Linguistica Computazionale – Lezione 7

Introduzione al Parsing (sintattico)

Mercoledì 15 Marzo 2007

Cristiano Chesi, chesi@media.unisi.it

Introduzione al Parsing (sintattico)

- Indice
  - Complessità e computabilità
    - in cosa consiste
    - grammatiche e complessità
    - un esempio di grammatiche: Tree Adjoining Grammars
    - nozioni di complessità psicolinguistica
  - Il parsing
    - grammatiche e strategie di esplorazione dello spazio del problema
    - algoritmi di esplorazione
      - top-down Vs. bottom-up
      - regola del left-corner
      - programmazione dinamica e l'algoritmo di Earley

Letture, approfondimenti

- **Bibliografia essenziale**
  - Hutchins & Somers (1992) *Cap. 5*
  - Jurafsky & Martin (2000) *Cap. 10, 13*
- **Approfondimenti**
  - Barton G.E., Berwick R. & Ristad E.S. 1987. *Computational Complexity and Natural Language*. MIT Press Shank
  - Joshi & Schabes (1994) *Tree-Adjoining Grammars*.
  - Van de Koot H. *The Computational Complexity of natural language recognition*. Ms. University College London

Cos'è il parsing

Definizione del problema

- data una grammatica ***G*** ed un input ***i*** fare **parsing** significa applicare una funzione ***p(G, i)*** in grado di:
  1. **accettare / rifiutare *i***
  2. **assegnare ad *i*** una **struttura adeguata SD** (**Structural Description**)

  (ad es. il parsing sintattico riconosce un input grammaticale e assegna ad esso una struttura sintattica)
- dove si colloca il problema:

Nozioni di computazione

- Per **computazione** si intende la **specificazione della relazione** tra un **input** ed un **output**
- tale relazione consiste in una serie di **stadi intermedi** in cui l'informazione di input può venire trasformata prima di raggiungere lo "status" di output
- **obiettivo della computazione**: ridurre ogni input ricevuto ad output significativo seguendo una serie di passi consentiti dal modello computazionale adottato
- **spazio del problema**: serie di **stati raggiungibili** applicando correttamente le regole ammissibili, dati:
  - uno **stato iniziale** del sistema
  - un preciso **input**

Nozioni di complessità

(inizio …)

- La **complessità** di una computazione è direttamente proporzionale alla **quantità di risorse impiegate** per mappare un input in un output.
- Tali risorse sono solitamente definite su due dimensioni:
  - **tempo** (time complexity): passi elementari da compiere
  - **memoria** (space complexity): quantità di informazione massima richiesta ad ogni passo
- La complessità è **direttamente proporzionale** alla **dimensione del problema** (ad es. ordinare 1000 parole sarà più complesso che ordinarne 10);
- si può intuire che la complessità di una grammatica sarà direttamente proporzionale alla sua **forza generativa**.

Nozioni di complessità

(… continua …)

- La **dimensione del problema** viene espressa in funzione dell'input da processare
- quindi l'**ordine di complessità** si potrà esprimere in funzione di tale input:

  $c \cdot n^2$ (problema con ordine di complessità temporale polinomiale)

  n = dimensione dell'input

  c = costante data (dipendente dal tipo di computazione)

  Si dirà che tale problema ha ordine di complessità pari a n^2 poiché la costante c sarà irrilevante sulla crescita della funzione per n che tende all'infinito. Tale ordine di complessità si definisce: ***O(n²)***.

Nozioni di complessità

(… continua …)

- interesse nel **tasso di crescita della funzione** che esprime il mapping tra input e output in termini di dimensione dell'input
- per **problemi limitati temporalmente** e **spazialmente** (uso di risorse sicuramente finite) il calcolo della complessità è relativamente rilevante
- Al crescere dell'input a piacimento (***n* che tende all'infinito**), come nel caso di qualsiasi grammatica formale, il tasso di crescita della funzione è cruciale per determinarne la **trattabilità**
- si dice che un problema è **trattabile** se esiste una procedura che fornisce una soluzione (positiva o negativa) in un **tempo finito**
- Un problema si dice **umanamente trattabile** se l'algoritmo fornisce una risposta in tempi accettabili

Nozioni di complessità

(… continua …)

