1. Einführung

Wir beginnen mit dem Klassiker :
Gegeben ist eine Anzahl von Städten. Alle Städte haben eine bekannte Entfernung zueinander und können in jeder beliebigen Reihenfolge bereist werden.
Gesucht ist eine Rundreise dieser Städte, in der die zurückgelegte Gesamtstrecke am Minimalsten ist und alle Städte nur einmal bereist werden.

Ob die Rundreise bei einer bestimmten Stadt beginnt oder beliebig wählbar ist, muss natürlich angegeben werden.
Wir möchten es jedoch zunächst offen lassen, in welcher Stadt gestartet wird und der Startpunkt ist frei wählbar.

Interessanter ist die Frage, ob der Startpunkt Einfluss auf die Länge der Gesamtstrecke hat.. (dazu später mehr)

Die Stadt in der gestartet wird, wird also zunächst festgelegt und die nächste Stadt muss gewählt werden.

Die offensichtlichste Lösung ist dabei, die Stadt zu wählen die am schnellsten zu erreichen ist, also die kürzeste Entfernung zu dem aktuellen Standort hat.

Diese Stadt wählen wir und wir verfahren mit diesem Ansatz bei jeder Stadt fort, bis wir alle Städte besucht haben und wir beim Startpunkt zurückgekommen sind.

Wir müssen bei dem Verfahren noch beachten, dass wir keine Stadt wählen, die schon bereist wurde, auch wenn diese zum momentanen Standort die kürzeste Entfernung hat.
Dieses Verfahren ist der *Nearest Neighbour Algorithmus* und es scheint zunächst so, als würden wir die Minimalste Gesamtstrecke zurücklegen.

Doch durch die Einschränkung, dass wir jede Stadt nur einmal besuchen dürfen, wählen wir irgendwann eine Stadt, die nicht die kürzeste Entfernung hat.

Doch damit “streichen“ wir im Prinzip eine Strecke, die theoretisch die Gesamtstrecke minimieren würde, falls man Sie in die Tour aufnehmen könnte. (Diese Strecke wird laut *Nearest Neighbour* nur dann aufgenommen, falls die folgende Stadt noch nicht besucht wurde).

Sehen Sie sich deshalb zur Begründung folgendes Beispiel an.

