# CS182 Final Project

Jordan Adler and Amy Danoff
https://github.com/jojoadler/cs182finalproject

December 19, 2018

## 1 Introduction

Our project uses the Yelp dataset to find an 'optimal' restaurant itinerary based on user input. Specifically, we use three different customized CSP solution algorithms to generate an itinerary: simulated annealing, greedy min-conflicts, and backtracking with forward check using AC-3 constraint propagation. In solving this problem, we took into account multiple user-given constraints, including location, number of desired restaurants in the itinerary, restaurant category limits, and an option for unique restaurant categories. In evaluating the goodness of fit of our algorithms, we took in user input on the importance of star rating and number of reviews per restaurant, as indicated in the Yelp dataset.

In the experimentation and analysis of our work, we explore the fitness of our three different algorithms with respect to success rate, run-time, and 'fitness' rating value with respect to user-inputted weights of importance on star rating and number of reviews.

## 2 Background and Related Work

The inspiration from this project comes primarily from the discussion of 'CSPs in the Wild', or as applied to real-world problems, as discussed in the CS182 Lecture 5 on CSPs [5]. We were interested in scheduling as related to restaurant itineraries. Thus, we used the publicly available Yelp dataset as a way to model this problem.

For simulated annealing, we modified the code from the simulated annealing 'knapsack problem' homework as a basis. We updated the code significantly to include our user-specified constraints. Additionally, we used guidance from section 4.1.2 of the Russell/Norvig textbook to modify the acceptance criteria for neighbor states to take the specific rating scale of the problem into account (more detail explained in section 4) [4].

We based our min-conflicts algorithm on the specifications outlined in chapter 6 of the Russell/Norvig textbook. We updated the algorithm such that the min-conflicts heuristic was used in tandem with a greedy selection criterion. [4]

For AC-3, we formulated the algorithm based off of the pseudocode in lecture, supplemented by MIT content online to see how AC-3 would best apply to our specific Yelp CSP problem [2].

For backtracking with forward check, we looked at a paper from the Computer Science Department at the University of Ontario [1]. This helped include the forward-checking heuristic that was specific to this problem.

# 3 Approach

The following is a description of each algorithm used in this project. The main data structure used in these algorithms are dictionaries.

The output CSP for each problem is a list of 'variables', where a variable is a single meal. The domain for each variable is the set of all possible restaurants that can fill that meal spot in the itinerary. The constraints are a user inputted location and category maximums. The setup of the constraints is elaborated on in section 4.2 of this report.

## 3.1 Dataset and Setup

The first part of solving this CSP problem was familiarizing ourselves with the Yelp dataset. For the scope of this project, we limited ourselves to the 'business' dataset, which contains json objects representing almost 200,000 different businesses. From this, we filtered down our data to 57,000 'Restaurants'. For our testing, we chose to run the CSP on cities that had a significant (>50) number of restaurants. More information about the distribution of restaurants/cities in the dataset given can be found by running data_playground.py.

All code is written in Python 2 and 3. The language for each is specified in the comments of the corresponding files.

## 3.2 Constraint Modeling and User Input

For all 3 CSP-solving algorithms, we chose a set of user-inputted constraints that includes location, number of meals, and specification on the categories of restaurant that the user would like to eat at. In the Yelp dataset, each restaurant is tagged with multiple types of 'categories' that correspond to the type of restaurant. Each restaurant can have multiple tags. For example, a cafe that serves drinks in the evening may be tagged as 'Breakfast & Brunch, American (New), Restaurants, Venues & Event Spaces, Event Planning & Services, Salad' for its list of categories. For the constraints, users can input a maximum number of times that they would like to eat at each given category. Additionally, users can specify an additional constraint called 'Unique', whereby the itinerary must include entirely unique categories (i.e. no category may be included more than once).

We model these constraints as a dictionary mapping categories to numbers that serve as the 'maximum' number of times each category can be included in the itinerary. Each constraint's dictionary contains a field called 'Unique', which is mapped to a boolean specifying whether the output itinerary must be unique. Each CSP solution also takes in a city, and a number of desired meals in the output itinerary.

