Seminararbeit:

„Lösung des Traveling Salesman Problems mit einem verteiltem Branch & Bound Algorithmus auf einem Prozessornetzwerk“

Proseminar „Algorithmen und Modelle für Planungsaufgaben“
Dozent: Dr. Ulf Lorenz
Semester: WS 2005/06
Fakultät EIM

Verfasser:

Dennis Groppe
Telefon: 05251/6932680
eMail: dennisgr@gmx.de
7. Semester
Matrikelnummer: 6212789

Inhaltsverzeichnis:

1. Einführung

Meine Seminararbeit soll sich mit der Lösung vom „Problem des Handlungsreisenden“ (in englisch: Traveling Salesman Problem, kurz TSP) beschäftigen, unter Zuhilfenahme einer Variante des Branch-and-Bound Algorithmus und einer Verteilung der Wegberechnungslast auf einen Torus mit 1024 vernetzten Prozessoren. Das Ziel ist es, eine möglichst schnelle und effiziente Lösung für ein beliebiges TSP errechnen zu lassen. Dies ist eine anspruchsvolle Aufgabe in der Informatik, da es sich bei diesem Problem um eines aus der Gruppe der NP-vollständigen Probleme handelt. Die Laufzeit ist exponentiell.

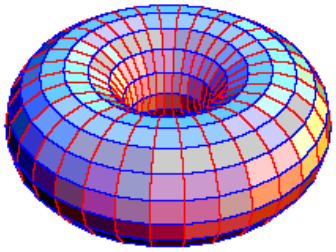

Abbildung 1: ein Ringtorus [2]

Tschöke, Lüling und Monien [1] haben auf diese Weise im Jahr 1995 Karten mit bis zu 318 Städten verarbeitet und dort jeweils eine kürzeste Rundreise errechnet. Die Seminararbeit bezieht sich hauptsächlich auf deren Ergebnisse.

Zunächst möchte ich einige Begriffe und Definitionen erklären.

Das „Problem des Handlungsreisenden“ (TSP) ist ein „recht populäres und alltägliches Optimierungsproblem aus der Kombinatorik“ [4]. Die Aufgabe ist, die Reihenfolge der Wege, die alle Städte auf einer Karte verbinden, so zu wählen, dass man möglichst kurze Strecken auf sich nimmt und am Ende auf kürzestem Wege wieder an seinen Ausgangspunkt ankommt. Es handelt sich hierbei also um das Finden einer möglichst kurzen Rundreise im Sinne von minimalen Kosten. Eine reelle Anwendung ist zum Beispiel, eine Tour mit einem LKW zu planen, die alle relevanten Lieferorte berücksichtigt und dabei möglichst kurz sein soll (Tourenplanung).

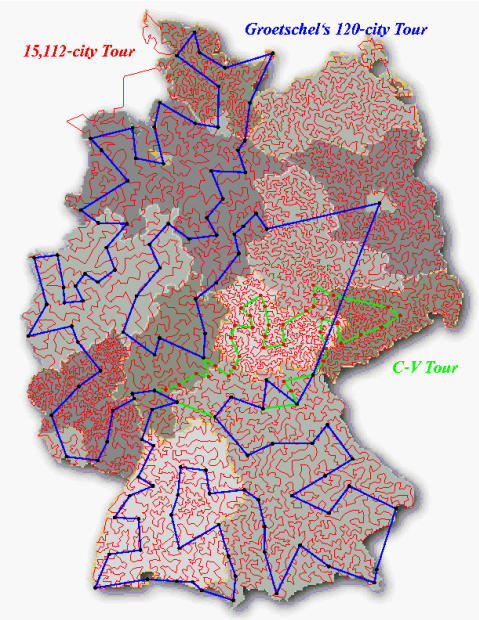

Abbildung 2: Rundreise durch deutsche Städte [3]

Im kombinatorischen Modell findet das TSP auf einem ungerichteten Graph $G=(V,E)$ statt, der aus Knoten (*V*ertices) und Kanten (*E*dges,) $e = (v_i, v_j) \in E$, die die Knoten untereinander verbinden, besteht. Die Kanten haben jeweils ein Gewicht $w(e_i)$. Die Knoten $v_1,...,v_n \in V$ sind alle gleichartig. Das Ziel ist nun, die Summe aus den Kantengewichten der zur Rundreise benötigten Kanten möglichst gering (minimal) zu halten. Die Zielfunktion ist also:

$$z_{TSP} = min(\sum_{i=1}^{n-1} w(v_i, v_{(i+1)})) + w(v_n, v_1) \quad \text{(vgl. [4])}$$

Die „Rundreise“ ist nichts anderes als ein Hamiltonkreis: Dieser Kreis im Graph enthält jeden Knoten aus V genau einmal und endet wieder im Ausgangsknoten.

Das TSP gehört zur Klasse der NP-vollständigen Probleme. D. h. die Komplexität liegt nicht unter exponentieller Laufzeit, und das gilt für alle bisher bekannte Algorithmen und Lösungswege. Es ist durch diese hohe Laufzeit in der Regel nicht möglich, in annehmbarer Zeit eine definitiv beste Lösung für ein größeres Problem zu finden. Vielmehr setzt man Heuristiken ein, um eine möglichst gute ('optimale') Lösung entsprechend der Zielfunktion des gegebenen Problems zu finden. Man nennt daher solche Probleme wie das TSP 'kombinatorische Optimierungsprobleme'.

Untersuchungen des populären symmetrischen TSP haben ergeben, dass ein guter Weg optimale Lösungen zu finden in der Benutzung des Branch-and-Bound – Algorithmus liegt. 'Symmetrisches TSP' bedeutet hier, dass das Problem ist, möglichst eine Tour von minimaler Länge zu finden. Es gibt auch noch weitere Varianten, sie sind an dieser Stelle jedoch nicht relevant.

Der Branch-and-Bound – Algorithmus „gehört zu den Entscheidungsbaum-Verfahren“ [5]. Unter Eingabe eines Graphen versucht dieser Algorithmus, den Lösungsraum und damit die Anzahl der Berechnungen „möglichst klein zu halten“ [5], indem er neu aufgespannte Zweige im Entscheidungsbaum als suboptimal identifizieren kann. Suboptimal bedeutet, dass der Algorithmus sagen kann, dass dieser Zweig keine bessere als die beste bis jetzt gefundene Lösung liefern kann. Daher ist es nun nicht mehr nötig, diesen Zweig zu gehen und zu untersuchen, er kann also abgeschnitten werden und wird nicht weiter betrachtet.

Wie der Name schon sagt, besteht der Algorithmus im Wesentlichen aus 2 Schritten: Zum einen das Branching, dem Aufteilen des Problems in zwei oder mehrere Teilprobleme (Unterprobleme), die eine „Vereinfachung des ursprünglichen Problems darstellen“ [5]. Der Branching – Schritt wird rekursiv ausgeführt und führt zur Entstehung einer Baumstruktur. Und zum anderen das Bounding, dem Beschränken der Lösungsmenge durch die oben angesprochene Identifikation von suboptimalen Zweigen im Entscheidungsbaum.

Das Branching geschieht je nach Variante des Branch-and-Bound – Algorithmus nach einer bestimmten Technik. Genutzt werden soll an dieser Stelle die Variante Best-First-Branch-and-Bound, welches ein als nächstes zu untersuchendes Unterproblem immer nach der zur Zeit besten „unteren Grenze“ auswählt. Die Auswahl erfolgt also qualitativ, wodurch erhofft wird, eine möglichst gute Lösung möglichst schnell zu erhalten. Ferner führt das Best-First-Branch-and-Bound zu sehr guter Rechenperformance, der Nachteil ist eine hohe Speichernutzung (vgl [1] S.2 Kap.1).

Für den Bounding – Schritt benutzen wir obere und untere Grenzen, um die Entscheidung zum Abschneiden eines Teilbaumes zu treffen. Dabei ist die untere Grenze das Gewicht des Weges „von der Wurzel des Suchbaumes“ (Ausgangsknoten) „bis zu einem Teilproblem“ [5]. Die obere Grenze ist der Wert der vorerst optimalen, zulässigen Lösung, d. h. die bis jetzt beste errechnete Lösung für das gegebene Problem. Falls nun die untere Grenze größer als die obere Grenze wird, kann man die Teillösung, zu der die betreffende unteren Grenze gehört, abschneiden. Sie ist schlechter als die bis jetzt beste Lösung. Sollte die untere Grenze des berechneten Teilproblems jedoch kleiner und damit besser „als die obere Schranke“ sein, „so wird diese ersetzt und als neue optimale Lösung vorgehalten“ [5].

