Computational Simulation and Analysis of Landscape Auctions

Submitted in fulfillment of the requirements of the
Honors Thesis in Economics
May 18th, 2016

by

Zachary I. Schutzman

Advisor: Timothy Hubbard
Second Reader: Sahan Dissanayake

Colby College Department of Economics
Waterville, Maine

Introduction

Governments, land trusts, and conservation agencies ("regulators") often have an interest in purchasing or leasing parcels of land owned by private citizens ("landowners") for environmental conservation. For example, the US Department of Agriculture's Conservation Reserve Program allows agricultural landowners to submit to hold their land in conservation instead of farming it in exchange for monetary payments. Each landowner has some individual opportunity cost for allowing her parcel of land to be conserved, which can be viewed as the value of the 'next best' use of her land. This could be the price at which she could sell it to a developer or the expected income from agriculture, for example. Under the assumption that the regulator does not know this private opportunity cost, a reverse auction is an ideal market structure to handle the asymmetric information.

In general, a reverse auction is a model of an auction where the party which owns the item being sold also submits the bid and the buyer can accept or reject. These are most prominently used for contracting of services and are often called procurement auctions. The landscape auction will be modeled as a first-price sealed bid reverse auction, which means the landowners will submit their bids in secrecy and if the regulator accepts a landowner's bid, he pays an amount equal to that bid.

Given a landscape where each site has an environmental value known to the regulator and a profile of bids, it is a relatively simple discrete optimization problem to select a set of sites to either maximize the total environmental value subject to a budget constraint, or minimize the total expenditure given a target environmental value. A more complex problem arises when complementarities from selecting neighboring sites are considered. Benefits from connectedness in the conserved landscape include ease of management and habitat space for threatened species, so a site's environmental value increases with the number of neighboring sites also conserved. A model which incorporates these spatial complementarities more realistically represents the goals of landscape conservation, but is much harder to solve compared to a model which does not account for these benefits.

At the same time, it may be an oversimplification to assign a fixed price to each site based solely on the landowner's opportunity cost of conservation. A

landowner may not know the exact distribution of the environmental values of her site, but she may know that the environmental value increases with the number of neighbors purchased. Therefore, the regulator's willingness to pay for a given site may be significantly higher than the landowner's private valuation. Realistically, a landowner trying to sell her site would want to extract the largest payment from the regulator, so the model should incorporate a mechanism for the landowners to submit rational bids above their private valuations. An extension to this project can focus on determining an equilibrium bidding strategy.

Existing Work

Stephen Polasky, et.al. take a different approach to solve this problem. They prove an efficient mechanism based on a Vickery-Clarke-Groves auction to honestly elicit the private value for each site and provides a subsidy to landowners based on the net environmental benefit of the conserved landscape. The mechanism forces honesty by divorcing the payment from the bid, so it is not a first-price model. This subsidy is independent of the bids, and because the Nash equilibrium in a VCG auction is to honestly report one's private value, there is no possibility for strategic behavior from the bidders (Polasky, et.al. 2014). This mechanism does provide a method of gathering bids in a way that gives the regulator full information about the individual private site values, thus removing the information asymmetry on one side of the market. What is missing from this mechanism is a scalable algorithm for selecting which sites to purchase. The authors give a numerical example of their mechanism on a landscape with eight sites in a four-by-two array. After eliciting the bids from the landowners, the mechanism simply performs an exhaustive search on all combinations of purchasable sites. Firstly, this approach simply purchases sites to maximize the net difference between environmental value and total cost without considering a constraint on the budget or the target value. Secondly, while the exhaustive search algorithm certainly finds the optimal solution for any objective function, it is an intractable method for solving problems of arbitrary size, as the number of candidate solutions is factorial in the number of sites, which necessitates an algorithm that approximates a solution with minimal error.

A genetic algorithm to solve the landscape optimization problem may be an

appropriate tool to provide accurate approximations to the true optimal solution in a much more time- and space- efficient way. Clemens van Dinther's paper "Agent-based Simulation for Research in Economics" introduces a framework for deciding which problems in economics could be solved or have solutions informed by agent-based simulations, where structures or phenomena are simulated with electronic agents which make decisions based on predefined rules and possibly learned information from the model. Van Dinther suggests that agent-based approaches could be useful in solving computational problems such as finding Nash equilibria or optimization. In one section, he outlines an evolutionary approach for a learning genetic algorithm which is useful for solving problems with quantifiable measurements of "fitness" and where solutions can be easily coded as "genes" (van Dinther, 2008).

The problem of optimization of the landscape satisfies the necessary conditions for the evolutionary process to solve the problem; because the value of a landscape is strictly increasing as sites are added to the solution, introducing a budget constraint guarantees the existence of a maximum. The fitness of a given solution is judged solely by the metrics of the environmental value and the cost, and a natural genetic representation of a solution is a binary matrix, where entry (i,j) is 1 if site (i,j) is purchased. The algorithm finds a solution by iteratively using the current generation along with reproductive and mutative rules to create the next generation. Convergence of solutions will be determined by terminating the algorithm after some number of generations without improvement in the best solution.

Framing the Problem

Mathematically, the problem we want to solve is the maximization of the total valuation across all chosen sites such that the total cost of these sites is less than or equal to some budget constraint, called the *value maximization* problem. It is also not unreasonable to consider the *expenditure minimization* problem, where we choose sites to minimize the total cost such that the environmental valuation is greater than or equal to some target value. This project focuses on the value maximization problem, but it is important to note that these problems are dual to one another, and by duality, the solution for one should be very close to the

solution for the other (the discrete nature of the problem results in there being a gap). That is, if V is the maximum value for budget B, then B should be very close to the minimum expenditure for target value V.

