



Generating Difficult and Fun Nonograms

Milo Roucairol  and Tristan Cazenave 

LAMSADE, Université Paris Dauphine - PSL
Place du Maréchal de Lattre de Tassigny, 75016 Paris

Abstract. Nonograms are Japanese logic puzzles where the player must find a 2D black-and-white image using information on the columns and rows. Here we present a new method to generate nonograms of varying difficulties and enjoyability using a human-like solver to estimate the difficulty of a puzzle, and Monte Carlo Tree Search algorithms to optimize the estimation of the difficulty.

Keywords: Nonogram · Monte Carlo · puzzle.

1 Introduction

Logic puzzles are a staple of the gaming landscape. Widespread as early as 1913 thanks to newspapers, or even in 1783 with Euler’s Latin square. Every living person has encountered one (be it crossword or sudoku), and likely solved one in their life. They are especially popular among retired people and computer scientists, as they are very similar to many computer science logic problems. Solving or programming a solver for these problems is common in CS studies and research.

Sudoku and Sokoban are two of the most notable logic puzzles of the 20th century. Another notable Japanese logic puzzle from the 80’s was the Nonogram. It was quickly taken over by Nintendo with the Picross series for the Game Boy and the SNES with Mario’s Picross in 1995 and entries using the sprites from Nintendo’s most popular franchises. Spinning off multiple variants, like color nonograms, mosaic nonograms, nonograms with unknowns, mega-picross, etc. We are interested in the original version of the nonograms.

A nonogram is a grid-based logic puzzle. The player has to fill a grid with either black or white squares until there are no more unknown cells and the picture is fully formed. The player is given instructions on the rows and columns in the form of numbers on top and left of the grid. Each number n indicates that there are exactly n adjacent black squares. The order in which the numbers appear is important too, the groups of adjacent black squares must appear in the same order as on the clue.

Here we set out to generate Nonogram puzzles of varying difficulties, this has been the subject of some research. First in 2009, K. Joost Batenburg et al. [1] introduced a method to evaluate the difficulty and generate simple nonograms. Followed in 2012 by another assessment of the difficulty of nonograms [2]. However, both research papers do not focus on player-related difficulty but only on a

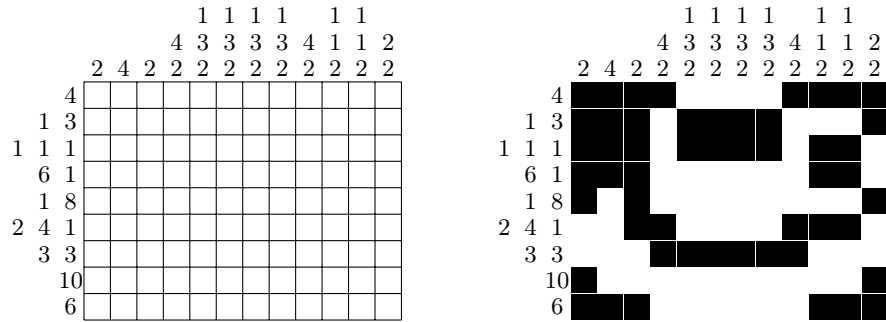


Fig. 1: Example of a medium sized nonogram, it depicts a cup of coffee on a saucer

solver/mathematical-related difficulty, here we intend to explore the nonogram design with human players in mind. Solvers for nonograms are also another point of interest, with genetic approaches like in Wiggers’ work [12].

In this paper we will use a human-like deterministic solver to evaluate the difficulty and enjoyability (fun) of the generated nonograms, and state of the art Monte Carlo Search algorithms for the generation process. Our goal is to make a fast nonogram generator with targetable difficulty to allow for new ways of playing nonograms: time attack, survival, or endless modes, with a set or increasing difficulties (Tetris-like gameplay).

2 “Human” Solver

An optimization process requires a way to evaluate the quality of a state. A state of our search tree is a small black-and-white image. We imagine two ways to evaluate the difficulty and enjoyability of that image:

1. A neural network trained on a set of images and their evaluation by humans
2. A solver that emulates human logic and stores data about the resolution

We decided to use a solver because designing a human-like solver is an interesting and intuitive task, and the neural network training datasets are not publicly available yet.