Weiter soll im Branch-and-Bound – Algorithmus die „1-tree-relaxation“ von Held und Karp genutzt werden. Das bedeutet, dass die Unterprobleme folgendermaßen als Suchbaum aufgebaut sind: Bei unserem gegebenen Graph $G=(V,E)$ mit n Knoten, d. h. $|V|=n$, bedeutet das, dass ein 1-tree G' von G ein Spannbaum aus den Knoten und Kanten $(V\setminus\{v_1\},E)$ ist. Also besteht G' aus allen Knoten bis auf v_1, und alle Knoten haben genau den Grad (Anzahl ausgehender Kanten) von 2. Der Knoten v_1 wird dann mit 2 Kanten mit G' verbunden, was zum Beispiel so aussehen kann:

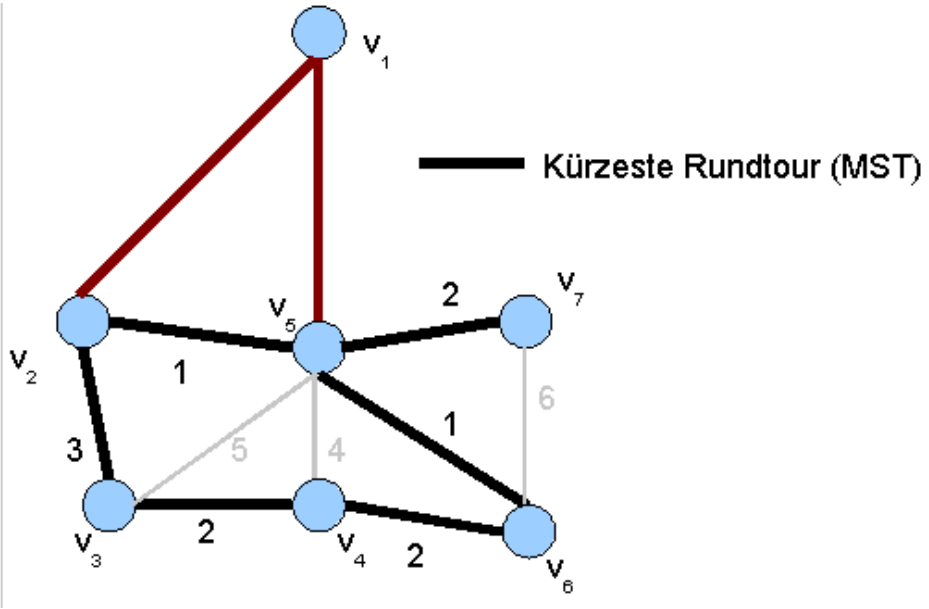

Abbildung 3: Beispiel für einen 1-tree (Zahlen sind Kantengewichte)

Per se kann jede Rundreise durch *G* ein 1-tree sein. In unserem Fall wollen wir aber im Sinne des TSP eine kürzeste Tour berechnen. Daher lassen wir auf *G'=(V\{1},E)* einen minimalen Spannbaum ('mininal spanning tree', MST) berechnen. MST bedeutet, alle Knoten in einem Graphen werden mit den Kanten minimalen Gewichts verbunden, ohne einen Kreis zu bilden. Außerdem nehmen wir die zwei Kanten mit geringstem Gewicht $w(e_i)$, um *G'* mit dem Knoten v_1 zu verbinden. Die Länge dieses kürzesten 1-trees von *G* ist dann eine 'untere Schranke' (Lower Bound, kurz LB) für die kürzeste Rundreise durch *G*. Auf diesem Prinzip basiert die Nutzung der '1-trees' im Branch-and-Bound – Algorithmus für die Lösung von TSPs (vgl. [6]), wie wir sehen werden.

Mit den Algorithmen von Prim und Kruskal können wir eine Serie aus minimalen Spannbäumen berechnen. Sowohl Prim als auch Kruskal ist in der Lage, den MST eines gegebenen Graphen zu berechnen, die Strategien und dadurch die Laufzeiten sind jedoch unterschiedlich:
Prim beginnt bei einem beliebigen Knoten und nimmt jeweils die Kante mit dem geringsten Gewicht zum sich so aufbauenden MST dazu. Die Laufzeit ist „$O(|V| \log(|V|) + |E| \log(|V|))$“ [7].
Kruskal nimmt sich nach und nach die kürzesten Kanten des gesamten Graphen, bis alle Knoten und alle die Kanten im MST sind, mit denen sich kein Kreis bilden würde. Die „Laufzeit von Kruskals Algorithmus ist $O(|V| \log(|V|) + |E| \log(|E|))$“ [8].
Aufgrund dieser Laufzeitunterschiede wird von unserem Branch-and-Bound – Algorithmus immer der Algorithmus von beiden ausgewählt, der im spezifischen Fall schneller arbeitet; bei Graphen mit vielen Kanten ist Prim effizienter.

Der nächste zu realisierende Schritt ist, die parallele Ausführung von Branch-and-Bound – Prozessen auf unserem Prozessortorus möglich zu machen und dabei eine möglichst gute Auslastung der Prozessoren und deren Arbeitsspeicher zu erreichen. Wir brauchen also eine dahingehend modifizierte Version des Branch-and-Bound – Algorithmus, die eine Verteilung auf den 1024 Prozessoren ermöglicht.

Der Algorithmus, der für die Lösung von TSPs verwendet werden soll, arbeitet vollständig verteilt. Die Kommunikation zwischen den Prozessoren erfolgt nur über Nachrichten, die untereinander verschickt werden. Durch diese Nachrichten tauschen sich die Prozessoren aus, um eine möglichst gute Lastverteilung zu erreichen. Jeder Prozessor besitzt außerdem einen eigenen Speicher, um einen Heap mit den zu bearbeitenden Unterproblemen zu speichern. Auf jedem dieser Heaps wird

dann der oben beschriebene Branch-and-Bound – Algorithmus ausgeführt. Sollte eine neue Lösung von einem der Prozessoren ausgearbeitet worden sein, wird dieses neue beste Ergebnis allen anderen Prozessoren mitgeteilt, damit diese die weiteren Berechnungen daran anpassen können.

Die Ziele sind also auf einen Blick die Folgenden:

- Einsatz eines möglichst effizienten Branch-and-Bound – Algorithmus mit 1-tree-relaxation
- möglichst optimale Lösungen für (Symmetrische) TSPs finden
- die Prozessorlast möglichst gleichmäßig auf die parallel arbeitende Hardware aus 1024 Prozessoren zu verteilen, dazu muss der Branch-and-Bound für eine parallelen Arbeitsweise angepasst werden.

2. Der Branch and Bound – Algorithmus

Das TSP definiert sich wie schon in der Einleitung erwähnt durch die Eingabe eines gewichteten, ungerichteten Graphen (vgl. [1] S.2 Kap.1) $G=(V,E)$ mit Gewichten $w(e_i)$. Außerdem sei für unseren Branch-and-Bound – Algorithmus eine „Kostenmatrix“ C_G mit Werten $c_{i,j}$, wobei $c_{i,j} = c_{j,i}$ gilt und i der Index für die horizontalen Werte der Matrix ist und j der für die vertikalen Matrixeinträge. Im folgenden soll der Branch-and-Bound – Algorithmus beschrieben werden, den wir benutzen wollen, um das in der Einleitung beschriebene TSP zu lösen.
Es wurde erkannt, dass vier Eigenschaften (vgl. [1] S.3 Kap.2) nötig sind, um einen guten, zunächst sequenziellen Best-First-Branch-and-Bound – Algorithmus zu finden:

- Wir müssen untere Grenzen (LBs) berechnen können. Wir haben mit den 1-trees bereits den Weg gefunden, um LBs zu definieren und zu berechnen. LB ist die Länge eines Unterproblems, und damit das Gewicht der kürzesten Rundreise durch einen 1-tree. Für die Lösung von TSPs sind gute LBs sehr wichtig.

