Hochschule
Albstadt-Sigmaringen

Albstadt-Sigmaringen University

Ant-Colony-System in der Logistik

Bachelor Thesis

Zur Erlangung des akademischen Grades
Bachelor of Engineering

Vorgelegt im Sommersemester 2013 bei
Prof. Dr. Tobias Häberlein (1. Prüfer) und
Prof. Dr. rer. nat. Walter Hower (2. Prüfer)

Von Felix Schumacher
Albstadt, 15. Juli 2013
Matrikel-Nr.: 77312

Abstract

Eines der Hauptprobleme der Transportlogistik ist, Routen mit minimaler Reisedistanz und minimaler Fahrzeuganzahl zu finden. Die vorliegende Arbeit behandelt die Ant-Colony-System (ACS) Heuristik als Lösungsansatz zur Ermittlung von Rundtouren zur Verwendung in der Logistik. Das Rundtourenproblem (Travelling Salesman Problem) ist NP-vollständig, kann also nicht von deterministischen Automaten in polynomieller Laufzeit gelöst werden (solange P $\neq$ NP). Daher ist eine Metaheuristik, wie das ACS, sehr gut zur Lösung geeignet. Das Vehicle Routing Problem ist eine Erweiterung des Travelling Salesman Problem, auf das die ACS-Heuristik angewendet werden soll. Es wird eine kurze Einführung zu Schwarmintelligenz gegeben, anschließend das ACS in Verbindung mit dem Travelling Salesman Problem betrachtet. Abschließend werden gängige Probleme der Logistik vorgestellt und das ACS zur Lösung des Vehicle Routing Problems erweitert.

Inhaltsverzeichnis

Kapitel 1

Schwarmintelligenz

Der Ausdruck *Schwarmintelligenz* wurde zum ersten mal im Bereich "Cellular Robotic Systems" verwendet [1, 2]. Im Rahmen dieser Arbeit werden statt Robotern allerdings soziale Insekten betrachtet, genauer gesagt Ameisen. Das Hauptaugenmerk soll hierbei auf der Tatsache liegen, dass eine Ameisenkolonie ohne Anführer auskommt. Jedes Koloniemitglied folgt lediglich ein paar wenigen, simplen Regeln.

Ein anschauliches Modell für solch ein Multiagenten-System ist beispielsweise ein Vogelschwarm. Von außen sieht es so aus, als ob der Schwarm gesteuert wird, als würde jemand entscheiden, wohin der Schwarm als nächstes fliegt. Wie in [3] beschrieben, entsteht diese Illusion, da jeder Vogel drei einfache Regeln befolgt:

- Fliege möglichst dicht bei benachbarten Vögeln
- Vermeide Kollisionen mit anderen Vögeln und sonstigen Hindernissen
- Passe deine Geschwindigkeit der Geschwindigkeit der benachbarten Vögel an

Um einen Vogelschwarm zu programmieren, wird zunächst nur ein Vogel betrachtet, welcher die oben genannten Regeln befolgen soll. Wann ändert er die Richtung? Wie verhält er sich in der Gegenwart von Nachbarn? Es wäre grundlegend falsch, zu versuchen das Verhalten des gesamten Schwarms zu programmieren. Hat man einen Vogel programmiert, startet man hunderte oder tausende gleichzeitig. Die Vögel beeinflussen sich gegenseitig und der Schwarm wird korrekt simuliert; ohne einen Anführer. [4]

Eine Ameisenkolonie funktioniert nach genau dem selben Prinzip.

1.1 Ameisen

Ameisen sind die wichtigste Klasse der sozialen Insekten und aufgrund ihrer Vielfalt und ihres außergewöhnlichen Verhaltens ein Vorzeige-Modell in der Biologie [5].

Wenn man eine Ameisenkolonie bei der Futtersuche beobachtet, fällt auf, dass die Ameisen zunächst planlos umherirren. Die Ameisen erkunden das umliegende Gebiet und hinterlassen währenddessen eine Pheromonspur. Stößt eine Ameise ihrerseits auf eine Pheromonspur, ist sie eher dazu geneigt, der Spur zu folgen, als eigene Wege zu erkunden. Findet eine Ameise eine Futterquelle, bildet sich recht schnell eine Ameisenstraße. Gibt es zwei Futterquellen, A und B, von denen A eine geringere Distanz zum Nest hat, bildet sich zuerst eine Straße zu A. Vereinzelt laufen auch Ameisen zur weiter entfernten Quelle B oder gar in komplett andere Richtungen (dies ist zwingend notwendig für die optimale Funktion eines selbstorganisierten Systems, mehr dazu in Kapitel 1.2), die Mehrheit der Ameisen bedient sich aber Quelle A. Erst wenn diese erschöpft ist, wird die Ameisenstraße zu B verlagert. Die Vorgehensweisen von Ameisen werden in [6] und [7] noch genauer erläutert.

Abbildung 1.1 (angelehnt an [7, S.54]) zeigt eine vereinfachte Darstellung, wie Ameisen den kürzesten Weg finden. Sei N das Ameisennest und, wie im vorigen Beispiel, A und B Futterquellen, wobei gilt: $2\overline{AN} = \overline{BN}$. Die Pheromonmenge (gestrichelte Linien) beider Strecken AN und BN ist zu Beginn äquivalent. Die Ameisen, die am Nest N starten, haben die Wahl zwischen Pfad AN, welcher zu Futterquelle A führt, und Pfad BN, welcher zu Futterquelle B führt. Da die Pheromonmenge von AN und BN zu Beginn äquivalent ist, ist die durchschnittliche Anzahl der Ameisen die AN wählen gleich der Anzahl derer, die BN wählen. Wenn alle Ameisen die gleiche Geschwindigkeit haben und eine Ameise für die Strecke AN t Zeiteinheiten und für BN 2t Zeiteinheiten benötigt, befindet sich nach $2t$ Zeiteinheiten an einem beliebigen Punkt auf AN folglich die doppelte Pheromonmenge von einem beliebigen Punkt auf BN. Starten nun n weitere Ameisen an N, wählen durchschnittlich $n/3$ Ameisen BN und $2n/3$ Ameisen AN.

1.2 Selbstorganisation

Das Ziel der Ameisen ist, die kürzeste Strecke zur am nächsten liegenden Futterquelle zu finden. Und das schaffen die Ameisen ohne zentrale Organisation.[1]

[1] Unter gewissen Umständen sogar schneller. Siehe auch Kapitel 2.1.

