AC 2012-3083: MOTIVATING STUDENTS TO LEARN PROGRAMMING USING GAME ASSIGNMENTS

Dr. Rajeev K. Agrawal, North Carolina A&T State University

Rajeev Agrawal is an Assistant Professor at the Department of Electronics, Computer, and Information Technology at North Carolina A&T State University.

Dr. Zachary Kurmas, Grand Valley State University

Zachary Kurmas is an Associate Professor at Grand Valley State University. He teaches primarily CS 1, CS 2, and computer architecture.

Dr. Venkat N. Gudivada, Marshall University

Venkat N. Gudivada is a professor of computer science at Marshall University, Huntington, W.V. He received his Ph.D. degree in computer science from the University of Louisiana, Lafayette. His current research interests are in personalized eLearning, verification and validation of SQL queries, high performance computing for software visualization, information retrieval, and natural language processing.

Dr. Naser El-Bathy P.E., North Carolina A&T State University

Naser El-Bathy is an Assistant Professor of electronics, computer, and information technology at North Carolina A&T State University. He earned his B.S. degree from Wayne State University, Mich., M.S. (computer science, 2006) from Wayne State University, and Ph.D. (information technology, 2010) from Lawrence Technological University. El-Bathy is currently teaching at the North Carolina A&T State University. His interests are in health informatics, bioinformatics, artificial intelligence, intelligent information retrieval, and intelligent web development. El-Bathy may be reached at nielbath@ncat.eduity.

Dr. Cameron Seay, North Carolina A&T State University

Motivating Students to Learn Programming using Game Assignments

Game development is one of the fastest growing areas of software development. Many institutions have introduced Bachelor degree program in game development to cater to the industry demand. In this paper, we discuss the use of important key algorithms and data structures in game design and development. We highlight the usage of sorting algorithms and illustrate that other algorithms, such as searching algorithms, are most efficient when given properly sorted data. We demonstrate how game-based programming assignments can teach students algorithms. We also emphasize the importance of knowing multiple algorithms before developing a full game. We discuss the lessons learned while assigning games in beginning and intermediate level programming courses. Finally, we demonstrate how students benefit from a solid understanding of multiple data structures.

1. Introduction

Games have been used at all levels of learning, from elementary to college. Using games in teaching can attract new students, engage them in learning, and incorporate synthesis level of learning[1]. A game-based assignment will typically contain several components such as GUI, data manipulation, file management, and, of course algorithm design. A completed computer game reflects a significant understanding of all the algorithms and concepts needed. The best algorithm to use in a given situation is determined by the required processing time and memory requirements. Algorithms used in game development strategies range from Artificial intelligence to 3D mapping order to simulating projectile motion and other physical systems. Many of these algorithms rely on other algorithms to work correctly. For instance, searching algorithms depend on sorting algorithms. If data have already been sorted in an array, a programmer can use the $O(\log_2 n)$ binary search instead of the much slower $O(n)$ linear search. Thus, developers who want to code a truly efficient program must know many possible algorithm combinations and be able to discern which combination is best. The sorting algorithms widely used in game development today include the heap sort, quick sort, and the radix sort. Other advanced algorithms included in game development are collision detecting, data compression, and path finding.

The other important aspect of game implementation is the selection of suitable data structures. Choosing data structures well can both reduce memory requirements and run time. The algorithms used for the particular data structures can further reduce run time. A game's main data structure can be as simple as a one dimensional array or as complex as graph. Often times data structures to be used in an assignment are decided by the instructor in advance, but some instructors let the students pick the data structure. It also depends on the level of the course, if students are familiar with many data structures, it is a good idea to let them conduct research multiple data structures before narrowing to down to one.

The rest of this paper is organized as follows: Section 2 covers commonly used sorting algorithms; Section 3 discusses the data structures for the games. Some useful advanced algorithms are discussed in Section 4. Section 5 discusses five games that were given as assignments to the students. Section 6 presents our conclusions.