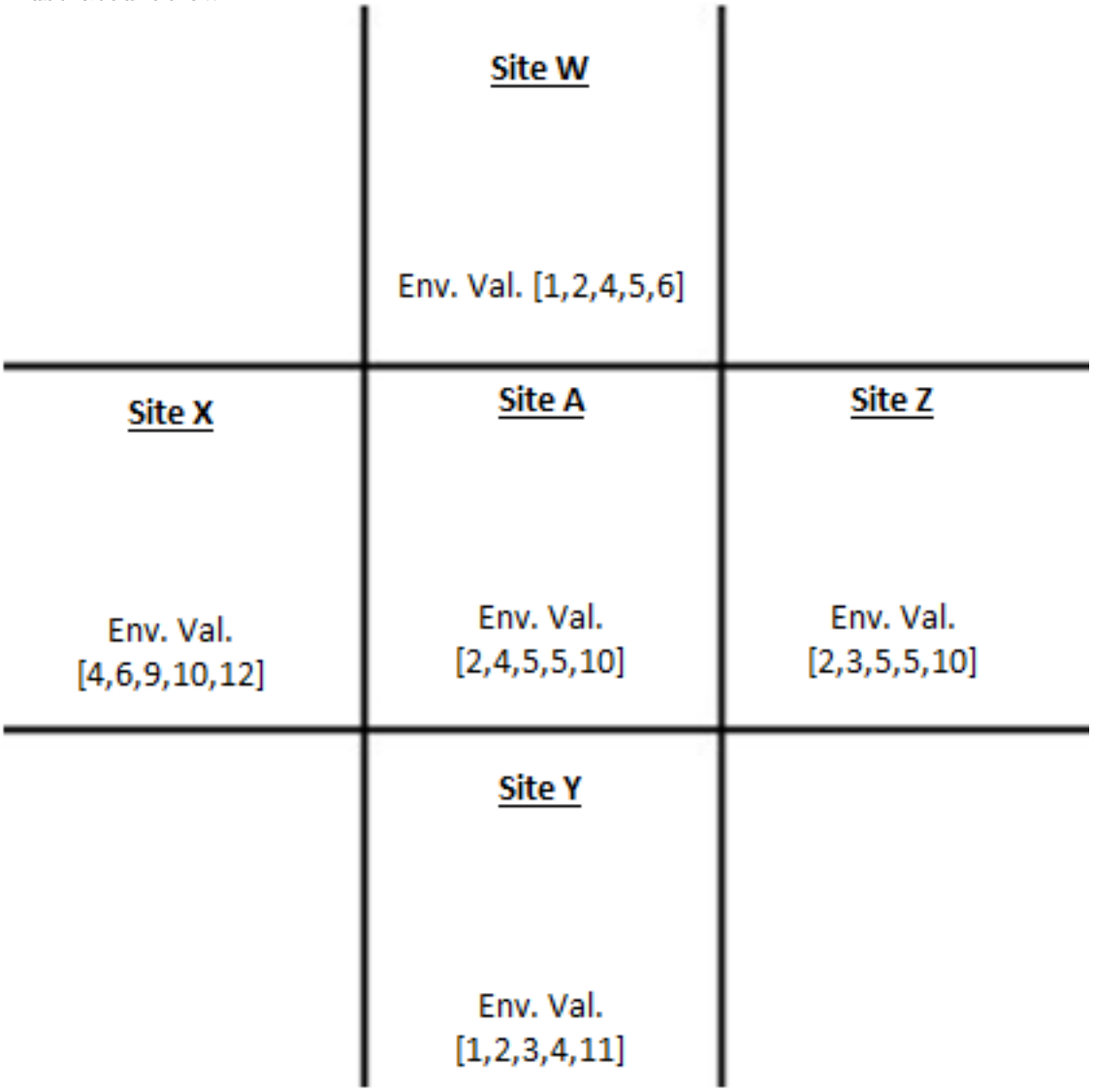
To solve the problem of landscape optimization, I am using a fabricated landscape and tools developed in the Python programming language. While this project has important consequences for real-world design and implementation of conservation auctions, the lack of complete data on bidder behavior and land sales along with the relative infrequency of the auctions makes it infeasible to construct an econometric model from historical data.

To begin, I constructed a theoretical landscape to be considered in the auction. The landscape has 225 sites arranged in a 15-by-15 square grid, and each site's neighbors are considered to be those sites immediately above, below, left, and right of it. Therefore, interior sites have four neighbors, edge sites have three, and corner sites have two. Each landowner has a private valuation for her site drawn independently from a discrete uniform distribution on [2,12]. Similarly, the environmental valuations for a site given some number of neighbors already purchased are a set of five ordered draws from a discrete uniform distribution on [1,13] [1]. That is, if the computer generates the draws {5,2,10,9,6}, that site has value 2 when none of its neighbors are selected, value 5 when one is selected, 6 when two are selected, 9 when three are selected, and 10 when four are selected. In this way, both the private and the environmental valuations for each site are exogenous to the model and both landowners and the regulator must take them as given.

Where this model builds on previous work is that adding a site to a conserved landscape may cause significant externalities due to the complementary effects of selecting neighboring sites. Spatial complementarities are discussed in environmental literature, such as providing species corridors and ease of management, but may be overlooked in computational models due to the added computational complexity of including them. Selecting that site not only adds that particular site's environmental value to the landscape's total, but additionally increases the

[1] The difference in the distributions is to include cases where one or more environmental values of a site are 1, so the private valuation is greater than the environmental valuation in isolation, thus forcing reliance on complementarities. The top end of the distribution was shifted so the means of the two distributions remained equal.

environmental value of its neighbors already in the landscape. A case of this is illustrated below:



Suppose the regulator has already purchased Sites W, X, Y, and Z, and three of each of their neighbors, such that the present environmental values of these sites are 5, 10, 4, and 5, respectively. If the regulator purchases Site A, the environmental value of the whole landscape increases by 10, due to the direct benefit from purchasing Site A when it has four neighbors already in the landscape. However, purchasing Site A also increases the number of neighbors that Sites W, X, Y, and Z have to four. Site W's value increases from 5 to 6, Site X from 10

to 12, Site Y from 4 to 11, and Site Z from 5 to 10. Therefore, the marginal value of adding Site A to the landscape is not 10, but 25, which is a natural maximum willingness-to-pay for the regulator. Therefore, it is entirely possible that a landowner could sell her site for a price significantly higher than her own private valuation in a first-price setting. Similarly, it is possible to construct a scenario where the marginal value of adding a site is always less than that landowner's private valuation, which implies that some landowners may never be able to sell their sites. For example, if in the previous example, the owner of Site A had agricultural income of 30, she would not sell her site, as the maximum willingness-to-pay does not exceed her private valuation.

From a game theoretic perspective, it is a difficult question to determine a set of conditions under which each landowner would or would not sell her site, and derive an optimal bidding strategy from those results. For the purposes of this project, which focuses on optimizing the selection of sites given a profile of bids from the landowners, I take bids as exogenously given, and assume they are equal to the landowners' private valuations.

Optimizing the Landscape

Finding the optimal set of sites to purchase given a budget constraint, the neighbor-conditional valuations, and the bids is a very difficult problem computationally. Formally, this problem is in the complexity class NP-Hard, which means that the only way to find a maximum is to examine every possible solution and save the best one. Additionally, given some solution, the only way to verify it as a maximum is to exhaustively check every solution for something better[2]. Given the computational complexity of the problem, the most efficient algorithm to guarantee a maximum environmental value given a constraint is to exhaustively search all possibilities and select the best solution. However, due to time and space constraints, this is not a tractable method for solving the problem on a 15-by-15 landscape, as illustrated in the following two pseudocode algorithms:

[2]To see that the problem is NP-Hard, consider that this is a 0/1 combinatorial knapsack problem, which is harder to solve than the non-combinatorial 0/1 knapsack problem. Optimization of knapsack problems is NP-Hard.

Algorithm 1 Exhaustive Search Optimization: Space Efficient

```
1: procedure OPTIMIZE(Landscape L, Budget B)
2:     b ← 0
3:     m_b ← None
4:     for i ∈ [0, 2^225] do                ▷ loop over all possible solutions
5:         Generate the corresponding landscape selection m
6:         if value (v) of m > b and cost of m ≤ B then
7:             m_b ← m
8:             b ← v                        ▷ save this better solution
```

Algorithm 2 Exhaustive Search Optimization: Time Efficient

```
1: procedure OPTIMIZE(Landscape L, Budget B)
2:     In Parallel:
3:     Generate all 2^225 solutions, calculate costs and values
4:     Sort the solutions by descending value
5:     Discard all solutions with cost > B
6:     Return the solution with the highest value
```

The issue with both of these solutions is in the number of sites. With 225 sites, there are 2^{225} (roughly $5*10^{67}$) possible solutions. Even at hundreds of comparisons per second, the space-efficient solution is entirely intractable. Similarly, the time-efficient solution would require roughly 10^{70} bits of memory, which is outside of the scope of modern computers. While bounds on the maximum number of sites that can be in a solution given a constraint are easy to calculate and drastically reduce the number of computations that must be made, the purpose of this project is to develop a method to approximate an optimal solution regardless of the size of the landscape.

Formalization of the Model

Because an exhaustive search method is computationally infeasible, it may be impossible to find the optimal choice of sites given a landscape. Formally, there is an evaluation function which maps a landscape and a profile of sites to the associated value and cost of the selection. This is therefore a function

$E : L \times \{0,1\}^N \to \mathbb{R}^2_+$,
where domain elements are a landscape and an N-bit binary number (where N is the number of sites in the landscape) and codomain elements are ordered pairs of positive real numbers corresponding to a cost and a value.

