An Evaluation of the Efficiency and Accuracy of Local Search Algorithms in the Context of Task Assignment and Constraint Satisfaction

Chris Smiles Serguei Balanovich Peter Bang

Final Project for Computer Science 182
Professor Barbara Grosz
Harvard College
December 11, 2013

Abstract: The allocation of tasks to a group of individuals in an effort to achieve the most optimal distribution of labor is a quite challenging yet very common problem. It can be formulated in a variety of ways and must therefore be approached from numerous angles before a coherent solution system can be constructed and scaled to larger versions of the problem. Consider Amazons Mechanical Turk marketplace, where anonymous workers select simple tasks to complete and are compensated accordingly. The task assignment problem that this project explores extrapolates upon this concept, allowing individuals self-report their skills and then matching them with an appropriate set of tasks to undertake such that the entire population of workers is performing as efficiently as possible. We model this as a constraint satisfaction problem, where most of the constraints are nonrigid but have associated weights that quantify their importance to the user. A variety of local search algorithms can then be used to find a suboptimal but low-cost solution relatively quickly. In our analysis, we found the Simulated Annealing algorithm had the best performance in terms of speed, as it very quickly narrowed down the search space while rarely getting trapped in local minima. However, it was actually Random Restart Hill Climbing that achieved the lowest-cost solutions, though the margin between the two was small. The solutions presented in this paper have a variety of broad and useful applications, and present very exciting possibilities for future work in this field: more sophisticated algorithmic analysis, an expansion of the constraint model, and the development of an intuitive system for worker self-reporting, which would allow this product to become viable for daily use.

1. INTRODUCTION

While the idea of crowdsourcing work is still considered novel, the underlying goal – efficiently allocating tasks across a large group of people workers – is hardly a new one. In most large organizations (companies, for example), this problem is solved by having each worker assigned a very specific role based on his or her skills, with each role entailing a very specific set of tasks. In contrast, todays crowdsourcing marketplaces make far fewer assumptions about workers skills, and instead let the workers complete whichever assignments they please.

Unfortunately, its often difficult to find the middle ground between these two options. Consider, for example, the leader of a student organization, and suppose this leader has a set of tasks she needs completed by the organizations members. If she were to use some sort of crowdsourcing solution, she runs the risk of her members selecting tasks that are poorly matched to their skillsets – or even worse, not selecting any tasks at all, and ultimately leaving some incomplete. On the other hand, members dont have clearly defined roles within the organization, since the types of tasks they need to complete arent consistent over time and there are simply too many members for role-assignment to be worthwhile. How, then, is the organizations leader to effectively assign the tasks?

We believe that elements of crowdsourcing and role-assignment can be combined to yield a better system for assigning tasks in this type of situation. First, instead of having workers select tasks themselves, the organizer has them self-report their skills (and whatever other attributes might be relevant to the situation) via survey, for example. Then, rather than manually assigning the tasks based on this information, the organizer needs only to define a set of preferences – or constraints – regarding how the tasks should be assigned.

For example, the organizer might specify that certain tasks require certain skills, or need a certain number of workers to be properly completed, and so on. Preferences regarding workers can also be specified – perhaps each worker can only be assigned a maximum number of tasks, or only tasks that (s)he finds enjoyable, etc. Ultimately, the organizers preferences can be as simple or as complex as the situation requires, so long as (s)he understands the demands of each task and has collected the requisite information about each worker. In our framework, these constraints are simply plugged in as necessary, and the

system returns an assignment of tasks to workers that satisfied as many of these constraints as possible – even allowing the organizer to weight some constraints more heavily than others.

How does it work? The answer lies in local search, a class of algorithms that allow us to intelligently explore the state space of task-worker assignments for better and better solutions. The next section describes the model we use to represent this state space, and to which we apply our local search algorithms later on.

2. MODEL

Though we utilize local search methods to find our solutions, the problem at hand is, at the most fundamental level, a constraint satisfaction problem (CSP). As noted in lecture, CSPs are formally defined as having three parts: a set of variables, a domain for each variable from which its value must be selected, a set of constraints specifying allowable combinations of values for certain variables. In this context, these three parts are defined as follows:

- Each variable represents an individual worker.
- The domain for a particular variable is the set of all possible task assignments for that worker. Thus, the value assigned to a variable is simply a list of the tasks that the worker has been assigned. This list can include any number (or possibly none) of the tasks specified in the problem.
- Constraints simply represent the organizers preferences for the overall assignment of tasks to workers, and (as well explain later) they are defined by the systems user as per the problems requirements. Essentially, a constraint is just a rule that specifies which values (tasks) can be assigned to some subset of variables (workers).

It is worth noting that we could have also modeled this problem using workers as variables and sets of tasks as values, and the distinction is somewhat arbitrary. We chose this representation because it simplified the calculation of certain constraints, particularly those involved with ensuring that all tasks had some number of assigned workers. With this representation, verifying it was far more simple, since we needed to simply verify the lengths of the values, or task lists per worker variable to determine the degree of satisfaction of the constraint.

As discussed in lecture, CSPs are traditionally solved using methods like basic backtracking search, which (combined with certain heuristics) can yield complete and optimal solutions in a reasonable timeframe. Ultimately, we chose to utilize local search over backtracking for several reasons. First of all, local search gives us the freedom to implement what we refer to as loose constraints. In a traditional CSP, a solution must be not only complete (assigning a value to every variable) but also consistent, meaning that all constraints are satisfied. But in our framework, these strict constraints present somewhat of a problem: put simply, wed rather have a suboptimal solution than no solution at all. Loose constraints, on the other hand, allow us to express preferences for the solution without worrying that the constraints we input might generate an unsolvable problem. (After all, our goal is to create a system that could conceivably be used on real-world problems without requiring the user to understand the system itself.)

Using local search has another advantage in that it allows us to assign relative degrees of importance to each constraint. As noted earlier, the user is allowed to assign a weight (or cost) to each constraint, which indicates to the algorithm how important it is that said constraint be satisfied in the final solution. (In contrast, since all CSP constraints are strict, they all have essentially equal importance.) In turn, this framework allows the user to be quite expressive in specifying the problem and allows constraints to be reused with varying degrees of importance depending on the context.

Finally, since one of our primary goals was to compare the performance of algorithms across a variety of inputs, local search was also attractive in that it offered a greater diversity of algorithms that could be tested (none of which had been implemented during the course). While the next section discusses these algorithms in greater detail, this discussion requires a definition of the state space with which were working. In this context, a state is just an assignment of values to variables, complete in that every worker is assigned some (possibly empty) set of tasks to do. (Well refer to states as assignments for the remainder of this report.) Our local search algorithms will explore the state space of all possible assignments and search for those that conflict least with the constraints (in accordance with their weights).

