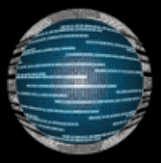


Τμήμα Μηχανικών Η/Υ και Πληροφορικής
Πανεπιστήμιο Πατρών
30 Δεκεμβρίου 2020

Παράλληλος Προγραμματισμός
3ο *Εργαστήριο*

Ανδρέας Καρατζάς

Περιεχόμενα

1

Το πρόβλημα του περιοδεύοντα πωλητή

1.1 Εισαγωγή

Για τη 3η εργαστηριακή άσκηση υλοποιήθηκαν διάφορες ενδιαφέρουσες προσεγγίσεις για την επίλυση του προβλήματος του περιοδεύοντα πωλητή. Τα στοιχεία του συστήματος στο οποίο έγιναν οι μετρήσεις παρουσιάζονται στον Πίνακα 1.

Πίνακας 1: Στοιχεία Συστήματος

Λειτουργικό Σύστημα	Windows 10 x64
Επεξεργαστής	Intel Core i7-9750H CPU @ 2.60 GHz
RAM frequency	2667 MHz
RAM size	16 GB
IDE	Visual Studio 2019
Compiler	Intel 19.1
Programming Language	C++ 17

1.2 GitHub

Η εργασία έχει ανέβει και στο GitHub. Υπάρχουν και οδηγίες εγκατάστασης, μαζί με το makefile που διευκολύνει τον εξεταστή. Συνοπτικά, η διαδικασία εγκατάστασης σε περιβάλλον Unix έχεις ως εξής:

- Άνοιγμα παραθύρου *Terminal*
- Περιήγηση σε ένα άδειο directory μέσω της εντολής `cd`, στο οποίο θα γίνει το cloning του repository
- Κλωνοποίηση του repository μέσω της εντολής
  `git clone https://github.com/andreasceid/tsp.git`
- Περιήγηση στο main directory του repository με την εντολή `cd tsp/`
- (Προαιρετικό) Τροποποίηση του project μέσω του αρχείου *Common.h*, με σκοπό την εκτέλεση κάποιου από τους 7 υλοποιημένους αλγορίθμους ή με

σκοπό μια δοκιμαστική εκτέλεση κ.λπ.

- (Προαιρετικό) Τροποποίηση του makefile του project με σκοπό την εμφάνιση πληροφοριών βελτιστοποίησης του κώδικα
- Μεταγλώττιση του project εκτελώντας την εντολή `make`
- Περιήγηση στο directory που βρίσκεται το *build* του project με την εντολή `cd build/`
- Δημιουργία του directory για την εξαγωγή των αποτελεσμάτων του build (για την περίπτωση που εκτελείται σε «test mode») εκτελώντας την εντολή `mkdir data`
- Εκτέλεση του build με την εντολή `./lab-3_tsp`

1.3 Project Configurations και η κεφαλίδα Common.h

Όλες οι ρυθμίσεις που αφορούν την εργασία μπορούν να γίνουν από την κεφαλίδα `Common.h`. Υπάρχουν ρυθμίσεις, όπως:

- Ποιός αλγόριθμος θα τρέξει
- Σε τι λειτουργία θα γίνει η εκτέλεση (test ή release)
- Το σύνολο δεδομένων θα είναι προκαθορισμένο ή τυχαίο
- Με πόσα threads θα τρέξει το project, αν ο αλγόριθμος που επιλέχθηκε είναι παράλληλος

Οι ρυθμίσεις αυτές θα πρέπει να αλλάζουν με προσοχή. Ακολουθούν μερικές συμβουλές:

- Ο αριθμός των *threads* στην περίπτωση του αλγορίθμου του *ant colony* ορίζεται από τη μεταβλητή `N_ANTS`. Επομένως, καλό είναι αυτή η μεταβλητή να μην ξεπερνάει το τριπλάσιο των physical cores του συστήματος στο οποίο γίνεται η εκτέλεση. Επίσης, είναι επιθυμητό ο αριθμός αυτός να διαιρείται τέλεια με τον αριθμό των physical cores έτσι ώστε να γίνεται καλύτερη εκμετάλλευση των πόρων του συστήματος
- Ανάλογα με τη ρύθμιση της μεταβλητής `N_ANTS`, θα πρέπει να προσαρμοστούν και οι μεταβλητές `ANT_MEMORY` και `ROULETTE_SIZE`. Ως προς τη μεταβλητή `ANT_MEMORY`, θα πρέπει να ισχύει $ANT\_MEMORY \cdot N\_ANTS > 1.1 \cdot N\_POINTS$, έτσι ώστε να είναι υψηλή η πιθανότητα να εξερευνηθούν όλα τα σημεία από τον *ACS*. Για το `ROULETTE_SIZE` θα πρέπει επίσης να ισχύει $ROULETTE\_SIZE > N\_POINTS - ANT\_MEMORY$
- Όταν έχει τεθεί η μεταβλητή `FIXED_MODE` σε 1, τότε καλό είναι να έχει τεθεί και η μεταβλητή `TEST_MODE`. Το αντίθετο δεν ισχύει

Σε κάθε περίπτωση, ο εξεταστής μπορεί απλά αλλάζοντας τη μεταβλητή `ALGORITHM` να δοκιμάσει όλους τους διαθέσιμους αλγορίθμους της εργασίας.

1.4 Ανάλυση λογικής

Σε αυτό το κεφάλαιο θα γίνει ανάλυση της λογικής πίσω από τους αλγορίθμους που υλοποιήθηκαν στα πλαίσια της εργασίας. Συγκεκριμένα, θα μας απασχολήσουν οι ζητούμενες παράλληλες εκδόσεις, και η σειριακή έκδοση του ant colony.

1.5 Παραλληλοποιώντας τον αλγόριθμο *Τυχαίας Αναζήτησης*

Για να παραλληλοποιηθεί ο αλγόριθμος τυχαίας αναζήτησης, αυτό που έγινε ήταν να τεθεί μια πιθανότητα 30% στο αρχείο `Common.h` που αντιστοιχεί στη μεταβλητή `NAIVE_PROBABILITY`. Έπειτα, δηλώθηκε μια γεννήτρια τυχαίων αριθμών που ανήκουν στο διάστημα $[0, 1]$. Με βάση την πιθανότητα που δίνει κάθε φορά η γεννήτρια συνάρτηση επιλέγεται ένας κόμβος - πόλη για να τροφοδοτήσει μετά τον αλγόριθμο τυχαίας αναζήτησης. Σε περίπτωση επιλογής, τότε αποκλείεται ο γείτονας κόμβος από την τροφοδότηση. Αυτό γίνεται για να μην υπάρχει λανθασμένος υπολογισμός κατά τη σύγκριση των αποστάσεων λόγω outdated TSP route. Αναλυτικά, περιγράφεται μια τέτοια περίπτωση παρακάτω:

- Έστω *νήμα 1* που ανταλλάσσει τους κόμβους 6 και 10
- Έστω *νήμα 2* που αναλαμβάνει τους κόμβους 2 και 9
- Έστω ότι το νήμα 1 αρχίζει την εκτέλεση του αλγορίθμου και βρίσκει ότι όντως η εναλλαγή των κόμβων 6 και 10 ελαχιστοποιεί το κόστος
- Το νήμα 1 αναστέλλεται προτού ενημερώσει το TSP route με την βελτιστοποιημένη έκδοση
- Το νήμα 2 αρχίζει την εκτέλεση του έχοντας ως TSP route την έκδοση πριν τα αποτελέσματα του πρώτου νήματος
- Το νήμα 2 βρίσκει ότι όντως η εναλλαγή των κόμβων 2 και 9 ελαχιστοποιεί το κόστος. Όμως το κόστος δεν έχει υπολογισθεί σωστά, καθώς πλέον ο γείτονας του κόμβου 9 είναι ο κόμβος 6 και όχι ο κόμβος 8
- Το νήμα 2 ενημερώνει το TSP route
- Το νήμα 1 ενημερώνει το TSP route

Επομένως, κάθε φορά που επιλέγεται ένας κόμβος να προστεθεί στο διάνυσμα εναλλαγής του αλγορίθμου, γίνεται αποκλεισμός του γείτονά του. Το τελευταίο βήμα είναι το «ανακάτεμα» (shuffling) του διανύσματος επιλογής, έτσι ώστε να

δημιουργηθούν τα τυχαία ζευγάρια. Έχοντας πλέον έτοιμο το διάνυσμα εναλλαγών, μπορεί να παραλληλοποιηθεί όλης η διαδικασία σύγκρισης των ζευγαριών με μοναδικό περιορισμό την *critical* περιοχή που συμβαίνει η ανανέωση της TSP διαδρομής. Το τελικό σχήμα του αλγορίθμου βρίσκεται στον αλγόριθμο 1.1.

Αλγόριθμος 1.1 Ο αλγόριθμος τυχαίας αναζήτησης

```
1  input: cities[N], ITERATIONS
2  output: cities[N]
3  begin
4
5          precompute exchange vector
6
7          foreach iteration repeat
8                  begin parallel region
9                          foreach element in exchange vector with 2 point increment repeat
10                                 exchange element(i) with element(i + 1)
11                                 compute new TSP cost
12                                 if cost is not better then
13                                         exchange element(i) with element(i + 1)
14                                 end if
15                         end foreach
16                 end parallel region
17         end foreach
18 end
```

1.6 Παραλληλοποιώντας τον Naive αλγόριθμο *Heinritz - Hsiao*

Για να παραλληλοποιηθεί η Naive έκδοση του αλγορίθμου *Heinritz - Hsiao* χρησιμοποιήθηκε το σχήμα που φαίνεται στον αλγόριθμο 1.2.

Αλγόριθμος 1.2 Ο Naive αλγόριθμος Heinritz - Hsiao

```
1  input: cities[N]
2  output: cities[N]
3  begin
4
5          initialize minimum cost variable with INF
6          initialize closest neighbors vector with self
7
8          foreach point a in cities  repeat
9                  begin parallel region
10                 (make minimum cost private, make closest neighbors vector private)
11                         foreach point b in cities repeat
12                                 if minimum cost > euclidean distance of a and b then
13                                         update the closest neighbors vector
14                                 end if
15                         end foreach
16                 end parallel region
17                 concat all private closest neighbors vectors
18                 randomly select one of the two closest points
19         end foreach
20 end
```

1.7 Ο αλγόριθμος βελτιστοποίησης *Αποικίας Μυρμηγκιών*

Η παραλλαγή του ACS που υλοποιήθηκε στα πλαίσια της εργασίας παρουσιάζεται στον αλγόριθμο 1.3. Για την υλοποίηση ορίστηκαν 5 σημαντικές μεταβλητές:

- Ant memory: Είναι ένας ακέραιος αριθμός που λειτουργεί ως άνω όριο εξερεύνησης. Κάθε μυρμήγκι μπορεί να εξερευνήσει μέχρι αυτό το όριο των κόμβων
- Roulette size: Είναι το μέγεθος της ρουλέτας στον αλγόριθμο *επιλογής με ρουλέτα*. Ουσιαστικά σε αυτόν τον αλγόριθμο τροφοδοτούνται N σημεία 2 διαστάσεων μαζί με τις αποστάσεις τους από ένα κοινό σημείο σ. Ο αλγόριθμος κρατάει έπειτα διαγράφει τα σημεία με τη μικρότερη σχετική απόσταση από το σ και αφήνει στη ρουλέτα τόσα σημεία όσα ορίζει η μεταβλητή *Roulette size*
- Pherormone matrix: Είναι ο πίνακας στον οποίο αποθηκεύεται η φερορμόνη που αφήνει το κάθε μυρμήγκι στην εκάστοτε ακμή
- Rho: Είναι ο ρυθμός με τον οποίο εξατμίζεται η φερορμόνη. Ουσιαστικά πρόκειται για ένα παράγοντα κανονικοποίησης
- Iterations: Είναι ο αριθμός επαναλήψεων του αλγορίθμου. Ουσιαστικά έχει το ρόλο της *εποχής* που βρίσκουμε σε βιβλιογραφία νευρωνικών δικτύων, ή το ρόλο της γενιάς που βρίσκουμε σε βιβλιογραφία γενετικών αλγορίθμων