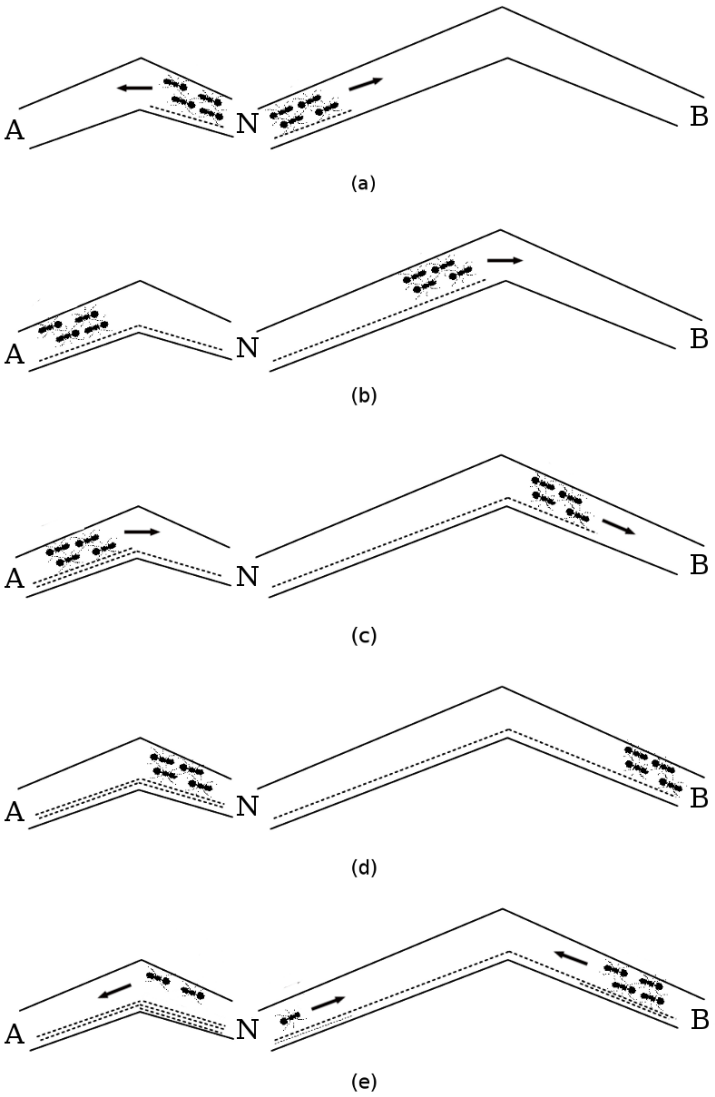

Abbildung 1.1: *Wegfindung bei Ameisen. Die Ameisen starten am Nest N, Futterquellen sind A und B. (a) Acht Ameisen werden losgeschickt, vier (Gruppe A') wählen AN, vier (Gruppe B') wählen BN. (b) Gruppe A' hat A erreicht, Gruppe B' hat die Hälfte des Weges zurückgelegt. (c) Gruppe A' ist auf dem Rückweg, Gruppe B' steht kurz vor B. (d) Gruppe A' ist bei N angekommen, Gruppe B' hat B erreicht. An einem beliebigen Punkt auf AN befindet sich mittlerweile die zweifache Pheromonmenge von einem beliebigen Punkt auf BN. (e) Werden drei weitere Ameisen losgeschickt, wählen aufgrund der ungleichen Pheromonmengen durchschnittlich zwei Ameisen Pfad AN und nur eine Ameise den Pfad BN.*

Sehr wichtig ist hierbei die Balance zwischen "Exploitation" und "Exploration". Bekannte Futterquellen müssen ausgebeutet werden (Exploitation), gleichzeitig sollte aber auch die Umgebung erforscht werden (Exploration), um nach eventuell näher liegenden oder ergiebigeren Quellen zu suchen. Angenommen, eine Ameise findet Quelle B und die übrigen Ameisen werden nicht fündig. Würden nun alle Ameisen lediglich dorthin laufen, wo sich am meisten Pheromon befindet, würde die Quelle A erst sehr spät entdeckt und nur B ausgebeutet werden. Um ein Optimum zu finden, darf die Erkundung der Umgebung nicht fehlen.

Eine gewisse Balance kann durch folgende Punkte, grob übernommen aus [1], erreicht werden.

Positives Feedback

Die Ameisen werden durch die Pheromone beeinflusst und dadurch rekrutiert. Dieses *positive Feedback* fördert die Entstehung von Routen. Anderen Ameisen wird bekannt gemacht, wo sich Futterquellen befinden und, je nach Pheromonmenge, wie ergiebig diese sind.

Allerdings würden sich die Ameisen so nach kurzer Zeit festfahren, auf eine sich möglicherweise nicht lohnende Futterquelle. *Negatives Feedback* und *Zufall* sorgen für eine Gegenbalance.

Negatives Feedback

Die Verdunstung von Pheromon kann als negatives Feedback gesehen werden und dient der "Exploration" (siehe Kapitel 2.2.2). Den Ameisen wird die Möglichkeit gegeben, die Umgebung weiträumiger zu erkunden und werden nicht auf den Pheromonpfaden "gefangen" [8]. Dies wirkt dem positiven Feedback entgegen und kann das System vor einem Deadlock bewahren. Lange Strecken sind dadurch automatisch im Nachteil, da das Pheromon mehr Zeit hat um zu verdunsten. Dies kann sehr effektiv als Optimierung bei der vorgestellten Heuristik (siehe Kapitel 2.2) verwendet werden.

Zufall

Zufall ist ein entscheidender Faktor um ein Optimum zu finden, da so ein ständiges Erkunden der Umgebung gewährleistet ist. Erst wenn Ameisen beginnen, Pfade durch Zufall zu wählen, können weitere kurze Routen gefunden werden. Im ACS basiert die Wahl des nächsten Knoten auf Wahrscheinlichkeiten und es ist so möglich, einen auf den ersten Blick "schlechten" Knoten zu wählen.

Kapitel 2

TSP & ACS

Das "Ant Colony System" (ACS) wurde von Marco Dorigo und Luca M. Gambardella 1997 unter dem Titel *Ant Colony System: A Cooperative Approach to the Travelling Salesman Problem* [7] vorgestellt.

Laut [7] ist das ACS ist einer der effektivsten Ant Colony Optimization Algorithmen und übertrifft andere Metaheuristiken, wie "Simulated Annealing" oder "Evolutionary Computation".

Die ursprüngliche Funktion ist die Lösung des Travelling Salesman Problem (TSP): Gegeben die Distanz zwischen jedem Paar aus n Städten und eine Konstante k, gibt es einen Hamilton-Kreis mit maximaler Länge k? [9, 10]

2.1 Travelling Salesman Problem

Das Travelling Salesman Problem ist ein entscheidbares, NP-vollständiges Problem und lässt sich als Sprache folgendermaßen beschreiben (übernommen aus [10, S.471]):

$$\begin{aligned} \text{Travelling Salesman} \quad := \quad & \{(G, g, k) | k \in \mathbb{N}, (G, g) \text{ ist ein vollständiger,} \\ & \text{bewerteter Graph mit } G = (V, E), \text{ wobei } g \\ & \text{die Kostenfunktion ist, und es existiert ein} \\ & \text{Hamilton-Kreis } H = (v_0, ..., v_n) \text{ in } G \text{ mit} \\ & \sum_{i=0}^{n-1} g((v_i, v_{i+1})) \leq k\} \end{aligned}$$

Da ein Computer lediglich alle Touren eines TSP systematisch berechnen und die kürzeste Tour auswählen könnte, ist das Travelling Salesman Problem durchaus entscheidbar. Allerdings gibt es bei n Städten $\frac{(n-1)!}{2}$ Touren und es wären rund $n!$ Schritte notwendig [9].

Bei nur 30 Städten[1] würde man bei einer Rechengeschwindigkeit von 10^{15} Schritten in der Sekunde immer noch mehrere Milliarden Jahre benötigen.

Das TSP liegt in NP und es lässt sich ohne große Probleme zeigen, dass es auch NP-vollständig ist [9, S.371].

Beweis. Es wird gezeigt, dass das Hamiltonkreisproblem ein Spezialfall des TSP ("Hamilton Circuit Problem" $\leq_p$ "Travelling Salesman Problem") ist.

Gegeben ein Graph $G = (V, E)$. Sei d_{ij} die Distanz zwischen Knoten v_i und v_j. Wir konstruieren nun einen vollständigen Graphen, für den gilt: $d_{ij} = 1$ if $[v_i, v_j] \in E, 2$ else. Somit gibt es im konstruierten Graphen eine Tour der Länge $|V|$ oder weniger dann und nur dann wenn ein Hamiltonkreis existiert. □

Eine Heuristik findet ausreichend gute Lösungen in angemessener Zeit und ist daher äußerst gut zur Lösung eines solchen Problems geeignet. Abstriche bei der Qualität der Tour werden durch eine kürzere Rechenzeit akzeptiert. Gerade bei praktischen Anwendungen muss nicht immer unbedingt die beste Rundtour gefunden werden und es ist geschickt, die Berechnung zu (fast) jeder Zeit abbrechen zu können. Bei normalen Algorithmen muss bis zur Terminierung gewartet werden, um ein vernünftiges Ergebnis zu bekommen. Eine Heuristik wie das ACS hingegen hat schon nach kurzer Zeit eine Lösung, die immer weiter verbessert wird. [8]

2.2 ACS-TSP

Vorgänger des Ant Colony System ist das Ant System (AS) [7]. Diese Heuristik unterscheidet sich aber nur geringfügig vom ACS, welcher eine optimierte Version des AS ist. Daher wird im Rahmen dieser Arbeit das AS nicht näher erläutert und die Funktionsweisen direkt am ACS behandelt.

Zu Beginn wird jede Ameise zufällig auf einen Knoten eines vollständigen Graphen gesetzt. Ist der gegebene Graph nicht vollständig, kann für jede fehlende Kante eine zusätzliche Kante eingefügt werden, deren Länge über den Dijkstra-Algorithmus berechnet wird (siehe Kapitel 2.2.1). Anschließend erzeugt jede Ameise eine Tour durch den Graphen. Hierbei verwenden sie die "State Transition Rule", durch die kurze Kanten und Kanten mit viel Pheromon bevorzugt werden. Nach jeder Knotenwahl wird die "Local Pheromone Updating Rule" auf die entsprechende Kante, welche den Ausgangsknoten und den gewählten Knoten verbindet, angewendet. Haben alle Ameisen ihre Tour beendet, wird die "Global Pheromone Updating Rule" auf alle Kanten

[1] 30! ist eine 33-stellige Zahl.

der besten Tour angewendet. Danach werden beliebig viele weitere Touren generiert, die sich, wie oben besprochen, durch Selbstorganisation immer weiter an das Optimum annähern [7]. Algorithmus 1 zeigt die wesentliche Funktionsweise.

