Evaluating the Reverse Greedy Algorithm

Shady Copty
shady@il.ibm.com

Shmuel Ur
ur@il.ibm.com

Elad Yom Tov
yomtov@il.ibm.com

April 7, 2004

Abstract

This paper present two meta heuristics, reverse greedy and future aware greedy, which are variants of the greedy algorithm. Both are based on the observation that guessing the impact of future selections is useful for the current selection. While the greedy algorithm makes the best local selection given the past, future aware greedy makes the best local selection given the past and the estimated future, and reverse greedy executes a number of greedy iterations and chooses the last one as the next choice. Future aware greedy depends on a future aware utility function which is problem specific. While we have found such utility functions for the set cover problem this paper concentrate on reverse greedy whose description is truly problem independent.

Both algorithms suggested, while not quite as efficient computationally as the greedy algorithm, are still very efficient. We show interesting problems on which the greedy algorithm has been extensively studied where the suggested algorithms outperform greedy. We also show a problem with different characteristics on which the greedy algorithm is better and try to categorize the kind of problems for which future aware and reverse greedy are expected to yield good result.

1 Introduction

Many optimization problems fall under the following description: the problem has a set of solutions which may be applied in any order, together with a monotonic utility function on subsets of the solutions, U. Given two subsets of solutions A and B if $A \supset B$, $U(A) \geq U(B)$. In addition, the utility function is usually such that for a given solution x, $U(x|A) \leq U(x|B)$. The optimization problem is to find the best utility for a solution of given cardinality. Optimization problems that fall under this description include the set cover problem with many of its variants [15, 7], the facility location problems [13, 12], the maximum coverage problem [11], and many others.

The greedy algorithm chooses solutions, one at a time, such that each solution, when chosen, gives maximum improvement to the utility function, until the cardinality of solutions is reached. The types of utility functions under which the greedy algorithm is optimal [20, 18] has been extensively studied. For example, if the utility function of a solution is independent of previously chosen solutions, then the greedy algorithm is optimal. For other problems, while greedy may not be optimal, it still is the algorithm of choice [9, 8, 16, 1, 17]. This is because the greedy algorithm is computationally efficient and tends to yield good results. There are many studies on the greedy algorithms for such problems that show upper and lower bounds [19, 10], as well as other properties.

The greedy algorithm, as its name suggest, is a very short-sighted algorithm. It always looks for the best short-term gain. The greedy algorithm gets very good performance by ignoring global optimizations which are very expensive. However, the greedy algorithm also ignores available relevant information that exists in the problem formulation, namely the cardinality of the solution set. We present here a new variant on the greedy algorithm, which we call reverse greedy or RGreedy, which takes that cardinality into account. We demonstrate, using experiments, that a large class of problems exists on which RGreedy outperforms the greedy algorithms. We explain our intuition regarding the kind of problems for which it is advantageous to use RGreedy, as well as situations where it does not help or actually hinders.

We created the RGreedy algorithm after studying a variant of the set cover problem[3]. In that paper we also defined future aware greedy - FWGreedy. FWGreedy is characterized by a modified weight of the objective function according to what is likely to occur in the future. For example, in the set cover problem, we use the problem formulation and the knowledge of how many more sets will be selected to estimate, for each task, the probability that it will be found in the future. We then greedily choose the set that finds the most tasks that have not been seen in the previous sets and that are less likely to be observed in the future.

Both RGreedy and FWGreedy have initially smaller utility than the greedy algorithm. Our prediction was that if RGreedy and FWGreedy use the knowledge of cost in an efficient way, then their utility will surpass that of the greedy algorithm when we get close to the last set. Indeed, as Figure 1 shows for the selection of 10 solutions, RGreedy surpasses Greedy close to the end. The initial experimental results have encouraged us to examine RGreedy-type of algorithms further and see how they work on other problems and how they can be improved.