2. Sorting Algorithms

Sorting algorithms can be classified by the following features:

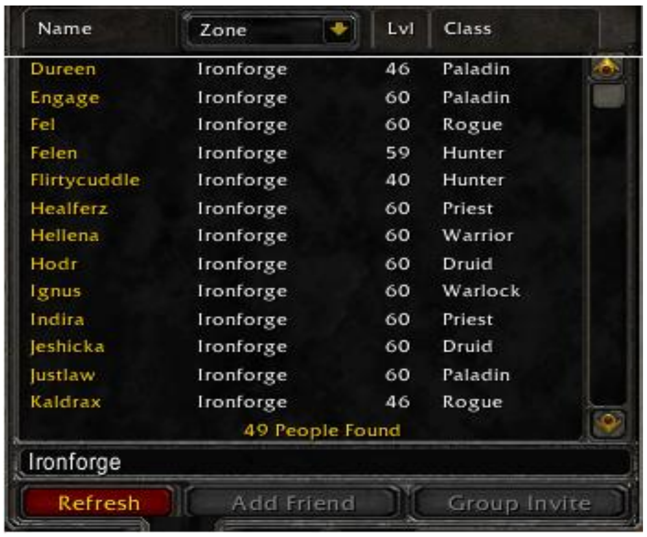
- Computational Complexity of Comparisons
- Computational Complexity of Swaps
- Memory Usage
- Recursion
- Stability
- General Method Used (e.g. Merging)

The sorting algorithms used in game development should reflect the best combination of all these features. A game may use several different sorting algorithms depending on the amount and type of data, and the degree to which that data is partially sorted.

2.1 Heap Sort: "The Heap Sort is well known as the *hacked* sort because heaps were never meant to sort data; this is just a neat side effect of the heap data structure. The heap is really efficient at inserting items in ($O(\log_2 n)$), and really efficient at removing the highest item also in $O(\log_2 n)$." [2] This sorting method is very widely used among today's programmers. The downside of this sorting algorithm is that it requires a "swap space" in memory equal to the size of the input. The cost of allocating and managing this swap space can make the heap sort less efficient than other sorting algorithms.

The heap sort is primarily used to sort a priority queue type of heap. A priority queue contains items with keys. It supports two basic operations: insert a new item, and remove the item with the largest key. This system of operations sorts items by priority. For example, a game could use a priority queue to retrieve all network, input, and interface updates in each of the game's frames. Each game function can be looped through entire queue. The looping occurs because the largest item isn't discarded like a normal heap structure; it is just placed at the bottom. After each function is processed the priority queue will be right back where it started for the next iteration.

2.2 Quick Sort: The general-purpose sorting algorithm quicksort is known as 'The Fastest Sort'. Even though the quicksort technically has the same $O(n^2)$ worst-case as bubble sort it almost always runs in O(nlog2n). The probability of unintentionally observing a running time larger than O(nlog2n) is exponentially small. Also, unlike the heapsort the quicksort works well with large amounts of data. Its ability to handle large inputs makes quicksort the best general-purpose sorting algorithm. It can be used on any type of data that can be sorted numerically or lexicographically. One example of this could be the output to the screen when a player searches for players in a region of an online game. As shown in figure 1, the players found matching 'Ironforge' are brought up in a list, the user can sort them by their name, level, or classes.



Name	Zone	Lvl	Class
Dureen	Ironforge	46	Paladin
Engage	Ironforge	60	Paladin
Fel	Ironforge	60	Rogue
Felen	Ironforge	59	Hunter
Flirtycuddle	Ironforge	40	Hunter
Healferz	Ironforge	60	Priest
Hellena	Ironforge	60	Warrior
Hodr	Ironforge	60	Druid
Ignus	Ironforge	60	Warlock
Indira	Ironforge	60	Priest
Jeshicka	Ironforge	60	Druid
Justlaw	Ironforge	60	Paladin
Kaldrax	Ironforge	46	Rogue

49 People Found

Ironforge

Figure 1. Searching for players in World of Warcraft

2.3 Radix Sort: The radix sorting algorithm is known in the gaming community as “The Clever Sort.” The two previous mentioned sorting methods are called general-purpose sorting algorithms, since they can sort any kind of data. In contrast, the radix sort is only useful when sorting discrete data; but it is the fastest sorting algorithm. When the data set gets extremely large, it is even faster than a quicksort. The radix sort is being used more and more frequently in today’s game development because computers commonly have enough memory to run it. For example, radix sort might be used when a program sorts a scoreboard of a multiplayer game.