3. ALGORITHMS

Ultimately, in the interests of a more depth-oriented analysis, the following algorithms were designed and customized for the problems unique search space. Some algorithms that were originally considered for implementation, such as stochastic beam search and genetic algorithms, were omitted from the final compilation because we did not feel they were significantly different to merit inclusion. The following algorithms were implemented in full:

- Hill climbing
- Random-restart hill climbing
- Stochastic hill climbing
- Random stochastic hill climbing
- Simulated annealing
- Beam search

Before we discuss the details of these implementations, their similarities and differences, and the ways in which these are reflected in their performance, it will be important to describe the evaluation and successor functions utilized by the algorithms to traverse the search space and locate the solution.

3.1 Evaluation Function

To carry out local search, we needed a consistent way to compare the quality of individual assignments. At least in the context of applying local search to a CSP, this evaluation function usually returns the number of constraints violated in a given state; but as mentioned earlier, we wanted the user to be able to weight constraints according to their relative importance. In turn, we used a

slightly more complex evaluation function in which the total cost of an assignment was calculated using the a linear equation of the form:

$$C = c_1p_1 + c_2p_2 + ... + c_np_n$$

Thus, the total cost (C) of an assignment is simply the weighted sum of the penalties for each of the n constraints. The variables c_1 to c_n represent the weights (or costs) of individual constraints, while the variables p_1 to p_n represent the baseline penalties for violating them. The distinction between costs and penalties is subtle but important: costs are specified by the user on a trial-by-trial basis and are used to weight the importance of constraints, while penalties are hard-coded into constraints and represent baseline punishments for their violation.

At first, it might seem as if these two features are redundant. Why not give each constraint a penalty of 1, and then weight them accordingly? In short, because using dynamic penalties allows us to specify several constraints in one. Our penalties are dynamic in the sense that they usually adjust their value in proportion to the magnitude of a violation. For example, (as is discussed below) one of our constraints requires that each task be assigned no less than a predefined number of workers, and its penalty adjusts in proportion to the difference between the number of workers required for a task and the number of workers actually assigned. Thought of another way, this constraint essentially encompasses a whole class of constraints, one for each assignment of values (tasks) to variables (workers) in which too few workers are assigned to a task. Each of these constraints exerts an appropriate penalty based on how understaffed that task is, but rather than define all these constraints and penalties individually, we use dynamic penalties to do it all at once.

How, then, are costs (weights) different from penalties? In essence, they are parameters, meant to be adjusted based on the users preferences and the context of the problem. Costs (unlike penalties) are independent of the constraints definition; they simply allow the user to specify to the algorithms that, when assignments are evaluated, certain constraints are more important than others. The beauty of this distinction lies in the fact that, once a constraint is defined, it can be used repeatedly in different contexts. Its not unreasonable to assume that the too few workers constraint just mentioned is applicable to most task assignment problems, but its also plausible that the importance of this constraint will vary depending on the situation. In some cases, tasks absolutely cannot be completed without a certain number of workers; in others, this number is a bit more flexible. Ultimately, our evaluation function allows for an extensible framework that permits great expressivity on the part of the user.

3.2 Successor Function

To actually traverse the state space, the algorithms also need a successor function that, when provided a specific assignment, returns the assignments neighbors. While there are a variety of ways in which this could have been implemented, we chose one that allowed for widespread exploration without generating too many states at a time (which, as we found, can be quite computationally expensive).

As mentioned in the description of our model, each state is an assignment that allocates some set of tasks to each worker. From there, we then defined what we call a flip as the addition or removal of exactly one task from the set of tasks assigned to a worker. For each worker, there are n possible flips (one for each of the n tasks), and with m workers total, there are mn possible flips from any given assignment.

Our successor functions simply defines the neighbors of an assignment as the set of assignments that can be generated from a single flip. While searching through mn neighbors may not sound computationally difficult (it appears to have polynomial-time complexity), keep in mind each of these neighbors must be analyzed in the context of all constraints, which are often computationally expensive themselves. As we will observe later on, algorithms which look only at a single neighbor per iteration (as opposed to all neighbors) have a significant advantage in terms of speed.

3.3 Algorithmic Breakdown

Well now provide a brief description of each of the algorithms implemented. Each begins at a randomly generated initial state (or set of initial states in the case of beam search) and continues from there.

Hill Climbing Hill climbing looks at all the neighbors of the current state and moves to whichever has the lowest total cost. If it finds a local minimum (meaning none of the neighbors has lower cost), it returns the assignment.

Random-Restart Hill Climbing Random-restart hill climbing simply runs multiple iterations of hill climbing, keeping track of the lowest cost solution. The user specifies how many iterations to run, and when these are completed the lowest cost solution is returned.

Stochastic Hill Climbing Like ordinary hill climbing, stochastic hill climbing looks at all neighbors of the current assignment, but rather than transitioning to the one that offers the greatest improvement in terms of cost it selects one at random according to a probability distribution defined across neighbors. The probability of transitioning to a particular neighbor is proportional to the improvement offered by that neighbor (or the difference in cost between that state and the current state). Neighbors that dont offer any improvement have zero probability of being selected. The distribution across the remaining neighbors is simply multinomial; the probability p_x of neighbor x being selected is the difference in cost between the current state and x after normalization (division by the sum of all differences between the current state and the eligible neighbors, such that they sum to 1). This selection process makes stochastic hill climbing less greedy than ordinary hill climbing; while it still moves only to better states, it doesnt always move to the best option available. Once none of the neighbors offers any improvement in cost, the current state is returned.

Random Stochastic Hill Climbing Though the name random stochastic hill climbing is somewhat redundant, we needed a way to distinguish a second variant of stochastic hill climbing used in our system. Surprisingly, the literature was somewhat vague in its definition of stochastic hill climbing: some defined stochastic hill climbing as choosing the next state according to an appropriate distribution across successor states (in this case multinomial); others implied that the algorithm looks only at a single successor at a time, again deciding whether to accept it with probability proportional to the magnitude of the improvement. Our random stochastic hill climbing algorithm takes this latter approach, randomly selecting a single successor for consideration from the set of all neighbors. It uses an exponential function to map the cost differential between the current state and this potential successor to a probability value between 0 and 1, which then determines whether or not we transition to that state. The algorithm terminates once it has completed a certain number of iterations (specified by the user), at which point it returns the current assignment.

