Proceedings of the Thirtieth International Conference on Automated Planning and Scheduling (ICAPS 2020)

Solving the Longest Simple Path Problem with Heuristic Search

Yossi Cohen,[1] **Roni Stern,**[1,2] **Ariel Felner**[1]
[1]Ben Gurion University of the Negev, SISE Dept., Be'er Sheva, Israel
[2]Palo Alto Research Center (PARC), SSL, Palo Alto, USA
{cohenyossi81, roni.stern}@gmail.com, felner@bgu.ac.il

Abstract

Prior approaches for finding the longest simple path (LSP) in a graph used constraints solvers and genetic algorithms. In this work, we solve the LSP problem with heuristic search. We first introduce several methods for pruning dominated path prefixes. Then, we propose several admissible heuristic functions for this problem. Experimental results demonstrate the large impact of the proposed heuristics and pruning rules.

1 Introduction and Background

In the longest simple path (LSP) problem the aim is to find the longest *simple path* (where no node is visited more than once) between two given states in a graph. LSP is a fundamental problem in graph theory, that is known to be NP-hard, and even hard to approximate within a constant factor (Karger, Motwani, and Ramkumar 1997). Nevertheless, LSP can be solved in polynomial time for square grids without obstacles (Keshavarz-Kohjerdi, Bagheri, and Asgharian-Sardroud 2012). The motivation to solve LSP comes from a variety of domains such as information retrieval on peer to peer networks (Wong, Lau, and King 2005), estimating the worst packet delay of Switched Ethernet network (Schmidt and Schmidt 2010), multi-robot patrolling (Portugal and Rocha 2010), and VLSI design where the longest path should be found between two components on a printed circuit board (Chen 2016).

Some prior work compiled a given LSP problem to a constraint optimization problem and used a constraints solver (Pham and Deville 2012). Others used genetic algorithms (Portugal, Antunes, and Rocha 2010). As far as we know, two prior works approached LSP as a heuristic search problem. Stern et al. (2014) showed how to modify common heuristic search algorithms that were designed for minimization (MIN) problems to solve maximization (MAX) problems. They used LSP to demonstrate their results and proposed an admissible heuristic for LSP. Palombo et al. (2015) proposed several admissible heuristics for solving the *Snake-in-the-box* (SIB) problem (Kautz 1958). SIB is a reminiscent of LSP that is important for a useful type of efficient error correction codes.

Copyright © 2020, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

In this paper, we build on these two works for solving LSP with heuristic search. We first propose several methods to detect and prune states that are *dominated* by other states. We then show how to integrate these pruning methods into A* (Hart, Nilsson, and Raphael 1968) and Depth-First Branch-and-Bound (DFBnB). Then, we propose several admissible heuristics for LSP as well as a preprocessing phase that reduces the search effort. Experimental results on both uniform- and non-uniform cost grid maps demonstrate substantial speedups of our approaches.

2 A* and DFBnB for LSP

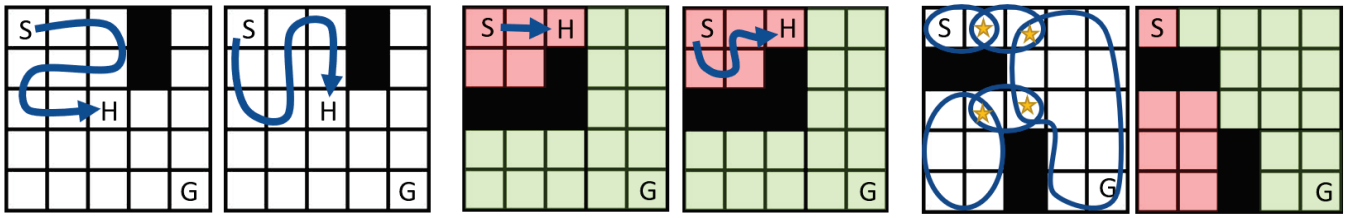
A path in a graph is called *simple* if it never passes through the same vertex twice. The *Longest Simple Path* (LSP) on a directed graph $G = (V, E)$, a start vertex $s \in V$, and a goal vertex $g \in V$ is a simple path from s to g such that no other simple path from s to g is longer. We focus on LSP in which the graph is explicitly given as input.

