MITSUBISHI ELECTRIC RESEARCH LABORATORIES
http://www.merl.com

BubbleSearch: A Simple Heuristic for Improving Priority-based Greedy Algorithms

N. Lesh, M. Mitzenmacher

TR2005-114 December 2005

Abstract

We introduce BubbleSearch, a general approach for extending priority-based greedy heuristics.

Elsevier - Information Processing Letters

BubbleSearch: A Simple Heuristic for Improving Priority-Based Greedy Algorithms

N. Lesh* M. Mitzenmacher†

Abstract

We introduce BubbleSearch, a general approach for extending priority-based greedy heuristics. Following the framework recently developed by Borodin et al., we consider *priority algorithms*, which sequentially assign values to elements in some fixed or adaptively determined order. For example, the First-Fit-Decreasing algorithm for bin packing considers the items in order of decreasing size, and assigns each item to the lowest-numbered bin in which it will fit. For many problems, there exist priority algorithms that are good heuristics and/or provably competitive algorithms.

BubbleSearch extends priority algorithms by selectively considering additional orders near an initial good ordering. While many notions of nearness are possible, we explore algorithms based on the Kendall-tau distance (also known as the BubbleSort distance) between permutations. Our contribution is to elucidate the BubbleSearch paradigm and experimentally demonstrate its effectiveness.

1 Introduction

Recently, Borodin, Nielsen, and Rackoff introduced a framework that encompasses a large subclass of greedy algorithms, dubbed *priority algorithms* [1, 5]. Priority algorithms have historically been used for and are especially effective at solving scheduling and packing problems. We provide a formalization below, but the essential contribution of the framework is to represent priority algorithms with two functions: a *placement* function that sequentially assigns values to elements, and an *ordering* function that determines the order in which elements are assigned values. For example, the First-Fit-Decreasing algorithm for bin packing sequentially places items in order of decreasing size, where each item is placed into the lowest-numbered bin in which it will fit. Priority algorithms are commonly used because they are simple to design and implement, run quickly, and often provide very good heuristic solutions. Additionally, many priority algorithms have been shown to have a bounded competitive ratio.

In work on heuristics and metaheuristics, the following idea has appeared in many guises: while the solutions produced by greedy algorithms are typically quite good, there are often better solutions "nearby" that can be found with small computational effort by perturbing the greedy construction. Early work in this area was done by Hart and Shogan [9]. This idea is also part of the basis for Greedy Randomized Adaptive Search Procedures, or GRASP algorithms (see, e.g. [16] and the references therein). GRASP algorithms have two phases. The construction phase generates solutions in a greedy fashion, introducing some randomness to obtain many solutions. The local search phase attempts to improve these solutions via a local search heuristic, in order to reach a local optimum.

We introduce a generic extension to the priority algorithm framework for specifying in which order variations of the priority algorithm's ordering should be evaluated. This approach yields algorithms for finding better solutions given additional running time. The GRASP literature provides several possible

*Mitsubishi Electric Research Laboratories, 201 Broadway, Cambridge, MA, 02139. lesh@merl.com

†Harvard University, Computer Science Department. michaelm@eecs.harvard.edu. Supported in part by NSF CAREER Grant CCR-9983832 and an Alfred P. Sloan Research Fellowship. This work was done while visiting Mitsubishi Electric Research Laboratories.

methods for perturbing the ordering that can be easily represented within our framework. In this paper, we focus on a different method for perturbing the original ordering based on the Kendall-tau distance. The Kendall-tau distance is also known as the BubbleSort distance, which is why we call our approach BubbleSearch. Additionally, as described in Section 2.5, BubbleSearch allows a simple mechanism for replacing the base ordering to perturb in future iterations. The mechanism substantially improves the performance of our algorithms in some of the examples below.

BubbleSearch is domain independent: our generic implementation treats the ordering and placement functions of a priority algorithm as black boxes, and can therefore easily extend any priority algorithm that uses these components. While the particular ordering and placement functions for a priority algorithm must be constructed to be effective in that problem domain, BubbleSearch does not require any additional domain-specific knowledge.

We provide results from case studies for several problems: rectangular strip packing, jobshop scheduling, and edge crossing. (We have results for other problems as well, but omit them for lack of space.) We show that BubbleSearch can significantly improve upon the priority algorithm it extends after evaluating only a small number of orderings. In particular, for all three problems, the average result of randomized BubbleSearch after evaluating 100 orderings was always at least 20% closer to the optimal than the average result of the original priority algorithm. The results continue to improve as BubbleSearch evaluates more orderings. We also compare BubbleSearch to the naïve randomized strategy that applies the placement rule to permutations chosen uniformly at random. This approach is well known and commonly tried, but is usually ineffective. Finally, BubbleSearch can also be seen as an alternative to the typical randomized greedy strategies used in the construction phase of GRASP algorithms. We therefore compare BubbleSearch to suggested strategies from the GRASP literature. We find that BubbleSearch performs at least as well or better in almost all cases.