Perhaps the most popular use of sorting algorithms is depth-sorting. In 2D games like *The Legend of Zelda* as shown in figure 2, it is common to have sprites appear to be 3D and have the screen look slanted to give some sense of depth. The only way to avoid the need for this sorting of the depth of objects would be to make a top view game where you are looking straight down on the players. The sprites on the screen are sorted by their y-position on the screen and drawn in the order given. Therefore, depth-sorting can be defined as putting the items on the screen in order of which appears above another to give a nice look to a game. Although depth-sorting isn’t an algorithm itself, it is a very important tool.

Figure 2. The Legend of Zelda screenshot

3. Game Data Structures

Linked lists are often not the most efficient data structure to use in a game. A linked list is not stored contiguously in computer's memory; therefore, searching, creating, and deleting operations may be slower. In practice, fixed size arrays are used commonly in grid-based games such as Numbrix, Minesweeper, and Connect Four. The most important data structures for game programming are Binary search tree (BST), stacks and queues.

3.1 Stacks: Stacks are most often used to store the state of a menu or the overall game. A typical game menu contains many options for the user. These menu options are often nested. A stack can model a nested menu system quite easily. Every time the user selects a sub-menu, the new menu is created and pushed onto the stack and when the user presses *Escape*, the current menu is popped off of the stack and control reverts to the previous menu. The current menu is always on the top of the stack.

3.2 Queues: Basically, a queue is a *FIFO* structure (*First In, First Out)*. The person who gets into line first will be checked out first. The queue is much like a stack but is implemented in a different way. In some games like a real time strategy, it is possible to tell a certain unit to move to one place and then move to another place after they are done making the first move. This is called *command queuing*. Each command is enqueued in order and then processed in order. Queues can also be implemented on a linked list, or plain array structure. Often, the array queue will be manipulated further into a *circular queue* where the end of the array connects to the front.

3.3 The Binary Search Tree (BST): A BST is a binary tree in which each internal node x stores an element such that the element stored in the left subtree of x are less than or equal to x and elements stored in the right subtree of x are greater than or equal to x. The BST search algorithm is roughly $O(\log_2 n)$, unless it's a linear chain of nodes the worst-case scenario is of order O(n). The possible use of a BST in a game is very general. A BST can be created to keep the records of game objects using the time of creation as the key, each with a number of properties. They can hold a very large amount of data, which can be searched very quickly.

Binary search tree is also used to search the solution space of a game such as tic-tac-toe, number guessing game etc. AVL tree, Red-black tree and Splay trees are binary search trees that keep their heights small when items are inserted arbitrarily[4]. The search function uses this structure to traverse the tree in one of three recursive ways.

1) Pre-order (compare to current node, move to left node, then move to right node)
2) In-order (move to left node, compare to current node, then move to right node)
3) Post-order (move to left node, then move to right node, compare to current node)

Another variation of BST is the n-way trees, which are used to keep the depth of the tree low and searching an item faster.

4. Advanced Algorithms

There are countless games exist in the game world, so are the ways to write algorithms for the tasks involved in them. Some of these tasks include finding a path for a character, or the problem of an entire physics system including collision detection so that when a ball hits a wall it will bounce back. Another more advanced algorithm used in games today is that of data compression.

4.1 Pathfinding: Pathfinding algorithms fall under the topic of Artificial Intelligence (AI). Pathfinding is probably the single most popular--and potentially frustrating--game AI problem in the industry[5]. A good example of pathfinding is in the popular game *Bomberman* which released on many console platforms. For the computer to beat you it must find a path to you and then trap you with a bomb so you get hit by the explosion. Likewise, a pathfinding algorithm will be needed for a character to find its way. It could be as simple as a basic coordinate comparison, with collision detection for walls along the way, or the algorithm could make use of graph techniques to calculate a shortest path.

A simple way to solve pathfinding is by comparing the position of the character and the position of the target. If that move is not possible choose a random direction to hopefully get around the obstacle blocking the path. Although this algorithm will run very fast, it is not very good and will lead to trouble in any semi-advanced game. The next best idea is known as object tracing. Whenever the character finds an obstacle, it traces the outline of the obstacle until it finds a way around it. This is a slight improvement from the original coordinate comparison because the movement after the comparison is set to one direction instead of random.

The previous two pathfinding algorithms are instant pathfinders. They determine the path that the AI logic will be following, every time path is found to a new square. A good example of a game that might only use a simple instant pathfinding algorithm is Bomberman as shown in figure 3. All three of the computers must try to get to the player and hit him with a bomb explosion. There are other ways that find a much more efficient path. In games where players control a unit's movement by clicking where they should go on the interface, the players expect their character to get to where they told them to go in the shortest time possible. This means that they need to find the shortest path from the beginning to the end.

