National Taiwan University
Department of Computer Science and Information Engineering

Computer Organization and Assembly Languages 2006 Final Project

MineSwee++ 2 – C[o]re

B94902075 呂敏中

2007.01

In this project, a standalone version of Minesweeper Flags game with the computer as the player's opponent is made.

Background

- **Minesweeper** [1]

Minesweeper has been a famous game since it appeared in Windows 3.1 over ten years ago. The object of the original Minesweeper is to clear a minefield (a ***board*** henceforth) without denoting a mine. The board contains ***grids*** which can be ***opened*** by clicking on it. If a grid containing a mine is opened, the game is over. If a grid opened does not contain a mine, then a number appears on the grid indicating the number of adjacent grids containing mines. When playing Minesweeper, the best strategy is to open the grid with the least probability to contain a mine.

Screenshot of Minesweeper

- **Minesweeper Flags** [2]

Minesweeper Flags was introduced in 2003 by Microsoft along with its famous instant messaging software, MSN Messenger (now Windows Live Messenger). Minesweeper Flags consists of two players; in a twist on the original game, players must now locate mines (and ***flag*** them), and whoever has flagged the most by the end wins. The board is 16 rows by 16 columns and has 51 mines randomly spread; that is, a player wins if he flags 26 mines. In one turn, a player can open a grid. If the grid contains a mine, then it is flagged; the player can open another grid. If the grid does not contain a mine, then the number as described above shows, and it is the opponent's turn to open a grid. When playing Minesweeper Flags, the best strategy is to open the grid with the best probability to contain a mine.

Screenshot of Minesweeper Flags

- **MineSwee++**

MineSwee++ was developed by me in 2004 [L2], which is essentially the standalone version of Minesweeper Flags. However, in MineSwee++, it is the computer that simulates the player's opponent. The original MineSwee++ has been considered to have very poor strategies, and therefore in this project, I try to implement a different algorithm to make it smarter. This new version is called MineSwee++ 2, and I name the new algorithm/implementation "C[o]re".

Solving Mine Boards

It is obvious that if we know the mine pattern given an incomplete board, we can easily beat Minesweeper and its variants. Finding out the pattern is called "solving mine boards". It is interesting to ask whether an efficient algorithm exists to solve any given mine board. Unfortunately, the answer is - a resounding "No". Determining the pattern of mines for a given board is known to be NP-Complete [3][L1]. There are many Minesweeper solvers on the internet,

but none guarantees to be accurate under every circumstance for a given board (in reality, solving boards is what many graduate students of AI field choose for their research). Obviously, brute-forcefully enumerating every possible mine board for checking is infeasible, since there are $\binom{256}{51} \approx 2.0 \times 10^{54}$ possible mine boards; it is practically impossible to calculate or store such a large number of boards with modern CPUs or storage media.

Also, there is even more to take into consideration when playing Minesweeper Flags than the original Minesweeper. For example, when playing Minesweeper Flags, if the mine-containing probability of the candidate grids to open is too little, it is wise to open a grid which should give your opponent the least information about the mines, instead of the grid with the best probability of containing a mine, due to the rule that if the player is unsuccessful in flagging a mine, the opponent will take the next turn. Minesweeper Flags also provides a Bomb for the player falling behind, which can be used only one time in the entire game: the bomb opens 5×5 grids at the same time. C[o]re will not take into consideration these two scenarios, as it will only open the grid with the best probability to contain a mine.

Why and Where to Use Assembly?

Since determining the mine pattern is not practical, the solvers available on the net use different techniques. Some have small-area brute-force enumeration (what MineSwee++ uses), some use combinatorial mathematics, and others implement sophisticated AI. Most of them have the same goal – to calculate the probability of a grid containing a mine as accurately as possible.

In the implementation of C[o]re, a different approach is used: the computer randomly generates mine boards, examines them with the known conditions, filters out those boards not matching the conditions, and calculates the probability of each grid using all the generated boards that match the conditions. The pseudo-code of the implementation looks like this:

```
GetGridWithBestProbability(board) returns (r, c, p)
board: the board being played
r: the row index of the grid with the best probability to contain a mine
c: the column index of the grid with the best probability to contain a mine
p: the probability the grid has to contain a mine
begin
     initialize board_sum with all 0
     num_mines ← Number of mines flagged in board
     num_mines_left ← 51 - num_mines
     num_matched_board ← 0
     for i ← 1 to ITERATIONS_PER_MOVE do
     begin
          board2 ← generate a board with num_mines_left mines spread randomly in suitable
                   places
          if (mines of board2) + (mines of board) matches conditions of board then
          begin
               add board2 to board_sum
               num_matched_board ← num_matched_board + 1
```

```
          end
     end
     if num_matched_board = 0 then
     begin
          return (0, 0, -1)
     end
     max_grid_r ← 0
     max_grid_c ← 0
     max_grid_p ← 0
     for r ← 1 to 16 do
     begin
          for c ← 1 to 16 do
          begin
               if board_sum[r][c] ≥ max_grid_p then
               begin
                    max_grid_r ← r
                    max_grid_c ← c
                    max_grid_p ← board_sum[r][c]
               end
          end
     end
     max_grid_p ← max_grid_p / num_matched_board
     return (max_grid_r, max_grid_c, max_grid_p)
end
```