Die Anzahl der Ameisen ist nach Experimenten in [7] in den meisten Fällen mit 10 optimal. Bei einer gewissen Größe des Graphen sollte allerdings auch die Anzahl der Ameisen angepasst werden.

Um geschickt auf die Längen und Pheromonmengen der Kanten zugreifen zu können, wird der Graph für das ACS am besten durch zwei Adjazenzmatrizen dargestellt. Die Werte der einen Matrix sind die Längen, die der anderen die Mengen an Pheromon der jeweiligen Kanten.

Algorithmus 1: Ant Colony System

```
Result : best tour
create complete graph;
position ants randomly;
while tour not good enough do
    while not all nodes visited do
        for every ant do
            choose next node (state transition rule);
            apply local updating rule;
        end
    end
    check for new best tour;
    apply global updating rule to best tour;
end
```

2.2.1 Dijkstra-Algorithmus

Der Dijkstra-Algorithmus, benannt nach seinem Erfinder Edsger Dijkstra, berechnet die kürzesten Wege von einem Knoten u zu jedem anderen Knoten des Graphen. Um dies zu erreichen, wird immer der von u am nächsten liegende, noch nicht besuchte Knoten betrachtet. Genauer behandelt in [11, S.162-165].

Beispiel. Sei u der Startknoten, v der nächste Nachbar von u, v' der zweitnächste Nachbar von u und w der nächste Nachbar von v, wobei gilt $w \neq v'$ und $w \neq u$.

Zunächst wird der Knoten v betrachtet, da er die kleinste Distanz zu u hat. Somit ist der kürzeste Pfad von v zu u auch schon bestimmt. Anschließend wird überprüft, ob die Distanz zwischen u und v' kleiner ist als die Distanz zwischen u und w. Falls ja, wird mit dem Pfad (u, v') fortgefahren. Falls nicht, wird mit (u, v, w) weitergemacht und der kürzeste Pfad zu w bestimmt. Dieses Verfahren wird solange angewendet, bis alle Knoten besucht wurden.

Um einen Graphen zu vervollständigen kann die Funktion aus Listing 2.1 in Verbindung mit der Funktion aus Listing 2.2 verwendet werden.

```
1 def makeComplete(graph):
2   # graph: 2D array with weight of edges as values
3
4   nodes = range(len(graph))
5   for i in nodes:
6     for j in nodes:
7       if ((j not in neighbour(i)) and (i != j)):
8         graph[i][j] = dijkstra(i)[1][j]
```

Listing 2.1: *Einen Graphen vervollständigen*

2.2.2 State Transition Rule

Die "State Transition Rule" [7, S.55-56] wird zur Bestimmung des nächsten Knotens verwendet. Eine Kante muss dabei zwangsläufig eine doppelte Gewichtung besitzen. Zum einen die Distanz zwischen den Knoten, zum anderen die Menge an Pheromon.

Durch Zufall wird bestimmt ob eine Formel zur "Exploration" oder "Exploitation" verwendet werden soll (siehe auch Kapitel 1.2).

Bei der "Exploitation" wird einfach der Knoten gewählt, dessen Kante das maximale Pheromon-Distanz-Verhältnis hat.

$$max_{u \in J_k(r)} \{ \frac{\tau(r,u)}{\delta(r,u)^\beta} \}$$

Um die Ameisen davor zu bewahren sich auf einer Tour festzufahren, wird die Formel zur Erkundung der Umgebung ("Exploration") verwendet. Es wird jedem Knoten eine Wahrscheinlichkeit zugeteilt, dass Ameise k auf Knoten r den Knoten s wählt.

Hierfür wird die Relation zwischen Pheromon und Distanz auf Kante (r, s) durch die Summe der Relationen der Kanten (r, u) geteilt, wobei u

```
def dijkstra(graph, s):
  # graph: 2D array with weight of edges as values
  # s: startnode

  numNodes = len(graph)
  nodes = range(len(graph))
  dist = {s:0} # dictionary containing distances
  path = []
  k = {} # current path
  for i in range(0,numNodes):
    # distv: Entfernung, v = nearest node, cn = current node
    distv, v = min( [ (dist[cn], cn) for cn in dist.keys() if cn in
        nodes ] )
    nodes.remove(v) # remove nearest node
    # if v not startnode: append shortest path
    if v != s:
      path.append(k[v])
    for v1 in neighbour(v):
      #original : [ v2 for v2 in graph.v[v] if v2 in nodes ]:
      if v1 not in dist or dist[v] + graph[v][v1] < dist[v1]:
        dist[v1] = dist[v] + graph[v][v1]
        k[v1] = (v,v1) # current path
  return (path,dist)
```

Listing 2.2: *Dijkstra-Algorithmus*

ein noch nicht besuchter Knoten ist. Alle schon besuchten Knoten erhalten selbstverständlich die Wahrscheinlichkeit 0.

$$p_k(r,s) = \begin{cases} \dfrac{\frac{\tau(r,s)}{\delta(r,s)^\beta}}{\sum\limits_{u \in J_k(r)} \dfrac{\tau(r,u)}{\delta(r,u)^\beta}}, & \text{if s} \in J_k(r) \\ 0, & \text{else} \end{cases}$$

Um aus diesen beiden Formeln zu wählen, wird ein Parameter q_0 zufällig bestimmt. Ist dieser kleiner als 0.9, wird "Exploitation" verwendet. β wird auf 2 gesetzt. Diese Werte wurden in [7, S.56-57] bestimmt.

Wie die "State Transition Rule" in Python implementiert werden kann, zeigt Listing 2.3.

2.2.3 Local Updating Rule

Um das Hinterlassen von Pheromonen zu simulieren, wird von jeder Ameise auf ihre gewählte Kante die "Local Updating Rule" [7, S.56] angewendet. Der

```
def stateTransitionRule(graph, pheromone, cN, remaining):
  # graph: 2D array with weight of edges as values.
  # pheromone: 2D array with pheromoneamount of edges as values.
  # cN: the current node of the ant.
  # returns the next node.

  q0 = 0.9
  beta = 2
  q = random.random()

  if (q <= q0): # use first formula
    maxval = 0
    for i in remaining: # for every remaining node
      tmp = pheromone[cN][i] / (graph[cN][i] ** beta)
      if ( tmp >= maxval ): # find max
        maxval = tmp
        s = i # set next node

  else: # use second formula
    # prob: array, where each index has the probability (of
    # being picked) of the value of the array at that index
    # initially all 0
    prob = [ 0 for k in range(len(graph)) ]
    sumval = 0

    for i in remaining: # for every remaining node
      tmp = pheromone[cN][i] / (graph[cN][i] ** beta)
      sumval += tmp

    for i in remaining: # for every remaining node
      tmp = pheromone[cN][i] / (graph[cN][i] ** beta)
      prob[i] = (tmp / sumval) # set probability of node i

    s = probfunc(prob) # choose next node based on probability

  return s
```

Listing 2.3: *State Transition Rule*

```python
def localUpdatingRule(pheromone, lN, cN, tau0val):
  # update pheromone[lN][cN] using the local updating rule.
  #
  # pheromone: list containing pheromone values.
  # lN: the previous node of the ant.
  # cN: current node of the ant.
  # tau0val: initial pheromone amount

  rho = 0.1
  nval = (1-rho)*pheromone[lN][cN] + rho*tau0val
  pheromone[lN][cN] = nval # update pheromone on edge (lN,cN)
```

Listing 2.4: *Local Updating Rule*

aktuelle Pheromonwert wird um einen gewissen Prozentsatz reduziert und ein Wert τ_0, abhängig von der "Nearest Neighbour Tour"[2], dazu addiert.

$$\tau(r,s) \leftarrow (1-\rho) \cdot \tau(r,s) + \rho \cdot \tau_0$$

Wie in [7, S.56-57] bestimmt, sei hierbei $\rho = 0.1$ (Zerfalls-Parameter), $\tau_0 = (n \cdot L_{nnt})^{-1}$, wobei L_{nnt} die Länge der "Nearest Neighbour Tour" und n die Anzahl der Knoten ist.