- Un problema con ordine di **complessità esponenziale** (es. $O(2^n)$) sarà difficilmente umanamente trattabile. Per avere un'idea intuitiva del tasso di crescita di alcune funzioni, ipotizzando di disporre di una macchina che gestisca **un milione di passi al secondo**, ecco il calcolo del tempo che occorrerebbe per risolvere alcune funzioni:

dimensione input → / ↓ funzione	10	20	50	100
N^2	0,0001 secondo	0,0004 sec.	0,0025 sec.	0,01 sec
N^5	0,1 sec.	3,2 sec.	5 minuti e 2 sec.	2 ore e 8 min.
2^N	0,001 sec.	1 sec.	35 anni e 7 mesi	400 trilioni di secoli
$N!$	3,6 sec.	circa 771 secoli	un numero di secoli a 48 cifre	un numero di secoli a 148 cifre
N^N	2 ore e 8 minuti	più di 3 trilioni di anni	un numero di secoli a 75 cifre	un numero di secoli a 185 cifre

Nozioni di complessità

(… fine)

- La tesi di **Turing-Church** ci dice che ogni compito computazionale che può essere realizzato da un qualsiasi dispositivo fisico può essere realizzato anche da una **macchina di Turing**
- se il dispositivo fisico riesce a completare il compito in ***F(n)*** passi, con ***n*** uguale alla dimensione dell'input, la macchina di Turing ci riuscirà in ***G(n)*** passi, con ***F*** che differisce da ***G*** di al massimo un polinomiale
- Questo implica che se una macchina di Turing richiede un tempo esponenziale (cioè t^n) per risolvere un determinato compito, allora non ci sono speranze di trovare un dispositivo fisico qualsiasi che realizzi tale compito.
- Calcolare la **complessità delle proprietà linguistiche** che si intendono catturare è quindi indispensabile per poi scegliere il meccanismo formale che implementerà la grammatica in grado di esprimerle

Problemi classici e calcolo della complessità

(inizio …)

- **3SAT problem** (variante del **problema del soddisfacimento, satisfability problem** o **SAT**)
  trovare, se esiste, una particolare **assegnazione di valori verofunzionali** alle lettere proposizionali di una formula booleana in modo che l'intera formula sia vera. La forma che la formula assume è la seguente:

  $$(a \vee \neg b \vee c) \wedge (\neg a \vee b \vee \neg c) \wedge (a \vee b \vee c) \wedge \ldots$$

  nel **peggiore dei casi**, si dovrebbero provare tutte le combinazioni di assegnazioni possibili, ovvero 2^N (dove **2** è il numero dei possibili valori dei singoli proposizionali, Vero e Falso, e **N** è il numero dei proposizionali **a**, **b**, **c**…).

  Il problema risulta quindi **temporalmente a crescita esponenziale**, ma mostra una caratteristica molto interessante: una volta proposta una soluzione, questa sarà facilissima da verificare! Un problema come **3SAT** si dice quindi **difficile da risolvere ma facile da verificare**.

Problemi classici e calcolo della complessità

(… continua …)

- **QBF problem**
  (problema della **Quantificazione di Formule Booleane**)
  trovare un'assegnazione di valori per le variabili quantificate in una formula al fine di far risultare l'intera formula vera. La struttura della formula è la seguente:

  $$Qx_1,Qx_2\ldots Qx_n\ F(x_1,x_2\ldots x_n)$$
  (con Q uguale a $\forall$ oppure $\exists$)

- Il problema è **difficile** da risolvere, **come 3SAT**, ma anche **difficile da verificare**: in effetti, ogni proposizionale di un problema di 3SAT potrebbe essere considerato come **quantificato esistenzialmente**
- Il fatto di introdurre su alcuni proposizionali una **quantificazione universale**, comporta che ogni assegnazione di valore va verificata, quindi la verifica della soluzione avrà lo stesso ordine di complessità del problema 3SAT

Problemi classici e calcolo della complessità

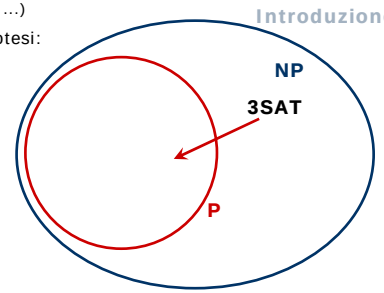
(… continua …)

- Ipotizzando che un computer **riesca a risolvere effettivamente** un problema come il 3SAT, tale computer dovrà utilizzare un **algoritmo al peggio polinomiale**.
- vista la natura del problema, tale algoritmo dovrà essere **necessariamente non-deterministico** (quindi implementabile in quella che viene chiamata una **Macchina di Turing** (**MT**) **non-deterministica**).
- Questi problemi vengono definiti di tipo **NP:**
  Non-deterministic Polynomial time
- I problemi di ordine **P** sono invece **deterministici e polinomiali** in funzione del tempo. La classe dei problemi in **P** è probabilmente inclusa in quella dei problemi in **NP**.
- Per quanto riguarda i problemi (come 3SAT) in NP non esiste prova concreta della loro **riducibilità** a problemi di ordine P.

