A Simple Yet Effective Heuristic Framework for Optimization Problems

Gaofeng Huang *and* **Andrew Lim**
Dept of Industrial Engineering & Engineering Management
Hong Kong University of Science & Technology
Clear Water Bay, Kowloon, Hong Kong
{gfhuang, iealim}@ust.hk

Abstract

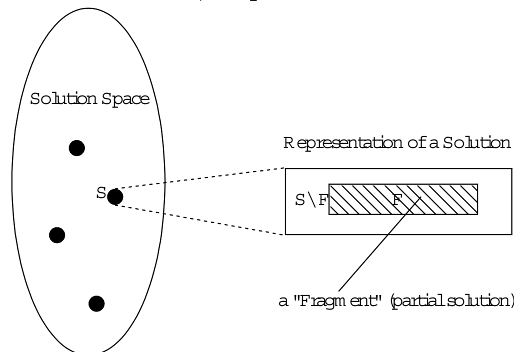
In this paper, we propose a simple yet effective heuristic framework called Fragmental Optimization (FO). In FO, there are two tightly coupled elements: Fragment Selection and Optimization. We formulate the FO technique and apply it to the **2-machine bicriteria flowshop scheduling problem** and the **3-Index Assignment Problem**. We conduct extensive experiments on standard benchmark instances for these problems. The experimental results show that our methods are superior to the previous best methods for the two problems. As the two problems are quite different, it suggests that our method is sufficiently general and can be adapted to solve other optimization problems effectively.

1 Overview

Search is one of the basic techniques in Artificial Intelligence. However, most real world optimization problems are still intractably hard because of their large search spaces. Many general heuristic search methods have thus been developed to find competitive solutions within a reasonable amount of time. These techniques include Simulated Annealing (SA), Tabu Search (TS), Genetic Algorithm (GA), Ant Colony Optimization (ACO), "Squeaky Wheal" Optimization (SWO), Greedy Randomized Adaptive Search Procedure (GRASP), etc.

The purpose of this paper is to present another effective heuristic framework termed *Fragmental Optimization* (FO). In its simplest form, FO is an iterative improvement algorithm, utilizing the basic principle of "easy things first". Since it is often computationally infeasible to optimize the whole solution, FO tries to achieve the relatively easier goal of optimizing a portion or *fragment* of the entire problem iteratively.

Figure 1: Solution, Representation and Fragment



As shown in Figure 1, within the solution space, each solution can have a unique representation S. A *fragment*, F, is defined as a small portion of S, i.e. a partial solution. As a result, a solution S is divided into two parts: F and $S\backslash F$. If we leave $S\backslash F$ unchanged, we may be able to optimize F such that the overall objective function value is improved. This idea can be formulated as:

$$\text{optimize } F \qquad | \qquad (S\backslash F) \text{ is fixed} \tag{1}$$

Algorithm 1 illustrates the general framework of FO. When we apply this framework to a specific problem, two issues need to be considered: (1) how to select the fragment (*Fragment Selection*), and (2) how to optimize the fragment effectively (*Optimization*).

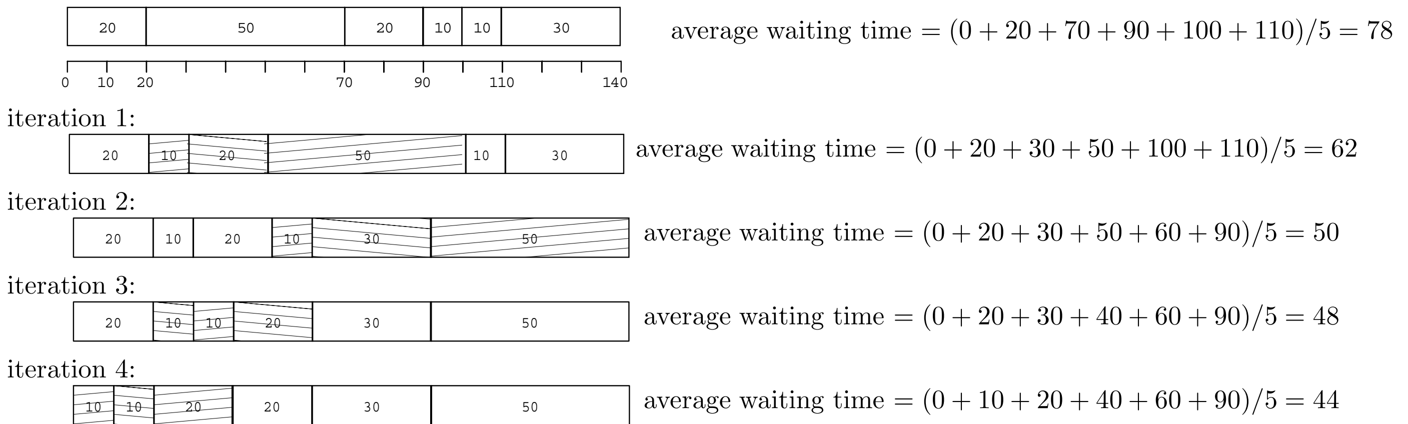
To demonstrate the core concept, consider a simple queuing problem as an example. Suppose n persons are waiting to be serviced, and the service processing time for person i is t_i. The objective

Algorithm 1 Fragmental Optimization framework

1: **while** *not* Termination Criteria **do**
2:     **select** a fragment from the whole solution space
3:     **optimize** this fragment subject to the other part of the solution being fixed
4: **end while**

is to queue these n persons (i.e. arrange the order of service) such that the average waiting time is minimized. A problem instance is shown in Figure 2. There are 6 persons with processing times $t = (20, 50, 20, 10, 10, 30)$. The initial solution takes 78 units waiting time. In each iteration, we randomly select 3 consecutive persons as a *fragment* and try to *optimize* the fragment. So in iteration 1, by optimizing the randomly selected fragment (shaded), the overall cost decreases to 62. After iteration 2, the cost is down to 50. After 4 iterations, FO reaches the optimal solution of cost 44.