One technique that is commonly used when solving nonograms is to check the possible extreme positions of a group of adjacent filled tiles and see if they overlap. If they do, the part where they overlap is necessarily black. This is the main technique used by humans, as such it is also the main one used by the solver. Conversely, spaces where no constraint-satisfying configuration had a black square on add a white square

In Figure 2, first the 8 white cells are deduced given all positions of the 10 consecutive white cells overlap on these 8 cells. Then, all possible positions of the 2 in the third column exclude the 6 first cells from top to bottom (denoted

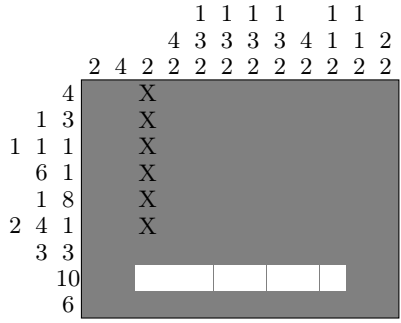


Fig. 2: Example of column and row swipes

with black “X”). These 6 cells must be black. This in turn helps us fill the 1st, 4th, 5th, and 6th lines from the top.

Our method takes a line or columns, its clue, and the already-filled cells, and computes all possible fillings that meet the requirements of the clue. Then if for a cell in this line, all possible filling leaves it black or white, the cell is filled in the grid. This method is called over the rows and columns according to the last cells modified, this mimics human solving as the attention and memory of a human player are focused on the last cells modified too.

This simple method is enough to solve most nonograms, but the hardest ones may require more advanced techniques, like the one called “edge solving”. Edge solving is a type of deterministic guessing, it is done by guessing a white square, usually on the edges of the grid, and unrolling the usual resolution until a contradiction is found. Once a contradiction is found, the cell that was guessed as white is colored black. That technique is unknown by most players, and should be avoided unless the difficulty target is high and it only happens 1 to 3 times in the puzzle.

The algorithm has a stack of rows and lines to check (starting with all rows and columns), it checks them all until if finds a line/row move, it then applies that move to the row or columns and adds the rows and columns of all modified cells to the stack. If the algorithm does not find a line/row move, it tries to find a deterministic guessing move that gives a contradiction. If no move is found it returns the grid in its state, completed or not. We decided to not allow the solver to make another deterministic guess while one is active, thus used as a score function it does not produce nonograms unsuitable for humans. If a nonogram is solvable but necessitates the use of nested deterministic guessing it is deemed unsolvable by the solver, and discarded during the generation process (see next section). This is not a problem since deterministic guessing is almost never used in most nonogram games, and never in a nested fashion to the best of our knowledge.

The solving time of the algorithm is a metric sufficient to design hard nonograms by maximizing it. But for more precise difficulty targets, the solver returns other metrics such as the number of time it used edge solving, the number of

step, the length of each backtracking, and others. These metrics are used to estimate the fun and the difficulty of a puzzle.

3 Generation

3.1 Search Space

The generation process takes the form of a search tree search as we use Monte-Carlo Tree Search and other deterministic search algorithms. All the algorithms share the same search tree (or search space). The root of the search tree is the initial nonogram that is submitted to the algorithm: an empty grid with black cells only. From that state, the available moves are swapping the value (white or black) of any rectangle inside the grid.

This simple single move allows the search algorithms to reach diverse grids in fewer moves than setting the value of each cell. It also allows for drastic changes in difficulty and grid layout in very few moves, while also setting the value of each cell individually if needed.

It is possible to start from an already existing nonogram, or from an image, and input a list of cells that cannot have their value swapped to preserve the depiction: nonograms usually depict objects.

3.2 Algorithms

Monte Carlo search algorithms are a family of search algorithms that use sampling (playouts) to learn about a search space to guide the search. They are a key element of recent breakthroughs in game playing such as Deepmind’s AlphaGo [11], but are also widely used in other machine learning applications like multi-step retrosynthesis [5] or graph theory [9]. We selected the following algorithms.