Problemi classici e calcolo della complessità

(… continua …)

- ipotesi:



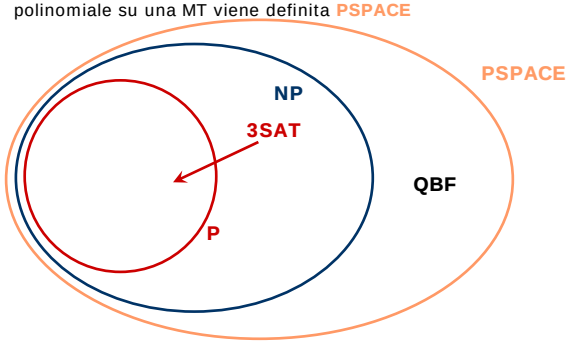
P ⊆ **NP**

- problemi come il **3SAT** si dice **NP-hard** (stesse difficoltà dei problemi della classe NP). Ipotizzando un ipotetico dispositivo "preveggente" in grado di risolvere il problema con una funzione di **ordine polinomiale** si dice che il problema è **NP-completo**.
- Un chiaro segnale di NP-completezza è il fatto che tale problema sia **difficile da risolvere ma facile da verificare** una volta proposta una soluzione.

Problemi classici e calcolo della complessità

(… fine)

- La classe di problemi che richiede uno **spazio** (o **memoria**) polinomiale su una MT viene definita **PSPACE**



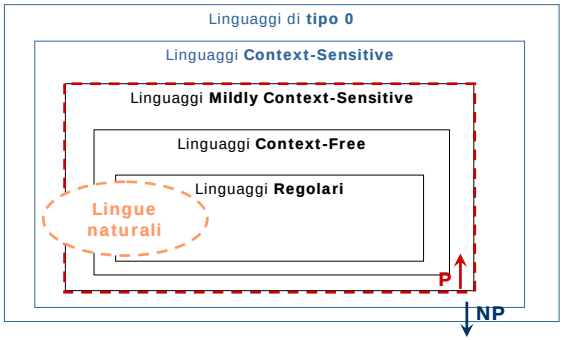
- la dimostrazione di questa relazione insiemistica risiede nel fatto che lo spazio, contrariamente al tempo può essere riusato durante la computazione

Universal Recognition Problem (URP) e riduzione

- Per scoprire la classe di complessità di un problema si ricorre alla **riduzione**: si prende cioè un problema di cui si conosce già la complessità, si trova un **mapping efficiente** che trasformi ogni istanza del problema noto in un istanza del nuovo problema e che **preservi i risultati richiesti**.
- È questo il caso dell'**Universal Recognition Problem** (**URP**) che viene così formulato:
  *Data una grammatica **G** (in qualsiasi framework grammaticale) e una stringa **x**, **x** appartiene al linguaggio generabile da **G**?*
- L'**URP** è un problema di parsing generalizzato che può essere ridotto ad un problema come **3SAT**

  intuizione di fondo: per una stringa ***x***, come un proposizionale ***a*** in una formula di **3SAT**, può esistere un'assegnazione **ambigua di valori** (ad esempio la stringa "*vecchia*" può essere sia *nome* che *aggettivo*, mentre il proposizionale *a* può essere *vero* o *falso*). Inoltre si può parlare di qualcosa come la verifica di accordo sia per la stringa ***x*** (tutte le occorrenze di x devono accordarsi in senso linguistico) che accordo tra i proposizionali in una formula 3SAT (inteso come consistenza dell'assegnazione di valori). Si deduce quindi che l'URP è almeno complesso quanto il 3SAT e che è **NP-completo**

Gerarchia di Chomsky e ordini di complessità



Tree-Adjoining Grammars (TAG, Joshi 75)

(inizio …)