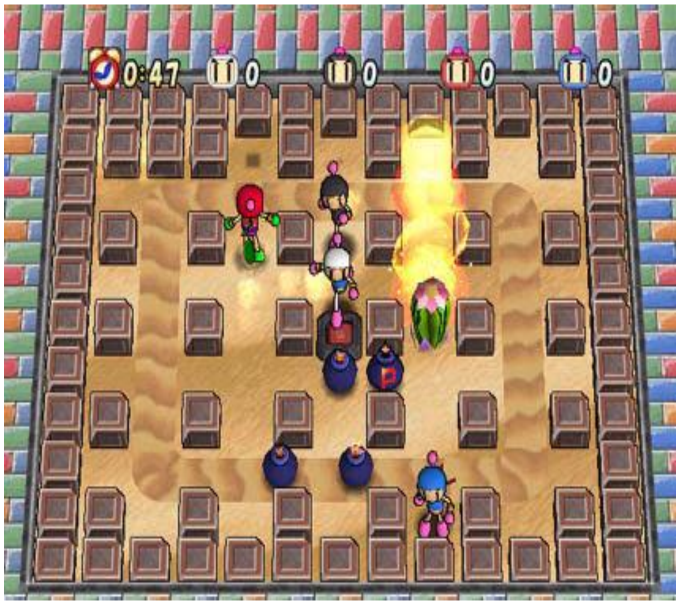


Figure 3. The characters figuring out their paths in Bomberman

The best solutions to pathfinding are based on graph algorithms or better known as a quadtree or an octree in 3D, and a shortest-path algorithm. This could be implemented by creating a treelike structure that resembles a graph where the nodes are intersections of the available movable locations for a map, and the edges between the nodes are the actual paths whose weights are the distance between them. Then by implementing a shortest-path algorithm whenever the character decides to move, it can choose the quickest route to reach the ending goal position. A graphing solution is much more complicated to the previous two solutions, but it has a lot more potential to lead an object to its goal when there are many obstacles or a large map.

4.2 Physics: The vast majority of games today contain simulating movement. As game progresses, the characters, objects, and scenery also move. To provide truly immerse experience; everything has to move convincingly enough that it models real-world physics. This requires a programmer to learn the essentials of physics, math, and 3D programming.

4.2.1 Collision Detection: Collision detection is a very common issue in game development that deals with the problem of deciding when an object hits a wall or another object. The Bounding Sphere method is a very popular method in detecting collision. In this method, a sphere is fit around the object and then radius around it, is shrunk until it's as tight as possible. The use of this method requires three steps to test for collisions[6]:

1) Check to see if object is above the plane.
2) If so check to see if center point of mass is inside the radius of the compared object.
3) If the object has collided with the other object, react to the collision.

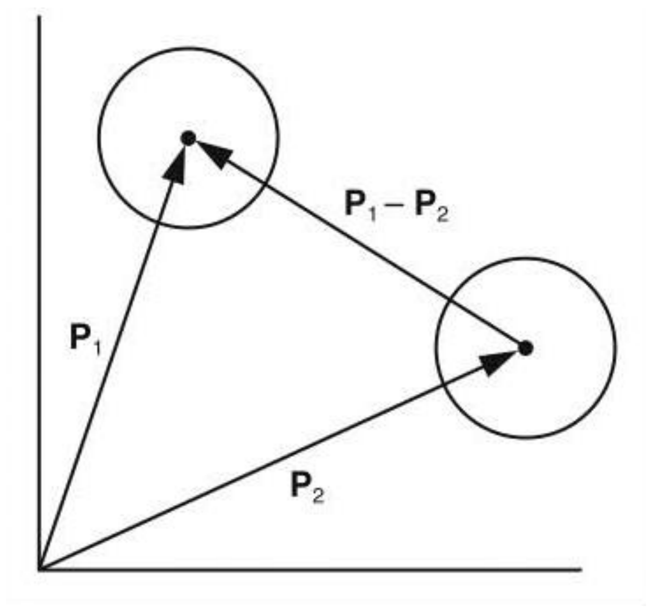


Figure 4. Collision Detection with vectors

The calculations are usually done with vectors to simplify things as shown in figure 4. This example could pertain to a pool game. We want to see when the two pool balls collide. Subtracting P_2 from P_1 gives a vector specifying the distance between the two centers of mass. Compare that outcome to the sum of the radii of the spheres. If the distance is less than the sum of the radii a collision has occurred, and vice versa.

