Chapter 4

EVALUATION IN TEST DOMAINS

This chapter presents the results of applying the algorithms described in the previous chapter to three test domains. I show that in many cases the utility loss from PS Map approximation is comparable to that of online repair. However, the performance of the approximation algorithms varies between problem domains and between different problem configurations within the same problem domain. In both cases, the differences in the size and quantity of heterogeneous regions appear to be suggestive as reasons for the differences in approximation accuracy.

I evaluated the approximation algorithms using the Dynamic Traveling Salesman Problem (TSP), a dynamic variant of the 0-1 knapsack problem, and an elevator passenger transport domain. The TSP and knapsack domains are NP-complete problems that translate to a wide variety of practical problems. The elevator domain is a NP-hard challenge problem developed by the Association for the Advancement of Artificial Intelligence (AAAI) for its International Planning Competition (Coles et al., 2013).

Traditional TSP problems consist of a set of unordered locations, sometimes referred to as "cities," that must be ordered such that length of the route is minimized. In the dynamic variant, one or more additional locations become known after the initial ordering is computed, and must be incorporated into the route while minimizing computation time

and total route distance.

In the knapsack problem, one chooses from a set of given items, each with a weight and value characteristic, such that the total value of the knapsack is maximized and the total weight does not exceed a given weight constraint. I use the 0-1 variant, in which each item may be selected a maximum of one time. After computing an initial solution, I present one or more additional items with which the system may revise its solution.

The elevator domain defines the initial and desired locations of a set of passengers, and several elevators of varying speeds with which to transport passengers. The goal is to move all the passengers to their desired floors as cheaply as possible through efficient use of elevator movements. For my testing, I use a variant in which one or more passengers' initial location may change after the initial plan is computed.

My primary metric for evaluation is utility loss, measured as a fraction of the utility of a problem instance's high-quality[1] solution, as calculated by a heuristic solver. For example, if the total value of a high-quality knapsack solution is 100, and the solution retrieved from the approximated PS Map has a value of 95, then the utility loss for that specific solution is .05 (i.e., $\frac{100-95}{100}$). The evaluation of an approximated PS Map is the average utility loss over all of the discrete locations in the map. Thus, the evaluation for a PS Map over all problem instances is $\frac{\sum_{i \in map} heuristic_i - approx_i}{\sum_{i \in map} heuristic_i}$ where $approx_i$ and $heuristic_i$ are the utility of the solutions given by the PS Map and a heuristic planner, respectively, for a given problem instance i. Lower utility loss is preferred; the best approximated solution will have a utility loss of zero.

[1] As previously mentioned, "high-quality" refers to solutions generated by heuristic search methods. Technically, they are not guaranteed to be optimal; therefore, I do not use that term.

4.1 Traveling Salesman Problem

The traveling salesman problem (TSP) is a classic NP-complete problem in computer science in which cities must be ordered such that the length of the resulting route is minimized. The dynamic variant, the DTSP, allows for cities to be removed or added while the route is being traversed, creating a more challenging problem in which the route should be reoptimized in real time.

High-Quality PS Map For testing in the TSP domain, I generated three instances each of 5, 10, 20, 50, and 100-city DTSPs. One of the cities included with each of the DTSPs has a variable location. As the gold standard, high-quality PS Maps were generated via the Clark-Wright (Clarke and Wright, 1964) and Gillett-Miller (Gillett and Miller, 1974) algorithms, as implemented by the Drasys library.[2] I then removed errors stemming from heuristic-based solver by executing the SSS algorithm over the PS Map. As described in Section 3.7, this process tests each unique solution against each problem instance, resulting in a more accurate PS Map.

Experiment Parameters The PS Map approximations were generated using 19 sample rates between .0001 and .01. The experimental configurations were drawn from the permutations created by the cross product of the DTSP problem, sample rate, and approximation technique. Each run was executed ten times.

Online Repair Baseline To compare how well PS Map approximation techniques perform against traditional online repair, I implemented the insertion approach (Psaraftis, 1988). This approach incorporates new cities into the route by finding the nearest city and

[2] As of this writing, this library appears to no longer be publicly available. I have placed a copy of the original download at `http://www.umbc.edu/~holder1/or124.jar`

inserting the new city into the route either before or after the nearest city. Although it is not the best repair technique, it is well suited for online repair due to its speed. In this case, the repair accuracy was within the expected loss of utility provided by other DTSP online repair algorithms as discussed by Larsen. This will be discussed in more detail when presenting the experimental results.

4.2 Knapsack Problem

The knapsack problem is a combinatorial optimization problem in which a subset of items of variable weight and value are chosen such that the total value is maximized and the total weight falls below a given threshold. For this experiment, I use the 0-1 knapsack problem variant, in which either zero or one copies of each item may be placed in the knapsack. The knapsack is prepopulated with a set of items that utilize 396 dekagrams (dag) of the total knapsack capacity of 400 dag, and one or more items of varying value and weight is added to the pool of items.

High-Quality PS Map For the experiment, I defined a set of 22 items, each with known weight and value characteristics as shown in Table 4.1, from which to maximize the value of the knapsack while conforming to its maximum weight capacity. I defined one additional item, varying the weight and value from 1-100 inclusive to create 10,000 (100^2) problem instances. As a baseline, I solved all 10,000 (100^2) problem instances to generate a high-quality PS Map. As with the TSP domain, I applied SSS over the PS Map to reduce errors from the heuristic solver. A visualization of the resulting two-dimensional PS Map is depicted in Figure 4.1.