Figure 2: Simple Example



Intuitively, this models how humans solve real-world problems: look at small fragments and try to solve the small fragment well in order to improve the results.

The rest of this paper is organized as follows. In the next 2 sections, we will illustrate the Fragment Optimization technique in greater detail by applying it successfully to 2 problems: the **2-machine bicriteria flowshop scheduling problem** and the **3-Index Assignment Problem**. Our experimental results are very encouraging as we managed to improve the results on benchmarks instances for these problems. In the last section, we present our conclusions.

2 2-machine bicriteria flowshop scheduling problem

The 2-machine flowshop scheduling problems $F2$ may be defined as follows. We have n jobs and 2 machines M_1 and M_2. Each job must be processed first on machine M_1, then on machine M_2. All the n jobs are available in the beginning. Once a job is started on a machine, it should not be interrupted("no preemption"). The processing time of job i on machine M_1 is a_i, and on machine M_2 is b_i. Let $C_{i,j}$ denote the completion time of job i on machine M_j, a job's completion time, C_i, is defined as the completion time of job i on machine M_2, that is, $C_i = C_{i,2}$.

Extensive research was done optimizing one single criterion[9]. But recently, more interest has been given to multiple criteria scheduling. The 2-machine bicriteria flowshop problem we discuss in this section, $F2||(\sum C_i/C_{max})$, is to find a feasible schedule that first minimizes maximum completion time(makespan) $C_{max} = \max\{C_i\}$ and then minimizes the total completion time $\sum C_i$.

Chen and Bulfin(1994) proved that $F2||(\sum C_i/C_{max})$ is $\mathcal{NP}$-hard in the strong sense[8]. Rajendran developed a branch-and-bound algorithm and two heuristics which can solve this problem with no more than 10 and 25 jobs respectively[7]. Neppalli et al.(1996) applied GA to solve this problem[6]. Gupta et al.(1999) developed a tabu search algorithm for this problem[5], and they also proposed 9 polynomial heuristic algorithms in [2]. Later a comparative study of several well-known local search heuristics was done by Gupta et al.[1]. Using a Lagrangian relaxation to develop the lower bound, T'Kindt et al.(2001) implemented an efficient branch-and-bound algorithm[4], which can solve this problem with

Figure 3: No waiting case: $t \geq a_i$

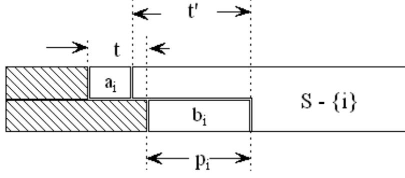


Figure 4: Waiting case: $t < a_i$

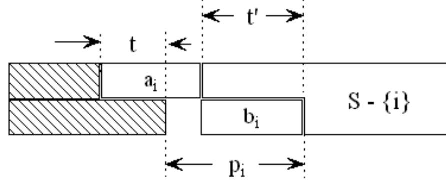
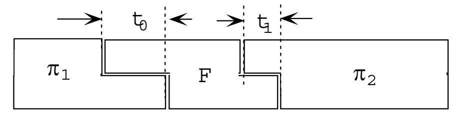


Figure 5: Fragment Selection



35 jobs. He also used Ant Colony Optimization(SACO) to solve this problem, which "yields better results than existing heuristics."[3].

In this section, we present our Fragmental Optimization (FO) heuristic for this problem, which combines dynamic programming and local search strategy. Experimental results show that our FO algorithm outperforms existing heuristics and provides solutions very close to the optimal.

2.1 Johnson's Algorithm

The problem $F2||C_{max}$ is solved by Johnson in $O(n \lg n)$ time[10].

Algorithm 2 JOHNSON'S ALGORITHM

1: $X \leftarrow \{i | a_i \leq b_i\}$;
2: $Y \leftarrow \{j | a_j > b_j\}$;
3: sort X as a partial sequence by non-decreasing a_i;
4: sort Y as a partial sequence by non-increasing b_j;
5: Johnson's permutation $J \leftarrow$ append sequence Y after X.

The completion time of job $J(k)$ and makespan can be computed as:

$$C_{J(k),1} = \sum_{i=1}^{k} a_{J(i)}$$

$$C_{J(k)} = \begin{cases} 0, & k = 0 \\ \max\{C_{J(k-1)}, C_{J(k),1}\} + b_{J(k)}, & k = 1, 2, ..., n \end{cases} \tag{2}$$

$$C^*_{max} = C_{max}(J) = C_{J(n)}$$

Although the job sequence that comes from Johnson's algorithm can provide a feasible solution to problem $F2||(\sum C_i / C_{max})$, for most of the instances the solutions provided are "bad"[1]. However, Johnson's algorithm is useful in checking whether a partial scheduling sequence is feasible. Suppose we have a partial scheduling sequence $\pi' = \{\pi(1), \pi(2), ..., \pi(m)\} (m < n)$, let $\overline{\pi'} = \{1, 2, ..., n\} - \pi'$ be the set of unallocated jobs, we can apply Johnson's algorithm on $\overline{\pi'}$ and get $J_{\overline{\pi'}}$. If the makespan $C_{max}(\pi' J_{\overline{\pi'}}) > C_{max}(J)$, there should be no complete feasible schedule with π' as its partial schedule, which means that the partial scheduling sequence π' is already not feasible[2].

