Übungsblatt 2

186.813 VU Algorithmen und Datenstrukturen 2 VU 3.0

25. September 2013

Aufgabe 1 *Für eine Katastrophenhilfe soll die Lieferung von schwerem Gerät mittels Transporthubschrauber optimiert werden. Konkreter stehen $m = 8$ Hubschrauber für den Transport von n Geräten $V = \{1, ..., n\}$ mit Gewichten $w_1, ..., w_n \in \mathbb{R}^+$ zu Verfügung. Für möglichst sichere und gleich schnelle Flüge sollen alle Geräte so auf die Hubschrauber aufgeteilt werden, dass das Ladegewicht der einzelnen Flüge möglichst ausgeglichen ist.*
Eine mögliche Lösung dieses Zuteilungsproblems sei durch m Mengen $Z_1, ..., Z_m\{1, ..., n\}$ repräsentiert, wobei die Menge $Z_i, i = 1, ..., m$, alle Gegenstände beinhaltet, die mit Hubschrauber i befördert werden. Es muss jedenfalls gelten:

$$Z_i \cap Z_j = \emptyset \ \ \forall i \neq j, \quad \bigcup_{i=1,...,m} Z_i = V.$$

Das Gesamtgewicht (Ladegewicht) der einem Hubschrauber i zugewiesenen Gegenstände ist $W(Z_i) = \sum_{j \in Z_i} w_j$. Um ein über alle Hubschrauber möglichst ausgegliches Ladegewicht zu erreichen, minimieren wir das größte auftretende Ladegewicht:

$$min\ max_{i=1,...,m} W(Z_i)$$

(a) *Definieren Sie eine sinnvolle Nachbarschaftsstruktur für eine lokale Suche durch die genaue Angabe erlaubter Züge. Entwerfen Sie einen ausführlichen Pseudocode für eine entsprechende lokale Suche mit der best improvement Schrittfunktion.*

(b) *Wieviele Nachbarlösungen besitzt eine Lösung in der von Ihnen definierten Nachbarschafsstruktur?*

Lösung

(a) Nachbarschaftsstruktur definieren:
Der folgende Algorithmus ist der allgemeine Lokale-Suche Algorithmus:

Algorithmus Lokale Suche:

Eingabe: eine Optimierungsaufgabe
Ausgabe: heuristische Lösung Z
Variable(n): Nachbarlösung Z' zu aktueller Lösung Z

1: Z =Ausgangslösung;
2: **wiederhole**
3: Wähle bestes $Z' \in N(Z)$; // leite eine Nachbarlösung ab
4: **falls** Z' besser als Z **dann {**
5: $Z = Z'$;
6: **}**
7: **bis** Abbruchkriterium erfüllt
8: Ausgabe;

Meine Nachbarlösung ist wie folgt definiert:

Algorithmus Nachbarlösung(Z):

Eingabe: die aktuelle Lösung Z
Ausgabe: eine bessere heuristische Lösung Z'
1: $Z' = Z$
2: min_Z =`get_min_z(`Z`)`; // liefert das $Z_i \in Z$ wobei $Z = \{Z_1, ..., Z_m\}$ mit kleinstem Gesamtgewicht
3: max_Z =`get_max_z(`Z`)`; // liefert das $Z_i \in Z$ wobei $Z = \{Z_1, ..., Z_m\}$ mit größtem Gesamtgewicht
4: $diff_g = g(max_Z) - g(min_Z)$;
5: **für alle** Geräte g aus max_Z **{**
6: // Da die Best-Improvement-Schrittfunktion gefordert ist, muss ich über alle Geräte aus max_Z iterieren und die beste Lösung zurückgeben.
7: min_g =`get_min_gerät(`min_Z`)`; // liefert das Gerät aus min_Z mit dem kleinsten Gewicht
8: **falls** $w(min_Z) + w(g) < w(max_Z) - w(g)$ **dann {**
9: // probiere das aktuelle g auf min_Z zu geben, ohne das Gewicht von max_Z zu erreichen.
10: **falls** $diff_g > w(max_Z) - w(g) - w(min_Z) + w(g)$ **dann {**
11: $diff_w = w(max_Z) - w(g) - w(min_Z) + w(g)$;
12: $Z' = Z \setminus \{min(Z), max(Z)\} \cup min_Z + g \cup max_Z - g$; // Z' ohne min und max, jedoch mit neuem min und max.
13: **}**
14: **} sonst falls** $w(min_Z) + w(g) - w(min_g) < w(max_Z) - w(g) + w(min_g)$ **dann {**
15: // probiere Tausch vom aktuellen g mit dem leichtesten g von min_Z
16: **falls** $diff_g > w(max_Z) - w(g) + w(min_g) - w(min_Z) + w(g) - w(min_g)$ **dann {**
17: $diff_g = w(max_Z) - w(g) + w(min_g) - w(min_Z) + w(g) - w(min_g)$;
18: $Z' = Z \setminus \{min(Z), max(Z)\} \cup min_Z + g - min_g \cup max_Z - g + min_g$; // Z' ohne min und max, jedoch mit neuem min und max.
19: **}**
20: **}**
21: **}**
22: Retouniere Z'

(b) Wieviele Nachbarlösungen?
Die Anzahl der Nachbarlösungen ist definiert als die Anzahl der Geräte in der Menge mit dem größten Gewicht.

Aufgabe 2 *Betrachten Sie die Aufgabenstellung aus dem letzten Beispiel. Sie möchten nun die lokale Suche auf eine Tabu Suche erweitern und die Tabu-Liste sollte Lösungsattribute von Zügen speichern. Wie würden Sie vorgehen, bzw. was würden Sie als Tabu Attribut verwenden?*

Was sind die Vor- und Nachteile, wenn Sie Lösungsattribute statt vollständige Lösungen in der Tabu-Liste speichern?