To summarize, our contributions are the following:

- A consistent formalization for BubbleSearch and priority algorithms. This formalization leads to domain-independent implementation, as well as several interesting and novel variations. While we have not used this formalization to prove theoretical results for BubbleSearch, we hope that it may be a step toward theoretical results in the future.
- A code base for implementing BubbleSearch algorithms. This code base is freely available for research purposes.
- The introduction of a replacement mechanism that improves both BubbleSearch and similar GRASP strategies.
- Experimental results demonstrating the effectiveness of BubbleSearch in general, including its effectiveness compared to similar standard heuristics.

2 Priority Algorithms and BubbleSearch

2.1 Priority Algorithms

We first introduce terminology for optimization problems. A *problem* is characterized by a universe U of elements and a universe V of values. A *problem instance* includes a subset of elements $E \subset U$. A *solution* is a mapping of elements in E to a value in V. We use the term *partial solution* to emphasize that only a subset of the elements in E may have values. The problem definition also includes a total ordering (with ties) on solutions.

Along the lines of Borodin et al., we define a *placement function* f as a function which maps a partial solution and an element to a value for that element.

A *fixed priority algorithm* can be defined from a placement function f and an *ordering function* o. The input to the priority algorithm is a problem instance I. The ordering function maps I to an ordered sequence of the elements in I. Let $o(I)$ be $x_1, \ldots, x_n$. Let S_0 be an empty mapping, and for $1 \leq i \leq n$, let S_i be the mapping defined by extending S_{i-1} by adding $v(x_i) = f(S_{i-1}, x_i)$. The priority algorithm returns the solution S_n. (In some cases, a partial solution S_k could be returned; this will not be the case for any algorithms in this paper.) The key points are that a fixed priority algorithm requires an ordering of all elements in the problem instance; the algorithm is greedy, in that the value assigned to x_i is a function only of previously assigned elements; and the value of an element is fixed once decided.

An *adaptive priority algorithm* is similar, except that elements are reordered after each placement. The ordering function o now takes as input a problem instance I and a partial solution S, and returns an ordered list of all elements in I not assigned a value in S. Let S_0 be an empty mapping, and for $1 \leq i \leq n$ let x_i be the first element of $o(I, S_{i-1})$. That is, x_i is the first unplaced element in the adaptive ordering based on the values assigned to the previously placed elements. As above, let S_i be the mapping defined by extending S_{i-1} with $v(x_i) = f(S_{i-1}, x_i)$.

A simple example distinguishing these variations is provided by the well-known heuristics for vertex coloring. The problem is to color the vertices of the graph so that no two endpoints of an edge have the same color; the goal is to accomplish this with the minimum number of colors. Let the set of possible colors be $\{1, 2, \ldots\}$. Here the elements are the vertices. (Note a problem instance also contains information in addition to the set of elements, namely the graph that relates the vertices.) The values are the colors. A natural placement function, given a partial solution and an additional vertex, is to assign the vertex the lowest-numbered color that yields a valid coloring consistent with the partial solution. The standard fixed priority algorithm orders the vertices by decreasing degree. The related adaptive priority algorithm orders the vertices by decreasing remaining degree, where the remaining degree does not count edges that involve a vertex that has already been colored in the partial solution.[1]

2.2 Anytime Priority Algorithms

An *anytime priority algorithm* is an extension of a fixed or adaptive priority algorithm that continues to apply its placement function to new orderings of the problem elements. It can be halted at any point and will return the best solution it has evaluated so far. Note that according to our definitions, a placement function yields a solution if applied to *any* ordering of the problem elements.

In fixed and adaptive priority algorithms, the highest-priority element is placed at each step. Our generalization is to choose elements other than the highest-priority element at each step. To more easily describe this generalization for both fixed and adaptive priority algorithms, we introduce the notion of a *decision vector* $(a_1, a_2, \ldots, a_n)$. The number a_j represents which remaining element should be considered at each step; if $a_j = k$, then the kth-highest-priority element is placed in step j. Specifically, for $1 \leq i \leq n$, we modify the above definition of S_i by letting x_i be the a_jth element of $o(I)$ for fixed priority algorithms or of $o(I, S_{i-1})$ for adaptive priority algorithms. It follows that we require $1 \leq a_i \leq n - i + 1$ in the decision vector.

With the above formulation, we can characterize priority algorithms by how they choose decision vectors to evaluate. For example, standard fixed and adaptive priority algorithms evaluate a single ordering corresponding to the all-ones vector $1_n = (1, 1, \ldots, 1)$. A simple anytime priority algorithm repeatedly applies the placement function to random orderings. In terms of decision vectors, this corresponds to choosing each a_i independently and uniformly at random from $[1, n - i + 1]$.

[1] Both algorithms have the issue that there may be ties according to this ordering. Ties can be broken in an arbitrary or random fashion.

