VEXED PUZZLE SOLVER:

DEVELOPING A PROGRAM INCREMENTALLY*

Thomas Hruska
Senior, Artificial Intelligence Track
Department of Computer Science
Taylor University
(765)-998-4244
thruska@css.tayloru.edu

ABSTRACT

The game of Vexed offers students an interesting programming exercise for at any level of education. It also offers instructors a project with incremental development opportunities and a context for introducing a wide range of topics throughout the undergraduate program. Instructors introduce students to techniques and optimizations in each class through the motivating search for performance.

INTRODUCTION

This paper describes the results of my own research into writing a quick, efficient, and optimal puzzle solver for the game of Vexed. The research would have been far less interesting had a not-quite-optimal solution been acceptable. This project initially started off as a simple, brute-force depth-first search into the problem space and evolved into a massively parallel program using a hybrid breadth-first/depth-first search, tree pruning (e.g. memoization, illegal board elimination), data compression, I/O elimination, and hashing techniques. To date I have developed ten different programs to solve Vexed puzzles, each one improving on the one before and providing me with more knowledge about search spaces. This is an ongoing research effort into solving search space problems.

THE GAME OF VEXED

Vexed is one of the more popular games for the Palm OS series of handhelds [1]. The game consists of puzzle sets of 50-60 puzzle boards each. The boards range in difficulty from trivial to wicked. Difficulty is indicated by the board's "par" value. "Par" has a similar meaning to the golf term, that is, the developer determined the average number of moves a skilled player would require to solve a particular puzzle. The game itself is conceptually simple: eliminate all of the pieces on each Vexed board. The game has a Tetris feel to it. Pieces are moved left or right until "Gravity" causes them to fall through spaces below them. Pieces are eliminated when two or more pieces of the same type come in horizontal and/or vertical contact with each other [2]. An example board can be seen in Figure A.

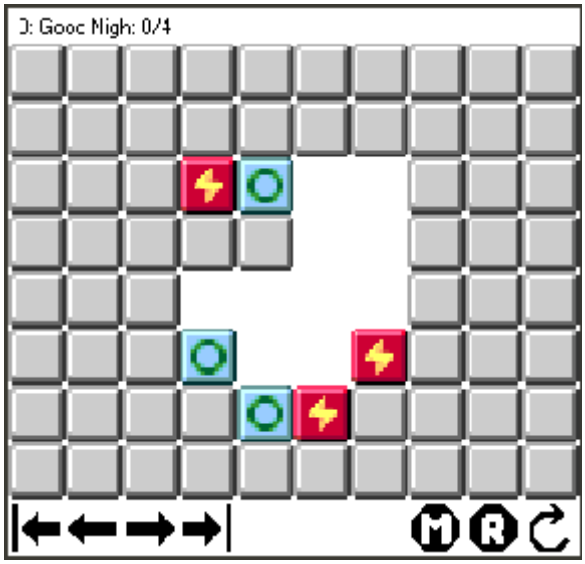

Figure A

Determining the optimal solution for the board in Figure A is straightforward. There can actually be multiple optimal solutions, but there is only one optimal solution that favors moving pieces top-to-bottom first, then pieces left-to-right, and finally moving directions left-to-right. Keeping this in mind, the optimal solution for this board is to move the top circle right one space, move the top lightning bolt right two spaces, and finally move the far left circle right one space (a total of four moves).

INITIAL STAGES

The following describes various approaches, insights, and suggestions to instructors for the potential integration of Vexed (or a similar search space problem) into the CS curriculum.

The game of Vexed can be introduced as early as CS1, allowing instructors in later classes to focus on particular design strategies and algorithms. At the CS1 level, the instructor for the class can focus on objectives such as loading boards into memory (1D and 2D arrays), displaying them to a console (or GUI-interface), and basic data manipulation. Working programs, such as the online game of Jexed (Java-based Vexed) can be used to demonstrate the game to CS1 students and build interest [3]. The final project could be an implementation of Vexed that allows players to specify moves.

Upon entering CS2, the student should learn about modular programming techniques. Modular programming can then be applied to Vexed by developing modules that apply "gravity," remove pieces, initialize, determine legal boards, move pieces, etc. The next objective is to get the program to solve the puzzle found in Figure A using a recursive depth-first search.