Lösung Der folgende Algorithmus ist der allgemeine Tabu-Suche Algorithmus:

Algorithmus Tabu-Suche:

Eingabe: eine Optimierungsaufgabe
Ausgabe: beste gefundene Lösung Z_{best}
Variable(n): Aktuelle Lösung Z; Nachbarlösung Z'; Tabu-Liste TL

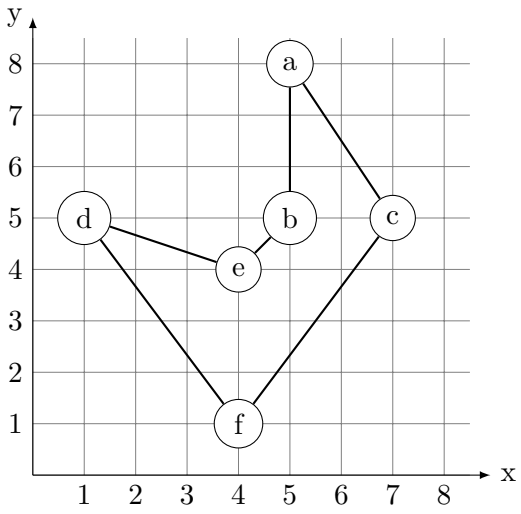
1: $Z = Z_{best}$ =Ausgangslösung;
2: $TL = \{\}$;
3: **wiederhole**
4: Wähle bestes $Z' \in N(Z, TL)$;
5: $TL = TL \cup$ get_max_z(Z); // Nimm das in N() gewählte max_Z und füge es in die TL hinzu
6: Lösche Elemente aus der TL, welche älter als t_L Iterationen sind.
7: $Z = Z'$
8: **falls** Z' besser als Z_{best} **dann {**
9: $Z_{best} = Z'$;
10: **}**
11: **bis** Abbruchkriterium erfüllt
12: Ausgabe Z_{best};

Achtung!!! die folgende Lsg ist nicht korrekt!
Meine Attribute sind einfach die in dem letzten Zug geänderten Geräte, d.h. wenn ein Gerät getauscht wurde, kann es in der Zeit t_l nicht mehr zurückgetausch werden (also solange es in der Tabuliste enthalten ist.)

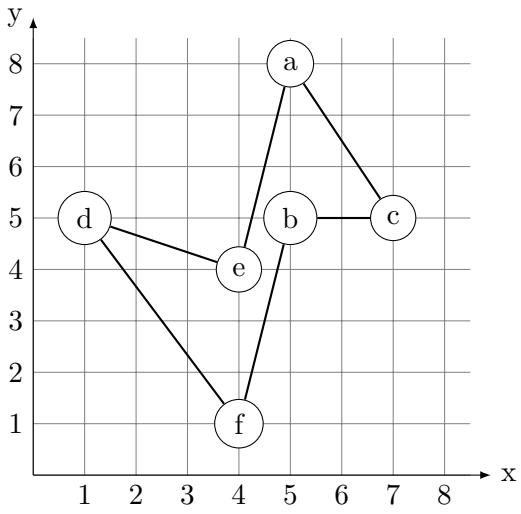
Der Vorteil, wenn man nicht alle Lösungen speichert ist, dass man nicht extrem viel Speicher benötigt, denn alle Permuationen abzuspeichern wäre Laufzeitmäßig und Speicherplatzmäßig ein Wahnsinn. Außerdem sind Algorithmen ohne TL meist einfacher zu implementieren. Aber ein guter Algorithmus mit ordentlicher TL is oftmals auch sehr langsam. Wenn man Attribute richtig wählt kann man auch sehr gute Ergebnisse erreichen (wenn nicht sogar die gleichen) als würde man sich die Ergebnisse immer mitspeichern.

Aufgabe 3 *Gegeben seien zwei Elterntouren in einem evolutionären Algorithmus für das symmetrische TSP: Für einen Knoten i bezeichnen $x(i)$ und $y(i)$ dessen Koordinaten auf dem Gitter. Das Gewicht einer Kante $e = (i, j)$ in G sei definiert als die Manhattan-Distanz*

$$M(i, j) = |x(i) - x(j)| + |y(i) - y(j)|.$$



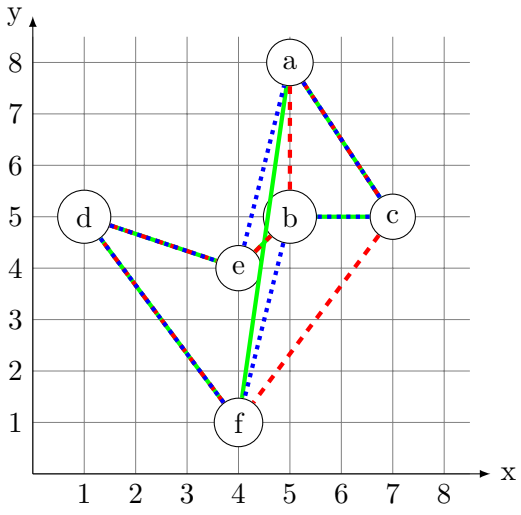
(a) T1



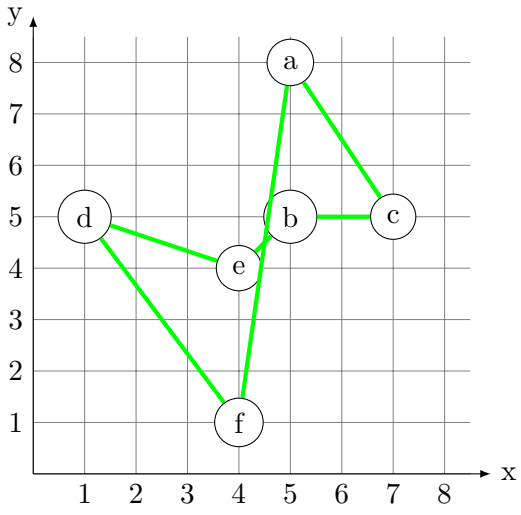
(b) T2

Führen Sie Edge-Recombination-Crossover (ERX) an den beiden gegebenen Eltern-TSP-Touren durch. Beginnen Sie bei Knoten d. Schreiben Sie in jeder Iteration auf, welche Möglichkeiten (Elternkanten) zur Wahl stehen und welche gewäht wird. Wenn mehrere Möglichkeiten zu Wahl stehen, soll jeweils die Kante mit dem geringsten Gewicht gewählt werden.