1. UCT: Upper Confidence bound applied to Trees, the most commonly used MCTS algorithm. [6]
2. RAVE: Rapid Action Value Estimation, a variant of UCT using All Moves As First (AMAF), a machine learning technique appropriate to our search model because the order of the moves does not matter. [4]
3. NMCS: Nested Monte Carlo Search, a method that recursively calls lower level version of themselves on child states to find the route leading to the best score. A level 0 NMCS is a playout. [3]
4. LNMCS; Lazy Nested Monte Carlo Search, a variant of NMCS that introduces the exploitation/exploration dilemma and prunes lower level LNMCS using playout based evaluation. [10]
5. NRPA: Nested Rollout Policy adaptation, an algorithm inspired by NMCS that learns online a policy to guide the playouts. [8]

For combinatorial optimization, two families of MCS exist. The main and most commonly used one is called MCTS and uses an iterative approach, it

includes UCT and RAVE. The other family is recursive, it includes NMCS, LNMCS, and NRPA.

Deterministic search algorithms are simpler search algorithms that do not involve sampling or machine learning. Intuitively we could expect them to provide inferior results compared to methods using machine learning. Recent approaches show that this is not always the case, they can outperform even the most complex state-of-the-art machine learning on certain problems, as shown in [7,9]. We selected two widely known deterministic search algorithms:

1. BFS: Best First Search uses a list to keep track of the best candidates, it opens the best one, evaluates its child, inserts them in the list according to their evaluation, and repeats. This algorithm is complete, it will explore the entire search space given enough time.
2. BEAM: Beam Search keeps width w states open at all depth. From a depth n it opens all the children of the w states and keeps the w best ones for depth $n + 1$.

4 Results

4.1 Experimental Setup

The experiments were made using a 2.60 GHz i5-13600K Intel single core.

We define three goals of optimization:

1. Difficulty for the solver program: time spent solving it by the solver.
2. Difficulty for players: estimation by the solver.
3. Fun: estimation by the solver.

Nonograms also come in many different sizes, we decided to compare our program on sizes 5x10, 10x10, and 15x15 as these are the most common nonogram sizes used in games. Larger nonograms exist too, but the focus is usually on the picture depicted and not on the difficulty or the puzzling aspect.

Monte Carlo Search experiments require multiple runs. We observed an important standard deviation among preliminary runs, so we decided to run each algorithm 10 times over each combination of size and goal.

Finally, this method is intended to be used in real-time by games for player versus player, or survival (like Tetris for example) game modes for example. In these game modes small new nonograms must be generated as fast as the player solves them so we decided to set the optimization time to 60 seconds on an Intel Core i5-13600K using a single core. This method is also fit for generating regular low-quality nonogram games, or daily challenges.

For Hyperparameters, BEAM uses a width of 10, UCT a learning ratio of 1, RAVE a threshold of 5, NMCS a level of 2, NRPA a level of 2, and LNMCS a level of 3, pruning ratio of 0.8 and 3 playouts per estimation.

4.2 Optimizing the Solver Difficulty

The solver was made to behave like a human, thus the time taken to solve a nonogram is indicative of its difficulty, but it is also an interesting task by itself as a benchmark for estimating MCS algorithms performances. The solving times of graphs optimized to be more complex to the solver by all algorithms are presented in Table 1.

	BFS	BEAM	UCT	RAVE	NMCS	LNMCs	NRPA
5x5	0.000417	0.000181	0.00150	0.00121	0.00179	0.00180	0.000131
5x10	0.0200	0.00219	0.226	0.145	0.103	0.0963	0.00272
10x10	0.0221	0.00305	0.591	0.568	0.308	0.563	0.0286
15x15	0.00995	0.00254	9.578	10.466	3.066	13.07	1.741

Table 1: Mean solving times of 10 nonograms each generated in 60s in seconds, greater values indicate success from the Monte Carlo optimization algorithm

4.3 Optimizing the Estimated Player Difficulty

To estimate the difficulty for a human player, we focus on three metrics:

1. The number of steps required to solve the puzzle
2. The number of times edge guessing is required
3. The number of times backtracking is required (the next available move is not in vicinity)

Only focusing on the number of steps like in [1] would not result in an actually difficult puzzle, but in tedious ones (like the one shown in Figure 4 of their paper). While the difficulty and tediousness often overlap in nonograms, and a minimum number of steps is required to make a hard enough puzzle, the number of steps is not sufficient to build difficult nonograms, and is here the least important part of the evaluation. The other two members of the difficulty estimation are the number of times the player must backtrack to find a new available move (the move is N cells away from any modified cells last), this is the most important part of the equation and where we think resides the true difficulty. The last part of the equation is the number of advanced deterministic guessing techniques the player must use (edge guessing), puzzles must refrain from using too many of these in order to remain enjoyable, as such only the reward follows a square root function.

$$difficulty(N) = n(N) + \sqrt{g(N)} * 50 + d(N) * 10$$

Where N is the solution to the nonogram, $n(N)$ the number of steps in that solution, $g(N)$ the number of deterministic guessing, and $d(N)$ the number of times no move is found in the 7 lines and rows checked after making a move.