Experiment Parameters In my experiments, I found that large regions of the problem space tended to be the same, particularly as the problem instances' variable features

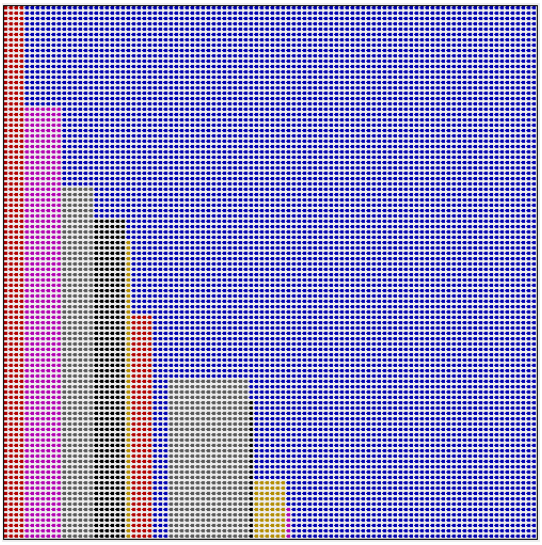

FIG. 4.1: High-quality PS Map for a knapsack problem. Best viewed in color.

tend towards infinity. To avoid positively skewing the results, I chose feature ranges to focus on the more heterogeneous regions of the problem instance space. For the two-dimensional experiment, I limited the problem space to problem instances with weights from 1-20, inclusive, and values from 50-70, inclusive. For example, when considering only one additional item, the first problem instance would consist of the static items plus an additional item with a (weight, value) pair (1,50), the second problem would consist of the static items plus an additional item with weight and value (2,50), and so forth, accounting for all possible combinations.

I then generated more complex problem spaces by adding multiple items of varying weight and value characteristics to the pool. Solving each of the resulting problem instances - consisting of the static items and two additional items - resulted in a four-dimensional PS Map consisting of two weight and two value dimensions. I generated a high-quality PS Map, solving all 176,400 ($20^2 \times 21^2$) problem instances.

Continuing, I generated an eight-dimensional problem space consisting of a weight and value axis for each of four variable items. The range of the weight was 16 to 20 inclusive and the range of the value was 31 to 35 inclusive, resulting in a problem space of $5^8 = 390,325$ problem instances. For each of the problem instances in the problem space, I created the full problem instance by adding the variable items indicated by the problem instance to the knapsack. For example, if a problem instance in the problem space is $(w_0, v_0, w_1, v_1, w_2, v_2, w_3, v_3)$, then I solved a knapsack problem consisting of the pool of items in Table 4.1 plus items with weight and value scores of $(w_0, v_0), (w_1, v_1), (w_2, v_2)$, and (w_3, v_3). I solved each of the knapsack problems and created a mapping from each of the instances in the problem space to each of the calculated solution, thus composing the PS Map.

Finally, I generated a second PS Map of an eight-dimensional problem space as above, but with the range of the weight expanded by one unit to 15 to 20, resulting in a problem space of $6^4 \times 5^4 = 810,000$ instances.

The approximation of all maps was done with the SVM+SBE approximation algorithm with an alpha value of 0.5 with sample rates ranging from .0001 to .001. With the alpha rate of 0.5, the initial sample rate is half of the allocated samples, leaving half for active sampling. As before, the evaluation of the approximation is the fraction of the utility lost with respect to the heuristically calculated heuristic solution.

4.3 Elevator Problem

The elevator domain was developed for the International Planning Competition. It specifies several elevators, floors, and passengers, and requires the planner to deliver the passengers from their starting floor to their destination floor at the lowest possible cost. Each elevator is either “fast” or “slow.” The slow elevators incur little cost for movement,

Object	Weight	Value
apple	39	40
banana	27	60
beer	52	10
camera	32	30
cheese	23	30
compass	13	35
glucose	15	60
map	9	150
note-case	22	80
sandwich	50	160
socks	4	50
sunglasses	7	20
suntan cream	11	70
t-shirt	24	15
tin	68	45
towel	18	12
trousers	48	10
umbrella	73	40
water	153	200
waterproof overclothes	43	75
waterproof trousers	42	70

Table 4.1: Knapsack static item pool

but more for stopping and starting. Conversely, the fast elevators incur more cost for movement, but little for stopping and starting. The planner specifies movements for the slow elevators, which are assigned to specific blocks of floors, and the fast elevators, which traverse the entire range of floors, but only stop at block centers and boundaries. For example, for a 12-floor problem with two slow elevators, one slow elevator will travel between the bottom six floors, and the other slow elevator will travel between the top six floors. The fast elevator will travel throughout the floors, but only stop at floors 0, 3, 6, 9, and 12. More formally, these features are specified with M and N parameters, which create a problem domain with M+1 total floors in blocks of N+1 floors, with fast elevators at floors that are multiples of $\frac{N}{2}$. Thus, in the example above, M is 12 (13 floors from 0 to 12) and N is 6 (two blocks each of seven floors, one from 0 to 6 inclusive, the other from 6 to 12 inclusive).

Typically, each planner submitted to the competition targets either the "optimal" or "sacrificing" track. The "optimal" track requires a planner to find the least costly means of transporting the passengers to their destinations. The "sacrificing" track does not require a planner to find the optimal plan, but only to find a feasible plan to deliver all the passengers. My experiments focused on the optimal track and used one of the more successful planners, the LAMA Planner (Richter and Westphal, 2010).

High-quality PS Map generation For this domain, I generated a 12-floor and two 24-floor elevator problems. The 12-floor problem contained two seven-floor blocks (M=12, N=6), two slow elevators, and one fast elevator. Each problem assumed two passengers with variable starting position, creating 169 problems to be solved with the LAMA planner. One 24-floor configuation consisted on six five-floor blocks (M=24, N=4), and the other contained four seven-floor blocks (M=24, N=6). The 24-floor problems vary the starting location of three passengers, thereby creating an high-quality map of 216 (i.e., 6^3) problem instances.

Similar to other hard problems, planners in this domain employ heuristics in order to solve these intractable problems, and thereby benefit from “smoothing” as described in Section 3.7: when generating the high-quality PS Map, each of the solutions are evaluated against each of the problem instances, and if necessary, the problem instance is assigned a new solution. This prevents the odd phenomenon of the occasional approximate solution having better utility than the “optimal” solution, which may skew the results.

