Extensive research has been devoted to resource constrained project scheduling problem (RCPSP). Resources are renewable and there is a unique way to perform the activities. This work develops a population based evolutionary algorithm namely differential evolution (DE) to schedule project activities to minimize makespan subject to precedence constraints and resources availability. The proposed DE uses a priority value based representation to encode a project schedule and a serial generation scheme to obtain the schedule. The DE algorithm is compared with some existing algorithms available in the literature on the basis of a computational experiment performed on Patterson’s test bed. Obtained results show that the performance of the proposed DE is quite satisfactory.

© 2013 Production and hosting by Elsevier B.V. on behalf of King Saud University.

1. Introduction

The resource-constrained project scheduling problem (RCPSP) is a combinatorial optimization problem consisting of activities that must be scheduled such that makespan be minimized. Constraints include observation of precedence relation between activities and satisfaction of the resources limitation. These constraints make the problem as a NP-hard one (Blazewicz et al., 1983). There are three basic solving approaches for RCPSP including exact methods, heuristics or priority-rule based approaches and meta-heuristic resolution procedures. Some good reviews about solution approaches can be found in Hartmann and Kolisch (2000), Kolisch and Padman (2001) and Kolisch and Hartmann (2006).

As mentioned above RCPSP belongs to NP-hard optimization problems, therefore application of exact algorithms lead to impractical execution time when the number of activities increases. Many studies solve the RCPSP by applying the meta-heuristics approaches. Some of these methods are briefly described below. Genetic algorithm (GA) is a problem solving technique based on the evolutionary ideas of natural selection as global evolution which have been successfully applied to a noticeable number of project scheduling problems (Hartmann, 1998; Hartmann, 2002; Kim et al., 2003; Kohlmorgen et al., 1999; Lee and Kim, 1996; Leon and Ramamoorthy, 1995; Mendes et al., 2009; Valls et al., 2008). It states a likeness between a set of solution problems to be solved and the set of individuals in a natural population. Solution information is codified in a string called chromosome. Then the algorithm
tries to improve the chromosome’s potential called fitness function by some operators. Zamani (2013) has recently developed a new GA that the innovative component of the algorithm is the use of a magnet-based crossover operator that can preserve up to two contiguous parts from the receiver and one contiguous part from the donator genotype. For this purpose, a number of genes in the receiver genotype absorb one another to have the same order and contiguity they have in the donator genotype. The ability of maintaining up to three contiguous parts from two parents distinguishes this crossover operator from the powerful and famous two-point crossover operator, which can maintain only two contiguous parts, both from the same parent. Simulated annealing algorithm (SA) is a stochastic method for combinatorial optimization problem. This algorithm tries to minimize the thermal energy of the system by cooling down temperature parameter. When the thermal energy of the system minimized that means this solution is a stable state and so is good solution. Also the SA uses a mechanism to avoid trapped on the local optimum. There are some papers about SA application to solve project scheduling problems (Boctor, 1996; Bouleimen and Lecocq, 2003).

Tabu search (TS) is an approach that records the solutions which have been ever obtained, therefore prevents the search from sinking into the local minimum (Glover, 1989, 1990; Thomas and Salhi, 1998). Ant colony optimization (ACO) mimics behavior of ants in finding food. In ACO a colony of artificial ants based on modifying pheromone trails iteratively constructs solution during the algorithm’s execution (Lo et al., 2008; Merkle et al., 2002). Another meta-heuristic that has been widely applied for solving scheduling problems is particle swarm optimization (PSO) (Zhang et al., 2006). In PSO a swarm of particles searches the solution space and the position of a particle indicates a solution of problem. In each generation each particle would searches for the best position with best fitness based on the global experience of the swarm and the individual experience of the particle. Artificial bee colony (ABC) is one of the most recently defined algorithms by Karaboga (2005), motivated by the intelligent behavior of honey bees. It is as simple as particle swarm optimization (PSO) and differential evolution (DE) algorithms, and uses only common control parameters such as colony size and maximum cycle number. Jia and Seo (2013) proposed two alternative approaches, applying the facility layout problem (FLP) concept and integrating the permutation-based artificial bee colony (PABC) algorithm, to effectively tackle the resource-constrained project scheduling problem (RCPSP). In the FLP formulation, the constraints are expressed to design the activities in the space constructed by resource and temporal restrictions, without violating the precedence relationships and overlaps between the activities. For dodging the difficulty of the FLP-based model to treat large-sized instances of NP-hard RCPSP, the permutation representation scheme of the PABC algorithm is in turn introduced utilizing the artificial bee colony (ABC) process to search the best solution for RCPSP. Variable neighborhood search (VNS) designed to find near-optimal solutions. VNS performs a systematic change of neighborhood in conjunction with a set of typical local search moves, and it has been successfully applied to scheduling problems (Fleszar and Hindi, 2004).