Wie die "Local Updating Rule" in Python implementiert werden kann, zeigt Listing 2.4

2.2.4 Global Updating Rule

Die "Global Updating Rule" [7, S.56] funktioniert fast wie die Local Updating Rule, mit dem Unterschied, dass die Pheromonmenge auf den Kanten der global besten Tour nicht so schnell verdunstet.

$$\tau(r,s) \leftarrow (1-\alpha) \cdot \tau(r,s) + \alpha \cdot \Delta\tau(r,s)$$

wobei

$$\Delta\tau(r,s) = \begin{cases} \frac{1}{L_{gb}}, & \text{if (r,s)} \in \text{global best tour} \\ 0, & \text{else} \end{cases}$$

Hierbei sei L_{gb} die Länge der global besten Tour und $\alpha = 0.1$, ein Zerfalls-Parameter.

[2]Bei der "Nearest Neighbour Tour" wird immer der am nächsten liegende Knoten gewählt

```
def globalUpdatingRule(graph, pheromone, bestTour):
  # Update graph using the global updating rule.
  #
  # graph: 2D array with weight of edges as values.
  # pheromone: 2D array with pheromoneamount of edges as values.
  # bestTour: array containing the best tour so far

  for r in range(len(graph)): # for every node
    for s in range(len(graph)): # for every node
      alpha = 0.1
      nval = (1-alpha) * pheromone[r][s]
      if isEdgeOfBestTour(bestTour, r, s):
        nval += alpha * (1/gtl(graph, bestTour))
      pheromone[r][s] = nval # update pheromone on edge (r,s)
```

Listing 2.5: *Global Updating Rule*

Wie die "Global Updating Rule" in Python implementiert werden kann, zeigt Listing 2.5.

2.2.5 Testprogramm

An einem vollständigen, ungerichteten Graphen kann demonstriert werden, wie die ACS-TSP Heuristik die Länge der "Nearest Neighbour Tour" (NNT) unterbietet. Bei der NNT wird ganz intuitiv immer der am nächsten liegende Knoten gewählt. Allerdings ist diese Tour oft bei weitem nicht die kürzeste. Listing 2.6 zeigt das Testprogramm.

Die Knotenmenge des Beispielgraphen sei $\{0, 1, 2, 3, 4, 5, 6, 7, 8\}$ und die Kanten wie folgt gewichtet.

$(0, 1) = (0, 2) = (0, 3) = (0, 4) = 4$
$(0, 5) = (0, 6) = (0, 7) = (0, 8) = 7$
$(1, 2) = (1, 3) = (2, 4) = (3, 4) = 8$
$(1, 5) = (2, 6) = (3, 7) = (4, 8) = 5$
$(1, 6) = (1, 7) = (2, 5) = (2, 8) = (3, 5) = (3, 8) = (4, 6) = (4, 7) = 12$
$(1, 8) = (2, 7) = (3, 6) = (4, 5) = 16$
$(1, 4) = (2, 3) = 11$
$(5, 6) = (5, 7) = (6, 8) = (7, 8) = 18$
$(5, 8) = (6, 7) = 22$

Abbildung 2.1 zeigt den Beispielgraphen, wobei die Kanten der NNT rot markiert sind. Die Kosten dieser Tour sind 67 Längeneinheiten. In Abbildung 2.2 sind die Kanten der ACS-TSP Tour markiert. Die Kosten sind hier nur

66 Längeneinheiten. Dies ist eine Verbesserung um nur eine Längeneinheit, allerdings hat der Graph nur 9 Knoten.

Das ACS-TSP hat, ausgehend vom Knoten 0, nicht direkt den Knoten 1 gewählt, sondern über einen Umweg den Knoten 5, kann dadurch aber den Pfad $(1, 6, 2, 8, 4, 3)$ benutzen, der die Kante $(3, 4)$ beinhaltet, die nur 8 Längeneinheiten hat. Es werden so nur zwei statt drei 12er Kanten gewählt. Die Kante $(3, 4)$ könnte bei der NNT nur gewählt werden, wenn Kante $(5, 6)$ oder $(6, 8)$ benutzt wird, welche aber beide mit 18 Längeneinheiten viel zu lang sind.

2.2.6 Multiprozessor-Unterstützung

Das ACS ist eine Heuristik, welche sich zur parallelen Berechnung eignet. Bisher wurde nacheinander jeder Ameise ein neuer, von ihr noch nicht besuchter Knoten zugewiesen. So besucht jede Ameise nach $n - 1$ Schritten jeden der n Knoten genau einmal.

Bei dem in Listing 2.7 dargestellten Ansatz wird der Ablauf aber leicht verändert. Beispielsweise ist die Generierung der Touren nicht geordnet und die Anzahl der Ameisen nicht genau bestimmbar. Es werden mehrere Prozesse parallel gestartet, wobei jeder eine Tour berechnet, nebenher die Local Updating Rule auf die verwendeten Kanten anwendet und die Route in einer für alle Prozesse verfügbaren Queue speichert. Einer der Prozesse durchsucht nach der Generierung seiner Tour die Queue nach der schnellsten Rundtour und wendet die Global Updating Rule darauf an. Während diesen Berechnungen generieren die anderen Prozesse immer neue Touren und somit ist nicht genau definiert wie viele Touren vor jeder Berechnung erstellt wurden.

2.3 Tourenplanung

Zur sinnvollen Verwendung des ACS zur Erzeugung von Rundtouren, und um nicht nur auf theoretischen Graphen Rundtouren zu berechnen, werden geographische Daten benötigt. Eine geeignete Quelle ist OpenStreetMap©, dessen Daten unter der Open Database License verfügbar sind.[3] Dies ist aber, je nach Ausmaß der Städteverteilung, eine möglicherweise enorme Menge an Daten und somit wird eine Verwendung einer Datenbank unerlässlich. Von der OpenStreetMap© Website können OSM-Dateien bezogen werden (XML-Format). Je nach Gebiet können die Daten einzelner Länder bis hin zu einer Datei der ganzen Erde heruntergeladen werden.

[3] http://www.openstreetmap.org/

```
nodes = [0, 1, 2, 3, 4, 5, 6, 7, 8]

graph = [[0, 4, 4, 4, 4, 7, 7, 7, 7],
[4, 0, 8, 8, 11, 5, 12, 12, 16],
[4, 8, 0, 11, 8, 12, 5, 16, 12],
[4, 8, 11, 0, 8, 12, 16, 5, 12],
[4, 11, 8, 8, 0, 16, 12, 12, 5],
[7, 5, 12, 12, 16, 0, 18, 18, 22],
[7, 12, 5, 16, 12, 18, 0, 22, 18],
[7, 12, 16, 5, 12, 18, 22, 0, 18],
[7, 16, 12, 12, 5, 22, 18, 18, 0]]

n = len(graph) # number of nodes
tau0val = tau0(graph) # initial pheromone amount

# pheromone array
pher = [ [ tau0val for i in range(n) ] for j in range(n) ]

# number of ants
if (n < 10):
  ants = n
else:
  ants = 10

# list of remaining nodes
remaining = [ nodes[:] for i in range(ants) ]

tours = [ [] for i in range(ants) ] # tours generated
bestTour = [] # the best tour

positionAnts(ants, tours, n, remaining)

for tmp in range(50):
  for count in range(1000):
    for i in range(n):
      chooseNext(graph, pher, remaining, tours)
    bestTour = checkForBestTour(graph, tours, bestTour)
    globalUpdatingRule(graph, pher, bestTour)
    reset(remaining, tours, nodes, ants)
    positionAnts(ants, tours, n, remaining)
  print 'best tour: ', bestTour
  print 'length of best tour: ', gtl(graph, bestTour)
  print 'nnt: ', nnt(graph, 0)
  print 'length of nnt: ', gtl(graph, nnt(graph, 0))
```