Trying to unpack this function E reveals the core of why this problem is so difficult to approach. E exists for all inputs (l,s), where l is a landscape in L and s is a binary number of appropriate length, $E(l, s)$ is easily computed as $E(l, s) = (\sum v_i s_i, \sum c_i s_i)$, where v_i is the value of site i as a function of how many of its neighbors are selected, c_i is the cost of site i, and s_i is the binary selection variable for site i, as indicated in the input s. Therefore, v_i is itself a function of the sum of s_j for site i's j neighbors, but each of these s_j is itself a function of the original s_i. Because the valuation of each site is implicitly a function of itself, a linear programming approach to approximating a solution will not work.

Two observations about E are important to the complexity of this problem. The first is that, subject to any budget constraint, there exists some selection of sites that maximizes the environmental value of the landscape. This is a result of the valuation of the landscape strictly increasing with sites being added. Because there are a finite number of possible site selection profiles, and the valuations of these profiles correspond to positive real numbers, this set of valuations has a maximal element. The second observation is that there is no guarantee of uniqueness to these solutions. For any landscape and constraint, there may be a multitude of profiles of sites corresponding to the same maximal valuation. More importantly, this means that for any attempt to approximate a solution to the constrained optimization problem, claims about solution strength can only be made regarding the environmental values for the solutions rather than about a comparison between the actual sets of sites in the solution, as an approximation can yield a possible solution with value very close to the true optimum while selecting a vastly different profile of sites.

There are numerous techniques to approximate solutions to problems that are otherwise computationally intractable. One very simple approach is a greedy heuristic. The idea behind a greedy heuristic is that at each iteration, makes the locally most optimal decision without considering future implications of this decision. An outline of a greedy approach to the landscape optimization problem

is as follows:

Algorithm 3 Landscape Optimization: Greedy Heuristic

```
1: procedure GREEDY HEURISTIC(Landscape L,Budget B)
2:     Cost C ← 0
3:     Selected sites m ← None
4:     while C > B do
5:         for each site s ∈ L not already in m do
6:             Calculate the added value of choosing s
7:         Add to m the site with the highest value
8:         Cost C ← C + s_c                    ▷ add the cost of the site
9:     Return m
```

This method is both easy to implement and quick to compute, but it may generate suboptimal solutions. To see that the greedy solution may not be optimal, consider a case where a landscape has a cluster of sites which each have average cost and very low environmental value when no neighbors are chosen but very high value when some neighbors are chosen. Because for all of these sites, the cost exceeds the value of the site in isolation, the greedy heuristic will never select these sites, even if they are part of the optimal solution. However, due to the ease of implementation, a greedy heuristic makes an excellent benchmark against which to compare other methods of approximation.

Due to the discrete nature of the domain (and solution space) and the non-linearity of the valuation function, a genetic algorithm is an appropriate tool to approximate the solution. A genetic algorithm functions metaphorically like a population of a species propagating over generations. Some factor in the model's input takes the role of genes or chromosomes and the fitness of each representative is determined by the strength of its associated solution. In this model, for a given landscape, the binary selection input is the 'gene', and fitness is evaluated by the environmental value of the solution, subject to the cost constraint. In the reproductive phase, the current generation of solutions is used to inform the creation of the next generation with some weighting scheme used to bias the creation of these solutions towards using more of the features from the stronger solutions in the previous generation.

Because of the strong complementarities between neighboring sites, for any

landscape and constraint, there are numerous possible selections of sites which meet that constraint, but may or may not be close to the true optimum. Taking one of these solutions and iterating a genetic algorithm on it may increase the value of the solution, but if there is a better solution for a given constraint which contains a dramatically different set of sites than the one used at the start of the iteration, it is highly unlikely that the algorithm will ever find that better solution. For this reason, the algorithm to approximate the solution to the constrained optimal site selection problem should incorporate a high amount of variance so as to explore a large number of possible solutions at once.

The Genetic Algorithm

In general, the structure of a genetic algorithm is as follows:

Algorithm 4 Genetic Algorithm: Basic Structure

```
1: procedure GENETIC OPTIMIZATION
2:     create an initial population of solutions
3:     while Terminating condition is not met do
4:         use the current generation of solutions to inform creation of the next generation
5:         evaluate your best solution against the terminating condition
6:     return the best solution
```

This outline is intentionally vague to highlight the degree of input the designer has in implementing such an algorithm and its flexibility in being able to be tailored towards a variety of different problems. The following paragraphs describe the specifics of the genetic algorithm used in this project. The full Python implementation of the models can be found in the Appendix.

First, generating the initial population is done mostly randomly with a little bit of guidance. Since there are so many possible solutions in the space, the first step is to find an upper bound on the number of sites in the solution with respect to the constraint. To do this, consider iteratively purchasing the sites with the lowest cost until the constraint is reached. Regardless of the value, this solution contains the absolute maximum number of sites of any solution given that constraint, so

call this number of sites M. Given a landscape and a constraint, the following sub-algorithm is used to generate the initial population of solutions:

Algorithm 5 Genetic Algorithm: Generating the Initial Population

```
1: procedure RANDOM GENERATION(Landscape L, Budget B)
2:     calculate an upper bound M on the number of sites in a solution given B
3:     for each n ∈ [0, M] do
4:         generate 50 random solutions with n sites
5:         discard solutions with cost exceeding B
6:     order all solutions by value
7:     use the 600 best solutions as the initial population
```

Next, these solutions must iteratively be used to inform the creation of the next generation of solutions. The implementation uses several techniques to accomplish this in order to achieve a high amount of variance across potential solutions. The first and simplest method is to carry over the best 100 solutions from the previous generation. This step ensures a constantly improving core of solutions being carried from one generation to the next. Additionally, some more random solutions are created and added to the next generation. Throughout the process, all steps will be taken under the assumption that infeasible solutions are discarded upon generation and lists of solutions are ordered by descending value.

The genetic process uses four subprocesses to create the next generation of solutions, three of which are described in detail below. The first technique is a mutagenic step. The idea behind this is to make small adjustments to solutions in the current generation to create solutions in the next generation. Solutions created by this process should look similar but not identical to members of the current generation with preference towards the strongest solutions. The second process is a probabilistic generation where the aggregated relative frequencies of each site's occurrence in the best solutions from the current generation are used to inform the creation of the next generation. As there is no weighting by solution strength, solutions created with this method should be random variations on a composite version of the best solutions from the current generation. Because of the way the random numbers are drawn, the relative frequencies of occurrence for the sites in the solutions created with this method should match those from the current generation. The third method is a two-step greedy adjustment where solutions

are slightly improved by greedily adding the best unselected site and removing the worst selected one. The final method is more random generation, which helps increase variety among candidate solutions.

The first technique to generate new solutions using the information from the current generation is a probabilistic bit switch. Looking at each solution as a binary matrix where entries of 0 and 1 represent the corresponding site being unchosen and chosen, respectively, an old solution can inform a new one by choosing a small number of entries and flipping the bit in that location. In order to bias using stronger solutions to create the next generation, a discretized beta distribution is used to weight the random selection process to favor those stronger solutions. The outline of the algorithm is as follows:

Algorithm 6 Genetic Algorithm: Bit-Switch Mutation

```
1: procedure BIT SWITCHING(Landscape L, Budget B, Solution set P)
2:     initialize P* to be the next generation of solutions
3:     for n ∈ [0, 125] do
4:         draw a value i from a discretized beta distribution
5:         index into m = P[i]              ▷ distribution favors stronger solutions
6:         for each entry e ∈ m do
7:             with probability .08, swap the state of the e          ▷ {0 ↔ 1}
8:         add the solution to P*
9:     return P*
```

The advantage to this technique is that it creates the probability for the creation of any arbitrary solution. In particular, given an initial solution, the process generates a specific target solution with probability $.96^A * .08^B$, where A is the number of corresponding pairs of sites that are either both chosen or both not chosen in each solution and B is the number of sites where the status differs between the two solutions. This technique therefore draws solutions from a roughly normal distribution centered at the seed solution, but when combined with the beta distribution used to pick each seed, the distribution of solutions generated in this way will be skewed towards the stronger solutions from the previous generation.

