Evolutionary Algorithms for MAXSAT

Ernesto Garcia, Marcus Christiansen, Konstantine Mushegian

February 24, 2017

Abstract

MAXSAT, or the maximum satisfiability problem, is a version of a Satisfiability problem aimed at finding the assignment of truth values to variables that maximizes the number of satisfied clauses in CNF Boolean formulas. This paper explores two evolutionary algorithmic approaches to MAXSAT problems: Genetic Algorithms and Population Based Incremental Learning. We evaluate both approaches on a set of publicly available satisfiability problems and attempt to make a recommendation on the best approach based on different scenarios.

1 Introduction

MAXSAT, or the maximum satisfiability problem, is the problem of determining the maximum number of clauses that can be satisfied in a Boolean formula in conjunctive normal form (CNF). Setting the variables to specific values may make certain clauses true, however, may also result in other clauses being set to false. The purpose of this problem is to determine the assignment to the variables that will maximize the number of clauses that are true.

The approaches that we used to solve the MAXSAT problem were two evolutionary-inspired algorithms: the Genetic Algorithm method and Population Based Incremental Learning (PBIL). Both these methods are inspired by biological evolution, implementing processes such as reproduction, mutation, recombination, and selection. These algorithms work by generating a series of individuals, which represent possible solutions, and then have these individuals reproduce, mutate, and choose the most fit individuals, so as to achieve a population of individuals that is more fit than the previous generation.

Genetic algorithms are an optimization technique, where the solution, and thus the individuals, are strings of symbols. Reproduction and mutation in this technique happens through the recombination of parent strings and changing of the symbols of the strings. The fitness of each of the children is evaluated at each generation, where the most fit children are used to reproduce so as to approach the optimal solution. PBIL is different from genetic algorithms in the way that it uses a probability vector, where each entry in the vector corresponds to the probability that that bit in the solution will be a 1. Here, the probability

vector is used to generate new individuals, where after each generation, the probabilities in the vector are updated to shift towards the optimal solution.

Our testing consisted of iterating through different candidate parameters that each algorithm takes as input using a Python script. Both the Genetic Algorithm and the PBIL algorithm take 6 parameters as input. These parameters will be explained further in the report. Initially we wanted to have every combination of sets of parameters but calculated that this would not be feasible because of the run time so ultimately we limited our testing to the set of parameters specified in Section 5. Once we got the results for our experiments we determined the best performing set of parameters. Performance benchmarks are described in Section 5. We then measured the performance of both algorithms using the optimal parameters in order to compare them directly.

Ultimately, we propose which algorithm is best if one wants to maximize the average percentage of correct clauses, minimize the average running time or minimize the iteration at which the best result was found.

In this paper, we will further elaborate on the problem we are trying to solve and our approaches to it, as well as our results from our experiments. In Section 2, we further describe MAXSAT and the formulation of the Boolean formula that form the clauses that we are trying to satisfy. In Section 3 and 4, we discuss our Genetic Algorithm method and PBIL approach respectively to solving MAXSAT, as well as provide pseudocode for each of our algorithms. In Section 5, we will describe our experimental methodology, detailing the experiments that we ran, and our reasons behind them. In Section 6, we discuss and analyze the results our experiment. In Section 7, we discuss possible further work for our project, before providing some conclusions in Section 8.

2 MAXSAT

MAXSAT or maximum satisfiability problem is a type of Boolean Satisfiability Problem in computer science and computational theory. The Boolean Satisfiability Problem is the problem of determining whether there exists a variable assignment of variables (TRUE/FALSE) that satisfies a Boolean formula, i.e. makes the Boolean formula evaluate to TRUE. If such assignment of variables exists, then we say that the given Boolean formula is satisfiable. On the other hand, if no assignment of variables results in the evaluation of the given Boolean formula to TRUE, we call the given Boolean formula unsatisfiable.

For example, the formula "a AND b" is satisfiable, because assigning $a = TRUE$ and $b = FALSE$ will evaluate this formula to TRUE. However, the formula "a AND NOT a" is not satisfiable, since all possible assignments of the variable a (TRUE/FALSE) will evaluate the formula to FALSE.