Αλγόριθμος 1.3 Ο ACS

```
1  input: cities[N], number of ants, ant memory, iterations
2  output: acs route[N]
3  begin
4          foreach iteration repeat
5                  foreach ant a repeat
6                          place a on a random point i
7                          repeat
8                                  foreach point j in cities repeat
9                                          compute euclidean distance between i and j
10                                 end foreach
11                                 select closest unknown point k
12                                 add that point to path
13                                 leave some pherormone
14                                 set i equal to k
15                         until size of path is greater than ant memory
16                 end foreach
17         end foreach
18
19         place a new ant f on point o
20         add f to acs route array
21
22         foreach point a repeat
23                 find closest point k based on pherormone matrix
24                 add k to acs route array
25                 set f equal to k
26         end foreach
27 end
```

1.8 Ο παραλληλοποιημένος αλγόριθμος βελτιστοποίησης *Αποικίας Μυρμηγκιών*

Για να παραλληλοποιηθεί ο αλγόριθμος 1.3, πραλληλοποιήθηκε η διαδικασία εξερεύνησης για το κάθε μυρμήγκι. Ο αλγόριθμος 1.4 περιγράφει αυτήν τη μετατροπή. Ενδιαφέρον έχει η λογική πίσω από τον τρόπο αρχικοποίησης του πίνακα φερορμόνης. Για να μην επιλέξει ποτέ κάποιο μυρμήγκι να παραμείνει στον κόμβο που βρίσκεται, τα στοιχεία στη διαγώνιο του πίνακα είναι 0. Επίσης, λόγω του cold start προβλήματος του ACS, τα υπόλοιπα στοιχεία αρχικοποιούνται με ουδέτερο τρόπο, δηλαδή αρχικοποιούνται όλα με 1.

Αλγόριθμος 1.4 Ο παραλληλοποιημένος ACS

```
1  input: cities[N], number of ants, ant memory, iterations
2  output: acs route[N]
3  begin
4
5          initialize pherormone matrix
6
7          foreach iteration repeat
8                  begin parallel region
9                          foreach ant a repeat
10                                 place a on a random point i
11                                 repeat
12                                         foreach point j in cities repeat
13                                                 compute euclidean distance between i and j
14                                         end foreach
15                                         select closest unknown point k
16                                         add that point to path
17                                         begin critical area
18                                                 leave some pherormone
19                                         end critical area
20                                         set i equal to k
21                                 until size of path is greater than ant memory
22                         end foreach
23                 end parallel region
24         end foreach
25
26         place a new ant f on point 0
27         add 0 to acs route array
28
29         foreach point a repeat
30                 find closest point k based on pherormone matrix
31                 add k to acs route array
32                 set f equal to k
33         end foreach
34 end
```

1.9 Συγκρίνοντας τους αλγορίθμους

Στο repository που υπάρχει στο GitHub για το Project υπάρχει ένας πίνακας που παρουσιάζει αριθμητικά αποτελέσματα της σύγκρισης των διαφορετικών αλγορίθμων. Οπότε παρακάτω παρουσιάζονται με λεκτική περιγραφή κάποια αποτελέσματα.

Ως προς τη χωρητικότητα Ο ACS είναι ο πιο απαιτητικός αλγόριθμος. Μόνο για την αποθήκευση του πίνακα με τη φερορμόνη χρειάζεται τετραγωνικό χώρο. Δηλαδή για $10,000$ πόλεις χρειάζεται $10,000 \cdot 10,000 \cdot$ `sizeof(double)`. Αυτό εκτιμάται σε 850 MB, όπως φαίνεται και στα diagnostic tools του Visual Studio. Επίσης, απαραίτητο δεδομένο επίσης είναι και το μονοπάτι που ακολουθεί το κάθε μυρμήγκι. Για να ενισχυθεί η επιλογή κόμβων που δεν έχουν εξερευνηθεί από το εκάστοτε μυρμήγκι θα πρέπει να υπάρχει το διάνυσμα αυτό χωρισμένο σε κόμβους που έχει ακολουθήσει το μυρμήγκι και σε κόμβους που δεν έχει ακολουθήσει. Παρά τις βελτιστοποιήσεις που γίνονται στη μνήμη, το Memory Profiler του Visual Studio δίνει 2 GB στην σειριακή εκτέλεση του ACS, και 9 GB[1] για την παραλληλοποιημένη έκδοσή του. Συμπερασματικά, ο αλγόριθμος ACS είναι καλός για μικρό πλήθος πόλεων. Για αυτό και προτείνεται η σειριακή έκδοση να μην εκτελείται για πάνω από $3,000$ πόλεις, και η παράλληλη για όχι περισσότερο από $2,000$ πόλεις. Όσον αφορά τη χωρική πολυπλοκότητα των υπόλοιπων αλγορίθμων, αυτή είναι γραμμική. Το μόνο που χρειάζονται οι υπόλοιποι αλγόριθμοι είναι το διάνυσμα των πόλεων.

Ως προς την ταχύτητα Ο γρηγορότερος σειριακός αλγόριθμος είναι ο *Heinritz - Hsiao*. Αυτό είναι φυσιολογικό, καθώς έχει ίδια χωρική πολυπλοκότητα με τον αλγόριθμο τυχαίας αναζήτησης και με τον τροποποιημένο *Heinritz - Hsiao*, χωρίς να χρειάζεται να ορίσει κάποια γεννήτρια τυχαίων αριθμών που *βαραίνει* το πρόγραμμα. Οι υπόλοιποι αλγόριθμοι είναι λίγο πιο αργοί λόγω της γεννήτριας τυχαίων αριθμών.

Ως προς την ελαχιστοποίηση της συνάρτησης κόστους Τις καλύτερες αποδόσεις πετυχαίνει ο *Heinritz - Hsiao*. Έπειτα, ακολουθεί ο *ACS*. Στην τρίτη θέση βρίσκεται ο τροποποιημένος (naive) *Heinritz - Hsiao* και στην τελευταία, ο αλγόριθμος τυχαίας αναζήτησης.

1.10 Οδηγίες για τον κώδικα στην εργασία

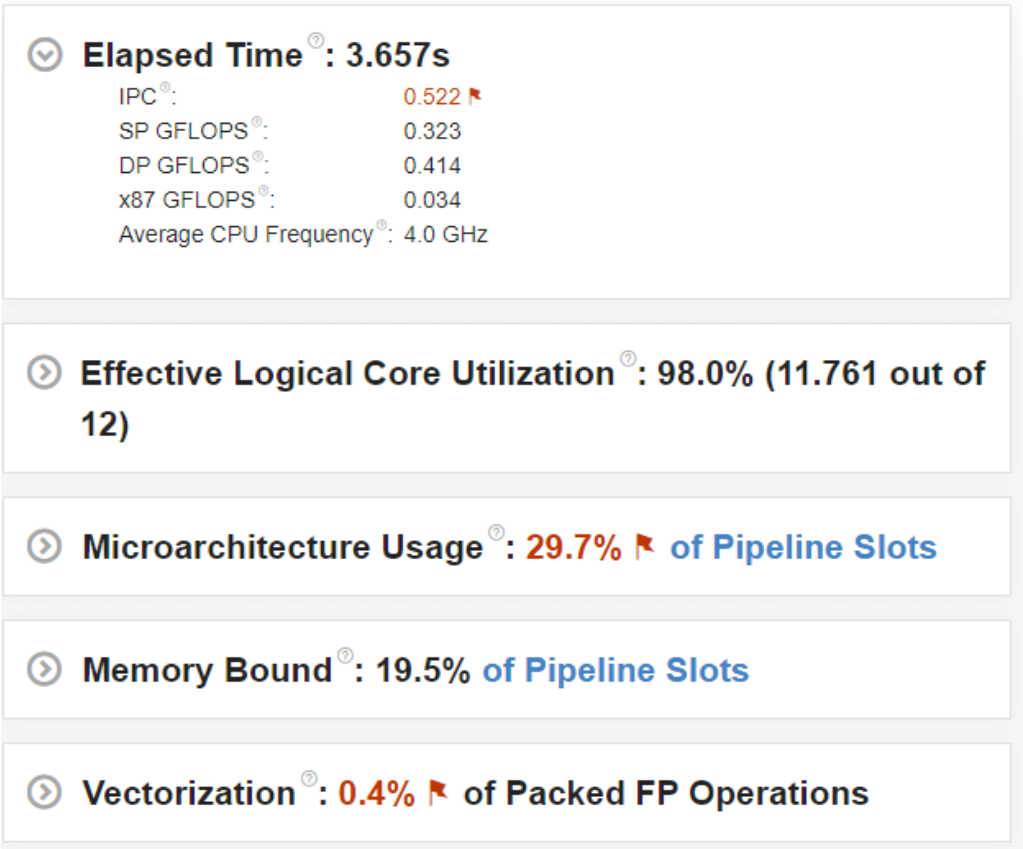
Ο εξεταστής μπορεί να ανοίξει όλα τα αρχεία τύπου `h` (κεφαλίδες). Εκεί, υπάρχει μια σύντομη αλλά περιεκτική περιγραφή με το ρόλο της κάθε κεφαλίδας. Τα πρώτα αρχεία που ενεργοποιούνται είναι τα αρχεία `City`, τα οποία αρχικοποιούν με τυχαίο τρόπο το σύνολο δεδομένων μας. Έπειτα, ενεργοποιούνται τα αρχεία `Utilities` τα οποία περιέχουν τις υλοποιήσεις των επιμέρους αλγορίθμων. Στην περίπτωση του ACS, προτού αρχίσει η εκτέλεση του αλγορίθμου, ενεργοποιούνται τα αρχεία `Pherormone`, τα οποία αρχικοποιούν τον πίνακα με τη φερορμόνη. Κατά την εκτέλεση των διάφορων αλγορίθμων χρησιμοποιούνται βοηθητικές ρουτίνες

[1] Η παραλληλοποιημένη έκδοση είναι πιο απαιτητική ως προς τη μνήμη καθώς υπάρχουν και private μεταβλητές. Ακόμα και οι shared μεταβλητές σε χαμηλό επίπεδο, για να τις επεξεργαστούν πολλά threads ταυτόχρονα αντιγράφονται στα διαφορετικά cores.