In this research evolutionary meta-heuristic algorithm DE to solve the RCPSP is investigated. DE invented by Storn and Price (1997), is a powerful technique to combine simple arithmetic operators with the classical crossover, mutation and acceptance operators. The basic scheme in DE is generating trial parameter vectors. Mutation and crossover are used to generate new vectors (trial vectors), and selection then determines which of the vectors will survive the next generation. There are researches about DE application to solve project scheduling problems. Damak et al. (2009) solved the multi mode resource constrained project scheduling problem (MRCPSP) with a differential evolution algorithm. In this approach a solution is represented by a mode assignment vector and a Position vector. Neighbor solutions are generated using two mutation and crossover operators. Selection operator uses the values of the objective function which is penalized for infeasible solutions. The performance of this algorithm is evaluated on the benchmark instances. The obtained results are compared with the results obtained by other approaches, simulated annealing by Bouleimen and Lecocq (2003) and particle swarm optimization by Jarboui et al. (2008). Rahimi et al. (2013) used a DE algorithm to solve the project scheduling problem under the mode identity constraints (MIRCPSP). In order to improve the quality of the employed DE a local search and learning module is combined with the proposed algorithm. The performance of the DE is evaluated on various test problems by statistically comparing their solution in term of the objective function and computational times.

Rest of the paper is organized as follows: Section 2 explains the RCPSP. Section 3 describes the DE algorithm and its adoption to RCPSP. Section 4 shows the results. Finally concluding remarks come in Section 5.

2. Problem definition

Practice shows that resources constitute an essential feature of any project. In this section we present a formulation of the basic resource-constrained project scheduling problem, referred to as the RCPSP. The RCPSP is a classical discrete problem, i.e. the planning horizon is divided into a discrete number of time periods, activity durations are discretely-divisible, and resources are discrete.

Let us consider a set of \( n \) non-preemptable activities of durations \( d_i, i = 1, 2, \ldots, n \). Precedence constraints between activities mean that no activity may start before all its predecessors are completed. Activities are labeled from \( A_0 \) to \( A_{n+1} \), with activity \( A_0 \) being the unique initial activity without predecessors (source), and \( A_{n+1} \) being the unique terminal activity without successors (sink). If such an activity \( A_0 \) (or \( A_{n+1} \)) does not naturally exist, then a dummy activity of zero duration and zero resource requirements is added appropriately. Moreover, each activity requires some discrete renewable resources, i.e. such that only their temporary availability at every moment is constrained. We assume that there are \( R \) scarce resources and the number of available units of resource \( k, k = 1, \ldots, R \), is \( R_k \). Moreover, all activities and resources are available at the start of the project. The objective of the RCPSP is to find precedence- and resource-feasible completion (or start) times for all activities such that the duration of the project is minimized.

