On A Recursive Algorithm For Pricing Discrete Barrier Options

Dennis G. Llemit
Department of Mathematics
Adamson University
1000 San Marcelino Street, Ermita, Manila

Abstract

An alternative and simple algorithm for valuating the price of discrete barrier options is presented. This algorithm computes the price just exactly the same as the Cox-Ross-Rubinstein (CRR) model. As opposed to other pricing methodologies, this recursive algorithm utilizes only the terminal nodes of the binomial tree and it captures the intrinsic property, the knock-in or knock-out feature, of barrier options. In this paper, we apply the algorithm to compute the price of an Up and Out Put (UOP) barrier option and compare the results obtained from the CRR model. We then determine the time complexity of the algorithm and show that it is $\Theta(n^2)$.

Keywords: UOP barrier option, time complexity, pricing algorithm

1 Introduction

An option is a derivative contract which confers to the owner (or buyer) the right but not the obligation to buy or sell certain amounts of an underlying at a future time at a predetermined price. Knock-out barrier options are derivative contracts which remain valid if and only if the underlying S_t has never breached the barrier level B throughout their lifespan. In particular, an Up and Out Put (UOP) barrier option is a contract which confers to the owner the right to sell units of the underlying instrument at strike price K at maturity date T if and only if

$$S_t < B \tag{1}$$

for all $t \in \{0, 1, \cdots, T\}$.

Determining the premium price V_0 of a derivative contract is the heart and soul of Option Pricing Theory. In continuous time, the Black-Scholes model is usually used to get V_0. This involves solving a partial differential equations (PDE) with certain boundary conditions [2]. However, an equivalent value can be obtained via risk-neutral valuation. Under this principle, it is assumed that the world is indifferent to risk such that the value of an option is the expected pay-off discounted by the risk-free interest rate r [6]. That is,

$$V_0 = e^{-rT}\,\mathbb{E}\left[V_T\right], \tag{2}$$

where V_T is the value of the option at maturity. This principle is also used in the Cox-Ross-Rubinstein (CRR) binomial model which is one of the most widely used valuation methods in discrete time.

In 2009, Tina Sol, together with her adviser Dr. van der Weide of Technical University of Delft, conceptualized a recursive algorithm that computed the price of a knock-out call barrier option. The values that they obtained using their algorithm were exactly equal to the values computed using the CRR model. The algorithm was then compared to other computing algorithms and found that it was unique [8]. Hence, they concluded that their's is a new algorithm in computational finance. For the purpose of identification and precision, we will call the algorithm developed by Sol as the Sol - van der Weide algorithm or SWA.

In this paper we intend to (1) verify the accuracy of the Sol - van der Weide algorithm (SWA) relative to the CRR model using a knock-out put barrier option, and to (2) determine the algorithm's time complexity. The SWA is a promising computational method since it is very simple and resilient compared to the CRR. Hence, it is imperative that we apply the algorithm to a knock-out put option in order to complete the story behind its viability as an alternative to the CRR. Although, we are using a particular type of knock-out put option (UOP), the framework of price valuation would just be similar and would be applicable to other knock-out put derivative contracts. If it can be shown that the algorithm does the same job as the CRR and better at that, then it will give finance practitioners an alternative computational tool in pricing barrier options. The second objective seeks to provide the time complexity analysis of the algorithm which was not provided by Sol. Determining the time complexity of the algorithm will enable us to compare the algorithm to the CRR's known time complexity.

2 Up and Out Put Barrier Option Price

2.1 CRR Price

The pay-off function of an option is the non-negative difference between the strike price K and the underlying value. That is,

$$V_t = (K - S_t)^+ = \max\{K - S_t, 0\}. \tag{3}$$

Suppose that there are n time steps, then we can write the maturity value of the option, as well as the underlying, to end at a node $(n, 2j - n)$ as

$$V_T\left(n, 2j - n\right)$$

and

$$S_T(n, 2j - n) = S_0 u^j d^{n-j}, \tag{4}$$

where j is the number of up-steps in the binomial tree.