2.2 Dynamic Programming

For the 2-machine flowshop problems with any regular criterion, the total set of permutation schedules contains at least one optimal solution[9]. This means that the job sequence on machine M_1 and M_2 is exactly the same. Now we allocate jobs one by one. Let S be the set of unallocated jobs, while t denotes current free time span of machine $M1$. As the two cases in Figure (3) & (4) show, the optimal solution of subproblem $(C_{max}, \sum C_i) = G(t, S)$ can be determined recursively by:

$$G(t, S) = \min_{i \in S} [(p_i, p_i \times |S|) + G(t', S - \{i\})] \tag{3}$$

$$= \min_{i \in S} \begin{cases} (b_i, \quad b_i \times |S|) + G(t - a_i + b_i, \quad S - \{i\}) & t \geq a_i \\ (a_i + b_i - t, \quad (a_i + b_i - t) \times |S|) + G(b_i, \quad S - \{i\}) & t < a_i \end{cases} \tag{4}$$

while the initial condition is $G(t, \emptyset) = (0, 0)$. Finally, by tracking G(0, {1, 2, ..., n}), we can find the optimal solution for the whole problem. However, this algorithm takes $O(T \times 2^n \times n)$ time and $O(T \times 2^n)$ space, which is exponential and not practical.

2.3 Fragmental Optimization

Since the time and space complexity grows exponentially, the basic idea of our FO heuristic is to apply Dynamic Programming not in the whole solution(length n) but in a small solution fragment such that the time and space needed are acceptable. In this problem, *fragment* is defined as a length L successive sub-sequence(our experiment sets $L = 10$), while the *optimization* is done by DP.

As illustrated in Figure 5, our FO heuristic starts with a feasible solution, which means that the makespan is already minimized. The FO algorithm then maintains a sliding-window ($windowL$, $windowR$) of width L and only considers the jobs inside the window. In each iteration, it selects the length L successive sub-sequence and tries to improve the *fragment* F by applying DP while the minimized makespan is maintained (checking by Johnson's algorithm). Therefore, in one iteration of FO, L jobs will be re-ordered, which will take $O(T \times 2^L \times L)$ time. After each iteration, FO moves the sliding-window forward. This process repeats until no further improvement can be obtained.

2.4 Experimental Results

We conduct extensive experiments to prove the efficiency of our FO heuristic. For each fixed n, the number of jobs involved, 100 instances are generated randomly. The processing time a_i and b_i are both integers and evenly distributed in (0, 100). All the algorithms are implemented in C/C++ and run on a Pentium III 800Mhz PC with 128M memory.

2.4.1 Problems with small size

We experiment with the size $n = 10$ to 17, for 800 instances in total. For each instance, we first use the Branch-and-Bound algorithm to find the optimal solution, and then we apply FO heuristic to that instance to see whether the solution FO provides is optimal or not.

Table 1 lists the result statistics for all these 800 instances. *#optimal* shows for how many instances FO heuristic can provide the optimal solution, while T is the running time for one instance. (The performance of SACO heuristic[3] is also shown in the table for comparison). As we can see, Branch-and-Bound is extermely slow, while FO heuristic is rather effective because it can find the optimal solution for most of the instances, especially when n is small.

Table 1: Statistics for small size problem (800 instances)

N	#instance	Branch-and-Bound			FO				SACO			
		Tmin	Tavg	Tmax	Tmin	Tavg	Tmax	#optimal	Tmin	Tavg	Tmax	#optimal
10	100	0.00	0.04	0.23	0.20	0.24	0.30	100	0.00	0.10	1.00	100
11	100	0.00	0.07	0.81	0.22	0.30	0.41	100	0.00	0.20	1.00	100
12	100	0.00	0.28	5.20	0.27	0.37	0.44	100	0.00	0.21	1.00	100
13	100	0.01	1.83	24.23	0.39	0.47	0.71	100	0.00	0.24	1.00	98
14	100	0.01	9.66	184.44	0.45	0.54	0.63	100	0.00	0.30	1.00	98
15	100	0.00	66.35	960.46	0.48	0.73	2.72	100	0.00	0.24	1.00	90
16	100	0.01	94.05	1286.75	0.61	0.86	5.25	100	0.00	0.45	1.00	84
17	100	0.01	258.86	6931.5	0.59	0.97	7.24	98	0.00	0.22	1.00	80

2.4.2 Problems with large size

We experiment the size $n = 20, 30, 50, 75, 100, 150, 200$, for a total of 700 instances. We compare FO heuristic with other remarkable heuristics: INSERT[2], UB2(EXCHANGE)[4], especially SACO[1], which was proved to "yield better results than other existing heuristics" for this problem[3], to see which algorithm offers the best solution for each instance.

Statistics are shown in Table 2. *#best* shows the number of instances when a particular algorithm provides the best solution. δ denotes the deviation of a particular algorithm's solution from the best solution. T_{avg} shows the average time for solving one instance. As you can see, except for two instances, our FO heuristic always offers the best solution among the 4 algorithms. Therefore, δ_{avg} of the other algorithm actually means the average deviation between that algorithm and our FO heuristic. Looking at δ_{avg} of SACO, we can see that δ_{avg} tends to increase with the growing of problem size n. Consequently, with the growing of n, our FO tends to provide a better solution than SACO. What is more important, with the growing of problem size n, the increase of the average time of FO is less than the increase of

[1]The authors would like to show special thanks to Vincent T'Kindt for providing the executable SACO program, which really helps to compare the efficiency of the algorithms.