The RCPSP may be formulated as an integer programming problem. The \( 0-1 \) decision variable \( x_{jt} = 1 \) if activity \( A_j \) is assigned a completion time at the end of period \( t \); otherwise, \( x_{jt} = 0 \). Associated with each activity \( A_j \) are its earliest finish
time $EF_j$ and latest finish time $LF_j$, calculated as in (Kelley and Walker, 1959). The value of $LF_{j+1}$ is set equal to the scheduling horizon $H$, which never exceeds the sum of all activity durations. Mathematical model of problem can be showed as follow (Pritsker et al., 1969):

Minimize $\sum_{j=1}^{LF_{j+1}} lx_{t+1,j}$ \hspace{1cm} (1)

Subject to:

$\sum_{j=1}^{LF_j} x_{j} = 1$ \hspace{0.5cm} for $j = 0,\ldots,n + 1$ \hspace{1cm} (2)

$\sum_{j=1}^{LF_j} lx_{t} \leq \sum_{j=1}^{LF_j} lx_{t} - d_j$ \hspace{0.5cm} for all $(A_i, A_j) \in P$ \hspace{1cm} (3)

$\sum_{i=1}^{n} \sum_{q=\text{max}(X_{EF_j})}^{\text{min}(X_{d_j}, X_{LF})} r_{kq} x_{kl} \leq R_k$ \hspace{0.5cm} for $k = 1,\ldots,R$; \hspace{1cm} (4)

$t = 1,\ldots,H$

$x_{jt} \in \{0,1\}$ \hspace{0.5cm} for $i = 0,\ldots,n + 1; t = EF_j,\ldots,LF_j$ \hspace{1cm} (5)

Constraints (2) ensure that each activity is completed exactly once. The set of all pairs of activities $(A_i, A_j)$ such that $A_i$ directly precedes $A_j$ is denoted by $P$. Hence, precedence constraints are represented by inequalities (3). Constraints (4) guarantee that no more than the available number of units of each resource are required in any time period, and constraints (5) state that we consider binary decision variables. The solution of the problem (1)–(5) defines an optimal schedule as a list of activity completion times.

3. Differential evolution

Differential evolution (DE) is a stochastic, population-based optimization. DE utilizes concepts borrowed from the broad class of evolutionary algorithms (EAs) like genetic algorithm (GA). Several mechanisms, such as mutation, crossover and acceptance, are applied to recombine existing solutions to obtain new ones and to find a near-optimal or at least satisfying solution. Individuals in DE are represented by $D$-dimensional vectors $x_i$, $\forall i \in \{1,\ldots, NP\}$, where $D$ is the number of objective parameters and $NP$ is the population size. The evolution process starts with the creation of an initial population, containing individuals with randomly generated element (gene) values. Initially, the mutation operation is applied, in which three individuals are selected randomly and then the gene values of the first individual are added to the differences of the gene values of the two other individuals. It can be stated as:

$$y_i(k) = \begin{cases} v_i(k), & \text{if } R_k \leq cr \text{ or } k = \text{random} \\ x_i(k), & \text{otherwise} \end{cases} \hspace{1cm} (7)$$

where $cr \in (0,1)$ is the predefined crossover rate constant, $R_k \in (0,1)$ drawn randomly for each $k$, and $k_{\text{random}}$ is a randomly chosen integer in the set $\{1, 2, \ldots, D\}$. After mutation and crossover processes, acceptance is applied. The trial individual’s fitness is calculated and compared to that of target individual and the fitter of the two individuals ($x'_i$) is accepted to move to the next generation:

$$x'_i = \begin{cases} y_i, & \text{if } f(y_i) \leq f(x_i) \\ x_i, & \text{otherwise} \end{cases} \hspace{1cm} (8)$$

A new population results from the execution of the above procedure for all individuals of a population, and this is repeated until a predefined termination criterion is reached. The best individual of the last generation is taken as the solution to the problem. The components of the proposed DE algorithm are explained as follows:

3.1. Solution representation

A solution is represented by a $n$ element vector $(I)$, in which the $j$th element $p_{ij} \in \{1, 2, \ldots, n\}$, $j = 1, 2, \ldots, n$ indicates the priority value of activity $j$ (priority list):