2.1 Boolean Formula

A Boolean formula in CNF is built from boolean variables, operators AND ($\wedge$), OR ($\vee$), NOT ($\neg$) and parenthesis. Like we said below, the formula is satisfiable

if there exists an assignment of its variables that make the formula evaluate to TRUE. The Boolean Satisfiability Problem is to check whether a given formula is satisfiable or not.

The formulas discussed in this paper and used in our experiments are in CNF. CNF is a conjunction of clauses, where each clause is a disjunction of individual variables. An example of a formula in CNF can be seen below

$$(x_1 \vee \neg x_2 \vee x_3) \wedge (x_4 \vee x_5) \wedge (\neg x_6 \vee x_7 \vee x_8) \wedge ...$$

Satisfiability problems are extremely hard and there is no known algorithm that solves each one of them in a feasible amount of time, however solutions for individual satisfiability problems exist and are used in the real world.

3 Genetic Algorithms

Genetic algorithms are inspired by natural selection and genetic recombination that occur in nature and the process of evolution. In nature the objects that evolve can be plants or animals, but, for the purposes of artificial intelligence, the individuals evolving are candidate solutions to a proposed optimization problem. The algorithm begins with a population of these candidate solutions which are randomly created. They consist of a a string of characters or symbols, though they are usually represented in binary as strings of 0's and 1's. The evolution is an iterative process where a new population, referred to as a generation, is created from the prior generation of candidate solutions. This process continues until a certain number of iterations has been reached or until the solution has been found.

The new generation is produced by applying three main operators that guide the pool of solutions to a solution. The most common ones are crossover, mutation, and selection which work together to make the algorithm successful. The main crossover techniques are one-point crossover, two-point crossover, and uniform crossover. In a one-point crossover a random character is chosen in a pair of individuals and all the chromosomes past that point are completely swapped.

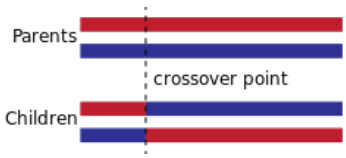

Figure 1: Two new candidate solutions after a one-point crossover.

In a two-point crossover there are two points chosen at random in the individuals. All of the chromosomes between these two points are then swapped to produce to new candidate solutions in next generation.

In a uniform crossover, there are also two solutions selected. We iterate through each corresponding bit in the solutions and swap them with a probability of 0.5

A genetic algorithm contains a fitness function which is used to evaluate the "fitness" of every candidate solution. The fitness is a quantitative measure of the effectiveness of each individual in solving the proposed problem. These fitness functions can be created in a number of ways. For our project, the fitness of an individual is equal to the percentage of clauses that each solution satisfies. The fitness of each candidate is then stored in a vector that will be used in the selection process for the next crossover phase.

In a rank selection we assign each individual a probability of being selected for the breeding pool that is proportional to its fitness. We do this in the following way. Find the sum of all the ranks each individual can have

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

where n is the size of the population of candidates. We then sort each individual from least to greatest according to its fitness and its probability of being chosen for the new breeding pool is:

$$\frac{i}{\frac{n(n+1)}{2}}$$

where i is the index of the individual after it is sorted. We iterate over the sorted individuals until our new breeding pool of size in is selected. In each iteration a random double, p, between 0 and 1 is chosen. For each individual we see if its probability is greater than p, if not then we keep track of a cumulative sum of the probabilities. Once this sum us greater than p, that individual in that iteration is selected for the breeding pool.

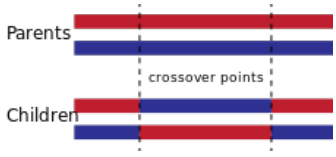

Figure 2: Two new candidate solutions after a two-point crossover.

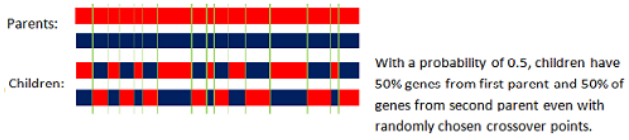

Figure 3: Two new candidate solutions after a uniform crossover.

For our tournament selection we select two random individuals from our population. We compare the fitness of these two random individuals and the one with a higher fitness is selected for our breeding pool.

For Boltzmann selection we choose the individuals for the breeding pool with the following probabilities. Note: f_i is the fitness of individual i.

$$\frac{e^{f_i}}{\sum_{j=1}^{N} e^{f_i}}$$

After the children are selected they are all considered for mutation probability, q, that is passed in as a parameter. So each child is mutated with probability q. If a child is going to be mutated then every bit in the solution is changed from 0 to 1 with a probability of 0.5.