2.3 GRASP algorithms

As previously mentioned, GRASP algorithms have a construction phase that generates solutions in a greedy fashion. This construction phase can generally be described in our framework above. Although not usually described as such, many GRASP implementations can be represented with an ordering function and a placement function. In our terminology, the construction phase randomly chooses each a_i in the decision vector uniformly over $\min(k, n-i+1)$ for some fixed constant k. That is, at each step, one of the top k elements remaining is selected. Usually k is two or three.

Another common GRASP approach is to assign each element a *score* after each placement, where elements with high scores are more desirable. That is, in the adaptive setting, there is a scoring function s which takes as input a problem solution I and a partial solution S and returns a score for all elements in I not assigned a value in S. The scoring function can be seen as a generalization of the ordering function; often an ordering function is derived from some scoring function, although in our framework a scoring function is not necessary. Let ℓ_{i-1} and u_{i-1} be the lowest and highest scores after $i-1$ elements have been placed. Then the next element be to be placed is uniformly selected from all elements whose score is at least $u_{i-1} - \alpha(u_{i-1} - \ell_{i-1})$, where $\alpha < 1.0$ is a predetermined parameter. (Alternatively, α can be changed dynamically as the process runs. See [16] for more information.) In this case the possible range of values for each a_i depends on the results of the scoring function, and hence the decision vectors cannot be computed in advance.

Notice that for both of these methods there are some orderings of elements which can never be evaluated.

2.4 Kendall-tau distance

Recall that our insight is to explore solutions close to the one determined by the priority algorithm, which corresponds to evaluating decision vectors close to the all-ones vector. While there are many notions of closeness, we found Kendall-tau distance to be well justified theoretically and useful algorithmically.

The Kendall-tau distance is defined as follows. Consider two orderings π and σ of an underlying set $\{x_1, \ldots, x_n\}$, so that $\pi(i)$ is the position of x_i in the ordering π. Then

$$d_{Ken}(\pi, \sigma) = \sum_{1 \le i < j \le n} I[\pi(i) < \pi(j) \text{ and } \sigma(i) > \sigma(j)],$$

where $I[z]$ is 1 if expression z is true and 0 otherwise. The Kendall-tau distance is the minimum number of transpositions needed to transform π to σ, and hence it is also referred to as the BubbleSort distance.

Suppose that we have an ordering of elements $\pi = x_1, \ldots, x_n$. Consider the ordering σ determined by a decision vector a applied to π. The following lemma is easily verified.

Lemma 1 *The value $|a - 1_n|$ is the Kendall-tau distance between π and σ where the norm is the L_1 distance.*

2.5 BubbleSearch

We now present BubbleSearch, a generic approach for producing anytime priority algorithms from fixed or adaptive priority algorithms. In particular, we describe natural exhaustive and random methods for determining which decision vectors to evaluate.

An exhaustive anytime algorithm must eventually consider all possible $n!$ decision vectors with $1 \le a_i \le n - i + 1$. We refer to this set of vectors as $\mathcal{O}_n$. As considering all $n!$ decision vectors is likely to be too expensive computationally, the order in which the vectors are evaluated is important to performance. Our *exhaustive BubbleSearch* algorithm uses the following order. We define a total ordering on $\mathcal{O}_n$:

1. $a < b$ if $|a - 1_n| < |b - 1_n|$;
2. if $|a-1_n| = |b-1_n|$, then $a < b$ is true if and only if a comes before b in the standard lexicographic ordering for vectors.

Note that in the case where $|a - 1_n| = |b - 1_n|$, alternative total orderings could be used to determine when $a < b$. Further, considering decision vectors in this order is easy to implement in practice.

The intuition for this total ordering on decision vectors derives from fixed priority algorithms. Given a fixed priority algorithm, let us call the ordering of elements in a particular problem instance determined by the ordering function the *base ordering*. Our exhaustive BubbleSearch algorithm searches outward from the base ordering in order of increasing Kendall-tau distance.

For many problems, small perturbations to an element ordering tend to make only a small difference in solution quality. In this case, larger perturbations may be more effective. This motivates our *randomized BubbleSearch* algorithm, which chooses decision vectors to try at each step randomly according to some probability distribution. A decision vector a is chosen with probability proportional to $g(|a - 1_n|)$ for some function g. We suggest using the function $(1 - p)^{|a-1_n|}$ for some parameter p, which determines how near the base ordering our randomly chosen orderings tend to be. In the case of fixed priority algorithms, this has a natural interpretation: if τ is the base ordering, then at each step an ordering σ is chosen with probability proportional to $(1 - p)^{d_{Ken}(\tau,\sigma)}$.

We note that we have found that a similar idea has appeared previously in the GRASP literature. Bresina [6] considers choosing candidate elements from a rank ordering in a biased fashion; one of the bias functions suggested is to chose at each step the ith ranked item with probability proportional to e^{-i}. Our suggestion is an obvious generalization, choosing the ith item with probability proportional to $(1 - p)^{-i}$ for some p. The generalization allows improved performance. In our experiments below, the best performing value of p varied by application.