After that, a breadth-first version can be created. Note that these are brute-force searches that will only work on the simplest of puzzles.

CACHING MECHANISMS

Grasping the concepts behind artificial intelligence is difficult for some students. Instructors tend to assume that students instantly understand concepts such as memoization, minimax, and search spaces. I found that the actual, hands-on experience of developing programs for each of these concepts aided in solidifying my understanding of and ability to apply the concepts.

The incremental development of a Vexed Puzzle Solver offers many opportunities introduce artificial intelligence concepts. For instance, using a student's CS2 brute-force solver to solve Puzzle 2 (Figure B) from the Classic Puzzle Set (displaying each board that is being processed) will allow an entire class to see the first major optimization that can be made: memoization. Once the student has implemented memoization such that the puzzle solver can solve Puzzle 2 of the Classic Puzzle Set, the instructor can teach the students about limited size memoization tables by throwing Puzzle 9 (Figure C) from the Classic Puzzle Set into the mix. Puzzle 9 is a fairly complex puzzle that will run most memoization tables out of memory.

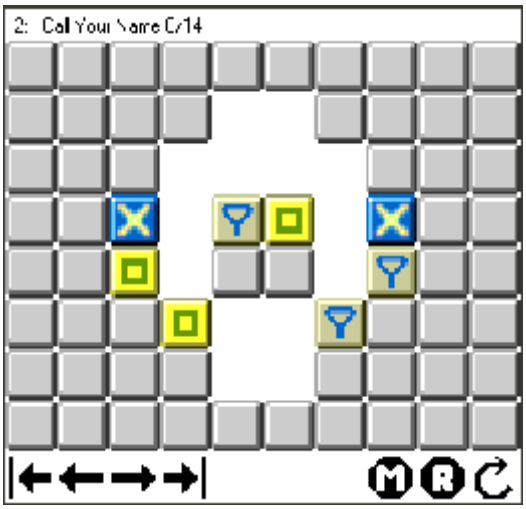

Figure B

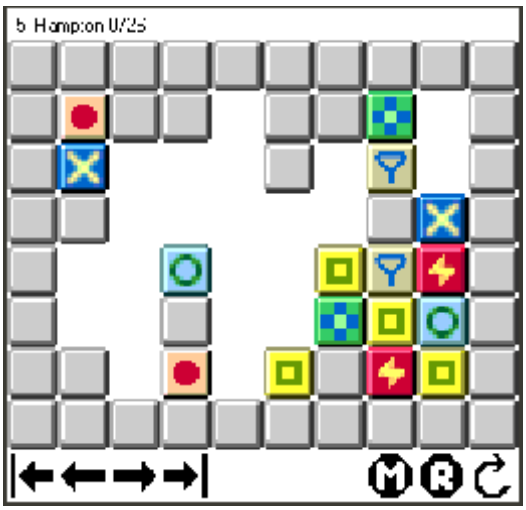

Figure C

But wait! There is more! Puzzle 9 is no ordinary puzzle. It demonstrates the size and complexity of search space problems. Once limited sized memoization tables have been added, the problem of storing the rapidly expanding number of puzzles in memory still exists. Since there are too many puzzles for a breadth-first search, the implementation must either use a depth-first solution or else the board states must be cached to the hard drive. Neither solution provides optimal performance, but caching leads to the best performance improvement. Combining the breadth-first search with caching to solve Puzzle 9 provided a solution within 1

minute, 30 seconds (on a dual Pentium III 500, 384MB RAM, 40GB 7200 RPM HD). In contrast, the depth-first approach without caching took more than 24 hours to complete.

In addition to these methods, there are many additional approaches to solving the puzzles, such as a hybrid breadth-first, depth-first search. Some students will want to continue on and perhaps a few will find their doctoral thesis in the ideas initiated with Vexed (a vexing dissertation, for sure!).

Finally, implementing the memoization cache using a hash table can dramatically increase the speed of the Vexed puzzle solver. Hashing is generally introduced in CS2, but good hashing algorithms such as CRC are not usually introduced until later courses such as networking/data communications.