The recombination of the good chromosomes in a fit individual is what leads to a convergence to an optimal solution. Mutation is also utilized in genetic algorithms during every iteration. Mutation works to maintain diversity in every generation and to escape premature convergence on a local optimum in the function space.

Algorithm 1 Genetic Algorithm

```
Input: Population_size, Problem_size, P_crossover, P_mutation
Output: S_best
population ← InitializePopulation(Population_size, Problem_size)
while !endCondition() do
    parents ← SelectParents(Population, Population_size)
    Children ← ∅
    for Parent_1, Parent_2 ∈ parents do
        Child_1, Child_2 ← Crossover(Parent_1, Parent_2, P_crossover)
        Children ← Mutate(Child_1, P_mutation)
        Children ← Mutate(Child_2, P_mutation)
    EvaluatePopulation(Children)
    S_best ← GetBestSolution(Children)
    population ← Replace(Population, Children)
return S_best
```

4 Population Based Incremental Learning

This is a method introduced by Shumeet Baluja from Carnegie Mellon in 1994. It is much simpler than using a genetic algorithm and is actually a combination of genetic algorithms and competitive learning. PBIL begins with a probability vector of a specified length. Every entry in the probability vector represents the probability that the corresponding bit in every candidate solution is a 1. The probability vector is initialized with every entry as 0.5 so that when the first

generation of candidate solutions is created their binary makeup is completely random. The fitness of every candidate solution in this generation is evaluated and the probabilities in the probability vector are adjusted so that they shift towards those representing the most effective solution vectors.

The only changes made in PBIL are made directly to the probability vector using a probability update rule. There is also a mutation phase where the probabilities in the probability vector are changed by a small amount with a mutation probability that is passed as a parameter. The next generation of candidate solutions is created based off of this new probability vector so that this new breeding pool is closer to the most optimal solutions in the prior generation. This process continues until the maximum number of iterations has been reached or until the solution has been found.

The pseudo-code for PBIL can be seen in Algorithm 2 below.

Algorithm 2 Population Based Incremental Learning

```
1: Input: Population_size, positiveLearningRate, negativeLearningRate,
   P_mutation, numberIterations
2: Output: BestSolution
3: P ← InitializeProbVector(P_i = 0.5)
4: population ← InitializePopulation(P_i)
5: while iterations < numIterations do
6:     WorstSolution ← EvaluateFitness(Population)
7:     BestSolution ← EvaluateFitness(Population)
8:
9:     for LENGTH of BestSolution do
10:        P_i=P_i(1 -positiveLearningRate)+BestSolution[i]*positiveLearningRate
11:
12:    for LENGTH of WorstSolution do
13:
14:        if WorstSolution ≠ BestSolution then
15:            P_i=P_i(1 -negativeLearningRate)+BestSolution[i]*negativeLearningRate
16:
17:    for LENGTH of P do
18:
19:        if Rand(0,1) < P_mutation then
20:            Mutate(P_i)
21: return S_BestSolution
```

5 Experimental Methodology

After we finished implementing both algorithms, it was our natural desire to measure the performance of our software, as well as make some guidelines for its future users. Initially we performed tests manually on a small set of hand-picked problems, however we soon realized that it would not feasible to continue

testing in this manner. Since it was our goal to benchmark the performance of both algorithms we first had to find a set of parameters for each one that resulted in the best performance. As you know from previous sections each algorithm requires 6 parameters, i.e. the search space for a "perfect" set of parameters has 6 degrees of freedom. At this point we started thinking about ways of automatizing testing in order to make our search of "perfect" parameters a little bit easier.

We decided to write a Python script that would let us run a large number of experiments on the Bowdoin College computing network, Dover. We chose Python for this because the language is simple and expressive enough that we could build a comparatively complicated script in a relatively short amount of time. The results of our automation attempts can be found in auto-test.py which included in the project repository.

Once the automated testing framework was completed we set out on our search. Below we present the parameters that we experimented with for each algorithm. We decided to try every possible combination of all parameters in order to find the best one. Each combination of parameters was run on a set of 48 problems that were pulled from the MAXSAT problems provided by Professor Majercik. These problems varied by number of variables, number of clauses, and number of variables in a every clause. We tried to include a problem from each of the provided categories in order to ensure that algorithms were tested on a diverse set of problems.