Experiment Parameters My initial experiment used the 12-floor problem with three passengers, two slow elevators, and one fast elevator. I varied the starting positions of two passengers, resulting in a 169-instance problem space. In my initial experiment, there were too many unique plans, and the algorithms could not create classifications from the sampling. To make this domain appropriate for the algorithm, I abstracted the plans such that a plan that moved elevators to a specific floor was transformed into a plan to move elevators to the location of specific passengers, thus making a plan that could be applied to other problem instances. For example, consider a raw plan with the steps

```
(move-down-slow slow0-0 n6 n0)
(board p0 slow0-0 n0 n0 n1)
(move-up-slow slow0-0 n0 n3)
(leave p0 slow0-0 n3 n1 n0)
```

This specifies that, first, the slow elevator with id slow0-0 moves from floor 6 to floor 0. Next, the passenger with id p0 boards the elevator at floor 0, and the number of passengers increases from 0 to 1. Then the elevator moves from floor 0 to floor 3, and in the final step, the passenger leaves the elevator at floor 3.

In order to make this plan reusable, it is transformed into the more general

```
elevator slow0-0 picks up passenger p0
```

```
elevator slow0-0 drops off passenger p0
```

This allows the plan to be applied to problem instances in which the elevator and passengers are on floors other than those assumed by the raw plan.

In addition to abstracting the plans, I canonicalize the plan so that differences in the ordering of independent actions are not interpreted as distinct plans. For example, consider the plan below, annotated with action ids for ease of reference:

```
1: elevator slow0-0 picks up passenger p0
2: elevator slow1-0 picks up passenger p1
3: elevator slow0-0 drops off passenger p0
4: elevator slow1-0 drops off passenger p1
```

Note that the only dependencies are that action 1 must occur before action 3, and action 2 must occur before action 4. Thus, there are six potential plans[3] representing the same overall process. I canonicalize the plan by grouping together as many actions as possible that are performed by the same elevator. In this case, the resulting canonicalization is:

```
1: elevator slow0-0 picks up passenger p0
3: elevator slow0-0 drops off passenger p0
2: elevator slow1-0 picks up passenger p1
4: elevator slow1-0 drops off passenger p1
```

My subsequent experiments used a 24-floor elevator problem with six passengers, three of which had variable starting locations. One experiment used six fast elevators and three slow elevators, and the other used four slow elevators.

[3](1,2,3,4), (1,2,4,3), (1,3,2,4), (2,1,3,4), (2,1,4,3), and (2,4,1,3)

Online Repair As a baseline, I implemented an online repair algorithm. van der Krogt and de Weerdt (2005) describe plan repair as consisting of removing actions from the original plan that conflict with or impede achieving the new goal, followed by adding actions to the original plan that allow it to achieve the new goal. My baseline online re-planning algorithm is consistent with this methodology. The new goal changes the initial location of the passenger, and thus I consider all actions that reference that passenger as candidates for deletion. van der Krogt and de Weerdt suggest that heuristics should be used to determine if a candidate action should be deleted. My heuristic is a simple one: I only remove the candidate action if it refers to a passenger whose starting position has moved outside the range of the elevator used by the action. For example, consider an abstracted action *elevator slow0-0 picks up passenger p0*. Elevator slow0-0's range is floors n0 through n6. If this action is applied to a problem instance in which p0's starting position is n7 or above, then the action would be removed.

In the event that an action is removed, then I proceed with the second component of plan repair in, which I add actions to the original plan to achieve the new goal. There are two alternatives for continuation: either remove all subsequent actions that refer to the passenger and replan the entire route, or preserve the subsequent actions and replan the passenger route to comply with the constraints implied by the subsequent actions. In the case of the former, I generate a solution to transport the passengers whose actions were removed. In order to plan without the influence of the passengers whose actions have already been established, the initial starting conditions of those passengers is set to be equal to their destination location. In the case of the latter, the final condition is set to the location expected by the action that moves the passenger to its final destination. For example, if an action moves p2 from n6 to n2 to complete its journey, then the planner will set the final destination to n6.

Algorithm 8 Unrefinement

```
1: for each passenger p in plan P do
2:     if first action referencing p is invalid then
3:         remove all actions referencing p
4:     end if
5: end for
```

Algorithm 9 Refinement

```
1: actions ← generate plan for deleted passenger actions
2: parse and abstract actions
3: add actions to P
4: canonicalize P
```

4.4 Results

This section discusses the results of applying the algorithms within three test domains.

4.4.1 TSP

Results for the approximation algorithms SC, SC+AL, SSS, and SBE when applied to 100-city TSP high-quality maps are displayed in Figure 4.2. The algorithms with "-distance" and "-distanceSquared" represent variants in which the weighting of the neighbors in the nearest neighbor polling is decreased as a function of the distance or distance squared from the problem instance being classified. In addition to the showing fractional loss results, the graph highlights the range of fractional utility loss. That range is suggested by Larsen (2000) as the typical utility loss when using online repair. I also implemented a nearest neighbor DTSP solver to insert the variable city into the route. The loss from that online repair method was 1.97%, which is consistent with Larsen's range.

These results show that these algorithms are comparable to online repair performance, and that all the algorithms except for SC perform better than online repair at sample rates of .002 and above. It is quire reasonable that the alternate algorithms would perform better

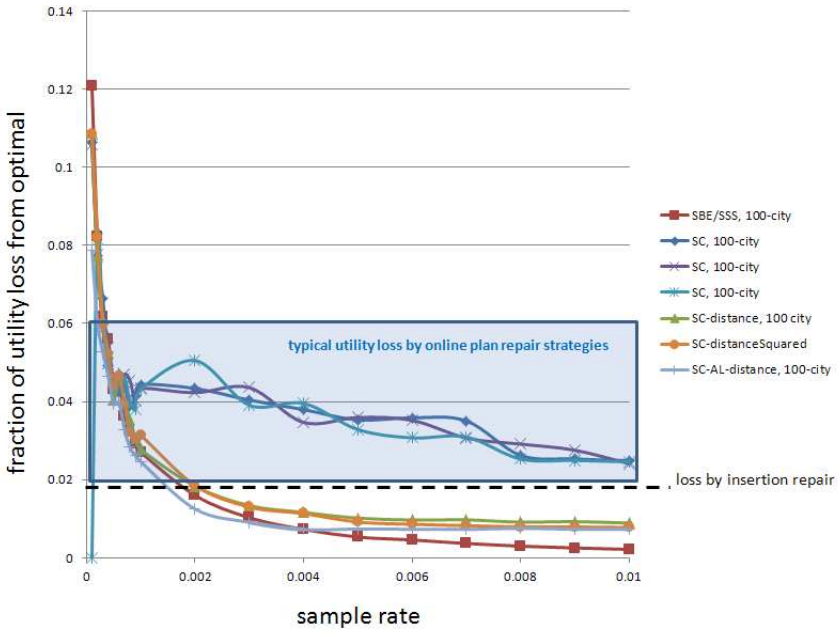

FIG. 4.2: Results of applying various approximation algorithms to the 100-city TSP domain.

than SC because SBE and SC+AL proactively attempt to find locations in the problem space, SSS, and SC+AL simply take more information into account during classification. The fact that the distance-biased versions of SC also before better means that problem instances that are closer to the problem instance being classified are far more likely to have a similar solution than more distance problem instances.