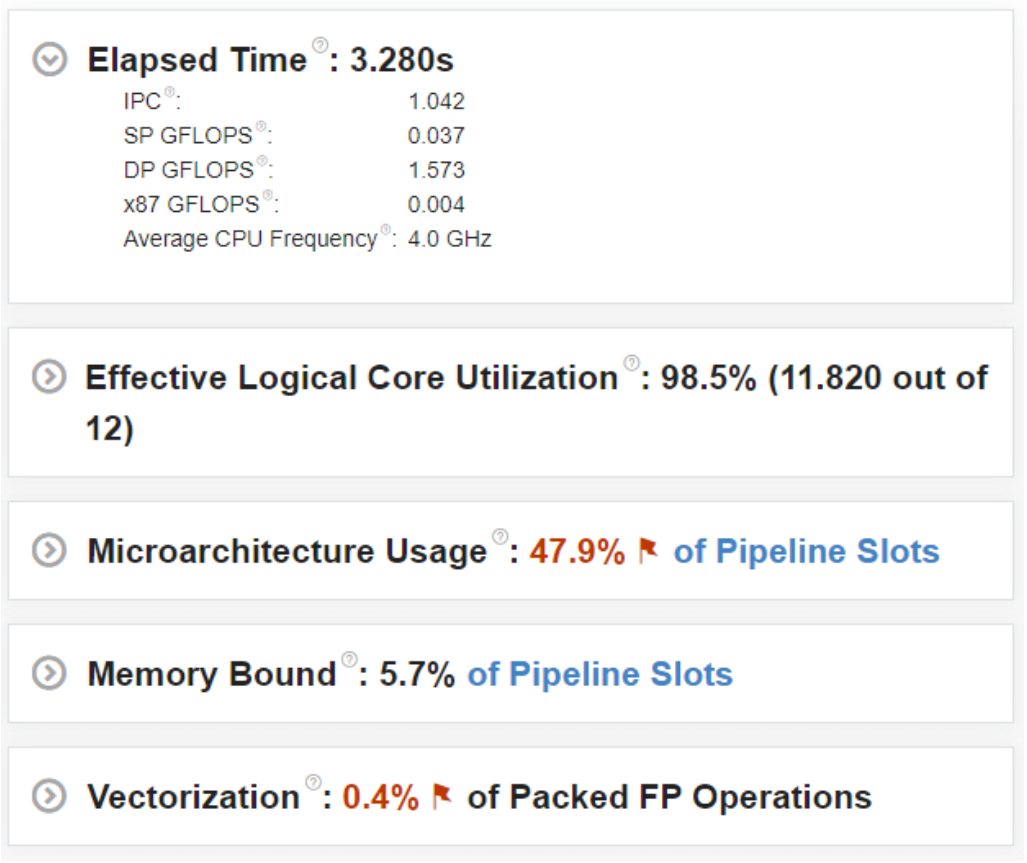
από τα αρχεία `Operations`. Ο αλγόριθμος *ACS* καλεί και τη ρουτίνα *Colonize* από τα αρχεία `Colonize`. Για τον υπολογισμό των αποστάσεων, είτε για μια ολόκληρη TSP διαδρομή, είτε μεταξύ 2 πόλεων κ.λπ. χρησιμοποιούνται τα αρχεία `Distance`. Τα αρχεία `Naive` περιέχουν τον αλγόριθμο *επιλογής με ρουλέτα*. Τέλος, για την εξαγωγή των αποτελεσμάτων σε CSV αρχεία, αλλά και για την εκτύπωση της προόδου του αλγορίθμου στο περιβάλλον terminal, υπάρχουν τα αρχεία `Validation` και `Interface` αντίστοιχα. Στα αρχεία `Driver` είναι δηλωμένη η *main* συνάρτηση του προγράμματος.

1.11 Ποσοστό παραλληλοποίησης

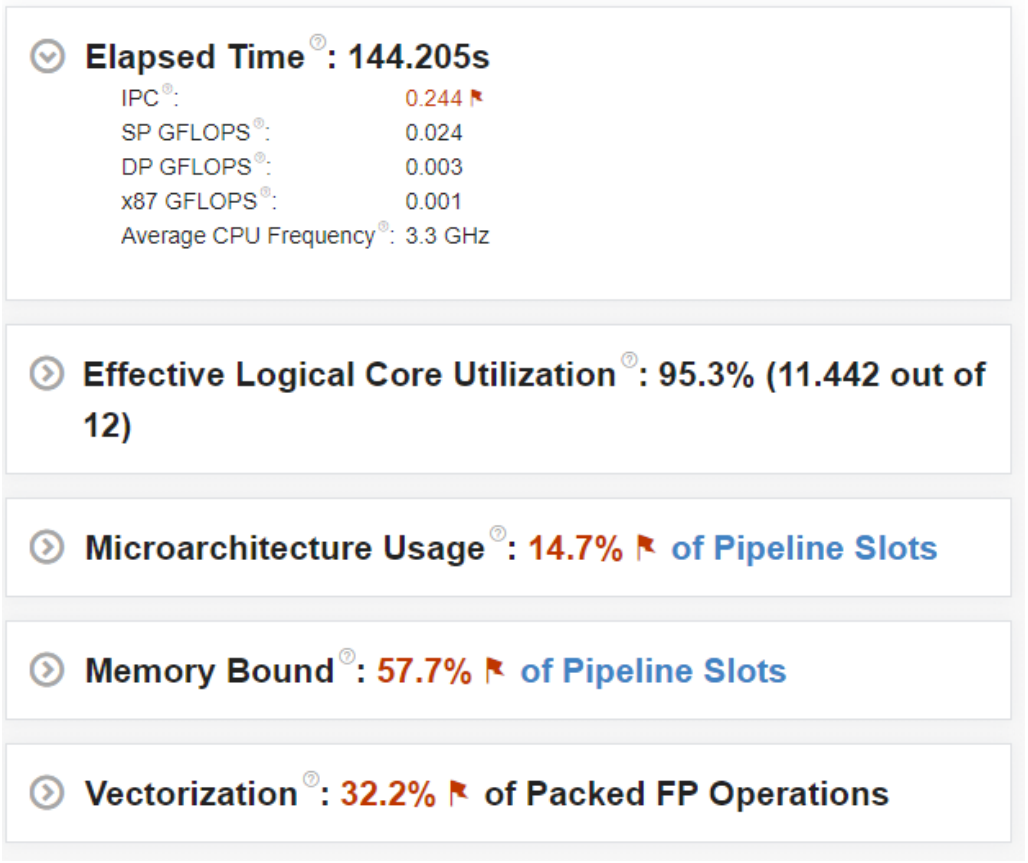
Στα σχήματα 1.1 έως και 1.3 φαίνονται τα ποσοστά *Effective Core Utilization* που εξήγαγε ο Profiler της Intel.



Σχήμα 1.1: Performance Snapshot από τον Profiler της Intel για τον παραλληλοποιημένο αλγόριθμο τυχαίας αναζήτησης



Σχήμα 1.2: Performance Snapshot από τον Profiler της Intel για τον παραλληλοποιημένο τροποποιημένο αλγόριθμο Heinritz - Hsiao



Σχήμα 1.3: Performance Snapshot από τον Profiler της Intel για τον παραλληλοποιημένο αλγόριθμο ACS

2

Ο κώδικας

Κώδικας 2.0.1: Η κεφαλίδα City.h

```cpp
/**
 * City.h
 *
 * In this header file, we define some function that are executed
 * only in the beginning of the program. These functions are used
 * either to generate a random set or to set the dataset to a
 * fixed state of points that will play the role of the cities
 * for the TSP.
 */

#pragma once

#include "Common.h"

void initialize_cities(std::array<std::pair<int, int>, N_POINTS>& cities);
void set_fixed_dataset(std::array<std::pair<int, int>, N_POINTS>& cities);
```

Κώδικας 2.0.2: Το αρχείο City.cpp

```cpp
#include "City.h"

/**
 * Initializes `cities` variable.
 *
 * @param[in, out] cities the random dataset generated for the different TSP approaches
 *
 * @remark [<random> Engines and Distributions](https:///<docs.microsoft.com/en-us/cpp/standard-library/random?view=msvc-160#engdist)
 */
void initialize_cities(std::array<std::pair<int, int>, N_POINTS>& cities)
{
    std::random_device x_rd;                                 /// non-deterministic generator
    std::mt19937 x_gen(x_rd());                              /// to seed mersenne twister
    std::uniform_int_distribution<int> x_dist(0, X_MAX);     /// distribute results between 0 and X_MAX inclusive

    std::random_device y_rd;                                 /// non-deterministic generator
    std::mt19937 y_gen(y_rd());                              /// to seed mersenne twister
    std::uniform_int_distribution<int> y_dist(0, Y_MAX);     /// distribute results between 0 and Y_MAX inclusive

    for (int i = 0; i < N_POINTS; i += 1)                    /// Initializes all `N_POINTS`
    {
        cities.at(i).first  = x_dist(x_gen);                 /// Generates a random X coordinate for point i
        cities.at(i).second = y_dist(y_gen);                 /// Generates a random Y coordinate for point i
    }
}

/**
 * Sets dataset to the fixed array declared in `Common.h`
 *
 * @param[in, out] cities the dataset that is set
 */
void set_fixed_dataset(std::array<std::pair<int, int>, N_POINTS>& cities)
{
    std::copy(std::begin(FIXED_DATASET), std::end(FIXED_DATASET), std::begin(cities));
}
```

Κώδικας 2.0.3: Η κεφαλίδα Colonize.h

```cpp
/**
 * Colonize.h
 *
 * In this header file, we define some functions
 * regarding the ACS algorithm. The functions initiate
 * the colonization procedure. There is also a function
 * that estimates the total distance covered by the
 * travelling salesman (`acs_tsp`).
 */

#pragma once

#include "Naive.h"
#include "Common.h"
#include "Distance.h"
#include "Operation.h"

void acs_tsp(const std::vector<std::array<double, N_POINTS>> pherormone_matrix, std::vector<int>& tsp_route);
void colonize(const std::array<std::pair<int, int>, N_POINTS> cities, std::vector<std::array<double, N_POINTS>>& pherormone_matrix);
void colonize_parallel(const std::array<std::pair<int, int>, N_POINTS> cities, std::vector<std::array<double, N_POINTS>>& pherormone_matrix);
```

