Procedia Computer Science

Volume 29, 2014, Pages 84–94

ICCS 2014. 14th International Conference on Computational Science

A CUDA Based Solution to the Multidimensional Knapsack Problem Using the Ant Colony Optimization*

Henrique Fingler[1], Edson N. Cáceres[1], Henrique Mongelli[1], and Siang W. Song[2]

[1] Universidade Federal do Mato Grosso do Sul
Campo Grande, MS, Brazil
caceresen,henrique.fingler,hmongelli@gmail.com
[2] Universidade de São Paulo
São Paulo, SP, Brazil
song@ime.usp.br

Abstract
The Multidimensional Knapsack Problem (MKP) is a generalization of the basic Knapsack Problem, with two or more constraints. It is an important optimization problem with many real-life applications. To solve this NP-hard problem we use a metaheuristic algorithm based on ant colony optimization (ACO). Since several steps of the algorithm can be carried out concurrently, we propose a parallel implementation under the GPGPU paradigm (General Purpose Graphics Processing Units) using CUDA. To use the algorithm presented in this paper, it is necessary to balance the number of ants, number of rounds used, and whether local search is used or not, depending on the quality of the solution desired. In other words, there is a compromise between time and quality of solution. We obtained very promising experimental results and we compared our implementation with those in the literature. The results obtained show that ant colony optimization is a viable approach to solve MKP efficiently, even for large instances, with the parallel approach.

Keywords: parallel programing, gpgpu, ant colony optimization, multidimensional knapsack problem

1 Introduction

The Multidimensional Knapsack Problem, or MKP [18], is an important optimization problem with many real-life applications. It has been studied for decades because there are many variations all of which can be proven to be NP-hard [13].

Informally, we can state the KP as follows. *A traveller has a knapsack with a certain capacity. This knapsack must be filled with different kinds of objects that will be useful during the trip. Each object takes some amount of space and has a certain value to the traveller. Given a collection of objects, which ones should he put in the knapsack?* In KP, there is only

*Partially supported by CNPq grant no. 30.1652/09-0

doi:10.1016/j.procs.2014.05.008

one constraint, namely, the total space of the selected objects should not exceed that of the knapsack capacity.

The multidimensional knapsack problem (MKP) is a generalization of the knapsack problem, in the sense that there are $m > 1$ constraints.

Finding optimal solutions for MKP may be intractable and approximation algorithms and heuristic algorithms have been used to obtain approximate solutions [22]. One of the most common heuristic methods used is the metaheuristic algorithm that iteratively search for sufficiently good solutions. In this paper we use a metaheuristic algorithm based on ant colony optimization (ACO) [10]. Since several steps of the algorithm can be carried out concurrently, we propose a parallel implementation under the GPGPU paradigm (General Purpose Graphics Processing Units), where the GPU (Graphics Processing Unit) computing power is utilized for general-purpose computing.

The ant colony metaheuristic attempts to imitate the behavior of ants leaving an anthill looking for food. The nearer the food source is from the anthill the better. Initially, while an ant walks randomly, it deposits pheromone along the path it takes. The deposited pheromone will guide it to return to the anthill, and also will help other ants to the path taken. This pheromone evaporates with time, therefore, a path taken by many ants will have more pheromone than that taken by less ants. When an ant finds a food source, it returns to the anthill. If this food source is far away from the anthill, a greater amount of pheromone will have evaporated. On the other hand, if it is near the anthill, there will still be a good amount of pheromone on the return path. Paths to food sources nearby implies less round trip time and thus will have more pheromone than more distant food sources. The more pheromone on a given path, the more likely it will be taken. The analogy is that the food sources are solutions for the problem. The nearer the food source is from the anthill, the better is the solution. It is through the deposited pheromone that ants communicate with each other, thus sharing information acquired towards a good solution.