One reason for choosing this function is that it has a simple algorithmic implementation. To choose a decision vector according to the distribution above, we determine each a_i as follows. Let q be 0, initially. Repeat the following: with probability p, terminate and output $a_i = q + 1$; otherwise increment q by 1 modulo $n - i + 1$. Of course more efficient implementations are possible.

Both the exhaustive and randomized BubbleSearch algorithms apply equally well to dynamic priority algorithms. Because the ordering function o changes as elements are placed, we cannot directly tie this ordering to the Kendall-tau distance between orderings, as we can in the fixed priority case. We feel however that this is approach is still quite natural for dynamic priority algorithms.

BubbleSearch offers a great deal of flexibility. For example, there can be several ordering functions, with the algorithm cycling though them (or running on several ordering functions in parallel). Similarly, there can be several placement functions. Local search can be done as postprocessing, as with GRASP; as a specific example, we can truncate the last k fields of the decision vector, and exhaustively search over all completions of the first $n-k$ fields. We have tried many of these variations in our experimental work.

The most successful variant of BubbleSearch we have found applies to fixed priority algorithms. In such algorithms, the base ordering does not need to remain static. If an ordering leads to a new best solution, it can replace the base ordering (or be used as an additional base ordering). Decision vectors are then applied from the new base ordering. We refer to the variation that replaces the base ordering as *BubbleSearch with replacement*, which is apparently novel and performs very well in the experiments described below.

Replacement can be seen as a simple type of memory or learning added to the BubbleSearch approach. While other types of memory or learning have been suggested for GRASP algorithms [16], this simple but useful technique does not appear to have been explored. However, it can be easily incorporated into many GRASP strategies.

3 Applications of BubbleSearch

In this section, we present experiments demonstrating the effectiveness of BubbleSearch, including its effectiveness compared to GRASP heuristics. Because BubbleSearch is simple and generic, we believe these results provide a strong case for the utility of BubbleSearch for any problem on which a priority algorithm is currently used. We note that we have experimented with both exhaustive and random BubbleSearch, but because random BubbleSearch performs better in all the cases below, we present only the results for the random BubbleSearch method.

For each experiment, we compare BubbleSearch to the naive algorithm that repeatedly places the elements in a random order, and to three variations of GRASP. We refer to the variation that returns one of the top k elements as GRASP-k. The second variation is GRASP-k with replacement. The third variation is the one described above that uses a scoring function on elements, which we refer to as GRASP-α. In all cases below, our ordering function was based on a scoring function and so it was easy to apply GRASP-α. Notice that replacement cannot be applied to GRASP-α, as it is not based on fixed priorities.

For these experiments, we use the same (unoptimized) Java implementation of BubbleSearch. We developed our generic BubbleSearch code using the HuGS Toolkit, Java middleware for rapidly prototyping interactive optimization systems [12]. (For the packing application described below, we have also implemented an interactive system that includes BubbleSearch [15].) This code provides an interface for defining the domain-specific components for each application, including the problem definition, placement function, and ordering functions. Thus our generic BubbleSearch code treats the priority algorithm's components as black boxes. In each iteration, BubbleSearch evaluates one ordering. Our BubbleSearch algorithm always first applies the placement function to the base ordering before considering perturbations. When more than one ordering function is provided, the BubbleSearch algorithm iterates through them in a round-robin fashion. This code is freely available for research or educational purposes.[2]

3.1 2D Strip Packing

Packing problems involve constructing an arrangement of items that minimizes the total space required by the arrangement. In this paper, we specifically consider the two-dimensional (2D) rectangular strip packing problem. The input is a list of n rectangles with their dimensions and a target width W. The goal is to pack the rectangles without overlap into a single rectangle of width W and minimum height H. We further restrict ourselves to the orthogonal, fixed-orientation variation, where rectangles must be placed parallel to the horizontal and vertical axes and the rectangles cannot be rotated. (We have considered the variable-orientation variation, in which rectangles can be rotated by 90 degrees, and obtained similar results [15].) For all of our test cases, all dimensions are integers. Even with these restrictions, 2D rectangular strip packing, like most packing problems, is NP-hard.

3.1.1 Applying BubbleSearch

The *Bottom-Left* (BL) placement function, introduced in [3], is probably the most widely studied and used heuristic for placing rectangles for the fixed-orientation problem. BL sequentially places rectangles first as close to the bottom and then as far to the left as they can fit relative the already-placed rectangles. The BL heuristic has been shown to be a 3-approximation when the the rectangles are sorted by decreasing width (but the heuristic is not competitive when sorted by decreasing height) [3]. While BL cannot find the always find the optimal packings even if applied to every permutation

[2]Contact lesh@merl.com for details.

