8.4 Zij A en B talen in PSPACE.

- PSPACE algoritme voor $A \cup B$: Op input x, test of $x \in A$ of $x \in B$. Beide tests kunnen in polynomiale space, het geheel dus ook.
- PSPACE algoritme voor complement van A: Op input x, test of $x \in A$, indien niet, aanvaard, indien wel, reject. De test $x \in A$ kan in polynomiale space, dit algoritme dus ook.
- PSPACE algoritme voor A^*:
  1. Eerste oplossing: gebruik het dynamic programming algoritme voor A^* dat we hebben gezien in oefening 7.13. Elke test gebruikt polynomiale space, we doen een polynomiaal (kwadratisch om precies te zijn) aantal testen, het geheel is dus nog steeds polynomiaal (we hoeven de space zelfs niet te hergebruiken). Ook de tabel waar we de tussenresultaten in bijhouden is maar kwadratisch groot.
  2. Tweede oplossing: gegeven een woord $w = w_1 \dots w_n$, kies niet-deterministisch een deelverzameling $K \subseteq \{1, \dots, n\}$, knip w in stukken op de plaatsen in K, en test of de stukken (slechts een lineair aantal) in A zitten. Dit is een niet-deterministisch PSPACE algoritme, maar dankzij Savitch kunnen we dit automatisch omzetten in een deterministisch polynomiaal-space algoritme.

8.5 Zij A en B talen in NL.

- $A \cup B$ in NL: op input x, test (niet-deterministisch) of $x \in A$, en of $x \in B$ (ook niet-deterministisch), en aanvaard x indien minstens 1 test aanvaardde. Twee maal logaritmische space is nog steeds $O(\log n)$.
- $A \cap B$ in NL: analoog.
- A^*: we kunnen het dynamic programming algoritme hier niet gebruiken, want de tabel met alle tussenresultaten is kwadratisch groot! We kunnen ook niet in 1 keer een verzameling knipplaatsen kiezen, want die is lineair groot! Wat we wel doen is het volgende. Op input x, test eerst of $x \in A$, indien ja, aanvaard. Indien neen, kies een enkele knipplaats, test of het eerste stuk in A zit, en test dan met hetzelfde algoritme of het tweede stuk in A^* zit, waarbij we ons geheugen hergebruiken. We hoeven dan enkel 1 knipplaats te onthouden, dit is een getalletje van 1 tot n waarvoor we slechts $O(\log n)$ bits nodig hebben.

8.6 Zij A een PSPACE-harde taal. Neem B een willekeurige taal in NP. Omdat NP een deel is van PSPACE (waarom?) zit B ook in PSPACE. Omdat A PSPACE-hard is kunnen we dus B polynomiaal reduceren naar A. Omdat B willekeurig in NP was, is dus A NP-hard.

8.9 Als elke NP-harde taal ook PSPACE-hard is, dan is in het bijzonder SAT PSPACE-hard. Neem nu een willekeurige taal A in PSPACE. Omdat SAT PSPACE-hard is kunnen we A polynomiaal reduceren naar SAT, noem deze reductie f. Volgend algoritme is nu een niet-deterministisch polynomiaal algoritme voor A: Op input x, test of $f(x)$ satisfiable is.

8.11 Loop door de string van links naar rechts, terwijl je het aantal huidig open haakjes telt. Bij een gesloten haakje trek je eentje af van de teller. De teller mag nooit negatief worden en moet op het einde nul zijn. De teller wordt nooit hoger dan n en behoeft dus slechts $O(\log n)$ bits.

8.12 Loop door de string van links naar rechts. Telkens we een gesloten haakje tegenkomen doen we het volgende. We onderstellen dat het een recht haakje is; als het een rond haakje is moet je in wat volgt "recht" en "rond" verwisselen.

1. Onthou de huidige positie (van het gesloten recht haakje dus).
2. Zoek vanaf de huidige positie naar links het overeenkomstig openend recht haakje op, door gebruik te maken van een tellertje. (Als we geen overeenkomstig openend haakje vinden verwerpen we de input.)
3. Vanaf het gevonden open haakje lopen we nu terug naar rechts naar ons gesloten haakje, en ondertussen controleren we of de ronde haakjes ertussen goed gebalanceerd zijn (met een tellertje). De rechte haakjes negeren we hier gewoon even. Indien de ronde haakjes niet gebalanceerd blijken, verwerpen we de input.

Indien we dit kunnen volhouden tot het einde aanvaarden we de input.

Dit algoritme hoeft enkel de huidige positie te onthouden (een positie van 1 tot n), en gebruikt daarnaast enkel een tellertje. De benodigde space is dus duidelijk logaritmisch.

De correctheid van dit algoritme kan ingezien worden per inductie op de lengte van de input. Een lege input wordt onmiddellijk aanvaard door het algoritme; dit is de basis van de inductie. Stel nu dat we middenin de input een gesloten recht haakje tegenkomen, en we zoeken het overeenkomstig openend haakje op. Noem de string ertussen y. In y zijn de rechte haakjes al zeker gebalanceerd (per definitie van "overeenkomstig"), en ons algoritme controleert daarbij dat ook de ronde haakjes gebalanceerd zijn. Hierdoor zal ons algoritme, als het enkel op input y zou lopen, y aanvaarden. Per inductie is dus y correct gebalanceerd. Omdat ons algoritme dit tot op het einde kan volhouden is dus de input volledig correct gebalanceerd.

8.20 De knopen van een bipartite graaf kunnen verdeeld worden in twee groepen: noem deze A en B. Vanuit een knoop in A kan je in 1 stap enkel in B terechtkomen, daarna in 1 stap enkel terug in A, enzovoort. Een pad van oneven lengte komt dus altijd in de andere groep terecht, een pad van even lengte in dezelfde groep. Een cycle (een pad waarvan de laatste knoop gelijk is aan de eerste) kan dus onmogelijk bestaan uit een oneven aantal pijlen, want dan zouden de eerste en de laatste knoop in verschillende groepen zitten en dus niet gelijk kunnen zijn.

Dit geeft een eenvoudig niet-deterministisch algoritme om te checken of een graaf *niet* bipartite is: kies een startknoop, onthou die, volg dan een willekeurig pad voor een willekeurig oneven aantal stappen, en check of je terug bij de startknoop bent uitgekomen. Enkel de startknoop en telkens de huidige knoop op het pad moeten bijgehouden worden, dit vereist slechts logaritmische space. Het complement van BIPARTITE is dus in NL. Dankzij Immerman en Szelepcsenyi is dus BIPARTITE zelf ook in NL.