Lösung Im folgenden Bild sieht man die beiden Elterntouren übereinander:



(a) Ergebnis + Eltern



(b) Ergebnis

Algorithmus Edge-Recombination(T^1, T^2):

Eingabe: Zwei gültige Touren T^1 und T^2

Ausgabe: Neue abgeleitete Tour T

Variable(n): Aktueller Knoten v; Nachfolgeknoten w; Kandidatenmenge für Nachfolgeknoten W

1: beginne bei einem beliebigen Startknoten $v = v_0; T = \{\}$;
2: **solange** es noch unbesuchte Knoten gibt **{**
3: Sei W die Menge der noch unbesuchten Knoten, die in $T^1 \cup T^2$ adjazent zu v sind;
4: **falls** $W \neq \{\}$ **dann {**
5: wähle einen Nachfolgeknoten $w \in W$ zufällig aus;
6: **} sonst {**
7: wähle einen zufälligen noch nicht besuchten Nachfolgeknoten w;
8: **}**
9: $T = T \cup \{(v, w)\}; v = w$;
10: **}**
11: Schließe die Tour; $T = T \cup \{(v, v_0)\}$;

Es gibt die folgenden Iterationsschritte: (Startknoten ist der Knoten d, die Werte in Klammer sind die Distanzen)

i) Aktueller Knoten: d
$W = \{e(4), f(7)\} \Rightarrow$ wähle Knoten e, dennn er ist der kürzeste.
$T = \{(d, e)\}$

ii) Aktueller Knoten: e
$W = \{b(1), a(5)\} \Rightarrow$ wähle Knoten b, dennn er ist der kürzeste.
$T = \{(d, e), (e, b)\}$

iii) Aktueller Knoten: b
$W = \{c(2), a(3), f(5)\} \Rightarrow$ wähle Knoten c, dennn er ist der kürzeste.
$T = \{(d, e), (e, b), (b, c)\}$

iv) Aktueller Knoten: c
$W = \{a(5), f(7)\} \Rightarrow$ wähle Knoten a, dennn er ist der kürzeste.
$T = \{(d, e), (e, b), (b, c), (c, a)\}$

v) Aktueller Knoten: a
$W = \{\} \Rightarrow$ wähle zufällig einen Knoten (Knoten f als einziger noch über).
$T = \{(d, e), (e, b), (b, c), (c, a), (a, f)\}$

vi) Verbinden vom 1. mit dem letzten Knoten:
$T = \{(d, e), (e, b), (b, c), (c, a), (a, f), (f, d)\}$

Aufgabe 4 *Betrachten Sie den Algorithmus von Knuth-Morris-Pratt (KMP), den Text* $T = DBDBDDBDDBDBDDBDBD$ *und das Muster* $P = DBDDBDBD$.

(a) Geben Sie für das Muster P das next[] Array an.

(b) Suchen Sie mittels KMP nach allen Vorkommnissen des Musters P im Text T. Visualisieren Sie jeden Schritt und markieren Sie dabei deutlich die Mismatches.

Lösung

(a) Muster P für $next[]$ Array:

Algorithmus InitNext(P):

Eingabe: Muster $P = P[1..M]$
Ausgabe: Initialisiert das Feld $next[]$ für Muster P

```
1:  next[1] = 0; l = 0;
2:  für q = 2, ..., M {
3:     solange (l > 0) ∧ (P[l + 1] ≠ P[q]) {
4:        l = next[l];
5:     }
6:     falls P[l + 1] == P[q] dann {
7:        l = l + 1;
8:     }
9:     next[q] = l;
10: }
```

idx :	1	2	3	4	5	6	7	8
P :	D	B	D	D	B	D	B	D
$next[]$:	0	0	1	1	2	3	2	3

(b) Suchen von Vorkommnissen des Musters P im Text T:

Algorithmus Knuth-Morris-Pratt(T, P):

Eingabe: Text $T = T[1..N]$ und Muster $P = P[1..M]$
Ausgabe: Ausgabe aller Vorkommen von P in T

```
1:  InitNext(P);  // Initialisiere das Feld next[]
2:  j = 0;
3:  für i = 1, ..., N {
4:     solange (j > 0) ∧ (P[j + 1] ≠ T[i]) {
5:        j = next[j];
6:     }
7:     falls P[j + 1] == T[i] dann {
8:        j = j + 1;
9:     }
10:    falls j == M dann {
11:       Ausgabe: "P an Stelle i − j + 1 im Text gefunden";
12:       j = next[j];
13:    }
14: }
```

idx :	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
T :	D	B	D	B	D	D	B	D	D	B	D	B	D	D	B	D	B	D
1. Schritt	D	B	D	D	B	D	B	D										
2. Schritt			D	B	D	D	B	D	B	D								
3. Schritt						D	B	D	D	B	D	B	D					
4. Schritt											D	B	D	D	B	D	B	D
5. Schritt																D	B	D

1. Schritt:
   Wir können bis $i = 3$ gehen, ohne einem Mismatch. Aber ab $i = 4$ und $j = 3$ passt das Pattern nicht mehr mit dem Text zusammen. ($T[4] \neq P[3 + 1]$). Somit müssen wir das Pattern um $j - next[j] = 3 - next[3] = 2$ nach rechts verschieben.

2. Schritt:
   Wir können bis $i = 8$ gehen, ohne einem Mismatch. Aber ab $i = 9$ und $j = 6$ passt das Pattern nicht mehr mit dem Text zusammen. ($T[9] \neq P[6 + 1]$). Somit müssen wir das Pattern um $j - next[j] = 6 - next[6] = 3$ nach rechts verschieben.

3. Schritt:
   Wir können bis zum Ende des Patterns ($i = 13$) gehen, ohne einem Mismatch. $\Rightarrow$ Der Text enthält das Pattern an der Stelle $i - M + 1 = 6$. Somit müssen wir das Pattern um $j - next[j] = 8 - next[8] = 5$ nach rechts verschieben.

4. Schritt:
   Wir können bis zum Ende des Patterns und Texts ($i = 18$) gehen, ohne einem Mismatch. $\Rightarrow$ Der Text enthält das Pattern an der Stelle $i - M + 1 = 11$. Somit müssen wir das Pattern um $j - next[j] = 8 - next[8] = 5$ nach rechts verschieben.