A fundamental difference between shortest path (SP) and LSP is that the shortest path must be simple, as otherwise, a shorter path would exist (note that the underlying graph is not weighted). By contrast, the longest path may not be simple. While finding the *longest* (not necessarily simple) path in a graph can be done in time polynomial, finding the longest *simple* path (i.e., LSP) is NP-Hard (Karger, Motwani, and Ramkumar 1997).

To formulate LSP as a search problem, one needs to first define a corresponding state space. For SP, a state represents a vertex. This is not sufficient for LSP, since to know which operators are applicable at a vertex n one must know the vertices in the path from s to n to verify they are not added twice to the resulting path. Following Stern et al. (2014) a state in the LSP state-space will be denoted by N. $N.\pi$ represents a graph-path in the underlying graph G from s to $N.head$ which is the last vertex in $N.\pi$. $N.tail$ will denote all other vertices in $N.\pi$. The applicable operators from an LSP state N correspond to extending $N.\pi$ with a single edge.

The following modifications to textbook A* are needed to adapt it to MAX problems and LSP (Stern et al. 2014).

Admissibility in MAX problems. A function h is said to be admissible for MAX problems iff for every state N in the search space it holds that $h(N)$ is *larger than or equal to* the



(a) Two symmetric states according to BSD. (b) RDD dominance (c) The BCC and the branch to the goal

Figure 1: Examples of the pruning methods concepts and BCC heuristic on a grid map, S is the start vertex

remaining optimal cost to the goal (the length of the longest simple path from $N.head$ to g in the case of LSP).

Choosing from OPEN. In MIN problems, A* pops from OPEN in every iteration the state N with the lowest $g(N) + h(N)$. In MAX problems, A* pops from OPEN in every iteration the state N with the highest $g(N) + h(N)$.

In MAX problems in general, the A* halting condition must also change, since the f value of a goal may be larger than its g value. However, in LSP there is a single goal and one can only visit it once, and so the f and g values of the goal are equal. Thus, choosing a goal for expansion is a sufficient stopping condition.

When using Depth-first Branch and Bound (DFBnB) for MAX problems a state N is pruned only if $g(N) + h(N) \leq$ the cost of the incumbent solution (Stern et al. 2014).

Solving LSP with A* or DFBnB raises several challenges. First, while for an underlying graph $G = (V, E)$, the SP state-space is linear in $|V|$ the LSP state-space can be exponential in $|V|$. For example, in a $n \times n$ square grid there are n^2 states but $O(2^n)$ different graph-paths from the top-left corner to the bottom-right corner. Second, the LSP state-space is a tree and every node has a unique path. Thus, the duplicate detection mechanism used by A* and other algorithms is not applicable. To address this, we propose more aggressive path pruning mechanisms for LSP. Finally, most prior work in developing admissible heuristics were for MIN problems. Standard techniques such as PDBs (Culberson and Schaeffer 1998; Felner, Korf, and Hanan 2004), delete relaxation (Hoffmann and Nebel 2001), true distance heuristics (Sturtevant et al. 2009) are all designed for MIN problems. We address the challenge by developing an effective admissible heuristic for LSP.

3 Pruning Dominated States

While searching for an LSP we often encounter nodes that can be pruned without losing optimally. We say that a state N *dominates* N‘ if N enables to prune N‘. Pruning dominated states is a well-known technique to speed up search algorithms (Ibaraki 1977). We cover such pruning methods for LSP next.

(1) Basic Symmetry Detection. We say that two nodes N and N' are *symmetric* if $N.head = N'.head$ and their tails cover exactly the same set of cells, not necessarily in the same order. Figure 1a shows an example of two symmetric paths. For every pair of symmetric states N and N‘, it holds that N dominates N‘ and vice versa. Therefore, if a search generates two symmetric nodes, it can safely prune one of them. This dominance detection method is called *Basic Symmetry Detection* (BSD) method. BSD implementation can be done naively such that every possible head location h points to a list of nodes that have h as their head. Then, a linear match between the lists of two tails should be done. To speed-up performance, we have implemented BSD with a fast non-cryptographic hashing function named FNV (Fowler, Noll, and Vo 1991) for $N.\pi$ and only compared nodes with the same hash values.