Κώδικας 2.0.4: Η κεφαλίδα Common.h

```cpp
/**
 * Common.h
 *
 * In this header file, we define the constants
 * used throughout the project. We also
 * include all the header files necessary to
 * make the implementation work.
 */

#pragma once

#include <array>                                         /// std::array
#include <cmath>                                         /// std::sqrt
#include <vector>                                        /// std::vector
#include <chrono>                                        /// std::chrono
#include <string>                                        /// std::string
#include <random>                                        /// std::random_device
#include <limits>                                        /// std::numeric_limits
#include <utility>                                       /// std::pair
#include <fstream>                                       /// std::ofstream
#include <numeric>                                       /// std::iota
#include <iomanip>                                       /// std::setw
#include <iostream>                                      /// std::cout
#include <iterator>                                      /// std::<T>::iterator
#include <algorithm>                                     /// std::find
#include <functional>                                    /// std::bind

#include <omp.h>                                         /// OpenMP Multiprocessing Programming Framework


constexpr int FIXED_MODE = 0;                           /// This sets execution mode to `fixed mode` where the dataset is fixed
constexpr int TEST_MODE = 0;                            /// If 0 then the algorithm runs in debug mode,
                                                        /// meaning less `N_POINTS` (cities), iterations
                                                        /// and more verbosity. If 1 then the algorithm
                                                        /// runs in release mode.
constexpr int ALGORITHM = 1;                            /// If 0 then the algorithm running is the `Naive TSP`, as described in `Utilities.cpp`
                                                        /// If 1 then the algorithm running is the `Naive TSP` (Parallel Implementation), as described in `Utilities.cpp`
                                                        /// If 2 then the algorithm running is the `TSP with nearest neighbor`, as described in `Utilities.cpp`
                                                        /// If 3 then the algorithm running is the `TSP with naive nearest neighbor`, as described in `Utilities.cpp`
                                                        /// If 4 then the algorithm running is the `TSP with naive nearest neighbor` (Parallel Implementation), as described in `Utilities.cpp`
                                                        /// If 5 then the algorithm running is the `ACS TSP`, as described in `Utilities.cpp`
                                                        /// If 6 then the algorithm running is the `ACS TSP` (Parallel Implementation), as described in `Utilities.cpp`
                                                        /// Else no algorithm runs and a warning is displayed
constexpr int N_THREADS = 12;                           /// This is the number of threads requested in any parallel implementations of the project

constexpr int X_MAX = 1000;                             /// This is the upper limit of any city's "Longitude". This means that a city can have X coordinates that belong in [0, X_MAX]
constexpr int Y_MAX = 1000;                             /// This is the upper limit of any city's "Latitude". This means that a city can have Y coordinates that belong in [0, Y_MAX]
constexpr int N_POINTS = (                              /// This variable sets the number of cities. For the ACS implementations, this number must be low, due to memory management issues.
    TEST_MODE == 1 ? 10   :                             /// That is mainly due to the pherormone matrix, which is a dense matrix, and cannot be easily factorized to optimize memory management.
    ALGORITHM == 5 ? 1000 :
    ALGORITHM == 6 ? 1000 : 10000);
constexpr int ITERATIONS = (                            /// This is the number of iterations that any algorithm will execute. The TSP is an NP-complete problem.
    TEST_MODE == 1 ? 10     :                           /// Therefore and since the approaches in this project are mainly naive (there is a statistical element,
    ALGORITHM == 1 ? 10000  :                           /// such as a random choice between N possible decisions), we set a number of iterations for the algorithm to run.
    ALGORITHM == 5 ? 100    :
    ALGORITHM == 6 ? 100    : 100000000);
constexpr double NAIVE_PROBABILITY = 0.3;               /// This probability is used in the Naive Heinritz - Hsiao approach of the TSP. This probability means that
                                                        /// there is a 70% chance that the algorithm will choose the nearest point to add to its path, and 30% chance
                                                        /// to choose the second nearest point to add to its path. This variable must never be set above 0.5.

constexpr int BOOST = 10;                               /// This variable boosts the ammount of pherormone added to the edge chosen by the ant.
                                                        /// This makes ACS convergence better, regarding time complexity.
constexpr int N_ANTS = 12;                              /// This is the ammount of ants running. This number also represents the ammount of ***threads*** running
                                                        /// for the parallel version of the ACS.
constexpr int ANT_MEMORY = (TEST_MODE == 1 ? 5 : 100);  /// This sets the ant memory. This means that each ant gets to cross 100 points, and since each ant starts
                                                        /// from a different point of the map, there is a pretty high probability that the ants will cover all the
                                                        /// map by the end of the algorithm. That is if `N_ANTS` multiplied by `ANT_MEMORY` is greater than `N_POINTS`.
                                                        /// Finally, this variable must always be greater than `N_POINTS`.
constexpr int ROULETTE_SIZE = (TEST_MODE == 1 ? 3 : 4); /// This the size of the roulette wheel. The roulette wheel is a decision making module that is used in the
                                                        /// ACS algorithm. This is to solve the computational error that surfaces due to the huge number of possible
                                                        /// paths the ant can follow. This variable must be changed with respect to the total number of points and to the
                                                        /// size of the ant memory. The math for this variable is that is has to be less than `N_POINTS` minus `ANT_MEMORY`.
constexpr double RHO = 0.1;                             /// This is the vaporazation ratio for the ACS.
constexpr int CHUNK = (int)(4 * (N_POINTS / N_ANTS)) + 1;   /// Sets chunk size for OpenMP loop

constexpr std::array<std::pair<int, int>, N_POINTS> FIXED_DATASET = {
    std::make_pair(42, 53),
    std::make_pair(364, 45),
    std::make_pair(84, 212),
    std::make_pair(67, 865),
    std::make_pair(145, 1),
    std::make_pair(875, 52),
    std::make_pair(764, 899),
    std::make_pair(193, 63),
    std::make_pair(2, 754),
    std::make_pair(263, 631)
};                                                      /// This vector is the initializer of the dataset if
                                                        /// the programmer wishes to compare the TSP approaches with a fixed dataset
```

Κώδικας 2.0.5: Η κεφαλίδα Distance.h

```cpp
/**
 * Distance.h
 *
 * In this header file, we define the
 * cost functions used throughout the project.
 * Those functions use Euclidean distance to
 * estimate the cost of a process.
 */

#pragma once

#include "Common.h"

std::pair<int, long double> tsp_hop_cost(const std::pair<int, int> point_one, const std::pair<int, int> point_two);
std::vector<long double> tsp_tour_cost(const std::array<std::pair<int, int>, N_POINTS> cities);
long double euclidean_difference(int is_neighbor, const std::pair<int, int> pre_point_one, const std::pair<int, int> point_one, const std::pair<int, int> suc_point_one, const std::pair<int, int> pre_point_two, const std::pair<int, int> point_two, const std::pair<int, int> suc_point_two);
void evaluate_universe(const std::vector<std::array<int, N_POINTS>> non_explored, const int ant_idx, const int last_explored_idx, const std::vector<std::array<double, N_POINTS>> pherormone_matrix, const std::array<std::pair<int, int>, N_POINTS> cities, std::vector<std::pair<int, double>>& evaluation);
void evaluate_universe_parallel(const std::array<int, N_POINTS> non_explored, const int last_explored_idx, const std::vector<std::array<double, N_POINTS>> pherormone_matrix, const std::array<std::pair<int, int>, N_POINTS> cities, std::vector<std::pair<int, double>>& evaluation);
double acs_tsp_cost(const std::vector<int> tsp_route, const std::array<std::pair<int, int>, N_POINTS> cities);
```

Κώδικας 2.0.6: Το αρχείο Colonize.cpp

```cpp
#include "Colonize.h"

/**
 * Initiates colonization sequence.
 *
 * @param[in] cities the dataset for ACO
 * @param[in, out] pheromone_matrix the matrix CAS array of sizes "N_POINTS" x "N_POINTS") with the pheromone amount left in each edge
 *
 */
void colonize(
    const std::array<std::pair<int, int>, N_POINTS>      cities,
    std::vector<std::array<double>, N_POINTS>>&          pheromone_matrix)
{
    std::vector<std::array<int, N_POINTS>> non_explored(N_ANTS);   /// Declares a vector that holds all possible points.
                                                                   /// This vector is fragmented into an explored part and an non explored part.
                                                                   /// The explored part is separated by -1s the non explored part by an index "k".
                                                                   /// On the left of "k" lie the explored nodes, and on the right the non explored.
                                                                   /// "k" always points to the node where each ant is currently at.
    std::vector<std::array<std::pair<int, double>, ANT_MEMORY>> explored(N_ANTS);
                                                                   /// Declares a vector that holds the points that the ant has crossed.
                                                                   /// This vector has mainly debugging purposes. However, it is not memory demanding.

    std::random_device ant_rd;                                     /// non-deterministic generator
    std::mt19937 ant_gen(ant_rd());                                /// to seed mersenne twister
    std::uniform_int_distribution<int> ant_dist(0, N_POINTS - 1);  /// distributes results between 0 and N_POINTS exclusive

    std::random_device wheel_rd;                                   /// non-deterministic generator
    std::mt19937 wheel_gen(wheel_rd());                            /// to seed mersenne twister
    std::uniform_real_distribution<double> wheel_dist(0, 1);       /// distributes results between 0 and 1 exclusive

    for (int i = 0; i < ITERATIONS; i ++)
    {
        std::cout << "Iteration [" << i << "]" << std::endl;       /// Prints progress info of the ACO since it takes some times to colonize the map
        for (int j = 0; j < N_ANTS; j ++)
        {
            std::iota(non_explored.at(j).begin(), non_explored.at(j).end(), 1);
                                                                   /// Initiates 'non_explored' for the 'j'-th ant
            explored.at(j).at(0) = std::make_pair(ant_dist(ant_gen), 0.0);
                                                                   /// Initiates 'explored' for the 'j'-th ant

            for (int k = 0; k < ANT_MEMORY - 1; k ++)
            {
                int idx = find(non_explored.at(j), explored.at(j).at(k).first);
                                                                   /// Locates the index of the previously added node

                std::swap(non_explored.at(j).at(k), non_explored.at(j).at(idx));
                                                                   /// Flushes previously added node on the left of index 'k' into 'non_explored'
                std::vector<std::pair<int, double>> evaluation;    /// Declares a vector to store all possible edge evaluations
                evaluation.reserve(N_POINTS - k);                  /// Reserves the proper memory size to increase performance
                evaluate_universe(non_explored, j, k, pheromone_matrix, cities, evaluation);
                                                                   /// Evaluates all possible edges

                int chosen_idx = roulette_wheel(evaluation, wheel_dist(wheel_gen));
                                                                   /// Calls roulette_wheel() to get the chosen edge
                explored.at(j).at(k + 1) = std::make_pair(chosen_idx, NODST / tsp_hop_cost(cities.at(explored.at(j).at(k).first), cities.at(chosen_idx).second));
                                                                   /// Updates 'explored' vector
                pheromone_matrix.at(explored.at(j).at(k).first).at(explored.at(j).at(k + 1).first) += explored.at(j).at(k + 1).second;
                                                                   /// Updates pheromone matrix
            }
        }
        for (int j = 0; j < N_POINTS; j ++)                        /// Vaporization loop
        {
            for (int k = 0; k < N_POINTS; k ++)
            {
                pheromone_matrix.at(j).at(k) = (1 - RHO) * pheromone_matrix.at(j).at(k);
                                                                   /// Vaporizes pheromone in each edge
            }
        }
    }
}

/**
 * Initiates colonization sequence. This is a fork of the 'colonize' function above, parallelized with OpenMP 4.0.
 *
 * @param[in] cities the dataset for ACO
 * @param[in, out] pheromone_matrix the matrix CAS array of sizes "N_POINTS" x "N_POINTS") with the pheromone amount left in each edge
 *
 */
void colonize_parallel(
    const std::array<std::pair<int, int>, N_POINTS>      cities,
    std::vector<std::array<double>, N_POINTS>>&          pheromone_matrix)
{
    std::array<int, N_POINTS> non_explored;                        /// In the parallel fork, the 'non_explored' array is set private for each ant (each ant represents a thread).
    std::array<std::pair<int, double>, ANT_MEMORY> explored;       /// In the parallel fork, the 'explored' array is set private for each ant (each ant represents a thread).

    std::random_device ant_rd;
    std::mt19937 ant_gen(ant_rd());
    std::uniform_int_distribution<int> ant_dist(0, N_POINTS - 1);

    std::random_device wheel_rd;
    std::mt19937 wheel_gen(wheel_rd());
    std::uniform_real_distribution<double> wheel_dist(0, 1);

    for (int i = 0; i < ITERATIONS; i ++)
    {
        std::cout << "Iteration [" << i << "]" << std::endl;
#pragma omp parallel for num_threads(N_ANTS) private(non_explored, explored) schedule(runtime)
        for (int j = 0; j < N_ANTS; j ++)
        {
            std::iota(non_explored.begin(), non_explored.end(), 1);
            explored.at(0) = std::make_pair(ant_dist(ant_gen), 0.0);
            for (int k = 0; k < ANT_MEMORY - 1; k ++)
            {
                int idx = find(non_explored, explored.at(k).first);
                std::swap(non_explored.at(k), non_explored.at(idx));
                std::vector<std::pair<int, double>> evaluation;
                evaluation.reserve(N_POINTS - k);
                evaluate_universe_parallel(non_explored, k, pheromone_matrix, cities, evaluation);
                int chosen_idx = roulette_wheel(evaluation, wheel_dist(wheel_gen));
                explored.at(k + 1) = std::make_pair(chosen_idx, NODST / tsp_hop_cost(cities.at(explored.at(k).first), cities.at(chosen_idx).second));
#pragma omp critical
                pheromone_matrix.at(explored.at(k).first).at(explored.at(k + 1).first) += explored.at(k + 1).second;
            }
        }
#pragma omp parallel for collapse(2) schedule(dynamic, CHUNK)
        for (int j = 0; j < N_POINTS; j ++)
        {
            for (int k = 0; k < N_POINTS; k ++)
            {
                pheromone_matrix.at(j).at(k) = (1 - RHO) * pheromone_matrix.at(j).at(k);
            }
        }
    }
}

/**
 * Finds TSP route based on pheromone matrix.
 *
 * This implementation is based on silicon from genetic algorithms.
 * This means that for the released TSP route, only the best solutions
 * based on the 'pheromone_matrix' will be concatenated to form the TSP route.
 *
 * @param[in] pheromone_matrix the matrix with the pheromone amount left in each edge
 * @param[in, out] tsp_route the node indexes with respects to 'cities' variable
 *
 * @remark https://en.cppreference.com/w/cpp/algorithm/iota
 * @remark https://en.cppreference.com/w/cpp/algorithm/transform
 * @remark https://en.cppreference.com/w/cpp/algorithm/sort
 * @remark https://en.cppreference.com/w/cpp/utility/functional/bind
 *
 */
void ant_tsp(
    const std::vector<std::array<double, N_POINTS>>    pheromone_matrix,
    std::vector<int>&                                  tsp_route)
{
    tsp_route.emplace_back(0);
    for (int i = 0; i < N_POINTS - 1; i += 1)
    {
        std::vector<double> pheromone_column;                      /// Declares a vector to store the a column from 'pheromone_matrix' variable
        std::vector<int> pheromone_idx(N_POINTS);                  /// Declares a vector to be used to associate each value in 'pheromone_column' variable with a node
        std::vector<std::pair<int, double>> roulette(N_POINTS);    /// Declares a vector to bind the vectors above
        copy(pheromone_matrix, tsp_route.at(i), pheromone_column);///Calls copy() from Operation.h and copies column 'i' from 'pheromone_matrix' variable into 'pheromone_column'
        std::iota(pheromone_idx.begin(), pheromone_idx.end(), 1);  /// Initializes 'pheromone_idx'
        std::transform(pheromone_idx.begin(), pheromone_idx.end(), pheromone_column.begin(), roulette.begin(), std::bind2f, std::placeholders::_1, std::placeholders::_2));
                                                                   /// Initializes 'roulette' with 'pheromone_idx' and 'pheromone_idx' using 'roulette'
        std::sort(roulette.begin(), roulette.end(), sortbysec_gt());/// Sorts 'roulette' vector
        filter(roulette, tsp_route);                               /// Filters 'roulette' based on the route computed by the 'i'th iteration in order to avoid selecting a node that has already been explored
        tsp_route.emplace_back(roulette.back().first);             /// Updates 'tsp_route' vector with the selected node
    }
}
```