Thus, using the risk-neutral principle, the option price at node $(n, 2j - n)$ is given by

$$\begin{aligned} V_0 &= e^{-rT}\,\mathbb{E}\left[V_T\left(n, 2j - n\right)\right] \\ &= e^{-rT}\sum_{j=0}^{n} P\left(n, 2j - n\right)\left(K - S_0 u^j d^{n-j}\right)^+ \end{aligned} \tag{5}$$

where $P(n, 2j - n)$ stands for the probability that the underlying will end at node $(n, 2j - n)$. Since our tree is binomial, we come up with

$$V_0 = e^{-rT} \sum_{j=0}^{n} \binom{n}{j} p^j q^{n-j} \left(K - S_0 u^j d^{n-j}\right)^+ . \tag{6}$$

To incorporate the "in-the-money" feature into equation (6), we determine the maximum number of up-steps x such that the underlying stays below the strike price. We do this by solving for j in

$$\begin{aligned} S_0 u^j d^{n-j} &< K \\ \left(\frac{u}{d}\right)^j &< \frac{K}{S_0 d^n} \\ j &< \frac{\log\left(\frac{K}{S_0 d^n}\right)}{\log\left(\frac{u}{d}\right)}. \end{aligned}$$

We then set

$$x = \lfloor j \rfloor = \left\lfloor \frac{\log\left(\frac{K}{S_0 d^n}\right)}{\log\left(\frac{u}{d}\right)} \right\rfloor .$$

To incorporate the knock-out property, we replace $\binom{n}{j}$ with $F(n, 2j-n)$ which represents the number of paths from the origin to a node $(n, 2j-n)$ that stay below the barrier level B. Consider nodes $(0, 2m)$ and $(n, 2j-n)$ where m is the height of the barrier level (see Figure 1). By Andre's symmetry principle [1], the number of paths from $(0, 2m)$ to $(n, 2j - n)$ that traverse the barrier is just

$$\begin{aligned} & \binom{\text{total steps taken}}{\text{height difference + half of remaining steps}} \\ = & \binom{n-0}{2m - (2j-n) + \frac{1}{2}\left(n - (2m - (2j-n))\right)} \\ = & \binom{n}{n+m-j} \\ = & \binom{n}{j-m}. \end{aligned} \tag{7}$$

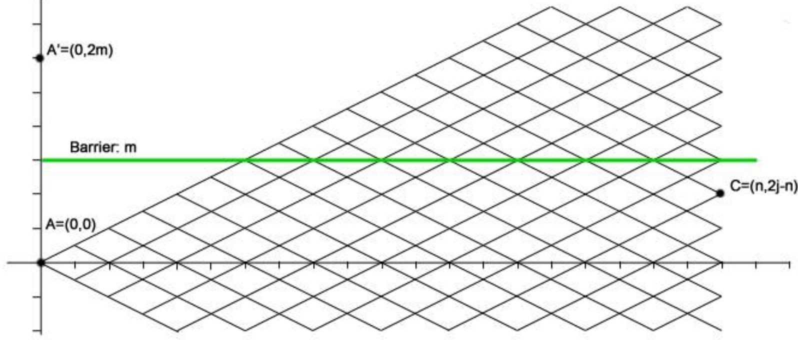

Figure 1: Binomial tree with the indicated nodes

Since, this value represents the paths that cross the barrier level, we subtract it from the total path from the origin to a node $(n, 2j-n)$ to obtain

$$F(n, 2j-n) = \binom{n}{j} - \binom{n}{j-m}. \tag{8}$$

Hence, we obtain the explicit pricing formula for an Up-and-Out put barrier option.
If $x > m$ then

$$V_0 = e^{-rT} \sum_{j=0}^{m-1} \left[\binom{n}{j} - \binom{n}{j-m} \right] p^j q^{n-j} \left(K - S_0 u^j d^{n-j} \right). \tag{9}$$

If $x < m$, then

$$V_0 = e^{-rT} \sum_{j=0}^{x} \left[\binom{n}{j} - \binom{n}{j-m} \right] p^j q^{n-j} \left(K - S_0 u^j d^{n-j} \right). \tag{10}$$

Cases $x = m$ and $j \geq m$ will cause the option to be worthless.

2.2 SWA Price

The first part of the algorithm is called the initialization subprocedure since it essentially initializes the vectors representing the underlying and pay-off values at maturity T for a number of time steps n.

Initialize the underlying and pay-off vectors

$$S_T = \begin{pmatrix} S_T(n,n) \\ S_T(n,n-2) \\ \vdots \\ S_T(n,-n+2) \\ S_T(n,-n) \end{pmatrix} = \begin{pmatrix} S_0u^n \\ S_0u^{n-1}d \\ \vdots \\ S_0ud^{n-1} \\ S_0d^n \end{pmatrix}$$

$$V_T = \begin{pmatrix} V_T(n,n) \\ V_T(n,n-2) \\ \vdots \\ V_T(n,-n+2) \\ V_T(n,-n) \end{pmatrix} = (K - S_T)^+ \, .\times \, (S_T < B) \, .$$

Here, $.\times$ represents pointwise vector multiplication. The vector $(S_T < B)$ contains logicals 0 or 1. If the underlying is below the barrier B then the vector contains only $1's$. Otherwise, the vector contains only $0's$. This vector clearly captures the knock-out property of UOP.