5. Schritt:
   Es geht sich nichts mehr aus, der letzte Rest vom Text ist zu kurz für einen Match.

Aufgabe 5 *Betrachten Sie den Algorithmus von Boyer-Moore (BM), den Text* $T = BBDCBCCACBBACACDC$ *und das Muster* $P = BBACAC$.

(a) Geben Sie das $last[]$ *Array an.*

(b) Geben Sie das $suffix[]$ *Array für* P *an.*

(c) Suchen Sie mittels BM in T *nach* P. *Visualisieren Sie jeden Schritt, wo das Muster angelegt wird. Geben Sie an*

- *welche Zeichen verglichen werden,*
- *welches Zeichen gegebenenfalls den Missmatch verursacht, und*
- *mit welcher Regel (last oder suffix) verschoben wird.*

Lösung

(a) Berechnung des $last[]$-Arrays:

Algorithmus InitLast(P):

Eingabe: Muster $P = P[1..M]$

```
1: für alle c aus dem Alphabet Σ {
2:     last[c] = 0;
3: }
```

4: **für** $j = 1, ..., M$ **{**
5: $last[P[j]] = j;$
6: **}**

$idx:$	B	A	C	**sonst**
$last[]:$	2	5	6	0

(b) Berechnung des $suffix[]$-Arrays:

Algorithmus InitSuffix(P):

Eingabe: Muster $P = P[1..M]$

1: Berechne InitNext(P)$\rightarrow next[]$
2: Berechne InitNext(P^{-1})$\rightarrow \overline{next}[]$
3: **für** $j = 1, ..., M$ **{**
4: $suffix[j] = M - next[M];$
5: **}**
6: **für** $q = 1, ..., M$ **{**
7: $j = M - \overline{next}[q];$
8: **falls** $suffix[j] > q - \overline{next}[q]$ **dann {**
9: $suffix[j] = q - \overline{next}[q];$
10: **}**
11: **}**

Der Algorithmus von InitNext, siehe voriges Beispiel...

$P:$	B	B	A	C	A	C
$idx:$	1	2	3	4	5	6
$next[]:$	0	1	0	0	0	0
$\overline{P}:$	C	A	C	A	B	B
$\overline{next}[]:$	0	0	1	2	0	0
$1.suffix[]:$	6	6	6	6	6	6
$j:$	6	6	5	4	6	6
$q - \overline{next}[q]:$	1	2	2	2	5	6
$2.suffix[]:$	6	6	6	2	2	1

Hinweis: der 2. Suffix muss immer absteigend sein. Außerdem muss der Suffix an der letzten Stelle immer 1 haben.

(c) BM in T anwenden:

Algorithmus Boyer-Moore(T, P):

Eingabe: Text $T = T[1..N]$ und Muster $P = P[1..M]$
Ausgabe: Ausgabe aller Vorkommen von P in T

1: InitLast(P);
2: InitSuffix(P);
3: $i = 0;$ // Positionen vor der Anlegestelle von P
4: **solange** $i \leq N - M$ **{**
5: $j = M;$
6: **solange** $(j > 0) \wedge (P[j] == T[i + j])$ **{**
7: $j = j - 1;$

```
8:    }
9:    falls j == 0 dann {
10:     Ausgabe: "P an Stelle i + 1 im Text gefunden";
11:     i = i + suffix[1];
12:   } sonst {
13:     i = i + max(suffix[j], j − last[T[i + j]]);
14:   }
15: }
```

$idx:$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
$T:$	B	B	D	C	B	C	C	A	C	B	B	A	C	A	C	D	C
1.Schritt	B	B	A	C	A	C											
2.Schritt				B	B	A	C	A	C								
3.Schritt										B	B	A	C	A	C		
4.Schritt																B	B

1. Schritt:
   Wir bekommen schon bei der 2. Überprüfung($j = 5$) einen Mismatch.
   $suffix[j] = suffix[5] = 2$ und $j - last[B] = 5 - 2 = 3 \Rightarrow$ wir nehmen die Last-Verschiebung. (um 3 nach rechts verschieben)

2. Schritt:
   Wir bekommen bei der 4. Überprüfung($j = 3$) einen Mismatch.
   $suffix[j] = suffix[3] = 6$ und $j - last[C] = 3 - 6 = -3 \Rightarrow$ wir nehmen die Suffix-Verschiebung. (um 6 nach rechts verschieben)

3. Schritt:
   Wir können bis zum Ende des Patterns gehen, ohne einem Mismatch. $\Rightarrow$ Der Text enthält das Pattern an der Stelle 10.
   Somit müssen wir das Pattern um $suffix[1] = 6$ nach rechts verschieben.

4. Schritt:
   Es geht sich nichts mehr aus, der letzte Rest vom Text ist zu kurz für einen Match.

Aufgabe 6 *Die Approximative Zeichenkettensuche ist eine Verallgemeinerung der klassischen Suche in Texten. Das Ziel besteht darin, Textstellen zu finden, wo das Muster ungefähr mit dem Text ubereinstimmt. Zum Beispiel könnte man nach einer Stelle suchen, sodass sich der gefundene Textausschnitt nur durch eine bestimmte maximale Anzahl von Zeichen vom Muster unterscheidet.*

(a) *Entwerfen Sie einen Pseudocode für die Approximative Zeichenkettensuche* `Approx-Search`(T, P, x)*, wobei x die maximale Anzahl von Zeichen angibt, die sich die Textstelle vom Muster unterscheiden darf. Bei der Ausgabe eines (ungefähren) Matches soll zusätzlich angeben werden, wieviele Zeichen gegebenenfalls nicht gematcht haben.*
Benutzen Sie den Algorithmus aus dem Skriptum bzw. der Vorlesung für die naive Textsuche als Ausgangspunkt.

(b) Geben Sie die Laufzeit Ihres Algorithmus möglichst genau in Abhängigkeit geeigneter Kenngrößen in $\mathcal{O}$-Notation an.

