node	The tree-node object to be used in the assignment.

Definition at line 104 of file node_dkp.hpp.

4.3.4.9 *template<class ValueType> void node_dkp< ValueType >::printSolution () [inline], [virtual]*

Prints the solution.

Implements **node< ValueType >** (*p.72*).

Definition at line 249 of file node_dkp.hpp.

4.3.4.10 *The documentation for this class was generated from the following file:*

- **node_dkp.hpp**

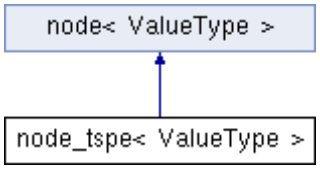
4.3.4.11

4.4 node_tspe< ValueType > Class Template Reference

The tree-node class for the TSP.

`#include <node_tspe.hpp>`

Inheritance diagram for node_tspe< ValueType >:



4.4.1 Public Member Functions

- **node_tspe** ()
  The default constructor.
- **node_tspe** (std::shared_ptr< **problem_tspv**< ValueType >> **problem**)
  The main constructor.
- **node_tspe** (const **node_tspe** &**node**)
  The copy constructor.
- **~node_tspe** ()
  The destructor.
- **node_tspe** & **operator=** (const **node_tspe** &**node**)
  The assignment operator.
- ValueType **getUpperBound** ()
  Returns the upper bound value.
- ValueType **getLowerBound** ()
  Returns the lower bound value.
- size_t **getDepth** ()
  Returns the depth of the branch and bound tree.
- bool **isLeaf** ()
  Returns true if the node is a leaf, otherwise false.
- size_t **getChildNodesNum** ()
  Returns the number of child-nodes.
- void **branch** (size_t childIndex)
  Branches to a child-node.
- size_t **calcBounds** ()
  Calculates the bounds.
- void **printSolution** ()
  Prints the solution.

4.4.2 Detailed Description

4.4.2.1 template<class ValueType>

4.4.2.2 class node_tspe< ValueType >

The tree-node class for the TSP.

Implements the basic tree-node class "node" for the traveling salesman problem. The branch and bound tree is binary.

4.4.2.2.1 Template Parameters:

ValueType	The numeric type of the upper and lower bounds.

Definition at line 14 of file node_tspe.hpp.

4.4.3 Constructor & Destructor Documentation

4.4.3.1 template<class ValueType> node_tspe< ValueType >::node_tspe () [inline]

The default constructor.

Definition at line 268 of file node_tspe.hpp.

4.4.3.2 template<class ValueType> node_tspe< ValueType >::node_tspe (std::shared_ptr< problem_tspv< ValueType >> problem) [inline]

The main constructor.

4.4.3.2.1 Parameters:

problem	Pointer to the problem object.

Definition at line 283 of file node_tspe.hpp.

4.4.3.3 template<class ValueType> node_tspe< ValueType >::node_tspe (const node_tspe< ValueType > & node) [inline]

The copy constructor.

4.4.3.3.1 Parameters:

node	The node object to be copied.

Definition at line 301 of file node_tspe.hpp.

4.4.3.4 template<class ValueType> node_tspe< ValueType >::~node_tspe () [inline]

The destructor.

Definition at line 316 of file node_tspe.hpp.

4.4.4 Member Function Documentation

4.4.4.1 template<class ValueType> void node_tspe< ValueType >::branch (size_t childIndex) [inline], [virtual]

Branches to a child-node.

4.4.4.1.1 Parameters:

childIndex	The index of the child-node.

Implements **node< ValueType >** (*p.71*).

Definition at line 371 of file node_tspe.hpp.

4.4.4.2 template<class ValueType> size_t node_tspe< ValueType >::calcBounds () [inline], [virtual]

Calculates the bounds.

Implements **node< ValueType >** (*p.71*).

Definition at line 392 of file node_tspe.hpp.

4.4.4.3 template<class ValueType> size_t node_tspe< ValueType >::getChildNodesNum () [inline], [virtual]

Returns the number of child-nodes.

Implements **node< ValueType >** (*p.71*).

Definition at line 363 of file node_tspe.hpp.

4.4.4.4 template<class ValueType> size_t node_tspe< ValueType >::getDepth () [inline], [virtual]

Returns the depth of the branch and bound tree.

Implements **node< ValueType >** (*p.71*).

Definition at line 352 of file node_tspe.hpp.

4.4.4.5 template<class ValueType> ValueType node_tspe< ValueType >::getLowerBound () [inline], [virtual]

Returns the lower bound value.

Implements **node< ValueType >** (*p.71*).

Definition at line 346 of file node_tspe.hpp.

4.4.4.6 template<class ValueType> ValueType node_tspe< ValueType >::getUpperBound () [inline], [virtual]

Returns the upper bound value.

Implements **node< ValueType >** (*p.71*).

Definition at line 341 of file node_tspe.hpp.

4.4.4.7 template<class ValueType> bool node_tspe< ValueType >::isLeaf () [inline], [virtual]

Returns true if the node is a leaf, otherwise false.

Implements **node< ValueType >** (*p.71*).

Definition at line 357 of file node_tspe.hpp.

4.4.4.8 *template<class ValueType> node_tspe& node_tspe< ValueType >::operator= (const node_tspe< ValueType > & node) [inline]*

The assignment operator.

4.4.4.8.1 Parameters:

node	The rhs of the operator.

Definition at line 322 of file node_tspe.hpp.

4.4.4.9 *template<class ValueType> void node_tspe< ValueType >::printSolution () [inline], [virtual]*

Prints the solution.

Implements **node< ValueType >** (*p.72*).

Definition at line 409 of file node_tspe.hpp.

4.4.4.10 *The documentation for this class was generated from the following file:*

- **node_tspe.hpp**

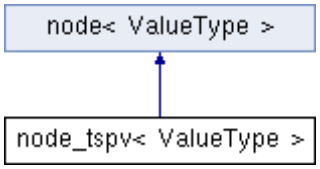
4.4.4.11

4.5 node_tspv< ValueType > Class Template Reference

The tree-node class for the TSP.

