Traveling C-AG

Technische Dokumentation

Conradi Caspar

Follner Andreas

Grill Tobias

Inhalt

Aufgabenstellung

Das Team wird von einer Speditionsfirma beauftragt, die Route Ihrer LKW-Flotte zu organisieren. Zur Verfügung steht eine CSV Datei mit folgenden Städteinformationen:

1. Name der Stadt
2. Einwohnerzahl
3. Koordinaten
4. Ländername
5. ID-Nummer
6. Städtekürzel
7. Etc.

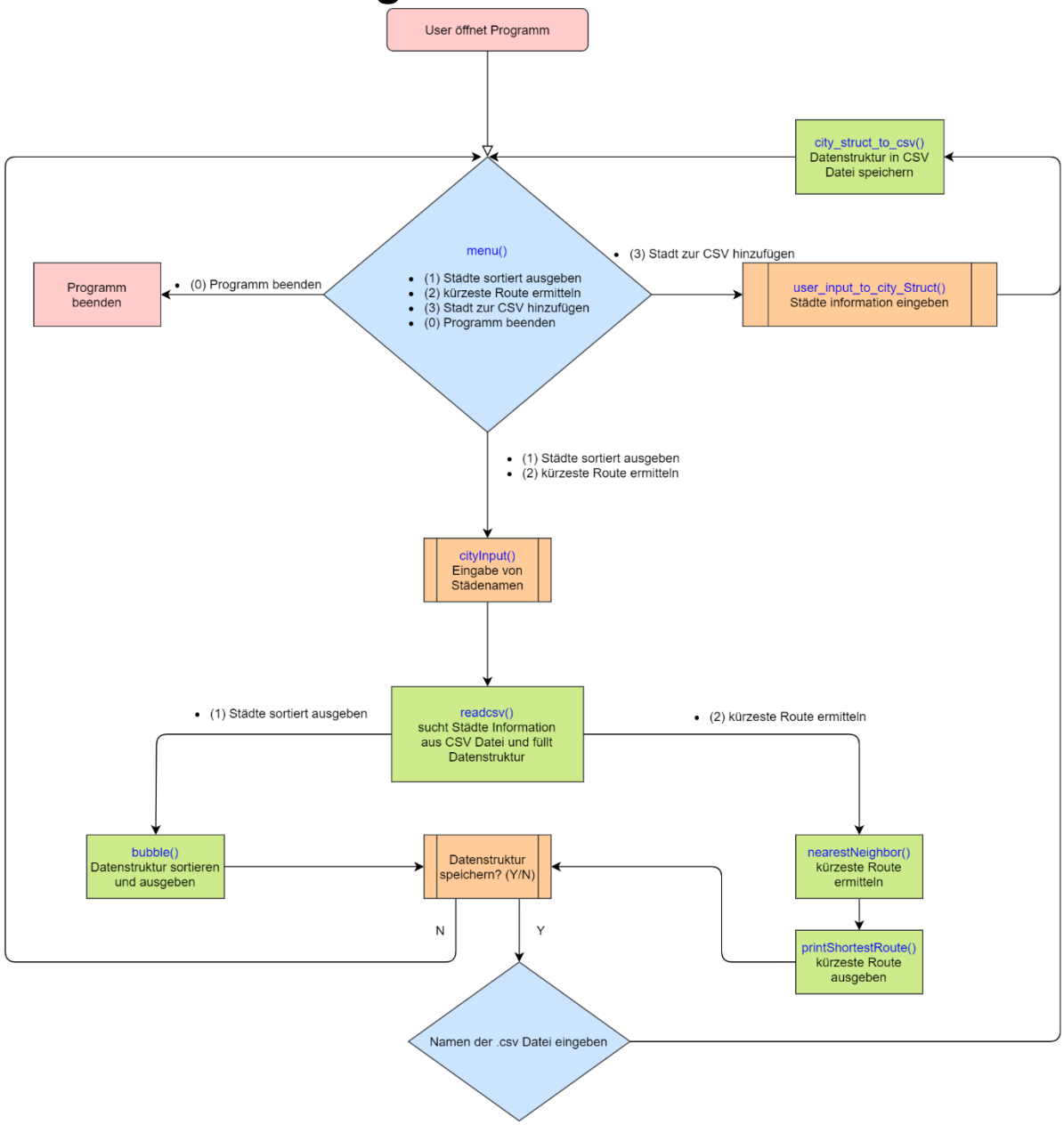
Der Benutzer soll die Möglichkeit haben die Städte sowohl Alphabetisch als auch nach Einwohnerzahl sortieren zu können.