A sequential algorithm based on the ant colony optimization approach to solve the MKP has been presented by Ke et al. [17]. Since the work carried out by each ant is independent, it is simple to parallelize an ant colony optimization algorithm. It suffices to allocate the work of each ant to a processor. In fact, we have shown the success of the parallel approach in a previous work, where we produced a GPGPU solution for solving the Quadratic Assignment Problem [5]. Furthermore, it has been shown that, through a different organization, where the work of each ant is assigned to more than one processor, the results are even better [6]. In this paper, we chose this second approach and implemented the ACO algorithm to solve MKP on a GPGPU, using CUDA, and obtained promising results. The results obtained show that ant colony optimization is a viable approach to solve MKP efficiently, even for large instances, with the parallel approach.

2 Ant Colony Optimization

Inspired by the behavior of ants previously explained, Dorigo proposed the Ant System (AS) algorithm, an implementation of the ACO, to solve the shortest path problem [10]. The initial results were good. However, they did not surpass the results of other techniques for large graphs. Nevertheless, the results were important in the sense that variations of AS were studied and proposed.

ACO can be described as a metaheuristic in which artificial ants communicate with each other through pheromone trails, that work as probabilistic functions to construct or modify solutions. During the execution of the algorithm, information is adapted and updated to reflect the experience of each ant in search of new solutions. Initialization and update of such information are the most important steps of the ACO algorithm.

Besides the work of the ants, there are two other steps: pheromone evaporation and global actions. Pheromone evaporation is a process of updating the entire pheromone table, where the quantity is reduced by an amount defined by a function. This update attempts to avoid the convergence of several ants toward a suboptimal local solution, and thereby allows ants to explore new paths. The global actions step is used to carry out centralized actions that cannot be executed by each ant individually, such as to initiate a local search process or to collect information on solutions constructed by all the ants, to update the pheromone table favoring the local optimal solutions. The ACO approach has been used with good results in several problems, such as the traveling salesman problem (TSP) [25], the quadratic assignment problem [24], the classification problem [20], the vehicle routing problem [9], among others.

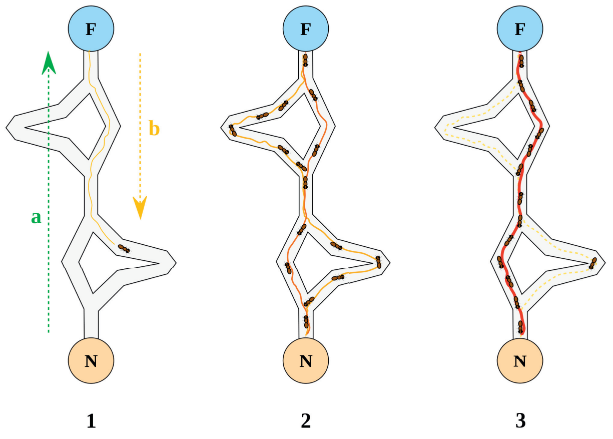

Figure 1: Example of trails between ant Nest and Food source [8].

Figure 1 shows an example. Initially (1) an ant leaves N and, by any path a tries to find a food source. Returning to N through path b, a pheromone trail is deposited by it, so that other ants can follow its path. Due to evaporation of pheromone, shorter trails will have more pheromone (2) and attract more ants. After a certain amount of time (3), the paths used by ants tend to converge to the shortest. Even with such convergence, some ants can still follow trails with few pheromone, due to is probabilistic characteristic.

3 The multidimensional 0-1 knapsack problem (MKP)

In the Binary Knapsack Problem (KP) [18] there is only one constraint, namely, the total space of the selected objects. When filling the knapsack the total space of objects should not exceed the knapsack capacity. The multidimensional knapsack problem (MKP) is a generalization of the knapsack problem, in the sense that there are $m > 1$ constraints, called dimensions from now on.