Table 2: Statistics for large size problem(700 instances)

N	#instance	INSERT			UB2			SACO				FO			
		δavg	δmax	#best	δavg	δmax	#best	δavg	δmax	Tavg	#best	δavg	δmax	Tavg	#best
20	100	0.92%	5.17%	6	0.58%	3.61%	17	0.05%	0.54%	0.41	64	0.00%	0.07%	1.19	99
30	100	0.97%	3.25%	0	0.65%	2.90%	2	0.20%	0.98%	1.04	14	0.00%	0.01%	2.02	99
50	100	0.96%	2.43%	0	0.58%	2.43%	0	0.36%	0.99%	4.29	0	0.00%	0.00%	4.63	100
75	100	0.85%	1.94%	0	0.54%	1.93%	0	0.44%	0.93%	13.50	0	0.00%	0.00%	9.16	100
100	100	0.80%	1.67%	0	0.53%	1.67%	0	0.46%	0.97%	30.66	0	0.00%	0.00%	14.69	100
150	100	0.70%	2.11%	0	0.50%	2.11%	0	0.49%	1.21%	89.63	0	0.00%	0.00%	30.01	100
200	100	0.59%	1.31%	0	0.43%	1.31%	0	0.50%	1.10%	201.13	0	0.00%	0.00%	53.67	100

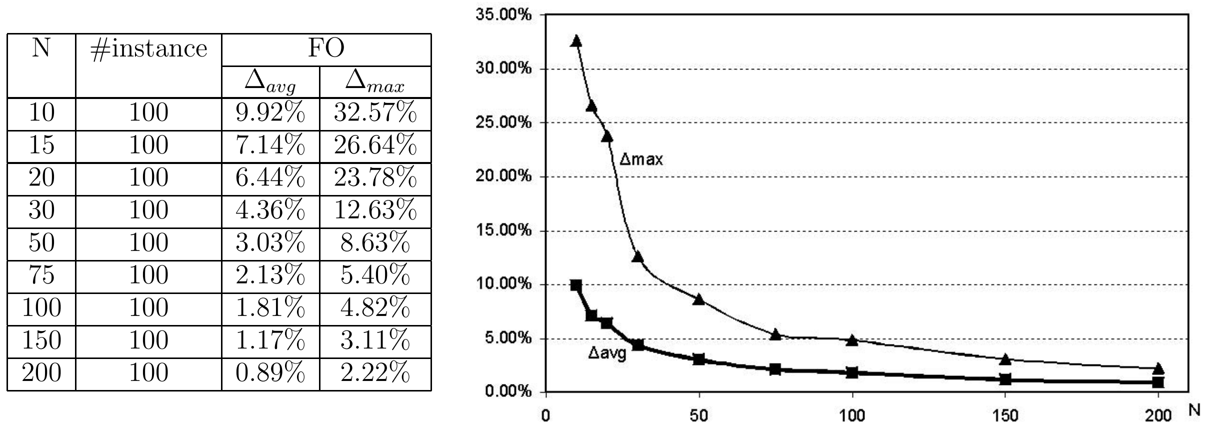
the average time of SACO. For the instances $n = 200$, our FO can provide solutions in approximately 1 minute, while SACO is 4 times slower.

Hence, FO heuristic outperforms SACO both in time and in the quality of solution.

2.4.3 Compare with LB_v

We also compare FO heuristic with the lower bound LB_v(Lower Bound by Lagrangian Relaxation)[4] to see the absolute quality of our solutions. We define the absolute deviation Δ of a solution π as $\Delta(\pi) = \frac{\sum C_i(\pi) - LB_v}{LB_v}$, which means the deviation between that solution and optimal solution is at most $\Delta(\pi)$. Statistics are shown in Figure 6.

Figure 6: Comparison between FO and LB_v



N	#instance	FO	
		Δ_{avg}	Δ_{max}
10	100	9.92%	32.57%
15	100	7.14%	26.64%
20	100	6.44%	23.78%
30	100	4.36%	12.63%
50	100	3.03%	8.63%
75	100	2.13%	5.40%
100	100	1.81%	4.82%
150	100	1.17%	3.11%
200	100	0.89%	2.22%

LB_v is a coarse lower bound, especially when n is small. As we can see, when $n = 10, 15$, the solutions that FO provides are already optimal, but Δ_{avg} is still about 10%.

However, when n becomes larger our FO algorithm can provide solutions that are very close to the lower bound, which means that the deviation between FO solution and optimal solution is small. Moreover, Δ_{avg} decreases with the growing of problem size n. This means that the larger n is, the closer the solution is to the optimal solution that our FO can provide. For the instances $n = 200$, our FO can almost offer solutions that deviate with only 1% from the optimal.

3 3-Index Assignment Problem

The Three-Index Assignment Problem (AP3), also known as the 3-Dimensional Assignment Problem, can be formulated as:

Instance	:	a cost matrix $C = \{c_{i,j,k}\}_{n \times n \times n}$
Solution	:	two permutations (p, q), $\quad p, q \in \pi_N$
Objective	:	to minimize $C(p, q) = \sum_{i=1}^{n} c_{i,p(i),q(i)}$

where π_N denotes the set of all permutations on the set of integers $N = \{1, 2, ..., n\}$.

AP3 is $\mathcal{NP}$-hard since the 3-D Matching Problem, which is one of the basic $\mathcal{NP}$-hard problems, is one of the special case of AP3. Both exact and heuristic algorithms have been proposed to solve AP3[11, 12, 13, 14]. Among these, Balas and Saltzman(1991)[14] proposed the MAX-REGRET and

VARIABLE DEPTH INTERCHANGE heuristics. Crama and Spieksma(1992)[13] studied a special case of AP3 by restricting the cost of edges in any triangle to obey the rule of triangle inequalities. Burkard et al(1996)[12] focused on AP3 with decomposable cost coefficients. However, even for these two special cases, AP3 is still $\mathcal{NP}$-hard. Recently, Aiex et al.(2003)[11] applied GRASP with Path Relinking for AP3 and obtained better results than all other existing heuristics.

In this section, we first design a Fragmental Optimization algorithm for AP3. And then we hybridize FO with Genetic Algorithm(GA) to demostrate the bybridization between FO and classical hueristic methods. Experiments indicate that GA benefits from this hybridization. We test our algorithm on three classes of standard benchmarks and report the computational results.