- Eine Strategie, nach der wir Branchen, also das Unterteilen eines großen Problems in mehrere leichtere Teilprobleme bewerkstelligen.

- Gute Heuristiken für das Bounding. Das heißt: Wann ist es also sinnvoll zu sagen, dieser Zweig ist nicht in der Lage, mir eine bessere Lösung zu liefern als eine, die ich bereits berechnet habe?

- Methoden, um den Lösungsraum zu beschränken. Eher kleine Suchbäume für eine günstige Rundreise erstellen, um sie dann auf die große Anzahl Prozessoren verteilen zu können. Dieses parallele Vorgehen sollte bessere und schnellere Ergebnisse liefern, als wenn ein großes Unterproblem seriell berechnet würde. Man sollte so viele Tasks (Arbeitsaufgaben) wie möglich parallel berechnen lassen.

2.1 Berechnung der unteren Grenzen

Das Prinzip der 1-trees haben wir kennengelernt. Nun wollen wir gute untere Grenzen (LBs) berechnen. Dazu benutzen wir die '1-tree-relaxation'.
Jeder 1-tree beinhaltet genau einen Kreis. Auf diesem Kreis berechnen wir auch einen MST im 1-tree. Im Idealfall für das TSP besäße so jeder Knoten $v_2,...,v_n$ den Grad 2 im 1-tree G', d. h. er ist mit genau zwei Kanten verbunden.
Held & Karp formulieren auf Basis dieser Feststellung das TSP als das folgende 0-1-integer

Programm (vgl. [1], S.4 Kap.2.1):

$$P: \quad min \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} c_{(i,j)} x_{(i,j)}$$

so dass gilt: $A_1 x = 2$

x repräsentiert dabei einen Vektor *(0,1)* vom Typ Integer. *0* ist der Wert, der zu einer Kante aufmultipliziert wird, wenn er nicht für eine minimale Rundreise ausgewählt wird, *1* wenn die Kante in der gewichtsminimalen Rundreise dabei sein soll.
A_1 ist eine Adjazenzmatrix der verfügbaren Kantenverbindungen der Knoten. Die Einschränkung $A_1 x = 2$ führt dazu, dass alle Knoten den Grad 2 haben müssen.

Ein einfaches Beispiel: Eine mögliche Adjazenzmatrix wäre etwa

A	v_1v_1	v_1v_2	v_1v_3	v_2v_1	v_2v_2	v_2v_3	v_3v_1	v_3v_2	v_3v_3
v_1	0	1	1	1	0	0	1	0	0
v_2	0	1	0	1	0	1	0	1	0
v_3	0	0	1	0	0	1	1	1	0

$* \; x \quad .$

A wird mit *x* multipliziert. Ein Beispiel: Das *x* = (0,1,1,0,0,0,0,1,0) führt zum Ergebnis 2. Die Gleichung auf einen Blick:

$$\begin{pmatrix} 0&1&1&1&0&0&1&0&0 \\ 0&1&0&1&0&1&0&1&0 \\ 0&0&1&0&0&1&1&1&0 \end{pmatrix} * \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 2 \\ 2 \\ 2 \end{pmatrix} = 2$$

Das bedeutet also, die Knoten v_1, v_2, v_3 werden mit den Kanten von v_1 nach v_2, v_1 nach v_3 und v_3 nach v_2 zu einer Rundreise verbunden:

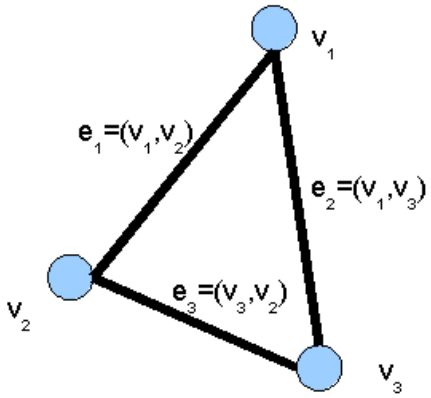

Abbildung 4: Erstellter 1-tree

Das Resultat ist eine gültige Rundreise mit Grad 2 an allen Knoten.

Nachdem wir jetzt einen 1-tree erzeugt haben, versuchen wir, die Berechnung leichter zu gestalten. Die Einschränkung, immer einen reinen Graph zu finden, wo alle Knoten den Grad 2 haben, lösen wir mit der '1-tree-relaxation' etwas auf. Allerdings: Sollte eine Lösung berechnet werden, die zwar gut ist, aber die Einschränkung 'Knoten mit Grad 2' bricht, wird das Ergebnis bestraft: Es wird ein konstanter Wert auf das Ergebnis aufaddiert. Der Raum der möglichen Lösungen wird so größer und es ist schneller möglich eine Lösung zu finden: Der Lohn für diese 'relaxation' ist, dass sich unser Problem nun nicht mehr in der Komplexitätsklasse NP, sondern in P befindet und damit die Lösung für ein Unterproblem in Polynomialzeit gefunden werden kann.

Hinzuaddiert zur Kostenfunktion wird beim Nichteinhalten der Einschränkung $A_1 x = 2$ der Term

$\pi^T*(A_1x-2)$. Das π^T in diesem Term ist der sogenannte 'Langrange Multiplikator' und steht für einen Vektor mit konstanten Werten.
Zunächst zur Bestrafung, dessen Name wegen dem Einsatz des Lagrange – Multiplikators das „Lagrange Problem“ ist. Die Zielfunktion ist jetzt etwas 'aufgeweicht', da ohne die Einschränkung zum Grad genau 2:

$$P'_0: \quad min\, C^T x + \pi^T (A_1 x - 2)$$

$$<=> P'_1: \quad min \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} (c_{(i,j)} + \pi_i + \pi_j) x_{(i,j)}$$

dabei ist C^T die Kostenfunktion, x der bekannte Vektor $(0,1)$ und der Teil hinter dem Additionszeichen (in P'_0) entspricht dem Term, der die Bestrafung darstellt. Auch möglich ist die Schreibweise in P'_1: An sich das Ursprungsprogramm, allerdings addiert mit zwei gewählten π_i und π_j.

Jede Lösung für P' ist eine untere Grenze (LB) im Sinne der Länge einer optimalen Tour (vgl. [1] S.4 Kap.2.1). Die Qualität dieser LB hängt stark von der Wahl der konstanten Lagrangen Multiplikatoren π ab. Wenn wir einen möglichst guten LB errechnen wollen, müssen wir einen Vektor π finden, der $L(\pi)$, die Menge der Lagrangen Multiplikatoren, maximiert.

Dazu wird ein Algorithmus eingesetzt, der je nachdem, wie schlecht der minimale 1-tree ist, einen höheren Wert als Bestrafung auf den LB aufaddiert. Als Eingabe bekommt der Algorithmus einen minimalen 1-tree des zu lösenden Unterproblems. Wenn dessen Gewicht nun größer ist als die Variable w_{max}, wird das aufzuaddierende π in jeder Iteration um einen Schritt erhöht, dessen Wert heuristisch bestimmten wird. Initialisiert wird $w_{max} = 0$, sodass jedes Gewicht eines 1-tree zunächst bestraft wird, nur: Je höher das Gewicht dieses 1-tree ist, desto öfters wird die Iteration durchgeführt, in der π jedesmal um einen Schritt erhöht wird. Es ist also vorteilhaft, einen 1-tree mit wirklich gutem minimalen Gewicht in den Algorithmus einzugeben; je besser, desto geringer fällt die Bestrafung aus. Das Ergebnis ist, dass 'schlechte' minimale 1-trees durch die zusätzliche Bestrafung mit einer noch geringeren Wahrscheinlichkeit beim Aufbau der TSP-Rundreise genutzt werden. Wir nehmen also vornehmlich wirklich gute minimale 1-trees dazu, um schnell eine optimale Lösung zu erhalten.

Der Branch-and-Bound – Algorithmus versucht nun also zunächst, eine Lösung für P' zu finden, die dann auch potentielle Lösung für das (gegenüber P' verschärfte) Problem P ist.
Wenn der Branch-and-Bound – Algorithmus dann eine optimale Lösung L für P' findet und dann diese Lösung L auch eine Lösung von P ist, dann folgt daraus: L ist optimal für P.
Andersherum, wenn mit x eine gültige Lösung für P gefunden wurde, dann muss dieses x auch gültig sein für P'.