Listing 2.6: *Testprogramm*

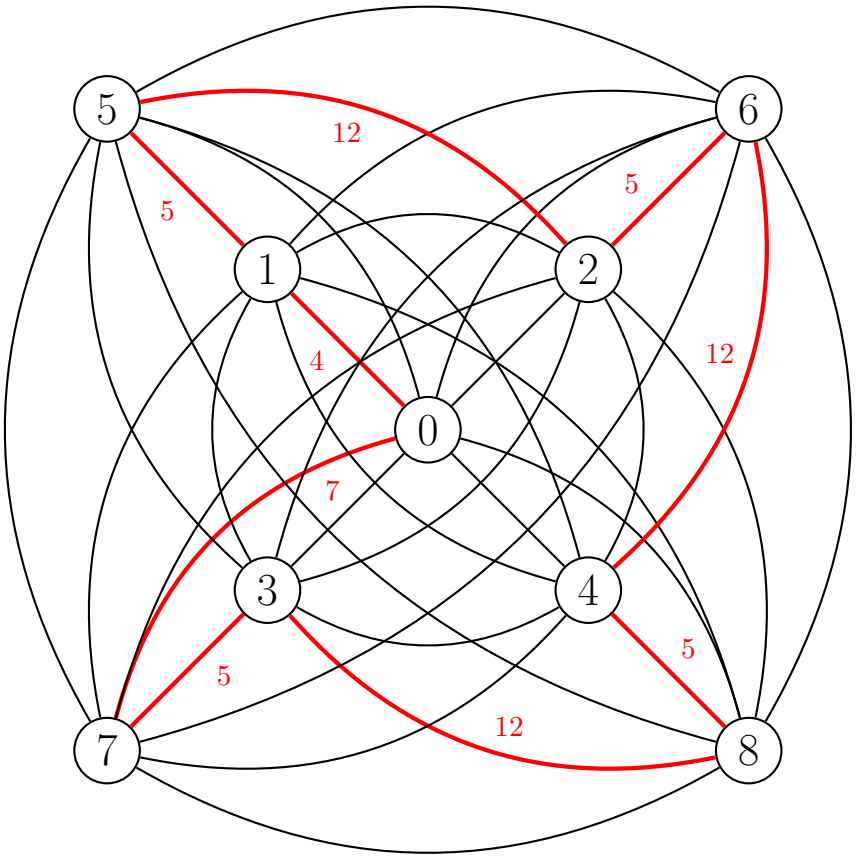

Abbildung 2.1: *Nearest Neighbour Tour*

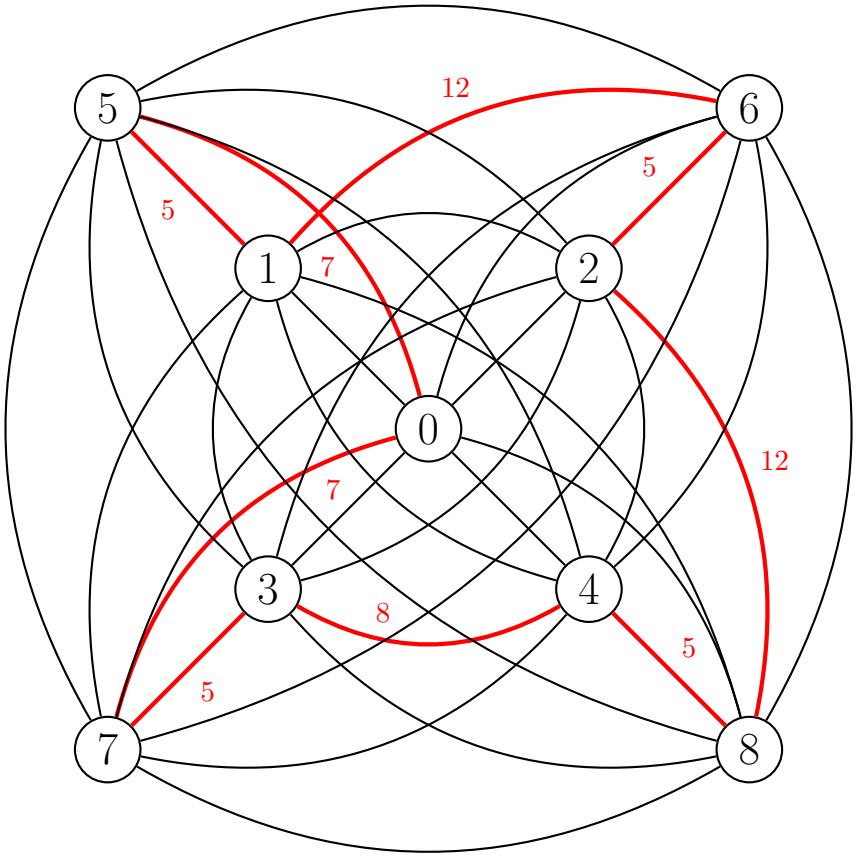

Abbildung 2.2: *ACS-TSP Tour*

```
def multiprocess(graph, pheromone, startNode, bestTour, lock, ants,
    q, i, counter):
  # Use multiple processes to calculate tours.
  #
  # graph: 2D array with weight of edges as values.
  # pheromone: shared Array containing pheromone values.
  # startNode: node from where to start the tour.
  # bestTour: the best tour so far.
  # lock: lock a code segment.
  # ants: "number of ants"; minimal number of tours generated.
  # q: queue containing the tours
  # i: id of this process.
  # counter: counter to control how many tours are calculated.

  numNodes = len(graph)
  loop = 10000
  tours = []

  while (True):
    # calculate tour using str and save to queue
    ant(graph, pheromone, startNode, lock, numNodes, q)
    counter.value = counter.value + 1

    # id 0 checks for new best tour and applies global
    # updating rule; number of tours in queue has to
    # be greater than "number of ants"
    if ((i == 0 and q.qsize() >= ants) or counter.value >= loop):
      lock.acquire()
      for x in range(q.qsize()): # loop through queue
        tours.append(q.get()) # save tours to separate list
      lock.release()
      bestTour = checkForBestTour(graph, tours, bestTour)
      globalUpdatingRule(graph, pheromone, bestTour, lock)
      tours = []

    if (counter.value >= loop): # end loop
      break

  if (i == 0): # id 0 prints best tour
    print bestTour
```

Listing 2.7: *Multiprocessing Implementation*

```
1  import psycopg2
2
3  def getDistance(a,b):
4    # Calculate the distance between two osm node-indices.
5    #
6    # a: first osm index.
7    # b: second osm index.
8    # returns the distance between a and b
9
10   if (a == b):
11     return 0
12   con = psycopg2.connect(database='osm', user='postgres')
13   cur = con.cursor()
14   query = "SELECT * FROM shortest_path('SELECT gid as id, source::
         integer, target::integer, length::double precision as cost
         FROM ways'," + str(a) + "," + str(b) + ", false, false);"
15   cur.execute(query)
16   rows = cur.fetchall()
17   s = 0
18   for row in rows:
19     s = s + row[2]
20   return s
```

Listing 2.8: *getDistance()*

Über das Tool *osm2pgrouting*[4] können diese Daten in eine PostgreSQL Datenbank geladen werden. Dabei werden die Daten so aufbereitet, dass bequem über *pgRouting*[5] die Distanz zwischen zwei geographischen Punkten berechnet werden kann. Wie dieses Tool in Python verwendet werden kann, zeigt Listing 2.8.

Eine Erleichterung der Bedienung kann durch eine geeignete Benutzeroberfläche erreicht werden. Beispielsweise könnten die Kartendaten gerendert werden um über Mausklicks die OSM-Indizes zu bestimmen. Dies wird hier allerdings nicht weiter behandelt.

[4]https://github.com/pgRouting/osm2pgrouting
[5]https://github.com/pgRouting/pgrouting

Kapitel 3

Einsatzmöglichkeiten in der Transportlogistik

> *Die Logistik ist die Lehre der ganzheitlichen Planung, Bereitstellung, Durchführung und Optimierung von Prozessen der Ortsveränderung von Gegenständen, Daten, Energie und Personen sowie der notwendigen Transportmittel selbst. Sie sichert den quantitativen und qualitativen Erfolg von Transportprozessen und die räumliche Mobilität der betrachteten Objekte.* [12, S.5]

Ein Ziel der Transportlogistik ist, die Reisedistanz sowie die Anzahl der verwendeten Fahrzeuge zu minimieren. Das "Vehicle Routing Problem" (VRP) beschreibt diese Problemstellung.

3.1 Vehicle Routing Problem

Das VRP ist eine Erweiterung des TSP, bei dem die kürzeste Route zur Belieferung von Kunden mit minimaler Anzahl von Fahrzeugen gefunden werden soll. Es kann in folgende Gruppen unterteilt werden, die das VRP realitätsnäher machen [13].

Das allgemeinste VRP ist das "Capacitated Vehicle Routing Problem" (CVRP). Bei diesem Problem sollen von einem Depot aus Kunden beliefert werden, wobei die Transportfahrzeuge eine begrenzte Ladekapazität besitzen.