Simulated Annealing Our implementation of simulated annealing is actually quite similar to that of random stochastic hill climbing, using an exponential function to determine whether or not to transition to a randomly generated successor. However, simulated annealing is unique from the stochastic hill climbing algorithms in that it can transition to worse assignments with nonzero probability. The exponential function is mediated by a temperature coefficient T, which varies over time according to a predetermined schedule. While the algorithm runs, the temperature cools down and reduces the probability with which worse assignments are accepted, allowing for exploration in the early stages while still converging on an optimal or nearly optimal assignment. The user can select from any of three standard temperature schedules – exponential, fast, and Boltz – the parameters of which can be adjusted as desired. (We use the exponential schedule in our testing.) Once the temperature falls below a prespecified minimum value, the algorithm terminates and returns the current assignment.

Beam Search Our implementation of beam search keeps track of k states at a time, and calculates all the neighbors of the k states, continuing with the k best states in the next iteration. By default, we set $k = 3$. Obviously, beam search with $k = 1$ would be equivalent to a regular hill-climbing algorithm. Every iteration of beam search is obviously slower than an iteration of hill-climbing, but it can allow for a more extensive search of the state space.

4. IMPLEMENTATION

Our projects implementation is designed to be as extensible as possible. To separate domain-related logic from the algorithms actual implementation, we include classes for both tasks and workers (task.py and worker.py). The motivation behind this was simple: the user should be able to easily add whatever information about tasks and workers is relevant to his or her task assignment problem, without having to worry about how this will interface with the rest of the system.

Only the constraints (implemented in constraints.py) actually interact with the task/worker classes, accessing properties of each class as is necessary to determine whether a constraint has been satisfied. To add a new constraint, the user need only implement a function that, using whatever task/worker information is available, determines 1) whether or not the constraint has been satisfied and 2) the total cost (penalty multiplied by weight) if it has been violated.

The local search algorithms (implemented in local.py) are totally unaware of this logic; they simply call the evaluation function as necessary. (It is the evaluation function that actually calls each constraint and returns the total cost of an assignment.) Successor functions (one for generating all neighbors and another for generating a random one) are also implemented separately from the algorithms (though still within local.py), such that one could theoretically apply them to domains outside the realm of task assignment by editing only the evaluation or successor functions.

As outlined in Appendix 2, the system is run using the script project.py, which – outside of adding constraints and task/worker information – is the only file with which the user needs to interact. (While an advanced user might be interested in adjusting parameters of the algorithms in local.py, we assume the average user does not have or should not need this level of sophistication.) Within project.py, the user specifies the costs (weights) for each individual constraint in a dict object ("constraints_dict"), and constraints can be shut off simply by specifying a cost of 0. The user must also provide the paths to two CSV files that represent the problem domain – one for workers and one for tasks. From there, using the system is as simple as running project.py from the command line, supplying the algorithm to be used as the sole argument.

Its also worth noting that project.py handles the process of reading in the CSV files and generating the appropriate worker and task objects based on that information. While we include code (in csv_gen.py) that can be used to generate worker / task CSV files for use with the constraints we implemented, testing the system is probably best done using the CSV files we have already generated (discussed in the following section). Though weve strived to make extending the system as easy as possible, the process of adding additional constraints is nontrivial, since there are multiple steps (defining the constraint in constraints.py, adding the appropriate information to the classes worker.py / task.py, and editing project.py to read in this information from a CSV file).

5. ANALYSIS

5.1 Generation of Data

Unfortunately, we were not able to find real data for us to test the implemented algorithms on. Therefore, we decided to create fake data, keeping in mind three different factors: the size of the dataset, how the skillset of each worker is determined, and how the skill requirement for each task is determined. To compare the performance of the algorithms on different types of datasets, we created

a different dataset for every possible combination of the criteria listed below:

- Size of the dataset:
  - Big (100 workers, 20 tasks)
  - Small (30 workers, 10 tasks)
- Determination of each workers skill set (4 skills)
  - Increasing difficulty - probability of some worker having skill A is 100
  - Uniform difficulty - probability of some worker having any skill X is 50
- Determination of the skill requirements of each task
  - Increasing rareness - probability of some task requiring skill A is 100
  - Uniform rareness - probability of some task requiring any skill X is 50
  - One skill required - every task requires one and only one skill, chosen randomly with equal probability.

5.2 Constraints

During testing, we discovered that assigning infinity costs for strict constraints were ineffective due to the possibility that an algorithm gets stuck. For example, if the score of the current state as well as all the neighbors of the current state is infinity, then the algorithm has no information about what flip would be advantageous, and it is possible for the algorithm to travel back and forth between two states (this is the negative effect of plateaus in local search). Therefore, we made the decision to assign a much larger penalty to violated constraints that should be treated more seriously than others. For example, assigning a task to someone who does not have the required skills, or the lack-required-skill constraint, incurs a penalty of 200 per violation, compared to the too-many-tasks violation which incurs a penalty of 10.

When deciding how to calculate the penalty for not having the optimal number of workers assigned to a certain task, we consider several factors. First, having too few workers is worse than having too many workers, so we reflect this in the weights given to the constraints too_few_workers and too_many_workers, which have weights of 20 and 10, respectively. Obviously, deviating from the optimal number of workers by a large number is worse than deviating by a small number. Furthermore, not having the optimal number of workers is worse in small teams more than in big teams (e.g. 2 workers instead of 3 is worse than 6 workers instead of 7). Considering all these factors, we devised an equation for each of the constraints. For example, the equation used for too_few_workers is shown here: $penalty = weight \times (wanted - needed) * (wanted + 1)/(needed + 1)$ if $wanted > needed$, otherwise $penalty = 0$.

The logic used to determine penalties for the other constraints was similar, and can be found in the constraints.py file.

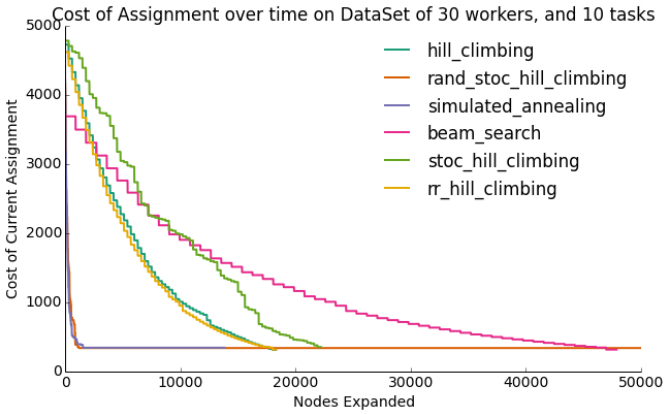
6. RESULTS

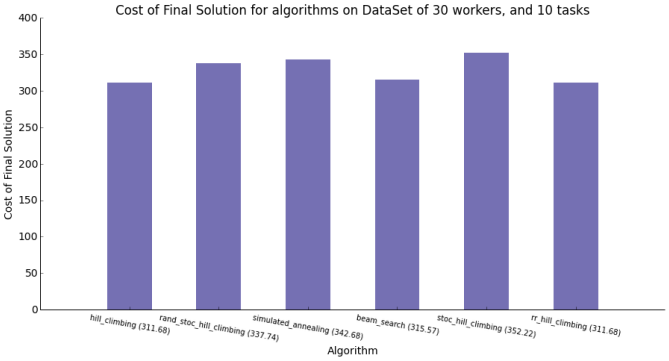
When comparing the performance of the local search algorithms, we initially considered three factors: the cost of the final solution generated, the number of states generated, and the runtime. However, the number of states generated was highly correlated to the runtime, so we can compare the results in terms of just the score of the solution generated and the number of nodes expanded.