`#include <node_tspv.hpp>`

Inheritance diagram for node_tspv< ValueType >:



4.5.1 Public Member Functions

- **node_tspv** ()
  The default constructor.
- **node_tspv** (std::shared_ptr< **problem_tspv**< ValueType >> **problem**)
  The main constructor.
- **node_tspv** (const **node_tspv** &**node**)
  The copy constructor.
- **~node_tspv** ()
  The destructor.
- **node_tspv** & **operator=** (const **node_tspv** &**node**)
  The assignment operator.
- ValueType **getUpperBound** ()
  Returns the upper bound value.
- ValueType **getLowerBound** ()
  Returns the lower bound value.
- size_t **getDepth** ()
  Returns the depth of the branch and bound tree.
- bool **isLeaf** ()
  Returns true if the node is a leaf, otherwise false.
- size_t **getChildNodesNum** ()
  Returns the number of child-nodes.
- void **branch** (size_t childIndex)
  Branches to a child-node.
- size_t **calcBounds** ()
  Calculates the bounds.
- void **printSolution** ()
  Prints the solution.

4.5.2 Detailed Description

4.5.2.1 template<class ValueType>

4.5.2.2 class node_tspv< ValueType >

The tree-node class for the TSP.

Implements the basic tree-node class "node" for the traveling salesman problem. The branch and bound tree is of degree higher than two.

4.5.2.2.1 Template Parameters:

ValueType	The numeric type of the upper and lower bounds.

Definition at line 14 of file node_tspv.hpp.

4.5.3 Constructor & Destructor Documentation

4.5.3.1 template<class ValueType> node_tspv< ValueType >::node_tspv () [inline]

The default constructor.

Definition at line 88 of file node_tspv.hpp.

4.5.3.2 template<class ValueType> node_tspv< ValueType >::node_tspv (std::shared_ptr< problem_tspv< ValueType >> problem) [inline]

The main constructor.

Definition at line 100 of file node_tspv.hpp.

4.5.3.3 template<class ValueType> node_tspv< ValueType >::node_tspv (const node_tspv< ValueType > & node) [inline]

The copy constructor.

4.5.3.3.1 Parameters:

node	The node object to be copied.

Definition at line 114 of file node_tspv.hpp.

4.5.3.4 template<class ValueType> node_tspv< ValueType >::~node_tspv () [inline]

The destructor.

Definition at line 126 of file node_tspv.hpp.

4.5.4 Member Function Documentation

4.5.4.1 template<class ValueType> void node_tspv< ValueType >::branch (size_t childIndex) [inline], [virtual]

Branches to a child-node.

4.5.4.1.1 Parameters:

childIndex	The index of the child-node.

Implements **node< ValueType >** (*p.71*).

Definition at line 175 of file node_tspv.hpp.

4.5.4.2 template<class ValueType> size_t node_tspv< ValueType >::calcBounds () [inline], [virtual]

Calculates the bounds.

Implements **node< ValueType >** (*p.71*).

Definition at line 196 of file node_tspv.hpp.

4.5.4.3 template<class ValueType> size_t node_tspv< ValueType >::getChildNodesNum () [inline], [virtual]

Returns the number of child-nodes.

Implements **node< ValueType >** (*p.71*).

Definition at line 170 of file node_tspv.hpp.

4.5.4.4 template<class ValueType> size_t node_tspv< ValueType >::getDepth () [inline], [virtual]

Returns the depth of the branch and bound tree.

Implements **node< ValueType >** (*p.71*).

Definition at line 159 of file node_tspv.hpp.

4.5.4.5 template<class ValueType> ValueType node_tspv< ValueType >::getLowerBound () [inline], [virtual]

Returns the lower bound value.

Implements **node< ValueType >** (*p.71*).

Definition at line 153 of file node_tspv.hpp.

4.5.4.6 template<class ValueType> ValueType node_tspv< ValueType >::getUpperBound () [inline], [virtual]

Returns the upper bound value.

Implements **node< ValueType >** (*p.71*).

Definition at line 148 of file node_tspv.hpp.

4.5.4.7 template<class ValueType> bool node_tspv< ValueType >::isLeaf () [inline], [virtual]

Returns true if the node is a leaf, otherwise false.

Implements **node< ValueType >** (*p.71*).

Definition at line 164 of file node_tspv.hpp.

4.5.4.8 template<class ValueType> node_tspv& node_tspv< ValueType >::operator= (const node_tspv< ValueType > & node) [inline]

The assignment operator.

4.5.4.8.1 Parameters:

node	The rhs of the operator.

Definition at line 132 of file node_tspv.hpp.

4.5.4.9 *template<class ValueType> void node_tspv< ValueType >::printSolution () [inline], [virtual]*

Prints the solution.

Implements **node< ValueType >** (*p.72*).

Definition at line 221 of file node_tspv.hpp.

4.5.4.10 *The documentation for this class was generated from the following file:*

- **node_tspv.hpp**

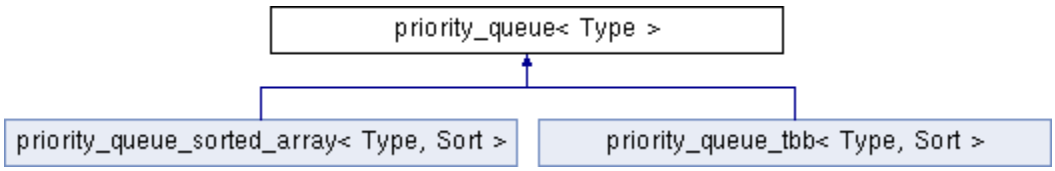
4.5.4.11

4.6 priority_queue< Type > Class Template Reference

The basic priority queue class.