Section 2 shows the intuition behind RGreedy and sample problems on which it works well. We then explain simple variants that may improve the result. Section 3 contains a number of experiments which show problems on which RGreedy works well. Some initial tuning work for RGreedy is also shown. Section 4 shows an experiment on a problem for which RGreedy is not suitable. It also tries to generalize the type of problems for which we do not expect this approach to work. Section 5 contains concluding remarks, as well as suggested avenues for future work.

2 Description and motivation for the RGreedy algorithm

Our original motivation for the RGreedy algorithm came when trying to optimize probabilistic regression suites [3, 6]. Denote by $\mathbf{t} = \{t_1, \ldots, t_n\}$ the set of tasks to be covered. Let $\mathbf{s} = \{s_1, \ldots, s_k\}$ be a set of heuristics for which statistical coverage predictors exist. The probability of covering the task t_j using heuristic s_i is denoted P^i_j. The probabilistic resource constraint regression suite problem is to choose, for any given number, a set of heuristics such that the number of expected distinct task seen is maximized.

The greedy algorithm for the probabilistic regression suite problem chooses heuristics, one at the time, such that each heuristic when chosen increases the objective function by the maximum amount. For each task the increase in the objective function is the probability of the heuristics to cover that tasks multiplied by the probability that it was not observed by prior heuristics.

The greedy algorithm for constructing probabilistic regression suites with limited resources provides efficient and high-quality regression suites, but these suites are usually not optimal. One reason for the sub-optimality of the greedy algorithm is that it does not consider future steps in the algorithm. Specifically, the greedy algorithm ignores the contribution of the heuristics selected in future steps to the overall coverage. As a result, the greedy algorithm may select a heuristic with a high probability of covering a given task, ignoring the fact that this task will be covered with high probability in the future, even without the selected heuristic.

We define FWGreedy as an algorithm that chooses solutions, one at a time, such that each solution when chosen gives maximum improvements to the future aware utility function, which takes the future into account, as well as the previously used solutions.

For example, consider a simple case, where the goal is to maximize the coverage of two tasks (T_1 and T_2) with 10 test cases selected from two heuristics (H_1 and H_2) and the coverage probability matrix shown in Table 1.

	$\mathbf{T_1}$	$\mathbf{T_2}$
$\mathbf{H_1}$	0.80	0.00
$\mathbf{H_2}$	0.30	0.05

Table 1: Coverage probability matrix example

When the greedy algorithm is used, heuristic H_1 in the first step of the algorithm is used because of its contribution to the coverage of T_1. For the same reason, the greedy algorithm also selects H_1 in the second step. After the second step, the probability of covering T_1 is high enough (0.96), such that the contribution of H_1 to its coverage in future steps is small. Therefore, the contribution of H_2 to the coverage of T_2 is dominant in the next 8 steps. The resulting regression suite created by the greedy algorithm is $W = \{2,8\}$, with an average coverage of 1.3343. The progress of the average coverage for the greedy algorithm is shown in Figure 1.

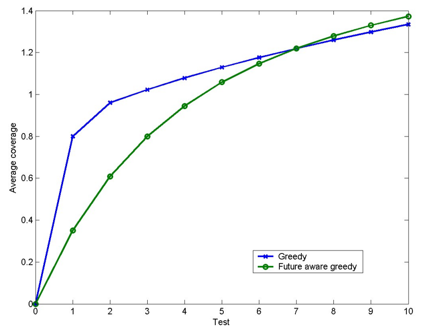

Figure 1: Progress of the greedy and future-aware greedy algorithms

The greedy algorithm ignores the fact that at each step the probability of covering T_1 increases regardless of the heuristics used. Even if H_2 is used in all 10 test cases, the probability of covering T_1 is $1-(1-0.3)^{10} = 0.9718$. Given this information, the contribution of selecting H_1 becomes much lower, and H_2 becomes the preferred heuristic. The resulting regression suite in this case is $W = \{0,10\}$, with an average coverage of 1.3730 (see the dashed line in Figure 1).