Another important user input of note is the use of 'weights'. This is expanded upon in the experiments and testing section, but allows the user to input 'weights' on a scale from 0-5, 5 being most important, to rate how important they consider the star ratings and the number of reviews (popularity) of their restaurants to be.

## 3.3 Simulated Annealing

As mentioned, we implemented the simulated annealing algorithm, which can be found in the simulated_annealing_yelp.py file, using the framework from the textbook, with modifications for the acceptance criterion.

The general algorithm was adapted from chapter 4 of the Russel/Norvig textbook (pseudocode can be found on page 126) [4]

This algorithm relies on two helper functions, 'Generate Neighbor' and 'Accept Neighbor'. We modified these two functions for use with our domain and constraints, and their pseudocode is as follows (on the following page of the document):

**Algorithm 1** Generate Neighbor

$CurrentState$, $Constraints$
Filter possible input items to ensure items blocked by constraints where max = 0 are not added to state
Pick new restaurant at random from filtered items and add to current state
**while** State does not satisfy constraints **do**
Delete previous items from state at random
**end while**
Return state

Note that this 'Generate Neighbor' function always returns an itinerary that does not violate category or uniqueness constraints, but may be shorter than the desired itinerary. Thus, the Simulated Annealing function will fail only if we fail to reach an itinerary of the desired length within the given number of iterations, as we begin with 0 items in the itinerary.

**Algorithm 2** Accept Neighbor

$NewState, OldState, Weights, Constraints$
**if** $NewState$ is longer than $OldState$ **then**
**return** TRUE
**else**
Calculate New state's rating

Calculate Old state's rating

**end if**
**if** $Newrating > Oldrating$ **then**
Return TRUE
**else**
Accept new state with acceptance probability $p$
**end if**

In the 'Accept Neighbor' function, the acceptance probability $p$ takes into account the ratings (which include the user weights) and the current temperature T. By specifications given in Section 4.1.2 of the Norvig textbook, $p$ should "decrease exponentially with the 'badness' of the move", and decrease as T decreases. As such, after some testing we chose

$$p = \frac{1}{e^{\frac{100-d}{T}}}$$

$$d = \frac{Oldrating - NewRating}{Sumofusergivenweights}$$

. This equation satisfies both above criterion, as $e^{\frac{100-d}{T}} >= 1$ for all our input $d$ and $T$, since $d > 0$ and $T > 0$, and this denominator decreases as $d$ and $T$ decrease.

### 3.4 Greedy Min-Conflicts

**Algorithm 3** Min-Conflicts

$CurrentState$, $NumberofMeals$, $Constraints$, $Weights$**for** Trial in Range of Trials **do**
**if** Current state satisfies constraints **then**
Return Current State

**else**
Find which variables violate constraints
Remove conflicted variable at random
Find which variables in domain have the minimum number of conflicts
Greedily add the highest rated, min-conflicted variable to current state
**end if**
**end for**
If no state fit constraints, return failure

This algorithm was adapted from Chapter 5 on CSPs of the 2005 version of the Norvig textbook, as published online by Berkeley [3]. However, it differs from that pseudocode in its greedy heuristic for selecting additional states. Because of the way constraints are defined in our domain, we define 'min-conflicted' states as all states that would not immediately violate constraints upon addition. Then, we select from these states greedily based on their ratings (with regard to user inputted weights). Additionally, note that the initial state given to the user is a greedily selected state, where we select the first $n = num\_state$ states with the highest rating. Thus, the min-conflicts algorithm will fail only if there are fewer iterations than states in the output itinerary (worst case scenario), or if there is no possible set of variables that satisfies the constraints.

### 3.5 Backtrack w/ Forward Check Using AC-3 Propagation

The following three functions comprise the backtracking algorithm. The bulk of the algorithm comes from the recursive backtrack function, which assigns each variable a value, and relies on the forward check algorithm to delete values from the domains of the neighbors of the current state that conflict with the constraints. This algorithm is further customized in that it uses an ordered domain and randomly chooses the next variable for assignment.