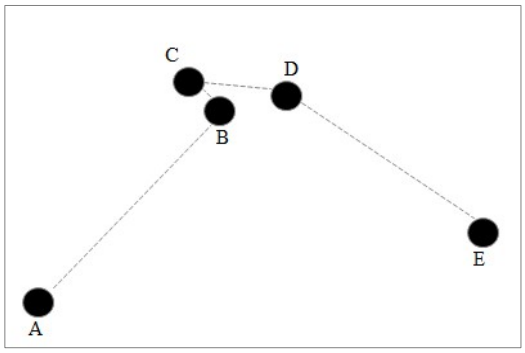


Abb 1.1

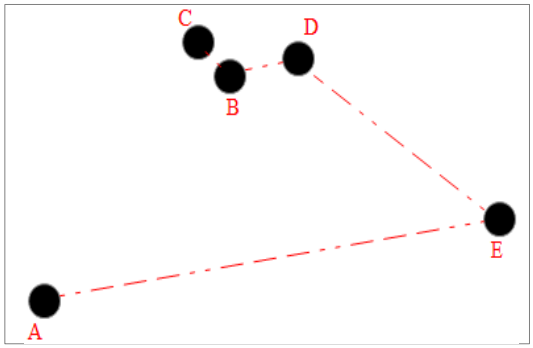


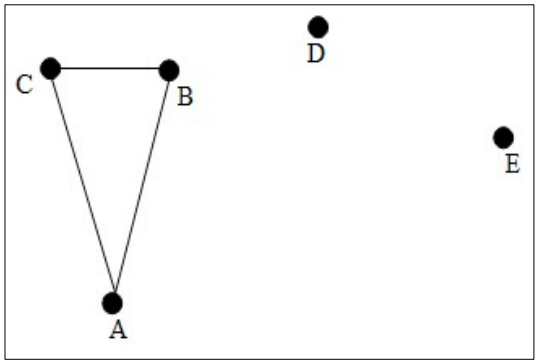
Abb 1.2

Abb 1.1 zeigt eine Tour mit den Strecken AB , BC , CD , DE. Diese Tour wird mit der Strecke EA beendet und ist das Ergebnis des *Nearest Neighbour Algorithmus*. Abb 1.2 zeigt eine andere Tour mit den Strecken AE , ED , DB , BC und der letzten Strecke CA , die nicht mit dem *Nearest Neighbour* Verfahren gebildet wurde.

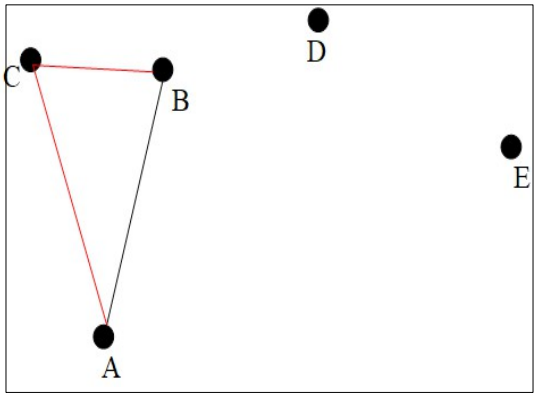
Wir können auch schreiben *tour 1 = [*AB*,BC,CD,DE,EA]* und *tour 2 =[AE,ED,DB,BC,CA]* (Die Begriffklärung von Graphen,Touren usw. folgt gleich).

Für beide Touren gilt : *EA = AE , DE = ED* und *BC* ist in beiden Touren die selbe Strecke. Es geht jetzt um die Frage ob eine Tour mit *DB* oder mit *CD* eine kürzere Gesamtstrecke bilden.

Bei dem Graphen aus Abb 1.1 ist dies zunächst nicht so offensichtlich, aber wir können diesen Graphen so umformen, dass *Nearest Neighbour* scheitert.



Zunächst verschieben wir die Punkte A B und C so, dass ein gleichschenkliges Dreieck entsteht. AB und AC haben die gleichen Längen.

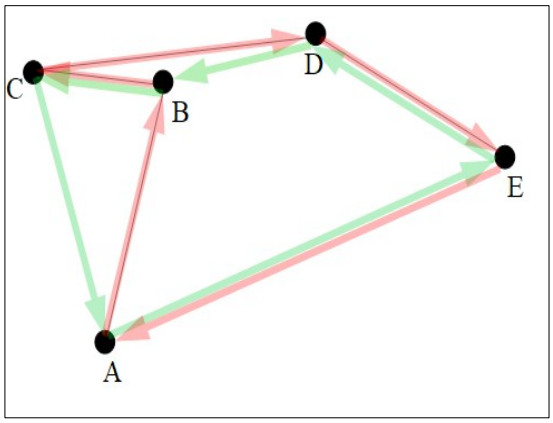


Nun verschieben wir C um einen minimalen Vektor v. Dabei entstehen 2 neue Strecken :

$BC_{neu} = BC_{alt} + v$

$CD_{neu} = CD_{alt} + v$

v ist so minimal das *Nearest Neighbour* die Strecke AB nimmt, wenn bei A gestartet wird. Zudem gilt BC < BD.

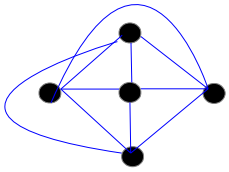


Beide Touren besitzen die Strecken AE (~ EA) und DE (~ ED).
Die grüne Tour benutzt CA die nur minmal größer ist als AB.
Der Knackpunkt liegt bei den Strecken CD und DB. Es gilt nämlich :
BC + CD > DB + BC
Die grüne Tour ist somit kürzer als die rote, welche *Nearest Neighbour* wählt.