The factors (50 and 10) were set to these values because a long backtracking is approximately 10 times more complex and time-consuming for a human than executing moves found directly. A deterministic guessing can be simple or very hard, for the sake of the method’s simplicity it is set 50, but could be computed more accurately depending on what happens during the guessing, or set to higher values like 100 by default.

The estimated difficulty scores of graph optimized by all algorithms are presented in Table 2

	BFS	BEAM	UCT	RAVE	NMCS	LNMCs	NRPA
5x5	103.710	103.710	158.670	148.930	168.956	166.241	96.536
5x10	112	112	197.533	184.609	211.582	212.475	116.202
10x10	181.602	217	277.778	280.949	271.944	272.314	198.370
15x15	299	401	308.975	313.443	320.596	296.749	282.483

Table 2: Mean estimated difficulty scores of 10 nonograms each generated in 60s

4.4 Optimizing the Fun

Estimating the human player’s fun is a harder task as it is even more subjective. According to players, the fun is “like dominos or finishing a jigsaw puzzle, seeing something meticulously set up and solved line by line finally stepping back to look at the result, it’s satisfying because of completion itself”, or “solving any nonogram gives me a dopamine hit, but the bigger ones definitely hit harder.”. Other players find satisfaction in hard puzzles (but we are not going to use this definition of fun in this section), otherwise the puzzles are generally fun by default, unless they are unfun.

An unfun puzzle can be defined as unnecessarily tedious and frustrating. As such, the number of total steps, edge guessing, and backtracking must be limited with some tolerance to avoid making the puzzles so simple they become tedious again.

To avoid puzzles so easy they become unfun, one of the terms in the parenthesis favors balance between the number of black and white cells. The other is the kurtosis value of the lengths of the moves of the solution. Both are multiplied by the number of unique lengths of the moves of the solution because players enjoy the most variety.

$$fun(N) = \left(\frac{4 * b(N)w(N)}{s(N)^2} - k(N) \right) * u(N) + 5 - max(d(N), 5) - g(N)^2$$

Where N is the solution to the nonogram, $s(N)$ the size of the grid (number of cells), $k(N)$ the kurtosis of the lengths of the moves used, $u(N)$ the number of moves of different lengths used, $b(N)$ the number of black cells, $w(n)$ the number of white cells, $n(N)$ the number of steps in that solution, $g(N)$ the number of

deterministic guessing, and $d(N)$ the number of times no move is found in the 7 lines and rows checked after making a move. Like with the difficulty, this function is an attempt to model the fun. It is to be improved, ideally supported by puzzles rated by players, which we do not have.

The estimated fun scores of graph optimized by all algorithms are presented in Table 3.

	BFS	BEAM	UCT	RAVE	NMCS	LMCS	NRPA
5x5	12.765	12.765	12.765	12.696	12.765	12.765	11.099
5x10	14.008	0	16.523	15.642	16.959	17.072	14.326
10x10	18.596	0	19.807	18.540	20.213	19.888	16.417
15x15	18.976	0	13.577	12.113	11.663	13.235	16.225

Table 3: Mean estimated fun scores of 10 nonograms each generated in 60s

4.5 Overview of the Solver’s Estimations

To assess our estimation function and solver, we ran it on 150 nonograms, in figure 3 we show 15 of them and their fun scores, difficulty scores, and solving times.