3.1 Reduce 3D to 2D

It is quite obvious that AP3 is a straightforward extension of the classical two-dimensional assignment problem (AP2) defined below:

Instance	:	a matrix $D = \{d_{i,j}\}_{n\times n}$(bipartite graph)
Solution	:	$q = (q_1, q_2, ..., q_n),\quad q \in \pi_N$
Objective	:	to minimize $D(q) = \sum_{i=1}^{n} d_{i,q(i)}$

Although AP3 is $\mathcal{NP}$-hard, it's well-known that AP2 can be solved by an efficient implementation of Hungarian algorithm in $O(n^3)$ time[15]. Here we consider how to make use of this results. Given an initial solution (p, q) for AP3,

$$\text{let } d_{i,j} = c_{i,p(i),j}, \forall i, j \in 1, 2...n \tag{5}$$

$$\text{we get } \min_{p,q\in\pi_N} \sum_{i=1}^{n} c_{i,p(i),q(i)} = \min_{q\in\pi_N} \sum_{i=1}^{n} d_{i,q(i)}$$

Consequently, if we fix p, the optimization of q becomes an AP2 problem, and vice versa.

Therefore, our idea is to optimize one permutation subject to the other permutation being fixed. To illustrate this, we use an example (instance bs_4_5.dat from Balas and Saltzman Dataset, see Section 3.4.1). As shown in Figure 7, a random initial assignment costs 177. Figure 8 shows the optimization of q by applying Hungarian Method. The objective function decreases from 177 to 72 (The dotted lines show the original assignment, while the new assignment is shown by bold lines).

Figure 7: random initial for bs_4_5.dat

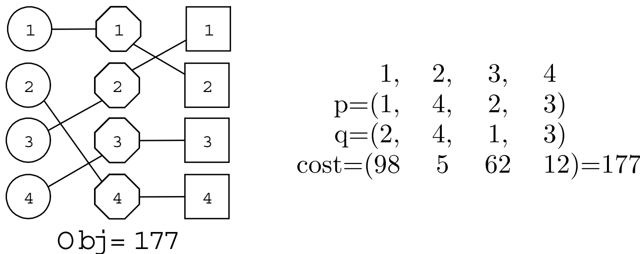


Figure 8: optimize permutation q("fragment")

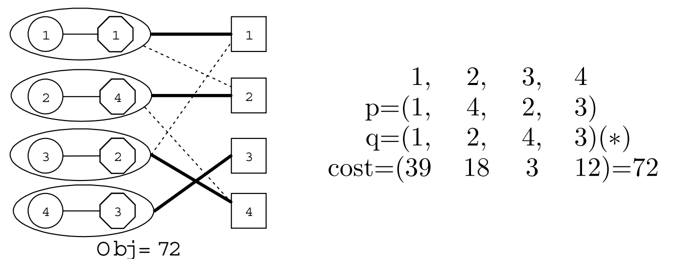


Figure 9: optimize permutation p

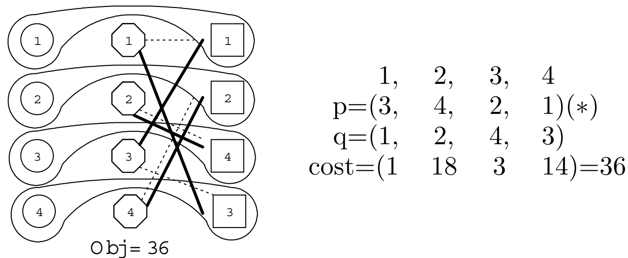
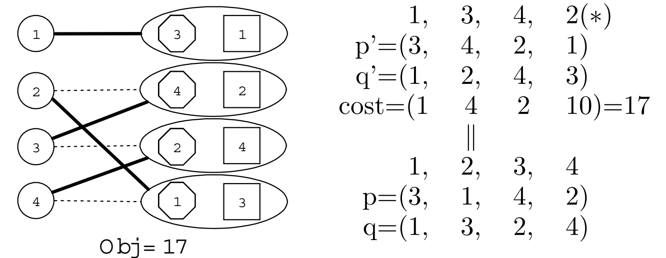


Figure 10: optimize the permutation index



3.2 Fragmental Optimization

As shown in Figure 8, we construct a bipartite graph based on Equation(5). Symmetrically, there are two other ways to construct such a bipartite graph:

$$\text{let } d_{i,j} = c_{i,j,q(i)}, \forall i, j \in 1, 2...n \tag{6}$$

$$\text{or let } d_{i,j} = c_{j,p(i),q(i)}, \forall i, j \in 1, 2...n \tag{7}$$

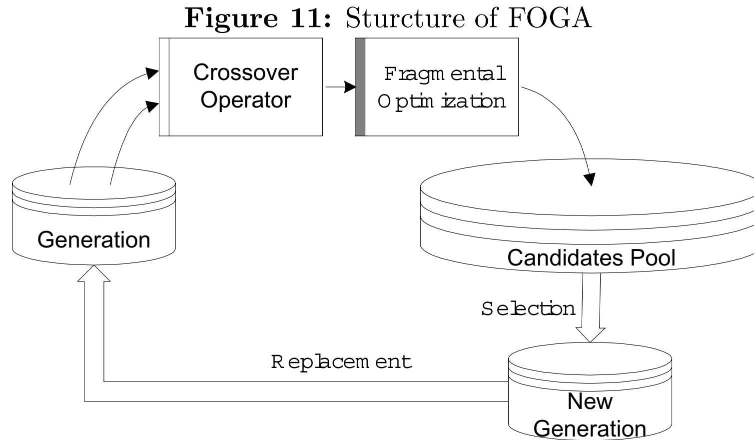
Figure 9 illustrates Equation(6), while Figure 10 corresponds to Equation(7).

Using the above, we define three ways to select a *fragment*, which actually correspond to the three parts of one solution to AP3, namely permutation p, permutation q and the permutation index. Our FO heuristic iteratively optimizes the *fragment* by using Hungrarian Method as the *optimization* method until no more improvement can be achieved.