**Algorithm 4** Backtrack

$States, Domains, Neighbors, Constraints$
current domain ← copy of initial ordered domains
current deleted ← empty list for each meal
**return** recursive backtrack(empty assignment, problem)

The AC-3 is run before the backtracking algorithm to reduce domain size before beginning to assign variables. The AC-3 algorithm consists of 2 functions, the first called AC-3 which keeps

**Algorithm 5** Recursive Backtrack

```
Assignment, States, Domains, Neighbors, Constraints
if length of unassigned variables is 0 then
  return assignment
end if
current variable ← random variable from unassigned
for value in sorted domain of current variable do
  assignment[current variable] ← value
  forward check current variable with assigned value
  next step ← recursive backtrack current assignment
  if next step != None then
    return next step
  end if
end for
```

**Algorithm 6** ForwardCheck

```
CurrentVariable, Value, Assignment, Constraints
if current domains then
  for each meal restaurant pair in current deleted for current variable do
    add back the restaurant to the current domain of the meal
    current deleted for current var ← []
  end for
end if
for meal in neighbors do
  if meal not in assignment then
    for restaurant in the current domain of the meal do
      num ← number of categories
      if constraints are not satisfied then
        remove restaurant from current domain of the meal
        remove meal and restaurant pair current deleted for variable
      end if
    end for
  end if
end for
```

track of the arcs checked and left to check, and $Remove_Arcs$, which checks for arc-consistency. This was based primarily off of pseudocode in AIMA [4].

# 4 Experiments

In our testing, we looked at several different user inputs that could affect the run-time and fitness of the solution provided by different algorithms. A copy of the discussed test-cases, as well as a few more test-cases, can be found in test_cases.py

## 4.1 Fitness

First, we must discuss the way that fitness was measured here. The equation, referred to equivalently as 'rating' and 'fitness,' is the following:

$$Fitness = \frac{(StarWeight) \cdot \log(StarRating) + (ReviewWeight) \cdot \log(Reviews)}{TotalWeight}$$

This equation relies on user-inputted 'weights' on the importance of star rating and number of reviews (an indicator of popularity) of a restaurant. These weights are given to the function as a dictionary called weights with the keys as 'stars' and 'reviews', each mapped to an integer between 0-5 indicating the importance of the respective element, with 5 being the most important.

We chose to use log in this equation because we wanted each additional review to have a greater impact on the overall fitness rating for restaurants with less reviews than for restaurants with more reviews; the addition of an incremental review for a restaurant with very few reviews has a bigger proportional impact on the overall rating and number of review reviews than it would for a restaurant with many reviews. The same logic applies to stars.

Additionally, note that the fitness rating for each restaurant is normalized by the total weight (defined as $TotalWeight = StarWeight + ReviewWeight$). This total weight can range from 0 to 10 (although we do not test for an overall rating of 0, as this would simply output a random itinerary and thus be trivial), so itineraries can effectively be compared to other itineraries with the same input weights, regardless of other constraints.

## 4.2 Test Case Generation

When testing, we looked at how the various constraint and weight factors could influence fitness of the solution outputted: City, Number of Meals in itinerary (Number of variables), Uniqueness, Star and Review Weighting, Number of Constraints.

Thus, we formulated 7 Categories of tests, different from a basic base case:

1. Different Cities
2. Different Number of Constraints for Unique
3. Different Number of Constraints for Not Unique
4. Different Star and Review Weightings
5. Different Number of Meals for Belmont with More Constraints
6. Different Number of Meals for Phoenix with More Constraints
7. Different Number of Meals for Concord with More Constraints

In each of these 7 testing categories, we include 3 test cases that vary only along the axis specified in the category.