Κώδικας 2.0.7: Το αρχείο Distance.cpp

```cpp
#include "Distance.h"

/**
 * Computes the Euclidean distance of 2 given points.
 *
 * @param[in] point_one this is the first point
 * @param[in] point_two this is the second point
 *
 * @return std::pair<int, double> where the integer is an possible overflow warning flag, and the double value is the Euclidean distance
 */
std::pair<int, long double> tsp_hop_cost(const std::pair<int, int> point_one, const std::pair<int, int> point_two)
{
    std::pair<int, long double> diff;                              /// Declares the return variable
    diff.second = sqrt((point_one.first - point_two.first) * (point_one.first - point_two.first) +
      ((point_one.second - point_two.second) * (point_one.second - point_two.second));
                                                                   /// Computes the Euclidean distance between the 2 points
    if (diff.second > std::numeric_limits<double>::max())          /// Check for possible overflow
    {
        diff.second = std::numeric_limits<double>::max();          /// Mark overflow
        diff.first = 1;                                            /// Set overflow warning flag
    }
    else
    {
        diff.first = 0;                                            /// Deactivate overflow warning flag
    }

    return diff;
}

/**
 * Accumulates all costs and computes total TSP tour cost.
 *
 * @param[in] cities the preprocessed dataset
 *
 * @return a vector of distances
 *
 * Quite of the module detects possible overflow, then it pushes back to
 *    the vector a new double element and updates that element. The sum
 *    of all three elements is the total distance which due to computer
 *    overflow warning are separated using that vector.
 */
std::vector<long double> tsp_tour_cost(const std::array<std::pair<int, int>, N_POINTS> cities)
{
    std::vector<long double> cost;                                 /// Declares the costs vector
    cost.reserve(100);                                             /// Reserves a proper amount of memory
    int overflow = 0;                                              /// Sets overflow flag
    for (int i = 0; i < cities.size() - 1; i += 1)
    {
        std::pair<int, long double> diff = tsp_hop_cost(cities.at(i), cities.at(i + 1));
                                                                   /// Computes the Euclidean distance between 2 points
        if (overflow)                                              /// Marks possible overflow error
        {
            cost.push_back(diff.second);                           /// Creates and updates a new element
        }
        else
        {
            cost.at(cost.size() - 1) += diff.second;               /// Updates cost vector
        }
        overflow = diff.first;                                     /// Updates overflow flag
    }
    return cost;
}

/**
 * Computes the Euclidean distance between:
 *    - Point A and its predecessor
 *    - Point A and its successor
 *    - Point B and its predecessor
 *    - Point B and its successor
 *
 * @param[in] is_neighbor this is a flag indicating that Point A is either a predecessor or a successor of Point B
 * @param[in] pre_point_one this is the predecessor of Point A
 * @param[in] point_one this is Point A
 * @param[in] suc_point_one this is the successor of Point A
 * @param[in] pre_point_two this is the predecessor of Point B
 * @param[in] point_two this is Point B
 * @param[in] suc_point_two this is the successor of Point B
 *
 * @return the sum of the distances described above.
 *
 * Note it also marks possible integer overflow fault.
 */
long double euclidean_difference(int is_neighbor,
    const std::pair<int, int> pre_point_one,
    const std::pair<int, int> point_one,
    const std::pair<int, int> suc_point_one,
    const std::pair<int, int> pre_point_two,
    const std::pair<int, int> point_two,
    const std::pair<int, int> suc_point_two)
{
    long int pred_diff_point_one = (pre_point_one.first - point_one.first) * (pre_point_one.first - point_one.first)) +
        ((pre_point_one.second - point_one.second) * (pre_point_one.second - point_one.second));  /// Computes Euclidean distance between Point A and its predecessor
    long int succ_diff_point_one = ((suc_point_one.first - point_one.first) * (suc_point_one.first - point_one.first)) +
        ((suc_point_one.second - point_one.second) * (suc_point_one.second - point_one.second));  /// Computes Euclidean distance between Point A and its successor
    long int pred_diff_point_two = ((pre_point_two.first - point_two.first) * (pre_point_two.first - point_two.first)) +
        ((pre_point_two.second - point_two.second) * (pre_point_two.second - point_two.second));  /// Computes Euclidean distance between Point B and its predecessor
    long int succ_diff_point_two = ((suc_point_two.first - point_two.first) * (suc_point_two.first - point_two.first)) +
        ((suc_point_two.second - point_two.second) * (suc_point_two.second - point_two.second));  /// Computes Euclidean distance between Point B and its successor
    if (pred_diff_point_one > std::numeric_limits<int>::max())
    {
        pred_diff_point_one = std::numeric_limits<int>::max();                                    /// Marks possible integer overflow in 'pred_diff_point_one'
    }
    if (succ_diff_point_one > std::numeric_limits<int>::max())                                    /// Marks possible integer overflow in 'succ_diff_point_one'
    {
        succ_diff_point_one = std::numeric_limits<int>::max();
    }
    if (pred_diff_point_two > std::numeric_limits<int>::max())                                    /// Marks possible integer overflow in 'pred_diff_point_two'
    {
        pred_diff_point_two = std::numeric_limits<int>::max();
    }
    if (succ_diff_point_two > std::numeric_limits<int>::max())                                    /// Marks possible integer overflow in 'succ_diff_point_two'
    {
        succ_diff_point_two = std::numeric_limits<int>::max();
    }


    if (is_neighbor)
    {
        return std::sqrt(pred_diff_point_one) + std::sqrt(succ_diff_point_two);                   /// If the given points are neighbors, it corrects the sum of distance
    }
    else
    {
        return std::sqrt(pred_diff_point_one) + std::sqrt(succ_diff_point_one) + std::sqrt(pred_diff_point_two) + std::sqrt(succ_diff_point_two);
    }
}

/**
 * Evaluates all possible edge costs in ACO.
 *
 * @param[in] non_explored the vector of the ant path sorted in non explored points and explored ones
 * @param[in] last_explored_idx the index to the last explored element in 'non_explored' vector
 * @param[in] last_explored_idx the index to the last explored element in 'non_explored' vector
 * @param[in] pheromone_matrix the matrix in which ant pheromone is stored
 * @param[in] cities the dataset generated in the beginning containing the "x, y" coordinates of the points (cities)
 * @param[in, out] evaluation this is the vector where we store each edge cost
 *
 */
void evaluate_universe(
    const std::vector<std::array<int, N_POINTS>>&          non_explored,
    const int                                              non_idx,
    const int                                              last_explored_idx,
    const std::vector<std::array<double, N_POINTS>>&       pheromone_matrix,
    const std::array<std::pair<int, int>, N_POINTS>        cities,
    std::vector<std::pair<int, double>>&                   evaluation)
{
    for (int i = last_explored_idx + 1; i < N_POINTS; i += 1)
    {
        double cost = tsp_hop_cost(cities.at(non_explored.at(last_idx).at(last_explored_idx)), cities.at(non_explored.at(last_idx).at(i)));
                                                               /// Computes the cost between the last explored node and every other possible node
        double pheromone = pheromone_matrix.at(non_explored.at(last_idx).at(last_explored_idx)).at(non_explored.at(last_idx).at(i));
                                                               /// Fetches the pheromone assigned to that edge
        evaluation.emplace_back(std::make_pair(non_explored.at(last_idx).at(i), pheromone + (1 / cost)));
                                                               /// Evaluates that edge based on the cost and its past pheromone
    }
}

/**
 * Evaluates all possible edge costs in ACO. This is a fork of the 'evaluate_universe' function above, fine tuned for the parallelized version of ACO.
 *
 * @param[in] non_explored the array of the ant path sorted in non explored points and explored ones
 * @param[in] last_explored_idx the index to the last explored element in 'non_explored' vector
 * @param[in] pheromone_matrix the matrix in which ant pheromone is stored
 * @param[in] cities the dataset generated in the beginning containing the "x, y" coordinates of the points (cities)
 * @param[in, out] evaluation this is the vector where we store each edge cost
 *
 */
void evaluate_universe_parallel(
    const std::array<int, N_POINTS>                        non_explored,
    const int                                              last_explored_idx,
    const std::vector<std::array<double, N_POINTS>>        pheromone_matrix,
    const std::array<std::pair<int, int>, N_POINTS>        cities,
    std::vector<std::pair<int, double>>&                   evaluation)
{
    for (int i = last_explored_idx + 1; i < N_POINTS; i += 1)
    {
        double cost = tsp_hop_cost(cities.at(non_explored.at(last_explored_idx)), cities.at(non_explored.at(i)));
        double pheromone = pheromone_matrix.at(non_explored.at(last_explored_idx)).at(non_explored.at(i));
        evaluation.emplace_back(std::make_pair(non_explored.at(i), pheromone + (1 / cost)));
    }
}


/**
 * Computes ACO TSP tour cost.
 *
 * @param[in] tsp_route the tsp route estimated after colonization
 * @param[in] cities the random generated dataset of points (cities)
 *
 * @return the cost of the estimated TSP route by ACO
 */
double aco_tsp_cost(const std::vector<int> tsp_route, const std::array<std::pair<int, int>, N_POINTS> cities)
{
    double cost = 0.0;
    for (int i = 0; i < tsp_route.size() - 1; i += 1)
    {
        cost += tsp_hop_cost(cities.at(tsp_route.at(i)), cities.at(tsp_route.at(i + 1)).second;
    }
    cost += tsp_hop_cost(cities.at(tsp_route.at(tsp_route.size() - 1)), cities.at(tsp_route.at(0))).second;

    return cost;
}
```