Darauf aufbauend gibt es das "VRP with Time Windows" (VRPTW). Hierbei hat jeder Kunde ein bestimmtes Zeitfenster, in dem er beliefert werden kann.

Des Weiteren gibt es das "VRP with Pick-up and Delivery" (VRPPD), bei dem die zu transportierenden Güter nicht unbedingt in den Depots gelagert

sind, sondern auf Knoten des Graphen verteilt sind. Hierzu zählt beispielsweise das "dial-a-ride" Problem. Es sollen Personen abgeholt werden und zu bestimmten Reisezielen transportiert werden.

Eine Weiterführung des VRPTW ist das "Time Dependent VRP". Die Kosten der Kanten des Graphen variieren hier von Zeit zu Zeit, was zum Beispiel Verkehrslast zu verschiedenen Tageszeiten simulieren soll.

Das "Dynamic VRP" lässt sich während der Routenberechnung anpassen. Oft sind zu Beginn der Berechnung nicht alle Parameter bekannt, und lassen sich zur Laufzeit verändern. Hierfür ist das ACS gut geeignet, da, wie in Kapitel 1.2 beschrieben, die Verdunstung der Pheromone (negatives Feedback) eine dynamische Anpassung erlaubt.

In dieser Arbeit wird das CVRP betrachtet, welches nach [14] folgendermaßen beschrieben werden kann. Es sollen n Kunden von einem Depot beliefert werden. Jeder Kunde will eine bestimmte Anzahl q_i von Gütern $(i = 0, ..., n-1)$, die mit Hilfe eines Fahrzeugs mit Kapazität Q geliefert werden sollen. Dieses Fahrzeug muss allerdings aufgrund der Kapazitätseinschränkung eventuell wieder zum Depot zurückkehren um weitere Kunden beliefern zu können. So entsteht eine Menge von Touren, bei denen jeder Kunde genau einmal besucht wird, vorausgesetzt eine Tour kostet maximal Q Güter.

3.2 ACS-VRP

Beim ersten in dieser Arbeit betrachteten Ansatz wird der Knoten 0 des Graphen in die gewünschte Anzahl von Depots aufgespalten. Die Distanz zwischen diesen Knoten wird auf 0 gesetzt. So wird dem ACS-TSP, welches das Ziel hat, jeden Knoten genau einmal zu besuchen, die Möglichkeit gegeben, das Depot mehrmals zu besuchen, wobei eigentlich nur eine ganz normale Rundtour generiert wird. Diese Rundtour kann anschließend an den Depot-Knoten getrennt werden, was die Touren liefert, die ein Fahrzeug nacheinander fährt, oder die Touren, die eine Menge von Fahrzeugen jeweils fährt. (Die Betrachtung macht keinen Unterschied, von hier an wird allerdings von einer Menge an Fahrzeugen ausgegangen.)

Bei n Knoten wird der Knoten 0 zu Beginn durch $n-1$ Depot-Knoten ersetzt, was $n-1$ Fahrzeugen entspricht. Somit ist die Möglichkeit gegeben, jeden Kunden mit genau einem Fahrzeug zu beliefern. Zusätzlich werden eine Liste der gewünschten Anzahl der Güter der $n-1$ Kunden und die Ladekapazität der Fahrzeuge benötigt.

Der eigentliche Algorithmus unterscheidet sich nur geringfügig vom ACS-TSP. Es müssen lediglich die Ameisen von der vorzeitigen Rückkehr zum

Depot abgehalten werden. Wenn sich noch ausreichend Güter im Fahrzeug befinden, sollen diese natürlich auch ausgeliefert werden.

Beispiel. Als praktisches Beispiel betrachten wir den in Abbildung 3.1 abgebildeten vollständigen Graphen $G = (V, E)$ mit der Knotenmenge $V = \{0, 1, 2, 3, 4\}$. Sei Knoten 0 das Depot und die Knoten 1 bis 4 die zu beliefernden Kunden, wobei die tiefgestellten Zahlen der Knoten die Nachfrage darstellen.

Wird darauf die ACS-TSP Heuristik angewendet, ergibt sich die Tour $(0, 2, 4, 1, 3, 0)$ mit der Länge 21, dargestellt durch die rot markierten Kanten.

Um nun die 4 Kunden zu beliefern, werden zu Beginn 4 Fahrzeuge verwendet, oder anders gesagt, es werden 4 Touren generiert. Hierfür wird Knoten 0 durch 4 Depot-Knoten ersetzt, dargestellt in Abbildung 3.2. Für eine bessere Übersichtlichkeit wurde allerdings auf einige Kanten verzichtet, beispielsweise ist Knoten 0 nicht mit 7 verbunden.

Legen wir eine Kapazität $Q = 40$ fest, was der Summe der verlangten Güter der Kunden entspricht, können alle Kunden in einer Rundtour beliefert werden. Es entsteht die Route $(0, 2, 1, 3, 5, 7, 4, 6, 0)$, was genau der Tour aus Abbildung 3.1 entspricht. Wird aber $Q = 19$ gewählt, können nur noch maximal 2 Kunden in einer Tour beliefert werden. Durch das ACS-VRP wird die Gesamttour $(0, 4, 2, 6, 1, 3, 5, 7, 0)$ generiert, die sich in die 3 Touren $(0, 5, 7, 0)$, $(0, 4, 0)$ und $(0, 6, 0)$ aufspalten lässt (hier soll 0 die vier Depot-Knoten repräsentieren). In Abbildung 3.2 sind die Touren rot markiert.[1]

Um den ACS-TSP Algorithmus zur Lösung des CVRP verwenden zu können, wird das Depot in n Knoten, deren Distanz zueinander 0 ist, über die Funktion addDepots(), dargestellt in Listing 3.1, aufgespalten. Folgen in einer Tour mehrere Depots aufeinander, kann die Anzahl der Fahrzeuge verringert werden. Ist die Kapazität der Fahrzeuge groß genug, ist es aufgrund der nicht vorhandenen Distanz zwischen den Depots sehr wahrscheinlich, dass diese in der Tour direkt aufeinander folgen. Dies würde einer einzigen Rundtour mit nur einem Depot entsprechen, wie sie auch vom ACS-TSP erzeugt worden wäre.

Daher müssen die Ameisen bei noch vorhandener Kapazität von der frühzeitigen Rückkehr zum Depot abgehalten werden. Bei der Wahl des nächsten Knotens wird eine Liste mit allen noch erreichbaren Kunden erzeugt, die mit der aktuellen Anzahl an geladenen Gütern beliefert werden können. Ist diese Liste leer, muss zum Depot zurückgekehrt werden. Eine genaue Realisierung dieser Einschränkung zeigt Listing 3.2.

[1] Die Kanten $(0, 4)$ und $(2, 6)$ stehen natürlich jeweils für zwei Kanten.