5.1 Genetic Algorithm Experimental Parameters

- Selection method:
  - Tournament selection
  - Rank selection
  - Boltzmann selection
- Crossover method:
  - 1-point crossover
  - Uniform crossover
- Crossover probability:
  - 0.6
  - 0.7
  - 0.8
- Mutation probability:
  - 0.01
  - 0.05
  - 0.1

This resulted in a total of 54 experiments, each run on a set of 48 problems, resulting in the grand total of 2,592 problem evaluations. This experiment took around 14 hours to run.

5.2 PBIL Algorithm Experimental Parameters

- Number of individuals:
  - 100
  - 175
  - 250
- Positive learning rate:
  - 0.1
  - 0.25
  - 0.4
- Mutation amount:
  - 0.05
  - 0.2
  - 0.4
- Negative learning rate:
  - 0.075
  - 0.25
  - 0.4
- Mutation probability:
  - 0.02
  - 0.1
  - 0.35

This resulted in a total of 243 experiments, each run on a set of 48 problems, resulting in the grand total of 11,664 problem evaluations. This experiment took around 36 hours to run.

5.3 Genetic Algorithm Further Testing

As running the set of 48 problems on all possible combinations of parameters took a very long time, we reduced the number of parameters that we wished to test in an effort to save time. However, this prevented us from fully testing which parameters would lead to the optimal results. In an effort to more fully explore the parameters, we wanted to see the effect of changing each parameter on the results of the algorithms. This could be done with GA, as our initial tests showed us the most successful selection and crossover method combinations, and thus, we set the most successful combination, and then varied the parameters, running these combinations on the same set of problems as above. This would give us the parameters that would optimize our GA. This could not be done for PBIL, as we could not set specific parameters from the initial testing, and running on too many parameters was not feasible from a time perspective. The parameters that we varied for our extended testing on GA was crossover probability and mutation probability.This testing took 4 hours to run and resulted in 1440 problem evaluations. Below are testing parameters we used:

- Mutation Probability:
  - 0.01
  - 0.02
  - 0.03
  - 0.04
  - 0.05
  - 0.1
- Crossover Probability:
  - 0.4
  - 0.5
  - 0.6
  - 0.7
  - 0.8

5.4 Comparing Optimized GA and PBIL

After having determined an optimal GA and PBIL from the above experiments, we finally compared these two algorithms on the same test set as above, to determine which of the two algorithms produced the best results.

6 Results

The goal of the initial experiments was to determine a set of parameters for each algorithm that results in the best performance. We measured performance based on three categories:

1. Greatest average percentage of satisfied clauses
2. Smallest average runtime
3. Least average number of iterations to get to the best one

While we will present the results for all three categories, we used the set of parameters that resulted in the highest percentage of satisfied clauses for subsequent testing.

Below we present sets of optimal parameters for each of the performance categories for each algorithm:

6.1 Genetic Algorithm Optimal Parameters

1. Greatest average percentage of satisfied clauses
   - 100 individuals in the population, tournament selection, uniform crossover, crossover probability = 0.8, mutation probability = 0.01, number of generations = 1000
   - Average Percentage of Satisfied Clauses: 91.86%
2. Smallest average runtime

- 100 individuals in the population, rank selection, 1-point crossover, crossover probability = 0.7, mutation probability = 0.1, number of generations = 1000
- Average time per problem: 12.63 seconds

3. Least average number of iterations to get to the best one
   - 100 individuals in the population, tournament selection, uniform crossover, crossover probability = 0.7, mutation probability = 0.1, number of generations = 1000
   - Average number of iterations to get to the best solution = 405

6.2 PBIL Algorithm Optimal Parameters

1. Greatest average percentage of satisfied clauses
   - 250 individuals per iteration, positive learning rate = 0.1, negative learning rate = 0.075, mutation probability = 0.1, mutation amount = 0.05, number of iterations = 1000
   - Average Percentage of Satisfied Clauses: 91.91%
2. Smallest average runtime
   - 100 individuals per iteration, positive learning rate = 0.4, negative learning rate = 0.075, mutation probability = 0.02, mutation amount = 0.05, number of iterations = 1000
   - Average time per problem: 6.67 seconds
3. Least average number of iterations to get to the best one
   - 250 individuals per iteration, positive learning rate = 0.4, negative learning rate = 0.25, mutation probability = 0.02, mutation amount = 0.05, number of iterations = 1000
   - Average number of iterations to get to the best solution = 386