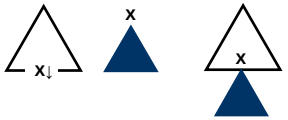
- Le **TAG** (Grammatiche ad unione di alberi (?)) generano linguaggi "**moderatamente" context-sensitive**
- Una TAG può essere definita come una quintupla **< Σ, N, I, A, S>** in cui:
  - **Σ** è un insieme finito di simboli terminali
  - **N** è un insieme finito di simboli non terminali ($N \cap \Sigma = \emptyset$)
  - **I** è un insieme finito di **alberi iniziali**, caratterizzati come segue:
    - i. i **nodi interni** sono etichettati con simboli non terminali;
    - ii. i **nodi estremi** possono essere etichettati con nodi non terminali (ed in tal caso dovranno sottostare all'operazione di **sostituzione**, marcata convenzionalmente con una freccia verso il basso ↓), o terminali.
  - **A** è un insieme di **alberi ausiliari**, caratterizzati come segue:
    - i. i **nodi interni** sono etichettati con simboli non terminali;
    - ii. i **nodi estremi** possono essere etichettati con nodi non terminali (ed in tal caso dovranno sottostare all'operazione di **sostituzione**, ad eccezione di un **nodo piede**, marcato convenzionalmente con asterisco *****, il cui simbolo è identico al **nodo radice**), o terminali.
  - **S** è il simbolo iniziale della derivazione (Sentence)

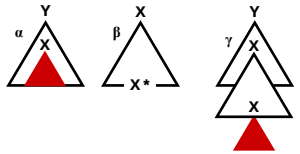
Tree-Adjoining Grammars (TAG, Joshi 75)

(… continua …)

- **I**x**A** è l'insieme degli **alberi elementari**.
- Un albero composto da più alberi binari combinati insieme secondo le regole di **sostituzione** e **aggiunzione** si chiama **albero derivato**.

sostituzione – è l'operazione che permette, all'interno di un albero α di rimpiazzare un nodo non terminale estremo marcato per tale operazione





aggiunzione – è l'operazione che permette di costruire un nuovo albero γ partendo dall'albero ausiliario β, il cui nodo radice è X, e da α (albero iniziale, ausiliario o derivato) che comprende un nodo anch'esso etichettato X, non marcato per la sostituzione

Proprietà delle TAG

(… fine)

- tra le caratteristiche interessanti e gradite delle TAG vanno sottolineate:
  - la proprietà di **gestire direttamente oggetti strutturati** e non semplici stringhe
  - la loro "**moderata**" **context-sensitività**
  - esistenza di una versione del **pumping lemma** per le TAG
  - **ordine di complessità accettabile** in fase di parsing: **$O(n^6)$**

TAG e Forma Logica (eventuale interlingua)

- **Synchronous Tree-Adjoining Grammars**

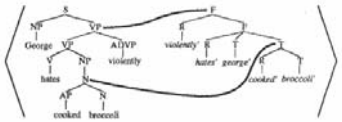
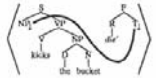


Figure 2: Derived tree pair for "George hates cooked broccoli violently."

rappresentazione logica



idiomi

Shieber & Schabes (1990) *Synchronous tree-adjoining grammars*. Proceedings of the 13th conference on Computational linguistics. pag. 253 - 258

Complessità psicolinguistica

(inizio …)

- **complessità** = **difficoltà** incontrata da un parlante nel comprendere/produrre una frase
  (es. una frase con tre **incassamenti centrali** è più complessa di una con due, come una che presenta tre **dipendenze cross-seriali** è più complessa di una con solo due. Mentre non c'è una grande differenza di complessità in frasi con due, tre o quattro **relative non incassate**: "e venne il cane che morse il gatto che scacciò il topo che mio padre alla fiera comprò")
- **ipotesi 1**: **complessità formale = complessità psicolinguistica**
- **ipotesi 2**: **memoria di processamento limitata**
  - da una parte la **capacità di tale buffer** potrebbe non essere sufficiente a contenere più di N strutture;
  - dall'altra inserire in questo deposito più di un certo numero di sintagmi (simili?) incompleti potrebbe causare **confusione**.