Furthermore, we found that all algorithms excluding simulated annealing and random stochastic hill climbing were essentially ineffective on big datasets (100 workers and 20 tasks), so we were limited to analyzing just the two algorithms on the big datasets.

Below is the summary of results for the six small datasets.

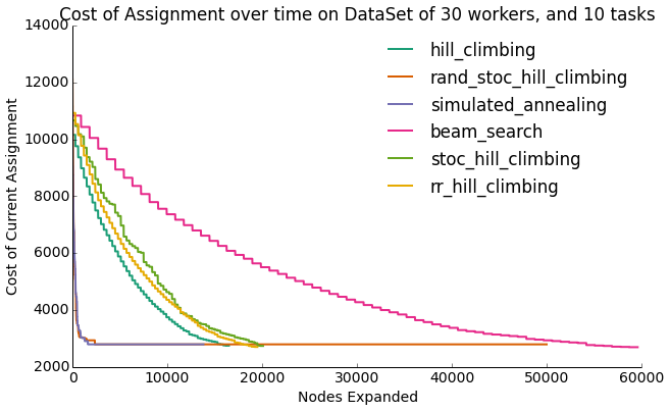
Dataset 1

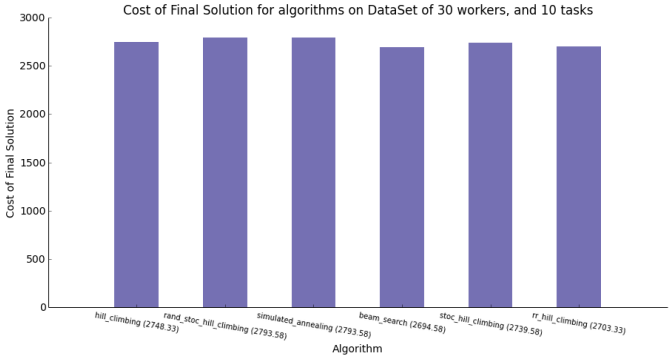




(1) Size: Small (30 workers, 10 tasks)
(2) Worker skill: Increasing difficulty
(3) Task skill: Increasing rarity

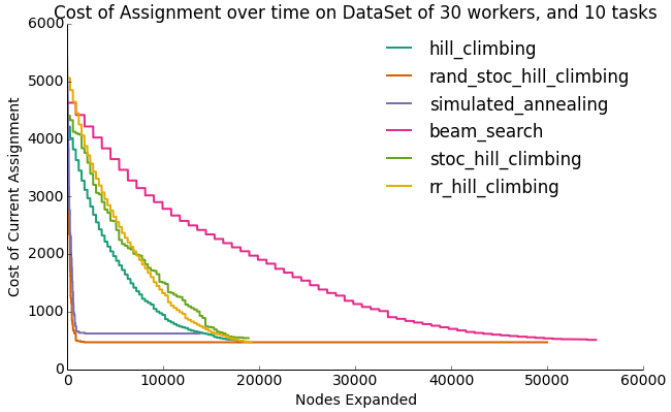
Dataset 2

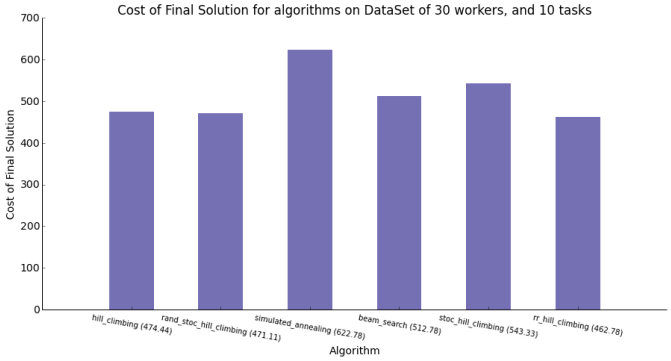




(1) Size: Small (30 workers, 10 tasks)
(2) Worker skill: Increasing difficulty
(3) Task skill: Random

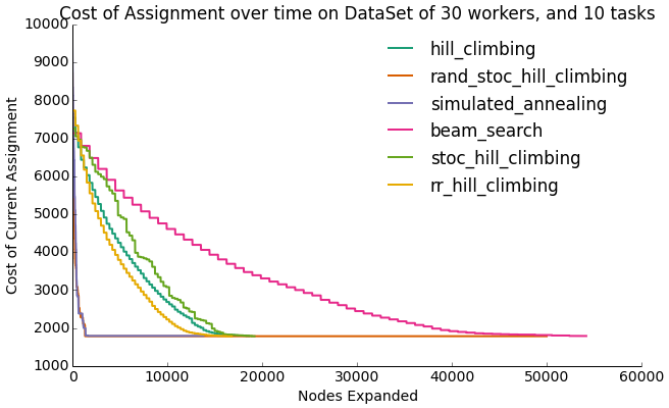
Dataset 3

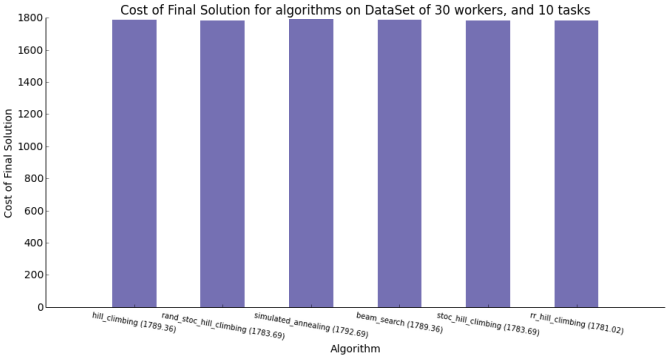




(1) Size: Small (30 workers, 10 tasks)
(2) Worker skill: Increasing difficulty
(3) Task skill: One skill per task