Lösung

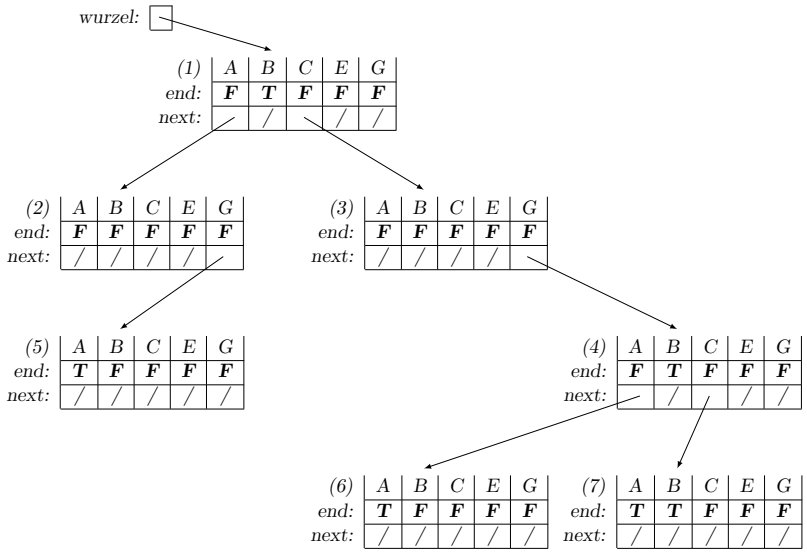
(a) **Algorithmus** Approx-Search(T, P, x):

Eingabe: Text $T = T[1..N]$, Muster $P = P[1..M]$ und Anzahl der Unterscheidungen x
Ausgabe: Ausgabe aller Vorkommen von P, oder Approx-p in T

```
1:  i = 0;  // Position vor der Anlegestelle von P
2:  solange i ≤ N − M {
3:      j = 1;
4:      remainingFails = x;
5:      solange ((j ≤ M) ∧ (P[j] == T[i + j] ∨ remainingFails > 0)) {
6:          j = j + 1;
7:          falls (P[j] ≠ T[i + j]) dann {
8:              remainingFails = remainingFails − 1;
9:          }
10:     }
11:     falls j == M + 1 dann {
12:         Ausgabe: "P an Stelle i + 1 im Text gefunden";
13:         falls (remainingFails ≠ x) dann {
14:             Ausgabe: "Es haben x − remainingFails Zeichen nicht gematcht.";
15:         }
16:     }
17:     i = i + 1;
18: }
```

(b) Laufzeit: Die Laufzeit im Best- und im Worst-Case ändert sich dadurch nicht wirklich. Nur im Average-Case, wird der Algorithmus nun etwas länger brauchen, da er erst ein paar Fehler abwarten muss, bevor er die nächste Stelle im Text bearbeiten kann. (im Algorithmus das i hochzählen kann)

Aufgabe 7 *Gegeben seien ein Alphabet $\Sigma = \{'A','B','C','E','G'\}$ und folgender Indexed Trie.*



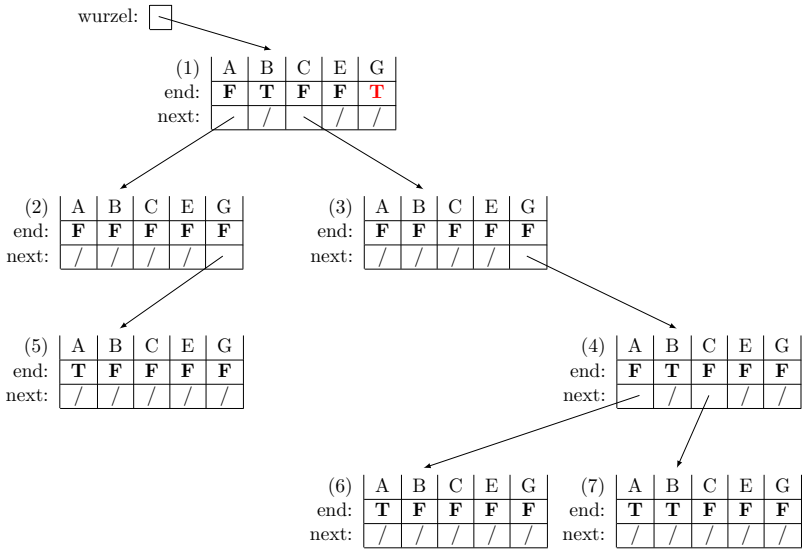
(a) Fügen Sie die Wörter `G, GA, AGB` *und* `CAA` *in den Trie ein und zeichnen Sie diesen neu.*

(b) Führen Sie Suffix Compression auf dem aus dem vorigen Punkt resultierenden Trie durch. Zeichnen Sie den Trie neu oder kennzeichnen Sie die Anderungen deutlich!

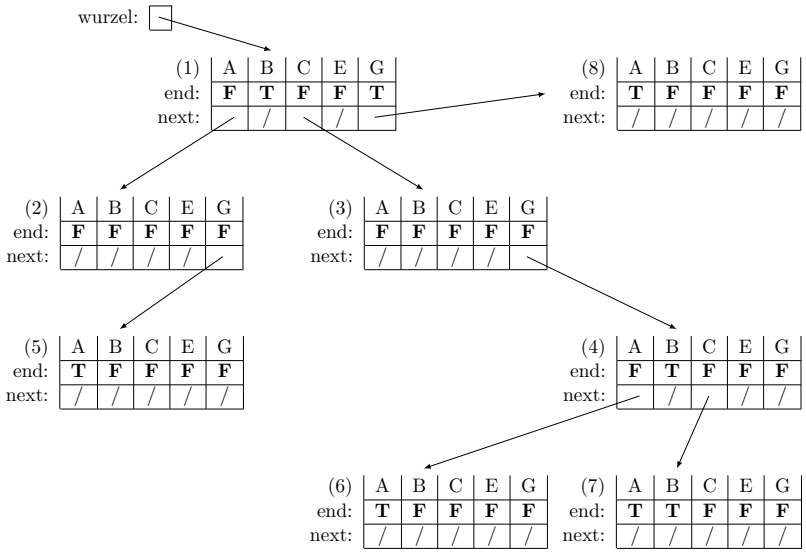
Lösung

(a) Einfügen von `G, GA, AGB` und `CAA`:

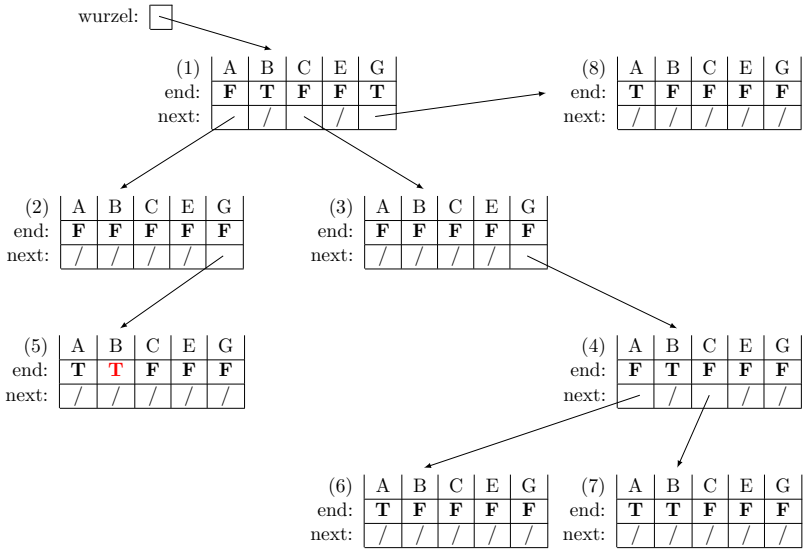
i) Einfügen von `G`: Es ändert sich nur das Flag bei G im Knoten (1) von **F** auf **T**.



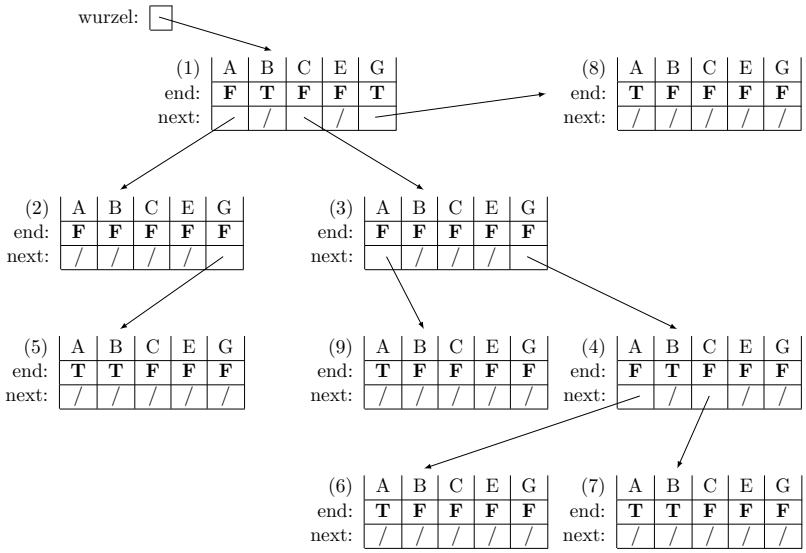
ii) Einfügen von `GA`: Es muss ein neuer Knoten (8) erstellt werden, bei diesem Knoten müssen alle next-Zeiger auf NULL gesetzt werden und alle Werte von end auf false.



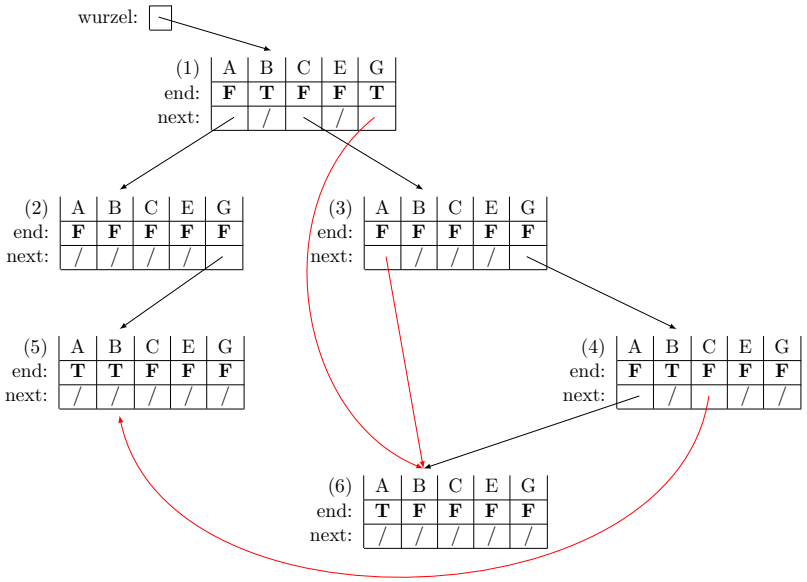
iii) Einfügen von `AGB`: Es ändert sich nur das Flag bei B im Knoten (5) von **F** auf **T**. Denn man muss von der Wurzel l-Mal nach unten gehen (l ist die Länge des Wortes)



iv) Einfügen von `CAA`: Es muss ein neuer Knoten (9) erstellt werden, bei diesem Knoten müssen alle next-Zeiger auf NULL gesetzt werden und alle Werte von end auf false.



(b) Suffix-Compression durchführen: Die Suffix-Compression löscht alle Knoten, die mehrmals genau gleich vorkommen. Daher kann es dann zu Problemen kommen, wenn man neue Worte in den Trie einfügen will. ⇒ Suffix-Compression erst durchführen, wenn sich der Trie nicht mehr verändert.



Aufgabe 8 *Betrachten Sie den endgültigen Indexed Trie aus der vorigen Aufgabe nach der Suffix Compression.*

(a) Erstellen Sie daraus einen Packed Trie. Verwenden Sie dazu die Greedy-Heuristik aus der Vorlesung bzw. aus dem Skriptum. Zeigen Sie dabei mit Hilfe einer kleinen Graphik (wie in der Vorlesung bzw. im Skriptum), auf welche Weise die Knoten gepackt werden, und zeichnen Sie den Packed Trie.