`#include <priority_queue.hpp>`

Inheritance diagram for priority_queue< Type >:



4.6.1 Public Member Functions

- virtual void **push** (Type element)=0
  The push operation. Must be thread-safe.
- virtual bool **try_pop** (Type &element)=0
  The try-pop operation. Must be thread-safe.
- virtual bool **empty** ()=0
  Returns true if the queue is empty, otherwise false.
- virtual size_t **size** ()=0
  Returns the number of the elements in the queue.

4.6.2 Detailed Description

4.6.2.1 template<class Type>

4.6.2.2 class priority_queue< Type >

The basic priority queue class.

A pure virtual class that describes the basic template for a concurrent priority queue in order to store the tree-nodes (partial solutions) of the branch and bound algorithm tree.

4.6.2.2.1 Template Parameters:

Type	The type of the elements to be placed in the queue.

Definition at line 13 of file priority_queue.hpp.

4.6.3 Member Function Documentation

4.6.3.1 template<class Type> virtual bool priority_queue< Type >::empty () [pure virtual]

Returns true if the queue is empty, otherwise false.

Implemented in **priority_queue_sorted_array< Type, Sort >** (*p.88*), and **priority_queue_tbb< Type, Sort >** (*p.90*).

4.6.3.2 template<class Type> virtual void priority_queue< Type >::push (Type element) [pure virtual]

The push operation. Must be thread-safe.

Implemented in **priority_queue_sorted_array< Type, Sort >** (*p.88*), and **priority_queue_tbb< Type, Sort >** (*p.90*).

4.6.3.3 template<class Type> virtual size_t priority_queue< Type >::size () [pure virtual]

Returns the number of the elements in the queue.

Implemented in **priority_queue_sorted_array< Type, Sort >** (*p.88*), and **priority_queue_tbb< Type, Sort >** (*p.90*).

4.6.3.4 template<class Type> virtual bool priority_queue< Type >::try_pop (Type & element) [pure virtual]

The try-pop operation. Must be thread-safe.

Implemented in **priority_queue_sorted_array< Type, Sort >** (*p.88*), and **priority_queue_tbb< Type, Sort >** (*p.91*).

4.6.3.5 The documentation for this class was generated from the following file:

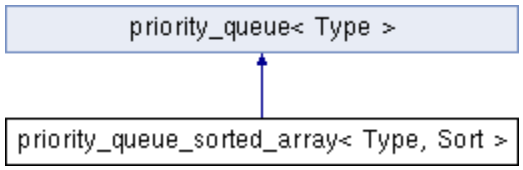
- **priority_queue.hpp**

4.6.3.6

4.7 priority_queue_sorted_array< Type, Sort > Class Template Reference

`#include <priority_queue_sorted_array.hpp>`

Inheritance diagram for priority_queue_sorted_array< Type, Sort >:



4.7.1 Public Member Functions

- **priority_queue_sorted_array** ()
- **priority_queue_sorted_array** (const **priority_queue_sorted_array** &**sortedArray**)
- **~priority_queue_sorted_array** ()
- void **push** (Type element)
  The push operation. Must be thread-safe.
- bool **try_pop** (Type &element)
  The try-pop operation. Must be thread-safe.
- bool **empty** ()
  Returns true if the queue is empty, otherwise false.
- size_t **size** ()
  Returns the number of the elements in the queue.

4.7.2 Detailed Description

4.7.2.1 template<class Type, class Sort>

4.7.2.2 class priority_queue_sorted_array< Type, Sort >

Definition at line 13 of file priority_queue_sorted_array.hpp.

4.7.3 Constructor & Destructor Documentation

4.7.3.1 template<class Type , class Sort > priority_queue_sorted_array< Type, Sort >::priority_queue_sorted_array () [inline]

Definition at line 25 of file priority_queue_sorted_array.hpp.

4.7.3.2 template<class Type , class Sort > priority_queue_sorted_array< Type, Sort >::priority_queue_sorted_array (const priority_queue_sorted_array< Type, Sort > & sortedArray) [inline]

Definition at line 31 of file priority_queue_sorted_array.hpp.

4.7.3.3 template<class Type , class Sort > priority_queue_sorted_array< Type, Sort >::~priority_queue_sorted_array () [inline]

Definition at line 37 of file priority_queue_sorted_array.hpp.

4.7.4 Member Function Documentation

4.7.4.1 template<class Type , class Sort > bool priority_queue_sorted_array< Type, Sort >::empty () [inline], [virtual]

Returns true if the queue is empty, otherwise false.

Implements **priority_queue< Type >** (*p.85*).

Definition at line 69 of file priority_queue_sorted_array.hpp.

4.7.4.2 template<class Type , class Sort > void priority_queue_sorted_array< Type, Sort >::push (Type element) [inline], [virtual]

The push operation. Must be thread-safe.

Implements **priority_queue< Type >** (*p.85*).

Definition at line 43 of file priority_queue_sorted_array.hpp.

4.7.4.3 template<class Type , class Sort > size_t priority_queue_sorted_array< Type, Sort >::size () [inline], [virtual]

Returns the number of the elements in the queue.

Implements **priority_queue< Type >** (*p.86*).

Definition at line 74 of file priority_queue_sorted_array.hpp.

4.7.4.4 template<class Type , class Sort > bool priority_queue_sorted_array< Type, Sort >::try_pop (Type & element) [inline], [virtual]

The try-pop operation. Must be thread-safe.

Implements **priority_queue< Type >** (*p.86*).

Definition at line 51 of file priority_queue_sorted_array.hpp.

4.7.4.5 The documentation for this class was generated from the following file:

- **priority_queue_sorted_array.hpp**

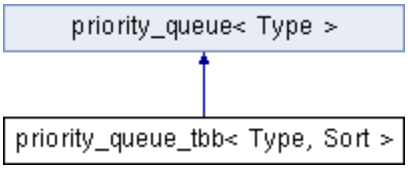
4.7.4.6

4.8 priority_queue_tbb< Type, Sort > Class Template Reference

Implements the basic priority queue class using tbb::concurrent_priority_queue.