Consider m capacities $b_i, 1 \leq i \leq m$. Consider a set of n objects $\{1, 2, \ldots, n\}$, where object j has value c_j and has weights a_{ij}, $1 \leq i \leq m$. Let $x_j \in \{0, 1\}, 1 \leq j \leq n$. The multidimensional knapsack problem (MKP) can be defined as follows.

$$\max z = \sum_{j=1}^{n} c_j x_j, \qquad \text{subject to } \sum_{j=1}^{n} a_{ij} x_j \le b_i, \forall i = 1, ..., m \tag{1}$$

Each object j has weights a_{ij} on dimension i and presents a value of c_j if included. The objective is to find a subset of objects that maximizes the total value, without exceeding the capacity of each dimension. We assume, without loss of generality, that $c_j > 0$ for each object, and $b_i > 0$ for each dimension. We assume also $\sum_{j=1}^{n} a_{ij} > b_i$, for all $1 \le i \le m$, so that we cannot pick all the objects.

3.1 Ant colony optimization for the MKP

In this section we present the ACO algorithm to solve MKP using GPU/CUDA. With some modifications due to the use of CUDA, it is based on the algorithm of Solnon and Bridge [23]. Algorithm 1 presents the high-level pseudo-code of the implemented algorithm. In all the algorithms shown, c represents the number of colonies, a is the total number of ants (ants per colony times c), n is the number of objects that can be added and m is the number of dimensions of the problem. We will now explain the ideas.

Algorithm 1 ACO Algorithm for the Multidimensional Knapsack Problem.

```
1: Initialize each position of the pheromone table as $t_{max}$
2: for i from 1 to the maximum number of rounds do
3:     for each ant $k = 1, \dots, m$ do
4:         Add a random object to the knapsack.
5:         Add objects to the knapsack until no more objects can be added, using the desirability
           formula and I-Roulette [6] to choose the objects.
6:     end for
7:     Adjust the best solution of each colony.
8:     Update the table using the best solution of the colony and the best solution generated in
       this round by the colony.
9: end for
```

Instead of representing each ant as a thread, which is the simplest and most intuitive model, our implementation represents an ant as a block when the code of the ants are being executed. When global steps such as the update and initialization are being executed, each block will no longer represent an ant, but rather an ant colony.

In most ant colony systems, there is only one colony, consisting of a set of ants, a table of pheromone and, possibly, variables to control the behavior of the ants. This is because the majority of the algorithms are implemented sequentially. If two colonies were implemented, the running time would be doubled. With the use of GPGPU, however, there are abundant parallel resources to use. Instead of creating many ants that use only one pheromone table, they can be divided into multiple colonies. Because the pheromone update is elitist (only the colony's best solution is used), if many ants use the same table most of their work will not be used, therefore it is better to divide the ants into more colonies.

The pheromone table is an array of n positions, one for each object, and uses the *min-max* concept. There is a minimum value, called t_{min}, and a maximum value, called t_{max} for each position of the table. At the beginning of the algorithm, all the positions are initialized as t_{max}, thus each object has the same pheromone amount.

The desirability of an object is expressed by a value that represents how good the object is for a certain knapsack. The higher the desirability value of an object, the more likely it will be inserted into the knapsack. We denote by S_k the state of the algorithm. This state contains all the data used in the execution, such as the value and weights of each object, the pheromone table, etc. The desirability of an object is computed in the following way.

$$D_i = \tau_{factor}(i, S_k)^{\alpha} + \eta_{factor}(i, S_k)^{\beta} \tag{2}$$

where $\tau_{factor}(i, S_k)$ is the pheromone factor of an object (its position in the pheromone table) and $\eta_{factor}(i, S_k)$ is the heuristic factor of the object, a measure of its cost-benefit value, and is calculated using Equation 3. The values of α and β are parameters that indicate the relative weight of each factor.

The heuristic factor is calculated as follows.

$$\eta_{factor}(i, S_k) = p_i / \sum_{j=1}^{m} (w_{ij}/cr_j) \tag{3}$$

where p_i is the value of object i, w_{ij} is the weight of object i on dimension j and cr_j is the remaining capacity of the knapsack on dimension j. By computing the heuristic factor of an object, each ant verifies if this object fits in the knapsack (an object does not fit if $w_{ij} > cr_j$ for any j) or if it is already in the knapsack. If the object does not fit or is already inside, its desirability value becomes 0, since it is no longer able to add it.