(b) Suchen Sie im Packed Trie nach den Wörtern `CGCA` *und* `CB`*. Beschreiben Sie, wie der Algorithmus vorgeht.*

Lösung

(a) Erstellen eines *Packed Trie*:

i) Sortieren nach Anzahl belegter Elemente:

(1)	4 bel. Elemente	B	B	B		B
(4)	3 bel. Elemente	B	B	B		
(3)	2 bel. Elemente	B				B
(5)	2 bel. Elemente	B	B			
(2)	1 bel. Elemente					B
(6)	1 bel. Elemente	B				

ii) Einfügen der Knoten in T (immer die 1. erlaubte Position, wobei es nicht immer nur ein Verschieben nach rechts sein muss, sondern auch nach links würde gehen. Bsp.(2))

(1)		B	B	B		B										
(4)							B	B	B							
(3)										B			B			
(5)											B	B				
(2)					B											
(6)													B			
T		B	B	B	B	B	B	B	B	B	B	B	B	B		

idx:	1	2	3	4	5	6	7	8	9	10	11	12	13
c:	A	B	C	G	G	A	B	C	A	A	B	A	G
end:	F	T	F	F	T	F	T	F	F	T	T	T	F
next:	0	N	9	10	12	12	N	10	12	N	N	N	6

Die Wurzel zeigt dabei auf 1.

(b) Suchen Sie im Packed Trie nach den Wörtern `CGCA` und `CB`.
Suchen nach `CGCA`:

i) Wir beginnen bei der Wurzel, die zeigt auf 1.

ii) Von 1 beginnend, erwarten wir bei $1+2=3$ das C. Wir finden bei 3 auch das C, $\Rightarrow$ das gehört noch zu unserem Knoten. Da unsere Zeichenfolge aber 4 Zeichen lang ist und wir erst ein Zeichen gefunden haben müssen wir weitersuchen: Das next von 3 zeigt auf 9.

iii) Mit 9 gehen wir weiter und erwarten das G somit an der Stelle $9+4=13$. Wir finden bei 13 auch das G, $\Rightarrow$ das gehört noch zu unserem Knoten. Da unsere Zeichenfolge aber 4 Zeichen lang ist und wir erst 2 Zeichen gefunden haben müssen wir weitersuchen: Das next von 13 zeigt auf 6.

iv) Mit 6 gehen wir weiter und erwarten das C somit an der Stelle $6+2=8$. Wir finden bei 8 auch das C, $\Rightarrow$ das gehört noch zu unserem Knoten. Da unsere Zeichenfolge aber

4 Zeichen lang ist und wir erst 3 Zeichen gefunden haben müssen wir weitersuchen: Das next von 8 zeigt auf 10.

v) Mit 10 gehen wir weiter und erwarten das A somit an der Stelle $10+0=10$. Wir finden bei 10 auch das A, $\Rightarrow$ das gehört noch zu unserem Knoten. Da unsere Zeichenfolge 4 Zeichen lang ist und wir auch 4 Zeichen gefunden haben müssen wir nicht mehr weitersuchen, sondern nur noch schauen, ob unter 10 ein True steht. Das ist der Fall. Somit sind wir fertig. $\Rightarrow$ Das Wort `CGCA` ist enthalten.

Suchen nach `CB`:

i) Wir beginnen bei der Wurzel, die zeigt auf 1.

ii) Von 1 beginnend, erwarten wir bei $1+2=3$ das C. Wir finden bei 3 auch das C, $\Rightarrow$ das gehört noch zu unserem Knoten. Da unsere Zeichenfolge aber 2 Zeichen lang ist und wir erst ein Zeichen gefunden haben müssen wir weitersuchen: Das next von 3 zeigt auf 9.

iii) Mit 9 gehen wir weiter und erwarten das B somit an der Stelle $9+1=10$. Wir finden bei 10 aber nicht das B, sondern nur ein A. $\Rightarrow$ der ursprüngliche Knoten war bei B nicht belegt und das Wort `CB` ist nicht enthalten.

Aufgabe 9 *Gegeben ist die folgende Punktmenge in 3D:*

$P_0 = (40, 23, 40), P_1 = (32, 41, 41), P_2 = (99, 40, 11), P_3 = (11, 11, 49), P_4 = (27, 76, 23)$
$P_5 = (76, 32, 68), P_6 = (49, 49, 27), P_7 = (23, 68, 32), P_8 = (41, 27, 99), P_9 = (68, 99, 76)$

Zeichnen Sie einen balancierten Baum für die 3-dimensionale Bereichssuche, dessen Knoten den angegebenen Punkten entsprechen. Fangen Sie mit der x-Koordinate an, verwenden Sie dann y und schließlich z.
Der Median einer sortierten Folge $\langle a_l, ..., a_r \rangle$ sei a_m mit $m = \lfloor \frac{l+r}{2} \rfloor$.

Lösung

1. Schritt: Sortieren der Punkte nach X, Y und Z und Selektion des Medians (in diesem Fall P_0):

$X = P_3, P_7, P_4, P_1, \boxed{P_0}, P_8, P_6, P_9, P_5, P_2$
$Y = P_3, P_0, P_8, P_5, P_2, P_1, P_6, P_7, P_4, P_9$
$Z = P_2, P_4, P_6, P_7, P_0, P_1, P_3, P_5, P_9, P_8$

2. Schritt: Sortieren der Punkte nach X_1, Y_1, Z_1, X_2, Y_2 und Z_2 und Selektion der jeweiligen Mediane (in diesem Fall P_1 und P_2):

$X_1 = P_3, P_7, P_4, P_1$
$Y_1 = P_3, \boxed{P_1}, P_7, P_4$
$Z_1 = P_4, P_7, P_1, P_3$

$X_2 = P_8, P_6, P_9, P_5, P_2$
$Y_2 = P_8, P_5, \boxed{P_2}, P_6, P_9$
$Z_2 = P_2, P_6, P_5, P_9, P_8$