Κώδικας 2.0.8: Η κεφαλίδα Driver.h

```h
/**
 * Driver.h
 *
 * In this header file, we include all the
 * custom made header files used in all
 * the different approaches implemented
 * to solve the TSP.
 */

#pragma once

#include "City.h"
#include "Brute.h"
#include "Common.h"
#include "Colonies.h"
#include "Chalkance.h"
#include "Interface.h"
#include "Utilities.h"
#include "Operation.h"
#include "Pheromone.h"
#include "Validation.h"
```

Κώδικας 2.0.9: Το αρχείο Driver.cpp

```cpp
#include "Driver.h"

/**
 * Implements the driver for the different approaches that solve the TSP.
 *
 * @return 0, if the executable was terminated normally
 */
int main(void)
{
    std::array<std::pair<int, int>, N_POINTS> cities;                                       /// Declares a vector to store the dataset
    if (FIXED_MODE) { set_fixed_dataset(cities); }                                          /// If in FIXED_MODE for - initialize the dataset
    else { initialize_dataset(cities); }                                                    /// Else initialize the dataset with random points.
    std::string algorithm;                                                                  /// Declares a string to associate it with the algorithm running
    std::chrono::time_point<std::chrono::system_clock> start = std::chrono::system_clock::now();  /// Declares a starting time point which helps in benchmarking
    switch (ALGORITHM)                                                                      /// Depending on the algorithm selected by the user call the appropriate routine
    {
        case 0:
            algorithm.assign("CALLED 0| \"Naive TSP\"");
            naive_tsp(cities);
            break;
        case 1:
            algorithm.assign("CALLED 1| \"Naive TSP\" (Parallel Implementation)");
            naive_tsp_parallel(cities);
            break;
        case 2:
            algorithm.assign("CALLED 2| \"TSP with nearest neighbor\"");
            heuristic_hatas(cities);
            break;
        case 3:
            algorithm.assign("CALLED 3| \"TSP with naive nearest neighbor\"");
            naive_heuristic_hatas(cities);
            break;
        case 4:
            algorithm.assign("CALLED 4| \"TSP with naive nearest neighbor\" (Parallel Implementation)");
            naive_heuristic_hatas_parallel(cities);
            break;
        case 5:
            algorithm.assign("CALLED 5| \"ACS TSP\"");
            ant_colony(cities);
            break;
        case 6:
            algorithm.assign("CALLED 6| \"ACS TSP\" (Parallel Implementation)");
            ant_colony_parallel(cities);
            break;
        default:
            std::cout << "[Warning]: Invalid algorithm setting\n\t|\"Unknown algorithm\" fault masked|" << std::endl;  /// Mask unknown algorithm fault
            break;
    }
    std::chrono::time_point<std::chrono::system_clock> end = std::chrono::system_clock::now();                      /// Declares an ending time point which helps in benchmarking
    std::chrono::duration<double> elapsed_seconds = end - start;                                                    /// Computes execution time
    std::cout << algorithm << " terminated after " << ITERATIONS << " * iterations with a total of " << elapsed_seconds.count() << " seconds" << std::endl;
                                                                                                                    /// Outputs results
    return 0;
}
```

Κώδικας 2.0.10: Η κεφαλίδα Interface.h

```h
/**
 * Interface.h
 *
 * In this header file, we define some functions which
 * implement a basic User Interface (UI) for the different
 * approaches to the TSP.
 */

#pragma once

#include "Common.h"

void print_tsp_tour_cost(const std::vector<long double>& cost);
void print_cities(const std::array<std::pair<int, int>, N_POINTS>& cities);
void print_row(const std::array<double, N_POINTS>& r);
void print_matrix(const std::vector<std::array<double, N_POINTS>>& matrix, const std::string matrix_name);
void print_ace_tsp(const std::vector<int>& tsp_route);
```

Κώδικας 2.0.11: Η κεφαλίδα Operation.h

```h
/**
 * Operation.h
 *
 * In this header file, we define some functions that
 * implement generic operations like complex comparison
 * on a container or vector normalization.
 */

#pragma once

#include "Common.h"

/**
 * Finds the lower bound from inside pair by the first pair element
 *
 * @param[in] value a std::pair container. Its first element will be compared with the second given parameter
 * @param[in] key this is the second parameter used for comparison
 *
 * @return boolean value that indicates equality or not
 *
 * @note This is the implementation for integer comparison
 *
 * @remark https://en.cppreference.com/w/cpp/algorithm/lower_bound
 */
struct compare_int
{
    bool operator()(const std::pair<int, int>& value, const int& key) { return (value.first << key); }
    bool operator()(const int& key, const std::pair<int, int>& value) { return (key << value.first); }
};

/**
 * Finds the lower bound from inside pair by the first pair element
 *
 * @param[in] value a std::pair container. Its first element will be compared with the second given parameter
 * @param[in] key this is the second parameter used for comparison
 *
 * @return boolean value that indicates equality or not
 *
 * @note This is the implementation for double precision number comparison
 *
 * @remark https://en.cppreference.com/w/cpp/algorithm/lower_bound
 */
struct compare_dbl
{
    bool operator()(const std::pair<int, double>& value, const double& key) { return (value.second < key); }
    bool operator()(const double& key, const std::pair<int, double>& value) { return (key < value.second); }
};


int find(const std::array<int, N_POINTS> non_explored, const int element);
bool sortDown_dbl(const std::pair<int, double>& a, const std::pair<int, double>& b);
bool sortDown_int(const std::pair<int, int>& a, const std::pair<int, int>& b);
int search(const std::vector<std::pair<int, double>> container, double value);
std::pair<int, double> normalized_value(std::pair<int, double> p, double min, double max);
void normalize(std::vector<std::pair<int, double>>& container);
void copy(const std::vector<std::array<double, N_POINTS>>& pheromone_matrix, const int col_idx, std::vector<double>& pheromone_column);
void filter(std::vector<std::pair<int, double>>& roulette, const std::vector<int>& tsp_route);
std::pair<int, double> flip(int i, double d);
```

Κώδικας 2.0.12: Το αρχείο Pheromone.cpp

```cpp
#include "Pheromone.h"

/**
 * Initializes a matrix.
 *
 * @param[in, out] pheromone_matrix the matrix given for initialization
 *
 * @note this is used to initialize the pheromone matrix
 */
void initialize_pheromone_matrix(std::vector<std::array<double, N_POINTS>>& pheromone_matrix)
{
    for (int i = 0; i < N_POINTS; i += 1)
    {
        pheromone_matrix.at(i).fill(1.0);     /// Fill the 'i'-th row of the matrix with ones. That way all nodes have an equal chance to be selected by any ant.
        pheromone_matrix.at(i).at(i) = 0.0;   /// Change the diagonal element to zero. That way we reinforce the ants not to chose the same node.
    }
}
```

Κώδικας 2.0.13: Το αρχείο Naive.cpp

```
#include "Naive.h"

/**
 * Selects an edge for the ant to follow
 * based on all precomputed evaluations.
 *
 * @param[in, out] evaluation the precomputed edge evaluations
 * @param[in] roulette_random the precomputed random selection probability
 *
 * @return the index of the next node for the ant to go
 *
 * @note this function calls modules from Operation.h
 */
int roulette_wheel(std::vector<std::pair<int, double>>& evaluation, double roulette_random)
{
    std::sort(evaluation.begin(), evaluation.end(), sortbysec_dbl);         /// Sorts all precomputed evaluations
    evaluation.erase(evaluation.begin(), evaluation.end() - ROULETTE_SIZE);  /// Reduces the amount of the evaluations
                                                                            /// deleting the less likely to be chosen.
                                                                            /// This marks computational error when the
                                                                            /// dataset is too large. It also makes the
                                                                            /// algorithm more deterministic.
    double accumulator = 0.0;                                               /// A variable used to accumulate the different edge evaluations for the roulette wheel
    for (int m = 0; m < evaluation.size(); m += 1)                          /// Reformat all evaluations to find the roulette wheel selection algorithm
    {
        double temp = evaluation.at(m).second;
        evaluation.at(m).second += accumulator;                             /// Accumulate previous variable value
        accumulator += temp;                                                /// Update accumulator
    }
    normalize(evaluation);                                                  /// Normalize the vector
    return search(evaluation, roulette_random);                             /// Find the selected next node based on the precomputed probability
}
```

Κώδικας 2.0.14: Η κεφαλίδα Naive.h

```
/**
 * Naive.h
 *
 * In this header file, we define a function that
 * implements the roulette wheel selection algorithm,
 * also known as the fitness proportionate selection.
 * It is an algorithm that is used to select an item
 * that corresponds to its probability or odds.
 */

#pragma once

#include "Common.h"
#include "Operation.h"

int roulette_wheel(std::vector<std::pair<int, double>>& evaluation, double roulette_random);
```