$I = (p_{11}, p_{12}, \ldots, p_{1n}) \hspace{1cm} (9)$

We employ the serial schedule generation scheme (SSGS) to derive the schedule related to an individual. Since the makespan criterion is a regular performance measure, i.e. a measure which is non-decreasing in activity completion times, we may use the serial SGS rule to construct the schedule. As a result, there is no danger of omitting an optimal schedule by using the serial SGS here. Hence, having got an individual $I$, the corresponding schedule is computed by the following procedure:

(1) Let $l = 1$.
(2) Set the dummy start activity 1 at time 0.
(3) $l = l + 1$.
(4) Select the activity $j$ with the priority value $p_{ij}$ equal to $l$.
(5) Compute the earliest precedence and resource feasible start time of activity $j$.
(6) If the dummy end activity $n$ is a scheduled activity, stop (where the finish time of the dummy end activity is defined as the objective function value of the related solution), otherwise go to step 3.

3.2. Initial population

Each individual $I$ of initial population is randomly computed as follows: Starting with an empty $n$ element vector, we obtain a priority value list with respect to the precedence constraints by repeatedly applying the following step: the next activity $j$ is randomly determined from the set of eligible activities $(EJ)$, that is, those activities the predecessors of which are already scheduled. Then the next member of the set $\{1, 2, \ldots, n\}$ (the set of order numbers from 1 to $n$) is assigned to the priority value of the activity $j (p_{ij})$. The same process is repeated for a pre-specified number of solutions equal to the size of population (pop-size).
Project scheduling with limited resources using an efficient differential evolution algorithm

Resource limit: $R_l = 6$ units, $R_r = 6$ units, $R_u = 6$ units

**Figure 1** Precedence relationship between activities in Patterson’s test bed.

**Table 1** Obtained schedule by DE.

|   | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|   | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 |
|   | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Total makespan = 64.

**Figure 2** Comparison of the proposed DE with other approaches.

**Figure 3** Convergence of makespan with the number of generations.
3.3. Recombination operators

Each individual of new population is formed by applying the DE operators to the individuals in the previous generation as follows:

**Mutation operators:** Let \( I_1, I_2, I_3, i_1, i_2, i_3 \in [1, NP] \) be three randomly chosen individuals from current population. To create a new (mutant) individual, the mutation operation is implemented in the three priority lists with scale factor \( F_{pv} \) according to the equation \( v_{mut} = pv_i + F_{pv}(pv_{i} - pv_{j}) \), where \( v_{mut} \) is the mutant individual and \( pv_{j} \) is the priority value list of individual \( j \).

**Crossover operators:** The crossover operator create a new (trial) individual by combining parts from two individuals, involving an individual of current population (target individual) and the mutant individual with a crossover rate equal to \( cr_{pv} \). To do this, a random number \( R_k \in (0,1) \) is generated for each element \( k \) of priority list. If \( R_k \leq cr_{pv} \), the element of the mutant individual is selected to copy into the trial individual, else the element of the target individual is selected.

After applying the crossover operator, the structure of the trial individual may violate from the defined solution representation (because the elements of the trial individual may transform into the undetermined values). To overcome this, after the crossover operator is implemented, the trial individual is converted into the prior representation by applying the following procedure:

1. Let \( l = 1 \).
2. Set the priority value of the dummy start activity to 1.
3. \( l = l + 1 \).
4. Determine the set of eligible activities \( EJ \), that is, those activities the predecessors of which are already scheduled.
5. Select the eligible activity \( j \) with the lowest priority value: \( pv_j = \min \{ pv_i | i \in EJ \} \) and convert the value of \( pv_j \) into the \( l \).
6. If the dummy end activity \( n \) is a selected activity, stop, otherwise go to step 3.

