Computational Methods and Optimizations on the Cox-Ross-Rubenstein Model for Options Pricing

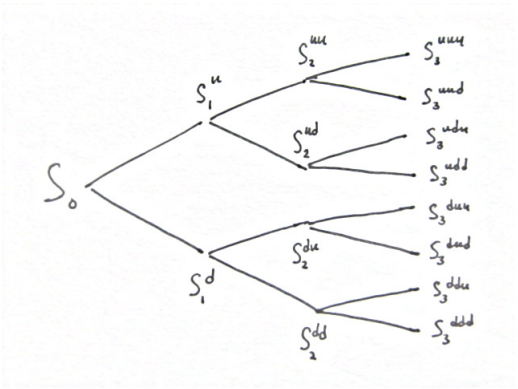
Andrew W. Cheong
Professor Chjan C. Lim
Financial Engineering

Abstract

In this paper, we explore several optimizations that significantly reduce the time and space complexity of the Cox-Ross-Rubenstein options pricing algorithm. These optimizations mainly take advantage of the recombinant nature of binomial trees for European and certain American options (i.e. those on securities with zero or fixed-rate dividends). Furthermore, we present interesting patterns and relationships that might be exploited for approximation algorithms. Finally, we discuss possible implementations that solve inverse problems such as computing implied volatility.

Background

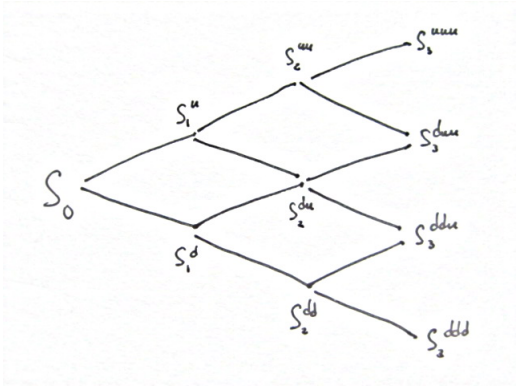
In 1979, Cox, Ross, and Rubenstein first proposed a binomial options pricing model. In this model, the possible evolutions of the price of an underlying financial instrument (such as a stock) are mapped to a discrete-time multi-period binomial tree. Such a tree is generated up to the maturity of the option. Since we are able to compute the possible values of the option at maturity, we recursively perform a backward walk along the tree in order to value the option at time zero.



Due to its flexibility, the binomial options pricing model is in many cases preferred over the Black-Scholes model, particularly for longer-dated options and options on securities with dividend payments. Unfortunately however, the binomial options pricing model is slow and computationally demanding. For a binomial tree with n periods, the total number of nodes is 2^{n+1}.[1] This means that, in terms of time, 2^n forward and backward walks must be computed, and in terms of space, 2^{n+2} stock and call prices must be stored. Since an accurate valuation of options requires thousands of periods, the limitations of using an unoptimized algorithm are clear.

Optimization

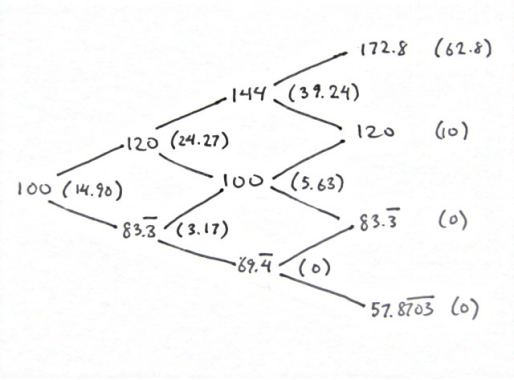
First, we narrow down the set of options to those that can be priced by *recombinant binomial trees*.



Options can be priced using recombinant binomial trees as long as the underlying stock prices are *path-independent*. That is, given j upward moves and n-j-1 downward moves, the resulting stock price should be constant regardless the order of upward and downward moves.[2] This condition is met for European options and American options (for stocks with zero or fixed-rate dividends). For n periods, a recombinant binomial tree requires only $\frac{1}{2}(n^2+n)$ nodes instead of 2^{n+1}. This brings down the number of forward and backward walks from 2^n to n^2+n, and also the number of stored stock and call prices from 2^{n+2} to n^2+n.