To add an object to the knapsack, the ant should compute the desirability of the n objects and choose one of them. To do this, each ant, represented by a block, launches t threads, where $t \leq n$. However, each block has a limited number of threads that can be launched. This limit is 512 for GPUs with *compute capability* up to 2.0, and 1024 for GPUS 2.0 or more. If this limit is reached each thread takes responsibility of n/t objects.

Each one of the t threads computes the desirability of approximately n/t objects and multiplies by a random number between 0 and 1. The largest computed value is then obtained by parallel binary reduction and is the object chosen to be added. With m threads and an array of m elements, a parallel binary reduction can find the largest value in $O(\log m)$ time. This method is called *I-Roulette* [6].

If n is much larger than t, and t is equal to the the maximum amount of threads, each thread must calculate n/t desirabilities, and only the highest of them is considered. This will not be discussed or shown in the algorithms as it did not happen in the instances shown in this paper and is an implementation detail.

The update of the pheromone table starts with all positions multiplied by $1 - ev$ where ev is the evaporation factor. If the evaporation factor is 1, then all the pheromone evaporates and the algorithm becomes a generator of random solutions. If ev is zero, then no pheromone evaporates. Then the table positions of the objects in the best solution generated are increased by a factor seen in Equation 4, where *Best* is the value of the best solution generated by the colony since the start of the algorithm and *Roundbest* is the best value generated in the current round.

$$1.0/(1.0 + Best - Roundbest) \tag{4}$$

The essential execution step of the algorithm is as follows. For the maximum number of rounds, each ant's selected objects, total profit, capacities and other round based variables are reset, then a random object is added (pseudo-code is shown in Algorithm 2). Then objects are added according to their desirability. Since reading from the GPU's memory is very slow, the algorithm does not keep checking if all knapsacks are full, instead we execute the code to add an object n times. The pseudo-code that adds an object is shown in Algorithm 3. After this step each colony has to find the best solution generated in this round and may have to update the best solution found since the start of the execution. This step is shown in Algorithm 4. At the end of the round the pheromone table has to be updated as in Algorithm 5.

Algorithm 2 Kernel to add a random object to each ant's knapsack. Launched as 1 block and a threads.

```
1: Call cuRAND to get a random $(0, 1]$ number $R$.
2: Add object $(R * n) - 1$ to the ant's knapsack.
3: Set the current knapsack's profit as the selected object's profit.
4: Set the selected object's index in the selected objects array as 1.
5: Set the remaining capacity of each dimension as the maximum capacity minus the object's
   cost.
```

Algorithm 3 Kernel to add an object to the knapsack. Launched as a blocks and n threads. If n is not a power of 2, it is set as the next closest power of 2. The number of threads has to be a power of 2 because of the reduction.

```
1: Get which colony the ant belongs to and which object it is responsible for (thread index).
2: Reset the shared memory at its index.
3: Synchronize all threads to make sure every position of the shared memory was reset.
4: Check if the object fits in the knapsack for all dimensions.
5: Calculate the desirability of the object.
6: Check if object is already in the knapsack, if it is, set its desirability as 0 since it cannot be
   chosen.
7: If this object's probability is higher than that of the previous object or it is the first de-
   sirability this thread computes, store in the shared memory its desirability and number.
8: Do a reduction to get the highest desirability stored in the shared memory.
9: If the desirability is not 0, add this object to the knapsack, updating each dimension's
   capacity, total knapsack profit and the selected objects array.
```

Algorithm 4 Kernel to find the best solutions of each colony generated this round. Launched as c blocks and a/c threads.

```
1: Store this ant's generated solution in the shared memory.
2: Do a reduction to get the highest generated solution in the shared memory.
3: One ant per colony - one thread per block - checks if this solution is better than best solution
   found since the start of the execution, if it is, update it.
```

3.2 Experimental results

The ant colony algorithm for MKP uses a set of input parameters that modifies the execution behavior. Several configurations were tested and the best is shown in the results.