Dabei sei zunächst nicht bekannt welcher Algorithmus *[AE,ED,DB,BC,CA]* konstruiert. Es soll *n*ur zeigen das *Nearest Neighbour* nicht die optimalste Tour konstruiert.
Nur falls der Startpunkt bei E liegt konstruiert es die grüne Route.
Das führt wiederum zu der Frage welcher Knoten der Startpunkt ist.

Der nächste Abschnitt setzt sich genauer mit Verfahren zum Konstruieren einer minmalen Tour auseinander und erläutert deren Vor,- bzw Nachteile.
Zuvor müssen noch ein paar Begriffe erklärt werden, die für die nächsten Abschnitte gelten.

- <u>Graph</u>



Ein Graph besteht aus Knoten und Kanten.
In nebenstehender Abbildung hat der Graph 4 Knoten 10 Kanten. Dieser Graph heißt *vollständiger Graph* da man von jedem Knoten alle anderen Knoten erreichen kann.
Die Kanten sind *ungerichtet* dh. das man von einem Knoten i_1 zu i_2 und wieder von i_2 zu i_1 zurück gelangt.
Bei gerichteten Graphen ist das nicht so.
Bei dem Travelling Salesman Problem betrachtet man meistens ungerichtete vollständige Graphen.

- <u>Kostenmatrix</u>

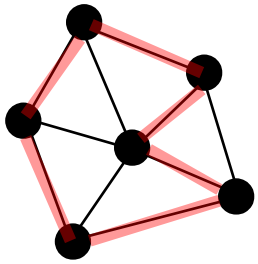
Die Kostenmatrix c speichert alle 'Kosten' eines Knotens zu seinen Nachbarknoten.
Kosten sind von Modell zu Modell unterschiedliche Maße.

c	**1**	**2**	**3**	**4**
1	0	4	1	3
2	4	0	2	8
3	1	2	0	3
4	3	8	3	0

Bei obigem Beispiel sind zb die Kosten die Entfernungen der Städte zueinander.
Es können aber auch phyiskalische Größen sein zb. Ein Routernetzwerk mit unterschiedlichen Verbindungen zwischen den Routern (Glasfaserkabel (schnell aber teuer),Kupferkabel,wlan adapter..)
Falls Knote i keine Verbindung zu einem Knoten j aufweist, so wird dies durch eine 0 definiert, also $c_{i,j}=0$ oder c(i,j)=0.
Knoten haben zu sich selbst im Allgemeinen keine Kosten, wodurch die Hauptdiagonale der Matrix Nullen hat.

- Tour $r = [1,3,5,4,2,1]$
Eine Tour ist eine Folge von besuchten Knoten, die als ersten und letzten Knoten den Startpunkt hat.
Die Knoten des Graphen sind dabei beginnend bei 1 durchnummeriert.
Bei dem Travelling Salesman Problem werden alle Knoten nur einmal besucht.
Eine Tour ist aus Sicht der Graphentheorie ein *Hamiltonkreis.*

- Hamiltonkreis
Ein Hamiltonkreis ist eine Folge von Knoten eines Graphen *G*.Er beinhaltet alle Knoten von *G.* Im Hamiltonkreis darf jeder Knoten nur einmal vorkommen.
Ob ein Hamiltonkreis für beliebige Graphen konstruiert werden kann ist im Allgemeinen nicht vorhersehbar (NP Vollständig).
Es gilt, wenn ein Graph vollständig ist hat er immer einen Hamiltonkreis.



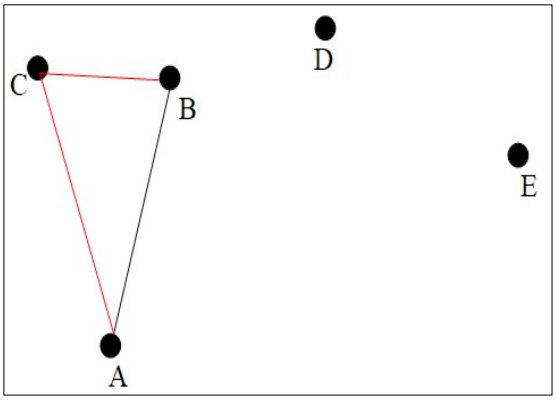
- NP Vollständig
Wenn ein Algorithmus der ein Problem lösen soll polynomiale Laufzeit besitzt $O(x^n)$ so wächst die Laufzeit für große n Werte exponentiel an und ist nicht 'exakt' lösbar.