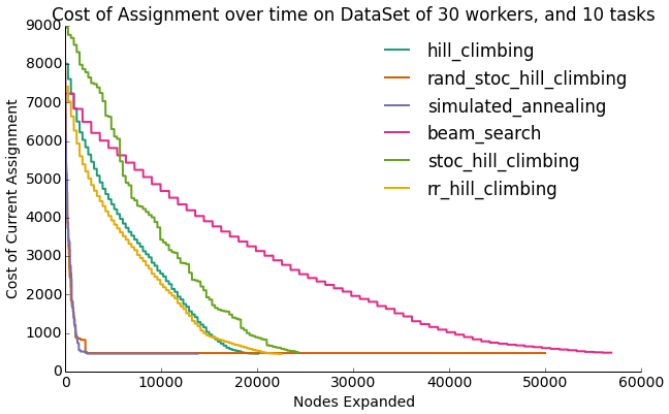
Dataset 4

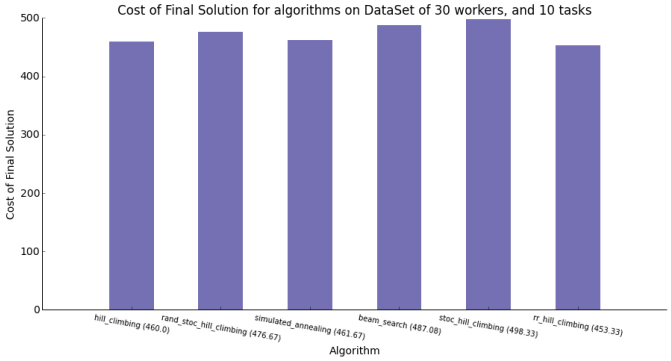




(1) Size: Small (30 workers, 10 tasks)
(2) Worker skill: Random
(3) Task skill: Increasing rarity

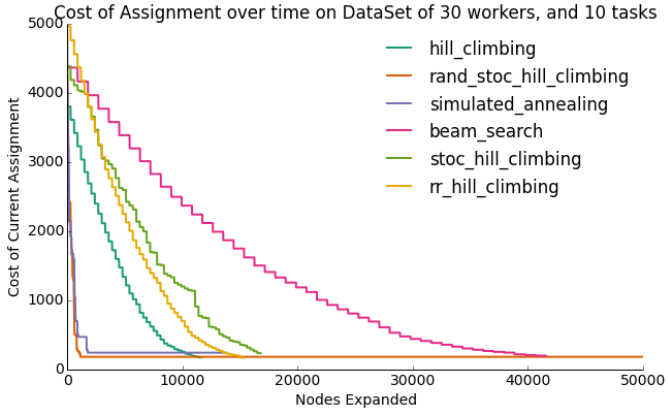
Dataset 5

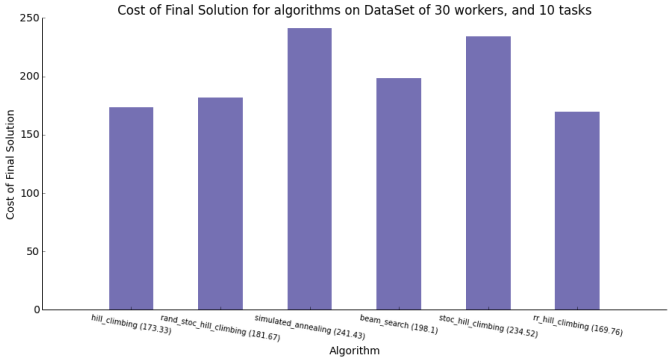




(1) Size: Small (30 workers, 10 tasks)
(2) Worker skill: Random
(3) Task skill: Random

Dataset 6

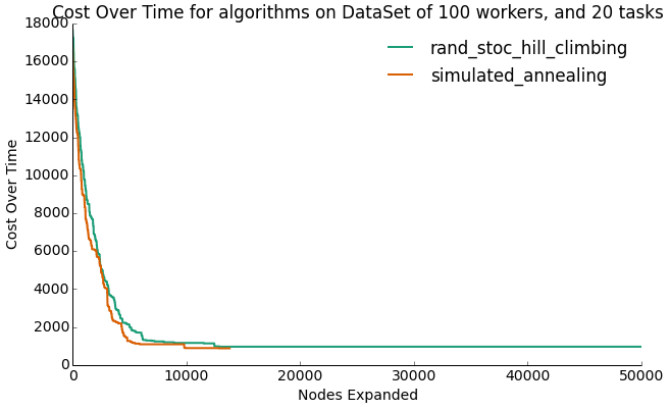




(1) Size: Small (30 workers, 10 tasks)
(2) Worker skill: Random
(3) Task skill: One skill per task

On big datasets, hill climbing, beam search, stochastic hill climbing, and random restart hill climbing were very ineffective. Below are the results for simulated annealing and random stochastic hill climbing, which were both equally effective at generating a close-to-optimal solution.

Dataset 7 (8-12 omitted, but results were nearly identical.)



(1) Size: Big (100 workers, 20 tasks)
(2) Worker skill: Increasing difficulty
(3) Task skill: Increasing rarity

Interestingly, the results were remarkably consistent for all types of datasets. This allows us to generalize the performances of each local search algorithm on this problem domain. By far the fastest algorithms were simulated annealing and its closely related cousin, random stochastic hill climbing. Both of these algorithms were able to find a close-to-optimal assignment in less than tenth of the times that other algorithms required, even in small datasets. The next fastest algorithms were hill climbing and random restart hill climbing, closely followed by stochastic hill climbing. Finally, beam search was the slowest to return a solution.

The total costs of the final solutions generated by each algorithm were within extremely close range of each other for each dataset, although there did exist small differences in the scores of the solutions returned. This shows that each of these algorithms were finding a different local minimum.

7. DISCUSSION

We can understand why there exists such a big difference in the runtime (and number of nodes generated) between the two fastest algorithms (simulated annealing and random stochastic hill climbing) and the others, especially in the big datasets. First, we see that all of the other algorithms are closely related to hill climbing, and require calculating the score of every possible neighbor from the current state on each iteration. For a problem domain with w workers and t tasks, we can flip any worker-task combination at any given iteration, so there exist $w \times t$ neighbors. Furthermore, for every neighbor, we calculate the constraints, which takes $w \times t$ per constraint. We see that every single iteration of an algorithm that is a variation of hill climbing requires at least $O(w^2t^2)$ steps, which explains why these algorithms are impractical in large problem domains.