`#include <priority_queue_tbb.hpp>`

Inheritance diagram for priority_queue_tbb< Type, Sort >:



4.8.1 Public Member Functions

- **priority_queue_tbb** ()
  The default constructor.
- **priority_queue_tbb** (const **priority_queue_tbb** &queue)
  The copy constructor.
- **~priority_queue_tbb** ()
  The destructor.
- void **push** (Type element)
  The push operation.
- bool **try_pop** (Type &element)
  The try-pop operation.
- Type **top** ()
  The top operation.
- bool **empty** ()
  Returns true if the queue is empty, otherwise false.
- size_t **size** ()
  Returns the number of elements in the queue.

4.8.2 Detailed Description

4.8.2.1 template<class Type, class Sort>

4.8.2.2 class priority_queue_tbb< Type, Sort >

Implements the basic priority queue class using tbb::concurrent_priority_queue.

4.8.2.2.1 Template Parameters:

Type	The type of the elements to be placed in the queue.
Sort	A class having the same functionality as std::less for the Type class.

Definition at line 12 of file priority_queue_tbb.hpp.

4.8.3 Constructor & Destructor Documentation

4.8.3.1 template<class Type , class Sort > priority_queue_tbb< Type, Sort >::priority_queue_tbb () [inline]

The default constructor.

Definition at line 22 of file priority_queue_tbb.hpp.

4.8.3.2 template<class Type , class Sort > priority_queue_tbb< Type, Sort >::priority_queue_tbb (const priority_queue_tbb< Type, Sort > & queue) [inline]

The copy constructor.

4.8.3.2.1 Parameters:

queue	The queue to be copied.

Definition at line 28 of file priority_queue_tbb.hpp.

4.8.3.3 template<class Type , class Sort > priority_queue_tbb< Type, Sort >::~priority_queue_tbb () [inline]

The destructor.

Definition at line 34 of file priority_queue_tbb.hpp.

4.8.4 Member Function Documentation

4.8.4.1 template<class Type , class Sort > bool priority_queue_tbb< Type, Sort >::empty () [inline], [virtual]

Returns true if the queue is empty, otherwise false.

Implements **priority_queue< Type >** (*p.85*).

Definition at line 56 of file priority_queue_tbb.hpp.

4.8.4.2 template<class Type , class Sort > void priority_queue_tbb< Type, Sort >::push (Type element) [inline], [virtual]

The push operation.

Implements **priority_queue< Type >** (*p.85*).

Definition at line 40 of file priority_queue_tbb.hpp.

4.8.4.3 template<class Type , class Sort > size_t priority_queue_tbb< Type, Sort >::size () [inline], [virtual]

Returns the number of elements in the queue.

Implements **priority_queue< Type >** (*p.86*).

Definition at line 61 of file priority_queue_tbb.hpp.

4.8.4.4 template<class Type , class Sort > Type priority_queue_tbb< Type, Sort >::top () [inline]

The top operation.

Definition at line 50 of file priority_queue_tbb.hpp.

4.8.4.5 template<class Type , class Sort > bool priority_queue_tbb< Type, Sort >::try_pop (Type & element) [inline], [virtual]

The try-pop operation.

Implements **priority_queue< Type >** (*p.86*).

Definition at line 45 of file priority_queue_tbb.hpp.

4.8.4.6 The documentation for this class was generated from the following file:

- **priority_queue_tbb.hpp**

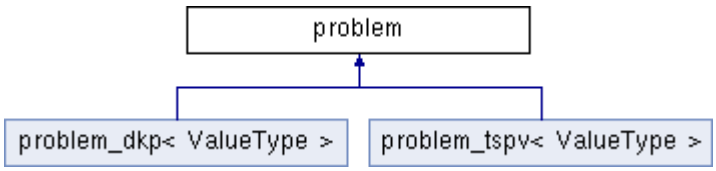
4.8.4.7

4.9 problem Class Reference

An abstract class for the description of problems solved by the B&B method.

`#include <problem.hpp>`

Inheritance diagram for problem:



4.9.1 Detailed Description

An abstract class for the description of problems solved by the B&B method.
Definition at line 7 of file problem.hpp.

The documentation for this class was generated from the following file:

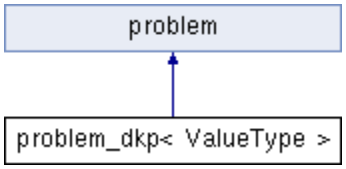
- **problem.hpp**

4.10 problem_dkp< ValueType > Class Template Reference

Implements the description of the discrete knapsack problem.

`#include <problem_dkp.hpp>`

Inheritance diagram for problem_dkp< ValueType >:



4.10.1 Public Member Functions

- **problem_dkp** ()
  The default constructor.
- **problem_dkp** (size_t variableSize)
  The main constructor.
- **problem_dkp** (const **problem_dkp** &**problem**)
  The copy constructor.
- **~problem_dkp** ()
  The destructor.
- void **initProblem** (size_t variableSize)
  Initializes the objective and constraint functions factors.
- void **setObjFuncFactor** (size_t elementIndex, ValueType factor)
  Sets the objective function factor for a specific item.
- void **setConstFuncFactor** (size_t elementIndex, ValueType factor)
  Set the constraint function factor for a specific item.
- void **setConstraint** (ValueType constraint)
  Set the constraint of the problem.
- void **sortItems** ()
  Sorts the items by "value by weight".
- ValueType **getObjFuncFactor** (size_t elementIndex)
  Returns the objective function factor for a specific item.
- ValueType **getConstFuncFactor** (size_t elementIndex)
  Returns the constraint function factor for a specific item.
- size_t **getVariableSize** ()
  Returns the size of the problem, the number of items.
- ValueType **getConstraint** ()
  Returns the constraint value.