5 Discussion

With the optimization of the solver’s time on Table 1: finding nonograms complex enough to maximize the time spent by the solver solving it, the size 15x15 is too large for most algorithms and does not allow to collect reliable data as the solver is rapidly countered by UCT, RAVE, LNMCS, and even NMCS despite showing inferior solving times. The standard deviation is high enough that LNMCS performance can be attributed to luck (UCT 5.019, RAVE 3.550, LNMCS 8.323, NMCS 2.544, NRPA 0.869). But this does not indicate that the solver is underperforming as seen in Figure 3 where it solves most nonograms rapidly. On lower grid sizes, LNMCS, UCT and RAVE are similar, with standard deviations of approximately 0.3 on 10x10. NMCS and NRPA are outperformed by the other MCS algorithms. BFS and BEAM results are even lower, this may be due to the use of playouts which produce complex puzzles without the need to evaluate at each step (MCS algorithms only evaluate at the end), leading the deterministic algorithms to be left behind with larger grids. However, deterministic algorithms seem to perform better on small grids.

The optimization of the estimation of difficulty with a handmade function on Table 2 shows most MCS algorithms providing similar scores. The standard deviation is approximately 30 with all the MCS algorithms for size 15x15. Differences between UCT, RAVE, NMCS, and LNMCS are minimal and can be attributed to their random nature. However, NRPA, the only algorithm learning a policy,

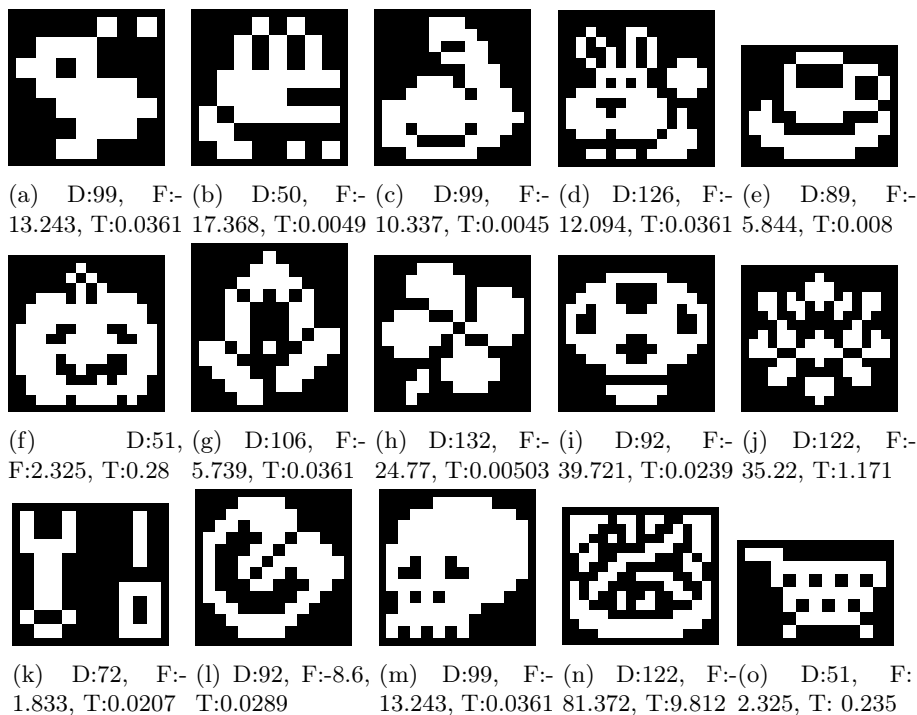


Fig. 3: Solver and its estimations applied to 15 nonograms, D: difficulty score, F: fun score, T: solving time by the solver in seconds

underperforms, while the BEAM search outperforms all MCS algorithms. The proximity of the 10x10 and 15x15 difficulty scores for the MCS algorithms seem to indicate that the solving times become too important and block the algorithms at the beginning of the search trees. BEAM evaluates the children of 10 states and goes down the search tree, thus allowing it to reach a harder puzzle.

The optimization of the estimation of fun with a handmade function on Table 3 shows that BEAM locks itself in a local maximum. Again the MCS algorithms share similar results, with a slight advantage for LNMCS, they all have a standard deviation of approximately 1 for sizes 10x10 and 15x15 and less than 0.5 for sizes 5x5 and 10x5. The 15x15 size is where our optimization process starts to struggle to optimize the fun further, it seems adequate since the tediousness of a puzzle increases faster than its fun with the size in our opinion.