The second technique is a probabilistic generation of new solutions where the probability of each site's inclusion is equal to the proportion of the best 125 solutions it occurs in. For example, a site which is included in every one of the top

125 solutions will be included in every solution generated by this method while a site that occurs in 13 of 125 should only be included in approximately 10 percent of these solutions. The following algorithm describes this process:

Algorithm 7 Genetic Algorithm: Probabilistic Generation

```
1: procedure PROBABILISTIC(Landscape L, Budget B, Solution set P)
2:     initialize a 15 × 15 matrix m to all zeroes
3:     initialize P* to be the next generation of solutions
4:     for each site s ∈ L do
5:         calculate the proportion r of the top 125 solutions in P that site s is
   selected
6:         store r in the corresponding entry of m
7:     for n ∈ [0, 125] do
8:         generate a 15 × 15 matrix m* with each entry drawn from [0,1]
9:         for each entry e in m* do
10:            if the corresponding entry in m is less than e then
11:                set e to 1
12:            else
13:                set e to 0
14:        add m* to P*
15:    return P*
```

While the first technique uses a beta draw to skew the distribution in favor of the stronger solutions, this method applies equal weight to each of the 125 selected members from the current generation, so the distribution of solutions generated in this way should be centered about the 'average' solution of this subset.

The final process used to generate the next generation of solutions is a greedy adjustment on the top 25 solutions from the current generation. The problem with using a greedy heuristic from the outset was that the distribution of values from the spatial complementarities may prevent a cluster of high value sites from being chosen due to having low value if chosen in isolation. Applying a greedy heuristic to an existing core solution reduces the probability of this occurring, as there may already be member sites from these clusters in the solution, so the one-step greedy algorithm does indeed add them to the solution. In this greedy update, the top 25 solutions are adjusted by removing the lowest value site currently in the solution and adding the highest value site not in the solution. The following algorithm describes this process:

Algorithm 8 Genetic Algorithm: Greedy Update

```
procedure GREEDY(Landscape L, Budget B, Solution set P)
    for each of the top 25 solutions in P do
        repeatedly add the highest and remove the lowest value sites
```

This greedy update process is very effective at achieving a stable solution, particularly in conjunction with the high variance of the bit switch and probabilistic generation processes. One criterion for terminating a genetic algorithm is to stop when there is no improvement in the best solutions from generation to generation. Using the previously described sub-algorithms, the greedy update steps allows the algorithm to converge to a solution in only about four generations, which is important, due to the relative computational cost of the greedy update in comparison with the various randomized features. Detailed results are discussed in the next section.

Results and Discussion of the Model

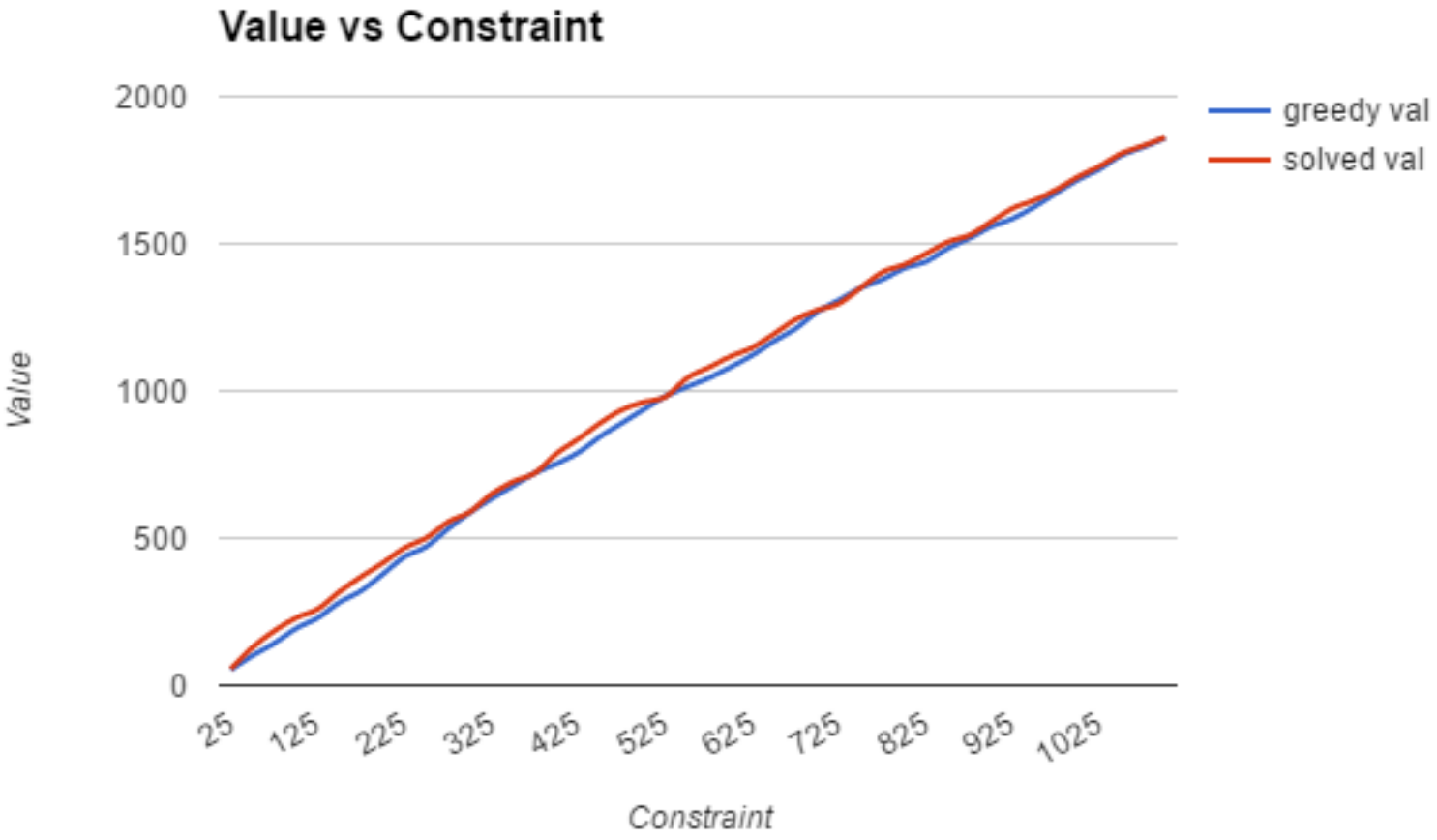
To gather comparative results, a fixed landscape was randomly generated and the processes described above were used to generate both greedy and optimized solutions for cost constraints of 25 to 1100 in increments of 25, yielding 44 observations. From an intuitive perspective, economic and mathematical theory informs a few predictions about the results of the model, which the data can either reaffirm or challenge. First, we should expect that within the greedy solutions and approximated optimal solutions the value of the landscapes should be strictly increasing with the cost constraint because each site has positive valuation. Additionally, we expect for both models that the rate of increase of value (an estimate of the derivative of value with respect to cost) should be higher for lower budgets and decrease as the budget constraint rises. This intuitively rises from the idea that both the greedy algorithm and optimized solution should add sites of high value early in the process leaving lower value sites to be added in settings where the budget constraint is higher. Economically, this would indicate diminishing marginal benefits to purchasing sites optimally. Underneath all of this is the expectation that the optimization process should provide significantly better solutions than the