The performance of the greedy algorithm can be improved by considering, at each step of the algorithm, not only the probability that a task is covered in previous steps of the algorithm, but also the probability that the task will be covered by future steps. This improvement is possible only if the size of the regression suite or an estimate of it are known in advance. Otherwise, prediction of the future is impossible, because the future may not exist (i.e., we are in the last step).

The basic greedy algorithm looks for the heuristic that maximizes the coverage after the $k+i$'th step, given the heuristics used in the previous k steps. That is, in each step the goal is to maximize

$$\arg\max_i \sum_j (1-S_j)P_j^i,$$

where S_j is the probability of covering task j in the previous steps and P_j^i is the probability of covering task T_j using heuristic H_i. The future-aware greedy algorithm replaces this goal function with

$$\arg\max_i \sum_j (1-S_j)P_j^i(1-F_j), \tag{1}$$

where F_j is an estimation of the probability of covering task T_j in future steps.

The quality of the estimation of F_j affects the quality of the solution provided by the future-aware algorithm. It is easy to show that exact knowledge of F_j leads to an optimal solution. The problem is that exactly calculating F_j is as hard as providing an optimal solution to the probabilistic regression suite. An optimistic estimation of F_j may degrade the quality of the solution, since it may unnecessarily punish good heuristics because of an overly optimistic future. In the extreme case, if we use $F_j = 1$, the greedy algorithm is reduced to a random selection of heuristics.

When a good method for estimating F_j is used, the future-aware algorithm should perform better than the greedy algorithm. But if we look at coverage progress as function of the step in the algorithm, the greedy algorithm should perform better than the future-aware greedy in the early steps. This happens because the greedy algorithm tries to maximize the current gain, while the future-aware algorithm looks at a farther horizon. Figure 1 illustrates this. In general, we expect the future-aware algorithm for n steps to perform better than the future-aware algorithm for m steps, $m > n$, after n steps.

In our experiments we examined several methods specific to the set cover problem of estimating the future to be used by FWGreedy. However, as the future aware utility function is problem-specific, we did not generalize this idea to other problems.

We did create one generic algorithm – RGreedy – that takes the estimation of the future into account. Instead of estimating the future, RGreedy chooses in the future! This is accomplished by running the regular greedy algorithm to completion but instead of choosing the solution for the first step, the solution that was chosen for the last step is selected and the process is repeated. The intuition is by reversing, we choose the solution that is chosen with the most knowledge. We start by working on the hard parts and when we get to the easy parts we work with full knowledge of the impact of the other solutions on the problem. RGreedy is fairly efficient, if n is the number of solutions to be chosen then the cost of RGreedy is at most $n/2$ times the cost of the greedy algorithm.

In this work we have evaluated the RGreedy algorithm on a number of problems in order to develop intuition on the types of problem on which it will work. We also checked if RGreedy could be improved by running the algorithm a fraction of the way to completion (half or three quarters for example) and choosing there. The intuition is similar to that of classification algorithms that try to avoid over-classification that can be caused by overly training a classifier on training data through methods such as cross-validation [5].

We denote by $RGreedy(\kappa = 0.5)$, for example, an implementation of the greedy algorithm in which at every step we execute the greedy algorithm half the way to the end and choose the last one. So if the number of heuristics to be executed is 20 in the first step we run the greedy algorithm 10 times and choose the tenth while after the twelfth step we run it four times (half way to 20) and choose the last.