3.3 Hybridized with Genetic Algorithm

As we all know, Genetic Algorithm (GA) has shown to be competitive technique for solving general combinatorial optimization problems. However, it is still possible to incorporate problem-specific knowledge into GA so that the results can be further improved. The hybridization between FO with Genetic Algorithm reflects this idea.

Figure 11: Sturcture of FOGA



A fairly standard GA structure is adopted in our implementation. The initial "generation" is randomly generated. The "Crossover Operator" randomly chooses two individuals (as parents) according to their fitness function values, and generates one new individual. This newly generated individual will "mutate" with a very small probability. All of these newly generated individuals are put into a candidates pool, whose size is normally twice the parent population. The algorithm then applies the "survival of the fittest" principle to the candidates pool and selects the top half of the individuals with good fitness value from the candidate pool will be selected to form the next "generation". After several generations, the GA terminates after satisfying some termination criteria.

In our algorithm, we did not implement mutation operators because it was not very useful. Others have reported similar experiences with mutation in GA[16, 17]. Instead, we replace Mutation Operator by our Fragmental Optimization module. When a new individual is generated from Crossover Operator, we apply FO to improve its quality before putting it into the candidates pool. By this method, we hybridize FO with Genetic Algorithm. Figure 11 illustrates the structure of our hybridization(FOGA).

Preliminary experiments were conducted to tune our algorithm because the parameter setting can influence the performance of a genetic algorithm substantially. Due to the space limitation, we do not report the details here. We set the population size to be 100.

Figure 12 shows the comparison between pure GA, a multi-round FO, and FOGA over instance bs_26_1.dat from Balas and Saltzman Dataset(see Section 3.4.1). As you can see,

1. Pure GA is rather bad in performance. Even if more time is given.

2. multiFO is a multi-round FO algorithm. In each round, it starts with a random initial permutation and uses FO to improve the solution. As you can see, multiFO has the ability to find relatively good solutions in a short time. This reflects the effectiveness of our FO algorithm. Unfortunately, even if much more time is given, this algorithm cannot improve the best solution. For the instance bs_26_1.dat, no better solutions can be found after 1 second.

3. FOGA shows the power of hybridization of Genetic Algorithm and Fragmental Optimization. It is capable of finding very competitive solutions.

Figure 12: Compasion between GA, multiFO and FOGA

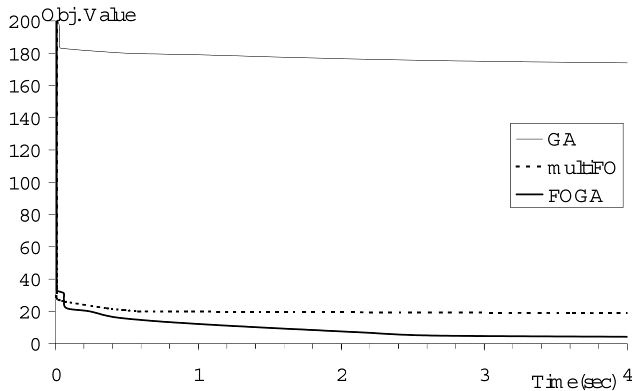


Table 3: Balas and Saltzman Dataset(12*5 instaces)

n	Optimal	B-S	GRASP with Path Relinking			multiFO		FOGA	
	Avg.Obj Value	Avg.Obj Value	Avg.Obj Value	Average CPU Time		Avg.Obj Value	Avg.Time	Avg.Obj Value	Avg.Time
				R10000	PIII800		PIII800		PIII800
4	42.2	43.2	-	-	-	42.2	0.00 s	42.2	0.00 s
6	40.2	45.4	-	-	-	40.2	0.01 s	40.2	0.01 s
8	23.8	33.6	-	-	-	33.6	0.01 s	23.8	0.03 s
10	19	40.8	-	-	-	22.6	0.01 s	19	0.37 s
12	15.6	24	15.6	74.79 s	> 18.7 s	26.2	0.02 s	15.6	0.87 s
14	10	22.4	10	106.55 s	> 26.64 s	26.4	0.02 s	10	1.73 s
16	10	25	10.2	143.89 s	> 35.97 s	26.0	0.03 s	10	1.89 s
18	6.4	17.6	7.4	190.88 s	> 47.72 s	24.6	0.03 s	7.2	2.95 s
20	4.8	27.4	6.4	246.70 s	> 61.68 s	26.8	0.04 s	5.2	4.01 s
22	4	18.8	7.8	309.64 s	> 77.41 s	26.4	0.05 s	5.6	4.54 s
24	1.8	14	7.4	382.45 s	> 95.61 s	23.2	0.06 s	3.2	5.66 s
26	1.3	15.7	8.4	465.20 s	> 116.3 s	23.2	0.07 s	3.6	10.78 s

Hence, it's evident that our hybridization of GA and FO is successful. Guided by the GA, it is possible for FO to improve the quality of solution consistently; and with the help of FO, Genetic Algorithm becomes competitive in finding good solutions.

3.4 Computational Results

In this section, we demonstrate the effectiveness of our hybrid genetic algorithm(FOGA) by testing our heuristic on three benchmark datasets. All the codes are implemented by C/C++ under a Pentium III 800MHz PC with 128M memory. For each instance, our FOGA is run only once.

However, the computing machine used in Aiex's paper[11] is SGI Challenge R10000. Therefore, in order to compare the CPU time, a scaling scheme is used according to SPEC[2].

3.4.1 Balas and Saltzman Dataset

This dataset is generated by Balas and Saltzman[14]. It includes 60 test instances with the problem size $n = 4, 6, 8, ..., 22, 24, 26$. For each n, 5 instances are randomly generated with the integer cost coefficients $c_{i,j,k}$ uniformly distributed in the interval $[0, 100]$.

