Vorlesung Real-Time Systems

4. Distributed RTS: Scheduling & Allocation

Übersicht

1. Ziel diese Kapitels
2. Einführung
3. Problembeschreibung
4. Globales Scheduling
5. Schedulability Analysis in Distributed Systems
6. Allocation

1. Ziel dieses Kapitels

Bisher sind in dieser Veranstaltung ausschließlich Ein-Prozessor-Systeme betrachtet. Bei **verteilten Echtzeitsystemen**

- kann eine Anwendung sich über **mehrere Knoten** (d.h. Prozessoren) eines Netzwerks erstrecken oder
- mehrere Anwendungen teilen sich ein **Kommunikationsmedium**.

In beiden Fällen kommt es zu einer zeitlichen Wechselwirkung zwischen Netzwerk und Prozessor.

In diesem Zusammenhang werden wir insbesondere betrachten:

- **Scheduling** (und **Analyse**) in verteilten RTS
- **Allocation**, d.h. die Zuweisung einer Task zu einem Prozessor des Netzwerks

1. Ziel dieses Kapitels

Zusätzliche Literatur-Angaben

Diese Kapitel setzt sich i.W. zusammen aus den folgenden Einzel-Publikationen:

[6] Ken Tindell, Alan Burns, Andy Wellings: ***Allocating Hard Real Time Tasks + (An NP-Hard Problem Made Easy)***, 1992

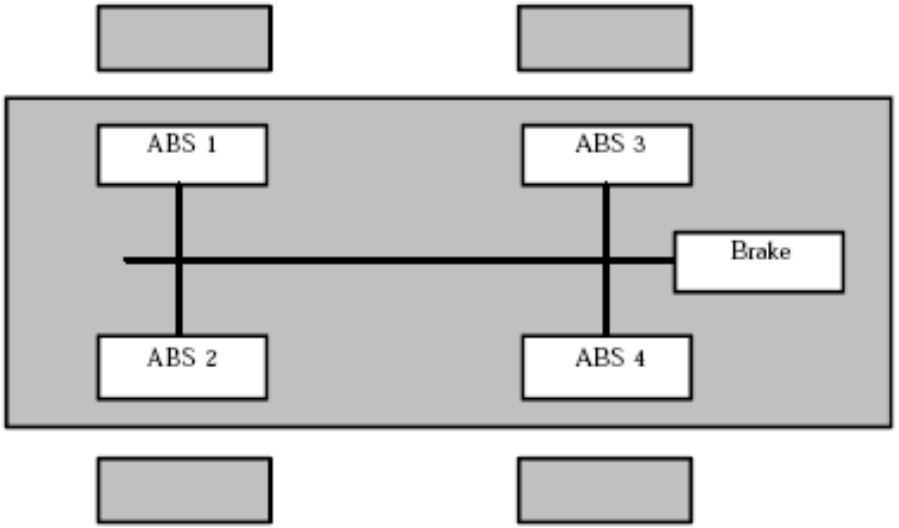
[7] K. Tindell, J. Clark. ***Holistic schedulability analysis for distributed real-time systems***, 1994

[8] Peter Altenbernd, Hans Hansson: ***The Slack Method: A New Method for Static Allocation of Hard Real-Time Tasks***, 1997

2. Einführung

Ein **Beispiel** für eine verteilte RTS-Anwendung ist ein intelligentes **ABS-System** [2],

- in dem jede Bremse durch einen eigenen Knoten geregelt wird und
- es einen weiteren Knoten gibt, der sich um das Bremspedal kümmert:



Die Anwendung besteht also aus einem verteilten Regel-Algorithmus, der die Bremskraft für jedes Rad anhand der lokal vorliegenden Daten und der Stellung des Bremspedals ausrechnet.

2. Einführung

Es gibt verschiedenen **Gründe**, eine verteilte Lösung zu bevorzugen, obwohl die Komplexität dadurch erhöht wird:

- Viele Anwendungen sind physikalisch verteilt und müssen daher ein Kommunikationsmedium benutzen.
- Das Preis-Leistungsverhältnis ist besser als bei einer Ein-Prozessor-Lösung.
- Das System lässt sich schrittweise ausbauen (Skalierbarkeit).

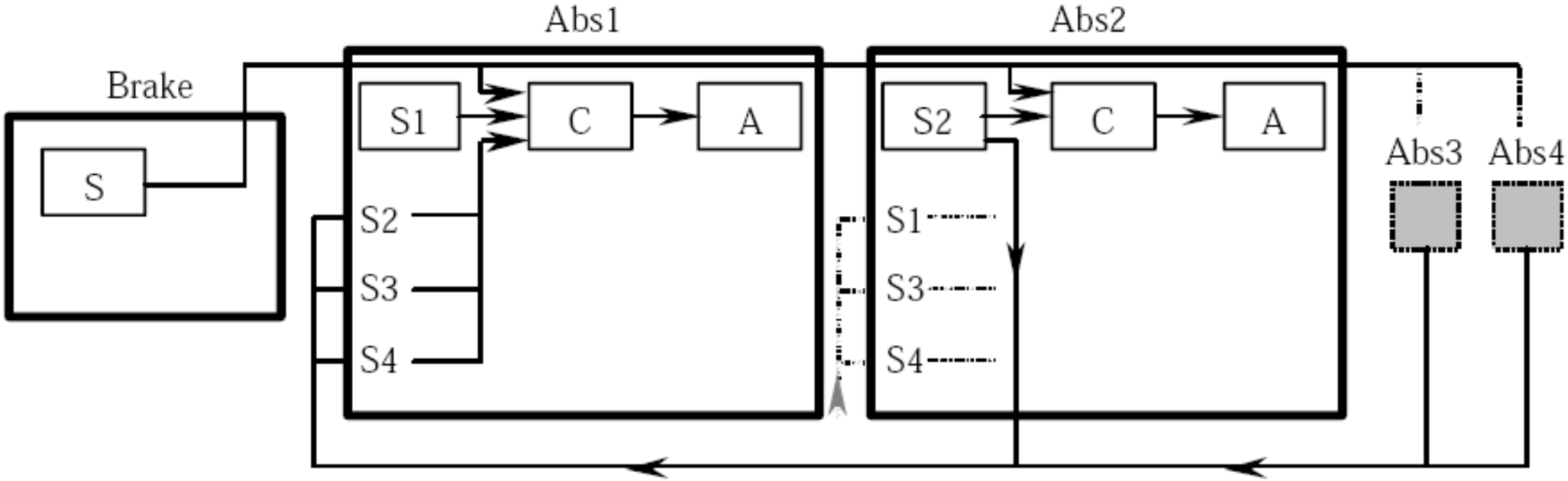
Das zu lösende **Problem** besteht daraus,