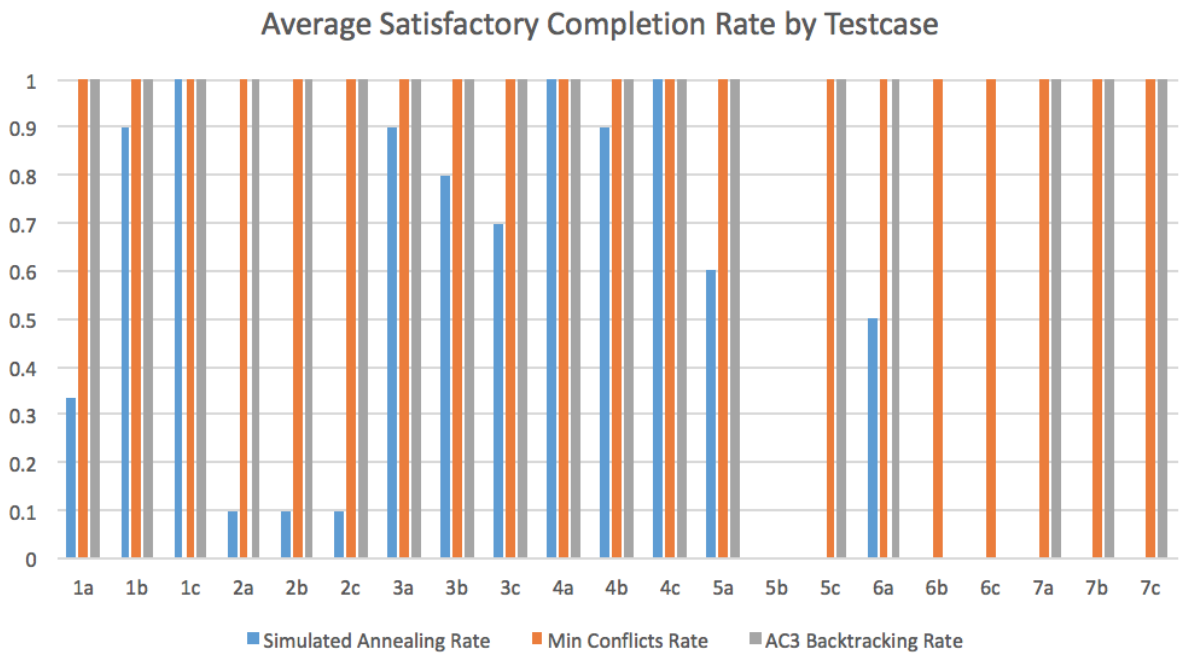
We chose the 3 cities mentioned as test cases because of their respective number of restaurants; in effect, they represent 'Small,' 'Medium,' and 'Large' domains, each with a significant amount of values in the domains. The number of restaurants for each city is as follows:
Belmont: 51 Restaurants
Concord: 336 Restaurants
Phoenix: 3832 Restaurants

### 4.3 Results

We ran the same 21 test cases on all 3 algorithms. For simulated annealing and Min-conflicts, we ran each of the test cases an average of 7 times and averaged their results in order to account for the element of randomness in these two algorithms. Thus, overall we ran 315 test cases, which we analyze below with respect to success rate, run-time, and output value (fitness rating).

The data for our discussed results can be found in the .csv and .xlsx files submitted.

#### 4.3.1 Success Rate



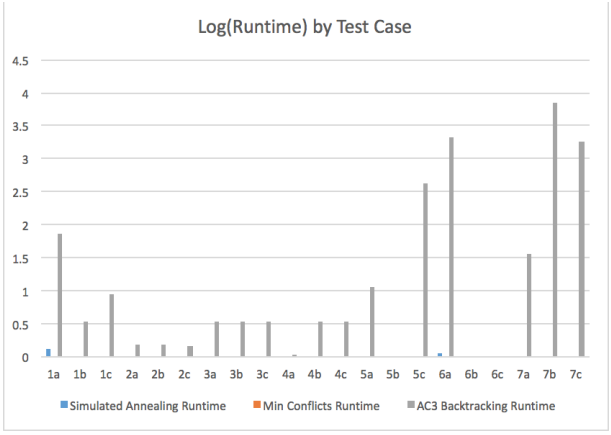

This graph represents the average success rate of the 3 algorithms. 'Success' is defined as a run of the algorithm that outputs an itinerary that meets the number of meals and all other constraints. For simulated annealing and Min-conflicts, the rate of success is an average over all the trials run for that test case (between 4 and 10 for each test case). For AC-3 Backtracking, the success rate is based on 1 trial due to the length of the trials. As noted previously, simulated annealing 'failed' if its outputted solution had fewer than the requisite number of meals. Min-conflicts failed only if it did not find a solution that satisfied constraints. We defined failure for AC-3