In contrast, simulated annealing and random stochastic hill climbing calculates the score for just one neighbor state per iteration, which takes $O(wt)$ time. While simulated annealing and random stochastic hill climbing take considerably more iterations to reach the final solution, each iter-

ation is much faster, as shown above. Furthermore, there is a constant bound, determined by the weights of each constraint, on how much an algorithm can improve a given assignments score with just one flip, which means that all the extra time that the variations of hill climbing takes per iteration is very ineffective. Therefore, in this problem domain, simulated annealing and random stochastic hill climbing are the clear winners, especially if the problem domain is large.

Another interesting observation we can make is that hill climbing, random restart hill climbing, and beam search all approach the final solution steadily and uniformly, while stochastic hill climbing approaches the solution erratically. This makes sense, since stochastic hill climbing chooses which neighbor to jump to probabilistically, unlike the other three which always opt for the best neighbors.

Finally, we see that simulated annealing and random stochastic hill climbing behave very similarly, reaching the final solution at approximately the same rate and returning solutions that have similar costs. Although simulated annealing makes some moves that increase the cost of the assignment, it makes every move that decreases the cost. On the other hand, stochastic hill climbing never makes moves that increase the cost, and only moves to a better state with a random probability. Theoretically, it is possible for random stochastic hill climbing to get stuck in a local minimum, whereas it is always possible for simulated annealing to find the optimal solution with an appropriate cooling schedule. However, in this problem domain, we saw that random stochastic hill climbing always returns a solution that is just as good as what simulated annealing is able to find, implying that in this problem domain, local minima that have significantly higher cost than the optimal solution are extremely rare, if they exist at all.

8. FUTURE WORK

While very satisfied with the results of our analysis, we believe that our project also opens the door to many new lines of research. First and foremost, with additional time, we believe conducting a more sophisticated complexity analysis of our solution would be a worthwhile endeavor. As mentioned earlier, the complexity of this system is difficult to pin down in that, while the number of neighbors per assignment is static, the calculations carried out by our constraint functions have their own complexity, and in most cases these constraints must be applied to all neighbors (not just one). Our representation also results in an overall state space that is quite large, with $m2^n$ possible assignments in total (n tasks and m workers). With that in mind, algorithmic complexity is incredibly important to consider as the input size grows larger.

Another problem that remains unresolved is the issue of strict constraints. As noted earlier, strict constraints present a problem in that they can create unsolvable problems, and we assume that the user of our system doesnt have the time, knowledge, etc. to construct inputs that will necessarily have a solution. Loose constraints address this, but we lose the ability to specify that certain constraints are so important that they must *never* be violated. To remedy this issue, we experimented with constraints of infinite cost to see how the system would respond, but we quickly found that too many of these strict constraints created plateaus in our evaluation function that the algorithms had difficulty navigating. However, we did find that simply raising the cost of a certain constraint far above the costs of the others could effectively serve as a pseudo-strict constraint (in that the solutions almost always respected it as if it was strict). Still, the proper way to implement multiple strict constraints in this context remains an open question. We did hypothesize one potential approach, which would involve the use of a traditional CSP method like backtracking search to generate an initial state that does *not* violate any strict constraints, and then running local search algorithms from there. We assume that our algorithms are intelligent enough not to fall into a plateau of infinite cost if initialized in this way.

Finally, we would love to explore the possibility of making this system useful for organization leaders with real world task assignment problems. The first and most obvious step in this process would be to add a user interface through which said leaders could easily specify constraints, add information about tasks and workers (collected through surveys, perhaps directly through the interface), and quickly compute and compare solutions. While the construction of such an interface was outside the scope of this project (since we wanted to focus on the implementation and analysis of our algorithms), we do believe our current framework is flexible enough to make this addition possible – and relatively straightforward – to carry out in the future.

9. CONCLUSION

Often, when faced with the problem of some sort of systematic assignment of entities, it would seem that the task is almost trivial in nature. When a group of individuals reports its interest or ability to perform some task, at first glance it seems that the problem simply involves matching, and can be done in a short amount of time. The applications of this problem are numerous - from assigning students to classes, to determining individual contribution to a project on a small team, we are confronted on an almost daily basis with this type of challenge and must find ways to determine the best solution in order to satisfy everyone and get the project completed.

The problem, however trivial it may seem, is actually extremely challenging, especially when scaled up to very large data sets. Because there are so many possible permutations of workers with tasks, the search space is exponentially large and with every addition of workers or tasks, becomes almost impossible to store and traverse efficiently. Thus, in this search space, traditional search algorithms are of little use. Instead, it is necessary to turn to local search with a well-defined cost function and try and create the most efficient algorithm to successfully and quickly provide the sought solution.

To achieve this, a combination of Local Search and a form of Constraint Satisfaction had to be utilized. The constraints were not strictly defined, and instead were used to construct an evaluation function for the cost of each state and thereby find the assignment that either violated

the most constraints or violated the least important ones. The algorithms were limited by the number of iterations each could maximally perform and the constraints were weighted based on what was found to be most important. In order to evaluate the best search functions, much cross validation was necessary across all the different implemented functions and possible constraint weights. Ultimately, however, it was found that the most important constraint was the skillset constraint. This made intuitive sense, as assigning workers to tasks they could not do (or analogously, assigning students to classes they could not take) was the most detrimental to our final solutions. The other constraints were weighted proportionally to the degree of violation. We found that assigning less tasks than could be handled or less workers than was necessary for a given task was worse than assigning too many tasks or too many workers, so the constraints were designed to account for this as well.

For the search functions, the results were not so easily determined. It was found that the Simulated Annealing function performed fastest and found the solution in milliseconds, but because of the temperature assignment, was hindered by its attempts to verify whether it was in a local maximum. Thus, the number of nodes expanded for this search was very high and interestingly, its ultimate result was not always the best. In fact, Random Restart Hill Climbing was clearly the best at finding the solution closest to the optimal, mostly due to the fact that it was allowed to expand almost 100,000 nodes. It was by far the slowest and yet with persistence, was able to arrive at the best solution of all the algorithms. Beam search did surprisingly well in terms of efficiency as well and often had results that were just as good as the Random Restart Hill Climbing, despite expanding only half as many nodes. Clearly, each of these algorithms had its own benefits. Simulated Annealing was a great function to run on large datasets because of its speed and Beam Search did extremely well on the smaller datasets. There is much future research to be done in this field, both in the field of constraint development and the introduction of such variables as monetary cost and time constraints, and in the area of algorithm development, where perhaps hybrids of the algorithms studied in our project could be created to achieve the best results. Through this project, many intriguing conclusions were drawn and many unexpected results gathered, and we hope that as the field of Artificial Intelligence grows, we will have access to newer technologies and more powerful computing machinery to allow us to experiment with far more complex theories and implement much more specific constraints for larger and more important problems. This is a fantastic start that has opened a great plethora of opportunity for us to branch out into other fields and apply this new knowledge to greater and even more interesting challenges.