4.3 Data Compression: It is not just storage space that drives the need for data compression. There are a few areas of game programming that still require efficient data compression. The most pressing concern today is the rapidly widening gap between the speed of the processors in a system and the data bus. Unfortunately, Graphic Processor Units (GPU) are getting so fast that they can draw more data than the Computer Process Unit (CPU) can actually send to it. This problem is caused by the fact that computer busses are very slow in comparison to the CPU and GPU speeds[2].

Data compression is a very advanced topic but it's important enough to be mentioned in anything about algorithms for game programs. The easiest example of data compression is the Run Length Encoding. In a simple compression example, a string "AAABBBBBCCCC" can be encoded to just "3A5B4C." The next step in Data Compression is an algorithm known as Huffman coding. The idea comes from the fact that when you have a text file in a game, since there are 256 characters each single character requires 8 bits of memory. An average text file can only use 50-60 characters. As shown in Figure 5, the Huffman Tree attempts to make the most frequent characters take up a small amount of space and the least frequent characters take up the most amount of space. As in the example, the character 'E' would only require the string "00". Huffman does have its advantages and disadvantages though[7].

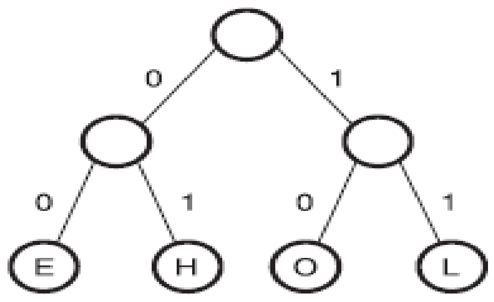


Figure 5. A Huffman Tree example

RLE and Huffman Decoding techniques are known for being *lossless* compressions because no data is lost. The actual topic of compression is much broader than just these two. Another popular type is called *lossy* compression. A great example of a lossy compression is the mp3 music file format. The mp3 compression completely removes the very high frequency data of a sound file. While most of these frequencies can't be heard by the human ear, it does reduce the quality of the sound to save space.

5. Discussion on Game Assignments

In this section, we will discuss five games we have assigned to freshman and sophomore level students. We briefly describe each of the game below:

Numbrix: Numbrix is a single player game created by Maryln Vos Savant. It has appeared in Parade Magazine for last few years. The game consists of a 9x9 grid, in which the player must fill in the missing numbers to form a continuous path from 1 to 81. When designed as a computer game, it is possible to create a grid of any size[8].

Metro: The basic idea of Metro is that players take turns laying tiles representing sections of subway onto the game board. The goal is to maximize the lengths of your subway lines while minimizing the lengths of your opponents' lines. A standard game is played on an 8x8 grid. The spaces along the edge of the grid represent subway stations. Each player "owns" some number of these stations. On his turn, the player draws a random tile representing a collection of four subway lines and places it on the board. This placement may (1) extend a subway line he owns, (2) complete a subway line owned by another player, or (3) both. A subway line is considered complete when it extends from a player's station to some other station on the board. Players score points based on the length of completed subway lines. There are four stations in the middle of the board. Subway lines ending at one of these stations score double. A player may keep one subway tile in his hand to be used in place of the randomly drawn tile. When used, this "held" tile is replaced with a randomly drawn tile. The game is interesting because when laying tiles, players must carefully balance between maximizing the lengths of their own subway lines and minimizing the lengths of their opponents' lines[9].

Connect Four: It is a two player game in which players take turns dropping colored discs from the top open slot. A standard game is designed played on a 6 row by 7 column grid; however, it is possible to create any size of game with any number of players. The game was first sold as a board game by Milton Bradley in 1974, then released as a video game in 1979.

Igel Ärgern (Hedgehogs in a Hurry): This game supports play by 2 to 6 players. This game gives players several wooden disks (the hedgehogs) to place in one of six rows. After rolling a die, a player moves one of her hedgehogs one row up or down into an adjacent row. The player is then allowed to move any hedgehog (even belonging to another player) forward in the column specified by the die. Some of the positions are marked as black spaces. Hedgehogs caught there can't continue moving until they're in last place. The first player to get three of her four hedgehogs to the end of the board wins. There are many interesting variants available to keep the game interesting[10].

