ECS 120 Lesson 26 – Space Complexity

Oliver Kreylos

Friday, June 1st, 2001

The last couple of lectures, we analyzed the time complexity of Turing Machines; in other words, we counted the number of computation steps a Turing Machine performs of an input word of given length. Another important property of Turing Machines is the amount of space they use, or, more exactly, the number of tape cells they ever look at during a computation. If this number is known in advance, it is not necessary to actually give a machine an infinitely long tape, but to only give it as many tape cells as it really needs.

1 Space Complexity

The definition of the space complexity of a Turing Machine is very similar to the definition of time complexity (just replace every occurrence of "time" by "space:"

Definition 1 (Space Complexity) *Let $f: \mathbf{N} \to \mathbf{N}$ be a function, and let M be a deterministic Turing Machine. If, on an input word of length n, the machine never moves its tape head past the first $f(n)$ cells on its tape, the* space complexity *of M is $f(n)$. We also say that M* runs in space $f(n)$*, or that M is an $f(n)$* space Turing Machine*.*

If N is a nondeterministic Turing Machine, and no branch of computation of N on input word w scans past the first $f(n)$ tape cells, then the space complexity of N is $f(n)$. We also say that N runs in space $f(n)$*, or that N is an $f(n)$* space nondeterministic Turing Machine*.*

In our discussion of time complexity, we also introduced the complexity classes $\mathrm{TIME}\big(f(n)\big)$ as the set of all problems that can be decided by

$O\big(f(n)\big)$ time Turing Machines. We define a classification of problems based on space complexity in the same way:

Definition 2 (Space Complexity Classes) *Let $f\colon \mathbf{N} \to \mathbf{N}$ be a function. Then the class* $\mathrm{SPACE}\big(f(n)\big)$ *is the set of all problems that can be solved by deterministic Turing Machines with space complexity $O\big(f(n)\big)$, and the class* $\mathrm{NSPACE}\big(f(n)\big)$ *is the set of all problems that can be solved by nondeterministic Turing Machines with space complexity $O\big(f(n)\big)$.*

2 Space Complexity Analysis

To analyze the space complexity of Turing Machines, we use the same techniques introduced in the chapter about time complexity: Counting of tape cells and asymptotic analysis using the big-O notation. As an example, let us reconsider the satisfiability problem for boolean formulas and analyze its space complexity. To recall, the algorithm to decide SAT is as follows:

Algorithm SAT On input $\langle \phi \rangle$,

1. Create all possible assignments of the variables of ϕ to the values true/false.
2. For each assignment generated, evaluate ϕ on it and check if the result is true. If so, accept.
3. If none of the assignments satisfied ϕ, reject.

We have already seen that this algorithm has a time complexity of $2^{O(n)}$; but as it turns out, its space complexity is much lower. In the main loop represented by steps 1 and 2, the machine generates an assignment of variables and writes it to tape. Since a formula of length n can at most contain $O(n)$ variables, any assignment can be written down using only $O(n)$ tape cells – one cell for each variable, containing either the symbol true or false. Since boolean formulas are described by a context-free grammar, to evaluate the formula the machine would invoke a PDA to parse the formula's structure and evaluate the subformulas' values on-the-fly. The PDA's stack can never contain more than $O(n)$ symbols, because the recursion depth of a formula of length n can be at most $O(n)$. Altogether, writing down and evaluating the formula can be done in space $O(n) + O(n) = O(n)$. After each loop iteration,

the machine will reuse the space it wrote the assignment into by updating the true/false values. This means that although $2^{O(n)}$ iterations are performed, the total space used is only $O(n)$. It looks as if space is "more powerful" than time; this intuition is backed by the fact that a Turing Machine can re-use space, but it cannot re-use time: SAT $\in$ TIME(2^n), but SAT $\in$ SPACE(n).