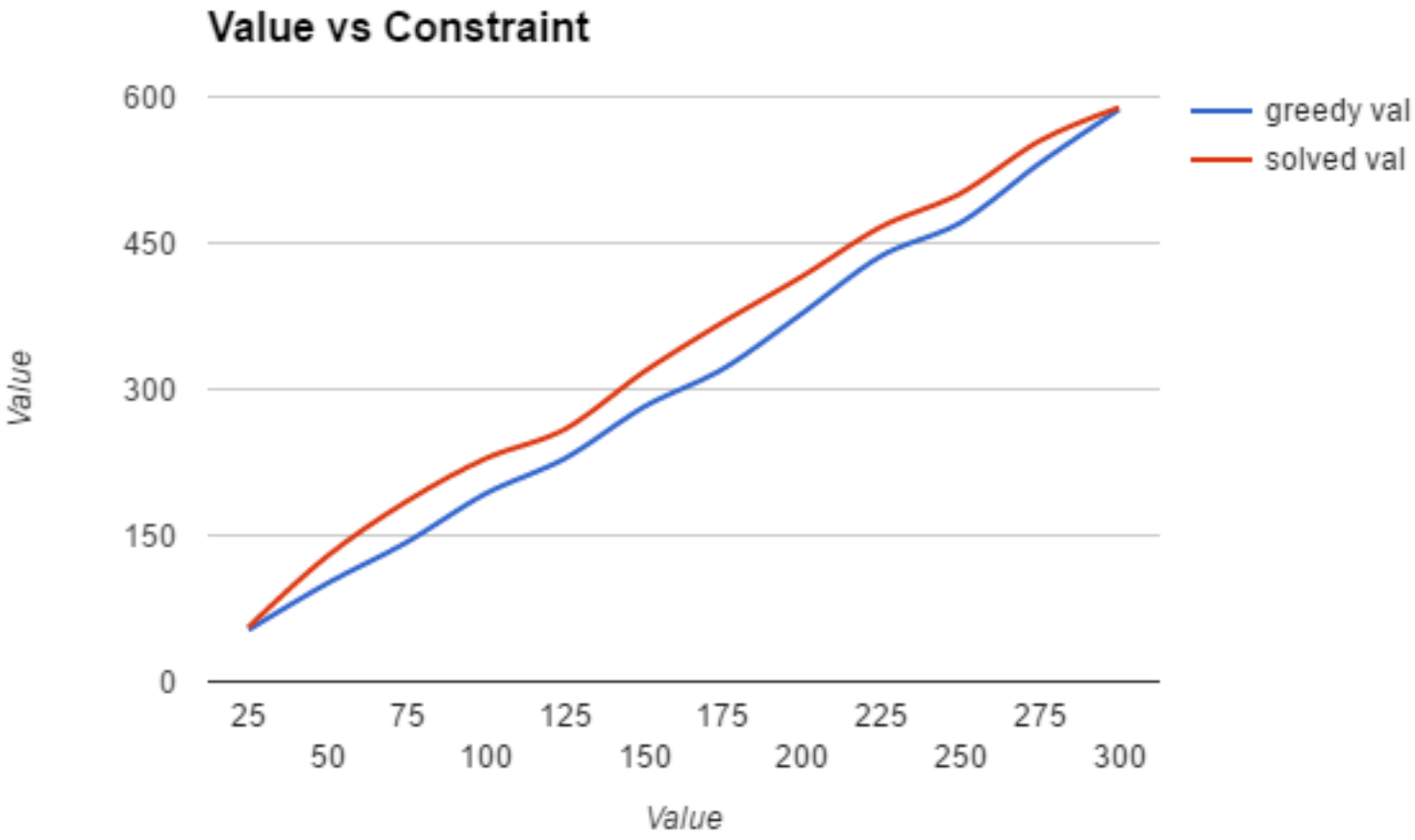
greedy algorithm for low budget constraints and there to be not a large difference for high budget constraints.

The following table contains the results of one full run of the model:

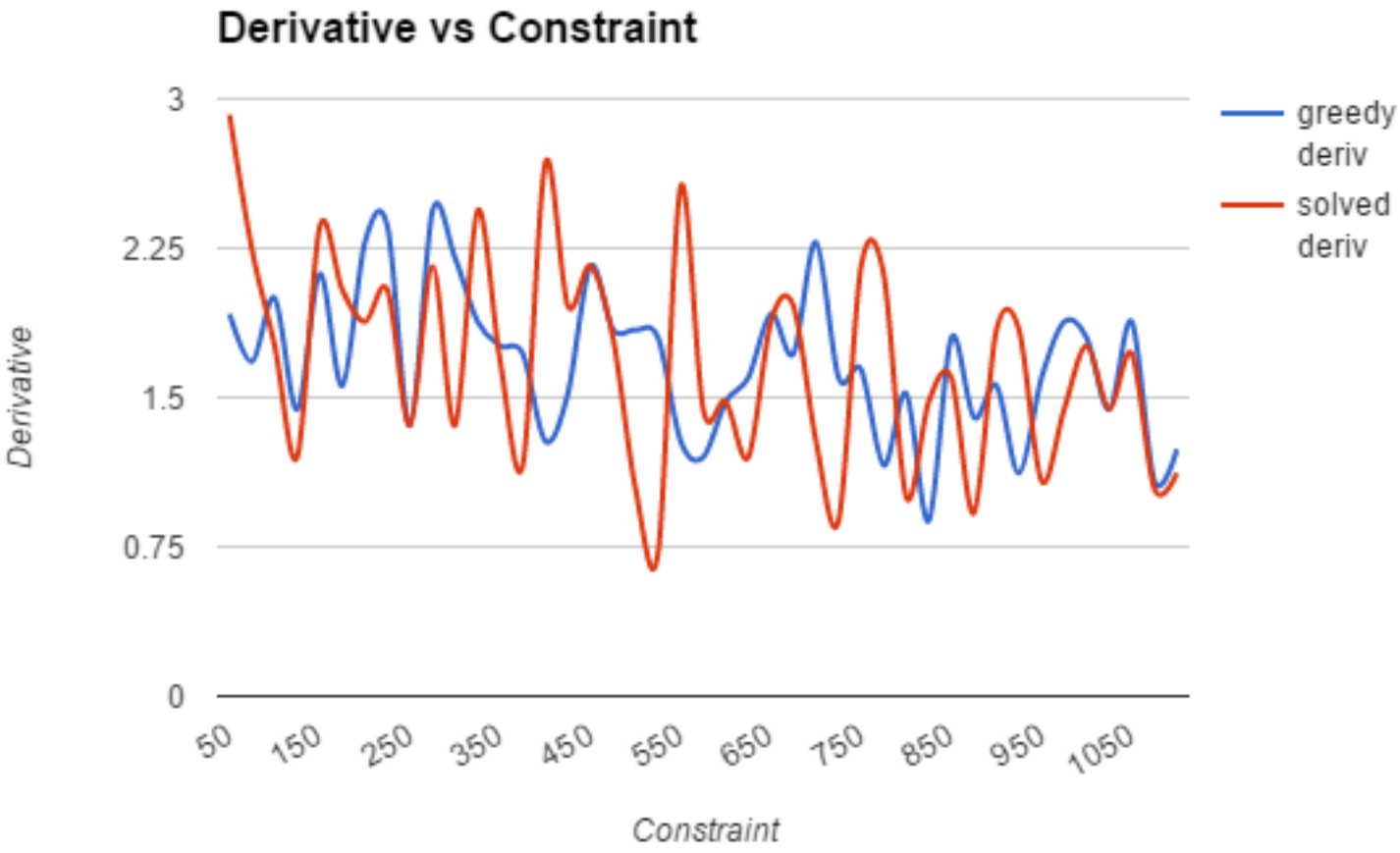
Constraint	Greedy Value	Greedy Cost	Solved Value	Solved Cost
25	53	24	56	24
50	101	46	129	49
75	143	72	185	75
100	193	98	229	99
125	229	124	259	120
150	282	147	318	150
175	321	171	369	172
200	378	200	416	196
225	437	225	467	225
250	471	242	501	241
275	532	271	555	265
300	587	300	589	291
325	634	325	650	322
350	678	347	692	348
375	721	373	721	368
400	753	398	788	399
425	791	423	837	417
450	845	446	891	449
475	891	469	936	474
500	937	495	962	495
525	982	525	980	524
550	1014	549	1044	543
575	1044	573	1080	565
600	1081	593	1117	599
625	1121	620	1147	620
650	1169	649	1194	650
675	1212	674	1243	668
700	1269	698	1275	700
725	1309	725	1297	723
750	1350	750	1351	746
775	1379	771	1404	775
800	1417	798	1429	800
825	1439	818	1466	824
850	1484	848	1506	849
875	1519	870	1529	875
900	1558	899	1575	898
925	1586	921	1621	921
950	1626	947	1648	948
975	1673	973	1684	973
1000	1718	999	1728	998
1025	1754	1016	1764	1024
1050	1801	1050	1807	1046
1075	1828	1072	1833	1073
1100	1859	1099	1861	1099