- dass die Anwendung auf mehreren Knoten gleichzeitig ausgeführt und dabei alle Realzeit-Bedingungen (Deadlines) eingehalten werden müssen.
- Die Spezifikation der Anwendung unterscheidet sich dabei (eigentlich) nicht von einer Ein-Prozessor-Lösung. Der **Unterschied** besteht daraus,
  - dass die Tasks bestimmten Knoten **zugewiesen** werden (müssen) und
  - dass die Benutzung des Kommunikationssystems zu Verzögerungen führt, die beim Design berücksichtigt werden müssen.

Scheduling (und Analyse) werden also komplizierter.

2. Einführung

Die folgende Abbildung illustriert das **verteilte Rechner-System** für die o.g. ABS-Anwendung inklusiver aller Tasks und deren Relationen untereinander [2]:



Dabei wird davon ausgegangen, dass alle Task dieselbe Periode benutzen und dass jede Präzedenz zugleich eine Kommunikation ausdrückt (jeweils dargestellt durch einen Pfeil):

- Task **S** auf Knoten **Brake** liest den gewünschten Bremsdruck vom Pedal
- Tasks **S1...4** ermitteln die Bremsverzögerung des jeweiligen Rads
- Diese Werte benutzt die Regelung (Task **C**) zur Berechnung neuer Größen

3. Problembeschreibung

Es geht i.W. um zwei **primäre Ziele**:

- die **Einhaltung der Präzedenzen**
- die Einhaltung der Deadlines

Darüber hinaus ist es aber bei der Zuteilung von Tasks zu Prozessoren (**Allocation**) von Bedeutung irgendeine **Nebenbedingung** zu optimieren **z.B.**

- die Last des Kommunikationssystems zu minimieren oder
- möglichst wenig Prozessoren zu benutzen

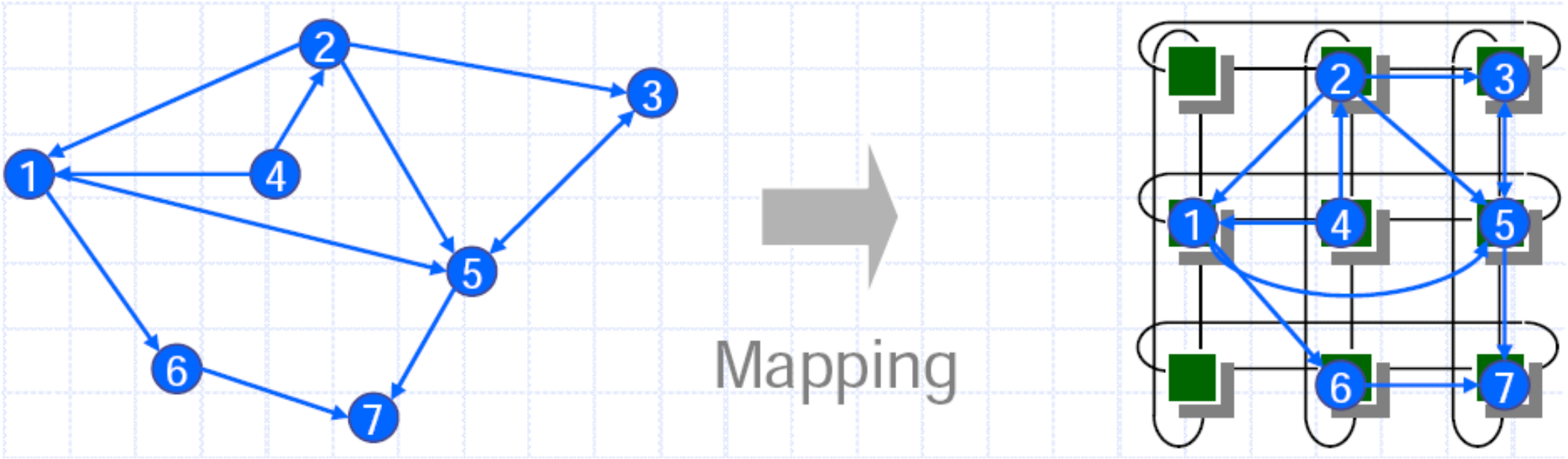
Im Allgemeinen könnte dabei jede Task auf einem beliebigen Prozessor ausgeführt werden. In der Praxis ist es aber oft so, dass es eine natürliche Zugehörigkeit mancher Tasks zu einem bestimmten Prozessor gibt, weil sich dort ein Sensor oder an Aktuator befindet, der von der Task benutzt werden soll. Daher braucht man nur eine Teilmenge der Task wirklich zu allokieren.

Die Wahl des „richtigen“ Prozessors für ein Task ist nicht einfach, denn wird die Task unglücklich platziert, kann dies weitreichende Folgen für den Rest des Systems haben, weil ggf. das Timing des gesamten Systems durcheinander gerät.

3. Problembeschreibung

Algorithmisch gesehen ergibt sich folgende **Problemstellung**:

Ein **Task-Graph** wird abgebildet auf einen **Prozessor-Graphen**



wobei sich mehrere Tasks einen Prozessor teilen können. Der Prozessor-Graph kann dabei auch eine einfache Bus-Kommunikation ausdrücken.