Κώδικας 2.0.15: Το αρχείο Operation.cpp

```
#include "Operation.h"

/**
 * Finds element inside an array.
 *
 * @param[in] non_explored the array to be parsed
 * @param[in] element the element to be found
 *
 * @return the index from the array associated with the given element
 *
 * @note there is a redundancy to trace the "Element Not Found" fault.
 */
int find(const std::array<int, N_POINTS> non_explored, const int element)
{
    std::array<int, N_POINTS>::const_iterator it = std::find(non_explored.begin(), non_explored.end(), element);
    if (it == non_explored.end())
    {
        std::cout << "[WARNING]: Element not found" << std::endl;
    }
    return it - non_explored.begin();
}

/**
 * Sorts a vector of pairs in ascending order by second element of pairs.
 *
 * @param[in] a the first pair
 * @param[in] b the second pair
 *
 * @return boolean that indicates if the first element was greater than the second or not
 *
 * @note Only second elements of the given pairs were compared
 *
 * @remark https://www.geeksforgeeks.org/sorting-vector-of-pairs-in-c-set-1-sort-by-first-and-second/
 */
bool sortbysec_dbl(const std::pair<int, double>& a, const std::pair<int, double>& b)
{
    return (a.second < b.second);
}

/**
 * Sorts a vector of pairs in ascending order by second element of pairs. This is a fork of the function sortbysec_dbl() implemented for integer comparison.
 *
 * @param[in] a the first pair
 * @param[in] b the second pair
 *
 * @return boolean that indicates if the first element was greater than the second or not
 *
 * @note Only second elements of the given pairs were compared
 *
 * @remark https://www.geeksforgeeks.org/sorting-vector-of-pairs-in-c-set-1-sort-by-first-and-second/
 */
bool sortbysec_int(const std::pair<int, int>& a, const std::pair<int, int>& b)
{
    return (a.second < b.second);
}

/**
 * Searches a vector of pairs for a value.
 *
 * @param[in] container the container to be searched
 * @param[in] value the value that we are trying to locate inside the container
 *
 * @return the index pointing to the element corresponding to the given value
 *
 * @note this function is called to find the proper element chosen by the roulette wheel selection algorithm
 *
 * @remark Naive.h
 */
int search(const std::vector<std::pair<int, double>> container, double value)
{
    std::vector<std::pair<int, double>>::const_iterator it = std::lower_bound(container.begin(), container.end(), value, compare_dbl());
    return container.at(it - container.begin())).first;
}

/**
 * Normalizes double values.
 *
 * @param[in] p the pair that we want to normalize
 * @param[in] min the minimum value found in the container
 * @param[in] max the maximum value found in the container
 *
 * @return the normalized pair
 *
 * @note only the second element of the container is to be normalized.
 *       This function is only called by function normalize() below.
 */
std::pair<int, double> normalized_value(std::pair<int, double> p, double min, double max) { return std::make_pair(p.first, (p.second - min) / (max - min)); }

/**
 * Normalizes a container.
 *
 * @param[in, out] evaluation the container to be normalized
 *
 * @remark https://en.cppreference.com/w/cpp/algorithm/transform
 */
void normalize(std::vector<std::pair<int, double>>& evaluation)
{
    double min = evaluation.at(0).second;
    double max = evaluation.at(evaluation.size() - 1).second;
    std::transform(evaluation.begin(), evaluation.end(), evaluation.begin(), std::bind(normalized_value, std::placeholders::_1, min, max));
}

/**
 * Copies column of a matrix.
 *
 * @param[in] pheromone_matrix the matrix from which data will be copied
 * @param[in] col_idx the index corresponding column from 'pheromone_matrix' to be copied
 * @param[in, out] pheromone_column the vector in which data will be copied
 */
void copy(const std::vector<std::array<double, N_POINTS>>& pheromone_matrix, const int col_idx, std::vector<double>& pheromone_column)
{
    pheromone_column.reserve(N_POINTS);
    for (int j = 0; j < N_POINTS; j += 1)
    {
        pheromone_column.emplace_back(pheromone_matrix.at(j).at(col_idx));
    }
}

/**
 * Custom vector filter.
 *
 * @param[in, out] roulette the vector to be filtered
 * @param[in] tsp_route the vector based on which elements will be deleted
 *
 * @note this is called to filter the roulette vector based on current tsp route. This prevents node repetition throughout the route
 */
void filter(std::vector<std::pair<int, double>>& roulette, const std::vector<int> tsp_route)
{
    for (int i = 0; i < tsp_route.size(); i += 1)
    {
        for (int j = 0; j < roulette.size(); j += 1)
        {
            if (tsp_route.at(i) == roulette.at(j).first)
            {
                roulette.erase(roulette.begin() + j);       /// Deletes any element found in tsp_route.
            }                                               /// That way the ants are encouraged to select unexplored nodes.
        }
    }
}
/**
 * This function is called to bind an integer and a double into an std::pair
 *
 * @param[in] i the integer variable
 * @param[in] d the double variable
 *
 * @return the pair containing the given variables
 */
std::pair<int, double> f(int i, double d) { return std::make_pair(i, d); }
```

Κώδικας 2.0.16: Η κεφαλίδα Pheromone.h

```
/**
 * Pheromone.h
 *
 * In this header file, we define a function that
 * executes only in the beginning of the ACO algorithm.
 * This function is used to initialize the pheromone
 * matrix that the ACO will use to colonize a map.
 */

#pragma once

#include "Common.h"

void initialize_pheromone_matrix(std::vector<std::array<double, N_POINTS>>& pheromone_matrix);
```

Κώδικας 2.0.17: Η κεφαλίδα Validation.h

```
/**
 * Validation.h
 *
 * In this header file, we define some functions
 * used to validate the data computed by the different
 * algorithms using an external program.
 *
 * @note The external program can be found at https://github.com/andreasvc/tsp2hatsu#ha
 */

#pragma once

#include "Common.h"

void export_graph_newtork_array(const std::array<std::pair<int, int>, N_POINTS> cities, std::string filename);
void export_ace_tsp_route(const std::array<std::pair<int, int>, N_POINTS> cities, std::string filename, const std::vector<int> tsp_route);
```

Κώδικας 2.0.18: Το αρχείο Validation.cpp

```
#include "Validation.h"

/**
 * Exports nodes of the computed TSP route to CSV file.
 *
 * @param[in] cities the vector with the ordered cities that is to be exported
 * @param[in] filename the name of the CSV file
 *
 * @note this function is called by:
 *          - the Naive TSP algorithm
 *          - the Naive TSP algorithm (Parallel Implementation)
 *          - the TSP with nearest neighbor algorithm
 *          - the TSP with naive nearest neighbor algorithm
 *          - the TSP with naive nearest neighbor algorithm (Parallel Implementation)
 *
 * @note for the module to work, there must exist a directory called data with the same parent
 *       directory with the project. If you want to store the data to a custom directory, then
 *       change the proper line of code.
 */
void export_graph_newtork_array(const std::array<std::pair<int, int>, N_POINTS> cities, std::string filename)
{
    std::ofstream export_stream;
    export_stream.open("./data/" + filename + ".csv");      /// If you want to store the data into a different directory change "./data/" to whatever you want
    for (int i = 0; i < cities.size(); i += 1)
    {
        export_stream << cities.at(i).first << "," << cities.at(i).second << std::endl;
    }
    export_stream.close();
}

/**
 * Exports nodes of the computed TSP route to CSV file.
 *
 * @param[in] cities the container that holds the dataset
 * @param[in] filename the name of the CSV file
 * @param[in] tsp_route the container with the ordered indexes of 'cities' computed by the ACO to be exported
 *
 * @note this function is called by:
 *          - the ACO TSP algorithm
 *          - the ACO TSP algorithm (Parallel Implementation)
 *
 * @note for the module to work, there must exist a directory called data with the same parent
 *       directory with the project. If you want to store the data to a custom directory, then
 *       change the proper line of code.
 */
void export_ace_tsp_route(const std::array<std::pair<int, int>, N_POINTS> cities, std::string filename, const std::vector<int> tsp_route)
{
    std::ofstream export_stream;
    export_stream.open("./data/" + filename + ".csv");      /// If you want to store the data into a different directory change "./data/" to whatever you want
    for (int i = 0; i < tsp_route.size(); i += 1)
    {
        export_stream << cities.at(tsp_route.at(i)).first << "," << cities.at(tsp_route.at(i)).second << std::endl;

    }
    export_stream.close();
}
```