The second part is called the recursion subprocedure since it recursively runs the operation until it obtains the single entry vector V_0 which is the option price of UOP.

For $i = 1, 2, \cdots, n$ and $h = \frac{T}{n}$ update the pay-off vector

$$V_{T-ih} = e^{-rih} \cdot \left(pV_i^{up} + qV_i^{down}\right) . \times (S_{T-ih} < B) \, . \tag{11}$$

Run recursively until V_0 is obtained.

Here, h is the length of each time step. The updating of vectors requires that they must be based from the original vectors V_T and S_T. The vector S_{T-ih} is attained by using either one of the two formulas:

$$S_{T-ih} = u \cdot S_i^{down} \tag{12}$$

or

$$S_{T-ih} = d \cdot S_i^{up} \tag{13}$$

where vectors S_i^{down} and S_i^{up} are obtained by deleting the first i and the last i entries, respectively, of S_T. The multiplier u or d makes sure that the nodes or elements of either S_i^{down} or S_i^{up} will coincide with the nodes or elements of the next i^{th} vector, S_{T-ih}. Similarly, the vectors V_i^{up} and V_i^{down} are obtained from V_T in the same manner. The deletion of entries goes on until they become single entry vectors.

3 Algorithm Implementation

The only algorithm modification that we made in this paper relative to that of Sol's is the expression $K - S_T$ in order to suit a put derivative contract. The resulting $Matlab$ code, $UOPSWAprice(S, K, BT, m, r, gamma, sigma)$ is found in the appendix.

As for the CRR formula, we implement it in $Matlab$ using Sol's framework [8]. Accordingly, the formula requires careful programming especially the term $F(n, 2j-n) = \binom{n}{j} - \binom{n}{j-m}$ because $Matlab$ cannot handle large values for n. The resulting $Matlab$ code is $UOPCRRprice(S, K, B, T, m, r, gamma, sigma)$ in the appendix.

In both $m.files$, we used Boyle and Lau's formula for determining the number of time steps n to avoid "bumping up against the barrier" [3]. Mispricing of barrier options happens when the true barrier lies between two nodes. This is prevented by choosing an appropriate number of time steps n. In Boyle and Lau's work, they came up with $n = \left\lfloor \frac{m^2\sigma^2 T}{log^2(B/S)} \right\rfloor$.

Lastly, the values that we used for the constants were $r = 0.056$, $\gamma = 0.007$, $\sigma = 0.13$, $S_0 = 0.0083$, $K = 0.0080$, $B = 0.0091$, and $T = 0.5$ which can be found in [5].

4 Time Complexity Analysis

To begin our analysis, we define the following measures of time complexity according to Cormen [4] :

Definition (Big O Complexity)**.** *For any monotonic functions $f(n)$ and $g(n)$ where $n \geq 0$, we say that $f(n) = O(g(n))$ when there exist constants $c > 0$ and $n_0 > 0$ such that $f(n) \leq c \cdot g(n)$, for all $n \geq n_0$.*

Definition (Big Omega Complexity)**.** *For any monotonic functions* $f(n)$ *and* $g(n)$ *where* $n \geq 0$*, we say that* $f(n) = \Omega(g(n))$ *if there exist a constant* c *such that* $f(n) \geq c \cdot g(n)$ *for all sufficiently large* n*.*

Definition (Big Theta Complexity)**.** *For any monotonic functions* $f(n)$ *and* $g(n)$ *where* $n \geq 0$*, we say that* $f(n) = \Theta(g(n))$ *if there exist constants* c_1 *and* c_2 *such that* $0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ *for all sufficiently large* n*.*

Then we state a series of theorems from Rosen [7].

Theorem 1. *For any monotonic functions* $f(n)$ *and* $g(n)$ *where* $n \geq 0$*, we say that* $f(n) = \Theta(g(n))$ *if and only* $f(n) = O(g(n))$ *and* $f(n) = \Omega(g(n))$ *for all sufficiently large* n*.*

Theorem 2. *Let* $f(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$*, where* $a_0, a_1, \cdots, a_n$ *are real numbers with* $a_n \neq 0$*. Then* $f(x)$ *is of order* x^n*.*

Theorem 3. *Suppose that* $f_1(x)$ *is* $O(g_1(x))$ *and* $f_2(x)$ *is* $O(g_2(x))$ *. Then* $(f_1 + f_2)(x)$ *is* $O\left(\max\left(\left|g_1(x)\right|, \left|g_2(x)\right|\right)\right)$ *.*

Theorem 4. *Suppose that* $f_1(x)$ *is* $\Omega(g_1(x))$ *and* $f_2(x)$ *is* $\Omega(g_2(x))$*. Then* $(f_1 + f_2)(x)$ *is* $\Omega\left(\max\left(\left|g_1(x)\right|, \left|g_2(x)\right|\right)\right)$*.*

Now, suppose that $S(n)$ is the running time function for the Sol-van der Weide algorithm. We have the following time complexity analyses.