Dies sind die wichtigsten Begriffe die im Zusammenhang mit dem Travelling Salesman Problem (absofort abgekürzt mit TSP) immer erneut auftreten.

2. Einfache Algorithmen zum Lösen des TSP

2.1 Nearest Neighbour

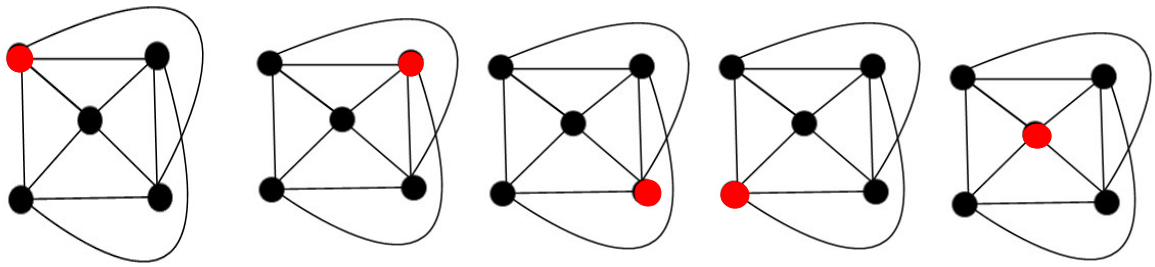
Dieses Verfahren wurde in der Einführung schon grob vorgestellt. Es ist ein sehr einfaches Verfahren führt aber wie auch schon gezeigt wurde bei jedem Graphen nicht zu optimalen Ergebnissen.
Schließlich ist man an der Minimalsten Tour interessiert. Im letzten Abschnitt haben wir gesehen, dass *Nearest Neighbour* nur dann eine optimale Tour konstruiert, wenn es im Punkt E beginnt. Dies kann es aber nicht 'wissen'.



Es startet nur bei einem Knoten und wählt dann immer die kürzeste Entfernung (absofort reden wir von Kosten).
Ist es bei dem Startpunkt zurückgekommen, bricht es ab.
Solche Algorithmen nennt man Greedy algorithms.
Im folgenden wollen den Algorithmus von *Nearest Neighbour* ermitteln.

Zunächst aber nur ein kurzer Einschub in der die Frage geklärt wird wieviele Kanten ein vollständiger Graph besitzt.

Gegeben ist ein vollständiger Graph mit *n* Knoten. Wieviele Kanten gibt es ?
Am Beispiel für n = 5 wollen wir die Frage lösen :



Der erste Knoten hat
4 unbekannte Kanten

Der zweite Knoten hat
3 unbekannte Kanten

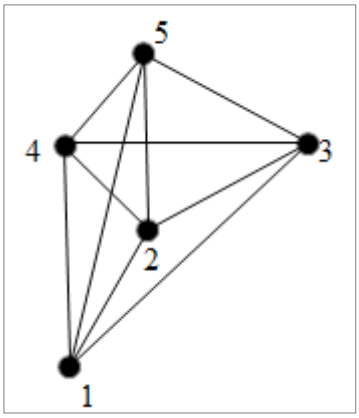
Der dritte **2** unbekannte Kanten

Der vierte **1**

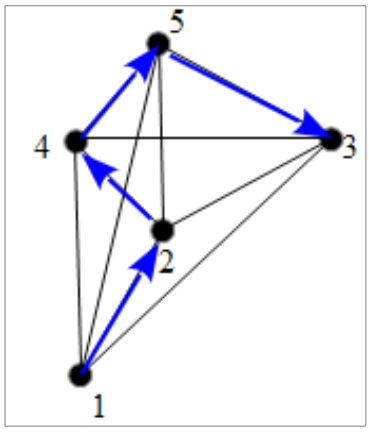
Der letzte **0**

Die Anzahl aller Kanten für einen Graph mit 5 Knoten beträgt
also $4 + 3 + 2 + 1 + 0 = 10 \rightarrow \sum_{i=1}^{n} n-i$ mit Gauß-formel : $\frac{n(n-1)}{2}$