3 Description and experimental results for the RGreedy algorithm

We demonstrate using two problems the advantage of the RGreedy algorithm over the greedy algorithm. For each of the two problems, different settings are investigated. In each setting, a large number of random instances is generated as input to the different algorithms. We evaluate the performance of each algorithm in comparison to the others. For each setting, an algorithm's performance is the number of times it achieved the best result, normalized by the number of executions - 10000 for the first experiment, and 1000 for the second. Note that for the same instance, two algorithms could achieve the best result. We compare the results of algorithms to each other, since we don't know the optimal result, as it is computationally hard to calculate. We collected other measures such as the frequency at which an algorithm achieved the best result exclusively, and how often the algorithm achieved the best result divided by the number of other algorithms that achieved the same result in the same experiment. These measures were omitted since they show the same trends as the simple measure.

3.1 Probabilistic Regression Suites with Limited Resources

Automated regression suites are essential in testing large applications while maintaining reasonable quality and timetables. The main objection to automation of tests, in addition to the cost of creation and maintenance, is the observation that if you run the exact same test many times it becomes a lot less likely to find bugs. To alleviate those problems, a new regression suite practice, which uses random test generators to create regression suites on-the-fly, is becoming more common. In this regression practice, instead of maintaining tests, regression suites are generated on-the-fly by choosing several specifications and generating a number of tests from each.

Given N tasks and K test specifications, in which each test specification covers each task in a given probability, choose M test specifications to cover the greatest number of tasks. This is obviously an instance of a probabilistic set cover which is an NP hard problem [7]. We discuss the variant with $K = N$, were each specification was originally written to target a certain task, but also in some probability hits other tasks. This is not guaranteed in the general form of the probabilistic set cover problem. In the case that the probability for each specification to hit tasks other than its targeted task is zero, RGreedy and the greedy algorithm yield the same outcome. The reason for this is that once a test specification is chosen, it no longer yields additional benefit, and so the only difference is the reverse order.

Copty at el. described the greedy algorithm and an RGreedy algorithm for this problem [3]. The greedy algorithm, chooses a test specification for each step that maximizes the expected coverage, given preceding choices. The RGreedy algorithm for the problem runs the greedy algorithm in each step with κ fraction of the number of test specifications left to choose, and picks the last test specification as the current chosen test specification.

To show the advantage RGreedy has over the greedy algorithm, we conducted the following experiment: For each given number of tasks, we randomly generate probability matrices, that conform with the requirement that each test specification targets a specific task and hits other tasks in a certain probability. In table2 an example of such a matrix, where each row represents a test specification and each column represent a task, the i-th test specification targets the i-th task. We executed each instance of the problem against the greedy algorithm and all the variations of the RGreedy algorithm. Notice that we had to generate matrices that were hard enough for the algorithms, yet with easy matrices that posed no challenge in terms of limiting the resources. In these instances both the greedy algorithm and RGreedy would yield the same result in most cases, since there is no meaning in being future aware. In this case the greedy algorithm is preferred because it costs less to yield the same result.

	Tasks				
Test Specs	0.50	0.05	0.55	0.01	0.20
	0.32	0.30	0.54	0.03	0.30
	0.21	0.14	0.60	0.04	0.35
	0.10	0.15	0.56	0.10	0.20
	0.00	0.17	0.55	0.01	0.70

Table 2: Test specification matrix

3.1 describes the performance achieved by RGreedy for different κ in comparison to one another. The complexity axis is our approximation of the complexity, achieved through different settings of the experiment: Different numbers of tasks, test specifications, and test executions. Note that $RGreedy(\kappa = 0)$ is the greedy algorithm. We clearly see that RGreedy is better than the greedy algorithm for every $\kappa \neq 1.1$. The reason is that $RGreedy(\kappa = 1.1)$ falsly estimates the future according to choices that would not have been taken by the greedy algorithm. We also see that for this problem the greedy algorithm becomes worse as complexity increases. Furthermore, we notice that for part of the experiment $RGreedy(\kappa = 1.0)$ is the best algorithm while for others $RGreedy(\kappa = 0.9)$ is best, which clearly indicates that the best κ to use depends on the problem itself.