4.10.2 Detailed Description

4.10.2.1 template<class ValueType>

4.10.2.2 class problem_dkp< ValueType >

Implements the description of the discrete knapsack problem.

4.10.2.2.1 Template Parameters:

ValueType	The numeric type of the objective and constraint functions factors.

Definition at line 11 of file problem_dkp.hpp.

4.10.3 Constructor & Destructor Documentation

4.10.3.1 template<class ValueType > problem_dkp< ValueType >::problem_dkp () [inline]

The default constructor.

Definition at line 25 of file problem_dkp.hpp.

4.10.3.2 template<class ValueType > problem_dkp< ValueType >::problem_dkp (size_t variableSize) [inline]

The main constructor.

4.10.3.2.1 Parameters:

variableSize	The size of the problem, the number of items.

Definition at line 31 of file problem_dkp.hpp.

4.10.3.3 template<class ValueType > problem_dkp< ValueType >::problem_dkp (const problem_dkp< ValueType > & problem) [inline]

The copy constructor.

4.10.3.3.1 Parameters:

problem	The problem object to be copied.

Definition at line 37 of file problem_dkp.hpp.

4.10.3.4 template<class ValueType > problem_dkp< ValueType >::~problem_dkp () [inline]

The destructor.

Definition at line 43 of file problem_dkp.hpp.

4.10.4 Member Function Documentation

4.10.4.1 template<class ValueType > ValueType problem_dkp< ValueType >::getConstFuncFactor (size_t elementIndex) [inline]

Returns the constraint function factor for a specific item.

4.10.4.1.1 Parameters:

elementIndex	The index of the item.

Definition at line 109 of file problem_dkp.hpp.

4.10.4.2 *template<class ValueType > ValueType problem_dkp< ValueType >::getConstraint () [inline]*

Returns the constraint value.

Definition at line 120 of file problem_dkp.hpp.

4.10.4.3 *template<class ValueType > ValueType problem_dkp< ValueType >::getObjFuncFactor (size_t* elementIndex*) [inline]*

Returns the objective function factor for a specific item.

4.10.4.3.1 Parameters:

elementIndex	The index of the item.

Definition at line 103 of file problem_dkp.hpp.

4.10.4.4 *template<class ValueType > size_t problem_dkp< ValueType >::getVariableSize () [inline]*

Returns the size of the problem, the number of items.

Definition at line 115 of file problem_dkp.hpp.

4.10.4.5 *template<class ValueType > void problem_dkp< ValueType >::initProblem (size_t* variableSize*) [inline]*

Initializes the objective and constraint functions factors.

4.10.4.5.1 Parameters:

variableSize	The size of the problem, the number of items.

Definition at line 49 of file problem_dkp.hpp.

4.10.4.6 *template<class ValueType > void problem_dkp< ValueType >::setConstFuncFactor (size_t* elementIndex*, ValueType* factor*) [inline]*

Set the constraint function factor for a specific item.

4.10.4.6.1 Parameters:

elementIndex	The index of the item.
factor	The factor value.

Definition at line 62 of file problem_dkp.hpp.

4.10.4.7 template<class ValueType > void problem_dkp< ValueType >::setConstraint (ValueType constraint) [inline]

Set the constraint of the problem.

4.10.4.7.1 Parameters:

constraint	The constraint value.

Definition at line 68 of file problem_dkp.hpp.

4.10.4.8 template<class ValueType > void problem_dkp< ValueType >::setObjFuncFactor (size_t elementIndex, ValueType factor) [inline]

Sets the objective function factor for a specific item.

4.10.4.8.1 Parameters:

elementIndex	The index of the item.
factor	The factor value.

Definition at line 56 of file problem_dkp.hpp.

4.10.4.9 template<class ValueType > void problem_dkp< ValueType >::sortItems () [inline]

Sorts the items by "value by weight".

Definition at line 74 of file problem_dkp.hpp.

4.10.4.10 The documentation for this class was generated from the following file:

- **problem_dkp.hpp**

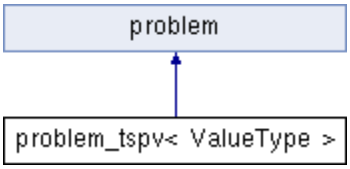
4.10.4.11

4.11 problem_tspv< ValueType > Class Template Reference

Implements the description of the travelling salesman problem.

`#include <problem_tspv.hpp>`

Inheritance diagram for problem_tspv< ValueType >:



4.11.1 Public Member Functions

- **problem_tspv** ()
- **problem_tspv** (size_t verticesNum)
  The main constructor.
- **problem_tspv** (const **problem_tspv** &**problem**)
  The copy constructor.
- **~problem_tspv** ()
  The destructor.
- void **initProblem** (size_t verticesNum)
  Initializes the problem.
- void **setEdgeWeight** (size_t source, size_t dest, ValueType weight)
  Sets the weight of an edge.
- ValueType **getEdgeWeight** (size_t source, size_t dest)
  Returns the weight of an edge.
- size_t **getEdgeDest** (size_t source, size_t index)
  Returns the i-th closest destination vertex to a source vertex.
- ValueType **getShortestWeight** (size_t source)
  Returns the weight of the shortest edge starting from a source vertex.
- size_t **getVerticesNum** ()
  Returns the number of vertices.

4.11.2 Detailed Description

4.11.2.1 template<class ValueType>

4.11.2.2 class problem_tspv< ValueType >

Implements the description of the travelling salesman problem.

4.11.2.2.1 Template Parameters:

ValueType	The numeric type of the objective and constraint functions factors.

Definition at line 11 of file problem_tspv.hpp.

4.11.3 Constructor & Destructor Documentation

4.11.3.1 template<class ValueType > problem_tspv< ValueType >::problem_tspv () [inline]

Definition at line 23 of file problem_tspv.hpp.

