Calcolo manuscript No.
(will be inserted by the editor)

Enhanced Tilley's Bundling Algorithm Using Memory Reduction Monte Carlo Method

Raymond H. Chan[1] ⋆, **Ka-Chun Ma**[2], **Chi-Yan Wong**[3]

[1] (rchan@math.cuhk.edu.hk) Department of Mathematics, The Chinese University of Hong Kong, Shatin, Hong Kong.
[2] (kcma@math.cuhk.edu.hk) Department of Mathematics, The Chinese University of Hong Kong, Shatin, Hong Kong.
[3] (cywong@math.cuhk.edu.hk) Department of Mathematics, The Chinese University of Hong Kong, Shatin, Hong Kong.

Received: / Revised version:

Abstract. When pricing American-style options by Tilley's bundling algorithm, one has to store the simulated asset prices at all time steps on all paths in order to determine when to exercise the options. If N time steps and M paths are used, then the storage requirement is $M \cdot N$. In this paper, we improve the Tilley's bundling algorithm [6] by applying our backward-path method, which requires only $O(M)$ storage. The only additional computational cost is that we have to generate each random number twice instead of once. For machines with limited memory, we can now use larger values of M and N to improve the accuracy in pricing the options.

1. Introduction

Monte Carlo method is one of the main methods for computing American-style options. The apparent difficulties in using Monte Carlo methods to price American-style options come from the backward nature of the early exercise feature. There is no way of knowing whether early exercise is optimal when a particular asset price is reached at a given time. There are different optimal stopping rules proposed to tackle the above difficulties, see for instance [6,1,5]. But, all these stopping rules are computationally inefficient because they have to save every intermediate asset price along every path.

One can look at this problem from another point of view. In usual Monte Carlo method, the simulated paths are all generated in the time-increasing direction, i.e. they start from the initial asset price S_0 and march to the expiry date according to a given geometric Brownian motion. But since the pricing of American options is a backward process starting from the expiry date back to S_0, these simulation algorithms require the storage of all asset prices at all simulation times for all simulated paths. Thus the total storage requirement grows like MN where M is the number of simulated paths and N is the number of time steps. The accuracy of the simulation is hence severely limited by the storage requirement.

In [2], we proposed a backward-path method for computing American-style options that does not require storing of all the intermediate asset prices. Our method can reduce the memory requirement of the least-squares approach [5] from MN to $O(M)$. Our main idea is to generate the paths twice:

⋆ The research was partially supported by the Hong Kong Research Grant Council grant CUHK4243/01P and CUHK DAG 2060220.

one in a forward sweep to establish the asset prices at the expiry date, and one in a backward sweep that computes the intermediate asset prices only when they are needed. The only additional cost in our method is that we have to generate each random number twice instead of once. The resulting computational cost is less than twice of that of those methods where all the intermediate asset prices are stored. In this paper, we apply our backward-path method in [2] to improve Tilley's bundling algorithm [6]. The storage requirement of the Tilley's bundling algorithm will be reduced from $MN + 3M + O(1)$ to $4M + O(1)$ only, so that we can improve the accuracy in pricing the options by using larger values of M and N.

The remainder of this paper is organized as follows. Section 2 recalls our backward-path method in [2]. In Section 3, we show how to use our method to compute American options by adopting it to the bundling algorithm proposed by Tilley [6]. Section 4 gives some numerical results to illustrate the effectiveness of our method.

2. The Backward-Path Method

In this section, we recall our backward-path method in [2], which we do not need to store the intermediate asset prices when computing the option prices. As usual, we let the asset price S follow a geometric Brownian motion

$$\frac{dS}{S} = rdt + \sigma dX, \tag{1}$$

where r is the risk-free interest rate, σ is the volatility, and dX is a standard Brownian motion. In the Monte Carlo simulation, we divide the time horizon into N time steps with each having the length

$$\Delta t = \frac{T - t_0}{N},$$

where t_0 is the current time and T is the expiry date of the option. Thus the time horizon is discretized as $t_0 < t_1 < \cdots < t_N = T$ where $t_j = t_0 + j\Delta t$.

Let the asset price at time t_0 be S_0. Given S_0, if we are to simulate M paths, then the i-th path can be defined by the recurrence:

$$S_j^i = S_{j-1}^i e^{(r-\frac{\sigma^2}{2})\Delta t + \sigma\sqrt{\Delta t}\varepsilon_j^i}, \quad 1 \le i \le M, \quad 1 \le j \le N, \tag{2}$$

where $S_j^i = S^i(t_j)$ is the asset price on the i-th path at t_j with $S_0^i = S_0$ for all i, and ε_j^i are independent, identically distributed, standard normal random numbers; see for instance [4, p.225].