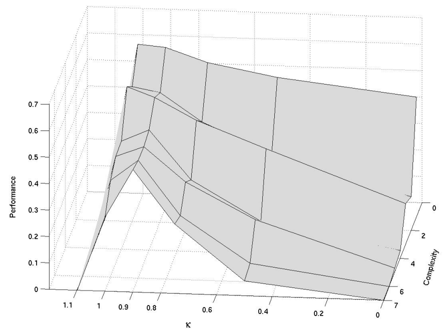

Figure 2: Comparing the performance achieved by RGreedy for different κ for the probabilistic regression problem. The complexity axis is our approximation of the complexity, achieved by different settings of the experiment. Note that $RGreedy(\kappa = 0)$ is the greedy algorithm. We clearly see that RGreedy is better than the greedy algorithm for every $\kappa \neq 1.1$. Note that for this problem the greedy algorithm becomes worse as complexity increases. Finally, for some of the experiments $RGreedy(\kappa = 1.0)$ is the best algorithm while for others $RGreedy(\kappa = 0.9)$ is best.

3.2 Facility locating problem

Given N cities, choose M cities in which to build a facility such that the sum of all minimum distances between cities and facilities is minimized. Figure 3 shows an example of such a problem with 12 cities and 4 facilities. This type of facility-locating problem is NP hard [7].

The greedy algorithm for this problem is quite trivial: Keep choosing cities to place facilities in until you reach M cities, at each step, given the preceding choices, choose the city that minimizes the sum of all minimum distances. The RGreedy algorithm for the problem is for each step to run the greedy algorithm with the number of facilities left to locate, and pick the greedy algorithms' last facility location as the current chosen facility location. In the variations that try to overcome the over-approximation problem, we run a fraction of the way to the end, and then the last city,

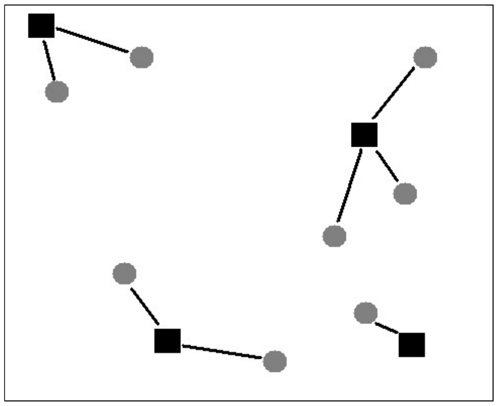

Figure 3: Cities and Facilities Example. Rectangles denote cities with facilities, circles denote cities where no facilities were placed.

$RGreedy(\kappa = 0.5)$ for example, runs the greedy algorithm half the way at each step and takes the last solution as the current solution.

Several experiments were conducted to show the advantage RGreedy has over the greedy algorithm for this problem. The number of cities was chosen to be 50 for all the experiments. The number of facilities chosen was 5, 10, 15, 20, 25, 30, 35 and 45. Since choosing where to place M facilities is like choosing where not to place $N - M$ facilities, the problems increase in complexity as they get closer to $M = 25$, and then decrease as they climb to 50. The random maps generated satisfy the triangle inequality. In this problem, we examine a different set of RGreedy variations, with $\kappa \in \{0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 1.1\}$. Again, we evaluate the performance of each of our algorithms in comparison to other algorithms.

Figure 4 shows the performance of each algorithm. Since the performance is relative to other algorithms' results, they should be examined as such. We clearly see that for the different number of facilities, $RGreedy(\kappa \leq 1.0)$ is better than that of the greedy algorithm. We again notice that the best performing RGreedy is different for different set-