	BubbleSearch p value								
replacement	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
without	8.033	7.433	6.533	6.300	6.033	5.900	6.400	6.567	7.767
with	9.000	8.367	7.567	6.400	6.400	6.167	6.267	6.267	6.767

Table 1: Tuning the input probability for randomized BubbleSearch on Packing

iter-ations	random	BubbleSearch		GRASP-k		GRASP-α
		w/o replace $p = 0.6$	w/ replace $p = 0.6$	w/o replace $k = 3$	w/ replace $k = 2$	$\alpha = 0.3$
4 (base)	N/A	6.25	6.25	6.25	6.25	6.25
100	12.20	4.50	4.52	4.48	4.65	4.85
500	10.65	3.92	3.97	4.05	4.18	4.33
1000	9.93	3.77	3.85	3.90	4.13	4.20
5000	8.90	3.50	3.40	3.65	3.65	3.63
10000	8.70	3.35	3.32	3.50	3.38	3.58
15000	8.40	3.30	3.22	3.48	3.35	3.50
20000	8.28	3.27	3.15	3.35	3.28	3.43

Table 2: BubbleSearch applied to Packing problems

of rectangles [3, 7], it performs very well in practice when applied under the ordering (or scoring for GRASP-α) functions of decreasing height, width, perimeter, and area [10]. We refer to taking the best result from BL on all four of these ordering functions as Bottom-Left-Decreasing (BLD).

The BLD priority algorithm has been shown to handily outperform the best-published implementations of more sophisticated search techniques, such as simulated annealing or genetic algorithms. For more details see the thesis of Hopper [10].

We evaluated BubbleSearch using BL and the four ordering functions on benchmarks recently developed by Hopper. All instances in this benchmark can be packed with no empty space into a square with side length 200. The instances are derived by recursively splitting the initial large rectangle randomly into smaller rectangles; for more details, see [10]. This benchmark set contains problems with size ranging from 17 to 197 rectangles. We use the non-guillotinable instances from this set, collections N1 (17 rectangles) through N7 (197 rectangles), each containing 5 problem instances. We found the instances in N1-N3 easy to solve using exhaustive search [14].

For all our experiments, we pack the rectangles into a strip of width 200, so that the optimal height is 200. We score a solution by the percentage over optimal. Each entry in our tables is an average over all relevant problem instances.

We tuned our BubbleSearch algorithms using collections N1-N3 and tested them on N4-N7. We ran randomized BubbleSearch with and without replacement on N1-N3 for 10,000 iterations and with $p = 0.1 * i$ for $1 \leq i \leq 9$. As shown in Table 1, the best p with and without replacement was 0.6. We use this p in our experiments on N4-N7.

We also tuned the GRASP algorithms on N1-N3 and tested them on N4-N7. We ran GRASP-k with and without replacement for 10,000 iterations and with $k = 2, \ldots, 10$. We ran GRASP-α with $\alpha = 0.1 * i$ for $1 \leq i \leq 9$.

3.1.2 Results

Table 2 shows the results for applying several BubbleSearch variations and random orderings to BL on the 20 instances in the N4-N7 collections. The first row shows the results from BLD, which requires an

iteration for each of the four base orderings.

While using even 20,000 random orderings does not perform as well as BLD, just 100 iterations of each variation of BubbleSearch significantly improved upon BLD. In particular, randomized BubbleSearch without replacement scored 28% closer, on average, to optimal than BLD after 100 iterations. These improvements continue, but taper off over time. BubbleSearch performs better than the GRASP variants over almost all iterations, although the improvement is small.

For both randomized BubbleSearch and GRASP-k, replacement offers minor gains. However, this was not the case for the training data N1-N3 above, in which randomized BubbleSearch performed better without replacement. More experimentation would be required to determine if replacement is truly valuable for this problem, although we believe these results are encouraging.

The BubbleSearch results are the best we know of for the 2D Strip Packing problem, with the exception of the results provided by our interactive system [15]. This system allows human users to guide the BubbleSearch algorithm and modify solutions directly.

3.2 Jobshop Scheduling

The *Jobshop* application is a widely-studied task scheduling problem [2]. In the variation we consider, a problem consists of n jobs and m machines. Each job is composed of m operations which must be performed in a specified order. The ordered list of operations for each job is called an itinerary. Each operation must be performed by a particular machine, and has a fixed duration. In our variation, every job has exactly one operation that must be performed on each machine. Each machine can process only one operation at a time.

A solution is a jobshop problem is a jobshop schedule that specifies a roster for each machine, which indicates the order in which the operations will be performed on that machine. Given a solution, each operation starts as soon as all predecessors on its machine's roster and predecessors on its job's itinerary have completed. The goal is to find a schedule which minimizes the time that the last job finishes, called the makespan.

3.2.1 Applying BubbleSearch

Several priority algorithms exist for Jobshop scheduling. These are often used to quickly find reasonably-good solutions, to find near-optimal solutions for easy problems, or to find a starting solution for a more sophisticated search algorithm such as tabu search.