Zusätzlich soll die kürzeste Route zwischen eingegebenen Städten ermittelt werden, ausgehend von der Zentrale in Wien.

Die ermittelte Route und die sortierten Städte soll man in eine neue CSV Datei abspeichern können.

Des Weiteren soll es möglich sein, Städte in der bereitgestellten Datei zu ergänzen.

Tatsächliche Programmstruktur



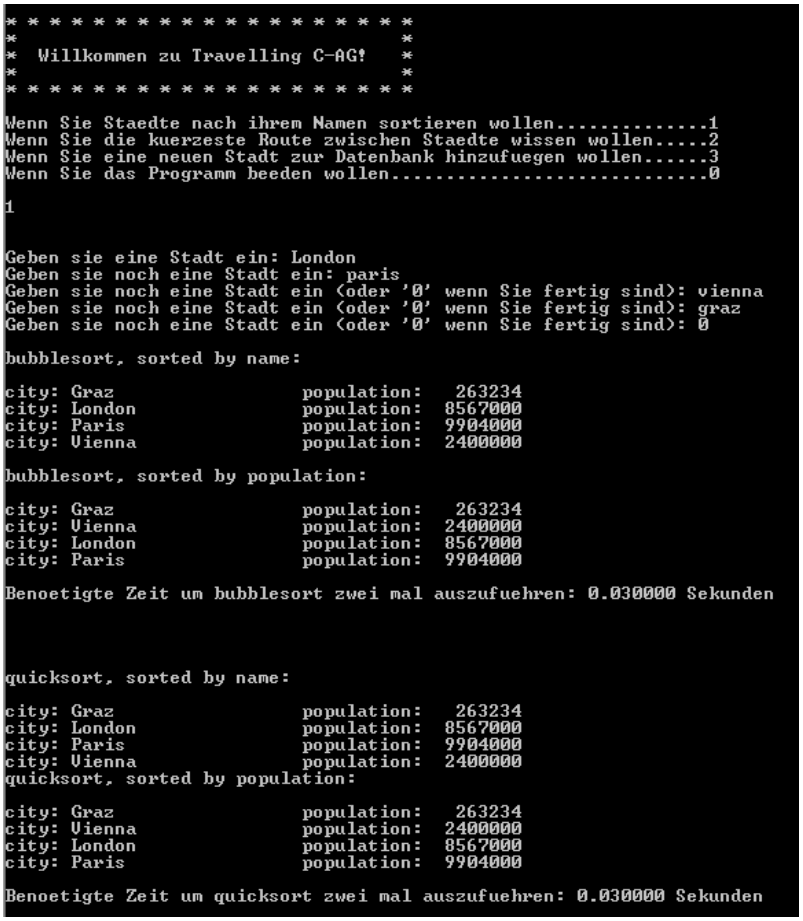
Beim Starten des Programm erscheint ein Menü, bei dem der Benutzer nach der gewünschten Aktion gefragt wird.

```
* * * * * * * * * * * * * * * * * * * *
*                                       *
*  Willkommen zu Travelling C-AG!       *
*                                       *
* * * * * * * * * * * * * * * * * * * *

Wenn Sie Staedte nach ihrem Namen sortieren wollen.............1
Wenn Sie die kuerzeste Route zwischen Staedte wissen wollen.....2
Wenn Sie eine neuen Stadt zur Datenbank hinzufuegen wollen......3
Wenn Sie das Programm beeden wollen.............................0
```