Algorithm 5 Kernel to update the pheromone trail. Launched as c blocks and n threads. Each thread is responsible for one cell (object index) in the matrix

```
1: Calculate the amount of pheromone that may be deposited.
2: Evaporate the pheromone in its cell.
3: If the cell's index is in the knapsack, add the amount pheromone calculated previously.
4: Check and update if the positions violate the maximum/minimum amount possible.
```

The pairs of number of ants/number of colonies tested were: 32/2, 128/4 and 256/8. The weights α and β that produced the best results were 4 and 1, respectively and the best evaporation factor found was 0.1. Varying the t_{min} and t_{max} parameters did not alter significantly the average running times.

The MKP instances used were from the OR library [2], a collection of integer programming problems that contain MKP instances, among others. There are two MKP problem sets in the OR library, the first contains instances from Fréville and Plateau [11]. However, since this set is not frequently used, the results are not shown.

For the second set, the number of dimensions m is 5, 10 and 30, and the number of objects n is 100, 250, and 500. For each combination thirty instances were generated, ten for each tightness factor. The tightness factor of an instance determines the amount of objects the optimal knapsack contains, the smaller the tightness factor, the less number of objects the optimal knapsack contains. More information on the OR library can be found in [12].

When an instance does not present a proven optimum value, the best known solution was considered.

The time shown in the tables are in milliseconds and was obtained as the average of ten executions for each instance. To measure how good the solutions generated are, an average gap is considered. How this value is computed is shown in Equation 5, the smaller the gap, the better. If the gap is zero, the optimum solution was found in the ten executions. The number of executions where the optimum solution was found was also recorded. This value is represented under column OPT. Tables 1 to 2 show the obtained results. The results using 32 ants in two colonies are not shown because they were not used in the comparisons.

$$\frac{100 * (\text{value of obtained solution} - \text{value of the optimum solution})}{\text{value of the optimum solution}} \tag{5}$$

To compare the results of this work with those in the literature ([1], [3], [4] and [14]) we have chosen the configuration of 256 ants and 100 rounds, since it has compatible quality and time consumed. This configurations will be denoted ACO configuration.

Tables 3 and 4 show the data of the ACO configuration. Only the second set will be compared, since it is the most used set and possesses more difficult instances. Table positions containing "-" are those where data were not informed in the literature.

In the column representing time in Table 4, we used a correction factor that takes into account the processor frequency. This factor is computed by dividing the clock frequency of the work in the literature by the clock frequency of this work ($3.3GHz$). The time shown in the table is the product of this factor and original time of each work. This factor is not completely fair, since the implementation in CUDA does not use the clock of the main processor, and there is estimate of the time consumed by a sequential code if it were run on a GPGPU, since this depends on the implementation and the characteristics of the algorithm.

The results of [3] are not shown in the tables since that work implements an exact method based on *branch-and-bound* with *resolution search*. Thus, its running time is hundreds times

Table 1: Execution results for the instances of the second set of 128 ants in four colonies.