SUB-OPTIMAL APPROACHES

While the previous examples are tied to finding optimal solutions, an artificial intelligence or algorithms class can go much further with Vexed. Since the problem space for many Vexed puzzles is so large (for example, Puzzle 6 of the Classic II Puzzle Set, see Figure D), it is often desirable that a sub-optimal solution be found within a short amount of time.

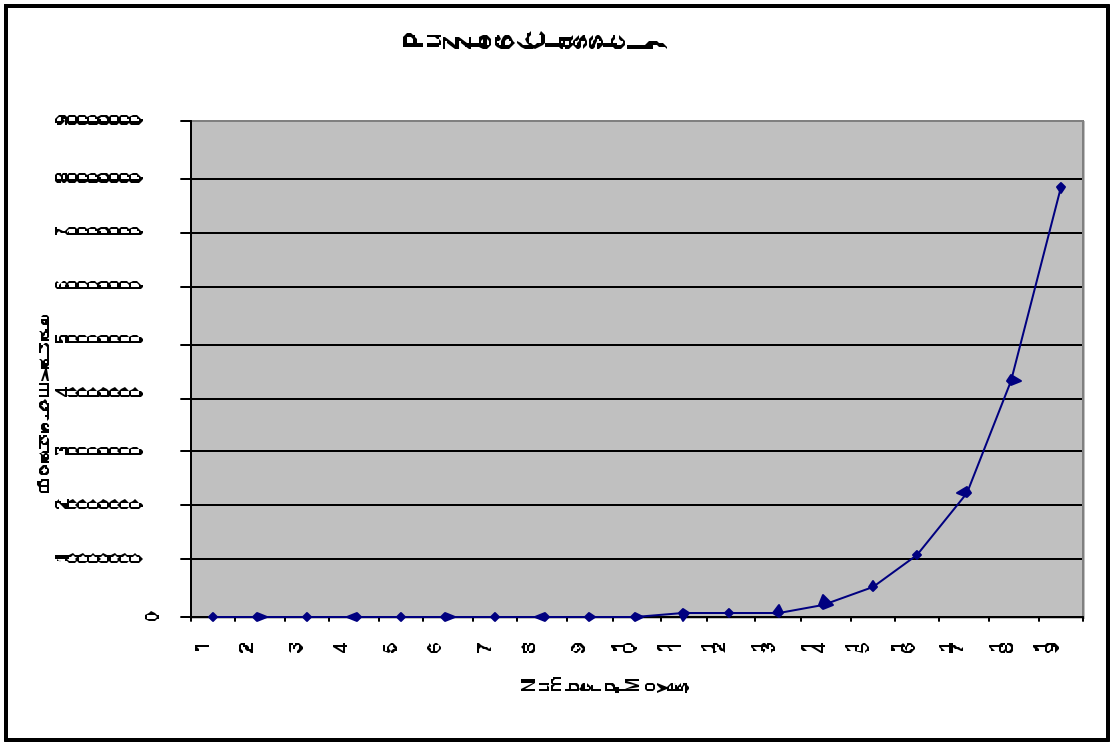

Figure D

Artificial Intelligence classes can have a big impact here. At this point, the student should already have a firm grasp on the ideas mentioned in the previous section. This will provide the firm foundation from which genetic algorithms, cost calculations, and other sub-optimal

approaches can be used to find solutions to a large search space problem such as Vexed in a reasonable amount of time. Typically, the traditional travelling salesman problem is used in an AI course, but Vexed offers a fresh look on the whole concept of search spaces and teaches the same principles.

OPTIMIZATIONS

Once the limited memoization tables and hard drive caches are implemented, the Vexed Puzzle Solver will be able to optimally solve most Vexed puzzles in a reasonable amount of time. Even then, several optimizations can be done to drastically improve the processing rate. These optimizations can be taught in a computer organization and architecture class that looks at how the various layers of an ISA (instruction set architecture) are implemented at the hardware level.

The first improvement that can be added to Vexed is to find out which functions are consuming the most CPU time and re-write them in pure assembler (assuming a language that is easily integrated with assembly code). Rewriting one specific function in assembly doubled the solver performance.