The straightforward approach, like those in [6,5], is to simulate the paths and then store all intermediate asset prices S_j^i up for later computation of the option prices. We will refer this approach as the *full-storage method*. It requires the storage of $M \cdot N$ standard normal random numbers.

In our backward-path method, each random number $\{\epsilon_j^i\}$ is generated twice, but the intermediate asset prices S_j^i are generated once only as in the full-storage method. To see how this is done, we note from (2) that the intermediate asset prices are given by

$$S_j^i = S_0 e^{j(r-\frac{\sigma^2}{2})\Delta t + \sigma\sqrt{\Delta t}(\varepsilon_1^i + \varepsilon_2^i + \cdots + \varepsilon_j^i)}, \quad 1 \le i \le M, \quad 1 \le j \le N. \tag{3}$$

We generate the random numbers ε_j^i in the following manner. Given an arbitrary seed `d`:

$$\begin{aligned}\text{set seed } \mathtt{d} &\longrightarrow \varepsilon_1^1 \longrightarrow \varepsilon_1^2 \longrightarrow \ldots \longrightarrow \varepsilon_1^M \\ &\longrightarrow \varepsilon_2^1 \longrightarrow \varepsilon_2^2 \longrightarrow \ldots \longrightarrow \varepsilon_2^M \\ &\longrightarrow \ldots \\ &\longrightarrow \varepsilon_{N-1}^1 \longrightarrow \varepsilon_{N-1}^2 \longrightarrow \ldots \longrightarrow \varepsilon_{N-1}^M \\ &\longrightarrow \varepsilon_N^1 \longrightarrow \varepsilon_N^2 \longrightarrow \ldots \longrightarrow \varepsilon_N^M \end{aligned} \tag{4}$$

More precisely, we generate the random numbers on every path i, $1 \le i \le M$, for the time step $j = 1$ first. Then we generate them on all paths for $j = 2$ etc.

According to (3), in order to compute S_N^i, we only need the sum $\sum_{j=1}^N \varepsilon_j^i$. We assign a vector $\mathbf{s}$ to hold this. Thus once ε_j^i is generated, we add it to $\mathbf{s}(i)$:

$$\mathbf{s}(i) = \mathbf{s}(i) + \varepsilon_j^i, \quad 1 \le i \le M, \tag{5}$$

and ε_j^i can then be discarded. When all the random numbers are generated, we compute S_N^i on all paths by (3):

$$S_N^i = S_0 e^{N(r-\frac{\sigma^2}{2})\Delta t + \sigma\sqrt{\Delta t}\mathbf{s}(i)}, \quad 1 \le i \le M. \tag{6}$$

After S_N^i are computed, we define the ith path, $1 \le i \le M$, to be:

$$\begin{aligned} S_N^i &= S_0 e^{N(r-\frac{1}{2}\sigma^2)\Delta t + \sigma\sqrt{\Delta t}(\varepsilon_N^i + \varepsilon_{N-1}^i + \cdots + \varepsilon_1^i)}, \\ &\vdots \quad \vdots \\ S_j^i &= S_0 e^{j(r-\frac{1}{2}\sigma^2)\Delta t + \sigma\sqrt{\Delta t}(\varepsilon_N^i + \varepsilon_{N-1}^i + \cdots + \varepsilon_{N-j+1}^i)}, \\ &\vdots \quad \vdots \\ S_1^i &= S_0 e^{(r-\frac{1}{2}\sigma^2)\Delta t + \sigma\sqrt{\Delta t}\varepsilon_N^i}. \end{aligned} \tag{7}$$

Comparing (3) with (7), and noting that in both equations, ε_j^i are independent identically distributed standard normal random numbers, we see that the sequence $S_0, S_1^i, \ldots, S_N^i$ defined in (7) represents a path that follows the geometric Brownian motion (1) starting from S_0. We will call the paths in (7) the *backward paths* as they are generated in the time-decreasing direction. The paths defined in (3) will be called the *forward paths*. We emphasize that the option prices obtained by the full-storage and the backward-path method should be statistically the same, since both the forward paths and the backward paths satisfy the same geometric Brownian motion (1). In Figure 1, we depict ten simulated forward paths and their corresponding backward paths.