Sollen dabei – wie gefordert – alle Deadlines eingehalten und eine Nebenbedingung optimiert werden, ergibt sich eine **NP-hartes Problem**, d.h. für das Finden des Optimums entsteht **exponentieller Aufwand**.

3. Problembeschreibung

Somit kann das Problem nur ansatzweise, d.h. durch den Einsatz einer **Heuristik** angegangen werden.

Allocation und Scheduling können dabei sowohl integriert (d.h. globales Scheduling) oder strikt getrennt (d.h. lokales Scheduling) betrieben werden. Beides wird im Folgenden betrachtet werden.

Darüber hinaus hat Allocation auch immer etwas mit Ausfall-Sicherheit zu tun:

- Fällt ein Knoten aus können ggf. dessen Tasks auf andere Prozessoren verteilt werden (Dynamisches Re-Allocation).
- In sicherheitskritischen Systemen werden oft mache Tasks doppelt ausgelegt (Redundanzen). Solche Tasks dürfen dann nicht auf dem selben Prozessor verarbeitet werden.

Die Liste dieser und anderer Restriktionen kann recht lang sein, und mag z.B. die Betrachtung des Speicherbedarfs von Tasks einschließen.

4. Globales Scheduling

Es gibt zahlreiche globale Scheduling-Ansätze (oft auch Multiprocessor Scheduling genannt), aber alle arbeiten nach dem selben Prinzip:

- Es gibt eine globale, priorisierte Warteschlange, aus der
- die Task mit der höchsten Priorität entnommen und
- auf einem freien Prozessor zu Ausführung gebracht wird.

Alle Scheduling-Entscheidungen werden also (wie bei einem Parallelrechner) auf globaler Ebenen getroffen, d.h. Allocation wird gleich mit erledigt. Es gibt viele verschiedene Möglichkeiten die Prioritätsvergabe zu regeln, z.B. gemäß RMS.

Die Vorteil eines solchen Verfahrens liegt in seiner hohen Verarbeitungs-Geschwindigkeit. Allerdings ist dieser Ansatz **wenig praktikabel**, wenn es darum die o.g. Nebenbedingungen zu berücksichtigen. Zudem ist die Wahrscheinlichkeit recht hoch, dass der Algorithmus **keine gültige Zuweisung** findet (d.h. eine Zuweisung, bei der alle Deadlines eingehalten werden).

Aus diesem Grund wird globales Scheduling hier nicht weiter betrachtet.

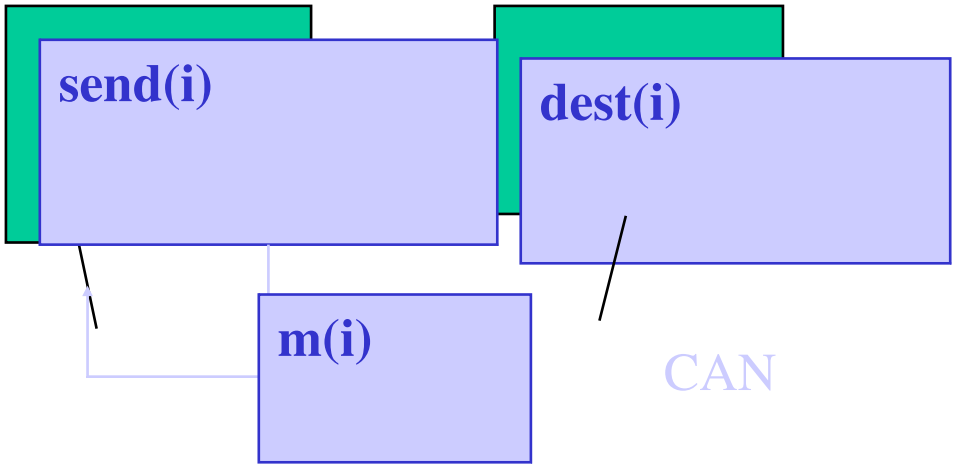
5. Schedulability Analysis in Distributed Systems

Werden Allocation und (lokales) Scheduling von einander getrennt, besteht der Bedarf, eine Zuteilung auf ihre Gültigkeit zu überprüfen. Dabei ist zu beachten, dass sich die Latenz-Zeiten von Tasks und Botschaften (Messages) gegenseitig beeinflussen. Dazu kann/muss die vorgestellte RTA erweitert werden. Dies kann allerdings nicht ohne Kenntnis des benutzten Kommunikationsprotokolls geschehen, da es hier recht viele unterschiedliche Ansätze gibt.

5.1 Holistische Analyse

Die holistische Analyse [7] berechnet nicht nur die Response-Zeiten der Tasks auf den Prozessoren sondern auch die entsprechenden Response-Zeiten von Messages auf dem Kommunikationsmedium. Im Falle des **CAN-Bus** (-> später) ist dies ebenfalls eine prioritätsgetriebene Ressource, deren Handhabung sich kaum von einem Prozessor unterscheidet.

Die Response-Zeiten von Tasks und Messages hängen allerdings voneinander ab, wie folgendes **Beispiel** zeigt:



Hierbei gilt:

$$R_{dest(i)} = R_{m(i)} + w_{dest(i)}$$

und

$$R_{m(i)} = C_{m(i)} + w_{m(i)} + \tau_{bit} + R_{send(i)}$$

(Die genaue Bedeutung der Formeln ist für den Moment uninteressant.)
Die holistische Analyse führt dafür eine weitere Iterationsebene ein, mit der schrittweise die vorhandenen Abhängigkeiten in Werte für **Jitter** (vgl. Kap. 3) umgerechnet werden. In jeder Iteration werden dabei

- getrennte RTAs für die Prozessoren und das Netzwerk durchgeführt,
- um am Ende mit möglicherweise geänderten Werten für **R** und **J** in die nächste Runde zu gehen.