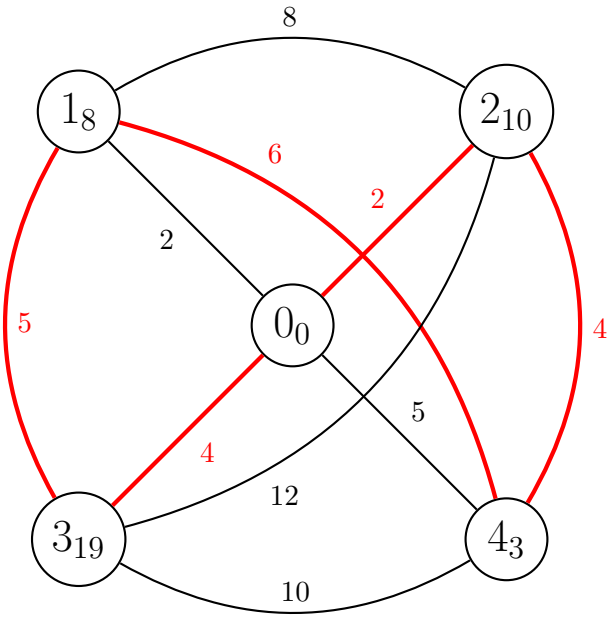

Abbildung 3.1: *ACS-TSP Tour*

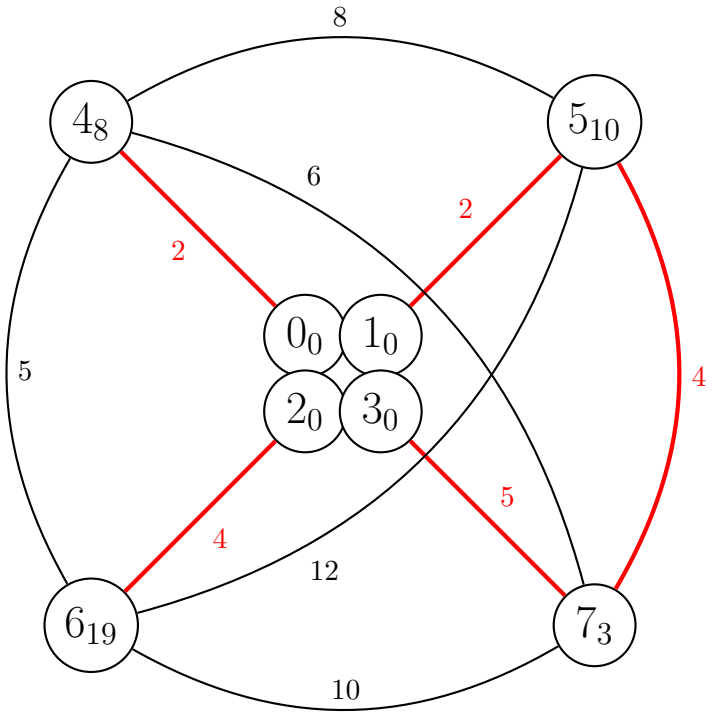

Abbildung 3.2: *Drei Routen mit* $Q = 19$

```
def addDepots(v, graph):
  # add v depots
  # vertex 0 gets replaced by those depots
  # distance between depots is 0
  #
  # v: number of vehicles/depots
  # graph: 2D array with the weight of the edges as values.
  # returns updated graph

  l = []

  for g in graph: # create copy of graph
    l.append(g[:])
  l.pop(0) # remove adjacency list 0, depots form list 0
  for i in l:
    for j in range(v-1):
      i.insert(0,i[0]) # insert distance to depots 0,1,...
  for i in range(v):
    tmp = [0 for j in range(v)] # distance between depots if 0
    for j in range(1,len(graph)):
      tmp.append(graph[0][j]) # append distance
    l.insert(0, tmp)
  return l
```

Listing 3.1: *addDepots()*

```
def chooseNext(g, p, remaining, tours, depots, maxCapacity, cap,
    demand):
  # For every ant: choose next node
  #
  # g: 2D array with weight of the edges as values.
  # p: 2D array with pheromoneamount of the edges as values.
  # remaining: list of remaining nodes
  # tours: list of tours
  # maxCapacity: maximum capacity of vehicles
  # cap: remaining goods of ants
  # demand: quantity of goods the customer asks for

  ant = 0 # current ant
  for a in tours: # for every ant tour
    reachable = []
    for i in remaining[ant]:
      if (cap[ant] >= demand[i] and demand[i] != 0):
        reachable.append(i)
    if (not reachable): # reachable is empty
      for i in remaining[ant]:
        if (cap[ant] >= demand[i]):
          reachable.append(i)
    if (not reachable):
      continue
    oldPos = a[len(a)-1]
    newPos = stateTransitionRule(g, p, oldPos, reachable, depots)
    cap[ant] -= demand[newPos]
    if (newPos in depots):
      cap[ant] = maxCapacity
    localUpdatingRule(p, oldPos, newPos, tau0(g))
    a.append(newPos)
    remaining[ant].remove(newPos)
    ant += 1
```

Listing 3.2: *chooseNext()*

```
graph = [[0, 20, 14, 10, 2, 7, 3, 20, 3, 40, 1, 22, 6, 20],
[20, 0, 2, 5, 4, 33, 10, 30, 3, 12, 42, 19, 8, 21],
[14, 2, 0, 10, 3, 22, 10, 3, 2, 33, 23, 7, 27, 5],
[10, 5, 10, 0, 6, 20, 20, 11, 21, 21, 73, 6, 14, 20],
[2, 4, 3, 6, 0, 1, 2, 40, 12, 18, 17, 25, 30, 7],
[7, 33, 22, 20, 1, 0, 40, 5, 3, 2, 3, 11, 10, 33],
[3, 10, 10, 20, 2, 40, 0, 8, 4, 7, 8, 24, 5, 13],
[20, 30, 3, 11, 40, 5, 8, 0, 9, 11, 4, 12, 3, 19],
[3, 3, 2, 21, 12, 3, 4, 9, 0, 12, 42, 33, 21, 18],
[40, 12, 33, 21, 18, 2, 7, 11, 12, 0, 6, 3, 17, 4],
[1, 42, 23, 73, 17, 3, 8, 4, 42, 6, 0, 6, 26, 8],
[22, 19, 7, 6, 25, 11, 24, 12, 33, 1, 6, 0, 20, 15],
[6, 8, 27, 14, 30, 10, 5, 3, 21, 17, 26, 20, 0, 18],
[20, 21, 5, 20, 7, 33, 13, 19, 18, 4, 8, 15, 18, 0]]

demand = [2, 10, 5, 18, 7, 8, 1, 16, 4, 18, 13, 12, 10]
```

Listing 3.3: *Initialization*

3.3 Multi-ACS-VRP

Allerdings findet die im vorigen Kapitel verwendete Methode nicht immer das optimale Minimum an Fahrzeugen. Eine Reduzierung der Fahrzeuganzahl auf ein annehmbares Minimum wird aber sehr schnell erreicht. Der Multi-ACS-VRP (MACS-VRP) Ansatz braucht hingegen relativ lange, um die Anzahl der Fahrzeuge zu minimieren, ermittelt aber fast immer die minimale Anzahl an Fahrzeugen.

Beim "Multi-ACS-VRPTW" Ansatz [15] arbeiten zwei ACS-TSP Prozesse zusammen, um die beste Tour zu finden. Ein Prozess minimiert die Distanz der Tour, während der andere die Anzahl der Fahrzeuge minimiert. Der erste Prozess startet ohne Einschränkungen der Fahrzeuganzahl[2], der zweite Prozess rechnet mit einem Fahrzeug weniger. Findet der zweite Prozess eine Tour, die besser ist als das derzeitige Optimum, werden beide Prozesse beendet. Anschließend wird der erste Prozess wieder mit der um ein Fahrzeug reduzierten Anzahl gestartet und der zweite mit wiederum einem Fahrzeug weniger.

Durch Kombination dieser beiden Ansätze kann eine schnelle und optimale Reduzierung der Fahrzeuganzahl erreicht werden.

Beispiel. Listing 3.3 zeigt die Kantenlängen des Graphen und die Nachfrage der Kunden. Die Kapazität der Fahrzeuge wird auf 20 gesetzt.

Wird der ACS-VRP Algorithmus aus Kapitel 3.2 verwendet, werden 7

[2]Im Normalfall Fahrzeuganzahl = Anzahl der Kunden

```
[0, 10, 0]
[0, 16, 0]
[0, 11, 15, 12, 0]
[0, 14, 0]
[0, 18, 0]
[0, 19, 0]
[0, 17, 9, 7, 8, 13, 0]
```

Listing 3.4: *ACS-VRP – Touren*

```
[0, 15, 0]
[0, 9, 0]
[0, 11, 17, 0]
[0, 13, 0]
[0, 7, 18, 0]
[0, 16, 14, 10, 12, 8, 6, 0]
```

Listing 3.5: *MACS-VRP – Touren*

Touren mit einer Gesamtlänge von 141 berechnet, dargestellt in Listing 3.4 (die 0 soll hier für die Depotknoten stehen). In seltenen Fällen wurden auch Touren mit geringerer Gesamtlänge generiert, die meisten Durchgänge blieben aber bei 141 Längeneinheiten hängen.

Der MACS-VRP Ansatz zeigt aber, dass es möglich ist, alle Kunden mit nur 6 Fahrzeugen in 120 Längeneinheiten zu beliefern, dargestellt in Listing 3.5.

Das ACS-VRP reduziert die Fahrzeuganzahl sehr schnell von 13 auf 7 (meistens gleich bei der zweiten Iteration), schafft es aber nicht, ein weiteres Fahrzeug abzuziehen. Beim MACS-VRP alleine hingegen dauert es mindestens 6 Iterationen um die Anzahl der Fahrzeuge auf 7 zu reduzieren, zieht jedoch ein weiteres Fahrzeug ab und kann so auch die Gesamtlänge weiter reduzieren.