6.3 Genetic Algorithm Further Testing

When further optimizing our genetic algorithm, we could have chosen to optimize it to increase the average percentage of correct clauses, to reduce the average running time, or to reduce the number of iterations were needed to get to the best result. We decided to optimize to increase the average percentage of correct clauses, as we saw this as the ultimate goal of MAXSAT problems. As seen from the above section, the combination that produced the highest average percentage of correct clauses for GA was tournament selection and uniform crossover. We set these selection and crossover methods, and then varied the

mutation probability and crossover probability with the values states in Section 5.3. Running these parameters on the 48 test cases produced the following results, presented in graphical form:

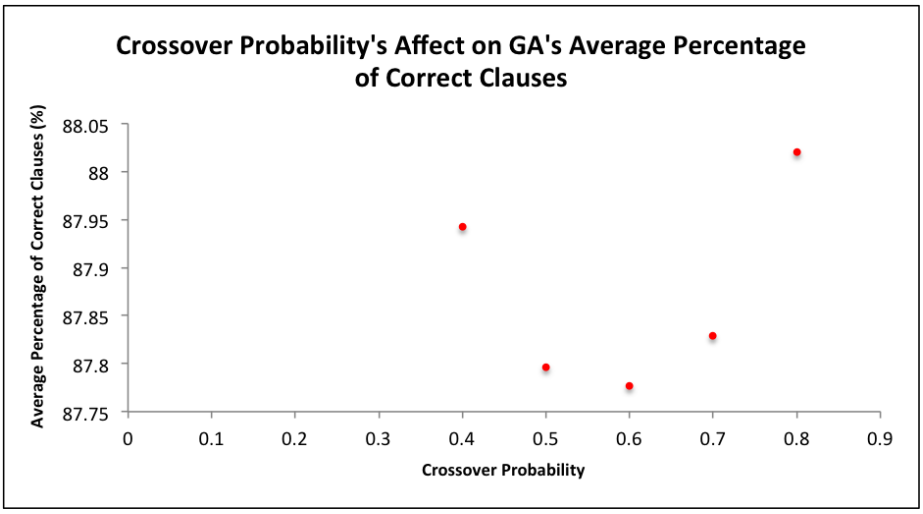

Figure 4: Crossover Probability vs.2 Average % of Correct Clauses.

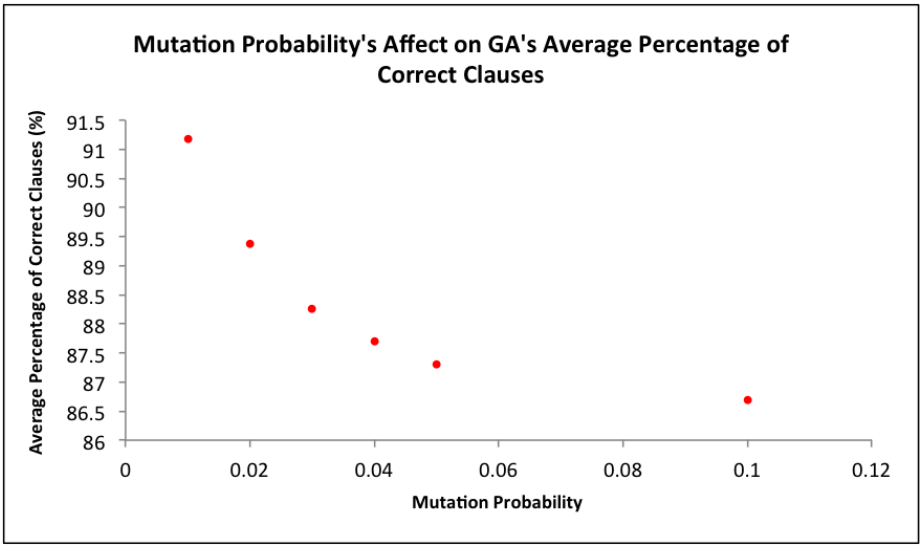

Figure 5: Mutation Probability vs. Average % of Correct Clauses.

