Grundlagen der Wissensverarbeitung-Übungsblatt 6

Übungsgruppe 2; Tom Kastek (4kastek@inf), Phil Sehlmeyer (4sehlmey@inf), Max Wutz (wutzmax@googlemail.com) · WiSe 17/18

1 Exercise 6.1: (Search and Parsing)

1. (a) Ein Parser arbeitet mit Tripeln <S, I, A>. S ist dabei der Stack, U die Input-Liste und A die Verbindungen im Graphen. Folgende Operationen können damit ausgeführt werden:

   * Left-Arc: Nimmt oberstes Element (n) aus Stack. Zusätzlich wird eine Verbindung hinzugefügt, die vom obersten Element des Inputs (n') zu diesem führt (n' $\rightarrow$ n, bzw. A $\cup$ (n', n)).
   * Right-Arc: Nimmt oberstes Element aus dem Input (n') und legt dieses auf den Stack. Zusätzlich wird eine Verbindung hinzugefügt, die vom vorher obersten Element des Stacks (n) zu diesem neuen Element führt (n $\rightarrow$ n', bzw. A $\cup$ (n, n')).
   * Reduce: Nimmt oberstes Element vom Stack (Vorausgesetzt, das Element hat einen Vater). Also <n|S, I, A> $\rightarrow$ <S, I, A>.
   * Shift: Es wird das oberste Element des Inputs genommen und auf den Stack gelegt. Also <S, n|I, A> $\rightarrow$ <n|S, I, A>

   (b) Der Parser terminiert, wenn die Input-Liste leer ist (<S, nil, A>). Der Inhalt von S und A kann beliebig sein.

   (c) Ein Abhängigkeitsgraph soll folgende Eigenschaften haben:

   * Single-head: Der Graph hat einen Kopf, von dem aus sich der gesamte Baum entwickelt.
   * Acyclic: Der Baum hat keine Kreise. Sobald im Graph zwei weitere Punkte verbunden werden, würde ein Kreis entstehen.
   * Connected: Alle Knoten im Graphen sind transitiv miteinander verbunden.
   * Projective: Ein Graph ist prejektiv, wenn jeder Knoten „graph adjacent“ zum Graphen-Kopf ist. Zwei Knoten sind „graph adjacent“, wenn jeder Knoten zwischen diesen zweien von einem der beiden dominiert wird.

   (d)

Grundlagen der Wissensverarbeitung-Übungsblatt 6

Übungsgruppe 2; Tom Kastek (4kastek@inf), Phil Sehlmeyer (4sehlmey@inf), Max Wutz (wutzmax@googlemail.com) · WiSe 17/18

2.

<nil, Der Mann isst eine Giraffe, 0>|S (1)

<Der, Mann isst eine Giraffe, 0>|LA (2)

<nil, Mann isst eine Giraffe, (Mann, Der)|S (3)

<Mann, isst eine Giraffe, (Mann, Der)|LA (4)

<nil, isst eine Giraffe, (Mann, Der), (isst, Mann)|S (5)

<isst, eine Giraffe, (Mann, der), (isst, Mann)|S (6)

<eine isst, Giraffe, (Mann, Der), (isst, Mann)|LA (7)

<isst, Giraffe, (Mann, Der), (isst, Mann), (Giraffe, eine)|RA (8)

<Giraffe isst, nil, (Mann, Der), (isst, Mann), (Giraffe, eine), (isst, Giraffe)|Terminierung, da I = nil (9)

(10)

3.
- What are search states?
  Wir gehen davon aus, dass wir die Suche mit exakt den gleichen Mitteln wie den Parser betreiben. Also einer Liste bzw. Stack, einer Liste bzw. String von übrigen Wörtern und einer Tupelliste als Knotenrelation. Also nach wie vor <S, I, A>.
- What is the start state?
  Genau wie beim Parser ist <nil, W, $\emptyset$> der Startzustand.
- What are the end states?
  Endzustand ist <S, nil, A>, also genau, wenn der gesamte String eingelesen wurde.
- What are the state transitions?
  Die Übergänge sind genau die 4 Möglichkeiten, die der Parser hat. Left-Arc, Right-Arc, Reduce und Shift. Jeder Knoten in unserem Suchraum hat also 4 Kanten.
- Can the seach space be created before the parsing starts?
  Da wir den String (W) und auch alle 4 möglichen Übergänge kennen (auch wenn einige keinen Sinn machen, hier wäre das Thema Cycle Checking und Multiple Path Pruning noch zu klären), sollte es möglich sein, den gesamten Search Space zu zeichnen, bevor das Parsen begonnen hat.
- What is the advantage of the proposed algorithm in contrast to simply trying to find a good dependency tree by enumerating all possible trees and selecting the best one from them?
- For the search strategies discussed so far: Are they a good fit for this problem and why (not)?
- How would you design a parser using the parser actions together with an appropriate search procedure?