4.11.3.2 template<class ValueType > problem_tspv< ValueType >::problem_tspv (size_t verticesNum) [inline]

The main constructor.

4.11.3.2.1 Parameters:

verticesNum	The number of vertices in the graph.

Definition at line 29 of file problem_tspv.hpp.

4.11.3.3 template<class ValueType > problem_tspv< ValueType >::problem_tspv (const problem_tspv< ValueType > & problem) [inline]

The copy constructor.

4.11.3.3.1 Parameters:

problem	The problem object to be copied.

Definition at line 40 of file problem_tspv.hpp.

4.11.3.4 template<class ValueType > problem_tspv< ValueType >::~problem_tspv () [inline]

The destructor.
Definition at line 46 of file problem_tspv.hpp.

4.11.4 Member Function Documentation

4.11.4.1 template<class ValueType > size_t problem_tspv< ValueType >::getEdgeDest (size_t source, size_t index) [inline]

Returns the i-th closest destination vertex to a source vertex.

4.11.4.1.1 Parameters:

source	The source vertex.
index	The index.

Definition at line 83 of file problem_tspv.hpp.

4.11.4.2 template<class ValueType > ValueType problem_tspv< ValueType >::getEdgeWeight (size_t source, size_t dest) [inline]

Returns the weight of an edge.

4.11.4.2.1 Parameters:

source	The source vertex.
dest	The destination vertex.

Definition at line 77 of file problem_tspv.hpp.

4.11.4.3 template<class ValueType > ValueType problem_tspv< ValueType >::getShortestWeight (size_t source) [inline]

Returns the weight of the shortest edge starting from a source vertex.

4.11.4.3.1 Parameters:

source	The source vertex.

Definition at line 89 of file problem_tspv.hpp.

4.11.4.4 template<class ValueType > size_t problem_tspv< ValueType >::getVerticesNum () [inline]

Returns the number of vertices.

Definition at line 95 of file problem_tspv.hpp.

4.11.4.5 template<class ValueType > void problem_tspv< ValueType >::initProblem (size_t verticesNum) [inline]

Initializes the problem.

4.11.4.5.1 Parameters:

verticesNum	The number of vertices in the graph.

Definition at line 52 of file problem_tspv.hpp.

4.11.4.6 template<class ValueType > void problem_tspv< ValueType >::setEdgeWeight (size_t source, size_t dest, ValueType weight) [inline]

Sets the weight of an edge.

4.11.4.6.1 Parameters:

source	The source vertex.
dest	The destination vertex.
weight	The weight of the edge.

Definition at line 65 of file problem_tspv.hpp.

4.11.4.7 The documentation for this class was generated from the following file:

- **problem_tspv.hpp**

4.11.4.8

4.12 solver< ProblemType, NodeType, ValueType > Class Template Reference

Implements the Branch and Bound method.

`#include <solver.hpp>`

4.12.1 Public Member Functions

- **solver** ()
  The default constructor.
- **solver** (std::shared_ptr< ProblemType > **problem**, tbb::concurrent_queue< NodeType * > *spareNodes)
  The main constructor.
- **~solver** ()
  The destructor.
- void **solve** (size_t threadsNum, **queues** queueType=**tbbPriorityQueue**, **sorts** sortType=**upperBound**, **strategies** parallelType=**sharedNodes**)
  Solves the problem.
- void **test** (std::ofstream &resStream, std::ofstream &statsStream, size_t threadsNum, **queues** queueType=**tbbPriorityQueue**, **sorts** sortType=**upperBound**, **strategies** parallelType=**sharedNodes**)
  Solves the problem and collects benchmark information.

4.12.2 Detailed Description

4.12.2.1 template<class ProblemType, class NodeType, class ValueType>

4.12.2.2 class solver< ProblemType, NodeType, ValueType >

Implements the Branch and Bound method.

Implements various functions that solve suitable problems with the Branch and Bound method in a shared memory system.

4.12.2.2.1 Template Parameters:

ValueType	The numberic type of the upper and lower bounds.
NodeType	The tree-node type.
Queue	The type of the elements in the priority queue.

Definition at line 41 of file solver.hpp.

4.12.3 Constructor & Destructor Documentation

4.12.3.1 template<class ProblemType, class NodeType, class ValueType> solver< ProblemType, NodeType, ValueType >::solver () [inline]

The default constructor.

Definition at line 971 of file solver.hpp.

4.12.3.2 template<class ProblemType, class NodeType, class ValueType> solver< ProblemType, NodeType, ValueType >::solver (std::shared_ptr< ProblemType > problem, tbb::concurrent_queue< NodeType * > * spareNodes) [inline]

The main constructor.

4.12.3.2.1 Parameters:

problem	Pointer to the problem object.
spareNodes	Pointer to a queue storing node objects for recycling.

Definition at line 982 of file solver.hpp.

4.12.3.3 template<class ProblemType, class NodeType, class ValueType> solver< ProblemType, NodeType, ValueType >::~solver () [inline]

The destructor.
Definition at line 997 of file solver.hpp.

4.12.4 Member Function Documentation

4.12.4.1 template<class ProblemType, class NodeType, class ValueType> void solver< ProblemType, NodeType, ValueType >::solve (size_t threadsNum, queues queueType = tbbPriorityQueue, sorts sortType = upperBound, strategies parallelType = sharedNodes) [inline]

Solves the problem.

4.12.4.1.1 Parameters:

threadsNum	Number of threads to spawn.
queueType	The priority queue implementation to be used. Default is tbb::concurrent_priority_queue.
sortType	The sorting method to be used. Default is sorting by best upper bound.
parallelType	The parallelization strategy to be used. Default is the shared queue strategy.

Definition at line 1013 of file solver.hpp.

