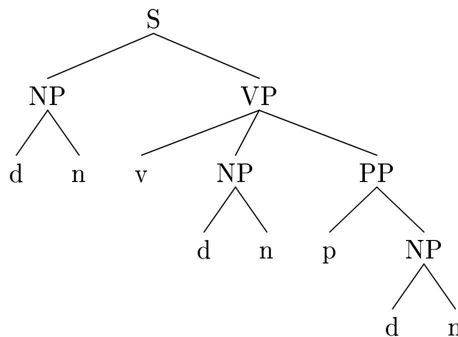
11-711: Algorithms for NLP

Midterm Sample Solutions

October 23, 2007

Problem 1: Formal Language Theory

1. [**5 points**] The string $w = dnvdnpdn \in L(G)$. Show a parse tree for w according to G.



2. [**10 points**] Convert the grammar G into a grammar G' that is in Chomsky Normal Form, such that $L(G') = L(G)$.

```
S  --> NP VP          C1 --> NP PP
NP --> D N            D  --> d
VP --> V NP           N  --> n
VP --> V C1           V  --> v
PP --> P NP           P  --> p
PP --> P C1
```

3. [**15 points**] Construct a push-down automaton M that has only one state and that accepts the language $L(G)$ by empty stack. You should specify each of the components of the machine $M = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$. Give a short explanation of why the machine you constructed accepts $L(G)$.

$Q = \{q\}$
$\Sigma = \{d, n, v, p\}$
$\Gamma = \{\mathrm{S}, \mathrm{NP}, \mathrm{VP}, \mathrm{PP}, \mathrm{C1}, \mathrm{D}, \mathrm{N}, \mathrm{V}, \mathrm{P}\}$
$q_0 = q$
$z_0 = \mathrm{S}$
$F = \emptyset$

$\delta(q, \epsilon, \mathrm{S}) = \{(q, \mathrm{NP\ VP})\}$
$\delta(q, \epsilon, \mathrm{NP}) = \{(q, \mathrm{D\ N})\}$
$\delta(q, \epsilon, \mathrm{VP}) = \{(q, \mathrm{V\ NP}), (q, \mathrm{V\ C1})\}$
$\delta(q, \epsilon, \mathrm{PP}) = \{(q, \mathrm{P\ NP}), (q, \mathrm{P\ C1})\}$
$\delta(q, \epsilon, \mathrm{C1}) = \{(q, \mathrm{NP\ PP})\}$
$\delta(q, d, \mathrm{D}) = \{(q, \epsilon)\}$
$\delta(q, n, \mathrm{N}) = \{(q, \epsilon)\}$
$\delta(q, v, \mathrm{V}) = \{(q, \epsilon)\}$
$\delta(q, p, \mathrm{P}) = \{(q, \epsilon)\}$

This machine simulates a left-most derivation of a word in $L(G)$. The machine's stack keeps track of the "tail" of each sentential form (the part that hasn't generated any terminals yet) by replacing a non-terminal on the left-hand side of a grammar rule with the string of terminals and non-terminals that is derived from it on the right-hand side of the rule. When a terminal is on the top of the stack, and an identical terminal is the next symbol in the input, the input symbol is consumed and the terminal is popped off the stack.

4. [**5 points**] Run the PDA M that you constructed above on the input $w = dnvdnpdn$ and show the sequence of IDs that the machine goes through until the input is accepted.

$(q, dnvdnpdn, \mathrm{S}) \vdash (q, dnvdnpdn, \mathrm{NP\ VP}) \vdash (q, dnvdnpdn, \mathrm{D\ N\ VP}) \vdash (q, nvdnpdn, \mathrm{N\ VP}) \vdash (q, vdnpdn, \mathrm{VP}) \vdash (q, vdnpdn, \mathrm{V\ C1}) \vdash (q, dnpdn, \mathrm{C1}) \vdash (q, dnpdn, \mathrm{NP\ PP}) \vdash (q, dnpdn, \mathrm{D\ N\ PP}) \vdash (q, npdn, \mathrm{N\ PP}) \vdash (q, pdn, \mathrm{PP}) \vdash (q, pdn, \mathrm{PNP}) \vdash (q, dn, \mathrm{NP}) \vdash (q, dn, \mathrm{D\ N}) \vdash (q, n, \mathrm{N}) \vdash (q, \epsilon, \epsilon)$

5. [**10 points**] What is the maximum depth of the PDA's stack content during the computation on w? What is the maximum depth in the general case on a given input word $x \in L(G)$? Provide a closed-form expression and briefly explain.

The maximum depth of the stack during the computation on w is **three**.

In general, the stack depth is limited to **three** for any word $x \in L(G)$. This is because M simulates the grammar G' that is in Chomsky Normal Form, every non-terminal in G' derives a pre-terminal in no more than two steps, and then the pre-terminal is popped off the stack in one step. Each right-hand side has at most length 2, so no more than two symbols are ever written to the stack at one time, and every time this happens, the stack has no more than one symbol already in it.

6. [**10 points**] Prove that if $w \in L(G)$, then $|w| = 3i + 5$ for some $i \geq 0$.

Since NP derives only $d\ n$ (and nothing else) in the grammar, the non-terminal can be replaced:

```
S --> d n VP
VP --> v d n
VP --> v d n PP
PP --> p d n
PP --> p d n PP
```

The shortest derivable terminal string is $\mathrm{S} \Rightarrow d\ n\ \mathrm{VP} \Rightarrow d\ n\ v\ d\ n$, which has length 5. All other rules (for VPs and PPs) add three additional terminal symbols, plus the possibility of recursively adding three more by deriving additional PPs. If i of these rules apply, then $3i$ terminals are generated. Thus the length of any string in $L(G)$ is $3i + 5, i \geq 0$.