Auswahl 1 – Städte sortieren

Der Benutzer kann nun eine Liste von Städten eingeben. Sobald er die Eingabe mit 0 beendet, wird werden die Städte einmal nach Namen und einmal nach Einwohnerzahl sortiert ausgegeben. Beide Sortierungen werden einmal mithilfe des „bubblesort“ und einmal mithilfe des „quicksort“ Algorithmus durchgeführt. Außerdem wird für beide Algorithmen die benötigte Zeit erfasst und ausgegeben.



Das Abspeichern der sortierten Liste ist im Anschluss möglich.

Auswahl 2 – kürzeste Route

Der Benutzer kann nun Städte eingeben. Ausgehend von der Zentrale wird die kürzeste Route ermittelt und auf der Konsole ausgegeben.

Das Abspeichern der Route ist im Anschluss möglich.

```
Geben sie eine Stadt ein: graz
Geben sie noch eine Stadt ein: linz
Geben sie noch eine Stadt ein (oder '0' wenn Sie fertig sind): berlin
Geben sie noch eine Stadt ein (oder '0' wenn Sie fertig sind): 0



#Kuerzeste Route: Zentrale  => Graz  => Linz  => Berlin  => Zentrale
```

Auswahl 3 – neue Stadt zur Datenbank hinzufügen

> Der Benutzer kann hier die bestehende Liste der Städte beliebig erweitern. Dazu müssen alle Daten eingegeben werden (Name, Einwohner, Koordinaten, etc.)

Auswahl 0 – Programm Ende

Lösungsansatz

Um alle Daten der eingegebenen Städte im Programm verarbeiten zu können, wurde ein struct angelegt, wo alle Informationen zur Verfügung stehen.

```
typedef struct
{
    char city[50];
    char city_ascii[50];
    float lat;
    float lng;
    char country[50];
    char iso2 [3];
    char iso3 [4];
    char admin_name[50];
    char capital[50];
    long population;
    long id;

}cityTemp;
```

Als erstes wollten wir die Städte in ein statisches Array speichern. Allerdings war damit die Eingabe limitiert und wenn weniger Städte eingegeben werden, wurde unnötig Speicher belegt.

Daher wird nun der Speicher dynamisch allokiert, somit kann eine beliebige Anzahl von Städten gespeichert werden und es wird nur so viel Speicher belegt wie notwendig.

```
cityTemp *citys = malloc(sizeof(cityTemp));
```

Bei jeder Eingabe wird der Speicher um den benötigten Speicherplatz erweitert.

```
citys = realloc(citys, 5*sizeof(cityTemp));
```

Somit stehen alle benötigten Daten zur weiteren Bearbeitung zur Verfügung. Die Beschreibung der einzelnen Funktionen folgt weiter unten.

Rollenbeschreibung

Teamleiter – Follner Andreas

Aufgaben:

- Eingabe von Städten
- Sucht Städte Namen in CSV Datei und füllt Datenstruktur mit der Information.
- Städte nach Einwohnerzahl und Namen sortieren.

Conradi Caspar

Aufgaben:

- Menü
- Städte zur CSV Datei hinzufügen
- Daten ausgeben
- Daten in neue CSV Datei speichern
- Testprogramme schreiben

Grill Tobias

Aufgaben:

- Kürzeste Route zwischen Städten berechnen (Traveling Salesman Problem, TSP)
- Route ausgeben

Zeitaufwand

Geplant:

Was	Wann	Dauer	Wer
Menü	Mo 18.Mai	1h	Caspar
Städte zur CSV hinzufügen	Sa 30.Mai	6h	Caspar
Eingabe von Städte Namen	Sa 30.Mai	2h	Andi
sucht Städte Namen in CSV Datei und füllt Datenstruktur mit der Information	Sa 30.Mai	10h	Andi
Datenstruktur nach Namen oder Einwohnerzahl sortieren	Sa 30.Mai	14h	Andi
Datenstruktur nach kürzester Route sortieren (Eröfnugsverfahren)	Sa 30.Mai	12h	Grü
Datenstruktur nach kürzester Route sortieren (Verbesserungsverfahren)	Sa 30.Mai	14h	Grü
Datenstruktur ausgeben	Mo 18.Mai	2h	Caspar
Datenstruktur in neuer CSV speichern	Sa 30.Mai	6h	Caspar
Testprogramme schreiben	Sa 30.Mai	8h	Caspar
Libarys und Git verwenden		8h	Caspar, Grü, Andi

Tatsächlicher Zeitaufwand:

Teilaufgabe	Zeitaufwand in Std.
Hauptmenü	8
CSV Datei einlesen	12
TSP	8
Sortieralgorithmen	16
Libraries implementieren	6
Git	7
CMAKE	5
Daten in CSV speichern	4
Dokumentation	4
SUMME	70

Caspar hat vor allem mehr Zeit damit verbracht die Struktur mit Git, den Librarys, Cmake und den header files aufzubauen als erwartet. Da er bei den tatsächlichen Programmieraufgaben außer dem dynamisch allokierten Speicher nicht viele neue Sachen verwendete konnte er hier etwas Zeit einsparen.

Git

Zur Versionsverwaltung wurde git verwendet. Da wir git bis zu diesem Projekt nie verwendeten, war es von beginn an schwer damit zu arbeiten und sich einzulesen.

Daher und wegen mangelnder Zeit haben wir auch nicht den vollen Umfang dieses Tools nutzen können. Gerade zu Beginn brauchten wir lange, um herauszufinden wie der optimale Workflow aussieht.

Aber auch während dem gesamten Zeitraum des Projekts traten immer wieder Schwierigkeiten auf. Da wir Git-hub eher als Kollaborierungssoftware verwendeten als als VCS hat wir mit unserem Nutzen teilweise den Sinn des Programmes

ausgehebelt.
Hauptsächlich kam es zu merge Konflikten, die wir oft nicht oder nur schwer beheben konnten. Ein Grund dafür war auch, dass wir wahrscheinlich zu viel (nicht relevante) Dateien über Git teilten.

Librarys

Um auch mögliche merge Konflikte bei Git zu vermeiden, haben wir für jedes Teammitglied eine Library erstellt. Das zusammenführen, bzw. das Einbinden in das Hauptprogramm erwies sich als herausfordernder als gedacht.

Es traten ständig Fehler auf, die wir uns nicht erklären konnten. Zum Beispiel wurden Funktionen nicht mehr gefunden nach kleinen Änderungen, auch wenn das Programm davor noch fehlerfrei lief. Oder es wurden Änderungen nicht übernommen, obwohl die Library neu kompiliert wurde.

Da wir für diese Probleme keine Lösung gefunden haben, bzw. nicht einmal den Ursprung der Fehler finden konnten, haben wir im weiteren Verlauf alle Funktionen in ein Programm, ohne Librarys geschrieben.

Als das Programm lief, wurde eine Library mit allen Funktionen erstellt und mittels header Datei ins Programm eingebunden.

Daten einlesen

CityInput

Die Aufgabe dieser Funktion besteht darin vom User Städtenamen einzulesen. Dafür wird am heap Speicher allokiert, und zwar in der Größe des zuvor deklarierten Structs. Den Pointer zu diesem Strukt bekommt er bereits aus der main() und vergrößert den Speicher mit realloc. Im zweiten Schritt gibt er den Pointer an die Funktion readcsv() weiter die dafür zuständig ist das restliche Struct mit der Information aus der .csv Datei aufzufüllen. Diese zwei schritte werden in einer endlosschleife solange wiederholt bis die User eine Null eingibt. Der Pointer zu dem dynamisch vergrößerte Speicher wird nun wieder an die main() zurück gegeben. Ursprünglich haben wir gedacht, dass sich der Pointer zu diesem Speicher ja nicht ändert, da sich aber dies Funktion schließt wird auch der Speicher frei gegeben und würde ohne dem return value nicht mehr zu Verfügung stehen.