4.12.4.2 template<class ProblemType, class NodeType, class ValueType> void solver< ProblemType, NodeType, ValueType >::test (std::ofstream & resStream, std::ofstream & statsStream, size_t threadsNum, queues queueType = tbbPriorityQueue, sorts sortType = upperBound, strategies parallelType = sharedNodes) [inline]

Solves the problem and collects benchmark information.

4.12.4.2.1 Parameters:

resStream	File stream to write the benchmark results in detail.
statsStream	File stream to write the statistics of the benchmark results.
threadsNum	Number of threads to spawn.
queueType	The priority queue implementation to be used. Default is tbb::concurrent_priority_queue.
sortType	The sorting method to be used. Default is sorting by best upper bound.
parallelType	The parallelization strategy to be used. Default is the shared queue strategy.

Definition at line 1039 of file solver.hpp.

4.12.4.3 The documentation for this class was generated from the following file:

- **solver.hpp**

4.12.4.4

4.13 sort_depth< Type > Class Template Reference

Implements std::less functionality for the tree-node classes, based on (primary) the depth and (secondary) the best upper bound.

`#include <sort.hpp>`

4.13.1 Public Member Functions

- **sort_depth** ()
  The default constructor.
- bool **operator()** (Type *lhs, Type *rhs) const
  The () operator.

4.13.2 Detailed Description

4.13.2.1 template<class Type>

4.13.2.2 class sort_depth< Type >

Implements std::less functionality for the tree-node classes, based on (primary) the depth and (secondary) the best upper bound.

Definition at line 26 of file sort.hpp.

4.13.3 Constructor & Destructor Documentation

4.13.3.1 template<class Type > sort_depth< Type >::sort_depth () [inline]

The default constructor.

Definition at line 31 of file sort.hpp.

4.13.4 Member Function Documentation

*4.13.4.1 template<class Type > bool sort_depth< Type >::operator() (Type * lhs, Type * rhs) const [inline]*

The () operator.

Definition at line 36 of file sort.hpp.

4.13.4.2 The documentation for this class was generated from the following file:

- **sort.hpp**

4.13.4.3

4.14 sort_upper_bound< Type > Class Template Reference

Implements std::less functionality for the tree-node classes, based on the best upper bound.

`#include <sort.hpp>`

4.14.1 Public Member Functions

- **sort_upper_bound** ()
  The default constructor.
- bool **operator()** (Type *lhs, Type *rhs) const
  The () operator.

4.14.2 Detailed Description

4.14.2.1 template<class Type>

4.14.2.2 class sort_upper_bound< Type >

Implements std::less functionality for the tree-node classes, based on the best upper bound.
Definition at line 8 of file sort.hpp.

4.14.3 Constructor & Destructor Documentation

4.14.3.1 template<class Type > sort_upper_bound< Type >::sort_upper_bound () [inline]

The default constructor.
Definition at line 13 of file sort.hpp.

4.14.4 Member Function Documentation

*4.14.4.1 template<class Type > bool sort_upper_bound< Type >::operator() (Type * lhs, Type * rhs) const [inline]*

The () operator.
Definition at line 18 of file sort.hpp.

4.14.4.2 The documentation for this class was generated from the following file:

- **sort.hpp**

4.14.4.3

4.15 statistics Class Reference

Calculates and handles various statistics on a set of benchmark objects.

`#include <statistics.hpp>`

4.15.1 Public Member Functions

- **statistics** ()
  The default constructor.
- void **add** (std::vector< **benchmark** > &bench)
  Adds a benchmark object to the set.
- void **calc** ()
  Calculates statistics on the set.
- void **write** (std::ofstream &fileStream)
  Writes the statistics to a file stream.

4.15.2 Detailed Description

Calculates and handles various statistics on a set of benchmark objects.
Definition at line 9 of file statistics.hpp.

4.15.3 Constructor & Destructor Documentation

4.15.3.1 statistics::statistics () [inline]

The default constructor.
Definition at line 23 of file statistics.hpp.

4.15.4 Member Function Documentation

4.15.4.1 void statistics::add (std::vector< benchmark > & bench) [inline]

Adds a benchmark object to the set.

4.15.4.1.1 Parameters:

bench	The benchmark object to be added to the set.

Definition at line 30 of file statistics.hpp.

4.15.4.2 void statistics::calc () [inline]

Calculates statistics on the set.
Definition at line 56 of file statistics.hpp.

4.15.4.3 void statistics::write (std::ofstream & fileStream) [inline]

Writes the statistics to a file stream.
Definition at line 77 of file statistics.hpp.

4.15.4.4 The documentation for this class was generated from the following file:

- **statistics.hpp**

5 File Documentation

5.1 benchmark.hpp File Reference

```
#include "stdafx.h"
```

5.1.1 Classes

- class **benchmark**

A benchmarking class.

5.2 BranchBound_DKP.cpp File Reference

```
#include "stdafx.h"
#include "node_dkp.hpp"
#include "priority_queue_tbb.hpp"
#include "problem_dkp.hpp"
#include "solver.hpp"
#include "sort.hpp"
```

5.2.1 Typedefs

- typedef boost::multiprecision::number< boost::multiprecision::cpp_int_backend< 4096, 4096, boost::multiprecision::unsigned_magnitude, boost::multiprecision::checked, void > > **checked_uint4096_t**
- typedef **checked_uint4096_t bigInt**

5.2.2 Functions

- int **getNumberOfCores** ()
- int **main** (int argc, char *argv[])

5.2.3 Typedef Documentation

5.2.3.1 typedef checked_uint4096_t bigInt

Definition at line 36 of file BranchBound_DKP.cpp.

5.2.3.2 typedef boost::multiprecision::number<boost::multiprecision::cpp_int_backend<4096, 4096, boost::multiprecision::unsigned_magnitude, boost::multiprecision::checked, void> > checked_uint4096_t

Definition at line 35 of file BranchBound_DKP.cpp.