Nun schreiben wir einen Algorithmus für *Nearest Neighbour* .Dabei betrachten wir die Kostenmatrix c eines Graphen. Startpunkt ist Knoten 1 :



	1	2	3	4	5
1	NaN	2	4	3	5
2	2	NaN	2	1	2
3	4	2	NaN	3	2
4	3	1	3	NaN	1
5	5	2	2	1	NaN



MatlabCode zum *Nearest Neighbour Algorithmus*

```
function tour = tsp_BesterNachfolger(M,V0)
%tsp_BesterNachfolger Bestimmt eine Tour mithilfe des Algorithmus 'Bester
Nachfolger'
%   M  quadratische Matrix, die Kosten aller Knoten speichert
%   V0 Knoten der Start und Endpunkt der Tour ist

[m,~] = size(M);
tour = ones(1,m);

tour(1,1) = V0;

ind=1;
vertex0 = V0;
vertex2 = V0;    %Speichert den Knoten der aktuell die gerinsten Kosten aufweist
M(:,V0) = NaN;

for i=2:m
    lowest = M(vertex0,1);
    vertex2 = 1;
    for j=1:m
        if(~isnan(M(vertex0,j)))
            if(isnan(lowest))
                lowest = M(vertex0,j);
                vertex2 = j;
            else
                if(M(vertex0,j) < lowest)
                    lowest = M(vertex0,j);
                    vertex2 = j;
                end
            end
        end
    end
    ind = ind + 1;
    tour(1,ind) = vertex2;
    M(:,vertex2) = NaN;
    vertex0 = vertex2;
end
%Strecke von VnV0 in die Tour aufnehmen
tour(1,m+1)=V0;
end
```

	1	2	3	4	5
1	NaN	2	4	3	5
2	NaN	NaN	2	1	2
3	NaN	2	NaN	3	2
4	NaN	1	3	NaN	1
5	NaN	2	2	1	NaN

	1	2	3	4	5
1	NaN	NaN	4	3	5
2	NaN	NaN	2	1	2
3	NaN	NaN	NaN	3	2
4	NaN	NaN	3	NaN	1
5	NaN	NaN	2	1	NaN

tsp_BesterNachfolger(M,V0):

- Der erste Knoten ist V0.Dieser wird in die Liste *tour* eingetragen.
- Alle Elemente der Spalten von der Kostenmatrix M deren Knoten schon in *tour* eingetragen worden sind werden mit NaN ersetzt.
- Es gibt 2 for-Schleifen:

  Die äußere Schleife läuft n-1 mal, weil zum Startpunkt n-1 Knoten noch nicht in *tour* eingetragen wurden.

  Die innere for-Schleife läuft über alle Spalten von M.
  Die geringsten Kosten werden in der Variable *lowest* gespeichert. Falls ein neuer Wert kleiner ist als *lowest* wird dieser in *lowest* gespeichert und der zugehörige Knoten (j Variable) in der variable *vertex2* gespeichert. Falls ein Wert NaN ist wird zum nächsten Knoten gesprungen.
  Falls es gleichgroße minimale Werte gibt, bleibt in *lowest* der erste gespeicherte Wert wegen `M(vertex0,j) < lowest` **erhalten.**
- Ist die innere Schleife zu Ende, wird Knoten *vertex2* in *tour* eingetragen und alle Elemente der Spalte des Knotens *vertex2* zu NaN überschrieben.

Bei n Knoten führt das Programm n-1 mal n Vergleiche aus um den aktuell geringsten Wert zu finden (auch isnan(M(i,j)) zählt zu den Vergleichen).

Insgesamt haben wir (n-1) * n = $n^2 - n$ Vergleiche.
Die Laufzeitkomplexität O(tsp_BesterNachfolger) = n^2 ist quadratisch.
Die Speicherkomplexität ist konstant (j,i,tour,ind,vertex0,vertex2,M,V0).