The results for the SC+bias applied to TSP of various sizes are shown in Figure 4.3. Again, the results are comparable to online repair, but not as good as other techniques. SC+bias has *bias factor* and *city radius* parameters that can be modified. Bias factor represents the degree to which to bias sampling to be near a city. The city radius indicates how close a problem instance has to be to a city to potentially benefit from the bias. Figure 4.4 shows utility loss results at sample rate .005 when SC+bias is applied with a range of parameter configurations. The results vary widely, and there does not appear to be any obvious correlation between specific parameter settings and the utility loss. This also appears reasonable. The goal of this algorithm was to attempt to exploit city locations as indicators of boundaries between solution regions. However, there are many solution regions that are not near cities, and thus this algorithm has uneven and limited benefit.

The early experiments demonstrate that SSS and SBE have the best performance. Figure 4.5 explores SSS and SBE's potential with additional problem sizes. The performance continues to be good for all problem sizes, but appears to converge more rapidly for the smaller problem spaces.

Focusing on the 100-city TSP as most representative of a complex domain, Figure 4.6 shows the performance of SVM+SBE against a 100-city TSP high-quality PS Map for two different alpha values. The performance is not as good as the SBE and SSS algorithms.

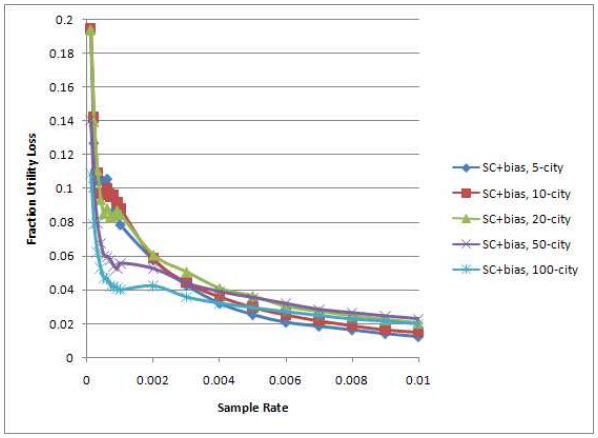

FIG. 4.3: Average utility loss of approximate PS Maps generated by SC+bias for DTSP problems of various sizes.

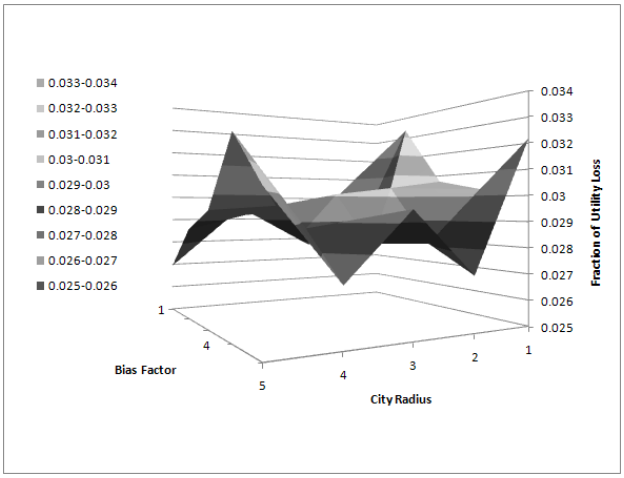

FIG. 4.4: Average utility loss of approximate PS Maps generated by SC+bias for 100-city DTSP problems at sample rate .005. SC-generated PS Maps generated under identical conditions have an average accuracy of .035.

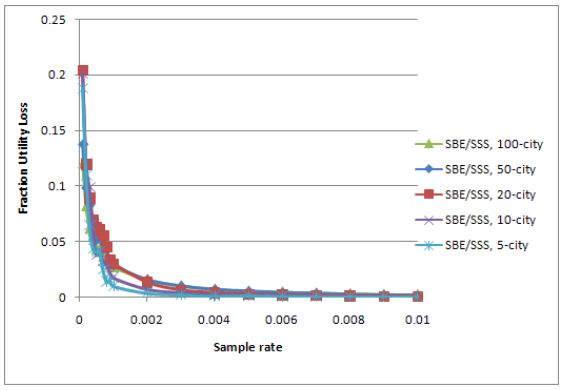

FIG. 4.5: Average utility loss of approximate PS Maps generated by SBE and SSS for DTSP problems of various sizes.

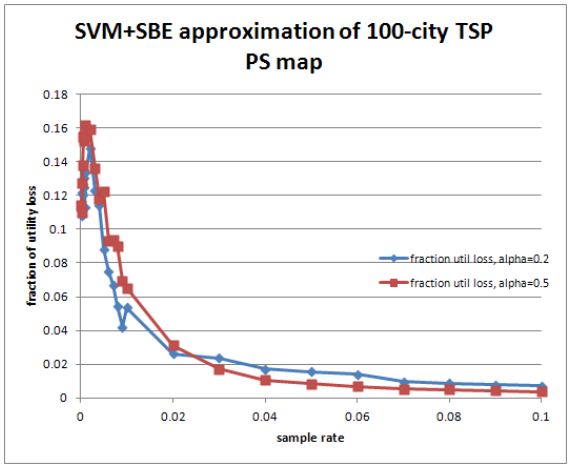

FIG. 4.6: Average utility loss of approximate PS Maps generated by SVM+SBE for 100-city DTSP problem.