Κώδικας 2.0.19: Το αρχείο Utilities.cpp

```cpp
#include "Utilities.h"

/**
 * Implements Naive TSP.
 *
 * @param[in, out] cities
 *
 * @note the TSP route is actually generated connecting the different
 *      nodes that are found inside the `cities` variable. Therefore,
 *      to update the TSP route, if a better order of those cities is
 *      found, then we swap the order of the cities to fit to the better
 *      cities order.
 *
 * @remark Naive TSP:
 *      - Select 2 random points (cities)
 *      - Compute current TSP tour cost
 *      - Permutate those cities
 *      - Recompute the TSP tour cost
 *      - If the later cost is less, then keep the later TSP route
 *      - Else change it back the way it was
 */
void naive_tsp(std::array<std::pair<int, int>, N_POINTS>& cities)
{
    std::random_device rd;                                  /// non-deterministic generator
    std::mt19937 gen(rd());                                 /// to seed mersenne twister
    std::uniform_int_distribution<int> dist(1, N_POINTS - 2);   /// distribute results between 0 and N_POINTS - 2 squared inclusive

    for (int i = 0; i < ITERATIONS; i += 1)
    {
        int point_index_one = dist(gen);                    /// Select the first point for the algorithm
        int point_index_two = dist(gen);                    /// Select the second point for the algorithm
        int is_neighbor = (std::abs(point_index_one - point_index_two) == 1 ? 1 : 0);
                                                            /// Set the neighbor flag with respect to the selected nodes
        if(is_neighbor == 1 && point_index_one > point_index_two) { std::swap(point_index_one, point_index_two); }
                                                            /// Sort points in ascending order if those are neighbors to solve fault in distance computation
        long double diff_before = euclidean_difference(is_neighbor,
            cities.at(point_index_one - 1),
            cities.at(point_index_one),
            cities.at(point_index_one + 1),
            cities.at(point_index_two - 1),
            cities.at(point_index_two),
            cities.at(point_index_two + 1));                /// Compute the TSP tour cost before permutation
        long double diff_after = euclidean_difference(is_neighbor,
            cities.at(point_index_one - 1),
            cities.at(point_index_two),
            cities.at(point_index_one + 1),
            cities.at(point_index_two - 1),
            cities.at(point_index_one),
            cities.at(point_index_two + 1));                /// Compute the TSP tour cost after the permutation
        if (diff_before > diff_after)                       /// Compare the two computed costs
        {
            std::swap(cities.at(point_index_one), cities.at(point_index_two));
                                                            /// Change the order of the cities if random permutation gave better results
            if (TEST_MODE)
            {
                std::cout << "ITERATION [ " << i << " ]\t";     /// If in debug mode, print out some information on the algorithm's progress
                print_tsp_tour_cost(tsp_tour_cost(cities));
                export_graph_newtork_array(cities, "graph" + std::to_string(i));
                                                            /// Export the new order of nodes to a CSV file
            }
        }
    }
}

/**
 * Implements Naive TSP. This is a fork of the `naive_tsp` function above, parallelized with OpenMP 4.0.
 *
 * @param[in, out] cities
 */
void naive_tsp_parallel(std::array<std::pair<int, int>, N_POINTS>& cities)
{
    std::random_device rd;                                  /// non-deterministic generator
    std::mt19937 gen(rd());                                 /// to seed mersenne twister
    std::uniform_int_distribution<int> dist(1, N_POINTS - 2);   /// distribute results between 0 and N_POINTS - 2 inclusive

    std::random_device perm_rd;                             /// non-deterministic generator
    std::mt19937 perm_gen(perm_rd());                       /// to seed mersenne twister
    std::uniform_real_distribution<float> perm_dist(0, 1);      /// distribute results between 0 and 1 inclusive

    for (int i = 0; i < ITERATIONS; i += 1)                 /// Precompute around {`N_POINTS` divided by 2} permutations
    {
        std::vector<int> permutations;
        permutations.reserve((int)(N_POINTS / 2));
        float const_prob = 0.0;
        for (int j = 1; j < N_POINTS - 2; j += 1)
        {
            if (perm_dist(perm_gen) - const_prob > NAIVE_PROBABILITY)
            {
                permutations.push_back(j);                  /// If a node is chosen to be added in the permutation vector
                const_prob = 1.0;                           /// then make sure not to chose a neighbor, since the permutations will be executed
            }                                               /// in parallel and there is a chance of a false distance calculation
            else
            {
                const_prob = 0.0;
            }
        }

        std::shuffle(permutations.begin(), permutations.end(), std::mt19937{ std::random_device{}() });
                                                            /// Shuffle the permutation vector precomputed above to generate random pairs for permutation
#pragma omp parallel for num_threads(N_THREADS) schedule(runtime)
        for (int k = 0; k < permutations.size() - 1; k += 2)
        {
            int point_index_one = permutations.at(k);
            int point_index_two = permutations.at(k + 1);
            int is_neighbor = (std::abs(point_index_one - point_index_two) == 1 ? 1 : 0);
            long double diff_before = euclidean_difference(is_neighbor,
                cities.at(point_index_one - 1),
                cities.at(point_index_one),
                cities.at(point_index_one + 1),
                cities.at(point_index_two - 1),
                cities.at(point_index_two),
                cities.at(point_index_two + 1));
            long double diff_after = euclidean_difference(is_neighbor,
                cities.at(point_index_one - 1),
                cities.at(point_index_two),
                cities.at(point_index_one + 1),
                cities.at(point_index_two - 1),
                cities.at(point_index_one),
                cities.at(point_index_two + 1));
            if (diff_before > diff_after)
            {
                std::swap(cities.at(point_index_two), cities.at(point_index_one));
                if (TEST_MODE)
                {
#pragma omp critical
                    {
                        print_tsp_tour_cost(tsp_tour_cost(cities));
                        export_graph_newtork_array(cities, "graph" + std::to_string(i));
                    }
                }
            }
        }
    }
}

/**
 * Implements TSP with nearest neighbor, or
 * as the project states the Heinritz - Hsiao algorithm.
 *
 * @param[in, out] cities the dataset which is to be optimized
 *
 * @remark Heinritz - Hsiao:
 *      - Place the travelling salesman in a city
 *      - Repeat:
 *      -   Find the closest city to the city the salesman is at
 *      -   Go to that city
 *      - Until all cities have been explored
 */
void heinritz_hsiao(std::array<std::pair<int, int>, N_POINTS>& cities)
{
    for (int i = 0; i < N_POINTS - 1; i += 1)
    {
        long double min_val = std::numeric_limits<double>::infinity();
        int min_idx = -1;
        for (int j = i + 1; j < N_POINTS; j += 1)          /// Iterate through all possible cities and find the one with the minimum Euclidean distance
        {
            long double cost = tsp_hop_cost(cities.at(i), cities.at(j)).second;
            if (cost < min_val)                             /// If a (local) minimum is found, update indicators
            {
                min_idx = j;
                min_val = cost;
            }
        }
        std::swap(cities.at(i + 1), cities.at(min_idx));    /// Swap the city order with respect to the selected city
        if (TEST_MODE)                                      /// If in debug mode, print out some information on the algorithm's progress
        {
            std::cout << "[ITERATION " << i << "]";
            print_tsp_tour_cost(tsp_tour_cost(cities));
            export_graph_newtork_array(cities, "graph" + std::to_string(i));
        }
    }
}

/**
 * Implements TSP with naive nearest neighbor.
 *
 * @param[in, out] cities the dataset which is to be optimized
 *
 * @remark Naive Heinritz - Hsiao:
 *      - Place the travelling salesman in a city
 *      - Repeat:
 *      -   Find the 2 closest cities to the city the salesman is at
 *      -   Randomly one of those cities
 *      -   Go to that city
 *      - Until all cities have been explored
 */
void naive_heinritz_hsiao(std::array<std::pair<int, int>, N_POINTS>& cities)
{
    std::random_device naive_rd;                            /// non-deterministic generator
    std::mt19937 naive_gen(naive_rd());                     /// to seed mersenne twister
    std::uniform_real_distribution<double> naive_dist(0, 1);    /// distribute results between 0 and 1 inclusive
    for (int i = 0; i < N_POINTS - 2; i += 1)
    {
        std::array<int, 2> neighbors_idx;                   /// Declare an array to store the indexes corresponding to the two closest cities
        long double min_val = std::numeric_limits<double>::infinity();
        for (int j = i + 1; j < N_POINTS; j += 1)
        {
            long double cost = tsp_hop_cost(cities.at(i), cities.at(j)).second;
            if (cost < min_val)                             /// If a (local) minimum is found, then update `neighbors_idx` array
            {
                min_val = cost;
                std::swap(neighbors_idx.at(0), neighbors_idx.at(1));
                neighbors_idx.at(0) = j;
            }
        }
        std::swap(cities.at(i + 1), cities.at(neighbors_idx.at((NAIVE_PROBABILITY > naive_dist(naive_gen) ? 1 : 0))));
                                                            /// Swap the city order with respect to the two closest cities given a `NAIVE_PROBABILITY`
        if (TEST_MODE)                                      /// If in debug mode, print out some information on the algorithm's progress
        {
            std::cout << "[ITERATION " << i << "]";
            print_tsp_tour_cost(tsp_tour_cost(cities));
            export_graph_newtork_array(cities, "graph" + std::to_string(i));
        }
    }
}

/**
 * Implements TSP with naive nearest neighbor. This is a fork of the `naive_heinritz_hsiao` function above, parallelized with OpenMP 4.0.
 *
 * @param[in, out] cities the dataset which is to be optimized
 */
void naive_heinritz_hsiao_parallel(std::array<std::pair<int, int>, N_POINTS>& cities)
{
    std::random_device naive_rd;                            /// non-deterministic generator
    std::mt19937 naive_gen(naive_rd());                     /// to seed mersenne twister
    std::uniform_real_distribution<double> naive_dist(0, 1);    /// distribute results between 0 and 1 inclusive
    for (int i = 0; i < N_POINTS - 2; i += 1)
    {
        std::array<std::array<int, 2>, N_THREADS> neighbors_idx;
        std::array<std::array<long double, 2>, N_THREADS> neighbors_val;
        for (int j = 0; j < N_THREADS; j += 1)
        {
            neighbors_idx.at(j).fill(i + 1);
            neighbors_val.at(j).fill(std::numeric_limits<long double>::infinity());
        }
#pragma omp parallel for num_threads(N_THREADS) schedule(runtime)
        for (int j = i + 1; j < N_POINTS; j += 1)          /// Each threads finds its own pair of closest cities
        {
            long double cost = tsp_hop_cost(cities.at(i), cities.at(j)).second;
            if (cost < neighbors_val.at(omp_get_thread_num()).at(1))
            {
                if (cost < neighbors_val.at(omp_get_thread_num()).at(0))
                {
                    neighbors_idx.at(omp_get_thread_num()).at(0) = j;
                    std::swap(neighbors_idx.at(omp_get_thread_num()).at(0), neighbors_idx.at(omp_get_thread_num()).at(1));
                    neighbors_val.at(omp_get_thread_num()).at(0) = cost;
                    std::swap(neighbors_val.at(omp_get_thread_num()).at(0), neighbors_val.at(omp_get_thread_num()).at(1));
                }
                else
                {
                    neighbors_idx.at(omp_get_thread_num()).at(1) = j;
                    neighbors_val.at(omp_get_thread_num()).at(1) = cost;
                }
            }
        }
        std::array<int, 2> reduced_cost_idx;
        std::array<long double, 2> reduced_cost_val;
        reduced_cost_val.fill(std::numeric_limits<long double>::infinity());
        for (int j = 0; j < N_THREADS; j += 1)             /// When all threads are done comparing distances, this final comparison is performed.
        {                                                   /// The difference is that now, only N_THREADS x 2 values have to be compared, not N_POINTS x 2.
            for (int k = 0; k < neighbors_val.at(j).size(); k += 1)
            {
                if (neighbors_val.at(j).at(k) < reduced_cost_val.at(1) && neighbors_val.at(j).at(k) > 0.0)
                {                                           /// Update `reduced_cost_idx`, which is the container with the final 2 closest cities
                    if (neighbors_val.at(j).at(k) < reduced_cost_val.at(0))
                    {                                       /// A better minimum was found
                        std::swap(reduced_cost_idx.at(0), reduced_cost_idx.at(1));
                        reduced_cost_idx.at(0) = neighbors_idx.at(j).at(k);
                        std::swap(reduced_cost_val.at(0), reduced_cost_val.at(1));
                        reduced_cost_val.at(0) = neighbors_val.at(j).at(k);
                    }
                    else
                    {                                       /// A second better minimum was found
                        reduced_cost_idx.at(1) = neighbors_idx.at(j).at(k);
                        reduced_cost_val.at(1) = neighbors_val.at(j).at(k);
                    }
                }
            }
        }
        std::swap(cities.at(i + 1), cities.at(reduced_cost_idx.at((NAIVE_PROBABILITY > naive_dist(naive_gen) ? 1 : 0))));
                                                            /// Update the city order with respect to the `reduced_cost_idx`
        if (TEST_MODE)                                      /// If in debug mode, print out some information on the algorithm's progress
        {
            std::cout << "[ITERATION " << i << "][SECOND HIGHEST COST " << reduced_cost_val.at((NAIVE_PROBABILITY > naive_dist(naive_gen) ? 1 : 0)) << "]\t";
            print_tsp_tour_cost(tsp_tour_cost(cities));
            export_graph_newtork_array(cities, "graph" + std::to_string(i));
        }
    }
}

/**
 * Implements ACS.
 *
 * param[in] cities the dataset of the random cities to be explored
 *
 * @note the ACS requires a huge ammount of memory. Only the pherormone matrix
 *          requires 100 million double values to be stored, which is around 850 MB
 *          of memory (according to the Visual Studio Memory Profiler). Then, there
 *          is the `non_explored` vector which is necessary since it keeps a track of
 *          each ant's explored nodes.
 *
 * @remark Ant colony:
 *          - Place N_ANTS in random cities
 *          - For each ant repeat:
 *          -   Evaluate all the possible edges
 *          -   Fetch each edge's pherormone
 *          -   Select an edge using Roulette Wheel
 *          -   Leave some pherormone on that edge
 *          - Until ANT_MEMORY is exceeded
 *
 * @remark https://staff.washington.edu/paymana/swarm/stutzle99-eaecs.pdf
 *
 * @remark https://youtu.be/783ZtAF4j5g
 */
void ant_colony(const std::array<std::pair<int, int>, N_POINTS> cities)
{
    std::vector<std::array<double, N_POINTS>> pherormone_matrix(N_POINTS);
                                                            /// Declare the pherormone matrix of dimentions N_POINTS x N_POINTS
    initialize_pherormone_matrix(pherormone_matrix);        /// Initialize the pherormone matrix
    std::vector<int> tsp_route;                             /// Initialize a variable to store the TSP route found by the ants
    tsp_route.reserve(N_POINTS);                            /// Reserve `N_POINTS` of memory slots to increase performance
    colonize(cities, pherormone_matrix);                    /// Colonize the map
    if (TEST_MODE)                                          /// If in debug mode, print out some information on the algorithm's progress
    {
        print_matrix(pherormone_matrix, "Pherormone Matrix");
        acs_tsp(pherormone_matrix, tsp_route);
        double acs_cost = acs_tsp_cost(tsp_route, cities);
        print_acs_tsp(tsp_route);
        std::cout << "\tCost estimated by \"ACS\": " << acs_cost << std::endl;
        export_acs_tsp_route(cities, "acs_tsp", tsp_route);
    }
}

/**
 * Implements ACS. This is a fork of the `ant_colony` function above, parallelized with OpenMP 4.0.
 *
 * param[in] cities the dataset of the random cities to be explored
 *
 * @note the parallel version of the ACS requires an even greater ammount of memory
 *          to be reserved. For 10,000 cities, the ammount of memory reserved was
 *          12 GB. That is why the number of cities, which is declared by `N_POINTS`
 *          is better to be less than 2,000 when running ACS.
 */
void ant_colony_parallel(const std::array<std::pair<int, int>, N_POINTS> cities)
{
    std::vector<std::array<double, N_POINTS>> pherormone_matrix(N_POINTS);
    initialize_pherormone_matrix(pherormone_matrix);
    std::vector<int> tsp_route;
    tsp_route.reserve(N_POINTS);
    colonize_parallel(cities, pherormone_matrix);           /// Call the parallel version of the colonize() function
    if (TEST_MODE)
    {
        print_matrix(pherormone_matrix, "Pherormone Matrix");
        acs_tsp(pherormone_matrix, tsp_route);
        double acs_cost = acs_tsp_cost(tsp_route, cities);
        print_acs_tsp(tsp_route);
        std::cout << "\tCost estimated by \"ACS\": " << acs_cost << std::endl;
        export_acs_tsp_route(cities, "acs_tsp", tsp_route);
    }
}
```