**Acceptance operator:** After completion of the mutation and crossover operations, the objective function value of the trial individual is compared to that of target individual. If objective function value of the trial individual is equal or less than the value at the target individual, the trial individual is selected to enter the next generation. Otherwise, the target individual is accepted to move to the next generation.

4. Implementation and comparisons

This section tests the performance of proposed algorithm on RCPSP by means of computational experiment using a specific problem.

The adopted DE algorithm to the RCPSP has been coded in the Borland C++ version 5.02. Details of the program code are provided in Appendix section. The experiment has been performed under windows XP professional on a personal computer with an Intel Core2Dou, 2.5 GHz processor and 3 GB memory. To compare proposed algorithm with other existing methods in the literature as presented in Wu et al. 2011, well-known Patterson’s test bed is selected. According to Patterson’s test bed there are 27 activities which start and end activities are dummies. Each activity uses fixed unit requirements of three types of resources. Other information needed to solve the problem is presented in Fig. 1.

According to Patterson’s test bed there are 27 activities that start and end activities are dummies. Each activity has fixed unit requirements of three different resources. Fig. 1 presents all the details required to solve the problem.

By implementing DE on mentioned test bed we reached the total makespan of 64. Obtained final schedule is presented in Table 1.

In Fig. 2 comparison of DE with other approaches in the literature (Wu et al., 2011) is provided.

As shown in Fig. 2 DE is superior to LFT, GRU, SIO, MINSLK, RSM, RAN, and MJP. Both DE and CBIIA have the same makespan of 64. But when the computational time are examined, it is founded the time needed to reach same excellent makespan in DE is 0.1 s while in CBIIA is 0.41185 s. Therefore DE is preferred algorithm when the time elapsed to reach near-optimal solution is investigated. Another significant index that do comparison 2 algorithms is number of iterations to get final result. It shows more the importance when the problem size is increased, hence number of iteration will get larger to explore whole solution space. DE reaches to excellent solution at 20 iterations while total iterations in CBIIA are 52. Consequently DE With less than half the number of iterations strictly is better than CBIIA. The convergence trend of DE is shown in Fig. 3.

