Algorithmus von Prim

Invariante:
Alle Knoten v_i außerhalb eines Baumes T_i kennen den Knoten in T_i mit minimalem Abstand zu ihnen.
$c(v_i) :=$ nächster Nachbar von v_i in T_i

Initialisierung: Ein Anfangsknoten v_0 wird festgelegt. T_0 besteht nur aus v_0.
$c(v_i) := v_0$ für alle $v_i \neq v_0$

Pseudo-Code:

for $i := 1$ to $n - 1$ do

i) Suche unter den Knoten außerhalb von T_i einen Knoten $\tilde{v}$ mit $w(\tilde{v}, c(\tilde{v}))$ minimal und füge $\tilde{v}$ mit der Kante $(\tilde{v}, c(\tilde{v}))$ zu T_i hinzu

ii) Die Knoten v_j, die außerhalb von T_i bleiben, berechnen $c(v_j)$ neu
$c(v_j) :=$ if $w(v_j, \tilde{v}) < w(v_j, c(v_j))$
then $\tilde{v}$ else $c(v_j)$

Analyse

sequentielle Laufzeit:

$$T(n) = 1 + O(n) + (n-1) * O(n) = O(n^2)$$

Implementierung auf CREW-PRAM:

n Prozessoren mit
1-1-Zuordnung Knoten $\leftrightarrow$ Prozessoren

↷ Initialisierung: $O(1)$
Schleife (i) Minimumbildung: $O(\log n)$
Schleife (ii): $O(1)$

↷ parallele Laufzeit:

$O(n \log n)$ mit $O(n)$ Prozessoren

↷ parallele Kosten: $O(n^2 \log n)$

Durch Re-Scheduling kann Kostenoptimalität erreicht werden.

Algorithmus von Sollin

Arbeitsweise ähnlich zu Hirschberg-Algorithmus

Initialisierung: Wald mit n isolierten Knoten, die als Bäume betrachtet werden

Iteration:

für jeden Baum:
bestimme die Kante mit dem kleinsten Gewicht, die diesen Baum mit einem anderen verbindet

Alle diese minimalen Kanten werden hinzugefügt. Dabei werden eventuell entstehende Zykel beliebig durchbrochen.

Die Anzahl der Bäume wird pro Iteration mindestens halbiert.
$\curvearrowright$ maximal $\lceil \log n \rceil$ Iterationen sind erforderlich

Pro Iteration werden maximal $O(n^2)$ Vergleiche zur Bestimmung der minimalen Kanten benötigt.

$\curvearrowright$ sequentielle Laufzeit: $O(n^2 \log n)$

Sequentieller Pseudo-Code

Parameter: n = Anzahl der Knoten

Globale Variablen:

closest[]	Abstand zu nächstem Baum
edge[]	Kante zu nächstem Baum
i	
T	MST (als Kantenmenge)
v,w	Endpunkte der aktuellen Kante
weight[]	Kantengewichte
Baum[]	Wald als Knotenmengen

Hilfsfunktionen:

FIND(v) liefert zu einem Knoten, den Baum, in dem v enthalten ist.

UNION(v, w) vereinigt die Bäume, in denen zwei Knoten v und w enthalten sind.

Pseudo-Code:

```
begin
    for i := 1 to n do Baum[i] := {v_i} od
    T:= ∅
    while |T| < n − 1 do
        für jeden Baum i setze closest[i] := ∞
        für jede Kante (v, w) do
            if FIND(v) ≠ FIND(w) then
              if weight[{v, w}] < closest[FIND(v)]
              then closest[FIND(v)] := weight[{v, w}]
                    edge[FIND(v)] = {v, w}
              fi
            fi
        od
        für jeden Baum i do
            {v, w} := edge[i]
            if FIND(v) ≠ FIND(w) then (∗)
              T := T ∪ {v, w}          (∗)
              UNION(v, w)              (∗)
            fi
        od
    od
end
```

Parallelisierung

Die äußere While-Schleife kann wegen Abhängigkeiten nicht parallelisiert werden.

Die erste innere for-Schleife kann parallel ausgeführt werden. Bei p Prozessoren bearbeitet jeder Prozessor $\frac{1}{p}$ Bäume.

Zweite innere Schleife: Jeder Prozessor kann für einen Anteil an Knoten jeweils die von diesen Knoten ausgehenden Kanten untersuchen.

Dritte innere Schleife: Unkontrollierte Parallelverarbeitung kann zu Fehlern führen. Hier ist eine Synchronisation auf dem durch $(*)$ markierten kritischen Bereich erforderlich.