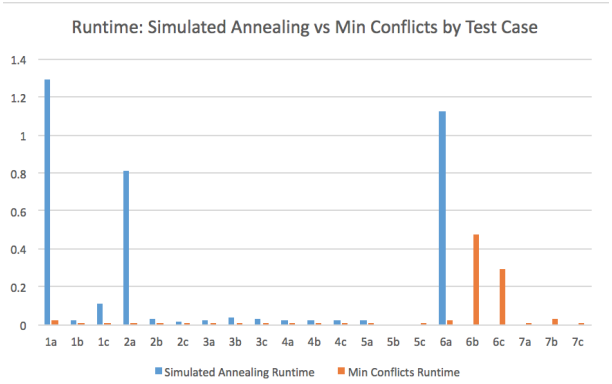
backtracking as a failure to output a correct solution or a run-time length longer than 5 hours. The success rate is on the y-axis and the respective trials are along the x-axis.

From this graph, we can see that Min-conflicts consistently had the highest completion rate, completing successfully at a rate of 100% for every test case except 5b. Note that all 3 algorithms had a success rate of 0% for test case 5b, as there is no possible solution in the domain given the constraints for this test case.

Conversely, simulated annealing had the worst success rate across the board; it completed at a rate of 100% for only 3 test cases and had a completion rate of 0% for 7 test cases, including 5b, for which all algorithms failed.