A wide variety of ordering rules have been proposed (e.g., [4]) for jobshop scheduling. After a small amount of experimentation, we focused on one of the more popular which orders operations by the Most Work Remaining (MWKR). The work remaining for an operation o is the sum of the durations of the operations subsequent to o on its job. MWKR is a fixed ordering function.

We experimentally compared two placement functions. We describe the comparison in some detail, here, in order to demonstrate how one can evaluate the tradeoffs between efficiency and effectiveness within BubbleSearch. (That is, we are not claiming to make an novel contribution for jobshop scheduling.) The *nonDelay* placement function simply schedules the given operation o at the earliest time possible after the last currently-scheduled operation on o's machine.[3] The second placement function *bestPlace*, considers inserting o at every location in o's machine's current roster, and chooses the location that produces the partial schedule with minimum makespan (after repacking to eliminate any unnecessary delays in the schedule). The bestPlace function is likely to produce a better schedule, but requires more computation time per placement.

[3] However, if scheduling o after the last scheduled operation would introduce a cycle, we introduce it earlier.

Note that the bestPlace function can change the time at which already placed operations are scheduled. However, once an operation is placed its relative position to (i.e., before or after) all the other placed operations on its machine never changes. Therefore, to define bestPlace as a placement function, it should return a relative value and not an absolute time for each operation. After all operations are placed, their times can be computed from these values.

We compared the two placement functions using randomized BubbleSearch, without replacement, and $p = 0.5$. We ran the algorithm with each placement function for 20 minutes on the four instances of size 20×20 named yn1-yn4 [17] that are available at the OR-Library (http://www.ms.ic.ac.uk/info.html). Table entries present the average makespan found for the relevant problems. For these instances and algorithm settings, BubbleSearch performed an average of 17,343.3 iterations with bestPlace placement and 802,516.8 iterations with nonDelay in 20 minutes. However, the average makespan of bestPlace was 1141.50 while nonDelay was 1253.75. Based on this experience, we use bestPlace for the remainder of our experiments.

This example demonstrates one strength of the BubbleSearch framework: it allows experimentation with multiple ordering and placement functions.

We then tuned the input probability for randomized BubbleSearch, as above, by running it on the yn1-yn4 problems for 10,000 iterations with $p = 0.1 * i$ for $1 \leq i \leq 9$. As shown in Table 3, the best p without replacement was 0.3 and with replacement was 0.7. We used the same test data and number of iterations to tune the GRASP algorithms. We ran GRASP-k with and without replacement with $k = 2, \ldots, 10$. We ran GRASP-α with $\alpha = 0.1 * i$ for $1 \leq i \leq 9$. Based on these results, it appeared that even smaller values of α would lead to better performance, so we also tested GRASP-α with $\alpha = 0.012, 0.025$, and 0.05 as well. We found that $\alpha = 0.025$ offered the best performance.

3.2.2 Results

We evaluated BubbleSearch on the 20 problems called la21-la40, also available in the OR-Library. These problems have sizes 15×10, 20×10, 30×10, or 15×15. We computed a conservative lower bound for each problem by taking the maximum total duration of operations scheduled for any single job or machine. The average lower bound was 1227.9 for these problems.

The results in Table 4 demonstrate the value of BubbleSearch. Even using 20,000 random orderings does not perform as well as the priority algorithm. On the other hand, even 100 iterations of randomized BubbleSearch significantly improved upon it. In particular, randomized BubbleSearch without replacement scored 22.3% closer, on average, to our conservative lower bound than the priority algorithm, after 100 iterations.

For this problem, we find that BubbleSearch without replacement performs essentially the same as the best GRASP variation without replacement. Our replacement mechanism consistently improves both BubbleSearch and GRASP-k with GRASP-k having a slight advantage.

For this application, sophisticated algorithms such as tabu search generally produce better solutions than BubbleSearch has found [11]. Often such algorithms use priority algorithms to generate an initial solution to search from. We expect that BubbleSearch could provide value by improving that initial solution. More generally, a complete GRASP for the problem could use BubbleSearch or GRASP-k with replacement in the construction phase and a tabu search for the local search phase.

3.3 Edge Crossing

Edge-crossing minimization is a graph layout problem [8]. A problem consists of m levels, each with n nodes, and edges connecting nodes on adjacent levels. The goal is to rearrange nodes within their level to minimize the number of intersections between edges.