Complessità psicolinguistica

(… fine)

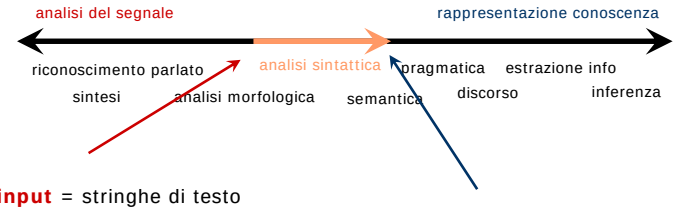
- **ipotesi 1**
  - il processamento di **strutture non context-free** causa le difficoltà maggiori (Pullum e Gazdar 1982)
- **ipotesi 2**
  - **memoria limitata** (**limited-size stack**, Yngve 1960)
    il processamento linguistico effettivo è basato su una memoria limitata (limited-size stack) in cui vengono inseriti i risultati parziali delle regole applicate finché i sintagmi non sono completati. Più frasi incomplete vengono inserite nello stack, più difficile da processare sarà il periodo.
  - **Syntactic Prediction Locality Theory** (SPLT, Gibson 1998)
    carico mnemonico totale richiesto da una struttura sintattica = somma dei carichi mnemonici richiesti dalle singole parole **necessarie per completare tale struttura**
    1. DP richiedono VP (in SVO) DP DP DP VP VP VP… è più complessa di DP VP
    2. l'introduzione di un **pronome** che si riferisce ad un'**entità già introdotta** nel discorso è meno complessa dell'introduzione di un **nuovo referente** che dovrà in qualche modo essere integrato nella struttura non ancora completata.

Parsing e grammatiche

- le grammatiche formali (***G***) sono (in genere) dispositivi dichiarativi che non specificano come un preciso input debba essere **analizzato** e come una struttura ad albero possa essere costruita.
- Fenomeni come il **non determinismo** (la possibilità di scegliere tra più alternative strutturali equivalenti dal punto di vista dell'informazione posseduta al momento della scelta) pongono seri problemi da questo punto di vista, poiché potrebbe facilmente accadere che la scelta strutturale fatta possa rivelarsi sbagliata proseguendo nel parsing. Si richiedono quindi strategie chiare per gestire efficientemente queste situazioni.

Da dove si inizia a fare parsing

- specificare i requisiti che l'input e l'output devono soddisfare (in termini "minimalisti", Chomsky 1995, si parla di **requisiti di interfaccia**)
  1. **fonologia** (Phonetic Form)
  2. **forma logica** (Logical Form)



input = stringhe di testo

output = alberi che esprimono relazioni di costituenza (quindi di dominanza, sia locale che "a distanza")

Spazio del problema e strategie di ricerca

(inizio …)

- data una **frase** ed una **grammatica** il compito del parser è dire se la frase può essere generata dalla grammatica (URP, Universal Recognition Problem) e, in caso affermativo, assegnare alla frase un adeguato **Indicatore Sintagmatico**
- Lo **spazio del problema** è l'**insieme di tutti gli alberi** e sottoalberi possibili che possono essere generati applicando adeguatamente le regole grammaticali

Spazio del problema e strategie di ricerca

(… continua …)

- frase:

  la vecchia legge la regola

- grammatica:

→ (non terminali)	→ (terminali)
S → DP VP	pro → la
VP → V DP	D → la
VP → pro V	AGG → vecchia
DP → D NP	N → vecchia
NP → (AGG) N;	N → legge
	N→ regola
	V → legge
	V→ regola

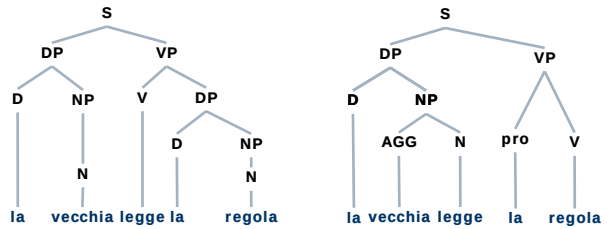
Spazio del problema e strategie di ricerca

(… fine)

- Due vincoli fondamentali:
  1. le **regole grammaticali** che predicono come da un nodo radice S ci siano solo alcune vie di scomposizione possibili per ottenere i nodi terminali;
  2. le **parole della frase**, che ricordano come la (s)composizione di S debba terminare
- nel primo caso, se si decide di **partire dal nodo radice S** per generare la struttura adatta alla frase data, si parla di strategia di ricerca **Top-Down** o **goal-driven**
- nel secondo caso, **partendo quindi dalle singole parole** e cercando di combinarle in strutture compatibili per giungere ad S, la strategia di ricerca si dice **Bottom-Up**, o **data-driven**.

Algoritmo di parsing Top-Down

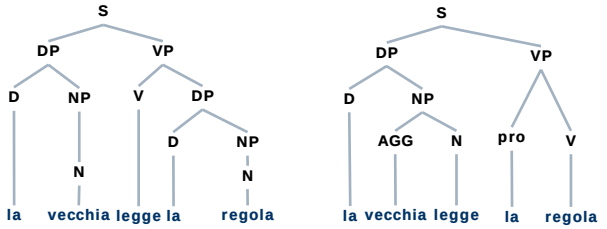
- un semplice algoritmo top-down inizia esplorando **tutte le possibilità di riscrittura** del nodo **S** che la grammatica offre (assumiamo che ogni albero generato possa poi essere esplorato parallelamente).