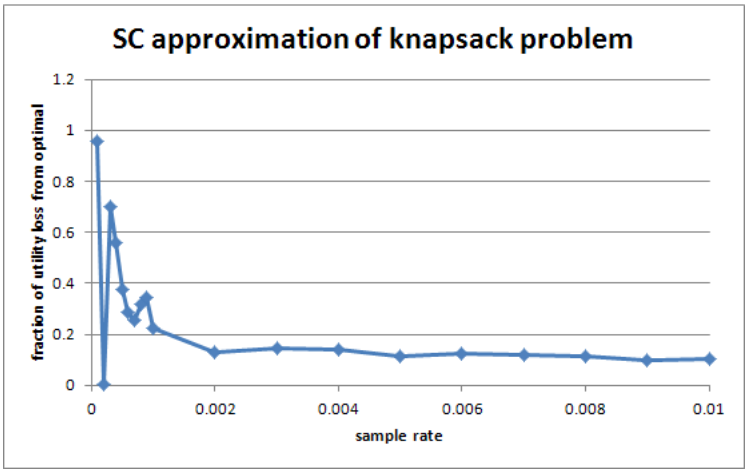

FIG. 4.7: SC applied to a two-dimensional knapsack problem domain.

4.4.2 Knapsack

The result of generating a high-quality PS Map is displayed in Figure 4.1. The map confirms an intuitive estimation of solutions: for problem instances in which the variable item's weight falls within the slack of the original solution, it is always included in the knapsack. Once the variable item's weight exceeds the available slack, it is excluded from the knapsack until it becomes valuable enough to replace an item currently in the knapsack. Moving along the weight dimension, the variable item remains in the knapsack until it becomes too heavy for its value to contribute to an optimal solution and is excluded from the knapsack. This pattern repeats, creating a set of solutions resembling a staircase of solutions, the edges of which represent a boundary in the solution space between where the variable item is included and excluded.

Figure 4.7 shows the results of applying SC to a knapsack problem with one variable item.

Figure 4.9 shows the results of applying SVM+SBE to a knapsack problem with two

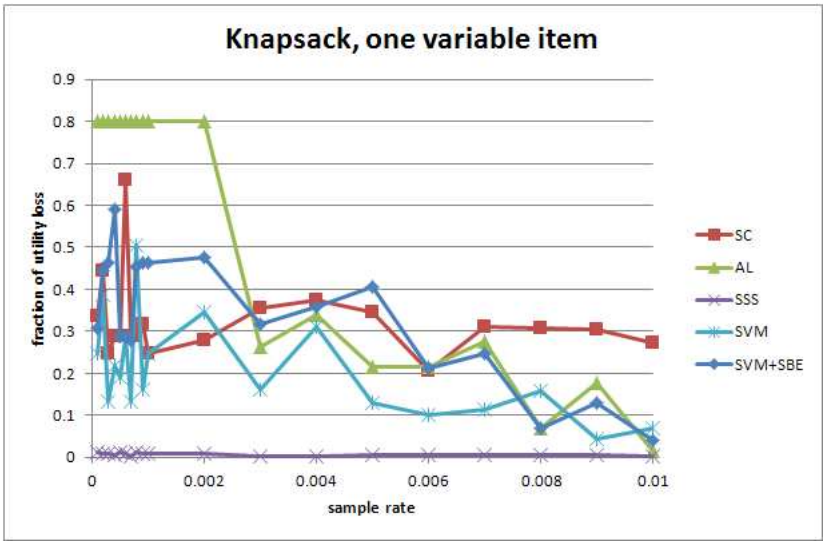

FIG. 4.8: Results of algorithms applied to a two-dimensional knapsack problem domain.

variable items. Because each item has a weight and height characteristic, this results in a four-dimensional PS Map. In this experiment, I limited the range of the weight and value of the items to [14,24] and [30,40], respectively, due to the computation time required to complete the experiment. The graph shows a loss of utility well under 1% at low sample rates.

The spikes in the SVM+SBE results are the effect of high variance that is a function of the manner in which SVM+SBE selects its sample points and the structure of the knapsack problem space. After the initial random sampling, SVM+SBE uses its additional samples to find problem instances that correspond to borders between pairwise solutions. Because of this, all of SVM+SBE's subsequent samples will be in a region of the problem space that is bounded by the initial sample set. An effect of this is that if the initial sample does not bound a region that represent all solutions, then no subsequent samples will discover those solutions. In this case, the region had a total of four solution. If the initial sample discov-

ered all four, then the average fraction utility loss was close to zero. If the initial sample discovered only three solutions and none were feasible with respect to the unrepresented problem instances, then it caused the average fraction utility loss to rise to around 0.15. If the initial sample discovered only two solutions, then it caused the average fraction utility to rise to around .45, indicating that the library did not have a feasible solution for almost half of the problem instances.

This effect is less pronounced in the other algorithms. For SC, SSS, and SVM the probability of excluding a solution region at a particular sample rate is smaller because, unlike SVM+SBE, all of the samples are used in the initial sample, rather than a fraction. For AL, which, like SVM+SBE, also reserves a fraction of its samples for targetted sampling, its scheme targets unrepresented regions thereby reducing the probability that a region of the problem space could be unrepresented in the discovered solutions. The last factor is the domain for which not all solutions are feasible for a given problem instance. In contrast to TSP in any solution can be applied to any problem instance, the knapsack problem domain defines a hard constraint - total weight - for which a solution can violate, rendering it unapplicable to a problem instance. This leads to large losses of utility because an infeasible solution has a utility close to zero[4], whereas in a domain like a TSP, a poor solution still does still contribute some portion of the optimal utility.