	BubbleSearch p value								
replacement	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
without	1159.00	1118.50	1114.25	1121.75	1118.50	1121.75	1129.50	1138.00	1141.25
with	1176.25	1129.50	1132.00	1104.00	1103.75	1101.50	1093.00	1102.00	1115.50

Table 3: Tuning the input probability for randomized BubbleSearch on Jobshop

iterations	random	BubbleSearch		GRASP-k		GRASP-α
		w/o replace $p = 0.3$	w/ replace $p = 0.7$	w/o replace $k = 8$	w/ replace $k = 2$	$\alpha = 0.025$
1 (base)	N/A	1672.60	1672.60	1672.60	1672.60	1672.6
100	2220.80	1573.45	1567.95	1580.20	1562.80	1590.70
500	2147.85	1554.65	1533.60	1559.90	1529.25	1580.90
1000	2104.15	1547.15	1523.65	1546.05	1522.60	1571.75
5000	2028.75	1528.40	1502.00	1531.45	1500.50	1568.75
10000	2007.20	1523.00	1497.35	1523.70	1489.10	1566.10
15000	1984.45	1520.45	1494.55	1521.25	1486.35	1564.9
20000	1981.55	1518.30	1492.25	1517.75	1485.70	1562.35

Table 4: BubbleSearch applied to Jobshop problems

3.3.1 Applying BubbleSearch

We used a placement function similar to bestPlace above, which positions each node in the best location (in its assigned level) to minimize intersections with the previously placed nodes. We considered two factors when developing the ordering function: the number of edges per node, and the closeness of the node to the middle level. After a modest amount of experimentation, we determined that the best combination was to order nodes by decreasing degree, breaking ties by closeness to middle.

We evaluated BubbleSearch on ten 12×8 graphs with 110 edges that are publicly available[4]. To tune the p parameter for randomized BubbleSearch, we randomly generated 10 other, similar 12×8 graphs. As above, we ran BubbleSearch 10,000 iterations with $p = 0.1 * i$ for $1 \leq i \leq 9$. As shown in Table 5, the best p without replacement was 0.2 and with replacement was 0.3. Again, we used the same test data and number of iterations to tune the GRASP algorithms, trying values $k = 2, \ldots, 10$ for GRASP-k with and without replacement and values $\alpha = 0.1 * i$ for $1 \leq i \leq 9$ for GRASP-α.

We also designed an adaptive priority function for this problem. After each node is placed, the remaining nodes are reordered in decreasing order of degree, but edges count differently if they are connected to an already-placed node. We tested variations in which these edges count twice as much and half as much. We ran 10,000 iterations of randomized BubbleSearch with the adaptive priority function on our random graphs with $p = 0.5$. Counting the "half-placed" edges twice as much produced the best results, with an average of 135.5 intersections, while counting them equally yielded an average of 147.7 intersections, and counting them half as much produced an average of 166.1 intersections. While these results show that the adaptivity helps, we choose to emphasize the comparison among BubbleSearch and the GRASP variants without adaptivity, for consistency with our previous experiments.

3.3.2 Results

The results in Table 6 show the average number of intersections of the algorithms over the 10 problems in our benchmark. (Again, all algorithms used the fixed priority function.) BubbleSearch without replacement has performance similar to but slightly better than the best GRASP variant without

[4] At http://unix.csis.ul.ie/~grafdath/TR-testgraphs.tar.Z

	BubbleSearch p value								
replacement	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
without	144.300	140.000	144.600	145.400	146.000	149.600	157.600	157.900	171.300
with	153.800	136.100	124.900	127.700	127.000	128.600	136.000	146.800	152.600

Table 5: Tuning the input probability for randomized BubbleSearch on Crossing

iter-ations	random	BubbleSearch		GRASP-k		GRASP-α
		w/o replace $p = 0.2$	w/ replace $p = 0.3$	w/o replace $k = 10$	w/ replace $k = 4$	$\alpha = 0.3$
1 (base)	N/A	754.90	754.90	754.90	754.90	754.90
100	868.40	414.30	378.00	408.20	378.30	429.10
500	816.30	353.40	266.40	353.40	238.90	379.30
1000	774.60	336.00	218.90	336.40	185.60	360.00
5000	699.30	263.00	139.30	277.40	147.30	325.40
10000	682.10	252.60	121.70	259.40	142.80	312.30
15000	661.30	232.40	114.40	243.20	137.60	295.60
20000	653.80	232.40	108.30	235.50	129.10	292.60

Table 6: BubbleSearch applied to Crossing problems

replacement. For this problem, replacement provided dramatic benefits for both BubbleSearch and GRASP-k. BubbleSearch with replacement is the best performer (after sufficiently many iterations).

The optimal answers to these problems are known to contain an average of 33.13 intersections [13]. Our results therefore suggest that our initial priority algorithm was rather weak; for example, even 2,000 iterations with random orderings also outperforms the original priority algorithm. This also explains why replacement offers significant advantages; being able to change the order from the priority function should be helpful if the initial ordering is not very good. Notice that for this problem the nodes were ordered by decreasing degree, which creates a lot of ties, along with an additional tie-breaking procedure. We suspect that replacement will be especially useful for many similar problems where the natural ordering is based on an integer-valued function (such as the degree) where there can be many ties that must be broken in an ad hoc fashion. While none of the algorithms came very close to the optimal answers, a local search could again be implemented in conjunction with them. Alternatively, we suspect that better ordering and placement functions would yield better results.