As we can see from the graphs above, with respect to crossover probability, it appears at if both a probability over below 0.5 and greater than 0.7 are producing good results. From our results, we found that a probability of 0.8 produced the highest average of average percentage of correct clauses. With respect to the mutation probability, our graph shows that a lower mutation probability led to better results. From our results, we found that a probability of 0.01 was best. From this, we concluded that the most optimized GA parameters (for when wanting to produce the highest average of average percentage of correct clauses) is: rank selection, uniform crossover, 0.8 crossover probability and 0.01 mutation probability.

6.4 Comparing Optimized GA and PBIL

Having determined the optimized GA and PBIL parameters, we run both of these on the same 48 test cases again to determine which of the two were best. Below we present the results from running Genetic Algorithm and the PBIL algorithm with optimal parameters.

1. Genetic Algorithm
    - Average % of Satisfied Clauses: 91.6423 %
    - Average Time Per Problem: 7.1722 seconds
    - Average # of Iterations to Best Solution: 441
2. PBIL Algorithm
    - Average % of Satisfied Clauses: 91.7952 %
    - Average Time Per Problem: 16.1731 seconds
    - Average # of Iterations to Best Solution: 454

7 Further Work

We were able to successfully implement both the Genetic Algorithm and the Population Based Incremental learning algorithm over the course of the project. The implementation was quite straightforward and closely followed the pseudo-code provided in Algorithm 1 and 2.

The hardest part of the project lied in the multitude of parameters to each algorithm. Both GA and PBIL require 6 parameters, resulting in a search space with 6 degrees of freedom, as noted in Section 5, which resulted in extremely long testing times.

The next step in this project would be to implement the ability of parallel processing in order to run multiple experiments at the same time automatically and not by spawning new instances of the program manually. This would enable us to expand the search space for the "perfect" parameters while reducing the time that is necessary to perform the search.

We automatized testing to a certain extent, but did not attempt to utilize any sort of high performance computing resources. Future work would involve modifying our Python script (auto-test.py) to fully utilize available computing resources.

8 Conclusions

Over the course of this project we successfully implemented the Genetic and Population Based Incremental Learning algorithms for solving the maximum satisfiability (MAXSAT) problems. Additionally we implemented an automated testing framework that let us thoroughly analyze the performance of our algorithms and efficiently search the large parameter search space for each algorithm.

As a result of our efforts we were able to find optimal parameters for each one of the algorithms, furthermore we were able to determine optimal parameters for 3 different characteristics: highest percentage of satisfied clauses, shortest execution time, fastest convergence to the best solution. The results that we found are summarized in Section 6.

Based on our results from Section 6, we can conclude that PBIL with 250 individuals per iteration, a positive learning rate of 0.1, a negative learning rate of 0.075, a mutation probability of 0.1, a mutation amount of 0.05, and 1000 iterations produces the greatest average percentage of correct clauses across all types of MAXSAT problems provided to us. Thus, if one wanted to achieve the highest percentage, we would recommend that they run PBIL with these parameters.

We are also able to conclude that GA with 250 individuals per iteration, rank selection, uniform crossover, 0.8 crossover probability, 0.01 mutation probability and 1000 generations also generates a very high average percentage of correct clauses, however, does so much faster. Thus, if one wanted to achieve a high percentage of correct clauses, but in a shorter amount of time, we would recommend GA with the above parameters.

Finally, if one wishes to achieve the best solution in the fewest iterations, we recommend PBIL with 250 individuals per iteration, a positive learning rate of 0.4, a negative learning rate of 0.025, a mutation probability of 0.02, a mutation amount of 0.05, and 1000 iterations.

We learned a great deal about ways and benefits of writing modular, testable code. Good coding practices turned out to be incredibly useful, especially considering the amount of testing involved.

References

[1] Shumeet Baluja. Population-Based Incremental Learning: A Method for Integrating Genetic Search Based Function Optimization and Competitive Learning. June 2, 1994. https://www.ri.cmu.edu/pub_files/pub1/baluja_shumeet_1994_2/baluja_shumeet_1994_2.pdf

[2] Jason Brownlee. Clever Algorithms: Nature-Inspired Programming Recipes. January, 2011.