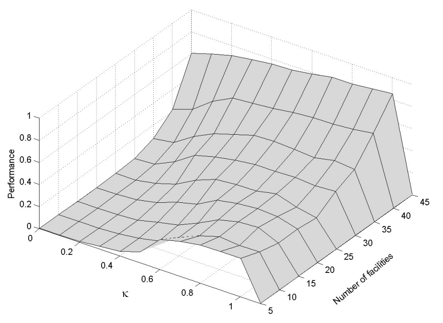

Figure 4: Comparing performance for the Facility Locating Problem. Since the performance is relative to other algorithms' results, they should be examined as such. We clearly see that for the different number of facilities, *RGreedy*$(\kappa \leq 1.0)$ is better than that of the greedy algorithm.

tings, for example the problem with $M = 25$ is best handled by *RGreedy*$(\kappa = 0.8)$ while $M = 15$ is best handled by *RGreedy*$(\kappa = 0.9)$ and $M = 10$ by *RGreedy*$(\kappa = 1)$. The behaviour in $M = 5$ originates from the little difference between the different fractions for a small horizon such as 5. As for the loss of significance for problems with $M > 25$, we relate it to the problem becoming easier and for the long horizon where over-approximation is unavoidable.

4 Matching pursuit

Matching pursuit (MP) is a method for the sub-optimal expansion of a signal in a redundant dictionary [4]. This algorithm, combined with a dictionary of Gabor functions, defines a time-frequency transformation. Matching pursuit works by iterative subtraction of the best matching dictionary functions (known as atoms) from the signal, with the appropriate amplitude and phase. Since at each iteration the best matching function is subtracted, this is a greedy algorithm.

Matching pursuit was used to expand a one-dimensional chirp signal of length 64, over a dictionary that contained approximately 21,000 Gabor functions. We ran the MP algorithm (Denoted by Greedy MP) for 20 iterations. At each iteration, after finding the best fitting atom to the signal, it's parameters were fine-tuned using 20 iterations of Nelder-Mead multidimensional unconstrained nonlinear minimization [2]. This atom was then subtracted from the remaining signal.

The RGreedy version of MP was also used for approximating the same chirp signal. The RGreedy version of MP is straightforward – Instead of using the best-fitting atom for subtraction, the atom used by Greedy MP after, for example, 50% of iterations (In the case of *RGreedy*$(\kappa = 0.5)$) is used. The amplitude and phase used for subtraction are those found for the Greedy MP.

The test was repeated with various levels of white Gaussian noise added to the chirp signal. Each variant of the algorithms was run against 10 realizations of noise at each noise level.

Figure 5 demonstrated the average energy (over 10 realizations) of the residual signal, when approximating a signal with no noise added. As can be expected, Greedy MP (RGreedy with $\kappa = 0$) reduces the residual energy more rapidly compared to RGreedy with $\kappa > 0$, which reduces most of the energy at latter iterations.

As figure 6 shows, at a given noise level Greedy MP attains a smaller residual energy compared to any variant of RGreedy MP. This effect is more prominent at lower noise levels where the difference in the residual energy of Greedy MP and RGreedy MP is larger than an order of magnitude.

This is most likely due to the fact that the dictionary contains smooth Gabor functions. Thus, Greedy MP first finds a fit for the global features of the signal, and gradually progresses to fitting more localized features. By using the atoms found in later iterations as the first atoms to be removed, the global features (That contain the most energy in the signal) are removed, and thus it is difficult for RGreedy MP to remove energy as efficiently as Greedy MP. This explains why, at high noise levels, where the smoothness of the signal is lost, the difference between Greedy MP and

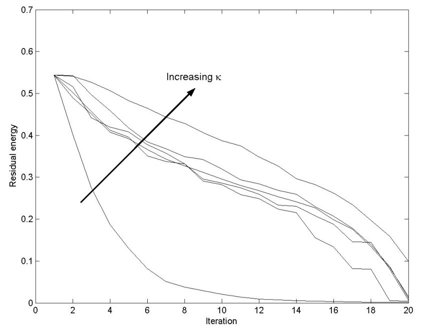

Figure 5: The residual energy of a signal through 20 iterations of the RGreedy algorithm with different parameters. This figure shows the average residual energy (Over 10 runs) of a noiseless chirp through 20 iterations of the RGreedy algorithm. Note that the RGreedy algorithm with $\kappa = 0$ is the Greedy MP (Greedy) algorithm. The residual energy decays rapidly for the first iterations when $\kappa = 0$, as opposed to a small decrease in initial iterations and a rapid decrease in later iterations when $\kappa > 0$.