Nun ist es ohne Probleme möglich diese beiden Ansätze zu kombinieren. Der Prozess, der mit reduzierter Fahrzeuganzahl rechnet, muss nur die ACS-VRP Funktionsweise verwenden um zu erkennen, ob wirklich alle Fahrzeuge benötigt werden. Listing 3.6 zeigt einen Ausschnitt einer Iteration.

```
# bestTour: best tour of current process
# bestLength: length of best tour of current process
# bestOverall: length of best tour of both processes (shared value)
# minV: current minimal number of vehicles (shared value)

# basic ACS loop
for count in range(1000):
  for i in range(n):
    chooseNext(graph, pheromone, remaining, tours, depots,
        maxCapacity, cap, Q)
  bestTour = checkForBestTour(graph, nodes, tours, bestTour)
  globalUpdatingRule(graph, pheromone, bestTour)
  reset(remaining, tours, nodes, ants, maxCapacity, cap)

if (not bestTour): # bestTour empty (maybe too few vehicles)
  continue

# check for best tour (this process)
if (gtl(graph, bestTour) <= bestLength):
  bestLength = gtl(graph, bestTour)

# check if it is the overall best tour of both processes
if (bestLength <= bestOverall.value):
  bestOverall.value = bestLength
  if (procID == 1): # if second process, reduce number of vehicles
    v -= 1
    minV.value = v # set global minimum value

# if first process, use the new minimal number of vehicles
if (procID == 0):
  v = minV.value + 1 # set vehicles to global minimum value

# use acs-vrp method to reduce number of vehicles
numRealTours = splitTours(bestTour, depots, procID)
if (procID == 1):
  if (v > numRealTours): # we don't need all vehicles
    v = numRealTours
    minV.value = v # set global minimum value
```

Listing 3.6: *ACS-VRP/MACS-VRP - Kombination*

Kapitel 4

Ausblick

Das "Ant Colony System" angewendet auf das Vehicle Routing Problem ist laut [15] auf gleichem Level mit vorhandenen Heuristiken und übertrifft auch von Menschen erstellte Touren bei der Tourlänge, Fahrzeuganzahl und Gesamtkosten [14].

Aufgrund der exponentiellen Laufzeit eines herkömmlichen Algorithmus zur Lösung des Vehicle Routing Problems ist eine Metaheuristik unerlässlich. Es ist oft nicht möglich, in vielen Fällen sogar unmöglich, einen "Brute-Force-Ansatz" zu verwenden und es genügt auch eine Annährung an das Optimum. Das globale Optimum ist meistens nur zweitrangig. Das MACS-VRP liefert schon nach kurzer Zeit eine erste Menge an Touren, die nach und nach immer weiter verbessert werden. So können Berechnungen nach Bedarf und Möglichkeit auch längere Zeit durchgeführt werden um näher an ein Optimum zu kommen.

Gerade beim ACS ist eine Anpassung der Parameter zur Laufzeit kein Problem. Wenn beispielsweise eine Kante aus dem Graphen entfernt werden muss, da sich eine Straße als nicht befahrbar herausstellt, werden die Touren in kurzer Zeit an die neuen Bedingungen angepasst.

In einigen Unternehmen, z.B. dem Schweizer Einzelhandelsunternehmen *Migros*, werden "Ameisen-Algorithmen" bereits erfolgreich eingesetzt. Täglich müssen für über 500 Fahrten an rund 600 Filialen optimale Routen geplant werden [14, 16]. Wie [16] berichtet, gewann Migros 2012 den "Swiss Logistics Award" für das Projekt "Transportplanung mit Ameisenlogik". Im Artikel besonders betont werden auch die Fähigkeiten, Restriktionen zu berücksichtigen und Touren dynamisch anzupassen. Tour-Kilometer, Transportkosten und Auslastung der Fahrzeuge können so minimiert und auch die CO-2 Emissionen reduziert werden. Die Jury bezeichnete das Projekt als innovativ und wegweisend.

Migros verwendet das Programm *CADIS*, das die AntRoute Algorithmen

beinhaltet, welche an MACS-VRPTW ([15]) angelehnt sind. Es wurde von *AntOptima* in Zusammenarbeit mit *Cantaluppi & Hug Sofware and Consulting* entwickelt. Das Ziel ist die Minimierung der Tourdistanz und Fahrzeuganzahl. Beachtet werden müssen Zeitfenster der Kunden und Einschränkungen der Anbindung, was dem "Time Dependent VRPTW" entspricht (siehe Kapitel 3.1). Es wird zuerst die Anzahl der Fahrzeuge und anschließend die Weglänge und Verletzung von Zeitfenstern minimiert. [14]

Schlussfolgerung

In dieser Arbeit wurde das ACS vorgestellt, und für die Anwendung auf eine Erweiterung des TSP analysiert, dem VRP. Das VRP beschreibt die Problemstellung der Transportlogistik, wo täglich Unmengen an Touren geplant werden müssen, erstaunlich genau. Der Vorteil des ACS liegt in der dynamischen Anpassungsfähigkeit und der schnellen Generierung von Touren, die nahe am absoluten Optimum liegen. Wie gezeigt wurde, kann gerade der Bereich der Transportlogistik von Ameisen-Heuristiken profitieren und nach vielen Jahren der Forschung wird "Ant Colony Optimization" erfolgreich eingesetzt.

Eidesstattliche Erklärung

Ich versichere an Eides statt durch meine eigene Unterschrift, dass ich die vorliegende Arbeit selbständig und ohne fremde Hilfe angefertigt habe; die aus anderen Quellen direkt oder indirekt übernommenen Daten und Konzepte sind als solche kenntlich gemacht.

Albstadt, 15. Juli 2013 Felix Schumacher

Literaturverzeichnis

[1] Dorigo, M., E. Bonabeau und G. Theraulaz: *Swarm Intelligence — From Natural to Artificial Systems.* Oxford University Press, 1999, ISBN 9780195131598.

[2] Beni, G. und J. Wang: *Swarm Intelligence in Cellular Robotic Systems.* In: *NATO Advanced Workshop on Robots and Biological Systems*, 1989.

[3] Dutta, K.: *How Birds Fly Together: The Dynamics of Flocking.* Resonance, 15(12):1097–1110, Dezember 2010.

[4] Dawkins, R.: *The Greatest Show on Earth: The Evidence for Evolution.* Black Swan, 2010, ISBN 9780552775243.

[5] Rabeling, C., J. M. Brown und M. Verhaagh: *Newly discovered sister lineage sheds light on early ant evolution.* PNAS, 105(39):14913–14917, September 2008.

[6] Gordon, D.: *How Ant Colonies Get Things Done.* Google Tech Talks, April 2008.

[7] Dorigo, M. und L. M. Gambardella: *Ant Colony System: A Cooperative Learning Approach to the Traveling Salesman Problem.* IEEE Transactions on Evolutionary Computation, 1(1):53–66, April 1997.

[8] Hower, W.: *Business and Computer Science.* Vorlesung, Hochschule Albstadt-Sigmaringen.

[9] Papadimitriou, C. H. und K. Steiglitz: *Combinatorial Optimization: Algorithms and Complexity.* Dover Publications, 1998, ISBN 9780486402581.

[10] Erk, K. und L. Priese: *Theoretische Informatik: Eine umfassende Einführung.* Springer, 2008, ISBN 9783540763192.

[11] Häberlein, T.: *Praktische Algorithmik mit Python.* Oldenbourg, 2012, ISBN 9783486713909.

[12] Puchinger, J.: *Optimierungsverfahren in der Transportlogistik.* Austrian Institute of Technology - Mobility Department, 2011.

[13] Gambardella, L. M., A. E. Rizzoli, E. Lucibello und R. Montemanni: *Ant Colony Optimisation for real world vehicle routing problems: from theory to applications.* Swarm Intelligence, 1(2):135–151, Dezember 2007.

[14] Gambardella, L. M., A. E. Rizzoli, F. Oliverio, N. Casagrande, A. V. Donati, R. Montemanni und E. Lucibello: *Ant Colony Optimization for vehicle routing in advanced logistic systems.* In: *Proceedings of MAS 2003 – International Workshop on Modelling and Applied Simulation*, Seiten 3–9, Oktober 2003.

[15] Gambardella, L. M., É. Taillard und G. Agazzi: *MACS-VRPTW: A Multiple Colony System For Vehicle Routing Problems With Time Windows.* In: *New Ideas in Optimization*, Seiten 63–76. McGraw-Hill, 1999.

[16] *Migros gewinnt Swiss Logistics Award 2012*, November 2012. http://www.migros.ch/de/medien/aktuelle-meldungen-2012/swiss-logistics-award-2012.html.