Figure 4.12 puts these results in context against various baselines. The taller blue bars represent the fraction of utility lost if one were to assume a PS Map consisting of a single solution. Because the high-quality PS Map had seven solutions, there are seven cases represented in the graph. In this scheme, it is possible that the penalty for plan infeasiblity could domainate the error results. The shorter red bar represents the result of applying a default solution to a problem instance, but allows the system to choose an alternate solution

[4]To avoid division by zero errors, the lowest utility in the knapsack problem domain is 1

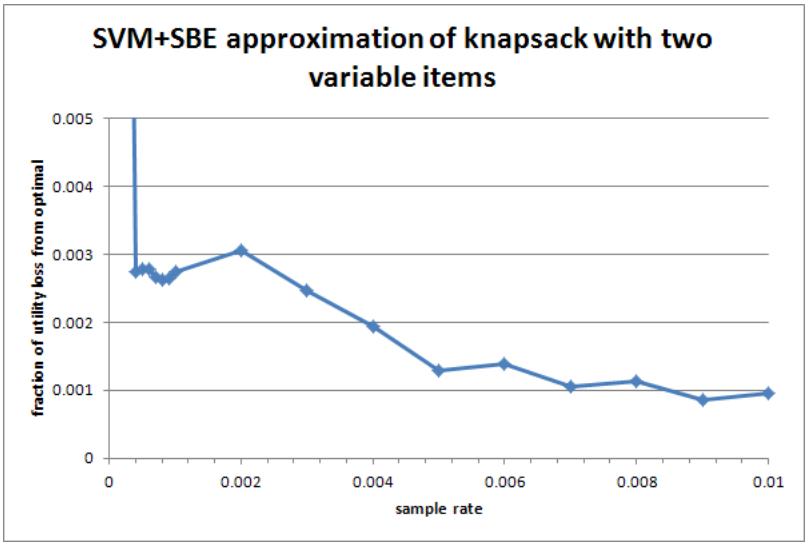

FIG. 4.9: SVM+SBE applied to a four-dimensional knapsack problem domain. Problem space is limited to weight 14-24 inclusive and value 30-40 inclusive.

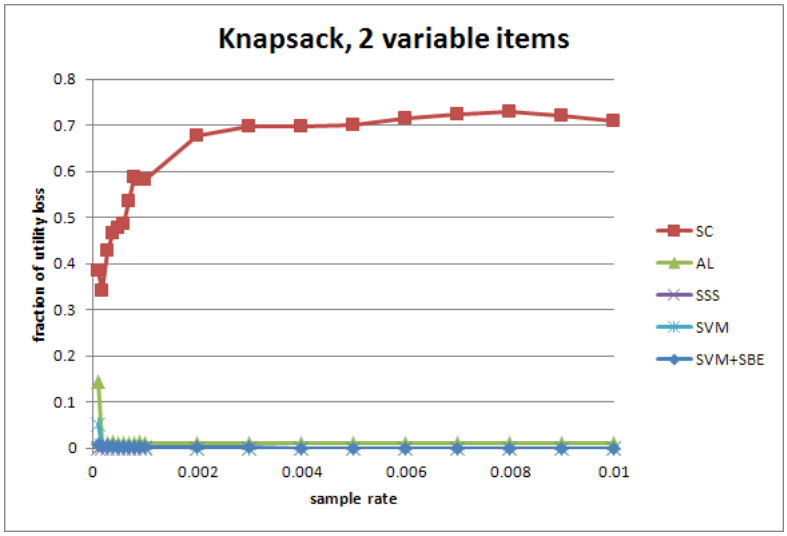

FIG. 4.10: Results of algorithms applied to a four-dimensional knapsack problem domain.

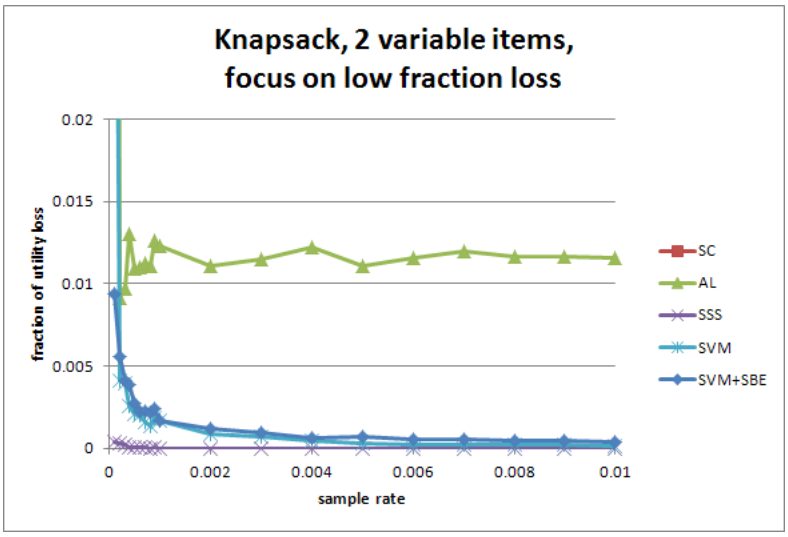

FIG. 4.11: Results of algorithms applied to a four-dimensional knapsack problem domain, highlighting low utility loss.

if the default solution violates the weight threshold of the knapsack. In this case, a feasible plan is randomly chosen. The upper dotted line represents the fraction of utility loss of SVM+SBE at a sample rate of .004. The lower dotted line represents the identical loss of the online repair method as well as when sampling at a rate of .006 using SVM+SBE. The online repair method is a greedy solver that selects the items with the highest value:weight ratios. This demonstrates that the performance of SVM+SBE algorithm when sampling at rate of .006 is roughly equivalent to that of the online repair technique.

4.4.3 Elevator

Results from the 12-floor elevator problem domain are displayed in Figure 4.13. The dotted lines represent the fractional utility loss of three independent runs of the online repair algorithm described in Algorithms 8 and 9. The solid lines represent the results of applying the SVM+SBE approximation algorithm with various SVM kernels and the SSS

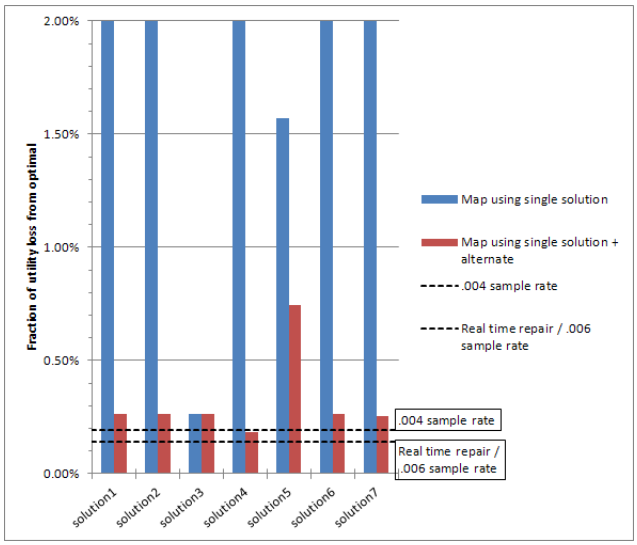

FIG. 4.12: Results of applying SVM+SBE to a knapsack with two variable objects. Dotted lines indicate fractional utility loss for online repair, .004 sample rate, and .006 sample rate as labeled. Bars indicate the fractional loss when using a default map consisting of either a single default solution, or a default solution and the best found feasible solution.

algorithm. The results demonstrate that the SSS algorithm has less fractional utility loss than the online algorithms, but the various SVM+SBE algorithms generally perform worse than the online repair algorithms.