When computing the option price, we have to move backward in time. From (7), we have

$$S_{j-1}^i = S_j^i e^{-(r-\frac{\sigma^2}{2})\Delta t - \sigma\sqrt{\Delta t}\varepsilon_{N-j+1}^i}, \qquad 1 \le i \le M, \quad 1 \le j \le N. \tag{8}$$

Thus given $\{S_j^i\}_{i=1}^M$, to obtain $\{S_{j-1}^i\}_{i=1}^M$, we only need the random numbers $\{\varepsilon_{N-j+1}^i\}_{i=1}^M$. Since we are moving backward in time, j here starts from N and decreases back to 1. In particular, given $\{S_T^i\}_{i=1}^M$, to obtain $\{S_{T-1}^i\}_{i=1}^M$, we only need the random numbers $\varepsilon_1^1, \varepsilon_1^2, \cdots, \varepsilon_1^M$, and we can generate them by resetting the seed to `d`. Therefore we need to generate the numbers $\{\varepsilon_{N-j+1}^i\}_{i=1}^M$ sequentially as in (4), and we can generate them by using the given seed `d` again. Repeating the idea, we can generate the asset prices $\{S_j^i\}_{i=1}^M$ for all time steps j in the backward manner, and at each time step, we require only one M-vector $\mathbf{s}$ for storing $\{S_j^i\}_{i=1}^M$ for that time step.

We end this section by pointing out that our backward-path method can be applied to most pseudo random number generators that can generate a unique sequence of random numbers for a fixed seed.

Fig. 1. Simulations of forward paths (left) and backward paths (right).

3. The Bundling Algorithm

Our path generating technique can reduce the memory requirement of Tilley's bundling algorithm for pricing American-style options [6]. In this section, we recall the bundling algorithm and illustrate our method by pricing an American put options.

At the final exercise date, the optimal exercise strategy for an American option is to exercise the option if it is in the money. Prior to the final date, however, the optimal strategy is to compare the immediate exercise value with the expected cash flow from continuing, and then exercise if immediate exercise is more valuable. Thus the key to optimally exercising an American option is to identify the early exercise boundary. In [6], the early exercise boundary between holding and immediate exercising is determined by finding the leading 1 of the first string of 1's such that its length exceeds the length of every subsequent string of 0's in the reordered sequence of the simulated paths at each time step. And hence, one can estimate an optimal stopping rule for the option.

According to the stopping rule, the cash flows generated by the option at each exercise time are identified. The option price is then calculated by discounting all these cash flows back to time t_0 at the risk-free rate r. However, as in other Monte Carlo methods for computing American options, the method has to save all the intermediate asset prices for the computation of the option price. It therefore requires a storage of size MN, where M is the number of stimulated paths, and N is the number of time steps. Here we apply our method in §2 to reduce the storage requirement to $O(M)$.

We now illustrate our method by a numerical example. Consider an American put option on a non-dividend paying stock with strike price E equals to \$10. The current stock price S_0 is \$6, the risk-free rate r is 0.1, the volatility σ is 0.4, and the time to maturity T is 0.5 years. We assume that the option is exercisable at time $j = 0$, 1, 2, 3, 4 and 5 (i.e. $N = 5$). We illustrate the algorithm by using 12 simulated paths, i.e. $M = 12$.

Algorithm

1. Initialization:
   (a) Set the seed to `d` which is chosen arbitrarily. Using (4)–(6), compute S_5^i and save it in $\mathbf{s}(i)$ for each path i. Then reset the seed to `d`.
   (b) For each path i, compute the present value of the payoff at the expiry date T, which is equal to $e^{-5r\Delta t}\max\{E - S_5^i, 0\}$, and save them in a vector $\mathbf{p}$. The value is the cash flow realized by the option holder conditional on not exercising the option before the expiry date $j = 5$.

Path i	S_5^i	$\mathbf{p}(i) = e^{-5r\Delta t}\max\{E - S_5^i, 0\}$
1	6.158435	3.654209
2	6.764154	3.078031
3	5.359064	4.414594
4	3.506414	6.176889
5	6.744713	3.096524
6	8.726923	1.210988
7	4.175771	5.540177
8	7.026605	2.828380
9	8.236295	1.677687
10	8.175020	1.735974
11	5.143846	4.619315
12	7.641987	2.243010

2. Backward time-marching to $j = 4$:
   (a) Use (4) to generate the random numbers $\{\varepsilon_1^i\}_{i=1}^M$. Then compute S_4^i for all paths by (8) and store them in $\mathbf{s}(i)$, i.e. the memory location of S_5^i will be overwritten by S_4^i.
   (b) Reorder the simulated paths in the descending order (or ascending order for the call option). Reindex the paths, denoted by k, according to the reordering and store the corresponding location in $\mathbf{w}$. Then, group them into Q bundles. In this example, we choose $Q = 3$ and each bundle has 4 paths.