We also ran randomized BubbleSearch with the adaptive algorithm on this problem, and it scored an average of 113.0 intersections on these benchmarks after 10,000 iterations which is slightly better than the best non-adaptive algorithm, but still far from the optimal.

4 Conclusions and Future Work

We have described a black-box approach for more fully exploiting the implicit knowledge embedded in priority functions. BubbleSearch can extend any priority algorithm that fits our formulation into an anytime algorithm. For all the problems we evaluated, BubbleSearch dramatically improved upon the original solution with very few iterations and continued to improve steadily given more time. BubbleSearch compares favorably with similar GRASP-based variations, and the simple addition of replacing the base ordering when a better ordering is found appears to improve performance, in some cases dramatically.

There is a great deal of potential future work. On the practical side, we expect that BubbleSearch could improve several other natural priority algorithms and existing GRASP algorithms. Determining

what types of problems BubbleSearch and BubbleSearch with replacement are most effective on would be useful.

On the theoretical side, it would be worthwhile to understand if using additional perturbed orderings in a manner similar to BubbleSearch can yield improved competitive ratios for priority algorithms. A specific question is whether there are any problems for which there is a natural and effective fixed priority algorithm with competitive ratio c_1, but for which the competitive ratio can be provably reduced to $c_2 < c_1$ by trying all orderings within Kendall-tau distance 1 (or some other fixed constant) from the base ordering.

References

[1] S. Angelopoulos and A. Borodin. On the Power of Priority Algorithms for Facility Location and Set Cover. In *Proceedings of APPROX 2002*, pp. 26-39, 2002. Available as Lecture Notes in Computer Science 2462, Springer.

[2] E. Aarts, P. v. Laarhoven, J. Lenstra, and N. Ulder. A computational study of local search algorithms for job-shop scheduling. *ORSA Journal on Computing*, 6(2):118–125, 1994.

[3] B. S. Baker, E. G. Coffman, Jr., and R. L. Rivest. Orthogonal packings in two dimensions. *SIAM Journal on Computing*, 9:846-855, 1980.

[4] J.W. Barnes and J.B. Chambers. Solving the job shop scheduling problem using tabu search. *IIE Transactions*, 27:257-263, 1995.

[5] A. Borodin, M. N. Nielsen, and C. Rackoff. (Incremental) Priority Algorithms. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 752-761, 2002.

[6] J. L. Bresina. Heuristic-biased stochastic sampling. In *Proceedings of the 13th National Conference on Artificial Intelligence*, pp. 271-278, 1996.

[7] D. J. Brown. An improved BL lower bound. *Information Processing Letters*, 11:37-39, 1980.

[8] P. Eades and N. C. Wormald. Edge crossings in drawings of bipartite graphs. *Algorithmica* 11:379–403, 1994.

[9] J. P. Hart and A. W. Shogan. Semi-greedy heuristics: An empirical study. *Operations Research Letters*, 6:107-114, 1987.

[10] E. Hopper. Two-Dimensional Packing Utilising Evolutionary Algorithms and other Meta-Heuristic Methods. PhD Thesis, Cardiff University, UK. 2000.

[11] A. S. Jain and S. Meeran. Deterministic Job-Shop Scheduling: Past, Present and Future. European Journal of Operational Research, 113:390-434, 1999.

[12] G. Klau, N. Lesh, J. Marks, and M. Mitzenmacher. Human-Guided Tabu Search. In *Proceedings of the 18th National Conference on Artificial Intelligence*, pp. 41-47, 2002.

[13] A. Kuusik. *Integer Linear Programming Approaches to Hierarchical Graph Drawing.* Ph.D. Dissertation, Department of Computer Science and Information Systems, University of Limerick, Ireland, 2000.

[14] N. Lesh, J. Marks, A. McMahon, and M. Mitzenmacher. New Exhaustive, Heuristic, and Interactive Approaches to 2D Rectangular Strip Packing. MERL Technical Report TR2003-05.

[15] N. Lesh, J. Marks, A. McMahon, and M. Mitzenmacher. New Heuristic and Interactive Approaches to 2D Rectangular Strip Packing. To be presented at IJCAI Workshop on Stochastic Search Algorithms, 2003. Available as MERL Technical Report TR2003-18.

[16] M. G. C. Resende and C. C. Ribiero. Greedy randomized adaptive search procedures. In F. Glover and G. Kochenberger, editors, *Handbook of Metaheuristics*, pp. 219–249. Kluwer Academic Publishers, 2003.

[17] T. Yamada and R. Nakano. A genetic algorithm applicable to large-scale job-shop instances. In *Parallel instance solving from nature 2.* North-Holland, Amsterdam: R. Manner, B. Manderick(eds). pp. 281–290, 1992.