Readcsv

In der Funktion readcsv() wird die Datei „worldcities.csv“ geöffnet und die Daten zeilenweise in einen Puffer-String eingelesen. Der Puffer-String wird mithilfe des Trennzeichens („) in Teilstrings unterteilt und ein Zeiger wird auf den ersten Teilstring gesetzt. Nun wird in einer Schleife ob der aktuelle Teilstring Informationen enthält oder ob er leer ist. Falls er Informationen enthält werden diese in das vorher angelegten, temporären struct kopiert. Es wird davon ausgegangen, dass die in der

CSV Datei eingetragenen Daten in der richtigen Reihenfolge stehen. Sobald die Zeile vollständig verarbeitet wurde, wird überprüft ob der vom Benutzer eingegebene Stadtname dem des gerade eingelesenen Datensatzes entspricht. Um die Eingabe für den Benutzer zu erleichtern, wird sowohl der eingegebene, als auch beim ausgelesene Stadtname auf Großbuchstaben gesetzt. Falls die Namen übereinstimmen, wird der Datensatzes in das dynamisch generierte Array kopiert. Die Funktion hat keinen Rückgabewert.

Daten speichern

user_input_to_city_Struct

In dieser Funktion hat der User die Möglichkeit neue Städte zu der .csv Datei hinzufügen. Er wird zur gesamten Information der Städte abgefragt. Dafür muss er auch genau wissen welche Information gefragt ist damit er zB. bei der Einwohnerzahl nicht ein Wort eingibt. Die würde natürlich bei einem kommerziellen Programm schnell zu Komplikationen führen.

Traveling Salesman Problem

Das traveling Salesman Problem beschäftigt sich mit der Berechnung der kürzesten Route zwischen gegebenen Städten. Um diese Route zu ermitteln und auszugeben wurden zwei Funktionen implementiert.

nearestNeighbour

Die Funktion nearestNeighbour() berechnet die Route. Dazu wird ein Pointer übergeben, der auf einen Counter zeigt, wie viel Städte zu sortieren sind. Und ein zweiter Pointer der auf das Array mit den Städten zeigt.

Als erstes wird ein weiteres Array erstellt, in dem die Route gespeichert wird.

Im ersten Schritt der Berechnung, wird die nächste Stadt zur Zentrale in Wien ermittelt. Dazu müssen die Längen und Breitengrade in km errechnet werden.

Steht fest welche Stadt welche Stadt der Zentrale am nächsten liegt, wird sie im Array der sortierten Liste auf Platz [0] gesetzt und die id der Stadt im unsortierten Array auf 0, damit wird später überprüft ob diese Stadt bereits angefahren wurde.

Nun wird mit einer versschachtelten for Schleife die Route ermittelt. Dazu wird immer die Entfernung von der aktuellen Stadt zu jeder anderen gegebenen Stadt verglichen. Die nächste davon, wird in dem Array der sortierten Städte auf den nächsten Platz gesetzt und ist der Ausgangspunkt für die nächste Berechnung. Außerdem wird die id der Stadt auf 0 gesetzt, um zu überprüfen ob eine Stadt bereits angefahren wurde.

Ist die kürzeste Route ermittelt, wird der Speicher des unsortierten Arrays freigegeben, und der Pointer zu dem sortierten Array wird zurückgegeben.

printShortesRoute

Diese Funktion ist lediglich für die Ausgabe der kürzesten Route zuständig. Dazu wird ein Pointer auf ein Array übergeben und ein Pointer, der auf einen Counter zeigt, wie viel Städte in dem Array gespeichert sind.

```
#Kuerzeste Route: Zentrale  => Graz  => Linz  => Berlin  => Zentrale
```

city_struct_to_csv

Hier bekommt die Funktion einen Pointer zum Anfang eines Struct-Arry, plus einen Counter der angibt wie groß das Struct-Array ist. Diese Information wird dann in eine .csv Datei geschrieben. Der User wird zuvor noch aufgefordert den Namen der Datei einzugeben. Ist diese Datei bereits vorhanden wird das File mit der Information erweitert.