3 Relationship between SPACE and NSPACE

In time complexity analysis, we saw that converting a nondeterministic Turing Machine into its deterministic counterpart increases its time complexity by an exponential amount: NTIME$\big(f(n)\big) \subset$ TIME$\big(2^{O(f(n))}\big)$. Again, the following theorem hints at space seemingly being more powerful than time:

Theorem 1 (Savitch's Theorem) *Let $f\colon \mathbf{N} \to \mathbf{N}$ be a function, and let $P \in$ NSPACE$\big(f(n)\big)$ be a problem that can be solved by an $f(n)$ space nondeterministic Turing Machine. Then $P \in$ SPACE$\big(f^2(n)\big)$; in other words, a deterministic Turing Machine solving the same problem only suffers a quadratic increase in space complexity.*

The proof for Savitch's theorem is interesting in its own right, because it uses a general technique from algorithm theory to improve the space complexity of the standard conversion algorithm we have seen before: It employs a Divide-and-Conquer strategy. Let us begin the proof by recalling the old conversion algorithm:

Algorithm NTM $\to$ DTM On input $\langle N, w\rangle$, where N is an NTM and w is an input word,

1. Put the initial configuration of N on word w, (ϵ, q_0, w), into a queue.
2. While there are configurations in the queue:
   (a) Remove the first configuration from the queue.
   (b) Create all configurations that directly follow from the one just removed, and add them to the end of the queue.
   (c) If one of the created configurations was an accepting one, accept.
3. Reject.

This algorithm performs a breadth-first search of N's computation tree on input word w. It visits all configurations at depth 1 in the tree, then all at depth 2, then all at depth 3 and so forth. That is, the maximum number of elements that can be in the queue at any time is the maximum number of configurations at the same level in the computation tree times two: In pessimistic analysis, we would keep all the configurations for level n, generate all configurations for level $n+1$, and then remove all those for level n. Since the computation tree is a k-ary tree for some constant k determined by N's transition function, the maximum number of nodes at the same depth is k^n, where n is the depth. The total depth of N's tree is $f(n)$, therefore the queue can contain up to $O\big(k^{f(n)}\big) = 2^{O(f(n))}$ configurations. This means the given algorithm has space complexity $2^{O(f(n))}$.

To improve the algorithm, we go back to our definition of computation: A nondeterministic Turing Machine accepts a word w if and only if its initial configuration (ϵ, q_0, w) is related to an accepting configuration $(u, q_{\text{accept}}, v)$ by the extended turnstile relation $\vdash^*$. Let us construct a Divide-and-Conquer algorithm to decide whether one ID can be computed from another one in at most t steps. The basic idea is that if there is a chain of configurations from ID_1 to ID_2 of at most t steps, by transitivity of $\vdash^*$ there must be an intermediate configuration, say ID_M, such that $\text{ID}_1 \vdash^* \text{ID}_\text{M}$ and $\text{ID}_\text{M} \vdash^* \text{ID}_2$, where each of the partial chains is at most $\lceil t/2 \rceil$ steps long[1]. If we can somehow find this intermediate configuration, we can answer the original question by answering the two smaller questions independently. Only if the chain is short enough, i. e., either $t = 0$ or $t = 0$, we will directly test whether the two configurations are related. The easiest, brute-force way to find the intermediate configuration is just to test all possible ones. As an algorithm, this becomes:

Algorithm Turnstile On input $\langle \text{ID}_1, \text{ID}_2, t \rangle$, where ID_1 and ID_2 are two configurations and $t \in \mathbf{N}$ is a number,

1. If $t = 0$, compare ID_1 and ID_2; if they are equal, accept; otherwise, reject.

2. If $t = 1$, check if ID_2 can be reached from ID_1 in a single step of computation; if that is the case, accept; otherwise, reject.

[1] If t is odd, we round up $t/2$ to the nearest integer. This is denoted by the notation $\lceil t/2 \rceil$.

3. Otherwise, create all possible configurations ID_M, and run the algorithm Turnstile recursively on the two input words $\langle \mathrm{ID}_1, \mathrm{ID}_\mathrm{M}, \lceil t/2 \rceil \rangle$ and $\langle \mathrm{ID}_\mathrm{M}, \mathrm{ID}_2, \lceil t/2 \rceil \rangle$. If both accept, accept.

4. If none of the earlier steps accepted, reject.

It is important to note that this algorithm only works because the space complexity of machine N is known to be $f(n)$. Since the machine only uses $f(n)$ tape cells, the total number of possible configurations is finite, and all of them can be created and tested by the algorithm.