Each row of Table 3 stores the average score of the 5 instances with the same size. The Column "Optimal" shows the optimal solution reported by Balas and Saltzman, while column "B-S" is the result of their VARIABLE DEPTH INTERCHANGE heuristic. Column "GRASP with Path Relinking" is the result reported in [11]. Column "multiFO" is the result of our multi-round FO algorithm, which is terminated after 100 rounds. Finally, Column "FOGA" shows the result of our hybrid genetic algorithm. The best results among these algorithms are highlighted in the table.

As you can see in Table 3, it is evident that our FOGA can provide much better solutions than GRASP and B-S. Furthermore FOGA is about 10 times faster than GRASP in these instances.

3.4.2 Crama and Spieksma Dataset

Crama and Spieksma generated this dataset by restricting coefficients $c_{i,j,k} = d_{i,j} + d_{i,k} + d_{j,k}$[13]. There are 3 types of intances in this dataset. For each type, 6 instances of size $n = 33$ and 3 instances of size $n = 66$ are generated.

Table 4, Table 5 and Table 6 report the experimental results. In these tables, column "C-S" reports the result of Crama and Spieksma's heuristic. As highlighted in these tables, for all these 18 instances, our FOGA can always provide the best solutions over all heuristics.

It is surprising that for these instances, FOGA is about 1000 times faster than GRASP. GRASP needs several hours to get the solutions while FOGA only takes several seconds.

3.4.3 Brukard, Rudolf & Woeginger Dataset

Brukard et al[12] described this dataset with decomposable cost coefficients, which means that $c_{i,j,k} = \alpha_i \bullet \beta_j \bullet \gamma_k$. For each problem size $n = 4, 6, 8, ..., 16$, 100 test instances are provided. Table 7 is the result statistics of all these 700 instances, where each row is the average of the 100 instances with same size n; column "B-R-W" reports the result of Brukard et al's heuristic.

[2]SPEC(Standard Performance Evaluation Corporation, `http://www.specbench.org/osg/cpu2000/`) points out that PIII 800 is not more 4 times faster than SGI Challenge R10000

Table 4: Crama and Spieksma Dataset, Type I (6 instances)

CaseID	C-S	GRASP with Path Relinking			multiFO		FOGA	
	Avg.Obj Value	Avg.Obj Value	Average CPU Time R10000	Average CPU Time PIII800	Avg.Obj Value	Avg.Time PIII800	Avg.Obj Value	Avg.Time PIII800
3DA99N1	1618	1608	660.5 s	> 165.13 s	1608	0.03 s	1608	0.03 s
3DA99N2	1411	1401	680.5 s	> 170.13 s	1401	0.02 s	1401	0.11 s
3DA99N3	1609	1604	676.1 s	> 169.03 s	1604	0.03 s	1604	0.11 s
3DA198N1	2668	2664	15470.1 s	> 3867.5 s	2662	0.21 s	2662	0.55 s
3DA198N2	2469	2449	15010.9 s	> 3752.7 s	2449	0.21 s	2449	0.27 s
3DA198N3	2775	2759	15084.6 s	> 3771.2 s	2758	0.22 s	2758	0.58 s

Table 5: Crama and Spieksma Dataset, Type II (6 instances)

CaseID	C-S	GRASP with Path Relinking			multiFO		FOGA	
	Avg.Obj Value	Avg.Obj Value	Average CPU Time R10000	Average CPU Time PIII800	Avg.Obj Value	Avg.Time PIII800	Avg.Obj Value	Avg.Time PIII800
3DIJ99N1	4861	4797	766.06 s	> 191.52 s	4797	0.06 s	4797	0.11 s
3DIJ99N2	5142	5068	772.84 s	> 193.21 s	5068	0.13 s	5067	0.26 s
3DIJ99N3	4352	4287	762.19 s	> 190.55 s	4287	0.08 s	4287	0.26 s
3DI198N1	9780	9694	14629.1 s	> 3657.3 s	9687	0.51 s	9684	4.86 s
3DI198N2	9142	8954	14922.9 s	> 3730.7 s	8947	0.53 s	8944	3.35 s
3DI198N3	9888	9751	14391.7 s	> 3597.9 s	9747	0.53 s	9745	3.09 s

Table 6: Crama and Spieksma Dataset, Type III (6 instances)

CaseID	C-S	GRASP with Path Relinking			multiFO		FOGA	
	Avg.Obj Value	Avg.Obj Value	Average CPU Time R10000	Average CPU Time PIII800	Avg.Obj Value	Avg.Time PIII800	Avg.Obj Value	Avg.Time PIII800
3D1299N1	135	133	490.79 s	> 122.7 s	133	0.01 s	133	0.01 s
3D1299N2	137	131	471.21 s	> 117.8 s	132	0.01 s	131	0.03 s
3D1299N3	135	131	451.72 s	> 112.93 s	131	0.01 s	131	0.02 s
3D1198N1	293	286	5322.97 s	> 1330.7 s	287	0.05 s	286	0.15 s
3D1198N2	294	286	5126.86 s	> 1281.7 s	286	0.05 s	286	0.16 s
3D1198N3	293	282	5059.06 s	> 1264.8 s	283	0.05 s	282	0.23 s

Table 7: Brukard, Rudolf & Woeginger Dataset (700 instances)

n	B-R-W	GRASP with Path Relinking			multiFO		FOGA	
	Avg.Obj Value	Avg.Obj. Value	Average CPU Time R10000	Average CPU Time PIII800	Avg.Obj Value	Avg.Time PIII600	Avg.Obj. Value	Avg.Time PIII800
4	443.7	-	-	-	433.6	0.00 s	443.6	0.00 s
6	634.2	-	-	-	633.72	0.00 s	633.72	0.01 s
8	819.94	-	-	-	819.16	0.01 s	819.16	0.03 s
10	960.55	-	-	-	959.42	0.03 s	959.41	0.07 s
12	1188.02	1186.81	68.3 s	> 17.1 s	1186.83	0.04 s	1186.81	0.13 s
14	1469.27	1467.74	98.1 s	> 24.5 s	1467.76	0.07 s	1467.74	0.23 s
16	1476.99	1475.13	139.3 s	> 34.8 s	1475.15	0.10 s	1475.13	0.40 s

As highlighted in Table 7, for these test instances, our FOGA provides the same results with GRASP with about 100 times faster speed. However, this dataset is considered to be easy since even multiFO can also offer very competitive solutions, which is even faster.