Sortieren

bubble

Der verwendete Bubblesort Algorithmus ist einer der bekanntesten und einfachsten Sortieralgorithmen. Der Algorithmus gehört zu den vergleichsbasierten Algorithmen und hat folgendes Verhalten bezüglich der Zeitkomplexität:

Günstigster Fall	Mittlerer Fall	Ungünstigster Fall
$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$

Der Funktion werden ein Zeiger auf das zu sortierende Array und die Länge des Arrayfeldes übergeben. Die Sortierung findet direkt im übergebenen Arrayfeld statt und die Ergebnisse werden auf der Konsole ausgegeben.

quicksort_name / quicksort_pop

Der verwendete quicksort Algorithmus ist deutlich komplexer als der bubblesort Algorithmus. Der Algorithmus gehört ebenfalls zu den vergleichsbasierten Algorithmen und hat folgendes Verhalten bezüglich der Zeitkomplexität:

Günstigster Fall	Mittlerer Fall	Ungünstigster Fall
$\Theta(n \cdot \log(n))$	$\Theta(n \cdot \log(n))$	$\Theta(n^2)$

Da es sich um einen rekursiver Algorithmus handelt und somit sich die Funktion bei Erfüllung einer bestimmten Bedingung selbst erneut aufruft, musste anders als beim

bubblesort Algorithmus für jedes Suchkriterium eine eigene Funktion erstellt werden. Den Funktionen werden jeweils zwei Zeiger übergeben, welche auf das erste bzw. das letzte Element des zu sortierenden Arrays zeigen. Das Array wird in zwei Teilbereiche unterteilt. Einer befindet sich links des errechneten Pivotelements, der andere rechts davon. Im Zuge des Vergleichs, werden sowohl die Zeiger verschoben, als auch Elemente des Arrays ausgetauscht. Der Vergleich der benötigten Laufzeit hat gezeigt, dass der quicksort Algorithmus bei steigender Anzahl von zu sortierenden Elementen schneller arbeitet als der bubblesort Algorithmus.

Cmake

Cmake war ein Faktor, der bei der Planung des Projektes außer Acht gelassen wurde und der später dafür verantwortlich war, dass die Zeitplanung angepasst werden musste. Anstatt noch mehr Zeit für Sortieralgorithmen aufzuwenden haben wir uns mit der Software und dem ganzen Konzept dahinter nochmals vertraut machen müssen da dieses Thema ganz am Anfang des Semesters am Plan stand. Es war aber sehr wichtig sein Gedächtnis mit doch relativ grundlegenden Dingen aufzufrischen was die Compiler, object und executable Files, make befehle in der Powershell usw. angeht. Am Ende wurde dann eine funktionierendes CMakeLists.txt Datei geschrieben mit deren Hilfe erfolgreich ein Makefile erstellt wurde. Cmake war ein Faktor, der bei der Planung des Projektes außer Acht gelassen wurde und der später dafür verantwortlich war, dass die Zeitplanung angepasst werden musste. Anstatt noch mehr Zeit für Sortieralgorithmen aufzuwenden haben wir uns mit der Software und dem ganzen Konzept dahinter nochmals vertraut machen müssen da dieses Thema ganz am Anfang des Semesters am Plan stand. Es war aber sehr wichtig sein Gedächtnis mit doch relativ grundlegenden Dingen aufzufrischen was die Compiler, object und executable Files, make befehle in der Powershell usw. angeht. Am Ende wurde dann eine funktionierendes CMakeLists.txt Datei geschrieben mit deren Hilfe erfolgreich ein Makefile erstellt wurde.