Next, we are able to make further optimizations on the forward walk on stock prices, as well as the backward walk on call prices.

In the absence of dividends, there are only 2n-1 unique stock prices. For example, in the 4-period tree below, there are only 7 unique stock prices.



1 Constant terms are neglected from all such expressions.

2 Under our notation, the first period is *0*, the last period is *n*-1, and there are *n* total periods.

Additionally, the unique stock prices all exist in the last two periods of the tree. Thus, we need only to compute the stock prices in the last two periods. The following array stores the stock prices for the example above.

k	0	1	2	3	4	5	6
S[k]	57.87	69.44	83.33	100	120	144	172.8

In the general case, such an array can be initialized with only 2*n* computations (instead of computing an entire sequence of forward walks). The initialization takes nothing more than a simple loop:

```
for (i=0..n-1)
{
        S[2*i] = S0*(d^(n-1-i))*(u^i);
        S[2*i+1] = S0*(d^(n-2-i))*(u^i);
}
```

Meanwhile, in order to find any stock price at period *i* after *j* upward moves, we simply access

```
S[2j+n-i-j]
```

In the case that fixed-rate dividends are involved, we simply multiply every entry of S[] by $(1-\delta)^{\eta}$, where δ is the dividend rate and η is the total number of dividend payments. Then, as we perform the backward walk, we divide the stock prices by $(1-\delta)$ whenever the current period contains an ex-dividend date.

Call prices can be mapped to a single array as well. At period *i*, there will be *i*+1 possible stock prices, and thus *i*+1 possible call prices. Since the last period is *n*-1, we need only to initialize an array of size *n*. Continuing with our example, we compute such an array below.

j	0	1	2	3
C[j]	0	0	10	62.8

During the backward walk, we do not need to retain future call prices. Thus, we may overwrite the j^{th} entry according to the formula

$$C_j = [pC_{j+1} + (1-p)C_j] / (1+rdt)$$

where *p* is the risk-neutral probability, *r* is the interest rate, and *dt* is the time-length of one period. Thus, the number of call prices to store in memory is at most *n*.

By implementing a recombinant tree and computing only the necessary stock and call prices, we reduce the algorithm's time complexity from $2n$ to $\frac{1}{2}(n^2+3n)$ and space complexity from 2^{n+1} to $3n$.

Implementation

Since we are mainly concerned with the efficiency of the algorithm, the specific programming language and machine architecture are irrelevant. Thus, for ease of coding and extensibility, we implement the algorithm in Perl.

Three programs have been written in order to compare running times. Program A uses recursive calls to build a non-recombinant tree and price the option. Program B uses recombinant trees, and also a compact array for call prices. Program C uses recombinant trees and compact arrays for both stock and call prices. The table below lists benchmark times (in milliseconds) as each program prices an American call option with dividends using different numbers of periods.[3]

n	**10**	**15**	**20**	**100**	**1000**	**5000**
A	91.8	465.4	6398.8	∞	∞	∞
B	77.9	79.5	80.0	157.6	2424.4	55741.1
C	78.5	80.0	80.4	143.8	2008.3	44009.1

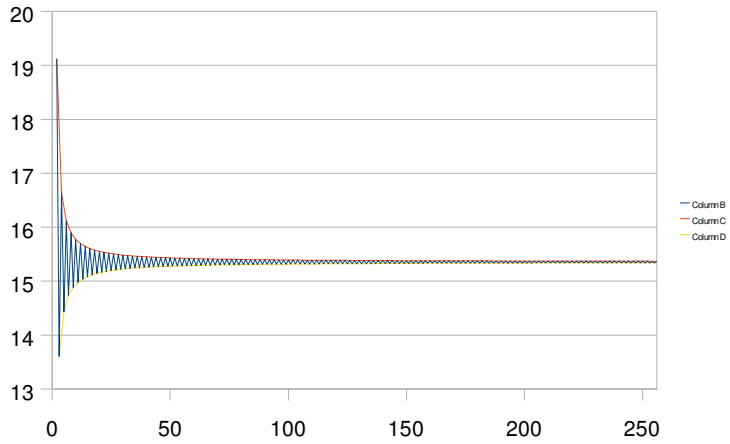
Despite that the algorithm is implemented in a high-level interpreted (slow) language, once armed with all the aforementioned optimizations, it is able to crunch over 6000 periods in less than a minute on a standard 2.0 GHz CPU.