Bundle q	Path k	$\mathbf{w}(k) = i$	S_4^k	$\mathbf{p}(k)$
1	1	2	9.834040	3.078031
	2	9	8.746178	1.677687
	3	10	7.936682	1.735974
	4	6	7.610601	1.210988
2	5	12	6.683837	2.243010
	6	5	6.418780	3.096524
	7	11	6.223718	4.619315
	8	3	5.952373	4.414594
3	9	8	5.895387	2.828380
	10	1	5.023360	3.654209
	11	7	3.909715	5.540177
	12	4	3.277212	6.176889

   (c) For each bundle, we estimate the expected payoff from continuing by averaging $\mathbf{p}(k)$ over all paths in the same bundle q for $q = 1, 2, 3$, i.e.

$$h(k) = \frac{1}{4} \sum_{\substack{\text{all } l \\ \text{in bundle} \\ \text{containing } k}} \mathbf{p}(l) \tag{9}$$

Then, compare them with the present value of immediate exercising to see if we should "tentatively" exercise (see the table below). Then we define an indicator variable $\mathbf{c}$ by:

$$\mathbf{c}(k) = \begin{cases} 1, & \text{if } e^{-4r\Delta t}\max\{E - S_4^k, 0\} \geq h(k), \\ 0, & \text{otherwise.} \end{cases} \tag{10}$$

Bundle q	Path k	S_4^k	Exercising $e^{-4r\Delta t}\max\{E-S_4^k,0\}$	Continuation $h(k)$	Indicator $\mathbf{c}$
1	1	9.834040	0.159452	1.925670	0
	2	8.746178	1.204658		0
	3	7.936682	1.982413		1
	4	7.610601	2.295708		1
2	5	6.683837	3.186133	3.593361	0
	6	6.418780	3.440797		0
	7	6.223718	3.628210		1
	8	5.952373	3.888916		1
3	9	5.895387	3.943668	4.549914	0
	10	5.023360	4.781502		1
	11	3.909715	5.851480		1
	12	3.277212	6.459182		1

(d) Examining the sequence $\mathbf{c}$, we determine a "sharp boundary" k_* between holding and immediate exercising by finding the leading 1 of the first string of 1's such that its length exceeds the length of every subsequent string of 0's. Here, the "sharp boundary" k_* is 7. Then, for those paths where we should exercise (i.e. for all $k \geq k_*$), we update the payoff vector $\mathbf{p}$ by the exercising value, and otherwise by expected payoff from continuing, i.e.

$$\mathbf{p}(k) = \begin{cases} e^{-4r\Delta t}\max\{E-S_4^k,0\}, & \text{for all } k \geq k_*, \\ h(k), & \text{otherwise.} \end{cases}$$

Path k	Should we exercise?	$\mathbf{p}(k)$
1	NO	1.925670
2	NO	1.925670
3	NO	1.925670
4	NO	1.925670
5	NO	3.593361
6	NO	3.593361
7	YES	3.628210
8	YES	3.888916
9	YES	3.943668
10	YES	4.781502
11	YES	5.851480
12	YES	6.459182

(e) Go back to Step 2(a) and backward time-marching to $j = 3$ etc.

In essence, given S_{j+1}^i and $\mathbf{p}$, the algorithm first computes S_j^i using (4) and (8). Then it sorts $\{S_j^i\}_{i=1}^M$ in the descending order (or ascending order for the call option) and groups them into Q bundles. For each bundle, it computes the expected payoff from continuing using (9), and then determine the indicator variable $\mathbf{c}$ by (10). With the sequence $\mathbf{c}$, it determines the "sharp boundary" k_* by finding the leading 1 of the first string of 1's such that its length exceeds the length of every subsequent string of 0's. Finally, it updates the corresponding entries of the payoff vector $\mathbf{p}$ by the exercising value in case we should exercise, and otherwise by the expected payoff from continuing.

To complete the example, we backward time-march to $j = 0$ and get

$$\mathbf{p} = [4.663828, 3.220023, 4.430495, 5.112066, 4.455488, 2.727368, \\ 4.956490, 4.088685, 3.166220, 3.575709, 5.440687, 4.640325]^T.$$

The option can now be valued by averaging all the entries of $\mathbf{p}$ at time $j = 0$. For this example, it will be \$4.2064.

From the example, we see that for the backward-path method, we only need to store $\mathbf{s}$, $\mathbf{p}$, $\mathbf{w}$ and $\mathbf{c}$. We summarize the memory requirement in Table 1. For completeness, we also list in the table the additional computational cost required by our method as compared to the full-storage method. From the algorithm, we notice that the additional costs are precisely the costs of generating the paths in the backward sweep. We will see in the numerical example in §4 that the total cost of our method is less than twice of that of the full-storage method.