7. [**10 points**] Does $L(G)$ satisfy the Pumping Lemma for regular languages? Briefly explain why or why not.

Yes. Let $n = 8$ be the Pumping Lemma constant. Then, by the lemma, every word $x \in L(G)$ can be decomposed as $x = uvw$ such that $|uv| \leq 8$, $|v| \geq 1$, and $uv^iw \in L(G)\ \forall i \geq 0$. Based

on the grammar and the previous analysis on the length of strings in the language, we can set $u = dnvdn$, $v = pdn$, and $w = x - uv$ (where $x - uv$ indicates the "tail" of w that is not contained in the prefix uv). Pumping the v portion of the string correponds to recursively adding more PPs in the grammar, so all strings uv^iw are also in $L(G)$.

8. [**5 points**] Is $L(G)$ a regular language? Briefly explain why or why not.

   Yes, $L(G)$ is a regular language. It can be represented by the regular expression $dnvdn(pdn)^*$.

Problem 2: Search

1. [**10 points**] What kinds of problems is depth-first search well-suited for, compared to breadth-first search? What kinds of problems is depth-first search not well-suited for? What kinds of problems is beam search well-suited for?

   Since it takes up a comparatively small amount of memory, depth-first search is well-suited for problems with very broad search spaces in which it would be infeasible to keep track of a large number of branches whose exploration is in progress, as would be necessary with a breadth-first search. It is also a generally faster algorithm for quickly finding the first solution in a search tree and returning it, so depth-first search is good for problems where it is expected that there are many solutions in the tree.

   Depth-first search is not well-suited for problems where the search space is very (or infinitely) deep, since the algorithm will attempt to exhaustively search a given branch before moving on to the next one. For the same reason, depth-first search may not be best for a situation in which the solution states are expected to be near the top of the tree, since the search will spend large amounts of time exploring the bottoms of the tree's other branches if it doesn't happen to choose the branch with the solution in it first.

   Beam search is an example of heuristic search, so it is useful for problems where not much is known about the layout of the search space, but where there is a way to evaluate the "score" or "cost" of a given state. It is especially useful for extremely large or infinite search spaces, since only a certain number of the best-scoring nodes that have been searched so far will be kept for future exploration. The algorithm will therefore concentrate its searching efforts in regions of the tree where there is a presumed good chance of finding a solution; this also makes it a good choice for applicaitons in which search time is constrained.

2. [**5 points**] In the breadth-first search algorithm described in the notes, there is a "NODE-LIST" for maintaining the list of not-yet-expanded nodes. The depth-first search algorithm does not have a similar data structure. Why is that?

   The depth-first search algorithm completely explores one branch of the search tree before moving on to another, so it can be written recursively (i.e. a call to depth-first search of a node generate a child node, then calls depth-first search on it). Thus, the program's call stack essentially keeps track of the nodes in the search tree that have not yet had all their children explored, which is equivalent to the NODE-LIST in breadth-first search.

3. [**5 points**] What is the motivation for the simulated annealing approach (e.g., what problems should it solve)? How does simulated annealing solve these problems?

   The simulated annealing approach is more complicated than hill climbing, but simple hill climbing only works in a concave search space (i.e. one that has exactly one optimum). If, on the other hand, the search space is a complex function with many local maxima, it is easy for hill climbing to reach one of the local high points, stop because all neighboring states look worse, and miss the global maximum.

   Simulated annealing overcomes this problem by allowing the search to jump over a better state and instead move to a state in the search space that looks worse, effectively allowing the

algorithm to move over a local maximum rather than get stuck at it. The goal is to eventually reach the global maximum without skipping over *it*, so an annealing schedule controls how often the algorithm is allowed to make a "bad" move. Near the beginning of the search, the probability of doing so is higher, but as the search converges, the probability of making a jump decreases.

Problem 3: Morphological and Lexical Analysis

1. [**5 points**] Why would an NLP application analyze the surface form of a word such as "cries" into a form such as **cry+s** before doing further processing? What role do lexical rules play in this process?

   In most NLP applications, where a certain amount of data (such as feature structures) is being stored for each lexical item, it would be inefficient to store mostly identical copies of the data for items that are very closely related. Since the lexical information for "cry" and "cries" is the same, except for the subject the words are conjugated for, an NLP system that did *not* analyze "cries" as "cry+s" would have to maintain two complete lexical entries.

   On the other hand, if words can be analyzed to their base forms plus some well-known morphemes, the lexical information that does not change can be extracted and stored once in the base-form entry, and a small set of general lexical rules can be created that write in the necessary additional features based on the particular morphemes found. This makes the overall lexicon simpler to extend, since properties of whole classes of words can be generalized.

2. [**5 points**] Why do most NLP systems store subcategorization frames in verb lexical entries (e.g. what is the alternative)? Why is the use of subcategorization frames generally considered preferable?

   The subcategorization of a verb affects how it can be used in a sentence; i.e. what sort of syntactic structures can be built around it. A verb such as "water," for example, must be used with an object NP and cannot be part of a VP that only contains a verb. If the subcategorization information is not stored at the lexical level, a variety of new parts of speech will have to be introduced (for transitive verbs, intransitive verbs, verbs that require locative PPs, etc.), and the grammar will have extra rules to allow only the correct type or types of verbs to combine with other constituents in the parse tree.

   For languages where verb morphology and subtyping is more extensive, this will result in a very large number of extra categories and rules, which will make the grammar more difficult to maintain and less efficient to work with computationally. Therefore it is preferable to store the subcategorization information as a feature at the lexical level and use it to constrain which higher-level constituents can be built.