Es wurde bewiesen, dass der Branch-and-Bound – Algorithmus mit '1-tree-relaxation' sehr effizient ist.

2.2 Branching

Beim Branching wird beim TSP die Menge der möglichen, erreichbaren Lösungen in Untermengen unterteilt, die ihrerseits auch wieder in Untermengen unterteilt werden und so weiter. Das Ziel ist es, die unteren Grenzen für jedes Unterproblem *SP* zu verbessern, um am Ende ein sehr gutes, eben optimales Endergebnis aus zusammenhängenden Rundreisen zu erhalten.
Die Strategie soll nun wie folgt sein: Aus der Lagrange – Optimierung haben wir einen minimalen

1-tree erhalten. Wenn alle Kanten den Grad genau 2 haben, sind wir fertig. Sollte es jedoch mindestens einen Knoten geben, der den Grad ≥ 3 hat, muss man Auswahlen treffen, um überall Grad 2 herzustellen. Man unterteilt die Kanten eines *SP* in benötigte (engl. '*R*equired') Kanten und in nicht benötigte, also für das Bilden einer Rundreise verbotene Kanten (engl. '*F*orbidden'). Diese halten wir in Menge *R* für die benötigten und in Menge *F* für die verbotenen Kanten fest.
Ein Unterproblem $SP_{R,F}$ besteht immer aus einem Satz aller möglichen 1-trees $T_{R,F}$, wo nun die Kanten in *R* für den Aufbau der Rundreisen benutzt werden, die Kanten in *F* jedoch nicht benutzt werden. Es gilt: Dieses *SP* hat dann eine untere Grenze in Form des Gewichtes $w_{T_{R,F}}$ der kürzesten Rundreise durch den 1-tree $T_{R,F}$. Für das TSP braucht man nur zwei Regeln, um die Mengen R und F aufzufüllen:

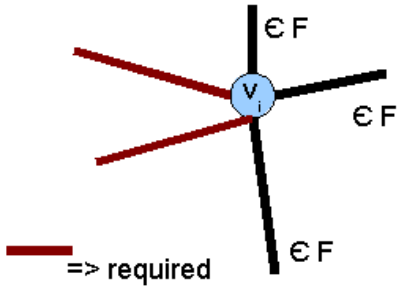

Abbildung 5: benötigte und verbotene Kanten (Regel 2)

1. Wenn zwei Kanten, die mit einem Knoten verbunden sind, als benötigt bekannt sind (beide in *R*) – und das bedeutet, dieser Knoten hat mit diesen beiden Kanten den Grad 2 – können alle anderen Kanten, die mit diesem Knoten verbunden sind, verboten werden. Sie werden der Menge *F* hinzugefügt.

2. Dementsprechend: Wenn ein Knoten nur noch mit zwei möglichen Kanten verbunden werden kann, die nicht schon vorher als verboten deklariert wurden, müssen diese Kanten auf jeden Fall als benötigt angesehen werden. Sie werden der Menge *R* hinzugefügt.

Sollte nun ein Knoten v_i im zur Zeit besten 1-tree einen Grad ≥ 3 haben, kann man aufgrund Regel 1 sagen, dass von diesen (mehr als zwei) Kanten mindestens zwei dabei sind, die noch nicht als benötigt festgesetzt (fixiert) sind. Wenn der Grad als einfaches Beispiel den Wert 3 haben sollte, kann man mit Sicherheit sagen, dass 2 Kanten in die Menge *R* als benötigt und 1 Kante in die Menge *F* für die verbotenen Kanten einsortiert werden muss. Jetzt brancht man an dieser Stelle nach folgendem Muster: Das Ausgangs – Unterproblem $SP_{R,F}$ wird aufgeteilt in drei disjunkte Unterprobleme SP_1, SP_2 und SP_3, die alle Möglichkeiten berücksichtigen, zwei von den drei Kanten aufzunehmen und eine zu verbieten.

$SP_1 = SP_1(R \cup e_1, e_2, F)$ Kanten 1 und 2 werden aufgenommen, Kante 3 verboten
$SP_2 = SP_2(R \cup e_1, F \cup e_2)$ Kante 1 wird aufgenommen, Kante 2 verboten => 3 oder weitere noch zu prüfen
$SP_3 = SP_3(R, F \cup e_1)$ Kante 1 wird verboten, 2, 3 oder weitere noch zu prüfen

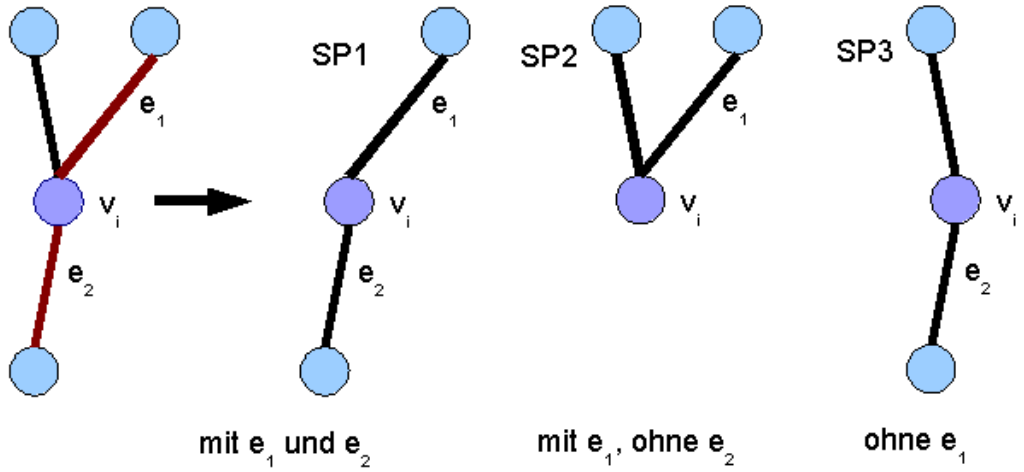

Abbildung 6: Veranschaulichung der drei Unterprobleme

SP_1 kann man sofort unter den Tisch fallen lassen, wenn der Knoten v_i bereits eine benötigte Kante hat – nähme man dann e_1 und e_2 dazu, hätte man ja 3 verbundene Kanten und damit Grad 3. SP_1 kann dann als nicht der optimalen Lösung zuträglich angesehen werden. In so einer Situation brancht man nur in zwei Unterprobleme (zweite und dritte Möglichkeit). Der Grad des Suchbaums eines Unterproblems variiert also immer zwischen 2 und 3 und wird im Laufe der Berechnung immer näher in die Richtung gehen, dass alle Kanten den Grad 2 haben.

Im Resultat soll das Vorgehen dazu führen, die untere Schranke, den lower bound, in ihrem Wert möglichst nah an die optimale Lösung heranzuführen. Je mehr Kanten als 'benötigt' markiert werden und je mehr Knoten im Suchbaum den Grad 2 haben, desto näher kommt man diesem Ziel (vgl. [1] S.6 Kap.2.2). Am Ende werden im optimalen Fall alle Knoten Grad 2 haben, damit eine Rundreise (im Sinne des Hamiltonkreises) entstanden ist und diese mit den richtigen, vom Gewicht her minimalen Kanten gebildet wurde, die nichts anderes als die 'benötigten' Kanten sind.

Wonach Kanten gewählt werden, ist nun bekannt. Zuletzt wurde das folgende heuristische Vorgehen als effizient zum Knotenwahl zur Erschließung von Unterproblemen erkannt (nach Volgenant und Jonkers, vgl. [1] S.6 Kap.2.2): Beim zur Zeit besten 1-tree wählt man einen Knoten aus. Wenn es möglich ist, wählt man diesen Knoten so, dass er mit einer Kante, die schon als benötigt einsortiert wurde, verbunden ist. Branchen mit Grad 2 ist hierbei vorzuziehen. Wenn man jetzt immer noch Möglichkeiten bei der Auswahl eines Knoten hat, nimmt man den Knoten, an dem möglichst wenig Kanten verbunden sind. So kann man erreichen, dass man vom zur Zeit besten 1-tree ausgehend weitere Knoten erschließen kann und dabei den Suchbaum klein hält und außerdem dabei eine Tendenz zu Knotengrad 2 beibehält.