(2) Reachable Dominance Detection (RDD). For a given LSP state N, we can partition the vertices in the underlying graph into three separate sets. **(1)** The set of *visited* cells ($N.\pi$). **(2)** The set of *reachable cells* ($N.R$) that may be visited by a path that is an extension of $N.\pi$. **(3)** The remaining cells are the *blocked cells* ($N.B$) – those that may never be visited by a path that is an extension of $N.\pi$.

Corollary 1. *State N dominates state N' if the following conditions hold (1) $N.head = N'.head$, (2) $|N.\pi| \geq |N'.\pi|$, and (3) $N'.R \subseteq N.R$.*

Figure 1b shows an example of two LSP states, that satisfy the dominance condition given in Corollary 1. The green area represents the $N.R$ vertices and the red area represents the $N.\pi$ and $N.B$. As can be seen, all conditions of Corollary 1 are valid. Thus, RDD will prune the right-hand state. Implementing RDD efficiently requires fast set inclusion computation. In our implementation, we used a simple linear data structure to check the dominance over the reachable states. Indeed, this causes RDD to be sometimes slower than BSD, as can be seen in our experimental results.

Pruning Dominated States During Search In A*, when a child state C is generated, we search for a state N in OPEN or CLOSED that may dominate C or vice versa. If N dominates C then C is not added to OPEN. If C dominates N then C is added to OPEN but N is thrown away. Observe that in RDD the dominance relation is asymmetric, that is, one state dominates the other and not vice versa. For DFBnB we add a *transposition table* that contains all generated states to match against any newly generated state. In DFBnB, however, RDD can only prune a new generated dominated state but cannot prune an old state that is dominated. This cuts down the effectiveness of RDD by half on average therefore we are skipping this combination for DFBnB in our experiments.

4 Heuristics

In this section, we describe several admissible heuristic functions for solving LSP problems.

(1) The Reachable Heuristic (Stern et al. 2014). For a state N consider the *reachable cells* $N.R$ defined above. Any of these cells might further appear in a valid simple path. Thus, $N.R$ can serve as an admissible heuristic (upper bound) for state N. This heuristic is called the *reachable heuristic* (h_R). It returns 16 for both states in Figure 1b, as there are 16 reachable vertices (the green area). To compute h_R for a state N, we run a simple DFS starting from $N.head$ spanning all reachable vertices.

(2) The Bi-Connected Components Heuristic (Palombo et al. 2015). A bi-connected graph is a graph that remains connected after removing any single vertex. An equivalent definition is that every two vertices have at least two vertex-disjoint paths connecting them. A *bi-connected component* (BCC) of a graph G is a maximal sub-graph of G that is bi-connected. A *cut-point* is a vertex that belongs to two different bi-connected components. The resulting set of cut-points and BCC form a *Block-Cutpoint-Tree* (BCT) that can be constructed in linear time (Harary and Prins 1966; Hopcroft and Tarjan 1973). Two bi-connected components cannot have more than a single cut-point connecting them. Thus, once a graph-path passes a cut-point, it can never return to that bi-connected component. Therefore, any simple graph-path passes through a single branch in the BCT. In particular, any solution to an LSP problem must pass through the BCT branch that starts in the BCC that contains s and ends in the BCC that contains g. The *BCC heuristic*($h_{BCC}(N)$) returns the total number of reachable vertices in each of BCC along this BCT branch, starting from the BCC that contains the $N.head$. This heuristic was used to solve SIB (Palombo et al. 2015). Figure 1c demonstrate $h_{BCC}(N)$, assuming that $N.head = S$. Components are marked with blue lines and cut points with yellow stars. Given this BCT, we can identify that the cells on the bottom-left area are not on the LSP from s to g. The cells that are on the BCT branch that leads from s to g are marked in green in the right frame. h_{BCC} counts only these vertices, returning a value of 14. H_R will count 20 hence BCC is better in this case because it is tighter upper bound.

The BCC formulation can also be used to prune states. This is done by identifying in a preprocessing phase all the cells that are *not* on the BCT branch leading to the goal cell, and considering these vertices as blocked cells. We found experimentally that doing this preprocessing proved beneficial, saving an average of 25% of the number of expanded states, and saving approximately 30% of the CPU time. Thus, all of our experiments below included this method.