5.2.4 Function Documentation

5.2.4.1 int getNumberOfCores ()

Definition at line 11 of file BranchBound_DKP.cpp.

*5.2.4.2 int main (int argc, char * argv[])*

Definition at line 38 of file BranchBound_DKP.cpp.

5.2.4.3

5.3 BranchBound_TSP.cpp File Reference

```
#include "stdafx.h"
#include "node_tspv.hpp"
#include "node_tspe.hpp"
#include "priority_queue_tbb.hpp"
#include "problem_tspv.hpp"
#include "solver.hpp"
#include "sort.hpp"
```

5.3.1 Functions

- template<class Type > bool **readTSP** (std::shared_ptr< **problem_tspv**< Type >> **problem**, std::string fileName)
- int **main** (int argc, char *argv[])

5.3.2 Function Documentation

5.3.2.1 int main (int argc*, char ** argv*[])*

Definition at line 94 of file BranchBound_TSP.cpp.

5.3.2.2 template<class Type > bool readTSP (std::shared_ptr< problem_tspv< Type >> problem*, std::string* fileName*)*

Definition at line 13 of file BranchBound_TSP.cpp.

5.3.2.3

5.4 node.hpp File Reference

```
#include "stdafx.h"
```

5.4.1 Classes

- class **node< ValueType >**

The basic tree-node class.

5.5 node_dkp.hpp File Reference

```
#include "stdafx.h"
#include "node.hpp"
#include "problem_dkp.hpp"
```

5.5.1 Classes

- class **node_dkp< ValueType >**

The tree-node class for the DKP.

5.6 node_tspe.hpp File Reference

```
#include "stdafx.h"
#include "node.hpp"
#include "problem_tspv.hpp"
```

5.6.1 Classes

- class **node_tspe< ValueType >**

The tree-node class for the TSP.

5.7 node_tspv.hpp File Reference

```
#include "stdafx.h"
#include "node.hpp"
#include "problem_tspv.hpp"
```

5.7.1 Classes

- class **node_tspv< ValueType >**

The tree-node class for the TSP.

5.8 priority_queue.hpp File Reference

```
#include "stdafx.h"
```

5.8.1 Classes

- class **priority_queue**< **Type** >

The basic priority queue class.

5.9 priority_queue_sorted_array.hpp File Reference

```
#include "stdafx.h"
#include "priority_queue.hpp"
```

5.9.1 Classes

- class **priority_queue_sorted_array< Type, Sort >**

5.10 priority_queue_tbb.hpp File Reference

```
#include "stdafx.h"
#include "priority_queue.hpp"
```

5.10.1 Classes

- class **priority_queue_tbb< Type, Sort >**

Implements the basic priority queue class using tbb::concurrent_priority_queue.

5.11 problem.hpp File Reference

```
#include "stdafx.h"
```

5.11.1 Classes

- class **problem**

An abstract class for the description of problems solved by the B&B method.

5.12 problem_dkp.hpp File Reference

```
#include "stdafx.h"
#include "problem.hpp"
```

5.12.1 Classes

- class **problem_dkp< ValueType >**

Implements the description of the discrete knapsack problem.

5.13 problem_tspv.hpp File Reference

```
#include "stdafx.h"
#include "problem.hpp"
```

5.13.1 Classes

- class **problem_tspv< ValueType >**

Implements the description of the travelling salesman problem.

5.14 solver.hpp File Reference

```
#include "stdafx.h"
#include "benchmark.hpp"
#include "priority_queue_sorted_array.hpp"
#include "priority_queue_tbb.hpp"
#include "sort.hpp"
#include "statistics.hpp"
```

5.14.1 Classes

- class **solver< ProblemType, NodeType, ValueType >**

5.14.2 *Implements the Branch and Bound method.* Enumerations

- enum **strategies** { **sharedNodes**, **privateNodes**, **privateNodesStealing** }*Enumaration of the different parallelization strategies.*
- enum **queues** { **sortedArray**, **tbbPriorityQueue** }
- enum **sorts** { **upperBound**, **depth** }

5.14.3 Enumeration Type Documentation

5.14.3.1 enum queues

Enumerator

sortedArray Sorted array implementation.

tbbPriorityQueue tbb::concurrent_priority_queue implementation.

Definition at line 20 of file solver.hpp.

5.14.3.2 enum sorts

Enumerator

upperBound Sorting by best upper bound.

depth Sorting by (primary) depth and (secondary) best upper bound.

Definition at line 27 of file solver.hpp.

5.14.3.3 enum strategies

Enumaration of the different parallelization strategies.

Enumerator

sharedNodes Share queue strategy.

privateNodes Private queues strategy.

privateNodesStealing Private queues with node stealing strategy.

Definition at line 12 of file solver.hpp.

5.14.3.4

5.15 sort.hpp File Reference

```
#include "stdafx.h"
```

5.15.1 Classes

- class **sort_upper_bound< Type >**
- *Implements std::less functionality for the tree-node classes, based on the best upper bound.* class **sort_depth< Type >**

Implements std::less functionality for the tree-node classes, based on (primary) the depth and (secondary) the best upper bound.

5.16 statistics.hpp File Reference

```
#include "stdafx.h"
#include "benchmark.hpp"
```

5.16.1 Classes

- class **statistics**

Calculates and handles various statistics on a set of benchmark objects.

5.17 stdafx.h File Reference

```
#include <stdio.h>
#include <algorithm>
#include <atomic>
#include <cassert>
#include <iomanip>
#include <iostream>
#include <fstream>
#include <memory>
#include <mutex>
#include <numeric>
#include <limits>
#include <random>
#include <thread>
#include <vector>
#include <boost/circular_buffer.hpp>
#include <boost/dynamic_bitset.hpp>
#include <boost/lockfree/queue.hpp>
#include <boost/multiprecision/cpp_int.hpp>
#include <tbb/tbb.h>
```

6 Index