### 4.3.2 Run Time



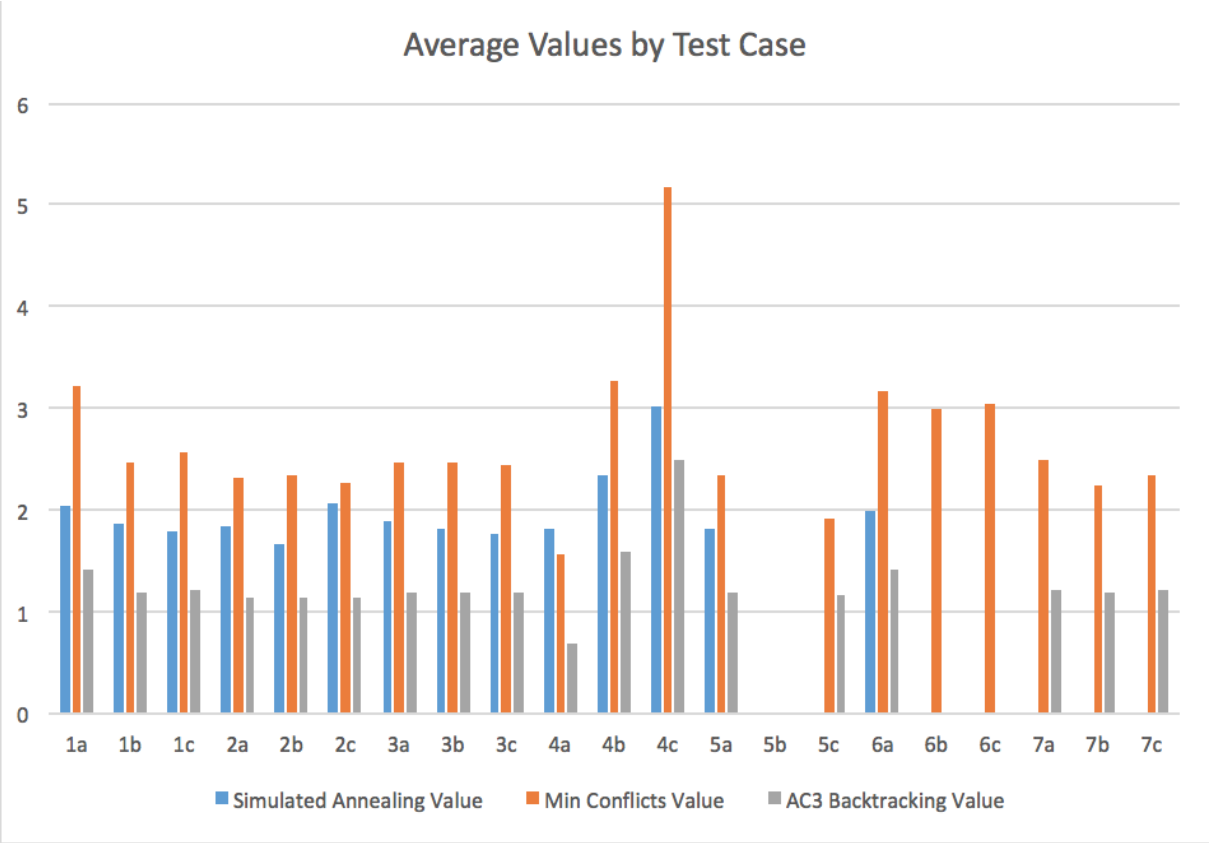

The first graph represents the respective run-times of the 3 algorithms, with the y-axis measured on a scale of max(0, log(run-time in seconds)), and the x-axis representing the respective test case. The second graph shows the run-times of just min-conflicts and simulated annealing, measured on the y-scale axis of seconds (not logarithmic).

We chose to represent run-time on a log scale in the first graph because of the outsized run-time of AC-3 backtracking. As seen here, AC-3 backtracking consistently had an exponentially larger run-time than the other algorithms. The average run-times of simulated annealing and Min-conflicts were each less than 1 second, with the exception of simulated annealing on test cases 1a and 6a. We did not include run-times for test cases that did not ultimately output a correct solution.

As for the variations among backtracking run time in the different test cases, we can see here that the size of the domain and the number of meals affect the backtracking algorithm's run time the most. As can be seen in test cases in category 3, the test cases for Phoenix, which had 3832 restaurants, took 34.55 minutes to run for 10 meals and 4 constraints, compared to Belmont with 51 restaurants that took 11 seconds to run with the everything else the same, and Concord with 336 restaurants and everything else the same, took 36 seconds to run. There is a similar trend for the test cases in categories 5, 6, and 7. Thus, we can conclude that with more restaurants in the problem, it will take (non-linearly) more time to run the algorithm.

We can also see through category 2 that an increase in the number of constraints does not increase run time.

#### 4.3.3 Output Rating Values (Fitness)




The graph above shows the relative fitness value, or rating of each of the 3 algorithms on each test case. Recall that the rating value is highly dependent on the respective weights put in by the user, so it makes most sense to compare within each test case than across test cases (i.e. comparing the fitness value of simulated annealing in test case 4c to the value of Min-conflicts in 1a does not yield pertinent information). Because of the normalization included in the rating function, higher rating values within a test case correspond to better fitness. Note that for test cases where the algorithm failed to output a successful solution, the value is set to 0.

We can see from the graph above that for every test case in which all 3 algorithms outputted at least one successful solution, Min-conflicts had the highest value, then Simulated Annealing, with AC-3 Backtracking consistently having the lowest fitness rating.

For category 1, we can see that the change in city affecting the average value of min-conflicts the most. The average value for min-conflicts correlates well with the size of the city in the test case, as 1a is Phoenix (large), 1b is Belmont (medium), and 1c is Concord (small). It thus makes logical sense that overall rating for the same constraints would be higher in a larger city, as there are more options in the domain.

Another interesting point to note is the performance (or lack thereof) of test case 6. In this section, the city Phoenix (large domain) was held constant, and the number of meals was 10 in 6a and increased to 51 and 30 in 6b and 6c, respectively. Simulated annealing and AC-3 backtracking failed to output solutions to these problems. Here, we can see that the size of the problem (particularly the number of states) greatly influences the performance of simulated annealing and AC-3 backtracking, but does not affect the performance of Min-conflicts, which

performs comparably across test case 6 despite the changing state size.

# 5 Discussion

In this section, we analyze the results of the 3 different algorithms, as well as discuss possible future optimizations for each.

## 5.1 Min-conflicts

Min-conflicts is the best algorithm across all 3 measured metrics: it is the only algorithm that consistently performed on all test-cases, it always (with the exception of 4a) had the highest value compared to the other two algorithms, and also had the shortest run time of any of the three algorithms by a large margin for every single test case. Thus, from a user perspective, min-conflicts would be the best algorithm to use, with no tradeoffs in any category.