(3) Alternate Steps Heuristic. The following novel heuristic is applicable for cases where the underlying graph can be represented as a bipartite graph, e.g., 4-connected grid and SIB. Like a board of chess, grid cells can be divided into black and white. A path along the board must alternate between white and black cells. For an LSP state N, let Δ be the difference between the number of white cells and black cells in the set of reachable vertices. Assume $N.head$ is white. If the goal is black, we can subtract Δ from the number of reachable grid cells (i.e., h_R). If the goal is white, then we subtract $\Delta - 1$ from h_R. We denote the resulting heuristic by h_{R+ALT}, i.e., $h_{R+ALT}(N) = h_R(N) - \Delta$ if $N.head$ and the goal are of the same color and $h_{R+ALT}(N) = h_R(N) - \Delta + 1$ otherwise.

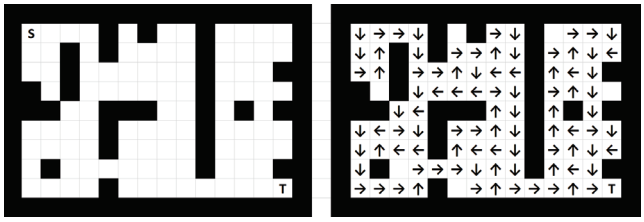


Figure 2: Room map and its solution ($C^* = 91$)

There are two ways to apply this enhancement on top of BCC: **(1)** Apply it on the entire set of all vertices in all bi-connected components of the branch in the BCT. This is called BCC+alternate ($h_{BCC+ALT}$). **(2)** Apply this enhancement separately for each bi-connected component and then add them up. This is called BCC+separate+alternate ($h_{BCC+S+ALT}$).

5 Experimental Results

In this section, we compare experimentally A* and DF-BnB using the proposed pruning methods and the proposed heuristics. The underlying graphs used in our experiments are based on *4-connected grids* with uniform cost and *life grids* that are the same grids but traversing an edge into a cell (x,y) costs y+1 (Thayer and Ruml 2008). We performed two main groups of experiments: short-time experiments (10 minutes) with all the combinations of algorithms and long experiments (1 hour) on large instances that will compare only the naive baseline to our best method. Using BCC preprocessing was always better than not using it and we always report results with BCC preprocessing.

(1) Short 10Min. Experiments In this set of experiments, we generated open grids with all integer combinations of sizes between 5x5 and 7x8 (total of 9 combinations). To each of these grids, we randomly set 4%, 8%, 12%, 16% cells as obstacles and created 10 random problem instances, yielding a total of 40 instances per gird size. In total, we had 360 random instances. A timeout of 10 minutes was set for solving each problem instance. Another set of problem instances was on *Room Maps* depicted in Figure 2 where every grid contains multiple rooms connected via narrow doors. The doors are randomly positioned and inside the rooms, there can be a few random obstacles. The number of rooms in our grids varies between 3x2 and 5x6, where the room size was between 2x2 and 5x5. 400 such room maps were created for our experiments. Table 1 shows the results averaged over all the instances of random grids and room maps that were solved by all of the different algorithms that was tested. There is a column for each of the pruning methods: no pruning (NP, BSD, and RDD) and a row for each of the

Heuristic	Expanded					Runtime				
	A*			DFBnB		A*			DFBnB	
	NP	BSD	RDD	NP	BSD	NP	BSD	RDD	NP	BSD
	Grids with random obstacles									
R	45,211	21,815	16,494	49,772	24,823	10,808	2,626	17,576	601	448
R+ALT	34,271	15,708	14,051	38,100	18,286	6,155	1417	14,386	463	338
BCC	8,366	2,703	2,271	9,623	3,447	377	110	294	195	105
BCC+ALT	7,491	2,187	2,077	8,651	2,869	300	**82**	262	166	86
BCC+S+ALT	7,348	**2,097**	**2,025**	8,499	2,771	311	**86**	261	188	94
	Room maps									
R	30,477	18,658	3,549	74,987	41,726	4,343	1,628	3411	1727	1403
R+ALT	23,865	13,817	3,324	62,534	32,201	3,504	1,158	3,262	1,392	1,050
BCC	10,935	5,577	1,618	34,059	17,428	1,400	714	1,069	2,619	1,789
BCC+ALT	8,731	4,063	1,516	29,114	13,679	1,090	512	1,008	2,113	1,344
BCC+S+ALT	6,135	2,326	**1,433**	22,047	9,020	596	**222**	1,005	1,295	732

Table 1: Expanded states and runtime (sec.) grids with obstacles.