Tic-tac-toe: Tic-tac-toe is a 2 player game in which players alternate marking X and O on a 3x3 grid. The first player to successfully place her marks in a horizontal, vertical, or diagonal line, wins the game. This game can be used to introduce students to the game programming at a very early stage in the course only when only a few topics (two-dimensional arrays, if-else, and loops) have been covered.

Table 1 lists the programming language used and the topics covered by these game assignments. All the games were assigned to sophomore students except Tic-tac-toe, which was assigned to freshman students. All of the games discussed here are grid-based, which allowes students to implement them using an array. Numbrix, Connect four and Tic-tac-toe were assigned as 3-4 weeks projects, whereas Metro and Igel Ärgern were 6- to 8-week term projects. Students were given intermediate deadlines to keep them focused.

Game	Programming Language	Topics Covered				
		File I/O	Data Structure	Exceptions	GUI/Text	Inheritance
Numbrix	Java	X	Array	X	Both	
Metro	Java		Array	X	GUI	X
Connect four	Java	X	Array	X	Both	
Igel Ärgern	Java	X	Array, Stack	X	GUI	X
Tic-tac-toe	C++	X	Array		Text	

Table 1. Details of game assignments

Lessons Learned: We asked students to provide feedback on the assignments. Students commented that initially assignments appeared to be very difficult, but as they started implemented few features of the game, they were confident and eventually were able to complete most of the features of the game. At the end of the semester, they appreciated the knowledge gained through these assignments. To help them with the implementation, they were provided some small intermediate assignments to explain the underlying concepts. We introduced Model-View-Controller (MVC) software architecture and included it as one of the important components of grading. In our experience, we found that the biggest challenge for students was integrating game logic with GUI concepts. Students were encouraged to first implement text version of the games, which helped them to focus on game data structure, file I/O and overall game logic and later combine these with the GUI. Based on students' feedback, we suggest the following to the instructors, who use game assignments in their courses:

1. Clearly outline the game logic and if possible bring a copy of the game to the class and let students play.
2. Divide the entire game into many small steps and assign a due date to each step.
3. Encourage students to design modifications to the game. Allowing students to design part of the assignment provides a sense of ownership that many students find motivating. (It is also refreshing to see just how creative some students can be.)
4. If possible, provide a program shell as a starting point.
5. Provide a grading rubric to make sure students complete all the components of the assignment.

6. Conclusion

Understanding how algorithms and data structures work together to produce the correct output is a skill that no programmer can overlook. Most of the topics discussed are learned through practice and experience. It is also important to realize that not all data structures or algorithms have a specific implementation. Everything in game programming is a tradeoff, and one should always analyze exactly what are requirements of a game in terms of speed, portable platforms, number of simultaneous players, online-offline etc. and then select the appropriate algorithms and data structures. Game assignments generate enormous interest among students in learning programming concepts. It is possible to design game assignments for introductory courses, as well as advanced data structure course as shown in section-5.

Acknowledgements: We would like to acknowledge the contribution of Jordan Smith, Kettering University student in identifying algorithms and data structures useful for games.

Bibliography

1. T. Baibak, R. Agrawal,“Programming Games to Learn Algorithms," *ASEE Annual Conference Proceedings CD (Paper #495),* 2007 ASEE Conference and Exposition, Honolulu, HI, June 24-27, 2007.
2. R. Penton, Data Structures for Game Programmers. The Premier Press. 2003
3. R. Sedgewick, Algorithms in Java*: Third Edition.* Addison Wesley. 2002
4. Donald Knuth. The Art of Computer Programming, Volume 3: Sorting and Searching, Third Edition. Addison-Wesley, 1997. ISBN 0-201-89685-0. Section 6.2.3: Balanced Trees, 458–481.
5. Game AI Resources: Pathfinding. URL: http://www.gameai.com/pathfinding.html
6. D. Conger, Physics Modeling for Game Programmers. Thompson Course Technology PTR. 2004.
7. D.A. Huffman, "A Method for the Construction of Minimum-Redundancy Codes", Proceedings of the I.R.E., September 1952, 1098–1102.
8. http://www.parade.com/numbrix
9. http://www.boardgamegeek.com/boardgame/559/metro
10. http://boardgamegeek.com/boardgame/95/igel-argern