3. Schritt: Sortieren der Punkte nach X_3, Y_3, Z_3, X_4, Y_4, Z_4, X_5, Y_5, Z_5, X_6, Y_6 und Z_6 und Selektion der jeweiligen Mediane (in diesem Fall P_3,P_4,P_5 und P_6):

$X_3 = P_3$
$Y_3 = P_3$
$Z_3 = \boxed{P_3}$

$X_4 = P_7, P_4$
$Y_4 = P_7, P_4$
$Z_4 = \boxed{P_4}, P_7$

$X_5 = P_8, P_5$
$Y_5 = P_8, P_5$
$Z_5 = \boxed{P_5}, P_8$

$X_6 = P_6, P_9$
$Y_6 = P_6, P_9$
$Z_6 = \boxed{P_6}, P_9$

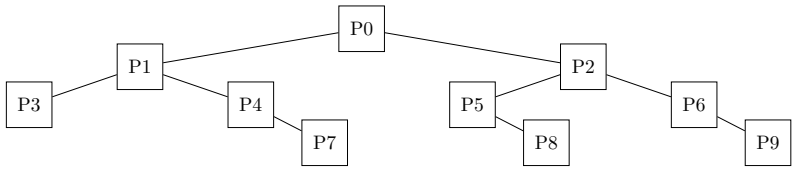
4. Schritt: Sortieren der Punkte nach X_7, Y_7, Z_7, X_8, Y_8, Z_8, X_9, Y_9 und Z_9 und Selektion der jeweiligen Mediane (in diesem Fall P_7,P_8, und P_9):

$X_7 = P_7$
$Y_7 = P_7$
$Z_7 = \boxed{P_7}$

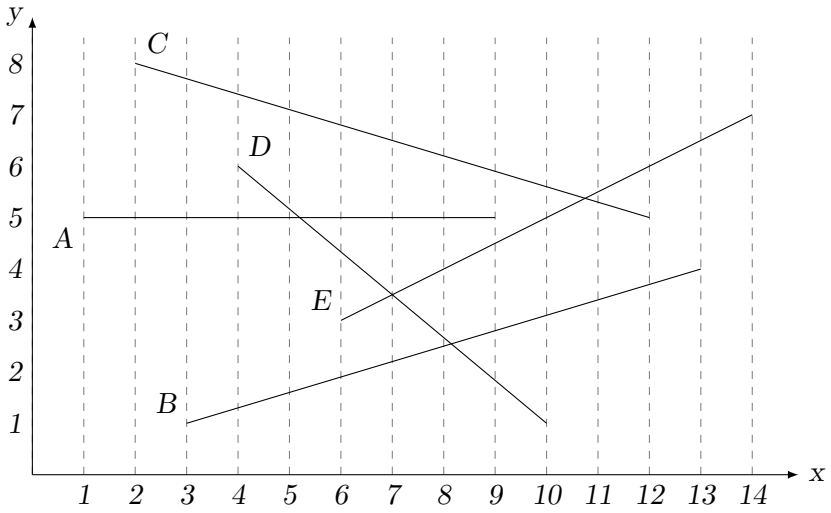
$X_8 = P_7$
$Y_8 = P_7$
$Z_8 = \boxed{P_8}$

$X_9 = P_7$
$Y_9 = P_7$
$Z_9 = \boxed{P_9}$

5. Schritt: Zeichnen des Baumes:



Aufgabe 10 *Führen Sie den aus der Vorlesung bekannten Algorithmus zum Schnitt von allgemeinen Liniensegmenten auf das folgende Beispiel aus.*

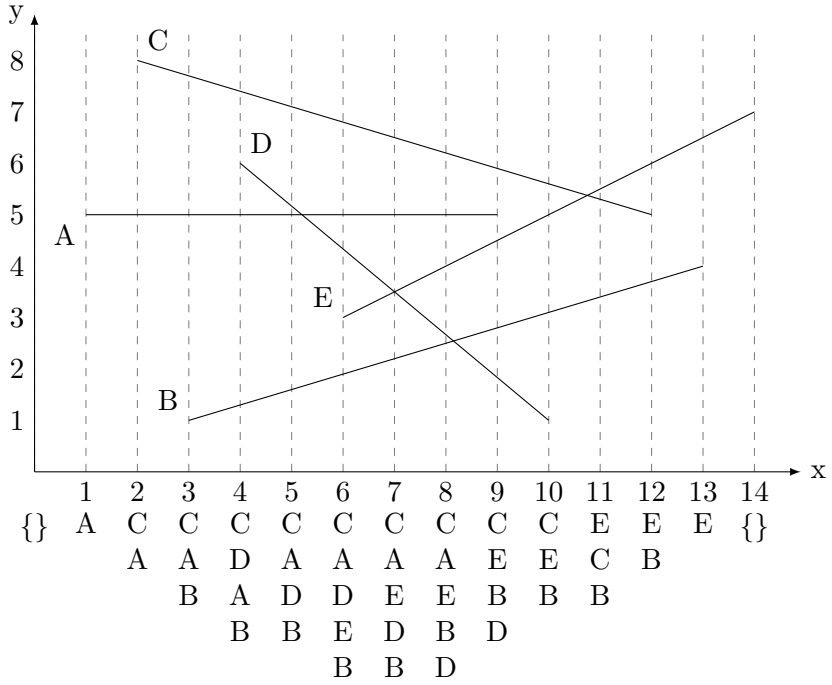


1. *Geben Sie dabei für jeden Zeitpunkt* (0...14) *im Verlauf der Bewegung der Scan-Linie den Zustand der Scan-Line-Status Struktur an.*

2. *Wann werden die Schnittpunkte $A \cap D$, $B \cap D$, $C \cap E$ und $D \cap E$ in die Ereignisstruktur eingefügt?*

Lösung

1. Zustand der Scan-Line-Status-Struktur:
   In der folgenden Grafik ist zu jeder Zeit der Zustand der Scan-Line-Struktur angegeben.



2. Zeitpunkte, wann die Schnittpunkte erreicht werden:

   i) $A \cap D$: Im Zeitpunkt 4.

   ii) $B \cap D$: Im Zeitpunkt 5.

   iii) $C \cap E$: Im Zeitpunkt 9.

   iv) $D \cap E$: Im Zeitpunkt 6.