For the worst-case time complexity, we argue that both vectors S_T and V_T have lengths $n+1$ in the initialization subprocedure. Putting them together, they contribute $2n+2$ instructions in this subprocedure. Hence, by theorem (2),

$$\text{initialization subprocedure time complexity } = 2n + 2 \in O(n).$$

As for the recursion subprocedure, it involves three vectors, S_{T-ih}, V_{T-ih} and $(K - S_{T-ih})^+$ which contract by length one every iteration. This reduction in length can be expressed as

$$\begin{aligned} 3(n) + 3(n-1) + \ldots + 3(2) + 3(1) &= 3\left[\frac{n(n+1)}{2}\right] \\ &= \frac{3}{2}n^2 + \frac{3}{2}n. \end{aligned}$$

According to theorem (2), $\frac{3}{2}n^2 + \frac{3}{2}n \in O(n^2)$. Thus the worst-case time complexity of the entire algorithm according to theorem (3) is

$$\begin{aligned} S(n) &= \text{initialization subprocedure time complexity} \\ &\quad + \text{recursion subprocedure time complexity} \\ &= O(n) + O(n^2) \\ &= O(n^2). \end{aligned}$$

As for the best-case time complexity, we note that the initialization subprocedure has to process all the input elements and reducing its length will affect the accuracy of the computation. Hence, we conclude that the initialization subprocedure has a linear time complexity. That is

$$\text{initialization subprocedure time complexity} = \Omega(n)$$

For the recursion subprocedure, we conclude that it is also quadratic similar to its worst-case counterpart for the following reasons:

1.) the recursion is iterative and contains no conditional statement.

2.) the recursion is based on a closed form equation (11).

Therefore, by theorem (4), the best-case time complexity of the entire algorithm is

$$\begin{aligned} S(n) &= \text{initialization subprocedure time complexity} \\ &\quad + \text{recursion subprocedure time complexity} \\ &= \Omega(n) + \Omega(n^2) \\ &= \Omega(n^2). \end{aligned}$$

Since, $S(n) = O(n^2)$ and $S(n) = \Omega(n^2)$, we conclude that $S(n) = \Theta(n^2)$ by Theorem (1).

5 Results and Discussion

We implemented the two $m.files$ on a desktop computer with an installed random access memory (RAM) of 2.0 GB and Intel Core 2 Duo processors with speeds of 2.0 GHz and 1.99 GHz. We also included the empirical running times, t_1 and t_2, of each method. We listed the results in the table below.

As we can see from the computed values, both methods give exactly the same option value which is a good thing for SWA (Sol - van der Weide algorithm). The CRR method appears initially to be slower than the SWA but as m (height of the barrier) grows large, the computational time of CRR grows slowly compared to SWA. One good aspect of SWA is that it is more resilient than CRR because it was able to compute option prices way beyond the capabilities of CRR.

m	n	x	CRRprice	t_1	SWAprice	t_2
10	101	48	1.1063e-004	0.0643	1.1063e-004	0.0013
20	406	198	1.0993e-004	0.0037	1.0993e-004	0.0071
30	914	450	1.0999e-004	0.0162	1.0999e-004	0.0203
40	1626	804	1.1001e-004	0.0396	1.1001e-004	0.0494
50	2541	1260	1.1005e-004	0.0547	1.1005e-004	0.1252
100	10166	5062	NaN	0.4960	1.1003e-004	1.1668
200	40664	20291	NaN	5.7070	1.1003e-004	18.4292
300	91495	45687	NaN	159.5513	1.1003e-004	196.1419

6 Conclusions

This paper had two goals. Firstly, it aimed to apply the Sol - van der Weide algorithm to a knock-out put barrier option in to order verify its accuracy relative to the CRR model. This is because in Sol's paper [8] it was only used to price a knock-out call barrier option. As shown by our results, it computed exactly the same values as the CRR model. Not only that, the algorithm showed that it can compute prices for larger values of barrier height m. Secondly and the most significant advancement in this paper, we were able to analyze its time complexity. We have shown that its time complexity is $\Theta(n^2)$.