This across-the-board success is likely due to the addition of the greedy heuristic in selecting an initial state and adding min-conflicted states. Given that the max number of substitutions this algorithm will have to make is the number of states, as we are guaranteed to remove 1 conflicting variable on each iteration, Min-conflicts rarely has to run the full number of iterations alloted, and usually ran fewer than 10 times, leading to this short run time. While we may have gotten stuck at local maximums at times with this approach, the other two algorithms never seemed to achieve a global maximum higher than this local maximum.

## 5.2 Backtracking with AC-3

Backtracking with AC-3 is clearly the slowest algorithm, growing exponentially with increases in domain and state size, as evidenced in the logarithmic run time graph. The run time of this algorithm is easily affected by the size of the problem, as evidenced by lack of results for the largest city, so it is clearly not scalable. It could never work on the world's largest metropolitan areas, like New York city, that has many times more than the number of restaurants in Phoenix. It also produces the lowest value. The values, in this case the restaurants, were ordered and assigned by the evaluation function.

One reason why backtracking with AC-3 might have a much higher run time than the other 2 algorithms is because of it's dependence on the size of the network. As we saw in class, the run-time of AC-3 is $O(n^2d^3)$ [2]. Additionally, forward checking must check all remaining values of every domain, adding exponentially to the run time of the algorithm. However, the run time of the simulated annealing and min-conflicts are less dependent on the size of the domain, as there are less instances in which the entire domain must be traversed throughout these other algorithms. In a future optimization of this problem, one of the first next steps would be to create more robust constraints that take more specific user input into account (for example, can't have Mexican twice in a row, rather than just twice overall). This would likely enhance the performance of AC-3 backtracking compared to the other algorithms, because the constraint propagation would help cut down the domain size in a way that is less applicable to the other two algorithms.

The success rate of AC-3 backtracking is relatively high, returning a result for every case except those which are too big or un-assignable, in which it will time out or return an incomplete, unacceptable solution. Thus, there is a trade-off between its success rate and its run-time,

especially with respect to simulated annealing, which consistently performed better in value but performed much worse in completion rate.

Another potential future optimization for AC-3 backtracking would be to implement the more optimal version discussed in our homework of AC-3 that had a smaller run-time. This would likely enhance the run-time performance of backtracking AC-3 against the other algorithms.

### 5.3 Simulated Annealing

Simulated annealing performed the worst out of the three algorithms in terms of success. It had a very inconsistent success rate for the majority of the test cases, which appears to be even worse compared to the two other very successful algorithms. Its run-time on all test cases was trivial compared to AC-3 backtracking, but still took more time to output a complete assignment in its successful cases than did min-conflicts. However, it had very fair scores. In one case, it even performed better than min-conflicts (test-case 4a), and consistently outputted better scores than AC-3 backtracking.

In the cases where simulated annealing failed to output a solution, this was because it returned an itinerary that did not have the requisite number of values (restaurants). As such, the failure rate is understandably higher on test cases that required a larger domain (such as test cases 6 and 7). Simulated annealing also performed quite poorly success-wise on cases where the domain was heavily constrained.

This poor success rate for larger itineraries could be explained in part with the design of the simulated annealing function. Given that the itinerary starts empty and must build up to the requisite size, with states being removed every so often, until the max number of trials has been reached, it is possible that the number of trials designated for the simulated annealing algorithm (100 trials) was simply too low to fill the larger itineraries. Thus, this problem could potentially be improved by increasing the limit of trials. However, this would certainly cause an increase in run-time as a trade-off (we found in early testing that run-time increased linearly as number of trials increased). One potential solution for a future iteration would be to dynamically increase the number of trials as the desired size of the state increases.

Additionally, there are great spikes in the run-time of simulated annealing for test-cases 1a, 2a, and 6a. What 1a and 6a have in common is that they city is Phoenix, suggesting that simulated annealing does not perform well with larger problems, and thus would be less scaleable than an algorithm like Min-conflicts, especially given that it could not perform for Phoenix size test-cases with even more variables.