REFERENCES

[1] AI: A Modern Approach Chapter 4, Chapter 6

[2] Kirkpatrick et al (1983) Optimization by Simulated Annealing. Science, New Series, Vol. 220, No. 4598. (May 13, 1983), pp. 671-680

[3] Selman, B. and Levesque, H.J., and Mitchell, D.G. (1992) A New Method for Solving Hard Satisfiability Problems. Proceedings AAAI92, San Jose, CA 440-446

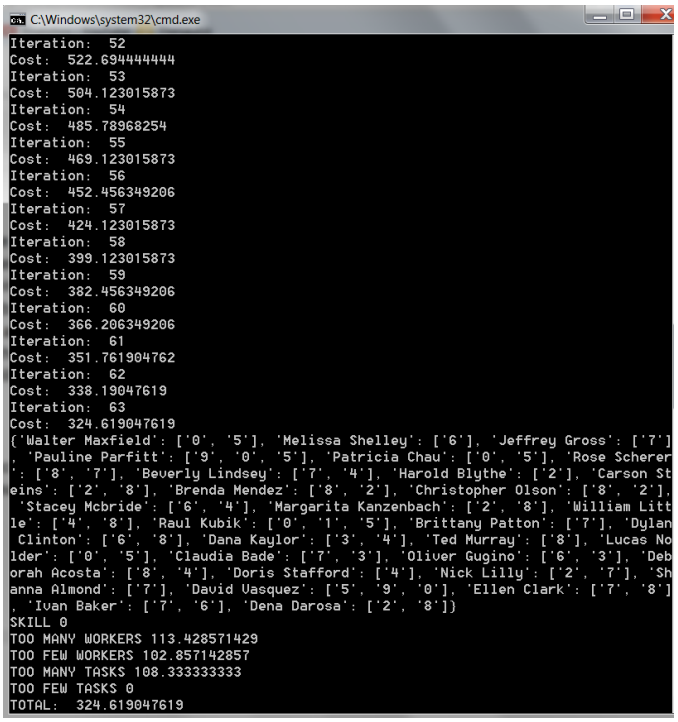
[4] Gerevini, Alfonso and Serina, Ivan (2002) LPGA Planner Based on Local Search for Planning Graphs with Action Costs Dipartimento di Elettronica per lAutomazione, Universita degli Studi di Brescia Via Branze 38, I-25123 Brescia, Italy

[5] Gerevini, Alfonso and Saetti, Alesandro and Serina, Ivan (Dec 2003) Planning through Stochastic Local Search and Temporal Action Graphs in LPG Dipartimento di Elettronica per lAutomazione, Universita degli Studi di Brescia Via Branze 38, I-25123 Brescia, Italy

[6] Selman Bart and Kautz Henry and Cohen Bram Local Search Strategies for Satisfiability Testing (Jan 22, 1995) DIMACS Series in Discrete Mathematics and Theoretical Computer Science

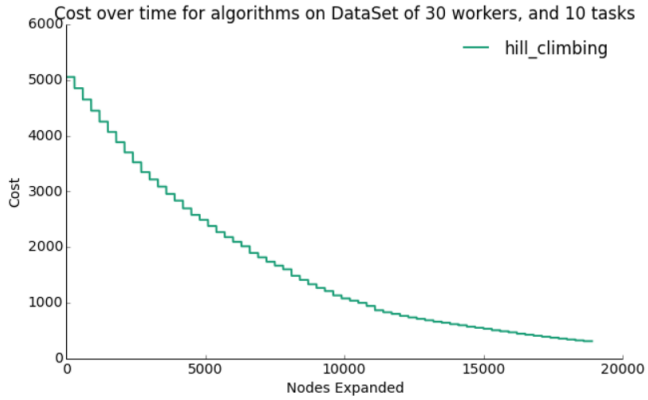
Appendix A. TRACE OF PROGRAM

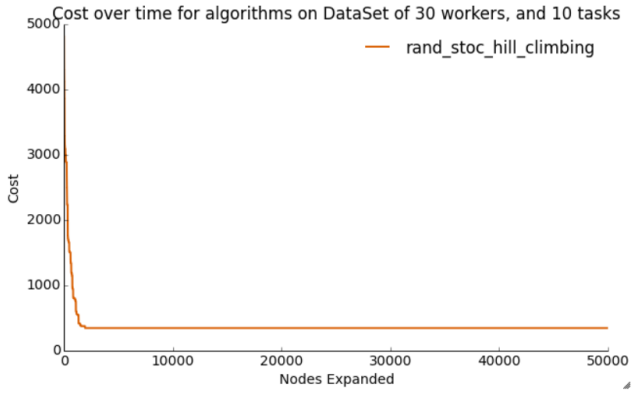
To demonstrate the program fully in action, we show an example of a small data set - assigning 10 tasks to 30 workers. The workers have skills the are distributed with increasing difficulty, meaning that as the skill becomes more challenging, fewer workers possess it. The tasks are distributed similarly - as skills become more challenging, fewer tasks require it. This is a very good and accurate example of a generic workplace and we show here how our program searches through this data. First, we run Hill Climbing. Below is the command line output for this algorithm:

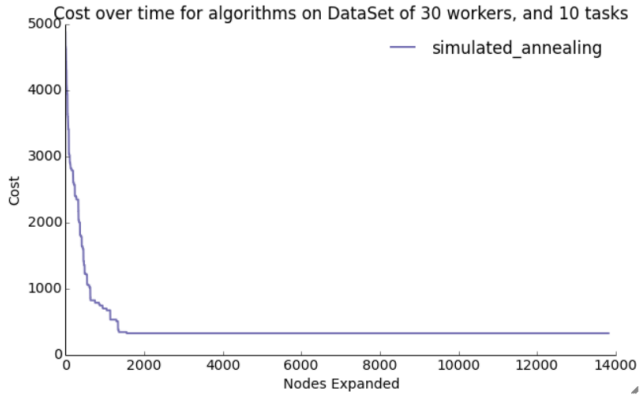


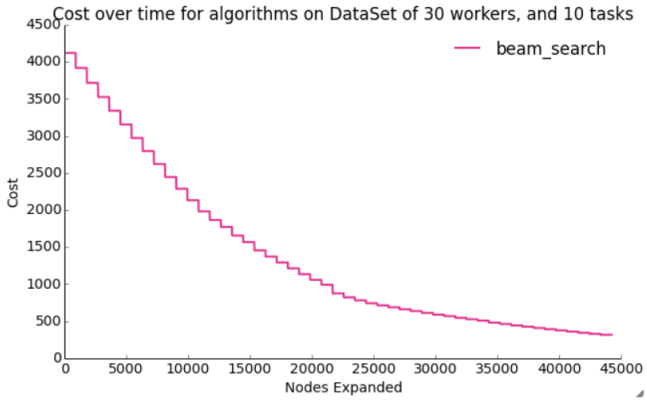
Here, we see that the program went through 63 iterations and resulted with a score of 324.619. We see a printout of the task assignment as well so we can evaluate it holistically if necessary. We proceed in this same fashion for all the algorithms and quantitatively compare their respective lowest cost solutions to see which ones perform