The data affirms the first prediction; for both the greedy and optimized solutions, the value is strictly increasing with the budget constraint. The following two charts plot the value on the vertical axis against the constraint on the horizontal axis. The first chart covers the whole model while the second is restricted to solutions for constraints less than or equal to 300.





Both these tables illustrate and confirm the first theoretical prediction that value is strictly increasing with the constraint for both techniques. It also demonstrates evidence that the approximation algorithm does a better job outperforming the greedy algorithm at low budget constraints than at high budget constraints. To aid in examining the third theoretical prediction, the following graph charts a discrete derivative against the constraint.



While the trajectory of the data seems relatively random, there is a general downward trend, suggesting weak evidence for diminishing marginal benefits from higher budget constraints. One quality of the model to note here is that because the landscape has a fixed number of sites, there does exist a maximum possible attainable value, which corresponds with the cost of purchasing every site in the system. At this point, the marginal returns must be zero.

Both the greedy solution, which by definition builds on a solution for a lower constraint, and the solved solution, which by its construction optimizes against each constraint independent of lower solutions, exhibit the high variance in the approximated derivative. This suggests that this variation is an artifact of the complexity and discreteness of the solution space rather than there truly not being diminishing marginal benefits from increased budgets. One natural way to confirm

or reject this notion would be to solve a wide variety of these models and determine if the average trend shows diminishing returns.

Another observation is that while the optimized solution does outperform the greedy algorithm, in a few instances by over 15 percent, the first graph shows them trending very close together, which raises the question of whether engaging in the design and implementation of the optimizer is worth the benefits. In general, this obviously depends on the cost and benefits of undertaking the action and cannot be universally determined based on the results of the fictional landscape discussed in this project. However, the relative success of the greedy algorithm compared to the optimizer may also be a result of assumptions about which party has access to various pieces of information. For example, the greedy algorithm depends on the regulator knowing, for every site, how the addition of that site affects the total landscape value. Restricting this information would significantly impede the greedy algorithm's ability to find strong solutions and may result in the solver doing significantly better than the greedy heuristic at a wider range of constraints.

An aspect of these auctions that this model leaves out is the behavior of the bidder. The models presented examined optimal selection given some exogenously determined price assigned to each site. In reality, each landowner wants to submit a bid to maximize expected profits from potentially selling her site under the restriction that she will not bid below a predetermined private valuation for her land. An interesting extension to this project would be to examine how the bidders should behave in this setting. The auction can be modeled as a one-shot sequential game where landowners simultaneously and independently select a bid and the regulator observes these bids and decides which sites to purchase. The problem here is determining how landowners can use the limited information available to them to make their own optimal choice.

Conclusion

Overall, examining this optimization in the context of spatial complementarities better reflects environmental goals in conservation but also adds layers of computational complexity to solving a theoretical version of the model. The models presented here show that a simple to implement greedy algorithm which in-

corporates cost and valuation information performs significantly better than an uninformed greedy model and nearly as well as the genetic solver. Due to the strong spatial complementarities, diminishing marginal returns to increasing the budget are not strong, and the marginal return from one increased unit of spending is more than one unit of environmental value at most levels, particularly for low constraints. The policy implications of these results are that there may be significant benefits from increasing spending on conserving sites, particularly when the budget constraint only allows purchasing of few sites. Additionally, these spatial complementarities should be factored into decision models for real-world applications due to the direct and indirect value effects from potentially conserving neighboring sites.

One aspect of such a selection model that was largely ignored in this project but presents interesting challenges to optimization from both a computational and a mechanism design perspective is combinatorial bidding and buying. The model and algorithm in this project are founded on an assumption that each landowner independently generates a bid for her site and the regulator can select which bids to accept and which to not. Due to the spatial complementarities which may cause some sites to have very low value in isolation but very high value when some number of its neighbors are selected, it may be in the regulator's best interest to be able to accept bids on bundles of sites as a way of ensuring that the spatial complementarities are captured. Exploring such an optimization problem would be a fascinating extension to this paper.

An additional facet of this problem that could be explored in an extension to this project is the role of strategic behavior for the bidders in this model. Here we treated the bids as being generated exogenously and the regulator took them as given. Hailu and Schilizzi (2004) explore a computational model where a landscape auction is iterated and bidders adjust their bids over time to try to capture as much surplus as possible. Extending their model to a setting with spatial complementarities and combinatorial bidding could better inform mechanism designers as to how individuals behave at these kinds of auctions.

References

Davis, Lawrence. "Handbook of genetic algorithms." (1991).

Hailu, Atakelty, and Steven Schilizzi. "Are auctions more efficient than fixed price schemes when bidders learn?." Australian Journal of Management 29, no. 2 (2004): 147-168.

Polasky, Stephen, David J. Lewis, Andrew J. Plantinga, and Erik Nelson. "Implementing the optimal provision of ecosystem services." Proceedings of the National Academy of Sciences 111, no. 17 (2014): 6248-6253.

Van Dinther, Clemens. "Agent-based simulation for research in economics." In Handbook on Information Technology in Finance, pp. 421-442. Springer Berlin Heidelberg, 2008.