			10 rounds			100 rounds		
n	m	α	Time (ms)	Average Gap	OPT	Time (ms)	Average Gap	OPT
100	5	0.25	17	1.0839	0	174	0.6104	6
		0.50	24	0.4798	0	239	0.2313	5
		0.75	30	0.2130	0	304	0.1185	14
250	5	0.25	88	0.6766	0	898	0.4617	0
		0.50	126	0.3192	0	1277	0.2238	0
		0.75	164	0.1496	0	1656	0.0940	0
500	5	0.25	308	0.3664	0	3101	0.2883	0
		0.50	439	0.1659	0	4409	0.1257	0
		0.75	567	0.0921	0	5737	0.0670	0
100	10	0.25	17	2.6951	0	179	2.0092	0
		0.50	25	1.2395	0	258	0.9198	0
		0.75	33	0.5866	1	334	0.4281	6
250	10	0.25	92	1.4070	0	943	1.1747	0
		0.50	138	0.7796	0	1402	0.6298	0
		0.75	184	0.3253	0	1862	0.2514	0
500	10	0.25	323	0.7062	0	3262	0.5699	0
		0.50	485	0.3680	0	4856	0.3039	0
		0.75	647	0.2079	0	6463	0.1713	0
100	30	0.25	22	3.9973	0	223	3.1965	0
		0.50	36	2.5919	0	361	2.1541	0
		0.75	50	1.1659	0	507	0.9684	0
250	30	0.25	113	3.3037	0	1148	2.9607	0
		0.50	185	1.8060	0	1873	1.6096	0
		0.75	261	0.8503	0	2627	0.7522	0
500	30	0.25	406	2.2872	0	4088	2.0467	0
		0.50	667	1.0776	0	6689	0.9784	0
		0.75	928	0.6165	0	9283	0.5539	0

greater in any instance.

In [4] the authors consider a hybrid dynamic programming method (HDP) with an improvement in lower bounds computation (LBC). Considering the table with correction factor, notice that HDP+LBC obtains the best results for all the instances, except in the smallest instance (100×5), where the difference in solution quality is significant (0.26 versus 0.57). The time difference is however very small.

A *kernel search* algorithm (KS) was implemented in [1]. Comparing the results, we notice that the quality of the solutions obtained by KS is extremely high, 0.062 in the worst case that are the largest instances, with size 500×30. The time consumed, however, is also high, around 148 times that of ACO for these large instances and up to 575 times larger for instances of size 250×30. Thus KS finds very good solutions, near the optimum, for all the instances executed, with the price of very long execution time.

Hill, Cho and Moore [4] compare several heuristic methods and implement a new method using problem reduction (NP(R)). The heuristics compared include HDP [4] and a genetic algorithm (GA) [7]. No running times are informed in that paper, thus we only compare the quality of solutions. ACO encountered better solutions for three instance sizes: 100×5, 250×5 and 100×10. However, since time is not shown, we cannot conclude which algorithm is better in each case.

Table 2: Execution results for the instances of the second set of 256 ants in eight colonies.

			10 rounds			100 rounds		
n	m	α	Time (ms)	Average Gap	OPT	Time (ms)	Average Gap	OPT
100	5	0.25	26	0.9442	0	267	0.5119	8
		0.50	36	0.3980	1	365	0.1915	5
		0.75	46	0.1775	3	461	0.0903	18
250	5	0.25	149	0.6090	0	1500	0.4369	0
		0.50	212	0.2872	0	2135	0.1974	0
		0.75	277	0.1368	0	2787	0.0816	0
500	5	0.25	557	0.3450	0	5598	0.2664	0
		0.50	787	0.1517	0	7909	0.1164	0
		0.75	1020	0.0824	0	10225	0.0635	0
100	10	0.25	28	2.5208	0	284	1.8316	0
		0.50	41	1.0806	0	415	0.8114	0
		0.75	53	0.5413	3	540	0.3838	10
250	10	0.25	156	1.3337	0	1571	1.0753	0
		0.50	235	0.7251	0	2364	0.5811	0
		0.75	314	0.3129	0	3157	0.2388	0
500	10	0.25	589	0.6553	0	5920	0.5355	0
		0.50	876	0.3482	0	8790	0.2865	0
		0.75	1162	0.1995	0	11673	0.1580	0
100	30	0.25	35	3.7178	0	356	3.0288	0
		0.50	58	2.4294	0	583	2.0567	0
		0.75	82	1.1367	0	824	0.9297	0
250	30	0.25	193	3.2283	0	1943	2.8289	0
		0.50	320	1.7218	0	3218	1.5595	0
		0.75	450	0.7946	0	4527	0.7258	0
500	30	0.25	736	2.1792	0	7401	1.9765	0
		0.50	1203	1.0448	0	12059	0.9600	0
		0.75	1666	0.5971	0	16730	0.5325	0

Table 3: Comparison of the average gap and running time between the configuration of 256 ants and 8 colonies with local search and data found in the literature for the second set.