The key to producing accurate probability is to have as many random board generations as possible. The number of generated boards in a short, human-acceptable period of time is directly related to how fast generating random boards, examining condition-matching, and storing possible boards for probability calculation are. This is where Assembly comes in to play a part. Since Assembly programming makes it possible to improve the speed of a program, using it to generate random boards, examine them for condition-matching and store possible boards may speed up the process and thus result in more accuracy in probability calculation.

Implementation

In this project, due to time constraint, the original GUI of MineSwee++ written in VB is used. The rest of the game, namely C[o]re, is coded in C++ (so as to provide a DLL for GUI to call) and Assembly, which takes the responsibility only for time-critical routines, i.e. random board generation, condition examination and board storage, for the board passed from C++ codes.

There is something related to probability calculation that Assembly does not do: If a grid has obviously 100% probability to contain a mine (determinable with a small constant-factor number of grid scans), then the C++ codes will find it, and not call the Assembly routine.

C[o]re supports multi-threading; the user can choose to run it in Single-Thread or Dual-Thread mode. If the user has a CPU that supports Intel Hyper-Threading or is multi-core (which is gaining prevalence), he may benefit from Dual-Thread mode. The codes of C[o]re are not specially optimized for Dual-Thread mode, but performance gain under that mode is still observed.

In this report, I do not focus on the VB/C++ implementation and the detailed, line-by-line code

implementation of the described algorithm in Assembly. In the following section, some notable pieces of Assembly implementation worth mentioning are introduced.

Implementation Directly Related to Assembly

- **Random Number Generation**

The first challenge encountered when implementing the pseudo-code is how to generate boards as fast as possible. The speed of board generation relates to that of random number generation, which in turn needs to be taken care of. Although there is `rand()` for use in Standard C Library, calling an external procedure from Assembly is relatively expensive and creates unresolvable bottleneck for speeding-up. Thus, an Assembly version of `rand()` must be remade.

The `rand()` function source code from Microsoft for Visual C++'s use is not suitable because it requires CRT (C Runtime) threading information. Consequently, other (pseudo-)random number generation algorithm is sought. What is needed is an algorithm that is easy to implement and fast to run; since it is only used in a game, it does not need to be extremely secure nor of high quality. Therefore, the Linear Congruential Generator is chosen [4]. The Assembly implementation of this algorithm can generate 2^{31} distinct pseudo-random numbers in about 10 seconds on Intel Core 2 Duo E6600[†].

Additionally, the seed of the generation is not taken from the return value of the traditionally-used `time()` function. Instead, the low-order 32-bits (`EAX`) of the return value of `rdtsc` instruction are used, which provide equal or even better randomness and avoid another external procedure call.

- **SSE**

A mine board is 16 rows by 16 columns. That is, when storing the boards that match the known conditions, 256 additions are done for one board. Instead of doing 256 separate scalar additions, SSE is used to perform parallel additions at the same time, effectively reducing the time required to store boards. Empirically, single-precision floating-points are sufficient to serve as *`board_sum`* in the pseudo-code, so `XXXps` instructions are used and thus four additions can be done at the same time.

When manipulating data, the faster `movaps` (compared to `movups`) is favored and `addps` requires the memory operand to be aligned on a 16-byte boundary, so `ALIGN 16` and `__declspec(align(16))` are added where needed, and the stack pointer (`ESP`) is taken extra care of to be aligned.

SSE is also used to reset board-storing variables and other auxiliary arrays to all zeros. This is done by doing `xorps xmm7, xmm7` and filling the destination array iteratively with `xmm7`.

Overall, for adding two 256KB arrays, SSE takes only 40% clock cycles that the standard `mov` and `add` take; for resetting a 512KB array to zero, SSE takes 50% clock cycles that the standard `mov` takes (both numbers are obtained on Intel Core 2 Duo E6600)[‡].

[†] See kernels\rand_new.cpp

[‡] See kernels\add_new.cpp and kernels\reset_new.cpp

These two Assembly implementations suggest that MineSwee++ 2 requires at least an Intel Pentium III CPU or an AMD Athlon XP CPU to run. Also, the Assembly is loop-unrolled wherever it is beneficial. Since the board size is fixed at 16 rows by 16 columns, a lot of codes can be optimized by loop-unrolling while not overflowing the instruction cache.