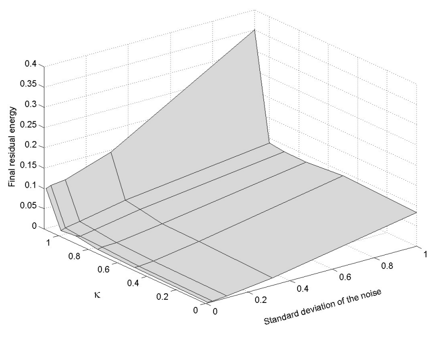

Figure 6: Comparison of the final residual energy obtained using the RGreedy algorithm with varying levels of noise added to the signal. This figure shows that as the signal becomes noisier the performance of the RGreedy algorithm with different κ values converge. Note that the RGreedy algorithm with $\kappa = 0$ is the Greedy MP (Greedy) algorithm.

RGreedy MP is less pronounced compared to the difference at low noise levels.

5 Conclusions

This paper present two meta heuristics, RGreedy and FWGreedy which are variants of the greedy algorithm. Both are based on the observation that guessing the impact of future selections is useful for the current selection. While the greedy algorithm makes the best local selection given the past, FWGreedy makes the best local selection given the past and the estimated future, and RGreedy executes a number of greedy iterations and chooses the last one as the next choice. FWGreedy depends on a future aware utility function which is problem-specific. While we found such utility functions for the set cover problem [3, 6], we decided to concentrate our checks on RGreedy, whose description is truly problem-independent.

Many problems exist which contain hard and easy components. It is a common strategy to try to solve the hard parts first and then deal with the rest. For example, the greedy implementation of the set cover problem contains an initial step in which all the subsets that contain an element which only they cover are selected. Another example is packing, in which items that are hard to fit are selected first. An even more extreme example is in [14], where two playing strategies for Othello were compared - A greedy strategy and an approach that tried to figure out the important aspects of the game and concentrate on them. In a game between the two the latter will be in a losing position all the way to the very end, where the situation will dramatically reverse. Pure greedy algorithms, due to their preference of quantity over quality, tend to miss those harder cases. RGreedy and FWGreedy both give preference to the harder problems and start with them. Both contain a built-in mechanism, based on the cardinality of the solution set to be chosen, which calibrates the selection of solutions whose contribution is not too small for the selected cardinality. The very esthetically pleasing result is that the utility function of the solutions suggested surpass that of the greedy algorithm only close to the solution cardinality, for any given cardinality.

We selected three problems, which were extensively studied, on which to compare RGreedy to the greedy algorithm. We showed that in two of these instances, set cover and facility location, RGreedy outperforms the greedy algorithm. But on one of them, the matching pursuit, greedy is better. We think that the difference is that the matching pursuit problem when solved using RGreedy has the property that applying small corrections first, gives it non smooth characteristic that detract from the first solutions. This is similar to packing first the small items and then being unable to pack the hard to fit items. On the set cover problem and the facility location problem we found that choosing first the harder solutions improve the final result. Our conjecture is that after choosing first the hard solutions the selection between the easy solutions, of which more exist, is more efficient.

This paper has shown the promise inherent in the two meta heuristics. Further research is needed in order to find out the characteristic of problems on which RGreedy and FWGreedy are expected to out perform the greedy algorithm. We have not touched at all on lower and upper bounds. The only theoretical result we have is that if the greedy algorithm is optimal so is RGreedy, which is not a very interesting result. It will be interesting to know if there are problems on which RGreedy has a better theoretical upper (and lower) bound then greedy. Many of the proofs that show bounds on the greedy algorithm use the fact that its short sighted nature to causes it to perform poorly. Such methods will be harder to deploy on RGreedy.