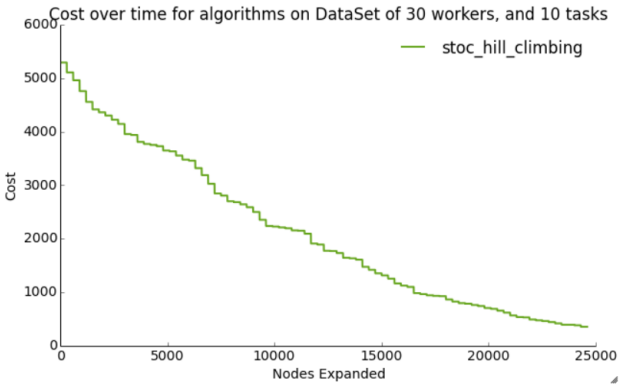
best. We then plot out the line graph for the runtime of each:

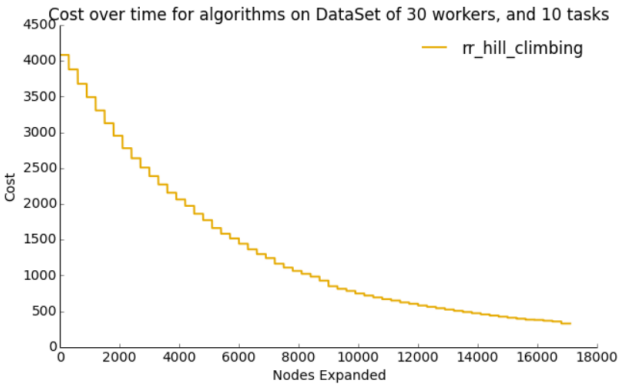












That is what a runthrough of our algorithms looks like. We can now proceed to other CSVs, or if there is something wrong or odd with the final assignment either qualitatively or quantitatively, we can edit the constraints and run the program again.

Appendix B. USING THE SYSTEM

Inside of the attached zip folder, there are 7 Python files, 1 IPython Notebook file, and a folder of 24 CSV files. To run the code, there are several options to be executed from the command line.

python csv_gen.py

This line will generate two CSV files - one of workers and one of tasks - with the parameters and filepath specified in the csv_gen.py file. Feel free to run this if you wish to generate a new collection of workers and tasks by assigning the sizes of each set and the parameters for generation inside the file. This call was used to generate the 24 different CSVs that can be found in the folder

python project.py hill_climbing

This command line will run the passed in parameter (in this case, the *hill_climbing algorithm*) on whatever worker and task CSVs are specified in the project.py file. Feel free to replace the *hill_climbing* call with any of the following algorithms:

- *rr_hill_climbing*
- *stoc_hill_climbing*
- *rand_stoc_hill_climbing*
- *simulated_annealing*
- *beam_search*

Any of the above will be a valid input to the python project.py call, and will run the specified constraints and CSV files that are defined in the project.py file.

Finally, to generate the graphs and results shown above, open the Data Analysis Final.ipynb notebook and select Cell → Run All. This will take quite a bit of time (≈30 minutes) since the function needs to run all algorithms on all possible CSV files. If you would like to test out only small portions of the code, select any cell and run it. For quickest results, we suggest running the cells with the small CSV files to see how the functions perform with a dataset of 30 workers and 10 tasks. This computation should not take much longer than 5 or 10 minutes, depending on the machine.

For full details about the purpose and functionality of each file, please consult the following:

csv_gen.py - This file contains all code to generate CSVs. It does not have a command line interface (in the interests of time and quick testing of features) so all CSV properties and sizes must be defined in the file itself. This will generate exactly two files - a workers file and a tasks file which have properties defined in the worker and task classes (below) and will be traversed when running local search.

task.py - The class definition of a task. Each task has a required name parameter and two optional parameters for the list of skills required to complete the task and the ideal number of workers to undertake the task which is defaulted to 1.

worker.py - The class definition of a worker. Each worker has a required name parameter and four optional parameters for the list of skills the worker possesses, the dictionary of preferences of tasks, the dictionary of efficiencies at the tasks, and the ideal number of tasks the worker should undertake which is defaulted to 0.

project.py - The central file from which the entire project can be run. The file itself specifies what dataset to run search on, and the code in this file can be called via a command line argument as described previously. This will simply run the specified search algorithm on the given CSV and print out the final cost of the found solution

local.py - The file which contains all of the search algorithms and helper functions. All 6 Local Search Algorithms are implemented here as well as all the neighbor generating and state space crawling functions that allow the search to function properly and efficiently. Every function in this file prints out the current and best cost at every iteration and returns only the final assignment.

local2.py - A second variation of the local.py file which is nearly identical but different in several crucial ways that make it ideal for use in the notebook for generating our graphs. The functions in this file do not have any sort of printout so as to make them more lightweight, fast, and usable in the scaled version that is run in the notebook. Additionally, these functions store and return analytics about themselves including number of nodes expanded and the best cost found per node expanded. This makes these functions ideal to run and generate key data that is then plotted in the IPython Notebook (below).

constraints.py - A file that contains all of the constraint functions. Each of these takes a given assignment, a domain, and the cost for the constraints violation and returns the total cost of the assignment by calculating the number of times the constraint was violated. Each of these can be passed a cost of 0 to toggle it off and is called by the project.py file.

Data Analysis Final.ipynb - The Notebook file that contains all of the graph generation algorithms. It scales up the project.py file and can run it over numerous CSV files and output and plot the data output for convenient comparison. Because of the amount of computation it must perform, this file takes quite a bit of time to run and should therefore be run with caution. It is an important file, however, and was crucial to our research and analysis in this project, so we felt it was important to include it.

Appendix C. CODE SUBMISSION

Please see zip file submitted with this paper.