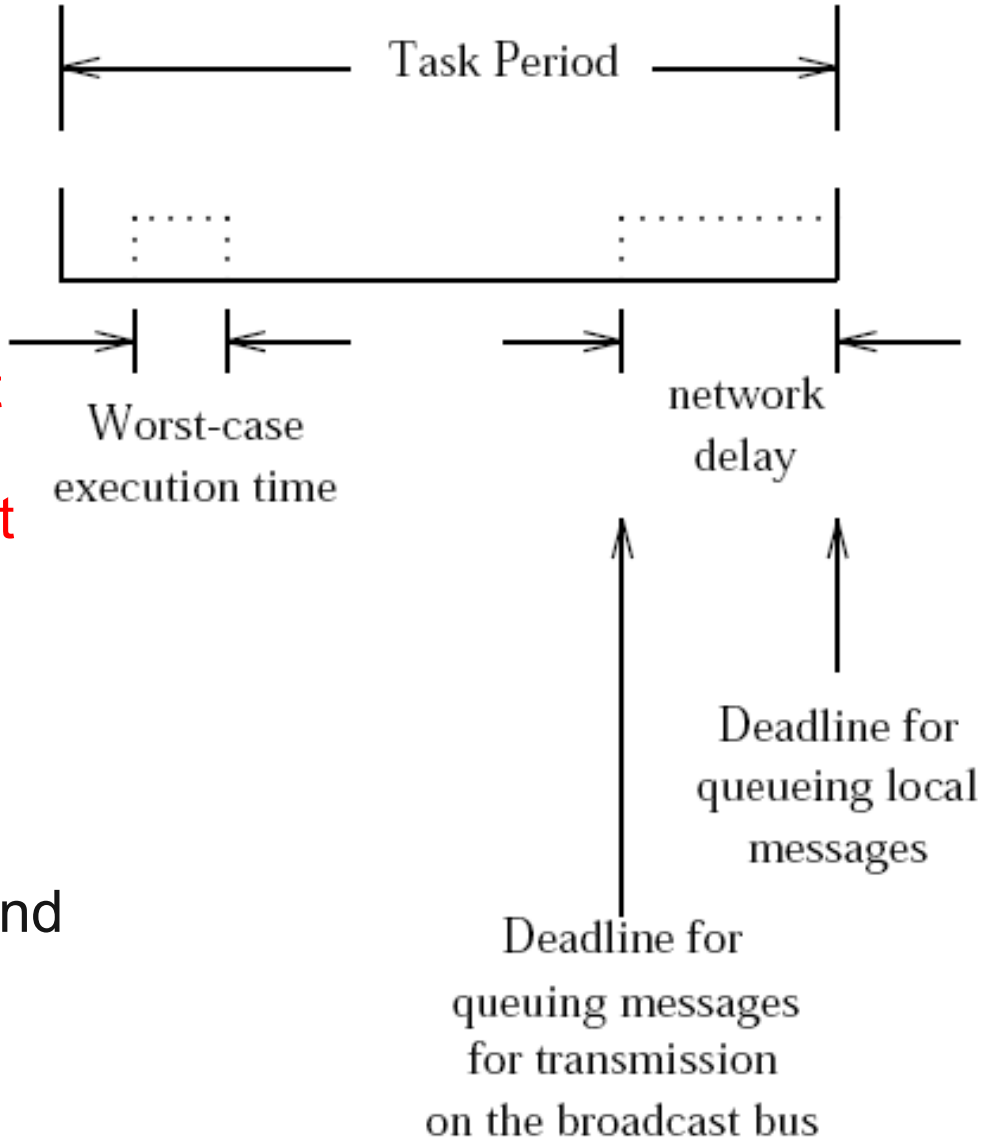
Die Details der holistische Analyse übersteigen allerdings den Rahmen dieser Vorlesung.

5.2 Deadline-Monotonic Scheduling in Distributed Systems

Eine einfacher Trick [5,7], die komplexe holistische Analyse zu umgehen, ist die **Latenz-Zeiten des Netzwerks bei der Deadline einer Task** zu berücksichtigen.
Dabei wird davon ausgegangen, dass eine sendende Tasks die Nachricht bis zum nächsten Perioden-Beginn ausgeliefert hat und somit dem Empfänger vorliegt.
Wenn die **Übertragungsdauer N** ist, ergibt sich die **Deadline** eine Task als

D = T – N.

Liegen Sender und Empfänger auf dem selben Knoten, gilt: **N=0**
Das Scheduling kann jetzt gemäß **DMS** durchgeführt werden (mit einer entsprechend einfachen Analyse).

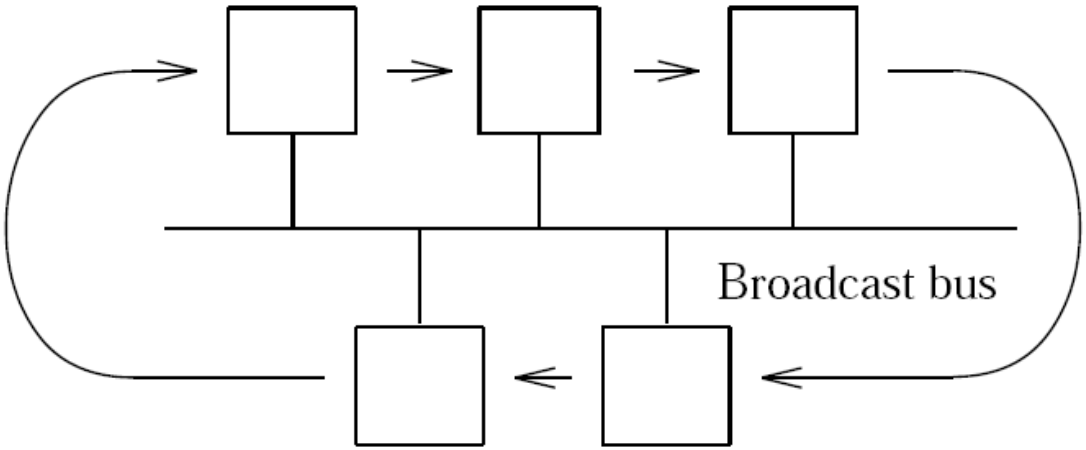


5. Schedulability Analysis in Distributed Systems

5.3 Berechnung der Übertragungsdauer bei Token-Ring

Je nach Charakteristik des unterliegenden Netzwerk-Protokolls ergeben sich unterschiedliche Berechnungsmodelle für die Ermittlung der Übertragungsdauer. Bei der hier vorgeschlagen Methode wird **ein** maximaler Wert für **alle** Messages eines Prozessors bestimmt.
Dies wird **exemplarisch** für ein Netzwerk gemacht, das nach dem Token-Ring-Prinzip erfolgt [6]:

- Dabei darf ein Prozessor (und somit eine Task) nur auf den Bus senden, wenn er im Besitz eines **Tokens** ist.
- Jeder Prozessor darf das Token nur eine bestimmte Zeit (Token Holding Time – **THT**) lang behalten, um alle Messages, die bis dahin in einer Queue gesammelt worden sind, versenden zu können, bevor es reihum an den nächsten weitergegeben wird.



- Die THT ist zwar durch einen gegeben Maximal-Wert beschränkt. Wir können aber davon ausgehen, dass diese ausreichend ist, um alle gesammelten Messages zu versenden.

Auch bei Netzwerken wird von der Existenz eines **kritischen Zeitpunkts** ausgegangen, d.h. alle Tasks eines Prozessors wollen ihre Nachrichten gleichzeitig schicken. Dann ist ergibt sich die *maximale* Token Holding Time eines Prozessors **p** als:

$$THT_p = \frac{\sum_{i=1}^{n(p)} M_{i,p} \left\lceil \frac{TRT}{T_{i,p}} \right\rceil}{S}$$

dabei gilt:

- **n(p)** ist die Anzahl, der auf **p** allokierten Tasks
- $\mathbf{M_{i,p}}$ ist die Gesamtgröße der Nachrichten, die Task **i** (auf Prozessor **p**) versendet
- $\mathbf{T_{i,p}}$ ist die Periodenlänge von Task **i** auf Prozessor **p**
- **S** ist die Busgeschwindigkeit
- **TRT** (**Token Rotation Time**) ist die maximale Zeitdauer zwischen zwei aufeinander folgenden Token-Ankünften

5. Schedulability Analysis in Distributed Systems

Die Token Rotation Time (**TRT**) ergibt sich wiederum i.W. aus der Summe der THT-Zeiten aller Prozessoren. Somit existiert in der o.g. Formel eine zyklische Abhängigkeit, die durch eine Iteration aufzulösen wäre.

Alternativ kann schnell eine Lösung gefunden werden, dadurch dass immer in einem lauffähigen System immer gelten muss **$TRT < T_{i,p}$**, d.h.

$$\left\lceil \frac{TRT}{T_{i,p}} \right\rceil = 1$$

Daher vereinfacht sich die Formel von eben zu:

$$THT_p = \sum_{i=1}^{n(p)} \frac{M_{i,p}}{S}$$

6. Allocation

Dieser Abschnitt befasst sich mit der **Zuteilung von Tasks zu bestimmten Prozessoren**. Wie eingangs erwähnt ist das Problem NP-hart, sofern dabei eine oder mehrere Nebenbedingungen optimiert werden. Es muss also ein **kombinatorisches Optimierungsproblem**, das wie folgt definiert ist, gelöst werden:

Gegeben:

- endliche Menge S („*Lösungen*“)
- Kostenfunktion $f : S \rightarrow R$

Gesucht:

- globales Minimum („optimale Lösung“)

$$s^* \in S^* := \{ s \in S : f(s) \leq f(t) \ \forall \ t \in S \}$$

Dabei gilt i.A., dass $|S|$ sehr groß ist: z.B. $|S| = n!$.

Es können also nur Heuristiken zur Lösung heran gezogen werden.

6. Allocation

6.1 Simulated Annealing

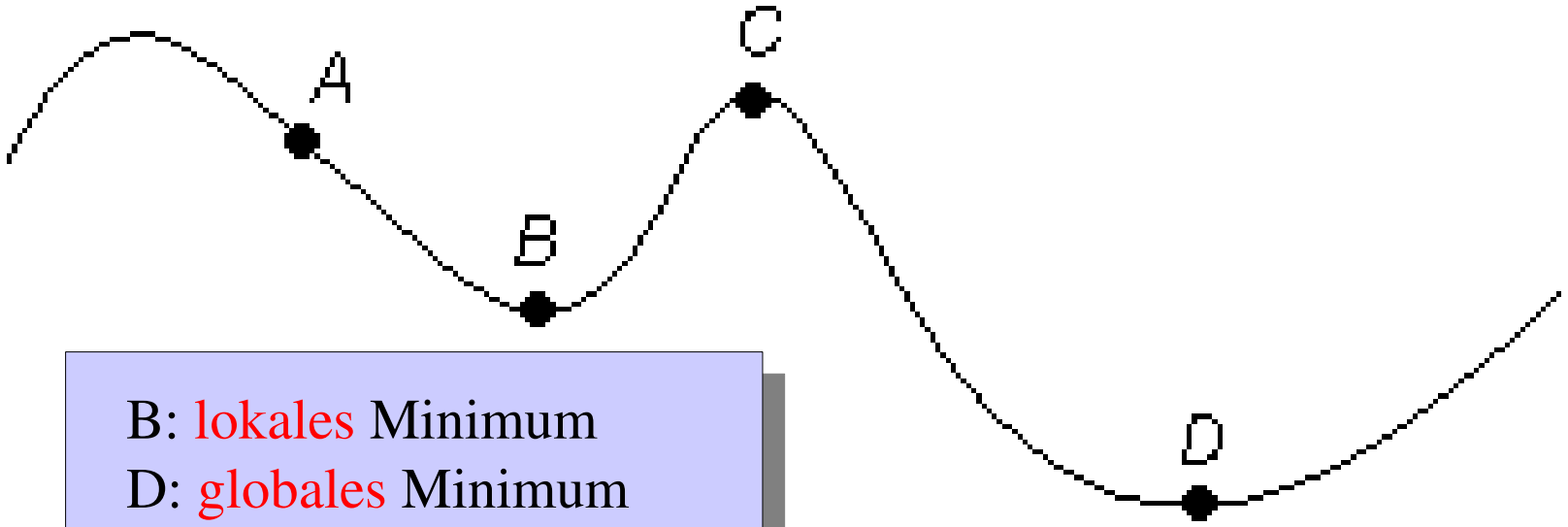
Simulated Annealing ist ein Algorithmus zur näherungsweisen Lösung von kombinatorischen Optimierungsproblemen aller Art und wird hier zur Allocation eingesetzt [6].
Der Algorithmus arbeitet nach dem Vorbild eines Abkühlungsprozesses *von der Schmelze zum Festkörper*:

- Ein System befindet sich zur Zeit **T** im Energie-Zustand $\mathbf{E_T}$
- Für **T+1** wird $\mathbf{E_{T+1}}$ vorgeschlagen
- Ist $\mathbf{E_{T+1}} < \mathbf{E_T}$ wird in $\mathbf{E_{T+1}}$ übergegangen
- Sonst wird mit „temperaturabhängiger“ Wahrscheinlichkeit $\mathbf{p(E_T, E_{T+1}, C)}$ in $\mathbf{E_{t+1}}$ übergegangen (d.h. ein lokales Minimum wird ggf. überwunden)
- Die Temperatur **C** und **p** sinken im Laufe der Zeit

Das zu lösende Problem wird dabei modelliert, indem die **Kostenfunktion** durch den Energie-Zustand ausgedrückt wird.

6. Allocation

Wichtig ist, dass Simulated Annealing in der Lage ist, lokale Minima zu überwinden: (Darstellung der Kostenfunktion)



B: lokales Minimum
D: globales Minimum

6. Allocation

Pseudo Code:

choose random starting point P_0
choose starting temperature C_0
repeat
    repeat
        E_P := Energy at point P_n
        choose T, a neighbour of P_n
        E_T := Energy at point T
        if $E_T < E_P$ then
            $P_{n+1} = T$
        else
            $x := \frac{E_P - E_T}{C_n}$
            if $e^x \geq$ random(0,1) then
                $P_{n+1} := T$
            else
                $P_{n+1} := P_n$
            fi
        fi
    until thermal equilibrium
    $C_{n+1} = f(C_n)$
until some stopping criterion

Initialisierung mit zufälliger Lösung

Verändere die momentane Lösung etwas

Versuche eine Zeit lang,
eine bessere Lösung zu finden

Schlechtere Lösung akzeptieren
bzw. zurück zur letzen

Senken der Temperatur (Zähler)

6. Allocation

Simulated Annealing kann auf das Allocation Problem wie folgt angewendet werden:

- Zuweisungen:
  - Zu Beginn werden die Task zufällig auf verschiedene Prozessoren verteilt. Dabei kann berücksichtigt werden, wenn eine Task auf einem bestimmten Prozessor liegen muss.
  - Eine Zuweisung (Lösung) wird verändert, indem eine Task zu einem anderen Prozessor verschoben wird.
- In die Energie / Kostenfunktion können mehrere Dinge gleichzeitig einfließen, **hier**:
  - Deadline-Überschreitungen (E_{sched})
  - Überschreitung der Speicherkapazität (E_{mem})
  - verschiedene Prozessoren für Replikas ($E_{replica}$)
  - die Bus-Auslastung, d.h. das eigentliche Optimierungsziel: (E_{bus})

6. Allocation

- Insgesamt erhält man folgende Kostenfunktion, wobei die o.g. Teil unterschiedlich stark gewichtet werden:

$$E = k_1 E_{sched} + k_2 E_{mem} + k_3 E_{replica} + k_4 E_{bus}$$

  - Dabei stellen E_{sched}, E_{mem}, und $E_{replica}$ **Bestrafungen** für verletzte Nebenbedinungen dar. Diese Werte müssen bei einer zulässigen Lösung jeweils **null** sein, im Einzelnen:

$$E_{sched} = \sum_{p=1}^{P} \sum_{i=1}^{n(p)} \max[0, C_{i,p} + I_{i,p} - D_{i,p}]$$

$I_{i,p}$ Interference for ith task on processor p

$$E_{mem} = \sum_{p=1}^{P} \max[0, mu(p) - mc(p)]$$

$mu(p)$ The memory used on processor p

$mc(p)$ The memory capacity of processor p

$$E_{replica} = \sum_{p=1}^{P} \sum_{i=1}^{n(p)} \sum_{j=1, j \neq i}^{n(p)} replica(i,j)$$

$replica(i,j)$ returns 1 if i is a replica of j, and zero otherwise.