Appendix: Code

```
\singlespacing
#site.py

'''
Created on Sep 17, 2015

@author: zachary
'''

import randomSeeded
import copy


def cloneSite(s):
newSite = Site(s.loc,s.index)
newSite.values = s.values[:]
newSite.curVal = newSite.values[0]
newSite.privateVal = copy.deepcopy(s.privateVal)
newSite.nbrs = []

return newSite

class Site:
'''
classdocs
'''

def __init__(self, loc, idx):
'''
Constructor
'''


self.loc = loc
self.index = idx
self.pastBids = []
self.pastSales = []
self.sold = False
self.lastBid = None
self.curVal = None
```

```
self.picked = False

self.varVal = 0
self.selVar = None
self.varCost = 0

self.values = []

for v in range(5):
self.values.append(randomSeeded.random.randint(1,13))

self.values.sort()
self.curVal = self.values[0]
for i in range(4):
self.values[i+1] = self.curVal

self.privateVal = randomSeeded.random.randint(2,12)
self.marginalVal = 0




def addNeighbors(self, nbrs):

self.nbrs = []

for n in nbrs:
self.nbrs.append(n)

def getPicked(self):
return self.picked

def updateValue(self):
self.curVal = self.values[sum(s.getPicked() for s in self.nbrs)]

def getCurVal(self):
self.updateValue()
return self.curVal
def getCurValLS(self):
self.updateValue()
return self.curVal * self.getPicked()
def getValues(self):
return self.values
```

```
def getBids(self):
return self.pastBids
def getSales(self):
return self.pastSales
def getSuccessfulBids(self):
return [a*b for a,b in zip(self.pastBids,self.pastSales)]
def getPrivateValue(self):
return self.privateVal
def getPrivateValueLS(self):
return self.privateVal * self.getPicked()
def getLocation(self):
return self.loc
def getNeighbors(self):
return self.nbrs
def choose(self):
self.picked = True
def unchoose(self):
self.picked = False
def setVar(self, v):
self.selVar = v
self.varVal = v * self.getCurVal()
self.varCost = v * self.privateVal

def getNumPickedNeighbors(self):
return sum([n.getPicked() for n in self.nbrs])
```

```
#randomSeeded.py


import random
import numpy

seed = 3302016


random.seed(seed)
numpy.random.seed(seed)
```

```
#landscape.py

'''
Created on Sep 17, 2015

@author: zachary
'''
import site
import sys


class Landscape:
'''
classdocs
'''


def __init__(self, size):
'''
Constructor
'''
self.heldSite = None
self.map = []
m = 0
for i in range(size):
row = []
for j in range(size):
k = site.Site((i,j),m)
row.append(k)
m = m+1
self.map.append(row)
self.size = size
self.buildNeighbors()

def clone(self):
newScape = Landscape(self.size)
for i in range(len(self.map)):
for j in range(len(self.map[0])):
newScape.map[i][j] = site.cloneSite(self.map[j][i])

return newScape
```

```
def getSite(self,loc):
return self.map[loc[0]][loc[1]]

def chooseSite(self,loc):
s = self.map[loc[0]][loc[1]]
s.choose()
def unchooseSite(self,loc):
s = self.map[loc[0]][loc[1]]
s.unchoose()

def unchooseAll(self):
for l in self.map:
for s in l:
s.unchoose()

def buildNeighbors(self):
for i in range(self.size):
for j in range(self.size):
n = []

try:
n.append(self.getSite((i,j-1)))
except IndexError:
pass
try:
n.append(self.getSite((i-1,j)))
except IndexError:
pass
try:
n.append(self.getSite((i+1,j)))
except IndexError:
pass
try:
n.append(self.getSite((i,j+1)))
except IndexError:
pass


self.getSite((i,j)).addNeighbors(n)
def updateEnvVal(self):
for i in range(len(self.map)):
for j in range(len(self.map)):
```

```
self.getSite((i,j)).updateValue()


def getEnvVal(self):
self.updateEnvVal()
enVal = 0
for i in range(len(self.map)):
for j in range(len(self.map)):
enVal = enVal + self.getSite((i,j)).getCurValLS()
return enVal

def getCost(self):
cost = 0
for i in range(len(self.map)):
for j in range(len(self.map)):
cost = cost + self.getSite((i,j)).getPrivateValueLS()
return cost

def printEnvVal(self):
print self.getEnvVal()
print self.getCost()

def map_to_choose(self,map):

self.unchooseAll()
for i in range(len(map)):
for j in range(len(map[0])):
if map[i][j] == 1:
self.chooseSite((i,j))
self.updateEnvVal()

def choose_to_map(self):

return [[int(s.getPicked()) for s in l] for l in self.map]

def updateMargVals(self):

for l in self.map:
for s in l:
st = s.getPicked()
s.choose()
m = self.getEnvVal()
s.unchoose()
```

```
m = m - self.getEnvVal()
s.marginalVal = m

if st:
s.choose()



def greedyOptimize(self):

self.updateMargVals()
self.updateEnvVal()

maxMargVal = -.01
maxSite = None
s = None

minMargVal = 1000000
minSite = None


for i in range(self.size):
for j in range(self.size):
s = self.getSite((i,j))
if not s.getPicked():
if (s.marginalVal-s.privateVal) > maxMargVal:
maxMargVal = s.marginalVal-s.privateVal
maxSite = s
if s.getPicked():
if (s.marginalVal -s.privateVal) < minMargVal:
minMargVal = s.marginalVal - s.privateVal
minSite = s

if minMargVal < maxMargVal:
minSite.unchoose()
maxSite.choose()

def greedyReduce(self):

self.updateMargVals()
self.updateEnvVal()
```

```
minMargVal = 1000000
minSite = None

for i in range(self.size):
for j in range(self.size):
s = self.getSite((i,j))
if s.getPicked():
if s.marginalVal - s.privateVal < minMargVal:
minSite = s
minSiteLoc = (i,j)
minMargVal = s.marginalVal - s.privateVal
if minMargVal < 1000000:
minSite.unchoose()


def greedyAdd(self):

self.updateMargVals()
self.updateEnvVal()

maxMargVal = -.01
maxSite = None
s = None


for i in range(self.size):
for j in range(self.size):
s = self.getSite((i,j))
if not s.getPicked():
if (s.marginalVal - s.privateVal) > maxMargVal:
maxMargVal = s.marginalVal - s.privateVal
maxSite = s
maxSiteLoc = (i,j)

#        print maxSite, maxSiteLoc, maxSite.getPicked()
if -.001 < maxMargVal:
#minSite.unchoose()
maxSite.choose()
return maxSiteLoc
else: return (-1,-1)
```

```
#genalg.py

import math
import sys
import randomSeeded
import landscape
import time
from multiprocessing import Process
import copy


class Solver:
def __init__(self, ls, obj = "ValMax", constr = sys.maxsize):

self.ls = ls
self.numSites = ls.size * ls.size
self.siteMap = list(self.ls.map)
self.siteMapCopy = list(self.siteMap)

self.sites = []
for i in range(ls.size):
for j in range(ls.size):
self.sites.append(self.siteMap[i][j])

self.objective = obj
self.constraint = constr

if self.objective == "ValMax":
self.bound = self.valmax_bound()

self.solTemp = [0] * self.numSites
self.curSols = []
self.survSols = []

self.bestSols = []


def valmax_bound(self):
```

```
self.sites.sort(key=lambda x: x.privateVal)

siteCount = 0
cost = 0
while cost <= self.constraint:
s = self.sites[siteCount]
cost = cost + s.privateVal
siteCount += 1

return siteCount

def generate_random_map(self,sels):

unshuff = [1]*sels + [0]*(225-sels)
randomSeeded.random.shuffle(unshuff)
randMap = [unshuff[k:k+15] for k in range(0,len(unshuff),15)]

return randMap
def stable(self, c, v):

if abs((self.prevCost - float(c))/c) < .0075 and abs((self.prevVal -
    float(v))/v) < .0075:
return True
return False

def valmax_sol2(self):

rands = []
self.curSols = []

if self.survSols == []:
#           a = [[0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 1], [0, 0,
    0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 1, 0], [0, 0, 0, 0, 0, 0, 0, 0, 1,
    0, 0, 0, 1, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1], [0,
    0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 1], [0, 0, 0, 0, 0, 0, 0, 0,