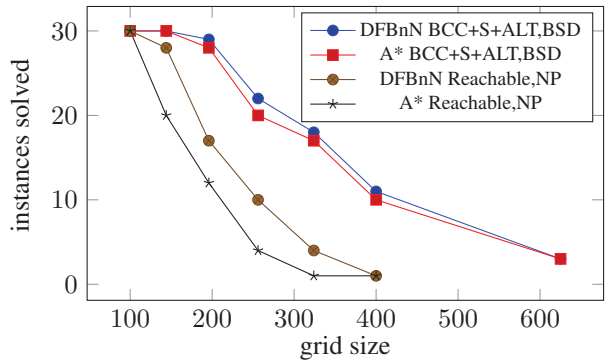


Figure 3: Success rate for the 4-connected grid.

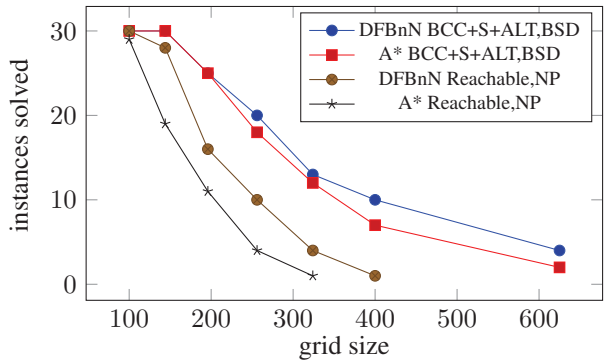


Figure 4: Success rate life grid.

heuristics: R, R+ALT, BCC, BCC+ALT, and BCC+S+ALT. The best two variants are highlighted in bold. Clearly, our strong heuristics and pruning method outperformed the simplest method by a factor of 20 in nodes and by a factor of 130 in time for random obstacles grids and factor 20 in time for room maps. As could be expected the best heuristic was BCC+S+ALT. The best pruning method was RDD in terms of nodes expanded. But BSD was better in time because of the simple data-structure is used.

(2) 1 Hour Experiments In our 1-hour time limit experiment, we only compared the most efficient heuristics $h_{BCC+S+ALT}$ and pruning method (BSD) with the baseline of H_R and no pruning (NP). We generated grids of size 10x10 up to 40x40 with 25%, 30%, 35% of obstacles. Figures 3 and 4 demonstrate the success rate of these variants on all the problem instances on *4-connected grid* and the *life grid*, respectively. In both cases, DFBnB with BCC+ALT when BSD was used had the best success rate.

To summarize the results, we observe the following. First, the BCC heuristic is always better than Reachable. Second, adding the alternating step on top of the BCC heuristic is helpful, but not by a large margin. Third, it is always worthwhile to perform the BSD prunning. The RDD pruning always saves node expansions, but its runtime is sometimes worse than that of BSD. Forth, on the small domains, A* and DfBnB were quite similar. However, on the large domains, DFBnB outperformed A*. So, DFBnB with BSD and the BCC+S+ALT heuristic is the best variant we have and we recommend using it.

6 Conclusion and Future Work

In this paper, we proposed several techniques for pruning states that are dominated by other states, and a range of admissible heuristics for solving LSP. Our proposed pruning methods and heuristics improve the ability to solve LSP problems in a reasonable time and proves the necessity to provide strong heuristics and pruning methods.

Future work will continue in the following directions. (1) We focused on finding an optimal solution to LSP. Future work can explore how to tradeoff optimality for runtime. (2) Our pruning methods, and in particular RDD, can be time-consuming. Future work may develop more efficient data structures that will allow fast searching of dominated states. (3) A new dynamic programming algorithm for LSP (Fieger et al. 2019) was proposed and deserves a deeper comparison.

7 Acknowledgments

This research was supported by the Israel Ministry of Science, the ISF grants #844/17 to Ariel Felner and #210/17 to Roni Stern.