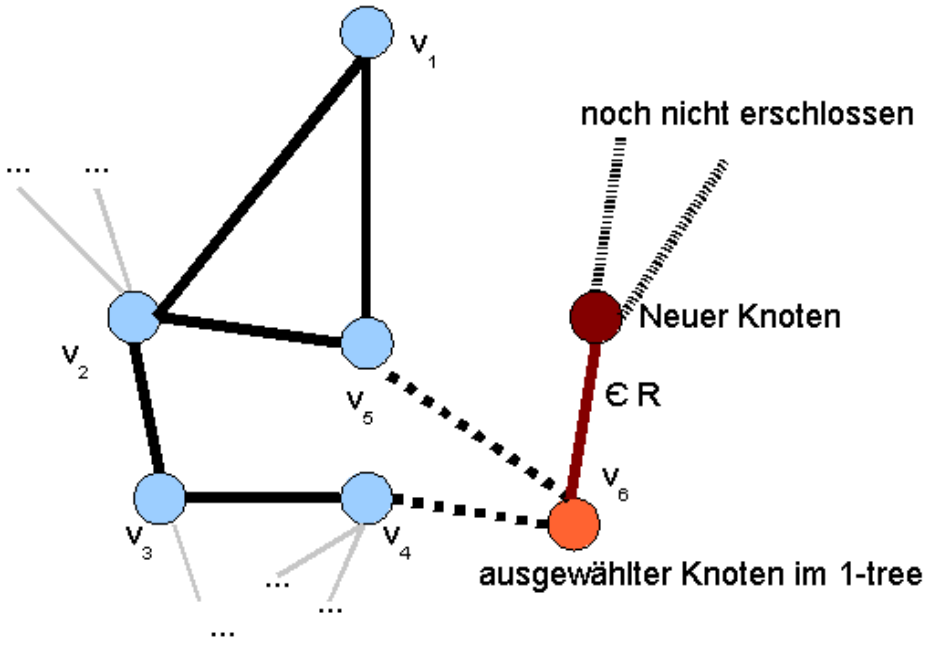

Abbildung 7: bester 1-tree und ein neuer Knoten

2.3 Berechnung von oberen Grenzen

Zusätzlich zu der unteren Grenze benötigen wir außerdem eine obere Grenze (UB), diese „stellt den bis jetzt besten gefundenen Wert für den Zielfunktionswert einer optimalen Lösung von *P*“ [9] dar und wird „heuristisch“ [9] bestimmt. „Schlimmstenfalls gilt UB = ∞“ [9]; die UB wird „im Laufe

des Verfahrens“ „verringert bis zum Minimum.“ [9]

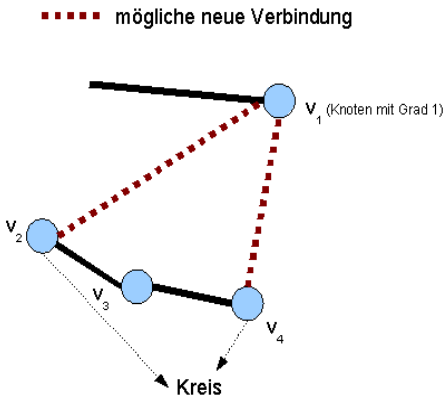

Abbildung 8: Beispiel für neue obere Grenzen

Die oberer Grenze soll dazu dienen, den Speicherverbrauch in unserem Prozessornetzwerk gering zu halten. Sie bestimmt, wann Zweige im Suchbaum abgeschnitten werden. Das soll immer dann geschehen, wenn die aktuelle untere die obere Grenze übersteigt: $LB > UB$. Dann kann man sagen, dieser Weg hat jetzt schon einen größeren Wert als der zuletzt beste Wert, dann braucht man diesen Pfad nicht mehr einzuschlagen – er kann nur noch schlechter werden.

Dazu berechnen wir aus dem zur Zeit besten 1-tree eine mögliche gültige Rundreise gemäß dem Ziel des TSPs (vgl. [1] S.7 Kap.2.3). Bedingung dabei ist, dass wir nur 1-trees betrachten, die aus einem Kreis ausschließlich mit Knoten des Grades höchstens 2 bestehen, um P auch wirklich zu erfüllen. Ein Beispiel: Wir wählen dazu für jeden Zweig des Suchbaumes einen Knoten mit Grad 1 und einen Knoten v_3, der schon im Kreis ist. Zwei weitere Knoten sind schon im Kreis mit v_3 verbunden (v_2 und v_4). Nun verbinden wir entweder v_2 oder v_4 mit dem Knoten, der vorher der Grad 1 hatte, je nachdem, welche Kante das niedrigere Gewicht hat. Der Knoten v_1 ist nun mit dem Kreis verbunden und hat wie gewünscht sowohl den Grad 2 als auch die minimal-gewichtige Verbindung zum vorher bereits bestehenden Kreis.

Die heuristische Lösung ('nehme die bis jetzt bekannte kürzeste Verbindung zu einem bekannten Stück dazu') wird jetzt in der Summe des Gewichtes des Weges um die minimale Kante aus (v_1, v_2) und (v_1, v_4) erhöht und wir erhalten so eine neue obere Grenze.

2.4 Reduktion des Lösungsraumes

Das TSP ist ein großes, sogenanntes 0-1-Integer Problem, d. h. man hat entweder zu entscheiden, ob eine Kante in der optimalen Lösung eine Rolle spielt (sie bekommt den Wert 1) oder nicht (Wert 0). Es ist daher wünschenswert, dass man schnell sagen kann, ob es Variablen bzw. in unserem Fall Kanten gibt, bei denen es zur Findung einer optimalen Lösung nicht von Bedeutung ist, ob deren Wert 0 oder 1 ist. Diese Kanten braucht man dann natürlich nicht in die Berechnungen zur optimalen Lösung miteinbeziehen. Wenn man keine Reduktion des Lösungsraumes um diese Variablen vornimmt, war es auch auf großen Prozessornetzwerken nicht möglich, TSPs mit über 150 Städten (Knoten) zu lösen (vgl. [1] S.7 Kap.2.4). Es würde zuviel Zeit damit verbracht, irrelevante Berechnungen auszuführen. Die Aufgabe ist also das Auffinden dieser überflüssigen Kanten genauso wie das Finden von Kanten, die in einer optimalen Lösung enthalten sein müssen (vgl. [1] S.7 Kap.2.4). Im Grunde haben wir mit der Einteilung der Kantenmenge in eine benötigte Menge R und eine verbotene Menge F nichts anderes getan. Um diese Methode zu verbessern und noch mehr Kanten ausschließen zu können, sollen nun noch zwei zusätzliche Methoden zum Einsatz kommen:

1. 'Edge Exchange': wenn eine Kante dem zur Zeit minimalem 1-tree hinzugefügt wird und diese übersteigt die aktuellen oberen Grenze, kann diese Kante sofort als überflüssig eingestuft werden. Die Qualität dieser Auswahl in R und F hängt direkt mit der Qualität der oberen Grenze zusammen – desto besser die obere Grenze, desto größer die Menge der R und F zuordneten Kanten (vgl. [1] S.7 Kap.2.4)

2. Identifikation von Ecken, die nicht optimal sind: Dies geschieht durch Einsatz des heuristischen 2-opt-Algorithmus: Dieser ist in der Lage, eine bereits berechnete TSP – Tour noch weiter zu verbessern. Der Ablauf ist iterativ. 2-opt „wählt“ in jeder Iteration zwei beliebige Kanten in der zu optimierenden Rundreise „aus und entfernt diese“ [10]. Das Ergebnis aus der Entfernung sind „zwei Teiltouren, die auf genau zwei verschiedene Weisen wieder zu einer Gesamttour verkettet werden können“ [10]. Nun wird geprüft, ob es nun zwei verschiedenen Möglichkeiten gibt, die Rundreise wieder zusammenzusetzen: Die eine „dieser beiden Möglichkeiten führt zu der ursprünglichen und die andere zu einer neuen Tour“ [10]. Sollte nun die „neue Tour kürzer“ [10] sein, „so behält der Algorithmus diese und verwirft das Original“ [10]. „Dieses Verfahren wird iterativ so lange fortgesetzt bis keine zwei Kanten mehr vorhanden sind“ [10], auf die man diese Methode anwenden kann, so dass sich eine Verbesserung ergeben kann. Die auf diese Weise „konstruierte Tour wird 2-optimal genannt und zurückgegeben.“ [10] Das Ergebnis ist, dass so jede berechnete, optimale Tour auch 2-optimal (vgl. [1] S.7 Kap.2.4)) ist.