		256-8		HDP+LBC [4]		Kernel Search [1]		NP(R) [14]	
n	m	Time (ms)	Average Gap	Time (ms)	Average Gap	Time (ms)	Average Gap	Time (ms)	Average Gap
100	5	364	0.2645	600	0.57	-	-	-	0.53
250	5	2140	0.2386	1000	0.16	196000	0.0	-	0.24
500	5	7910	0.1487	1130	0.07	1088000	0.002	-	0.08
100	10	413	1.0089	1000	0.95	-	-	-	1.10
250	10	2364	0.6317	970	0.32	1465000	0.001	-	0.48
500	10	8794	0.3266	4030	0.16	1579000	0.019	-	0.19
100	30	587	2.005	3970	1.81	-	-	-	1.45
250	30	3229	1.7047	21200	0.77	2189000	0.013	-	0.80
500	30	12063	1.1563	93370	0.42	2100000	0.062	-	0.49

4 Conclusion

When approximate solutions for MKP are sufficient, a heuristic solution such as ACO can be used, for it can find good solutions and does not require much time. To use the algorithm presented in this paper, it is necessary to balance the number of ants and the number of rounds used. In other words, there is a compromise between time and quality of solution: better the solution desired, longer the time required. If there is no exact algorithm for the problem and a solution very close to the optimum is desired, we can configure the algorithm to use a large number of ants and colonies in several rounds.

Table 4: Comparison of the average gap and running time between the configuration of 256 ants and 8 colonies with local search and data found in the literature for the second set with time correction factor.

		256-8		HDP+LBC [4]		Kernel Search [1]		NP(R) [14]	
n	m	Time (ms)	Average Gap	Time (ms)	Average Gap	Time (ms)	Average Gap	Time (ms)	Average Gap
100	5	364	0.2645	291	0.57	-	-	-	0.53
250	5	2140	0.2386	484	0.16	166286	0.0	-	0.24
500	5	7910	0.1487	548	0.07	923059	0.002	-	0.08
100	10	413	1.0089	485	0.95	-	-	-	1.10
250	10	2364	0.6317	470	0.32	1242906	0.001	-	0.48
500	10	8794	0.3266	1954	0.16	1339623	0.019	-	0.19
100	30	587	2.005	1925	1.81	-	-	-	1.45
250	30	3229	1.7047	10278	0.77	1857147	0.013	-	0.80
500	30	12063	1.1563	45266	0.42	1781640	0.062	-	0.49

The hybrid dynamic programming method for MKP with lower bounds computation (HDP+LBC) [4] obtained better results (better quality and less time) than the proposed algorithm, for almost all instances for the second data set of the OR library. Our algorithm compare favorably with respect to the other algorithms. The *kernel search* method [1] obtains very high quality solution, close to the optimum. However, the time consumed if up to 575 times longer than that of this work. The exact *branch-and-bound* method [3] illustrates the difficulty to obtain an optimum solution. Even for a not too large instance (250×10), it may require ten hours. Finally, several metaheuristic solutions [14] were compared and a heuristic of problem reduction is proposed. Since there are no data on the running times, we have no conclusive comments on these algorithms.

As future works, we will implement and compare the ACO method with other parallel solutions, mainly with those using GPGPUs. As examples of such methods, we can mention the multi-start algorithms [21], genetic algorithms [15, 16], and particle swarm optimization algorithms [19].

References

[1] Enrico Angelelli, Renata Mansini, and M. Grazia Speranza. Kernel search: A general heuristic for the multi-dimensional knapsack problem. *Computers & Operations Research*, 37(11):2017 – 2026, 2010.

[2] J. E. Beasley. OR-Library: distributing test problems by electronic mail. *Journal of the Operational Research Society*, 41(11):1069–1072, 1990.

