The dream of automatic programming

Back in the 1960’s, there was a dream that one could simply state a problem and have it automatically translated into an algorithm that would solve it efficiently.

But few general results along this line were ever found.

Once exception is Cook’s theorem: a 2DPDA can be simulated in linear time on a RAM.

A 2DPDA is a deterministic pushdown automaton with a two-way tape: you can move both left and right on the tape. There are endmarkers surrounding the input: ♯ on the left and♭ on the right. Input is accepted if a final state is reached.

A RAM is a random access machine: an abstraction of assembly language, a general-purpose computer with registers and instructions.
2DPDA’s are a very interesting model that can accept a wide class of languages.

They are still not completely understood!

Even unary 2DPDA languages are very complicated!

Let’s look at an example of a 2DPDA language that is not a CFL: \( \{ a^n b^n c^n : n \geq 0 \} \).

How does a 2DPDA recognize this language?
How do we recognize \( \{a^n b^n c^n : n \geq 0\} \) with a 2DPDA?

First, make a left-to-right pass over the input to make sure it belongs to \( a^* b^* c^* \). Then move left to the left endmarker and start again.

Next, read the \( a \)'s and push them onto the stack. When we see \( b \)'s, pop them off and compare them to the \( b \)'s. If a mismatch, halt and reject. Now move left to the last \( a \) and push the \( b \)'s onto the stack. When you \( c \)'s, pop off and compare to the \( b \)'s. If it passes all of these tests, enter the halting state.
A 2DPDA is a 9-tuple $M = (Q, \Sigma, \Gamma, \#, b, \delta, q_0, Z_0, F)$ where

- $Q$ is a finite set of states;
- $\Sigma$ is the input alphabet;
- $\Gamma$ is the stack alphabet;
- $\#$ is the left endmarker;
- $b$ is the right endmarker;
- $\delta : Q \times (\Sigma \cup \{b, \#\} \times \Gamma \rightarrow Q \times -1, 0, 1 \times \Gamma^*$ is the transition function;
- $q_0$ is the initial state;
- $Z_0$ is the initial stack contents;
- $F$ is the set of final states
The meaning of $\delta(q, a, A) = (p, j, \alpha)$ is that if we are in state $q$, scanning the input symbol $a$, with $A$ on top of the stack, then the 2DPDA enters state $p$, moves $j$ cells to the right, and replaces $A$ on the stack with $\alpha$.

A full configuration is a triple of the form $(q, h, \alpha)$, where $q$ is the current state, $h$ is the number of the current cell being scanned (first cell is cell 0) and $\alpha$ is the current stack contents.

Then if the current full configuration is $(q, h, A\beta)$ and the $h$'th cell equals $a$, and $\delta(q, a, A) = (p, j, \alpha)$, then the next full configuration is $(p, h + j, \alpha\beta)$. We write

$$(q, h, A\beta) \vdash^* (p, h + j, \alpha\beta).$$

The language recognized by DPDA $M$ is

$$\{ x \in \Sigma^* : (q_0, 0, Z_0) \vdash^* (q, i, \alpha) \text{ for some } i \text{ and } \alpha \text{ and } q \in F \}.$$
Let us show how to recognize \( \{xx : x \in \{a, b\}^*\} \).

How can we do that?

A first pass through the input can ensure the length of the input is even.

Next, the basic idea is to push the first half of the input onto the stack from left to right (so it ends up on the stack \textit{unreversed}) and then compare it to the last half of the input.

But how do we find the middle of the string, so we can push the first half?
A deeper example

We use the following trick:

Move right, pushing a symbol onto the stack for each symbol of the input, up to the right endmarker. Next, move left, popping two symbols for each move, until the symbol $Z_0$ reappears on top of the stack. Now we are at the middle of the string.

After the stack is loaded with the first half, do the same middle-finding trick and then move right on the input, comparing the stack contents to the rest of the input.
Can you see how to recognize \( \{a^{2^n} : n \geq 0\} \) with a 2DPDA?
Another example

Can you see how to recognize \( \{ a^{2n} : n \geq 0 \} \) with a 2DPDA?

*Solution*: 1. Start by pushing an \( X \) onto the stack (there is a \( Z_0 \) initially).

2. Move to left endmarker, then to first symbol of the input. For each \( X \) on the stack, move right in input.
   
   ▶ If right endmarker ♭ is seen at the same time as \( Z_0 \) appears on the stack, accept.
   
   ▶ If right endmarker ♭ is seen at the same time that \( X \) appears on the stack, reject.

3. Otherwise when \( Z_0 \) appears on the stack, move left one cell in the input. Keep moving left on the input until the left endmarker ♯ is seen, pushing two \( X \)'s onto the stack for each \( a \) seen.