    0, 1, 1, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1], [0, 0, 0, 0, 0, 0, 0,
    0, 0, 1, 1, 0, 0, 0, 0], [0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0,
    0], [0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 1,
    0, 1, 0, 0, 0, 1, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0,
    0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0], [0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 1, 1, 1, 0]]
#           self.ls.map_to_choose(a)
```

```
#           rands.append((a,self.ls.getEnvVal(),self.ls.getCost(),'a'))
for i in range(self.bound):
for j in range(20):
m = self.generate_random_map(i+1)
self.ls.map_to_choose(m)
rands.append((m,self.ls.getEnvVal(),self.ls.getCost(),'r'))

rands.sort(key = lambda x: x[1], reverse = True)
rands = [t for t in rands if t[2] <= self.constraint]
mvc = rands[:]
#           print "initial randoms generated, sorted, and filtered"


mvc.sort(key = lambda x: x[1], reverse = True)
p = min(150, len(mvc))
gen_cur = mvc[:p]
self.survSols = gen_cur[:]

for s in self.survSols[:25]:
if s[0] not in [c[0] for c in self.survSols[:25]]:
self.curSols.append(s)



p = len(self.survSols)
pmap = [s[0] for s in self.survSols]
for i in range(len(pmap)):
pmap[i] = [val for sublist in pmap[i] for val in sublist]

pmap = [sum(x) for x in zip(*pmap)]
pmap = [x/float(p) for x in pmap]

#        print "probabilities generated"

probMaps = []
while len(probMaps) < 50:
probs = [randomSeeded.random.random() for i in range(225)]
for r in xrange(225):
if probs[r] < pmap[r]:
probs[r] = 1
else:
probs[r] = 0
probs = [probs[k:k+15] for k in range(0,len(probs),15)]
```

```
self.ls.map_to_choose(probs)
if self.ls.getCost() <= self.constraint:
probMaps.append(probs)

for m in probMaps:
self.ls.map_to_choose(m)
self.curSols.append((m,self.ls.getEnvVal(),self.ls.getCost(),'p'))

#        print "probabilistic solutions generated"

rands = []
for i in range(self.bound,self.bound/2,-1):
for j in range(25):
m = self.generate_random_map(i)
self.ls.map_to_choose(m)
rands.append((m,self.ls.getEnvVal(),self.ls.getCost(),'r'))

rands.sort(key = lambda x: x[1], reverse = True)
rands = [t for t in rands if t[2] <= self.constraint]

self.curSols = self.curSols + rands[:75]

#        print "random solutions generated"



######
tSols = self.survSols[:min(25,len(self.survSols)-1)]
count = 0
gt = time.time()
for s in tSols:
self.prevCost = sys.maxsize
self.prevVal = sys.maxsize
if count == 1:
pass
#                print "one greedy time: ", time.time() - gt
count += 1
#            print count, " greedy"
self.ls.map_to_choose(s[0])
loops = 0
while self.ls.getCost() <= self.constraint and loops < 7:
loops += 1
```

```
#               print "stability: ", self.stable(self.ls.getCost(),
    self.ls.getEnvVal())
self.prevCost = self.ls.getCost()
self.prevVal = self.ls.getEnvVal()
#               print "loops: ", loops
#               print self.ls.getCost(), self.ls.getEnvVal()
self.ls.greedyOptimize()
self.ls.updateEnvVal()

count = 0
while self.ls.getCost() > self.constraint:

self.ls.greedyReduce()


while (self.ls.getCost() + 12 <= self.constraint) and count < 7:
#               print "greedy add: ",count
#               print "cost, value, numsites: ", self.ls.getCost(),
    self.ls.getEnvVal(), sum([sum(l) for l in self.ls.choose_to_map()])
count += 1
self.ls.updateEnvVal()
if self.ls.greedyAdd() == None:
continue


self.curSols.append((self.ls.choose_to_map(),self.ls.getEnvVal(),self.ls.getCost(),'g'))



#       print "greedily modified solutions generated"
#######


self.curSols.sort(key = lambda x: x[1], reverse = True)
stt = time.time()
sol_maps = [s[0] for s in self.survSols]

mut_genes = []
for i in range(25):
x = randomSeeded.random.betavariate(1,5)
ind = len(self.survSols)-1
ind = ind*x
ind = int(ind)
```

```
mut_genes.append(ind)
#        print "mutation selections:", mut_genes
drt = time.time()

#        print "draw time: ", drt-stt

#        print "sol_maps length", len(sol_maps)

mut_sol_maps = []

for i in mut_genes:

mut_sol_maps.append(sol_maps[i])


for m in mut_sol_maps:
d = randomSeeded.numpy.random.binomial(225,.08)
for i in range(d):
m[randomSeeded.random.randint(0,14)][randomSeeded.random.randint(0,14)]
self.ls.map_to_choose(m)
self.curSols.append((m,self.ls.getEnvVal(),self.ls.getCost(),'m'))


tmt = time.time()
#        print "total mutation time: ", tmt-stt

self.curSols.sort(key = lambda x: x[1], reverse = True)
self.curSols = [t for t in self.curSols if t[2] <= self.constraint]


i = 0
while i < len(self.curSols):
if self.curSols[i][0] not in [c[0] for c in self.bestSols]:
self.bestSols.append(self.curSols[i])
break
i += 1
self.survSols = self.curSols[:min(150,len(self.curSols))]


#        print "FINISHED", [s[1] for s in self.curSols[:15]]
```

```
def main(constraintVal):

global ls
t1 = time.time()
copyScape = copy.copy(ls)

solver = Solver(copyScape, constr = constraintVal)

print solver.bound, "bound"
mrp = (0,0)
while (solver.ls.getCost() < solver.constraint) and mrp != (-1,-1):
mrp = solver.ls.greedyAdd()

solver.ls.updateEnvVal()
if solver.ls.getCost() > solver.constraint:
print "IT HAPPENED", mrp
solver.ls.unchooseSite(mrp)
solver.ls.updateEnvVal()


gc = solver.ls.getCost()
gv = solver.ls.getEnvVal()
print "Greedy cost, greedy val",gc,gv

print "greedyMap", solver.ls.choose_to_map()
t2 = time.time()
print "GREEDY TIME: ",t2-t1
a = [[0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 1], [0, 0, 0, 0, 0, 0,
    0, 0, 1, 0, 1, 1, 1, 1, 0], [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1,
    0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1], [0, 0, 0, 0, 0,
    0, 0, 0, 1, 1, 1, 0, 0, 1, 1], [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0,
    0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0], [0, 0, 0, 0,
    0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1], [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1,
    0, 0, 0, 0], [0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0], [0, 0, 0,
    0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 1, 0, 1, 0, 0,
    0, 1, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0], [0, 0,
    0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 1, 1, 1, 0]]

print "sum", sum([sum(x) for x in a])
solver.ls.map_to_choose(a)
```

```
solver.ls.printEnvVal()
print solver.ls.getEnvVal()
print solver.ls.getCost()
print "cosnstr type", type(solver.constraint)
print "cost type", type(solver.ls.getCost())
#     start = time.time()
#
#     global it
#
#     for i in range(7):
#         print "iteration", i
#         it = i
#         solver.valmax_sol2()
#         solver.bestSols.sort(key = lambda x: x[1], reverse = True)
#         #print solver.bestSols[0][1:]
#     print [s[1:] for s in solver.bestSols]
#
#     stop = time.time()
#
#     print "time: ", stop-start
#     print "constraint", solver.constraint
#     print solver.bestSols[0]
#
#     solver.ls.map_to_choose(solver.bestSols[0][0])
#
#     return solver.ls
ls = landscape.Landscape(15)
def mainLoop():
Pros = []


for i in range(12):
l = 25+(i*25)
p = Process(target = main,args=(l,))
Pros.append(p)
p.start()

for t in Pros:
t.join()


if __name__ == "__main__":
```

```
print randomSeeded.seed
mainLoop()

import winsound
winsound.Beep(554,1000)
```