Results and Follow-ups

Now let's see how really fast the board generation is when implemented in Assembly, compared to when in C++. Here is the result on some machines: the figures read how many board-generation iterations the machine can do in 5 seconds [†‡].

CPU (ordered by release date)		C++ implementation (/O2 Optimization)	Assembly implementation	
			Iterations	Improvement
Intel Pentium 4 1.6 GHz (Willamette)		0.95×10^6	**1.0×10^6**	**5%**
Intel Pentium 4 2.8 GHz (Northwood)		1.65×10^6	**1.75×10^6**	**6%**
AMD Athlon XP 2500+ o/c 2.32 GHz		2.25×10^6	**2.5×10^6**	**11%**
AMD Athlon 64 3000+ o/c 2.4 GHz (Venice)		2.35×10^6	**2.7×10^6**	**15%**
AMD Turion 64 X2 TL-52 1.6 GHz (dual-core)	Single-Thread	1.6×10^6	**1.8×10^6**	**13%**
	Dual-Thread	2.0×10^6	**2.2×10^6**	**10%**
Intel Core 2 Duo E6600 2.4 GHz (dual-core)	Single-Thread	2.9×10^6	**3.25×10^6**	**12%**
	Dual-Thread	3.1×10^6	**3.25×10^6**	**5%**
			Average Improvement	**9.625%**
C++ Compiler: Microsoft ® 32-bit C/C++ Optimizing Compiler Version 14.00.50727.42 for 80x86 Assembler: Microsoft ® Macro Assembler Version 8.00.50727.42 Linker: Microsoft ® Incremental Linker Version 8.00.50727.42				

The Assembly implementation has improved the efficiency of board generation, in terms of the number of iterations within a fixed period of time, by about 10%. Does this mean that C[o]re is smart and "playable"? Yes! Empirically, for any given board, a couple million generation iterations are needed to produce good results. Therefore, C[o]re does play well with accurate probability calculation with most recent CPUs, thanks to the performance boost brought by Assembly.

There is only one scenario where C[o]re may lose ground: C[o]re opens a grid where the total number of mines contained by the neighboring grids is zero. When a grid as such is opened, the grids around it are automatically opened, according to the rule of MineSweeper Flags. This auto-opening process recurs if any of the newly-opened grids "has zero mines around"; then C[o]re's opponent (the player) takes the turn. As a result, the player can open several grids that definitely have mines (see Figure 1 for illustration). This gives the player a great chance to win over C[o]re. In the future, C[o]re may be improved by taking this scenario into consideration, and

† The board used as known conditions to examine with has all grids unopened except (9,9) marked "1".
‡ The process of the benchmarker (src\benchmark\MSPPTester.exe) is granted "High" priority when running.

then outplay most people due to its accuracy of probability calculation.

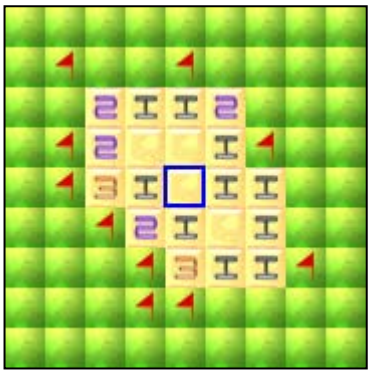

Figure 1

Acknowledgements

- Special thanks to my classmate 張顯之 (B94902061, hcsoso) for discussing related topics with me
- Special thanks to charlie_su1986 for generously providing the AMD Athlon 64 machine for benchmarking
- Special thanks to my classmate 程耀鋒 (B94902098, changefor) for generously providing the AMD Turion 64 X2 machine for benchmarking

References

[1] Wikipedia – Minesweeper (computer game)
http://en.wikipedia.org/wiki/Minesweeper_%28computer_game%29

[2] Wikipedia – Games and applications for Windows Live Messenger; 1.6 Minesweeper Flags
http://en.wikipedia.org/wiki/Games_and_applications_for_Windows_Live_Messenger#Minesweeper_Flags

[3] Kaye, R. Minesweeper is NP-Complete. *Mathematical Intelligencer* (Springer Verlag, New York) Volume 22 number 2 (Spring 2000), pp. 9-15

[4] Wikipedia – Linear congruential generator
http://en.wikipedia.org/wiki/Linear_congruential_generator

Links

[L1] Richard Kaye's Minesweeper Pages
http://for.mat.bham.ac.uk/R.W.Kaye/minesw/minesw.htm

[L2] # Middle Sea – MineSwee++
http://mnjul.net/mspp-intro.php
The final version of MineSwee++ 2 is scheduled to be released by this spring. Stay tuned if you are interested :)