[3] Sylvain Boussier, Michel Vasquez, Yannick Vimont, Saïd Hanafi, and Philippe Michelon. A multi-level search strategy for the 0-1 multidimensional knapsack problem. *Discrete Applied Mathematics*, 158(2):97 – 109, 2010.

[4] V. Boyer, M. Elkihel, and D. El Baz. Heuristics for the 0-1 multidimensional knapsack problem. *European Journal of Operational Research*, 199(3):658–664, December 2009.

[5] E.N. Cáceres, H. Fingler, H. Mongelli, and S.W. Song. Ant colony system based solutions to the quadratic assignment problem on gpgpu. In *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on*, pages 314 –322, sept. 2012.

[6] Jos M. Cecilia, José M. García, Andy Nisbet, Martyn Amos, and Manuel Ujaldûn. Enhancing data parallelism for ant colony optimization on gpus. *Journal of Parallel and Distributed Computing*, (0):–, 2012.

[7] P.C. Chu and J.E. Beasley. A genetic algorithm for the multidimensional knapsack problem. *Journal of Heuristics*, 4:63–86, 1998.

[8] Ringo Doe. File:aco branches.svg, may 2006.

[9] Alberto V. Donati, Roberto Montemanni, Norman Casagrande, Andrea E. Rizzoli, and Luca M. Gambardella. Time dependent vehicle routing problem with a multi ant colony system. *European Journal of Operational Research*, 185(3):1174 – 1191, 2008.

[10] M. Dorigo. *Optimization, Learning and Natural Algorithms.* PhD thesis, Politecnico di Milano, Italy, 1992.

[11] A. Freville and G. Plateau. Hard 0-1 multiknapsack test problems for size reduction methods. *Investigation Operativa*, 1:251–270, 1990.

[12] Arnaud Fréville. The multidimensional 0-1 knapsack problem: An overview. *European Journal of Operational Research*, 155(1):1 – 21, 2004.

[13] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA, 1990.

[14] Raymond R. Hill, Yong Kun Cho, and James T. Moore. Problem reduction heuristic for the 0-1 multidimensional knapsack problem. *Comput. Oper. Res.*, 39(1):19–26, January 2012.

[15] John H. Holland. Outline for a logical theory of adaptive systems. *J. ACM*, 9(3):297–314, July 1962.

[16] John H. Holland. *Adaptation in natural and artificial systems.* MIT Press, Cambridge, MA, USA, 1992.

[17] Liangjun Ke, Zuren Feng, Zhigang Ren, and Xiaoliang Wei. An ant colony optimization approach for the multidimensional knapsack problem. *Journal of Heuristics*, 16(1):65–83, February 2010.

[18] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack problems.* Springer, 2004.

[19] J. Kennedy and R. Eberhart. Particle swarm optimization. In *Neural Networks, 1995. Proceedings., IEEE International Conference on*, volume 4, pages 1942 –1948 vol.4, nov/dec 1995.

[20] D. Martens, M. De Backer, R. Haesen, J. Vanthienen, M. Snoeck, and B. Baesens. Classification with ant colony optimization. *Evolutionary Computation, IEEE Transactions on*, 11(5):651 –665, oct. 2007.

[21] Rafael Mart, Mauricio G.C. Resende, and Celso C. Ribeiro. Multi-start methods for combinatorial optimization. *European Journal of Operational Research*, 226(1):1 – 8, 2013.

[22] Jakob Puchinger, Günther R. Raidl, and Ulrich Pferschy. The multidimensional knapsack problem: Structure and algorithms. *INFORMS J. on Computing*, 22(2):250–265, April 2010.

[23] Christine Solnon and Derek Bridge. An ant colony optimization meta-heuristic for subset selection problems. In N. Nedjah and L. M. Mourelle, editors, *Systems Engineering using Swarm Particle Optimization*, pages 7–29. Nova Science Publishers, 2006.

[24] Thomas Stützle. Max-min ant system for quadratic assignment problems, 1997.

[25] Thomas Stützle and Marco Dorigo. Aco algorithms for the traveling salesman problem, 1999.