5. Conclusion

In this paper well-known RCPSP with minimization of makespan was investigated which is a well-known computationally complex problem. The importance of the objective in today competition world is clear that force companies to finish the project in a minimum time. Because of NP-hardness of the problem heuristic or meta-heuristic approaches was needed to solve the problem. So we developed an efficient evolutionary algorithm named DE. Then the results compared with some existing algorithms available in the literature. Results showed that DE is superior to compared approaches. In future, the DE can be hybrid with different types of local search, heuristics, meta-heuristics and constraint handling approaches in order to improve its performance to solve more complex RCPSP test instances (J30, J60 and J120) provided in PSPLIB.
Appendix. Program code of the proposed DE

```c
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include<time.h>
#include<time.h>
#include<dos.h>

float random_generate();
int makespan(int *);
void initialGant();
void inputFromFile();
void mutation();
void decode(float *,int);

int Pnum,Remun,resource[4],duration[150],op_resource[150][4],pre_op[150][100],SF,CS[150],CS
F_n_mode,Gamt[4][20000],mi[150];

int past_op[150][100],totalact,totalact2,n_activity[20],time2;
int chrom[150],popsize,L1,L2,parent1[150],parent2[150];
int pop[5000][150],child[5000][150];
int nc,nn,n,fitness[5000];
float probability[10000];
float A1,A2,Ca,Cm,RANDa,RANDm;

int main()
{
    int i,j,ign,k,preBS,Best_time,time2,Bfitness,NR,B_index;
    int hour1,hour2,minute1,minute2,second1,second2,hssecond1,hssecond2,NP;
    ifstream fp4("input_parametr");
    if (!fp4)
    {
        cout<<"cannot open file."
exit(0);
    }
    fp4>>A1>>Ca>>popsize>>RANDa>>NR>>NP;
    ofstream fp2("output file");
    if (!fp2)
    {
        cout<<"cannot open file."
exit(0);
    }
    ofstream fp3("output file2");
    if (!fp3)
    {
        cout<<"cannot open file."
exit(0);
    }
    ifstream fp("input file");
    if (!fp)
    {
        cout<<"cannot open file."
exit(0);
    }
    srand(time(0));
    fp>>totalact2;nmode=1;Remun=3;
    for (int np=0,npc=NP,np++)
    {
        totalact=totalact2;
        for (i=1;i<totalact-1;i++)
        {
            fp>>ign>>ign>>past_op[i][0];
            for (j=1;j<=past_op[i][0];j++)
                fp>>past_op[i][j];
            if (past_op[i][0]==1&&past_op[i][1]==totalact)
                past_op[i][0]=0;
        }
        for (i=0;i<totalact-1;i++)
            for (j=1;j<=past_op[i][0];j++)
```
```c
post_op[i][j++];
past_op[i][j]=0;
for (i=0;i<totalact;i++)
{
    post_op[i][j]=0;
    for (j=0;j<Renum[j]+1)
    {
        post_op[i][j]=0;
        for (k=0;k<op_resource[j][k];)
            post_op[i][j]=0;
    }
}
for (i=0;i<totalact;i++)
{
past_op[i][j]=0;
for (j=0;j<Renum[j]+1)
{
past_op[i][j]=0;
    for (k=0;k<op_resource[j][k];)
        past_op[i][j]=0;
}
}

//makespan
int makespan(int *solution)
{
    int i,j,k,pointer,tc[150],tct;
    for (i=0;i<totalact;i++)
    {tc[i]=0;
    for (j=0;j<totalact;j++)
    {x=solution[i];
        pointer=0;
        for (j=1;j<pre_op[x][j][j]+1)
        {z=x;pre_op[x][j][j];
            if (z)pointer
            pointer=z;
        }
        pointer++;t=deacart[x];
        for (i=0;i<t;j++)
        {if (k=0;k<Renum;k++)
            if (op_resource[x][k]>=Gant[k][pointer])
            {j=1;break;
                pointer++;}
        pointer--;t[pointer]=pointer;
        for (j=0;j<k;j++)
        {for (k=0;k<Renum;k++)
            Gant[k][pointer]=op_resource[x][k];
        pointer--;
    }
}
tc[0]=0;
for (i=0;i<totalact;i++)
if (tc[i]>tct)
tct=tc[i];
```
float random_DTABLE[0][1];

void mutation()
{
    int r = random_popsize();
    for (i = 0; i < popsize(); i++)
    {
        if (random() < 0.5)
        {
            r = random_popsize();
            for (j = 0; j < popsize(); j++)
            {
                if (random() < 0.5)
                {
                    x = solution[i][j];
                    y = solution[j][i];
                    solution[i][j] = y;
                    solution[j][i] = x;
                }
            }
        }
    }
}

void decode(float* m, int vector, int k)
{
    int i, j, k, pointer, i(150), x;
    for (i = 0; i < totalactivity + 1; i++)
    {
        solution(i, i); //solution[i][i]=0;
        pointer = 0;
        for (j = 1; j < popsize(); j++)
        {
            x = solution(i, j); //solution[i][j]=x;
            if (x == 0)
            {
                pointer = j;
                for (j = 1; j < totalactivity + 1; j++)
                {
                    solution(x, j) = 0;
                    pointer = j;
                }
                break;
            }
        }
    }
}

int main()
{
    int i, j, k, pointer, i(150), x;
    for (i = 0; i < totalactivity + 1; i++)
    {
        solution(i, i); //solution[i][i]=0;
        pointer = 0;
        for (j = 1; j < totalactivity + 1; j++)
        {
            x = solution(i, j); //solution[i][j]=x;
            if (x == 0)
            {
                pointer = j;
                for (j = 1; j < totalactivity + 1; j++)
                {
                    solution(j, i) = 0;
                    pointer = j;
                }
                break;
            }
        }
    }
}
References