3. Paralleles Rechnen - Balancieren der Last

Durch den Einsatz eines Torus von 1024 Prozessoren soll erreicht werden, dass man eine effiziente Erhöhung der Rechenleistung für das TSP bekommt. Im Idealfall soll eine Problemlösung auf *n* Prozessoren *n*-mal so schnell wie auf einem einzelnen Prozessor ausgeführt werden (*n*-faches Speedup). Jedem Prozessor steht ein eigener, unabhängiger Arbeitsspeicher von 4 Megabytes zur Verfügung. In diesem wird der Heap aus den Knoten der Suchbäume (Teilprobleme), die der spezifische Prozessor bearbeiten soll, gespeichert.

In diesem Kapitel möchte ich zeigen, welche Ziele durch die Parallelisierung erreicht werden können, wie man die Kommunikation zwischen den Prozessoren und die Kriterien zur Lastverteilung effizient gestaltet.

3.1 Ziele

Bei der Berechnung eines TSP ändert sich der Suchbaum ständig, ebenso die obere und untere Grenze. Dadurch ändert sich dann auch die Situation der Prozessor- und Speicherauslastung. Daher müssen wir dynamisch auf diese Veränderungen reagieren und jederzeit Rechenlast und Unterprobleme neu verteilen können. Parallel zum verteilt laufenden Branch-and-Bound – Algorithmus muss also auch ein Algorithmus laufen, der die Rechenlast dynamisch verteilt. Dieser Algorithmus soll

- die Zeiten minimieren, in denen Prozessoren keine Arbeit haben und im Leerlauf sind

- möglichst wenig Mittel zur Kommunikation aufwenden: Lastinformationen und Rechenlast selber (Teilprobleme = 'Workload') muss durch das Prozessornetzwerk gesendet werden, um die Arbeit zu verteilen. Problem dabei ist, dass bei wenig Kommunikation die Gefahr von Leerläufen größer ist.

- idealerweise untersucht unser Best-First-Branch-and-Bound – Algorithmus bei jedem Schritt das Unterproblem mit dem besten lower bound (ist das erste Heap – Element). Dass immer auf jedem der 1024 Arbeitspeicher das erste Heapelement das mit dem zur Zeit minimalen lower bound ist, wäre zwar optimal, ist aber schwer zu realisieren. Auch wenn mehrere Unterprobleme

den gleichen minimalen lower bound haben sollten, sind das nicht alle Unterprobleme die auf den 1024 Prozessoren verteilt wurden. Je weniger Suchbäume mit nicht mehr minimalen unteren Grenzen berechnet werden, desto besser (Ziel: minimaler 'search overhead'). Allerdings gibt es bei diesem Ziel das Problem, dass bei einer besseren Information der Prozessoren über neue minimale unteren Grenzen der Kommunikationsaufwand steigt.

Ein Mittelweg aus diesen drei Zielen wurde als die mit dem besten Speedup ermittelt (vgl. [1] S.8 Kap. 3.1).

3.2 Gewichtsfunktionen

Die Last ist im parallelen Branch-and-Bound – Algorithmus nichts anderes als Unterprobleme, die beim Branchen erzeugt wurden und dann auf den Heap mit den Knoten gelegt wurden (vgl. [1] S.8 Kap. 3.1). Um dem Prozessornetzwerk jetzt den Befehl zur Verteilung zum richtigen Zeitpunkt geben zu können, benötigt man zwei Lastfunktionen: Eine erfasst die Qualität der Last und richtet sich nach den zur Zeit besten unteren Grenzen. Die andere zeigt die Quantität der Last an, nämlich die Anzahl der Heapelemente auf einem Prozessor – also die Nutzung des Arbeitsspeichers.

Die Definition der Lastfunktionen für das TSP ist wie folgt:
Qualität der Last: $w_{LB}(p_i) = min(LB(sp)) \, wobei \, sp \in H_{pi}$
Quantität der Last: $w_{\#}(p_i) = |H_{pi}|$
wobei $|H|$ die Anzahl der Heapelemente in Speicher eines Prozessors darstellen, p_i ist ein Prozessor mit Index i. *LB (sp)* repräsentiert die untere Grenze der Unterproblems *sp*.

Es ist nun wünschenswert, dass man sowohl qualitativ (die untere Grenze ist auf vielen oder am besten auf allen Prozessoren gleich) als auch quantitativ (die Menge der Heapelemente und damit der Speicherauslastung) eine gleichmäßige Auslastung der Prozessoren erreichen kann. Beide Lastfunktionen müssen also beim Balancieren der Last berücksichtigt werden.

3.3 Kriterien zur Lastverteilung

Im Prozessortorus hat jeder Prozessor vier Nachbarn. Wenn ein Prozessor 'merkt', dass die eigene Rechenlast um einen bestimmten Wert größer ist oder der eigene Speicher voller ist als der seiner Nachbarn, sendet er Unterprobleme an seine Nachbarn. Sobald sich die eigene Heapgröße ändert, informiert der Prozessor seine vier Nachbarn. So kann man am Besten die gesteckten Ziele erreichen: maximale Prozessorauslastung, minimaler search overhead, wenig Kommunikation und wenig Leerlauf (vgl. [1] S.9 Kap. 3.1).

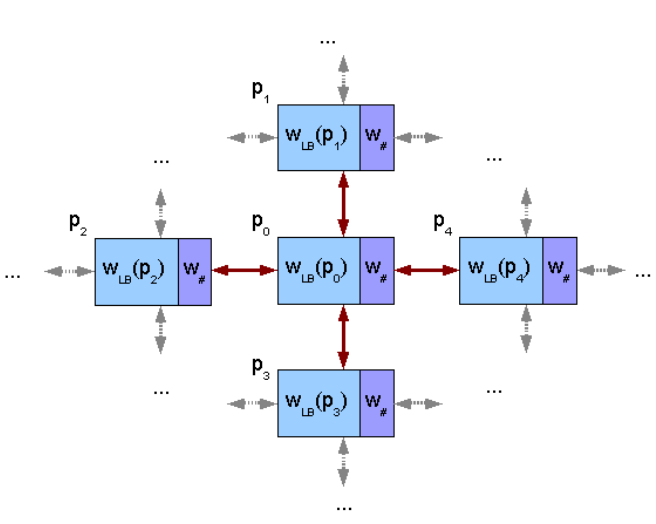

Abbildung 9: jeder Prozessor hat 4 Nachbarn

Dadurch, dass die Kommunikation nur mit den vier Nachbarn stattfindet, können wir von „local decision“ sprechen: Es gibt keinen zentralen Prozessor, der die Last verteilt, sondern jeder Prozessor ist nur an seinem lokalen, unmittelbaren Umfeld interessiert. Auch die Auslagerung, erfolgt, wenn es nötig ist, lokal, also nur bei den eigenen Nachbarn. Wir sprechen von „local migration space“ (vgl. [1] S.9 Kap. 3.2). Sowohl Entscheidungen zur Lastverlagerungen als auch

die Auslagerungen selber passieren also im eigenen, lokalen Umfeld.