4. Numerical Examples

In this section, we test our method on an example given in [7, p.176]. It is an American put option with strike price E equals to \$10, the riskless rate r is 0.1, the volatility σ is 0.4, and the expiry date T is 0.5 years. We emphasize that the option prices obtained by the full-storage and the backward-path method should be statistically the same, since both the forward paths and the backward paths satisfy the same geometric Brownian motion (1). This is verified by our numerical results below. In our experiment, all our computations were done by MATLAB on an Intel Pentium 800 MHz processor with 512 Megabyte RAM. Again we use M and N to denote the number of paths and the number of time steps respectively.

Tables 2 and 3 show the effect on the errors by increasing M from $5,040$ to $504,000$ while N is fixed. In the tables, the data under the column "CNM" are results computed by the Crank-Nicolson method and are given in [7, p.176]. The results under the "Mean" and "STD" are the means and standard deviations obtained after 10 trials. The final column "Error" is the difference between "CNM" and the "Mean". From the tables, we see that the error decreases by one decimal point when M is increased 100 times. This is consistent with the error estimate $O(1/\sqrt{M})$ of the Monte Carlo method, see for instance [3].

Table 4 gives the CPU time for one run of the methods when $S_0 = 10$. The symbol "†" signifies that memory is not enough to run the problem of that size. From the table, we observe that the time increases roughly like linear with respect to M and N. Moreover, as expected, the time required by our backward-path method is always less than twice of that of the full-storage method.

References

1. Barraquand J. and Martineau D., *Numerical Valuation of High Dimensional Multivariate American Securities*, J. Financ. Quant. Anal, 30 (1995), 383–405.
2. Chan R., Chen Y., and Yeung K., *A Memory Reduction Method in Pricing American Options*, J. Statist. Comput. Simulation, 74 (2004), 501–511.
3. Galanti S. and Jung A., *Low-Discrepancy Sequences: Monte Carlo Simulation of Option Prices*, J. Dervi., 5 (1997), 63–83.
4. Kwok Y., *Mathematical Models of Financial Derivatives*, Springer-Verlag, 1998.
5. Longstaff F. and Schwartz E., *Valuing American Options by Simulation: A Simple Least-Squares Approach*, Rev. Financ. Stud., 14 (2001), 113–147.
6. Tilley J., *Valuing American Options in a Path-Simulation Model*, Trans. Soc. Actuaries, 45 (1993), 563–577.
7. Wilmott P., Howison S., and Dewynne J., *The Mathematics of Financial Derivatives: A Student Introduction*, Cambridge University Press, 1995.

Method	Memory	Additional Computational Cost
Full-Storage	$MN + 3M + O(1)$	—
Backward-Path	$4M + O(1)$	MN additions to find $\mathbf{s}$ in (5) MN calls to generate $\{\varepsilon_j^i\}$ in (4)

Table 1. Memory and Cost Comparison.

S_0	CNM	Mean	STD	Error
2	8.0000	8.0000	0.0000	0.0000
4	6.0000	6.0000	0.0000	0.0000
6	4.0000	4.0000	0.0000	0.0000
8	2.0951	2.0847	0.0042	−0.0104
10	0.9211	0.9094	0.0042	−0.0117
12	0.3622	0.3769	0.0030	0.0147
14	0.1320	0.1486	0.0019	0.0166
16	0.0460	0.0572	0.0006	0.0112

Table 2. Backward-Path Method with $M = 5,040$ and $N = 10$.

S_0	CNM	Mean	STD	Error
2	8.0000	8.0000	0.0000	0.0000
4	6.0000	6.0000	0.0000	0.0000
6	4.0000	4.0000	0.0000	0.0000
8	2.0951	2.0858	0.0003	−0.0093
10	0.9211	0.9168	0.0004	−0.0043
12	0.3622	0.3612	0.0002	−0.0010
14	0.1320	0.1315	0.0002	−0.0005
16	0.0460	0.0463	0.0001	0.0003

Table 3. Backward-Path Method with $M = 504,000$ and $N = 10$.

M	$504,000$				10^4	$4 \cdot 10^4$	$25 \cdot 10^4$	10^6
N	10	20	50	100	100			
Full-storage	93.555	196.793	504.896	1025.234	19.258	78.693	502.662	†
Backward-path	94.486	198.625	511.886	1035.769	19.538	79.795	509.432	2065.550

Table 4. CPU time in seconds for different methods.