Remember to Use Memoization!

Seminar at University Of Wisconsin - Parkside

Clément Aubert

13/02/2017

Abstract

Memoization is an optimization technique that trades (a bit of) space for (a lot of) time. It comes in multiple forms, and you may already use it without knowing it. This technique cannot always be applied, but when properly used, it can speedup the running time from exponential to linear!

We'll motivate the introduction of memoization with some examples, introduces to its origins, and mention the limitations of this technique. Although not necessary to understand this lecture, some preliminary material, as well as exercises, will be posted at http://lacl.fr/~caubert/UWP/.

When and Where?

When

Friday, February 13, 2017, 12:00 PM – 1:00 PM

Where

MOLN 130, Molinaro Hall, University Of Wisconsin - Parkside, 900 Wood Road · P.O. Box 2000 · Kenosha, WI.

Preliminary Material

Motivational Example

Assume your colleague/classmate Bertrand asks you to review his (`java`) code:

```java
public class Example{
    public static int f(int n){
        int x = 0;
        // Assume some code here, possibly changing the value of x
        return x;
    }

    public static void main(String[] args){
        System.out.println(f(35));
        // Some other code.
        // And then, after a while
        System.out.println(f(35) + "is a nice number");
    }
}
```

You feel like Bertrand is uselessly computing `f(35)` twice, and decide to “improve” his code by doing the following:

```java
public class Example{
    // You leave his function f unchanged,
    public static int f(int n){
        int x = 0;
        // Assume some code here, possibly changing the value of x
        return x;
    }

    public static void main(String[] args){
        // But you store the value of f(35) into a variable:
        int x = f(35);
        // And then use that variable:
        System.out.println(x);
        // Some other code, unchaged
        System.out.println(x + "is a nice number");
    }
}
```

You send back your “optimized code” to Bertrand, but Bertrand starts complaining that you broke his program.

1. Is Bertrand right, could it be that you broke his program?

2. Give two examples of definition of the function `f` where your optimization would be correct.
3. Can you give three examples of definition of the function `f` where you optimization *wouldn't be* correct?
4. What are the condition on `f` and on the system for this transformation to be indeed an optimization?

Solutions:

Automata Theory

To study memoization, we'll have a peek at the context in which it was created: automata theory. It is not needed to have an extensive knowledge of this topic to understand the lecture, but the following should help you get some familiarities with the main concepts.

Definition of Automata Generally speaking, an *automaton* (a.k.a. "finite automaton", or "finite state machine") is

- A *finite alphabet* Σ,
- A *finite set of states* S, including an *initial state* s_i and an *accepting state* s_a,
- A *transition function* $\delta : \Sigma \times S \to S$.

Intuitively, the automaton is given an input word $n \subseteq \Sigma^{|n|}$, and then it reads the first character of n and applies its transition function: according to its initial state s_i, it changes its state, and move to the next character. The transition function is then applied until it is not defined for the couple (current character, current state). When there is no more character to read in n, or when the transition function can't be applied, the automaton halts. If the automaton stops in the state s_a, then we say that *the automaton accepts the word n*, otherwise we say that it *rejects the word n.*

There are plenty of variations on automaton. We are going to study two of them:

1. The addition of the capacity to move back and forth on the input word,
2. The addition of a "pushdown stack".

Exercises:

1. Try to write the definition of a "Two-way finite automata", i.e., of an automaton capable of moving back-and-forth on the input word. *Hint: in a first approximation, you only need to change the co-domain of the transition function, and the definition of halting.*
2. You want to prevent your automaton from "falling out of the input", i.e., to read an instruction "go fetch the previous character" when you're already at the first character. How to do that? *Hint: suppose the input word is surrounded by "end*

markers", that tell the automaton that it reaches the beginning or end of the word, and impose a condition on the transition function.

3. Now, you supposedly have the correct definition of a "Two-way finite automaton". Compare your definition with the one on Wikipedia.

Code Used During the Lecture

All the code showed during the lecture was in `Java`. You can download it, or play on-line with it at rextester or at tutorialpoint (please fork it before editing it).

To Go Further

How to Use Memoization with your Favorite Language?

Numerous resources are dedicated to this question, cf. for instance the ~700 questions tagged "memoization" on stackoverflow.com. If you are interested in using memoization for a particular language, you may be interested in the following resources:

- **In `C++`**, memoization is one of the common technique to optimize your program.
- **In `C#`**, this article will give you the main keys to memoize functions with multiple arguments.
- **In `Haskell`**, the wiki has a really complete page, with numerous examples and useful references.
- **In `Java`**, in complement of the code I gave, you can first look at a simple implementation before considering an interesting variation.
- **For `Javascript`**, a very complete article, including benchmarking and comparisons of implementation techniques, should get you started.
- **In `Perl`**, "memoize" is already a part of the core modules, and its regex engine actually uses memoization! For an excellent explanation of this latter point, in `Perl` and linking it to automaton, cf. this article.
- **In `Python`**, this page will provide a gentle introduction to memoization, and on how to perform it using decorators.
- **In `R`**, consult James Balamuta's slides on *Functions, Recursion, Benchmarking, and Memoization.*
- **In `Ruby`**, this article will give you some hints on how to use memoization, but the bottom line is: use memoist (which relies on ActiveSupport::Memoizable, that happens to be sometimes counter-productive!)

Memoization is virtually available for every programming language. If you feel like I forgot your favorite language, if you find a better technique that the ones listed above, feel free to send a line! If you have an account on Codewars, you can also browse the memoization tag to get some hand-on experience. You can also browse the memoization tag on codereview to get a practical sense of when is this method used.

Research Papers

As a warm-up, to understand how memoization is used to prove that the halting problem for Pushdown Automata (called PDA) is decidable in polynomial time, this answer is probably well-suited. It is clear and compact, but assume some knowledge of automata and languages.

- *Simulation of Two-Way Pushdown Automata Revisited* is a clear, brilliant, and compact presentation of the memoization technique used to simulate Two-way pushdown automata.
- *Partial memoization for obtaining linear time behavior of a 2DPDA* is an accessible, self-contained and relatively short (10 pages) paper that introduces a generalization of memoization called "partial memoization".
- *Compile-Time Function Memoization* is a recent (it was presented at CC 2017, a conference held last week at Austin!) paper that proposes to implement memoization in `C++` compilers!

Misc.

- Download a `pdf` version or the `md` source of this page
- HTML5 and CSS3 valid
-
- Contact: aubertc@appstate.edu
- Created with debian, pandoc, latex and emacs.