Der parallel zum Branch-and-Bound – Algorithmus laufende Lastverteilungsalgorithmus arbeitet in folgender Weise:
Das Ziel ist zusammengefasst, dass die Heapgewichte (Lastfunktionen) w_{LB} und $w_{\#}$ eines Prozessors p_i und seiner vier Nachbarn $\{p_1,...,p_k\}$ auf einem möglichst gleichen Niveau gehalten werden sollen. Wann wird nun Last ausgetauscht? Wir müssen eine maximale Differenz Δ festlegen, sodass immer dann Last ausgetauscht wird, wenn die eigene Last um dieses Δ von der Last der Nachbarn abweicht.
Formal ausgedrückt:

$$\Delta = \max_{j \in \{1,...,k\}} |w(p_i)-w(p_j)| \qquad \text{(vgl. [1] S.10 Kap. 3.2)}$$

Wenn jeder der 1024 Prozessoren auf diese Weise dynamisch mit seinen vier Nachbarn Last austauscht und Veränderungen der eigenen Auslastung kommuniziert, erreicht man eine gute Verteilung über das gesamte Netzwerk. Vier Parameter sind nötig, um die Lastverteilung zu steuern:

Δ_{LB}: Prozessor p_i initiiert oder nimmt an Lastaustausch mit seinem Nachbarn p_j teil, falls sich die minimalen unteren Grenzen auf ihren Heapelementen w_{LB} um mehr als Δ_{LB} unterscheiden. Für die Berechnung hat sich die Wahl von $\Delta_{LB} = 1{,}0$ als sinnvoll erwiesen, so kann search overhead so gut wie möglich verhindert werden und das Netzwerk gut ausbalanciert werden (vgl. [1] S.10 Kap. 3.2).

$\Delta_{\#}$: Hier ebenso: Prozessor p_i initiiert oder nimmt an Lastaustausch mit seinem Nachbarn p_j teil, falls die Anzahl seiner Heapelemente $w_{\#}$ sich um $\Delta_{\#}$ von denen seines Nachbarn unterscheidet. Hier wählen wir $\Delta_{\#} = 3$ für gut balancierten Arbeitsspeicher. Das heißt, der Grad des vom Branchen erzeugten Suchbaums darf 3 nicht überschreiten, sonst werden Teile dieses Suchbaums abgegeben.

info.delay ist eine Anzahl von Zeiteinheiten. Wenn eine bestimmte Zeit *info.delay* keine Informationen über die lokale Lastsituation, bezogen sowohl auf w_{LB} als auch $w_{\#}$, zum Nachbar gesendet wurden, wird dieser nach Ablauf dieser Zeitspanne informiert. Dieser Wert wird so gewählt, dass er der durchschnittlichen Zeit zur Berechnung eines Unterproblems entspricht. So wird sichergestellt, dass nach der Abarbeitung eines Unterproblems möglichst zeitnah neue Informationen zur Last der Umgebung vorliegen.

$send_{LB}.rate$ und $send_{\#}.rate$: Bleibt noch mitzuteilen, wie viel Last jeweils an den Nachbarn abzugeben ist. Das wird mit den beiden „rate“ - Werten realisiert. Der Wert von $send_{LB}.rate$ ist die Anzahl der Unterprobleme, die während einer Balancierungsoperation verteilt werden und liegt idealerweise zwischen mindestens 1 und höchstens 3. $send_{\#}.rate$ – Werte sind Prozentwerte. Wenn man beispielsweise 50% Last abgibt, heißt das, die beiden austauschenden Prozessoren machen „halbe-halbe“ und beide haben danach die gleichen Lastwerte. Die Prozentzahl bezieht sich dabei auf die Last von p_i + Last von p_j : wenn ein Prozessor 75% an seinen Nachbarn abgibt, verbleiben folglich nur 25% der ursprünglichen Last von sich selber und der Last seines Nachbarn auf diesem Prozessor. Für die beste Balancierung erwies sich ein Prozentsatz von 65% als sinnvoll.

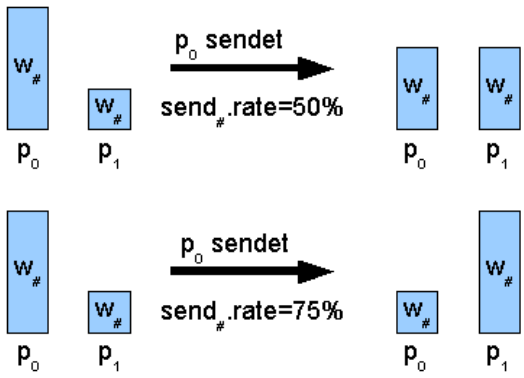

Abbildung 10: send#rate

3.4 Nachrichten

Wir haben jetzt Kriterien zur Lastverteilung definiert. Nun brauchen wir noch einen Modus, wie wir die Kommunikation zur Lastverteilung realisieren. Das soll durch vier verschiedene Nachrichten geschehen, durch die die Prozessoren sich austauschen können (vgl. [1] S.12 Kap. 3.2).

IDLE: Ein Prozessor sendet die *IDLE* – Nachricht, wenn sein Branch-and-Bound Prozess im Leerlauf läuft, also keine Rechenarbeit hat.

SOLUTION: Diese Nachricht enthält obere Grenzen, die bis jetzt besten berechneten Lösungen. Diese sind nötig, um durch den Abgleich unterer und oberer Grenze über das Bounden zu entscheiden.

INFO: Enthält die aktuellen Heapgewichte w_{LB} und $w_{\#}$ des sendenden Prozessors.

WORK: Enthält ein Unterproblem als Suchbaum.

Beim Beginn der Operation, also der Lösung eines TSPs, berechnet einer der Prozessoren p_0 ein erstes Unterproblem. Alle anderen Prozessoren senden *IDLE*, weil sie auch bereit sind, an diesem Problem mitzuarbeiten. Sie halten sich ständig bereit, Last anzunehmen. Also sendet der p_0 Teile seines Unterproblem weiter an seine direkten vier Nachbarn und die Last veteilt sich auf das Netzwerk.

Im Verlauf der Operation warten die Prozessoren jeweils auf ein Unterproblem. Bei Eingang eines solchen wird es vom Lastbalancierer – Prozess des empfangenden Prozessor angenommen. Falls der Branch-and-Bound – Prozess *IDLE* ist, wird es an ihn weitergegeben und ein Branch durchgeführt, LBs werden für die neu entstandenen Unterprobleme berechnet und dann werden diejenigen Unterprobleme, die nicht durch Brechen der Grenzen gleich als nicht optimal aussortiert werden, an den eigenen Lastbalancierer-Prozess gesendet, der daraufhin das Management des Heaps übernimmt, d. h. das Hinzufügen und Einordnen auf dem aktuellen Heap mit Unterproblemen. Wenn eine neu berechnete Lösung besser ist als die bisher berechneten, wird diese Lösung an die Nachbarn weitergegeben. Wenn alle Schritte abgeschlossen sind, werden die Nachbarn über die neue Lastverteilung informiert und außerdem wird eine *IDLE* – Nachricht an die Nachbarn gesendet, um anzuzeigen, dass man wieder im Leerlauf ist und damit bereit, um weitere Berechnungen zu tätigen. Nach der Lastbalancierung wird außerdem noch geprüft, ob die Operation terminiert wurde, d. h. alle Berechnungen abgeschlossen wurden (vgl.[1] S.12 Kap 3.2).

3.5 Regeln zur Lastverteilung – genauer Ablauf

(1) Wenn nach Abschluss einer Berechnung eine *IDLE* – Nachricht gesendet wird, wird auch versucht, sofern es vorhanden ist, das Unterproblem mit der minimalen, lokalen unteren Grenze zu übermitteln. Dieses Unterproblem befindet sich immer ganz oben auf dem Heap mit den Unterproblemen, es ist das 1. Element.

(2) Zuerst wird nach w_{LB} balanciert, dann nach $w_{\#}$.

(3) Falls *info.delay* verstrichen ist, und ein Prozessor merkt, dass er eine bessere minimale untere Grenze hat als seine Nachbarn, sendet er das zweite Element auf dem Heap (die zweitbeste untere Grenze) zu seinem Nachbarn und behält das Beste selbst – das beste

Unterproblem wird für den eigenen Branch-and-Bound – Prozess behalten, damit sich der Prozessor nicht nach Ende seiner Berechnungen die beste untere Grenze irgendwo bei seinen Nachbarn zurückholen muss sondern gleich auf der Basis des eigenen besten Ergebnis weiterarbeiten kann. Das reduziert wiederum den search overhead (vgl. [1] S.12 Kap 3.2).