Another topic for a computer organization class is minimizing the amount of data moved across the bus to the hard drive. During processing of a Vexed board, millions of boards are cached to the hard drive and billions are moved across the memory bus. Therefore, minimizing I/O is key to maximizing performance. Modifying the Vexed state data structure to take up less space when storing data to the hard drive decreases storage space by 75% (250 bytes to 60 bytes per board), decreases I/O time significantly, and increases overall performance. Bit manipulation and compression techniques can also be applied; my typical 22GB+ hard drive cache on large board spaces was reduced by 85-95%, drastically cutting total hard drive I/O time. Compression packages such as zlib can be used or the student can write their own [6].

Pre-calculating possible move sequences and storing them in a lookup table provided further performance gains.

The final category of improvement can be introduced as soon as parallel processing is discussed. Most universities have machines with many unused CPU cycles that a PVM/MPI cluster can harness to process Vexed puzzles [4] [5]. The implementation of a Vexed puzzle solver in a parallel environment is up to the programmer, but various approaches can be used. To solve Puzzle 5 of the Classic II puzzle set (see Figure E), my loosely coupled parallel implementation used a cluster controller to distribute individual boards (representing a part of the search space) to clients via

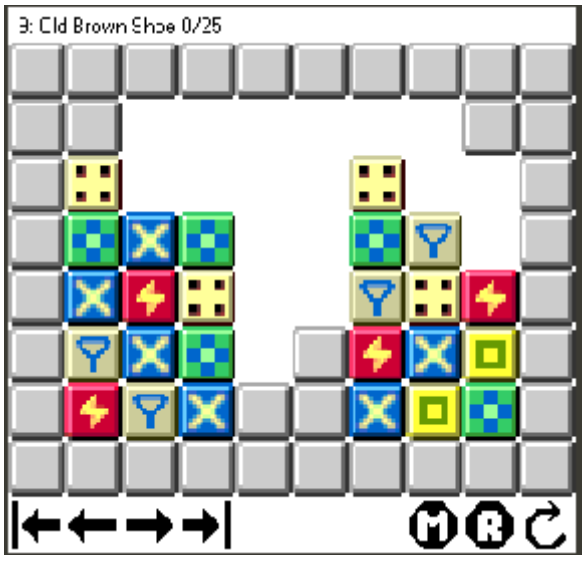

Figure E

TCP/IP sockets. Upon processing completion, each client returned its results to the cluster controller and picked up the next board in the queue. This approach cut processing time from 4 days on a single processor system down to 2 hours using 45 processors (approximately a 98% time savings).

CODING FEASIBILITY

The game of Vexed is one of the simplest programming exercises for CS1 and provides a challenging exercise for CS2 and more advanced courses. To implement a playable version of the game at the CS1 level requires the student to write 50 to 300 lines of C/C++ code (depending on their implementation and assuming a text-based console). A CS2 implementation of a solver can run anywhere from 400 to 1000 lines of code (my first version was 480 lines of code). Later implementations and modifications may increase code size by 200 to 400 lines and parallel implementations can run an additional 1000 to 1500 lines of code (my final parallel version was about 2500 lines of code). Key portions of code converted to assembler resulted in 370 lines of assembler for my optimized implementation (mileage may vary).

CONCLUSION

The game of Vexed and other search space problems offer great opportunities for teaching in an educational setting, permitting incremental performance enhancing concepts. From its humble beginnings in CS1 as an example of an implementation of a one player game, to a robust parallel program, students learn about many aspects of computer science.

BIBLIOGRAPHY

[1] Vexed Download for Palm OS with Review http://www.palmgear.com/software /showsoftware.cfm?prodID=28092

[2] Vexed Instructions, http://vexed.sourceforge.net/instructions.html

[3] Jexed - Java Based Vexed, http://dev.knowledgeassociates.com/hodglim/jexed2 /jexed2.shtml

[4] PVM - http://www.epm.ornl.gov/pvm/pvm_home.html

[5] MPI - http://www-unix.mcs.anl.gov/mpi/

[6] Zlib - http://www.gzip.org/zlib/