4 Conclusions

The Fragmental Optimization (FO) technique proposed in the paper is a simple yet effective heuristic framework to solve combinatorial optimization problems. We applied the FO technique on the following well-known $\mathcal{NP}$-hard problems :

- **2-machine bicriteria flowshop problem** $F2||(\sum C_i/C_{max})$: A FO-based heuristic is proposed for this problem, which selects length L segment as *fragment* and uses dynamic programming as the *optimization* method. Experimental results show that our FO heuristic outperforms other existing heuristics both in time and in the quality of the solution. Moreover, by comparing FO to a coarse lower bound LB_v, we conclude that the solution FO provides is very close to the optimal. For small size problems, most of the time FO heuristic can provide the optimal solution; while for large size problems, the solution that FO provides is only about 1%–2% from the optimal.

- **3-index assignment problem AP3**: Unlike $F2||(\sum C_i/C_{max})$, here the solution to AP3 is represented as 2 permutations. In our FO heuristic, by defining each permutation as a *fragment*, we reduce the dimension from 3 to 2; thus the *optimization* can be performed by Hungarian Method in $O(n^3)$ time. We further hybridize FO with GA. Experiments indicate that this hybridization is successful. Computational results show that our hybrid genetic algorithm(FOGA) outperforms other existing heuristics. For those benchmark instances we have tested, FOGA is about 10 $\sim$ 1000 times faster than GRASP and can always offer the best solutions in several seconds.

Based on the above examples, the two elements of a Fragmental Optimization algorithm, *Fragment Selection* and *Optimization*, are tightly coupled. Generally speaking, any technique can be used in *Optimization*: such as Dynamic Programming, Network Flow, or even Branch-and-Bound. However, The key consideration is that this *Optimization* algorithm must be efficient for the selected *Fragment* since it is involved in every iteration.

In conclusion, we proposed the Fragmental Optimization heuristic in this paper. The success of applying FO to problems from different domains suggests that FO is a sufficiently general framework for optimization problems.

References

[1] Jatinder N.D. Gupta, Karsten Hennig, Frank Werner, *Local search heuristics for two-stage flow shop problems with secondary criterion*, Computers & Operations Research 29 (2002) 123–149.

[2] Jatinder N.D. Gupta, Venkata R. Neppalli, Frank Werner, *Minimizing total flow time in a two-machine flowshop problem with minimum makespan*, Intl Journal of Production Economics 69 (2001) 323–338.

[3] Vincent T'Kindt, Nicolas Monmarché, Fabrice Tercinet, Daniel Laügt, *An Ant Colony Optimization algorithm to solve a 2-machine bicriteria flowshop scheduling problem*, EJOR 142 (2002) 250–257.

[4] Vincent T'Kindt, Jatinder N.D. Gupta and Jean-Charles Billaut, *A Branch-and-Bound algorithm to Solve a Two-Machine Bicriteria Flowshop Scheduling Problem*, ORP3 (2001), Paris, September 26–29.

[5] J.N.D. Gupta, Nagarajan Palanimuthu, C.L. Chen, *Designing a tabu search algorithm for the two-stage flowshop problem with secondary criterion*, Production Planning and Control, (10), 1999, 251–265.

[6] Venkata R. Neppalli, Chuen-Lung Chen, Jatinder N.D. Gupta, *Genetic algorithm for the two-stage bicriteria flowshop problem*, European Journal of Operational Research 95 (1996) 356–373.

[7] Rajendran C. *Two-stage flow shop scheduling problem with bicriteria*, Journal of the Operational Research Society, Volume 43, Issue 9, 1992, Pages 871–884.

[8] C.L. Chen , R.L. Bulfin, *Complexity results for multi-machine multi-criteria scheduling problems.* Proceedings of the Third Industrial Engineering Research Conference, 1994, p.662–5.

[9] Conway, R.W., Maxwell, W.L., Miller, L.W., *Theory of Scheduling*, Addison Wesley, Reading, Mass., (1967), USA.

[10] S.M. Johnson, *Optimal two- and three-stage production schedules with setup times included*, Naval Research Logistics Quarterly 1 (1954) 61–68.

[11] R.M. Aiex, M.G.C. Resende, P.M. Pardalos, and G. Toraldo: "*GRASP with path relinking for the three-index assignment problem*", Technical report, accepted by INFORMS J. on Computing, 2003. `http://www.research.att.com/~mgcr/doc/g3index.pdf`

[12] R. E. Burkard, R. Rudolf, and G. J. Woeginger: "*Three dimensional axial assignment problems with decomposable cost coefficients*", Discrete Applied Mathematics 65, 123–169, 1996.

[13] Crama, Y. and F.C.R. Spieksma: "*Approximation algorithms for three-dimensional assignment problems with triangle inequalities*", European Journal of Operational Research 60, 273-279, 1992.

[14] E. Balas and M.J. Saltzman: "*An algorithm for the three-index assignment problem*". Oper. Res., 39:150–161, 1991.

[15] H.W. Kuhn: "*The hungarian method for the assignment problem. In Naval Research*". Logistics Quarterly, volume 2, pages 83-97, 1955.

[16] D. Goldberg: "*Genetic Algorithms in Search, Optimization and Machine Learing*", Addison-Wesley, Reading, MA, 1989

[17] G. Syswerda: "*Schedule Optimization Using Genetic Algorithms*", in L.Davis (Ed.) Handbook of Genetic ALgorithms, pp.332–349, Van Nostrand Reinhold, New York, 1991