- "la regola regola la regola", "la legge legge la vecchia legge"… saranno scartate solo all'ultimo livello, in fase di mappatura con l'input realmente fornito.

Algoritmo di parsing Bottom-Up

- primo vero algoritmo associato storicamente all'operazione del parsing (Yngve 55) e probabilmente il più usato (ad esempio nei parsers per i linguaggi di programmazione). La logica che guida la ricerca è quella di partire dagli elementi della frase, cioè dai nodi terminali, e costruire da qui, applicando tutte le regole possibili, una struttura che termini con S:



Quale è la strategia migliore?

- La strategia **Top-Down** non perde tempo cercando di costruire **alberi non consistenti con la grammatica**, ma genererà tutte le alternative possibili prescindendo dall'input.
- La strategia **Bottom-Up**, sarà **consistente** (almeno localmente) **con l'input fornito**, ma potrà generare per molti livelli sotto-alberi che alla fine si riveleranno incombinabili per ottenere S.
- Anche se talvolta si possono ottenere risultati comparabili, nella stragrande maggioranza dei casi si deve tener conto di due fondamentali caratteristiche dello spazio del problema:
  a. si parte sempre dalla parte dell'albero in cui si dispone dell'**informazione più precisa**
  b. si risale l'albero nella direzione in cui il **fattore di ramificazione** è **minore**.

Il costo del parallelismo

- Finora abbiamo irrealisticamente assunto che le varie regole potessero essere **applicate in parallelo** (cioè che ogni ramificazione dello spazio del problema potesse essere **espansa in contemporanea**). In realtà la quantità di memoria richiesta per registrare gli stati dello spazio di un problema creato da un parser che usa una grammatica credibile sarebbe enorme.
- Di solito la soluzione "**brute-force**" avviene attraverso l'**esplorazione per ampiezza** (**breadth-first**) o per **profondità** (**deep-first**): con quest'ultimo metodo si esplora lo spazio del problema incrementalmente, espandendo fino ai nodi terminali, un nodo alla volta, solitamente **da sinistra a destra** (**Top-Down**, **depth-first**, **left-to-right parser**). Ogni volta che l'operazione fallisce, si sceglie un nuovo percorso iniziando dalla ramificazione più recente.
- Questo tipo di ricerche sono comunque **cieche**, nel senso che non esistono **euristiche** che suggeriscono, di fronte ad un'alternativa, la scelta migliore e soprattutto non si rendono conto dell'errore finché l'espansione non è stata completamente effettuata.

Algoritmo di parsing LEFT CORNER

- **idea di base**
  combinare una strategia Top-Down per la generazione di strutture filtrando le soluzioni con considerazioni di natura Bottom-Up.

- **regola dell'angolo sinistro** (**left-corner**)
  ogni categoria alla fine verrà riscritta come una serie di parole linearmente ordinate e conoscere la prima parola della serie aiuterà a prevedere la categoria o almeno ad escludere soluzioni inconsistenti con questa parola

  Formalmente si può dire che B è l'angolo sinistro della categoria A sse $A \rightarrow^* B \rightarrow \alpha$.

- Da tale strategia si ottengono i risultati migliori se preliminarmente viene compilata una tabella che direttamente associa alle categoria un loro possibile left-corner:

categoria	S	DP	VP
left-corner	D, $N_{proprio}$, V	D	aux, V

Alcuni problemi irrisolti

(inizio …)

a. **ricorsività a sinistra**
   una grammatica si dice ricorsiva a sinistra se ammette una regola del tipo $A \rightarrow^* A\alpha$ (es. DP → DP PP)

b. **ambiguità**
   a vari livelli si possono trovare molteplici strutture che soddisfano i requisiti dell'input e delle regole grammaticali:
   - attaccamento dei PP (ho visto l'uomo con il cannocchiale)
   - coordinazione (papaveri e paperi rossi)

   È stato calcolato (Church e Patil 82) che il numero di strutture possibili in corrispondenza di PP cresce esponenzialmente con il numero di PP introdotti (se con 3 PP si hanno fino a 5 NP possibili, con 6 PP si arriva a 469 NP possibili… con 8 a 4867).