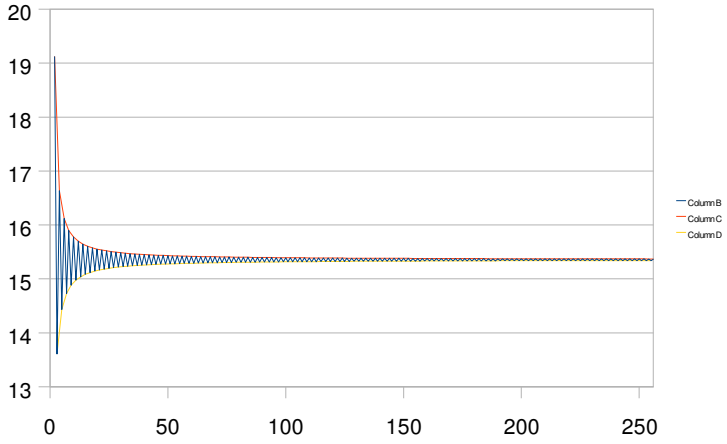
3 Each benchmark time is an arithmetic average of 10 independent trials.

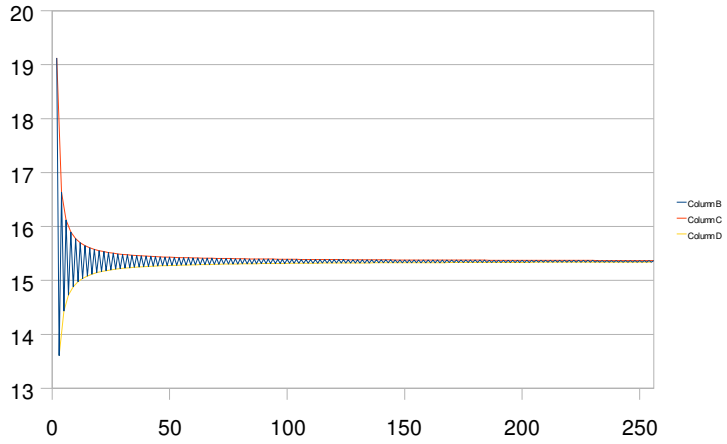
Observations

Another program has been written to study the fluctuation of prices as we increase the number of periods, holding all other variables (including time to maturity) constant. The program starts by computing a price with the single-period binary model, then with a two-period binary model, and so on, up to 256 periods.