6. Allocation

6.2 Beispiel für die Anwendung von Simulated Annealing

Die folgende Menge von Tasks soll auf 8 Prozessoren verteilt werden [6]:

Task	Period	WCET	Memory	Messages	Location
0	60	4	3000	50→1,150→2	0
1	60	4	1500	60→3,70→4,30→5	
2	60	2	1200	20→3	
3	60	2	1700		1
4	60	2	3000	60→6	
5	60	4	3000	80→6	
6	60	6	1100		2
7	35	2	500	40→8	1
8	35	2	700		1
9	35	8	900	90→11	0
10	35	14	2200	250→11	
11	35	4	1000		1
12	14	2	1000	150→13,150→14	2
13	14	2	1500	50→15	
14	14	2	1600	50→15	
15	14	2	1300		3
16	14	2	1100	50→17	3
17	14	2	1000		2
18	35	1	1000	50→19	1
19	35	1	1600		1
20	14	1	1900	40→21	
21	14	2	2000		3
22	14	1	1000	40→23	
23	14	1	2000	40→24	
24	14	1	1000	20→25	
25	14	1	2000	20→26	
26	14	2	7000	20→27,20→28	
27	14	1	1100	50→29	
28	14	1	900	30→29	
29	14	1	500		6
30	14	1	600	50→31	7
31	14	2	800	70→32	
32	14	2	1300		7
33	20	3	1000	50→35	2,3
34	20	2	1000	50→35	0,1
35	20	2	1000	60→36,60→37	
36	20	2	1000		6,7
37	20	2	1000		
38	20	3	1000	50→40	2,3
39	20	2	1000	50→40	0,1
40	20	2	1000	60→41,60→42	
41	20	2	1000		6,7
42	20	2	1000		6,7

6. Allocation

Initial wird eine zufällige Zuweisung vorgenommen („ * “ kennzeichnet dabei eine Deadline-Überschreitung) → die Zuteilung ist also nicht lauffähig:

```
  0    1    2    3    4    5    6    7

14* 13* 12* 16*      35* 22* 30*
20* 39* 26* 24*      40* 29  10*
25*  7  27* 31*      37      32*
28*  8  33* 15        4      36*
34* 11  38* 21                41*
 9* 18  17   1                42*
23* 19   6   5                 2*
 0*  3

Processor 0: processing utilisation 82.4%, memory utilisation 133.0%
Processor 1: processing utilisation 56.2%, memory utilisation 90.0%
Processor 2: processing utilisation 90.0%, memory utilisation 132.0%
Processor 3: processing utilisation 77.6%, memory utilisation 89.2%
Processor 4: processing utilisation 0.0%, memory utilisation 0.0%
Processor 5: processing utilisation 33.3%, memory utilisation 47.1%
Processor 6: processing utilisation 14.3%, memory utilisation 12.5%
Processor 7: processing utilisation 94.8%, memory utilisation 83.0%
TRT=23.4ms, bus speed=90 bytes/ms, bus utilisation=96.2%
```

6. Allocation

Nach einigen Zwischenlösungen (z.T. auch mit gestiegener Energie) erhält man:

```
  0   1   2   3   4   5   6   7

 35  39  13  16  25      26  30
 34   7  14  38  22      27  31
 37   8  33  15  23      28  32
  9  10  12  20  24      29  40
  1  11  17  21          36  41
  2  18   6   5              42
  4  19
  0   3
```

```
Processor 0: processing utilisation 72.9%, memory utilisation 99.0%
Processor 1: processing utilisation 81.9%, memory utilisation 97.0%
Processor 2: processing utilisation 82.1%, memory utilisation 72.0%
Processor 3: processing utilisation 71.7%, memory utilisation 85.8%
Processor 4: processing utilisation 28.6%, memory utilisation 85.7%
Processor 5: processing utilisation 0.0%, memory utilisation 0.0%
Processor 6: processing utilisation 45.7%, memory utilisation 87.5%
Processor 7: processing utilisation 65.7%, memory utilisation 57.0%
TRT=8.7ms, bus speed=90 bytes/ms, bus utilisation=29.4%
```

Dabei ist ersichtlich, dass der Algorithmus eine Lösung gefunden hat, bei der viele in Verbindung stehende Task auf demselben Prozessor allokiert worden sind, womit der Kommunikations-Overhead verringert wird.

Bemerkungen:

- Wie gezeigt ist Simulated Annealing sehr flexibel was das Einbeziehen der hier genannten und weiterer Nebenbedingungen angeht.
- Allerdings kann die Kostenfunktion dabei ziemlich komplex werden, so dass es dann um so schwieriger wird eine ***gültige*** Zuweisung zu erlangen, so dass man damit rechnen muss das Verfahren erfolglos abbricht.

6.3 Slack Method

Das Problem bei Simulated Annealing und verwandten Verfahren ist, dass der Algorithmus z.T. mehr oder weniger **ziellos** nach ***gültigen*** Lösungen Ausschau hält. Hat er erst einmal eine gültige Zuweisung gefunden, läuft das „Nachoptimieren“ dann recht problemlos.

Ein konstruktives d.h. zielgerichtetes Verfahren ist dagegen die **Slack Method** [8], die versucht, eine *gültige* Lösung schrittweise zusammen zu stellen. Dies geschieht ohne Rücksichtnahme auf das eigentliche Optimierungskriterium (die Bus-Auslastung im letzten Beispiel). Im Fokus liegt dabei, alle Deadlines garantieren zu können.

Eine von der Slack Method gefundene *gültige* Zuweisung kann dann anschließend mit Hilfe von Simulated Annealing hinsichtlich der Kostenfunktion nachoptimiert werden.