References

Chen, W. 2016. *The VLSI Handbook, Second Edition*. Electrical Engineering Handbook. CRC Press. 77-31.

Culberson, J. C., and Schaeffer, J. 1998. Pattern databases. *Computational Intelligence* 14(3):318–334.

Felner, A.; Korf, R. E.; and Hanan, S. 2004. Additive pattern database heuristics. *Journal of Artificial Intelligence Research* 22:279–318.

Fieger, K.; Balyo, T.; Schulz, C.; and Schreiber, D. 2019. Finding optimal longest paths by dynamic programming in parallel. In *Proceedings of the Twelfth International Symposium on Combinatorial Search, SOCS 2019, Napa, California, 16-17 July 2019*, 61–69. AAAI Press.

Fowler, G.; Noll, L. C.; and Vo, P. 1991. Fowler / noll / vo (FNV) hash, retrieved from http://isthe. com/chongo/tech/-comp/fnv.

Harary, F., and Prins, G. 1966. The block-cutpoint-tree of a graph. *Publ. Math. Debrecen* 13(103-107):19.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics* 4(2):100–107.

Hoffmann, J., and Nebel, B. 2001. The ff planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research* 14:253–302.

Hopcroft, J., and Tarjan, R. 1973. Algorithm 447: efficient algorithms for graph manipulation. *Communications of the ACM* 16(6):372–378.

Ibaraki, T. 1977. The power of dominance relations in branch-and-bound algorithms. *Journal of the ACM (JACM)* 24(2):264–279.

Karger, D.; Motwani, R.; and Ramkumar, G. D. 1997. On approximating the longest path in a graph. *Algorithmica* 18(1):82–98.

Kautz, W. H. 1958. Unit-distance error-checking codes. *j-IRE-TRANS-ELEC-COMPUT* EC-7(2):179–180.

Keshavarz-Kohjerdi, F.; Bagheri, A.; and Asgharian-Sardroud, A. 2012. A linear-time algorithm for the longest path problem in rectangular grid graphs. *Discrete Applied Mathematics* 160:210–217.

Palombo, A.; Stern, R.; Puzis, R.; Felner, A.; Kiesel, S.; and Ruml, W. 2015. Solving the snake in the box problem with heuristic search: First results. In *Symposium on Combinatorial Search (SoCS)*, 96–104.

Pham, Q., and Deville, Y. 2012. Solving the longest simple path problem with constraint-based techniques. In *Integration of AI and OR Techniques in Contraint Programming for Combinatorial Optimzation Problems - 9th International Conference, CPAIOR 2012, Nantes, France, May 28 - June1, 2012. Proceedings*, 292–306.

Portugal, D., and Rocha, R. 2010. Msp algorithm: Multi-robot patrolling based on territory allocation using balanced graph partitioning. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, 1271–1276. New York, NY, USA: ACM.

Portugal, D.; Antunes, C. H.; and Rocha, R. P. 2010. A study of genetic algorithms for approximating the longest path in generic graphs. In *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics, Istanbul, Turkey, 10-13 October 2010*, 2539–2544.

Schmidt, K., and Schmidt, E. G. 2010. A longest-path problem for evaluating the worst-case packet delay of switched ethernet. In *SIES*, 205–208. IEEE.

Stern, R.; Kiesel, S.; Puzis, R.; Felner, A.; and Ruml, W. 2014. Max is more than min: Solving maximization problems with heuristic search. In *Symposium on Combinatorial Search (SOCS)*.

Sturtevant, N. R.; Felner, A.; Barrer, M.; Schaeffer, J.; and Burch, N. 2009. Memory-based heuristics for explicit state spaces. In *International Joint Conference on Artificial Intelligence (IJCAI)*.

Thayer, J. T., and Ruml, W. 2008. Faster than weighted a*: An optimistic approach to bounded suboptimal search. In *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling, ICAPS 2008, Sydney, Australia, September 14-18, 2008*, 355–362.

Wong, W. Y.; Lau, T. P.; and King, I. 2005. Information retrieval in p2p networks using genetic algorithm. In *Special Interest Tracks and Posters of the 14th International Conference on World Wide Web*, WWW '05, 922–923. New York, NY, USA: ACM.