(4) Nachdem eine Balancierung nach w_{LB} abgeschlossen ist und $w_{\#}$ noch zu sehr unbalanciert sein sollte, müssen wir erreichen, dass wir durch die neue Balancierung nicht gleich wieder das gerade neu balancierte w_{LB} umzuverlegen. Dazu gibt der Prozessor mit der zur Zeit größeren Speicherbelegung die Elemente von seinem Heap ab, die sich ganz am Ende befinden. Auch hier wird die Arbeit an den schon guten unteren Grenzen, die sich oben am Anfang des Heaps befinden, nicht gestört und man erreicht trotzdem den Ausgleich in der reinen Speicherbelegung, auch wenn die abgegeben Heapelemente die mit den schlechtesten unteren Grenzen des abgebenden Prozessors sind.

(5) Nach Lastbalancierungen, sowohl von w_{LB} als auch von $w_{\#}$, werden auch alle vier Nachbarn informiert.

(6) Ein Prozessor speichert auch immer Informationen zur Last auf seinen lokalen Nachbarn in einem Array. Wenn ein Prozessor die Informationen w_{LB} und $w_{\#}$ von einem seiner Nachbarn zugesandt bekommt, macht der empfangende Prozessor eine dementsprechende Aktualisierung auf seinen gespeicherten Informationen zur Last der Nachbarn. Auf Basis der neuen Werte wird dann wiederum eine Balancierung ausgeführt, die Nachbarn werden nach dessen Abschluss noch einmal über die neue Lastbalancierung informiert.

4. Fazit

4.1 Ergebnisse

Zum Ende der Seminararbeit möchte ich einige der Erfahrungen von Tschöke, Lüling und Monien zusammenfassen, die sie aufgrund ihrer Arbeit mit der vorgestellten Technik zur Berechnung von TSPs gemacht haben.

- wenn bei einem gegeben Problem nur wenige Unterprobleme berechnet werden müssen, um eine Lösung bekommen, kann man nicht mit einem großen Speedup rechnen (Parallelisierung wird nicht ausgelastet) (vgl. [1] S.13 Kap.4). TSPs mit weniger als 4000 Unterproblemen können die 1024 Prozessoren nicht voll auslasten.

- zu Beginn der Berechnungen, bevor genug Unterprobleme berechnet sind um sie auf alle Prozessoren zu verteilen, entstehen Leerlaufzeiten (viele Prozessoren sind noch *IDLE*). Diese Leerlaufzeit kann man nicht verhindern, da es immer etwas Zeit braucht, bis alle Prozessoren Arbeit bekommen haben. Aber man kann sie für andere, vorausgreifende Berechnungen nutzen: Man kann die Prozessoren obere Grenzen berechnen lassen – gute obere Grenzen habe einen positiven Effekt auf die unteren Grenzen, die während des Bounden berechnet werden. Dadurch läßt sich die Anzahl der von vorne herein als nicht mehr zu überprüfenden Zweige vergrößern. Eine weitere Möglichkeit zum Nutzen der Leerlaufzeit wäre eine Zusammenfassung mehrerer Prozessoren zu einem 'Cluster' und diesen zu einer schnelleren Berechnung eines einzelnen Unterproblems zu benutzen (vgl. [1] S.13 Kap.4). Zu Beginn der Berechnungen hat der

Lastbalancierer auch mehr zu tun bis zu dem Zeitpunkt, wo die Last auf das ganze Netzwerk verteilt ist. Wenn wir jedoch dann eine gleichbleibende Lastverteilung einmal erreicht haben, wird das Ende der Berechnungen (Ende der TSP-Lösung) nahezu gleichzeitig auf allen Prozessoren im Netzwerk erreicht, was sehr zeitsparend ist (vgl. [1] S.15 Kap.4).

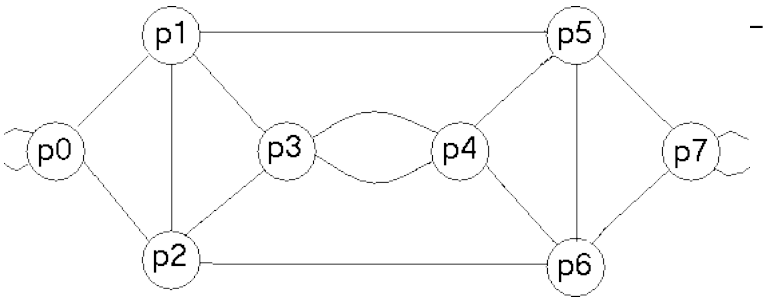

Abbildung 11: De-Bruijn - Anordnung mit 8 Prozessoren (vereinfacht) [11] – (p0 ist bidirektional verbunden mit p7)

- Zur Auswahl der Art des Prozessortorus: In Anbetracht der genutzten Nachbarschaftskommunikation der Prozessoren liefert ein Torus mit kleinem Durchmesser (Anordnung 'De Bruijn' oder als zweidimensionaler Torus, ein Ring ist eher unvorteilhaft) das beste Speedup.
- Wenn neue obere Grenzen gefunden wurden, werden die Heapelemente mit schlechteren oberen Grenzen gelöscht. Auch das trägt zur guten Verteilung der Last bei (vgl. [1] S.15 Kap.4).
- Die Kommunikation gestaltet sich wie gewünscht sehr moderat. In einer Zehntelsekunde werden pro Prozessor durchschnittlich 3 Nachrichten über die Lastsituation und 0 bis 1 Unterprobleme versendet. Beim Finden einer neuen oberen Grenze steigt die Kommunikation natürlich an, was jedoch nur von kurzfristiger Dauer ist, bis die Last wieder wie gewünscht verteilt ist (vgl. [1] S.15 Kap.4).

4.2 Schlußbemerkung

Tschöke, Lüling und Monien haben mit ihrer Problemlösung zum Traveling Salesman Problem eine offensichtlich geschickte und effiziente Methode gefunden, um möglichst optimale Lösungen für TSPs in ausreichender Zeit zu finden. Auch wenn die Anzahl der Städte (also der Knoten) auf 318 mögliche beschränkt war, damit das Konzept effizient nutzbar ist, sind die Möglichkeiten von daher bemerkenswert, dass das TSP zwar ein sehr populäres, aber auch nach wie vor ein Problem mit größer Komplexität und damit hoher Laufzeit ist. Da die Ausarbeitung, mit der ich mich beschäftigt habe, aus dem Jahre 1995 stammt, sind eventuell mit den stetig steigenden Rechenleistungen heutiger Prozessoren die Lösung von größeren TSP möglich.

Die effiziente Lösung des Problem kann auch in Alltagsanwendungen hohen Stellenwert besitzen, z. B. in der Planung von möglichst kurzen Routen im Sinne von Rundreisen. Ein guter Ansatz zur effektiven Lösung des TSP ist also auf jeden Fall eine Bereicherung, wann immer eine Berechnung von Rundreisen vonnöten ist.

5. Literatur:

[1] S. Tschöke, R. Lüling, B. Monien: *Solving the Traveling Salesman Problem with a Distributed Branch-and-Bound Algorithm on a 1024 Processor Network*. Proc. of the 9th International Parallel Processing Symposium (IPPS '95), Santa Barbara, CA, pp. 182-189, April 1995.

[2] http://gregegan.customer.netspace.net.au/DIASPORA/02/02.html

[3] http://tsp.gatech.edu

[4] Wikipediaartikel „TSP“, http://de.wikipedia.org/wiki/Problem_des_Handlungsreisenden

[5] Wikipediaartikel „Branch and Bound“, http://de.wikipedia.org/wiki/Branch_and_bound

[6] Webseite http://www.cs.toronto.edu/~neto/research/research.old.html#research_hk1tree

[7] Vorlesungsfolien Datenstrukturen und Algorithmen Kapitel 22 von Prof. Blömer, SS2004, Universität Paderborn

[8] Vorlesungsfolien Datenstrukturen und Algorithmen Kapitel 23 von Prof. Blömer, SS2004, Universität Paderborn

[9] Vorlesungsfolien Operations Research, FH Rhein-Sieg, WS 2005/06, Kapitel 3 – Branch and Bound

[10] http://www.mathematik.uni-dortmund.de/lsv/lehre/ss2005/kombopt/progaufg5.php (zum 2-opt – Algorithmus)

[11] http://www-lehre.informatik.uni-osnabrueck.de/~pa/skript/3_2_11_de_Bruijn.html