Alcuni problemi irrisolti

(… fine)

b. **ambiguità** (…continua)

   Da una parte il **Top-Down parser**, se perseguisse il suo compito efficientemente, si dovrebbe fermare appena recupera una struttura possibile (nel caso di 3 PP ha una possibilità su 5 di individuare la soluzione più probabile), dall'altra sarebbe imprudente lasciarlo considerare tutte le strutture generabili.

   L'**ambiguità** (quella lessicale soprattutto: "vecchia" è Aggettivo o Nome?) causa facilmente errori. La soluzione deep-first, left-to-right potrebbe rivelarsi in tal senso estremamente inefficiente in caso di ambiguità iniziali.

c. **inefficienza nel ripetere l'analisi dei sottoalberi**
   il backtracking causato da un errore di parsing, può facilmente provocare il disfacimento di una porzione di albero che in realtà sarebbe costruita bene e che verrà ricostruita tale e quale al prossimo tentativo, come nel caso seguente:

   un volo da Roma per Milano su un 747

Programmazione Dinamica

- Per **programmazione dinamica** (**dynamic programming**) si intende un approccio che fa uso sistematico di tavole per la registrazione di soluzioni ai sottoproblemi incontrati.

- Una volta risolti tutti i **sottoproblemi** (nel caso del parsing sintattico, tutti i sottoalberi), la soluzione al problema globale consiste nel **combinare adeguatamente le singole soluzioni trovate**. Tale metodo risulta essere molto più efficiente degli approcci precedenti e risolve, in linea di principio, almeno il problema dell'inefficienza del ripetere l'analisi corretta dei sottoalberi.

Programmazione Dinamica: l'algoritmo di Earley

(inizio …)

- L'**algoritmo di Earley** (Earley 1970) è un esempio classico di programmazione dinamica che usa un approccio **top-down parallelo**.
- La complessità del problema, in linea teorica sempre NP-hard, viene ridotta a polinomiale (nel peggiore dei casi abbiamo **$O(n^3)$** eliminando le soluzioni ripetitive dei sottoproblemi dovuti al backtracking da una struttura scorretta.
- L'algoritmo è caratterizzato da un **unico esame**, **da sinistra a destra**, dell'input che permette di riempire una struttura dati (del tutto simile ad un **array**) chiamata **grafo** (**chart**) che avrà ***n*+1 entrate**, con ***n*** uguale al numero delle parole in input.
- Le entrate corrispondono alle posizioni tra le parole dell'input (comprese la posizione iniziale, prima della prima parola, e quella finale dopo l'ultima). Per ogni posizione il chart conterrà una lista esaustiva delle strutture fin lì generate ed utilizzabili per ulteriori elaborazioni che cercheranno di integrare posizioni successive. La rappresentazione compatta di questa informazione permette di sfruttare efficientemente, senza doverla ricomputare in caso di backtrack, ogni struttura ben formata associata ad un **input parziale**.

Programmazione Dinamica: l'algoritmo di Earley

(… continua …)

- In concreto quindi ogni entrata del **chart** avrà tre tipi di informazioni:
  - un **sottoalbero** corrispondente ad una **singola regola grammaticale**
  - l'**informazione del progresso fatto** al fine di completare il sottoalbero in questione (solitamente si usa un punto • nella parte destra della regola per indicare la posizione a cui si è giunti, la "**regola puntata**" si dice quindi **dotted rule**)
  - la **posizione** del sottoalbero **in relazione all'input** (definito da due numeri indicanti la posizione di inizio della regola e quella dove si trova il punto; es. DP → D • NP [0,1])

Programmazione Dinamica: l'algoritmo di Earley

(… continua …)

- L'algoritmo procede con tre operazioni fondamentali:
- **Previsione (Predictor)**
  crea **nuovi stati** nell'entrata corrente del chart, rappresentando le aspettative top-down della grammatica; verrà quindi creato un numero di stati uguale alle possibilità di espansione di ogni nodo non terminale nella grammatica.
  es. S → • DP VP [0,0] DP → • D NP [0,0]
- **Scansione (Scanner)**
  verifica se nell'input esiste, nella posizione adeguata, una parola la cui categoria combacia con quella prevista dallo stato a cui la regola si trova. Se il confronto è positivo, la scansione produce un nuovo stato in cui l'indice di posizione viene spostato dopo la parola riconosciuta. Tale stato verrà aggiunto all'entrata successiva del chart.
  es. DP → • D NP [0,0] (scansione articolo) se D → articolo: DP → D • NP [0,1] (entrata successiva del chart)
- **Completamento (Completer)**
  quando l'indicatore di posizione raggiunge l'estrema destra della regola questa procedura riconosce che un sintagma significativo è stato riconosciuto e verifica se questo avvenuto riconoscimento è utile per completare qualche altra regola rimasta in attesa di quella categoria: ad esempio se nella situazione precedente viene completata positivamente la regola NP → AGG N • [1,3], l'operazione di completamento cercherà tutti gli stati che terminavano nella posizione 1 e che aspettavano un NP per completare la regola. Ma questo era proprio il caso di DP → D • NP [0,1]; quindi il riconoscimento del NP aggiunge anche (all'entrata corrente) lo stato DP → D NP • [0,3];