The application of the evaluations to select 15 nonograms in Figure 3 shows that the difficulty score, while far from being perfect, is quite correlated to the difficulty of the puzzles and is sometimes better at evaluating the difficulty than the solver time. However, our attempt at a fun function is able to produce fun puzzles when used as a goal function for optimization but fails to recognize the fun in these nonograms. This may be explained by the penalization of determin-

istic guessing, and the kurtosis, for example, Figure 3n has a high kurtosis as it has a mean of move length of 2, same for the kurtosis of Figure 3j. The imbalance between black and white, which does not necessarily have to be respected in handcrafted puzzles, can skew the estimation of fun too. The fun evaluation function aims to avoid tedious puzzles, even if it may produce false negatives.

6 Conclusion

In this paper we provided two new ways of evaluating the difficulty of Nonogram puzzles, improving on the previous work. These two new ways being using the solving time of a solver designed to tackle the puzzle in approximately the same way as a human, and a new difficulty function taking backtracking and deterministic guessing into account. We also experimented with the definition of fun, with mixed success, fun is highly subjective and our method only avoids unfun puzzles, but can flag fun puzzles as unfun, partly due to the assumption on the fun definition which is not the same for all players.

We then used these functions to design new nonogram puzzles using search algorithms including State of The Art Monte Carlo Search algorithms. While MCS algorithms offered similar results, we noticed that NRPA was either able to learn a policy and outperform the others, or be outperformed.

Our method is apt for this task and can generate many nonograms of target difficulty rapidly. You can find the code here:

<https://github.com/RoucairolMilo/nonoGen>

References

1. Batenburg, K.J., Henstra, S., Kusters, W.A., Palenstijn, W.J.: Constructing simple nonograms of varying difficulty. *Pure Mathematics and Applications (Pu. MA)* **20**, 1–15 (2009)
2. Batenburg, K.J., Kusters, W.A.: On the difficulty of nonograms. *ICGA journal* **35**(4), 195–205 (2012)
3. Cazenave, T.: Nested Monte-Carlo Search. In: Boutilier, C. (ed.) *IJCAI*. pp. 456–461 (2009)
4. Gelly, S., Silver, D.: Monte-carlo tree search and rapid action value estimation in computer go. *Artif. Intell.* **175**(11), 1856–1875 (2011). <https://doi.org/10.1016/j.artint.2011.03.007>, <https://doi.org/10.1016/j.artint.2011.03.007>
5. Genheden, S., Thakkar, A., Chadimová, V., Reymond, J.L., Engkvist, O., Bjerrum, E.: AiZynthFinder: a fast, robust and flexible open-source software for retrosynthetic planning. *Journal of Cheminformatics* **12**(1), 70 (Dec 2020). <https://doi.org/10.1186/s13321-020-00472-1>, <https://jcheminf.biomedcentral.com/articles/10.1186/s13321-020-00472-1>
6. Kocsis, L., Szepesvári, C.: Bandit based Monte-Carlo planning. In: *17th European Conference on Machine Learning (ECML'06)*. LNCS, vol. 4212, pp. 282–293. Springer (2006)
7. Mehrabian, A., Anand, A., Kim, H., Sonnerat, N., Balog, M., Comanici, G., Berariu, T., Lee, A., Ruoss, A., Bulanova, A., et al.: Finding increasingly large extremal graphs with alphazero and tabu search. *arXiv preprint arXiv:2311.03583* (2023)

8. Rosin, C.D.: Nested rollout policy adaptation for monte carlo tree search. In: In IJCAI. pp. 649–654 (2011)
9. Roucairol, M., Cazenave, T.: Refutation of spectral graph theory conjectures with monte carlo search. In: International Computing and Combinatorics Conference. pp. 162–176. Springer (2022)
10. Roucairol, M., Cazenave, T.: Solving the hydrophobic-polar model with nested monte carlo search. In: International Conference on Computational Collective Intelligence. pp. 619–631. Springer (2023)
11. Silver, D., Huang, A., Maddison, C.J., Guez, A., Sifre, L., Driessche, G.v.d., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., Hassabis, D.: Mastering the game of Go with deep neural networks and tree search. *Nature* **529**, 484–489 (2016)
12. Wiggers, W., van Bergen, W.: A comparison of a genetic algorithm and a depth first search algorithm applied to japanese nonograms. In: Twente student conference on IT. Citeseer (2004)