Results from the 24-floor elevator domain experiments are shown in Figures 4.14 through 4.17. These results show that for each problem configuration SSS performs better than SVM+SBE. Additionally, the algorithms perform better against the problem configuration with fewer elevators. This is not unexpected, given the nature of the problem space of each configuration and the algorithms used. In all of the configurations, the problem spaces have homogenous regions, but they are small, which can make it difficult for an SVM-based algorithm to converge and find the appropriate boundaries. However, those small regions are not a disadvantage for the SSS algorithm, because it chooses a solution for each unsolved problem instance, rather than attempting to find groupings like SVM+SBE. This

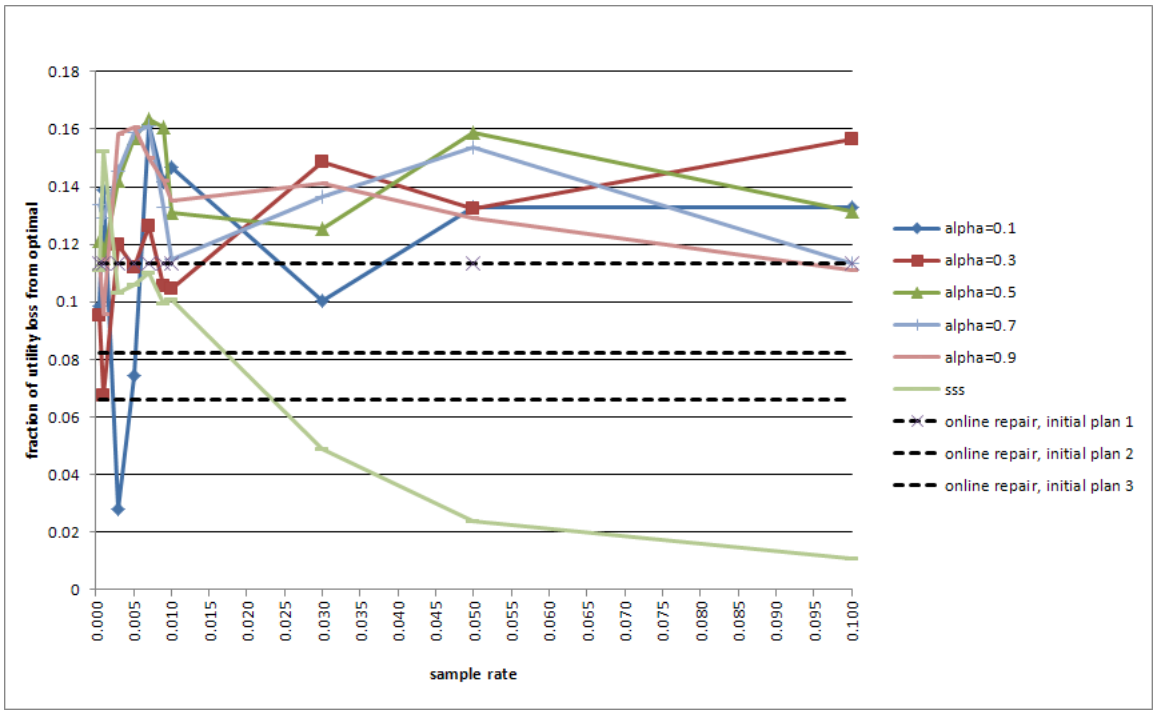

FIG. 4.13: Results of applying the SVM+SBE algorithm to a 12-floor elevator problem consisting of 2 slow elevators, 1 fast elevator, and 3 variable passenger starting locations. Dotted lines represent the utility loss of online repair. Solid lines represent approximations using various alpha values.

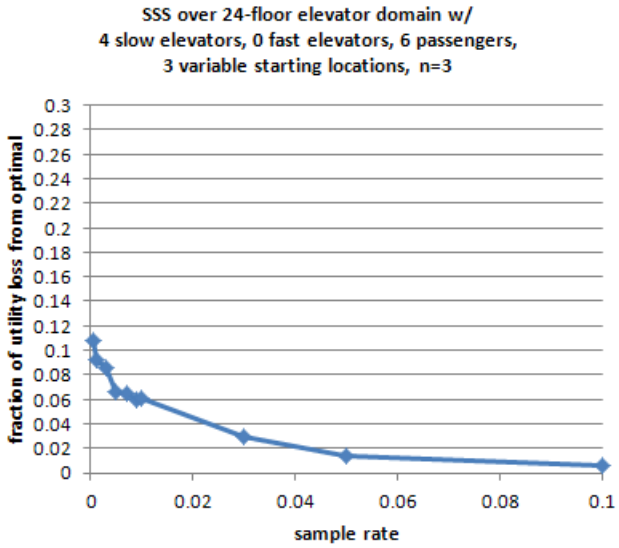

FIG. 4.14: Results of applying the SSS algorithm to a 24-floor elevator problem consisting of 4 slow elevators, 6 passengers, and 3 variable passenger starting locations.

same logic is applicable to the generally better results for the problem configuration with fewer elevators. In the configuration with four slow elevators, the homogenous regions are larger than in the problem space with six slow elevator, and thus the SVM+SBE algorithm performs better. Because there are less total solutions in the configuration with fewer elevators, the SSS algorithm performs better as well.