We can now use the above algorithm to answer the original question whether N accepts w: We create all possible accepting configurations ID_A of N (there can only be finitely many), and for each of those, we run algorithm Turnstile on input $\langle (\epsilon, q_0, w), \mathrm{ID}_\mathrm{A}, |Q| \cdot f(n) \cdot |\Gamma|^{f(n)} \rangle$. Since the maximum number of tape cells N uses is $f(n)$, the total number of possible configurations is $|Q| \cdot f(n) \cdot |\Gamma|^{f(n)}$ – recall our treatment of LBAs – and so the maximum number of steps N can perform is limited by the same number. If N would perform more steps, it would have to loop[2]; this can never happen because N is a deciding Turing Machine.

In terms of time complexity, this algorithm is no better than the original one[3]. But it seriously cuts down on space usage: To keep track of the recursion, the machine builds a stack containing the parameters passed to the recursive function calls. The total size of each parameter set is $O\big(f(n)\big)$, and since the size of t is cut in half in each recursion step, the maximum recursion depth is $\lceil \log_2 t \rceil$. Since $t = O\big(|\Gamma|^{f(n)}\big) = 2^{O(f(n))}$, the maximum recursion depth is $\log_2\big(2^{O(f(n))}\big) = O\big(f(n)\big)$. Therefore, the space complexity of the algorithm is $O\big(f(n)\big) \cdot O\big(f(n)\big) = O\big(f^2(n)\big)$ as claimed.

3.1 $\mathcal{P}$SPACE and $\mathcal{NP}$SPACE

We can define the classes $\mathcal{P}$SPACE and $\mathcal{NP}$SPACE analogously to $\mathcal{P}$ and $\mathcal{NP}$:

Definition 3 ($\mathcal{P}$SPACE and $\mathcal{NP}$SPACE) *$\mathcal{P}$SPACE is the class of all problems that can be decided by deterministic Turing Machines in polynomial space; i. e.,*

$$\mathcal{P}\mathrm{SPACE} = \bigcup_{k>=0} \mathrm{SPACE}(n^k) \ .$$

[2]Pigeonhole Principle strikes again!

[3]In fact, it is probably much worse.

$\mathcal{NP}$SPACE *is the class of all problems that can be decided by nondeterministic Turing Machines in polynomial space; i. e.,*

$$\mathcal{NP}\text{SPACE} = \bigcup_{k>=0} \text{NSPACE}(n^k) \ .$$

As opposed to the relation between $\mathcal{P}$ and $\mathcal{NP}$, we know by Savitch's theorem that $\mathcal{P}$SPACE and $\mathcal{NP}$SPACE must be identical: If a problem $P \in \mathcal{NP}$SPACE is decided by an NTM in $O(n^k)$ space, then it is decided by a DTM in $O\big((n^k)^2\big) = O(n^{2k})$ space. Since $O(n^{2k})$ is a polynomial of degree $2k$, $P \in \mathcal{P}$SPACE as well. Therefore, $\mathcal{P}\text{SPACE} = \mathcal{NP}\text{SPACE}$.

We also can deduce the following inclusion: If a problem is in $\mathcal{P}$SPACE, it must run in exponential time – the total number of configurations of an $O(n^k)$ space Turing Machine is limited by $2^{O(n^k)}$; therefore, if it runs longer than $2^{O(n^k)}$, it must loop – which it cannot. Thus, the class $\mathcal{P}$SPACE is embedded in the class EXPTIME that contains all problems that can be decided in exponential time, and our current view of the fine structure of the class of decidable languages becomes:

$$\text{CFL} \subset \mathcal{P} \subset \mathcal{NP} \subset \mathcal{P}\text{SPACE} = \mathcal{NP}\text{SPACE} \subset \text{EXPTIME} \subset \text{DL}$$

Where does CH–1, the class of context-sensitive languages, fit into this picture? I have not found any references about this question, but I assume that CH–1 straddles $\mathcal{P}$, $\mathcal{NP}$ and $\mathcal{P}$SPACE: Every language in CH–1 must by definition be decided by an $O(n)$ space Turing Machine, but some problems in CH–1 might take exponential time to run – testing the primality of a number, for example – and, on the other hand, some problems in $\mathcal{NP}$ or even $\mathcal{P}$ might need more than $O(n)$ space.