Programmazione Dinamica: l'algoritmo di Earley

(… continua …)

funzione Earley-Parser(*stringa_parole*, *grammatica*) **restituisce** *grafo*

metti_in_coda(($\gamma \rightarrow \bullet S$, [0,0]), *grafo*[0])

per ogni *i* **tale che** (*i* va da 0 a lunghezza(*stringa_parole*)) **esegui**
  per ogni *stato* **tale che** (*stato* è in *grafo*[*i*]) **esegui**
    se (incompleto(*stato*) **e non** parte_della_frase
      (prossima_categoria(*stato*)))
    allora esegui **Previsione** (*stato*)

    altrimenti_se (incompleto(*stato*) **e**
      parte_della_frase(prossima_categoria(*stato*)))
    allora esegui **Scansione** (*stato*)

    altrimenti esegui **Completamento** (*stato*)
  fine
fine
restituisci *grafo*

Programmazione Dinamica: l'algoritmo di Earley

(… fine)

procedura Previsione ($A \rightarrow \alpha \bullet B\beta$, $[i,j]$)
  per ogni ($B \rightarrow \gamma$) **tale che** (Regole_della_Grammatica(*B*, *grammatica*)) **esegui**
    metti_in_coda(($B \rightarrow \bullet\gamma$, $[j,j]$), *grafo*$[j]$)
fine
procedura Scansione ($A \rightarrow \alpha \bullet B\beta$, $[i,j]$)
  se (*B* è parte_della_frase(*parola*$[j]$)) **allora esegui**
    metti_in_coda(($B \rightarrow$ *parola*$[j]$, $[j,j+1]$), *grafo*$[j+1]$)
fine
procedura Completamento ($B \rightarrow \gamma \bullet$, $[j,k]$)
  per ogni ($A \rightarrow \alpha \bullet B\beta$, $[i,j]$) **tale che** (($A \rightarrow \alpha \bullet B\beta$, $[i,j]$) è nel *grafo*$[j]$) **esegui**
    metti_in_coda(($A \rightarrow \alpha B \bullet \beta$, $[j,k]$), *grafo*$[k]$)
fine
procedura metti_in_coda (*stato*, *entrata_grafo*)
  se (*stato* non è già nell'*entrata_grafo*) **allora esegui**
    inserisci(*stato*, *entrata_grafo*)
fine

Alcune applicazioni concrete

- **Cascate di automi a stati finiti**

  il presupposto che giustifica questi semplici dispositivi è che non tutti i tipi di processamento linguistico richiedono un parsing completo.
  In particolare molte strutture risultano localmente non ambigue e quindi elaborabili indipendentemente dal resto della frase (ad es. gran parte dei **sintagmi nominali**).
  Questo principio è quello che guida molti algoritmi ad esempio per l'estrazione di informazioni (ma anche per la traduzione mirata di specifici costituenti, es. sintagmi nominali) che si avvalgono di analisi parziali (**partial parse** o **shallow parse**).

- Usare **FSA** anziché **CFG** comporta ad esempio l'esclusione dalla grammatica di regole ricorsive in senso stretto (es. DP → Agg DP PP) e quindi ad una minore potenza generativa corrispondente ad una difficoltà nel catturare fenomeni potenzialmente significativi.

Prossima lezione

(Martedì 20 Marzo, ore 16-18, Aula 456, Palazzo S.Niccolò)

- **Parsing avanzato**
  - Grammatiche ad unificazione
    - tratti e gerarchie
    - HPSG
    - fenomeni linguistici da catturare
  - Principle & Parameter Parsing
    - dalle regole ai principi
    - un esempio di P&P parser (PAPPI, Fong 1991)
  - Accenni a formalizzazioni minimaliste