6. Allocation

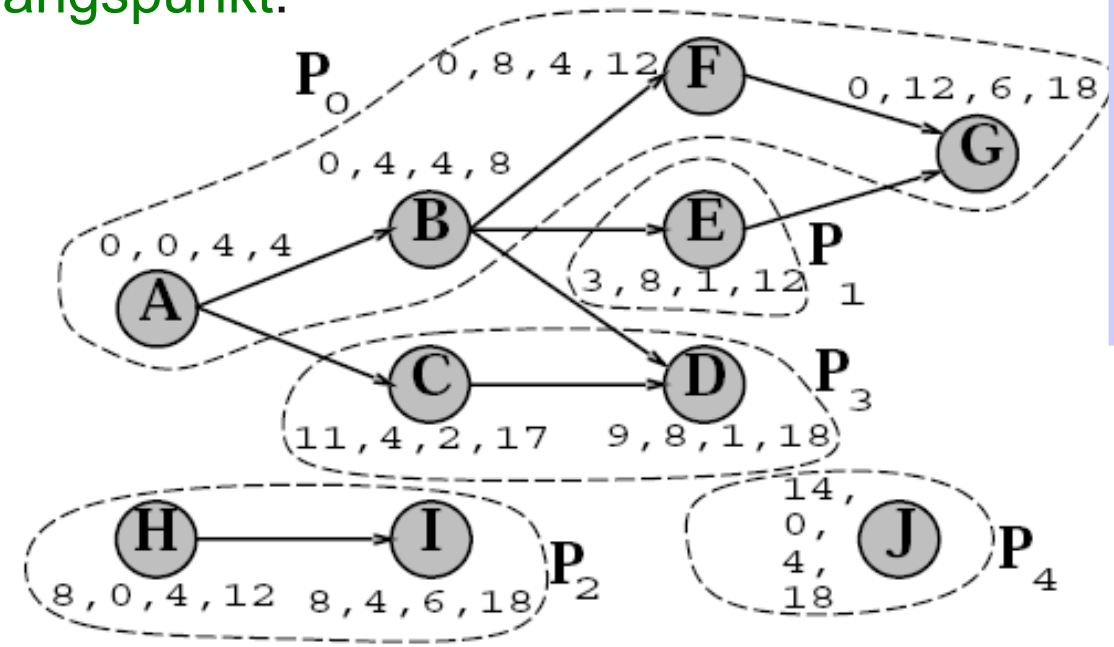
Die **prinzipielle Arbeitsweise** der Slack Method ist wie folgt:

- Der Algorithmus wird jeweils getrennt auf Tasks mit unterschiedlichen Periodenlängen angewendet (d.h. alle je Durchlauf betrachteten Tasks haben denselben Takt).
- Das Vermischen von Tasks mit unterschiedliche Periodenlängen erfolgt also erst während der Nachoptimierung. Ggf. muss daher zunächst von mehr Prozessoren ausgegangen werden, als eigentlich vorhanden sind.
- Der Trick des Verfahrens liegt darin, den gegebenen Präzedenz-Graphen schrittweise zu verkleinern (**Graph-Reduktion**), indem kommunizierende Task logisch zusammengefasst werden, mit dem Ziel sie auf demselben Prozessor zu allokieren (vgl. Erweiterung der RTA hinsichtlich Präzedenzen in Kap. 3).
- Somit kommt es ebenfalls zum Einsatz von **DMS** für das lokale Prozessor-Scheduling. Allerdings gilt hier nicht die Einschränkung, dass eine Task erst in der darauf folgenden Periode ihre Daten empfangen kann.
- Da DMS angewendet werden soll, werden die Deadlines (und andere Werte) entsprechend der vorhandenen Präzedenzen errechnet. Die Details der Berechnung übersteigen allerdings den Rahmen der VL.

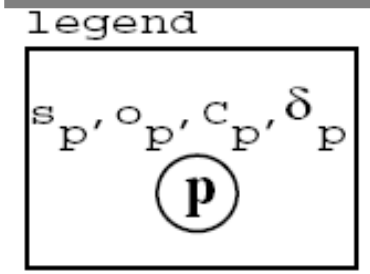
6. Allocation

Die **Graph-Reduktion** funktioniert wie am **Beispiel** illustriert:

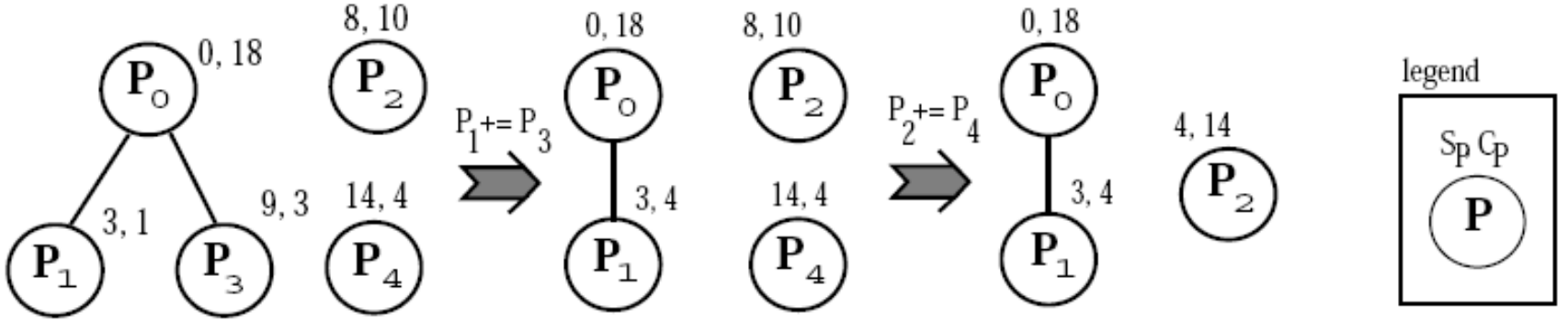
Ausgangspunkt:



Zuerst werden die Knoten, die einen **kritischen Pfad** ausmachen, vereinigt. Danach gibt der **Slack-Wert S** jeweils darüber Aufschluss, ob 2 Tasks zusammen gefasst werden können.



Weitere Reduktion:



Bemerkungen

- Die Graph-Reduktion trägt entscheidend dazu bei, die **Komplexität** des Allocation-Problems zu reduzieren (im Beispiel von eben von **n=10** auf **n=3**).
- Im Allgemeinen sind daher keine größeren Laufzeit-Probleme zu erwarten, so dass die Knoten des reduzierten Graphen mit Hilfe eines **optimalen Backtracking-Algorithmus** den Prozessoren zugeteilt werden können.
- Bei den in [8] vorgestellten Experimenten konnte so ein Beispiel mit 1710 Knoten problemlos verarbeitet werden.
- Wie oben erwähnt kann das Resultat der Slack Method dann weiter verfeinert werden, indem z.B. Simulated Annealing darauf angewendet wird. Dies gilt nicht nur für das eigentliche Optimierungskriterium sondern auch für ggf. vorhanden Nebenbedingungen.