4. Return to step 2.
The language \{xcy : x, y \in \{a, b\}^* and y is a subword of x\} is a 2DPDA language. ("String matching")

Proof.
This one is a bit harder.
1. Move right on input until \(c\).
2. Move left, copying symbols of \(x\) onto stack until \(#\) seen. Now \(xZ_0\) is on the stack. Let \(x = a_1a_2 \cdots a_n\).
3. Move right until \(c\) seen. Move one cell right.
4. While the symbol on the stack matches the input pop the stack and move read head one cell to the right.
5. If the read head is scanning \(♭\), accept. If \(Z_0\) is on top of the stack, reject. Otherwise there is a mismatch between stack and input. Restore the stack by moving read head to the left, pushing symbols seen onto the stack until \(c\) is seen. At this point the stack has \(a_ia_{i+1} \cdots a_nZ_0\) on it for some \(i\). Pop one symbol from the stack. Move read head one cell to the right. Return to step 4.
Let’s see how this works on input *abbababcaba*.

<table>
<thead>
<tr>
<th>input</th>
<th>stack</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>♯abbababcaba</code></td>
<td><code>Z_0</code></td>
<td>push first part onto stack</td>
</tr>
<tr>
<td><code>♯abbababcaba</code></td>
<td><code>abbabababZ_0</code></td>
<td>match <em>a</em></td>
</tr>
<tr>
<td><code>♯abbababcaba</code></td>
<td><code>bbabababZ_0</code></td>
<td>match <em>b</em></td>
</tr>
<tr>
<td><code>♯abbababcaba</code></td>
<td><code>babababZ_0</code></td>
<td>mismatch</td>
</tr>
<tr>
<td><code>♯abbababcaba</code></td>
<td><code>abbabababZ_0</code></td>
<td>restore stack</td>
</tr>
<tr>
<td><code>♯abbababcaba</code></td>
<td><code>bbabababZ_0</code></td>
<td>pop one symbol</td>
</tr>
<tr>
<td><code>♯abbababcaba</code></td>
<td><code>bababababZ_0</code></td>
<td>mismatch</td>
</tr>
<tr>
<td><code>♯abbababcaba</code></td>
<td><code>bababZ_0</code></td>
<td>pop one; mismatch</td>
</tr>
<tr>
<td><code>♯abbababcaba</code></td>
<td><code>ababZ_0</code></td>
<td>pop one; match</td>
</tr>
<tr>
<td><code>♯abbababcaba</code></td>
<td><code>babZ_0</code></td>
<td>pop and new match</td>
</tr>
<tr>
<td><code>♯abbababcaba</code></td>
<td><code>ababZ_0</code></td>
<td>pop and new match</td>
</tr>
<tr>
<td><code>♯abbababcaba</code></td>
<td><code>ababZ_0</code></td>
<td>accept</td>
</tr>
</tbody>
</table>
Exercise: show to recognize the language of primitive words with a 2DPDA.
**Theorem.** If $L$ is recognized by a 2DPDA (using any number of moves), then it is recognized in linear time on a RAM.

*Proof idea:* Keep track of configurations; use short-circuiting to remove large sections of a sufficiently long 2DPDA computation.

**Corollary.** String matching can be done in linear time.

This should indeed be a little surprising, because our 2DPDA algorithm for string matching ran in quadratic time in the worst case.

Some 2DPDA algorithms actually can run in exponential time.
Cook’s theorem

A *partial configuration* is a triple or the form \((q, p, A)\), where \(q\) is the current state, \(p\) is the position that the head is scanning, and \(A\) is the current symbol on top of the stack.

A *terminator* of a partial configuration \(i\) is the configuration from which, on the next move, the stack height dips below that of \(i\).
The RAM uses a stack to store partial configurations.

As the 2DPDA executes, the RAM’s stack mimics the 2DPDA’s stack height. We assume WLOG that every 2DPDA move either pops, makes a horizontal move, or pushes a single additional symbol onto the stack.

The RAM’s stack holds configurations whose terminator is sought.

When we pop the stack we finally learn the terminator for some configurations.
Cook’s theorem

If we are in a partial configuration $i$, and we know $i$’s terminator $j$, we can “short circuit” the computation by jumping immediately to $j$.

Since the stack between $i$ and $j$ never dips below the height of $i$ and $j$, symbols buried in the stack can never affect the computation between them.

If we ever re-enter the same configuration at the same or higher stack height, then we must be in an infinite loop, and we can immediately reject.

The linear time bound comes from the fact that there are only linearly many different configurations. Each configuration is pushed onto the stack only once, and also popped only once.

Some individual steps might cause many terminators to be discovered, and hence cost more than $O(1)$, but the total cost over all such steps is still linear (amortized analysis).
Congratulations! You’ve made it to the end of CS 462 (except for the final exam).

I’ve enjoyed teaching this course and I hope you got something out of it.

If you didn’t fill out the course evaluations at https://evaluate.uwaterloo.ca yet, please do so.

The times are difficult now, but things will get better. Stay safe.