Vorteile/Nachteile von *Nearest Neighbour :*

Nimmt man an, dass eine Vergleichoperation auf der cpu $1*10^{-6}$ dauert kann man die Laufzeiten *l* besser einschätzen.
Für n = 10^3 beträgt $l = (10^3)^2 = 10^6 * 10^{-6} = 1$ Sekunde.
Für n = 10^6 beträgt $l = (10^6)^2 = 10^{12} * 10^{-6} = 10^6$ Sekunden → 11.57 Tage

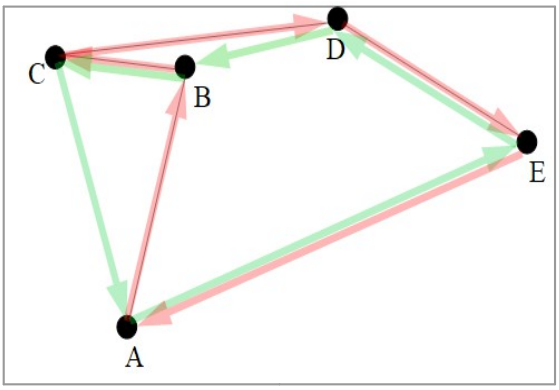
Obwohl obige Rechnungen nur grobe Schätzungen sind (andere Operationen unberücksichtigt, eventuell liegen Vergleichsoperationen im Nanosekunden-Bereich), sieht man, dass das Programm für nicht allzu große n Werte 'schnell' Touren berechnet.
Diese Touren können für weitere Optimierungsverfahren benutzt werden, um bessere Touren zu berechnen. Bei zeitkritischen Anwendungen, die

schnell eine Lösung brauchen, kann dieses Verfahren ebenso benutzt werden. Allerdings wäre bei solchen Systemen ein Programm mit konstanter oder logarithmischer Laufzeit am geeignetesten (Falls es solche gibt).

Wie schon gezeigt wurde, sind die berechneten Touren nicht optimal. Startet man bei einem bestimmten Knoten, berechnet es 'vielleicht' die Tour mit den minimalsten Gesamtkosten. Bei größeren n Werten ~ 10^3 ist dies jedoch unwahrscheinlich.
Unbrauchbar ist *Nearest Neighbour* jedoch nicht.

Zwar ist ein Startpunkt gegeben, aber weil die tour **alle Knoten** beinhaltet, kann man *Nearest Neighbour* mit unterschiedlichen Startknoten durchprobieren und dann die minimalste Tour nehmen.
Falls man die Zeit dazu hat... (n Knoten mal *Nearest Neighbour* = $O(n^3)$)
Wünschenswert wäre also ein Verfahren, dass **garantiert** die optimalste Tour findet.
Ein Programm, das genau dies kann ist der *Brute Force* Algorithmus und wird im nächsten Abschnitt vorgestellt.
Es sei jetzt schon gesagt, dass dieses Verfahren einen großen Haken hat und für die Praxis eigentlich unbrauchbar ist.

2.1 Brute Force



Am liebsten würde man die grüne Tour sofort berechnen, auch wenn A der Startpunkt ist oder ein anderer Knoten. Falls man alle möglichen Touren des Graphen kennt, ist es möglich über Vergleiche der Gesamtkosten die optimalste Tour zu bestimmen.

Genau dies tut der Brute Force Algorithmus.

Dabei reicht ein Startknoten aus um alle möglichen Touren des Graphen zu bestimmen und so die Optimalste von diesen auszurechnen. Dabei gelten die bekannten Bedingungen wie beim
Nearest Neighbour Verfahren auch :

- Der Graph ist vollständig zusammenhängend
- Es gibt zu dem Graphen eine Kostenmatrix c

Um alle Kombinationen aller Touren zu bestimmen benutzen wir eine ähnliche Herangehensweise wie die Bestimmung aller Kanten eines vollständig zusammenhängenden Graphen (siehe Einführung).