Another interesting venue of research is optimization of RGreedy. We have found out that there is correlation between the "‘hardness"’ of the problem and the best κ to use. Work is needed to find out if this is indeed the case and how to predict which value of κ to use. Another, quite likely, possibility is that the optimal κ is not a constant fraction of the number of the solutions left, but maybe just a constant, or a number that could change during the calculation, being either longer or shorter in the beginning.

References

[1] E. Buchnik and S. Ur. Compacting regression-suites on-the-fly. In *Proceedings of the 4th Asia Pacific Software Engineering Conference*, December 1997.

[2] T. Coleman, M. Branch, and A. Grace. Optimization toolbox for use with matlab, 1990.

[3] S. Copty, S. Fine, S. Ur, and A. Ziv. Future aware algorithms for probabilistic regression suites. submitted to ISSRE 2004, 2004.

[4] G. Davis, S. Mallat, and Z. Zhang. Adaptive time-frequency approximation with matching pursuits. *Proceedings of the SPIE*, (2242):402–413, 1994.

[5] R. Duda, P. Hart, and D. Stork. *Pattern classification*. John Wiley and Sons, Inc, New-York, USA, 2001.

[6] S. Fine, S. Ur, and A. Ziv. A probabilistic regression suite for functional verification. DAC04, 2004.

[7] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, 1979.

[8] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel. An empirical study of regression test selection techniques. In *Proceedings of the 20th international conference on Software engineering*, pages 188–197. IEEE Computer Society, 1998.

[9] M. J. Harrold, J. A. Jones, T. Li, D. Liang, and A. Gujarathi. Regression test selection for java software. In *Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 312–326. ACM Press, 2001.

[10] K. Jain, M. Mahdian, E. Markakis, A. Saberi, and V. V. Vazirani. Greedy facility location algorithms analyzed using dual fitting with factor-revealing lp. *J. ACM*, 50(6):795–824, 2003.

[11] S. Khuller, A. Moss, and J. S. Naor. The budgeted maximum coverage problem. *Inf. Process. Lett.*, 70(1):39–45, 1999.

[12] M. R. Korupolu, C. G. Plaxton, and R. Rajaraman. Analysis of a local search heuristic for facility location problems. In *Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms*, pages 1–10. Society for Industrial and Applied Mathematics, 1998.

[13] M. Mahdian, E. Markakis, A. Saberi, and V. Vazirani. A greedy facility location algorithm analyzed using dual fitting. *Lecture Notes in Computer Science*, 2129:127–??, 2001.

[14] D. E. Moriarty and R. Miikkulainen. Evolving complex Othello strategies using marker-based genetic encoding of neural networks. Technical Report AI93-206, Department of Computer Sciences, The University of Texas at Austin, 1993.

[15] M. J. Moshkov. Greedy algorithm for set cover in context of knowledge discovery problems. In A. Skowron and M. Szczuka, editors, *Electronic Notes in Theoretical Computer Science*, volume 82. Elsevier, 2003.

[16] G. Rothermel and M. J. Harrold. A framework for evaluating regression test selection techniques. In *Proceedings of the 16th international conference on Software engineering*, pages 201–210. IEEE Computer Society Press, 1994.

[17] A. G. M. S. Elbaum and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 28(2):159–182, 2002.

[18] V. V. Shenmaier. A greedy algorithm for maximizing a linear objective function. *Discrete Appl. Math.*, 135(1-3):267–279, 2004.

[19] P. Slavik. A tight analysis of the greedy algorithm for set cover. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 435–441. ACM Press, 1996.

[20] A. Vince. A framework for the greedy algorithm. *Discrete Appl. Math.*, 121(1-3):247–260, 2002.