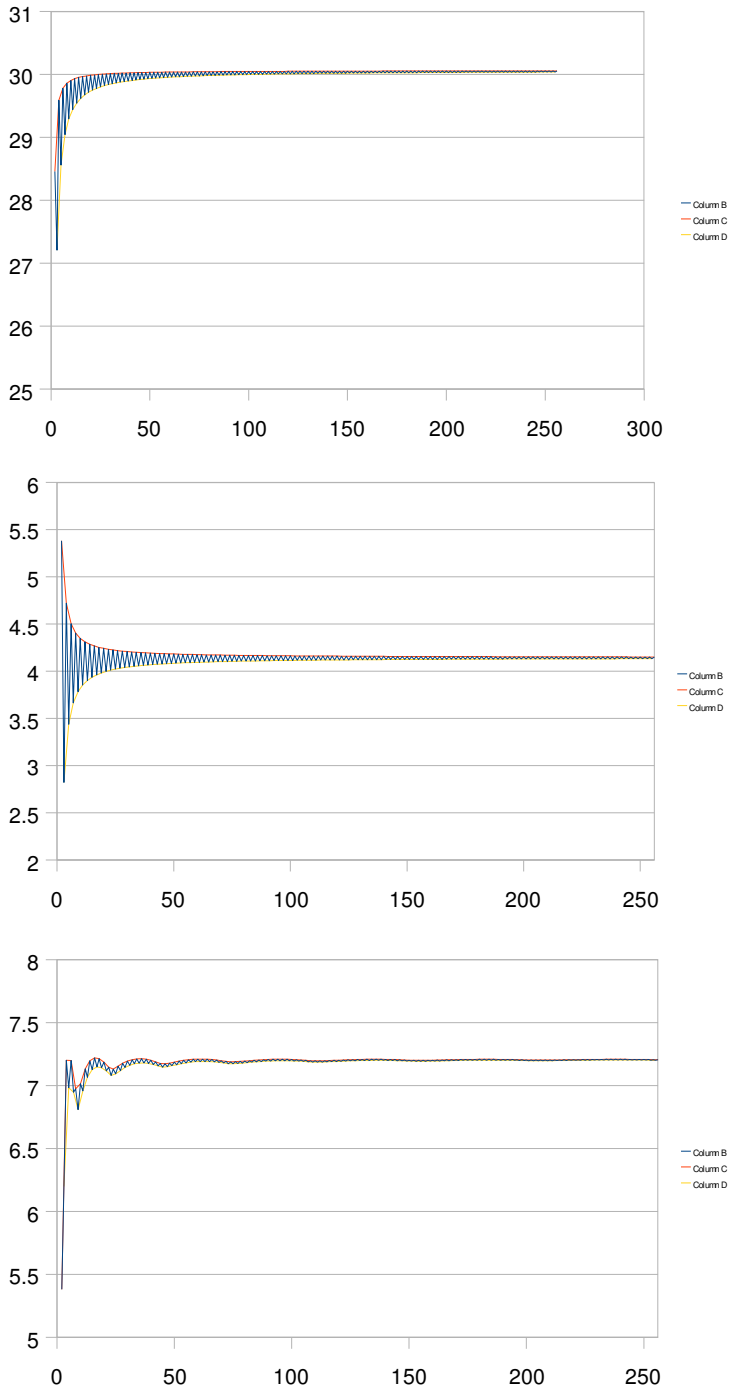
Below are graphs for a European call, European put, and American put, respectively, that has 30 days until maturity and a strike price of 100. The current underlying stock price is 100, the volatility is 0.005, and the interest rate is zero.







As it happens in this example, the call and put prices are almost exactly equal. However, the interesting trend comes in the analysis of even and odd numbers of periods. In each graph, the blue zig-zagging line is the actual fluctuation of prices as periods are increased. The top red line represents option prices generated from 2-, 4-, 6-...period models, and seems to act as an upper bound. The bottom yellow line represents option prices generated from 3-, 5-, 7-...period models and seems to act as a lower bound. When interest rates are set to (an exaggerated) 1% per day, the graphs fluctuate more wildly. However, the even-odd-bounding trend continues to appear.[4]



4 All graphs are of the form, P(n), where n is the number of periods used in the model to determine the option price P(n).

With a few more experiments, it is clearly visible that the interest rate and dividends are strong determinants for the level of initial fluctuation. However, all fluctuations are almost immediately damped, so that after a sufficient number of periods, prices are fairly consistent.

It is possible to develop (or improve) approximation algorithms using the upper and lower bound conjectures.[5] Note that computing several prices with lower periods are significantly faster than computing even a single price with many periods. For example, one might compute P(1000), P(1111), P(1222), P(1333); then analyze the extrema by finding ΔP(1000), ΔP(1111), ΔP(1222), ΔP(1333); and use the fact that even-period computations (1000, 1222) are upper bounds and odd-period computations (1111, 1333) are lower bounds, in order to extract a fairly accurate and precise approximation of the exact price. If we were to use Program C, such an estimate would take only about 6% of the time it would take to compute P(10000).

Approximation algorithms are especially useful for developing fixed-point algorithms (or any other algorithms that start with an initial guess) that converge to solutions of inverse problems. For example, in order to compute the implied volatility starting with a guess of 0.01, the resulting call price needs to be *only as precise as the magnitude of error in the previous adjustment to the implied volatility*. (We know how precise a call price is by evaluating that price with both *n*-period and *n+1*-period models.) This sufficiently adjusts implied volatility in the correct general direction during each iteration.

5 The conjecture can be proven analytically by proving that there exists an M such that for all n > M, P(2n-1) < P(2n) and P(2n+1) < P(2n). This proves to be a very difficult problem, even in the absence of interest rates and dividends, because *no backward walk can be a subset of a larger backward walk, given the same volatility*.