Finally, it is worthy to note that on average, simulated annealing performed better than AC-3 Backtracking in terms of fitness value rating for every test case, and better than min-conflicts for 1 test case. This is likely due to the strength of the acceptance criterion; this particular version of simulated annealing moves downhill with relatively low probability. Of course, the acceptance probability $p$ could always be further tweaked to optimize this algorithm further; we picked a $p$ value based on testing on a few different test cases.

Additionally, the value of this algorithm could be further improved in 2 ways of note. Firstly, it could be improved by selecting the itinerary from the iteration with the highest value that satisfies all constraints, including length, rather than naively selecting the last output. Secondly, it could be improved by implementing a greedy heuristic in the 'Generate Neighbor' function, that is, picking a new value greedily based on the rating function instead of at random. This second modification would likely make the performance of simulated annealing more similar to the greedy min-conflicts algorithm, as it would use a similar heuristic. This modification may

even cause the simulated annealing algorithm to surpass the performance of min-conflicts, as it would be selecting effectively the same states for a given set of constraints, but potentially moving away from any local maximums towards a global maximum because of the randomness element.

### 5.4 Dataset

The data we used was limited in that we did not have access to the reviews or some other relevant information about the restaurants that may have made for interesting measures of fitness. We chose to measure fitness on star ratings and number of reviews as a way to measure popularity. However, had we had access to the reviews, we may have been able to change the evaluation function to include a sort of rudimentary sentiment analysis and give different weights to different Yelp users, for example, the reviews of frequent Yelp reviewers would get more weight, because they are more likely to be reliable. Additionally, we did not have access to price as a metric to look at. Thus, the metrics we picked for our fitness evaluation may well have not been optimal if we had used a different dataset.

### 5.5 Evaluation

The way we chose to evaluate our results biased their relative values in that those who had more preference for number of reviews over the number of stars automatically had a higher score for their solution, even after normalizing the scores. This is because the number of stars is bounded by 5, but the number of reviews has no bounds, which means that even after normalizing, the reviews added more weight to the value rating, as they were typically larger in number than the number of stars. We accounted for this in our discussion by comparing only within test cases that had the same input weightings. However, in the future, a more balanced evaluation function would be helpful to more objectively analyze the score of each solution against different test cases.

## A System Description

### A.1 Simulated annealing and Min-Conflicts

To run these systems:
1. Download the Yelp dataset at https://www.yelp.com/dataset/download

2. Change lines 9 and 10 in min_conflicts_yelp.py and simulated_annealing_yelp.py specifying the filepath of the dataset

3. Choose a test case from the 'testing' section at the bottom of either file. To view the output itinerary and corresponding information printed in your terminal for either, use the call $format\_meals(sim\_state, weights, run\_time)$ after calling $simulated\_annealing$ or $format\_meals(output, weights, run\_time)$ after calling $min\_conflicts$. You can also make up your own test case following the format of the existing test cases. Note that these files are written in Python 3.

### A.2 Backtracking with Forward Check using AC-3 Constraint Propagation

To run this system:
1. Download the Yelp dataset at https://www.yelp.com/dataset/download

2. Change line 12 in the backtracking_yelp.py specifying the filepath of the dataset

3. Choose a test case from those in *test_cases.py*

4. Execute the following function call $test\_maker(test\_casex)$ where x is the test case you would like to run, or you can create your own test case as a tuple or $(city, starweight, reviewweight, numberofmeals, constraintsdictionary)$. Note that this file is written in Python 2.

# B Group Makeup

Jojo: wrote the code for AC-3 and Backtracking.
Amy: wrote the code for Simulated Annealing and Min-conflicts.
Both: wrote test cases and contributed to the write up.

# References


[1] Fahiem Bacchus and Adam Grove. On the forward checking algorithm.

[2] Luis Ortiz. Notes on algorithms for constraint satisfaction problems, Oct 2006.

[3] STUART NORVIG PETER. RUSSELL. *ARTIFICIAL INTELLIGENCE: A Modern Approach*. PEARSON, 2005.

[4] STUART NORVIG PETER. RUSSELL. *ARTIFICIAL INTELLIGENCE: A Modern Approach*. PEARSON, 2018.

[5] Haifeng Xu. Lecture 5: Constraint satisfaction problems, Oct 2018.