4.5 Analysis & Discussion

The results generally show little difference between approaches at low sample rates. In fact, the results tend to be highly volatile, perhaps due to the high dependence on small number of samples, which leads to large variances in the solutions available for the algorithms to consider.

SC tends to be a reasonable approach with the benefit that it is very simple to imple-

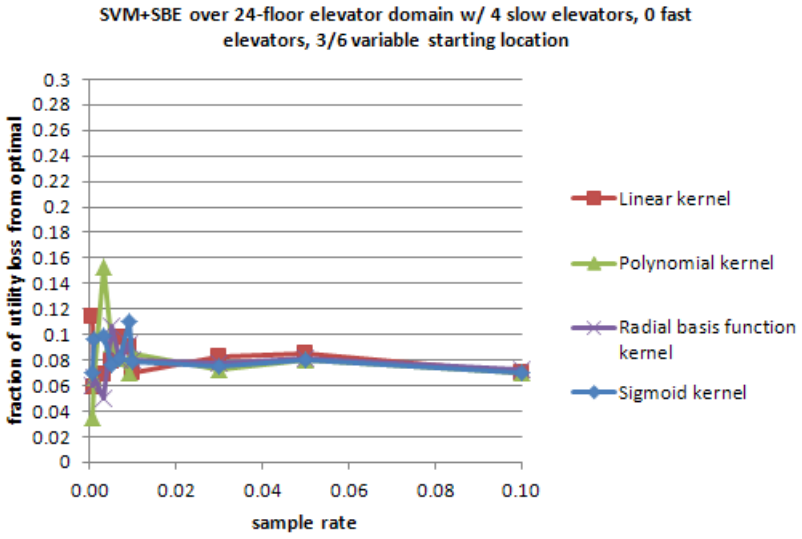

FIG. 4.15: Results of applying the SVM+SBE algorithm to a 24-floor elevator problem consisting of 4 slow elevators, 6 passengers, and 3 variable passenger starting locations.

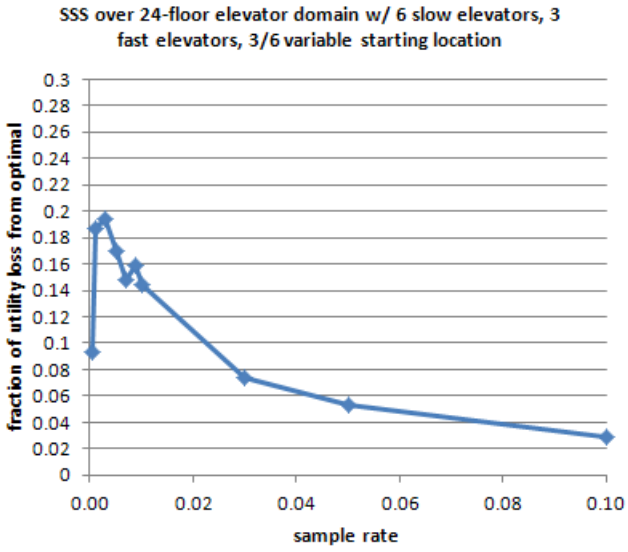

FIG. 4.16: Results of applying the SSS algorithm to a 24-floor elevator problem consisting of 6 slow elevators, 3 fast elevators, 6 passengers, and 3 variable passenger starting locations.

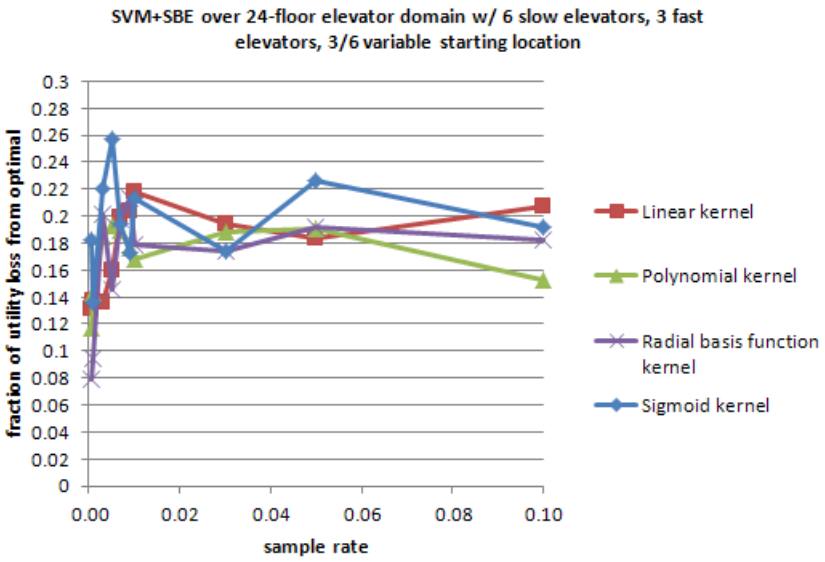

FIG. 4.17: Results of applying the SVM+SBE algorithm to a 24-floor elevator problem consisting of 6 slow elevators, 3 fast elevators, 6 passengers, and 3 variable passenger starting locations.

ment. SC+bias shows the ability to improve on SC, however, it is not clear how to tune its parameters to achieve consistently good results. SBE clearly is the best performer. However, my implementation of SBE is limited to two dimensions. SSS has the same results as SBE, but SSS is more computationally intensive, potentially leading to scaling issues in large problem spaces. The SVM algorithm, although not shown here, did not provide consistently good results. However, the results improved in SVM+SBE because the SBE component helps to inform the SVM's vector calculations. SVM+SBE did not achieve the results of SBE or SSS, but it does have the advantage of being less computationally intensive and being applicable to domains of more than two dimensions.

The results of all algorithms appear sensitive to problem domain characteristics. In the case of TSP, for example, larger size and smaller quantity of homogeneous solutions regions generally resulted in better performance by the algorithms.

Most importantly, the results do demonstrate that these techniques are useful as an alternate to online plan repair. At the appropriate sample rate, performance tends to be comparable, and sometimes better, than the online solution. The benefit of my approach is that, assuming the ability to compute the necessary library before the environment changes, the new plan can be accessed much more rapidly than the online repairer can calculate a new plan. These tradeoffs are discussed in more detail in the next chapter.