For further works, it would be interesting to study whether it is possible to optimize the algorithm and determine its memory complexity. Obviously, we would like to apply the Sol - van der Weide algorithm to trinomial trees. This is a logical step since trinomial trees are more general compared to binomial trees.

References

[1] Andell, J. *Mathematics of Chance.* John Wiley and Sons, Inc., 2001.

[2] Black, F. and Scholes, M. The Pricing of Options and Corporate Liabilities. *Journal of Political Economy*, 1973.

[3] Boyle, P. and Lau, S. Bumping Up Against the Barrier with the Binomial Method. *Journal of Derivatives*, 1994.

[4] Cormen, T.H., Leiserson, C.E., Rivest, R., Stein, C. *Introduction to Algorithms, 3rd Ed.* The MIT Press, 2009.

[5] Costabile, M. *A Combinatorial Approach for Pricing Parisian Options (Decisions in Economics and Finance).* Springer-Verlag, 2002.

[6] Hull, J.C. *Options, Futures, and other Derivatives, 4th Ed.* Prentice Hall, 2000.

[7] Rosen, K.H. *Discrete Mathematics and Its Applications, 6th Ed.* McGraw-Hill, 2008.

[8] Tina Sol. Pricing Barrier Options in Discrete Time. Bachelor Thesis, Technische Universiteit Delft, 2009.

Appendices

A UOP CRR Matlab Code

```
function UOPCRRprice(S, K, B, T, m, r, gamma, sigma)
% INPUT:
% m=minimum number of up steps in order to hit the barrier
% r =  interest rate
% gamma= dividend yield
% sigma = volatility
% S=initial asset price
% K=strike price
% B=height of barrier
```

```
% T=lifetime
% OUTPUT:  V_0 = premium price
n=floor(T*(m*sigma/log(B/S))^2) % number of time steps
                                              %to be in the money
h=T/n;                          % size of each step
u=exp(sigma*sqrt(h));           % up factor
d=1/u;                          % down factor
p=(exp((r-gamma)*h)-d)/(u-d);   % probability of up-step
q=1-p;                          % probability of down-step
%tic;
%Construct vectors with final asset prices and option pay-offs
Sf=S*u.^(n:-2:-n);
Vf1=(K-Sf).*((Sf<B).*(Sf<K));
%Calculate the max. no of up steps to be in the money at maturity
x=max(floor(log(K/(S*d^n))/log(u/d)),0);
%Construct vectors with probabilities for asset price to end
%at points that are in the money and below the barrier
Pf=zeros(n+1,1);
for k=0:floor(0.5*n)
    N=(n-k+1):1:n;
    D=1:1:k;
    Pf(k+1)=prod(p*q*N./D)*q^(n-2*k);
end
for k=ceil(0.5*n):floor(0.5*(n+m))
    N=(k+1):1:n;
    D=1:1:n-k;
    Pf(k+1)=prod(p*q*N./D)*p^(2*k-n);
end
Pf=flipud(Pf);
%Construct a 'correction' vector that eliminates value from
% expired paths
Cf=zeros(n+1,1);
for j=n-floor(0.5*(n+m)):n
    N=n-j-m+1:1:n-j;
    D=j+1:1:j+m;
    Cf(j+1)=1-prod(N./D);
    XX2(j+1)=Cf(j+1);
end
```

```
%Multiply the 3 vectors pointwise, sum and discount to get V_0
Vfe=exp(-r*T)*sum(Vf1'.*Pf.*Cf)
%t2=toc
```

B UOP SWA Matlab Code

```
function UOPSWAprice(S, K, B, T, m, r, gamma, sigma)
% INPUT:
% m=minimum number of up steps in order to hit the barrier
% r = interest rate
% gamma= dividend yield
% sigma = volatility
% S=initial asset price
% K=strike price
% B=height of barrier
% T=lifetime
% OUTPUT: V_0 = premium price
n=floor(T*(m*sigma/log(B/S))^2) % number of time steps to be
                                                    %in the money
h=T/n;                          % size of each step
u=exp(sigma*sqrt(h));           % up factor
d=1/u;                          % down factor
p=(exp((r-gamma)*h)-d)/(u-d);   % probability of up-step
q=1-p;                          % probability of down-step

%tic;
%Construct vectors with final asset prices and option pay-offs
Sf=S*u.^(n:-2:-n);
Vf2=(K-Sf).*((Sf<B).*(Sf<K));
%Calculate backwards to find V_0
for i=1:1:n
    Sf=d*Sf(1:end-1);
    Vf2=exp(-r*h)*(p*Vf2(1:end-1)+q*Vf2(2:end)).*(Sf<B);
end
Vf2
%t2=toc
```