Annals of Software Engineering 11, 107–139, 2001

Genetic Algorithms for Project Management

CARL K. CHANG chang@uic.edu
Department of EECS (M/C 154), The University of Illinois at Chicago, Chicago, IL 60607, USA

MARK J. CHRISTENSEN, PH.D. markchri@concentric.net
Independent Consultant, St. Charles, Illinois

TAO ZHANG Tao_Zhang-CTZ020@email.mot.com
Motorola-iDEN Engineering Development, 1301 E. Algonquin Rd, Schaumburg, IL 60196, USA

Abstract. The scheduling of tasks and the allocation of resource in medium to large-scale development projects is an extremely hard problem and is one of the principal challenges of project management due to its sheer complexity. As projects evolve any solutions, either optimal or near optimal, must be continuously scrutinized in order to adjust to changing conditions. Brute force exhaustive or branch-and-bound search methods cannot cope with the complexity inherent in finding satisfactory solutions to assist project managers. Most existing project management (PM) techniques, commercial PM tools, and research prototypes fall short in their computational capabilities and only provide passive project tracking and reporting aids. Project managers must make all major decisions based on their individual insights and experience, must build the project database to record such decisions and represent them as project nets, then use the tools to track progress, perform simple consistency checks, analyze the project net for critical paths, etc., and produce reports in various formats such as Gantt or Pert charts.

Our research has developed a new technique based on genetic algorithms (GA) that automatically determines, using a programmable goal function, a near-optimal allocation of resources and resulting schedule that satisfies a given task structure and resource pool. We assumed that the estimated effort for each task is known a priori and can be obtained from any known estimation method such as COCOMO. Based on the results of these algorithms, the software manager will be able to assign tasks to staff in an optimal manner and predict the corresponding future status of the project, including an extensive analysis on the time-and-cost variations in the solution space. Our experiments utilized Wall's GALib as the search engine. The algorithms operated on a richer, refined version of project management networks derived from Chao's seminal work on GA-based Software Project Management Net (SPMnet). Generalizing the results of Chao's solution, the new GA algorithms can operate on much more complex scheduling networks involving multiple projects. They also can deal with more realistic programmatic and organizational assumptions. The results of the GA algorithm were evaluated using exhaustive search for five test cases. In these tests our GA showed strong scalability and simplicity. Its orthogonal genetic form and modularized heuristic functions are well suited for complex conditional optimization problems, of which project management is a typical example.

Keywords: genetic algorithm, scheduling, objective function, optimization, project management

Computer programs that "evolve" in ways that resemble natural selection can solve complex problems even their creators do not fully understand [Holland 1992].

1. Introduction

Software project management involves the scheduling, planning, monitoring, and control of the people, processes, and resources to achieve specific objectives, while satisfying a variety of constraints. In the most general form, the resource-constrained scheduling problem poses the question:

> "*Given a set of tasks, resources, and the way to evaluate performance, what is the best schedule to assign the resources to the activities such that the performance is maximized*?" [Wall 1996].

Tasks may be anything from maintaining documents to writing a C++ class. Resources include people, skills, time, equipment, and facilities. Typical PM objectives include minimizing the duration of the project, maximizing the quality of the product and minimizing the cost of completing the project. Note that scheduling and planning are different topics. A plan defines what must be done along with any conditions, constraints, and restrictions that must be satisfied, while a schedule specifically describes both how and when it will be done. Thus they are related by the act of assigning individuals and other resources to tasks. Task assignment specifically addresses on the question of "*Who does what and when*?" In this paper, scheduling is synonymous to job assignment unless otherwise specified.

In general, the scheduling problem is *NP complete*, meaning that there are no known algorithms for finding optimal solutions in polynomial time [Ozdamar 1995]. Exhaustive search methods can be used to solve scheduling problems, but requires forbiddingly long execution times as the problem size increases. This paper presents a heuristic method, a *genetic algorithm* (GA), to solve the scheduling problem in project management. Instead of evaluating every possible point in the solution space to find an optimal one, heuristic methods explore the *landscape* of possible solutions by a combination of iterative "rules of thumb" or "trial and error" techniques, with the key discriminate between such methods being the technique used to drive, or direct, the iteration steps. Many strategies can be used to provide this direction. Genetic algorithms are one such strategy. John Holland of the University of Michigan first developed genetic algorithms in 1975 [Holland 1992]. Borrowing ideas from Darwin, Holland devised mechanisms that operate on a selected population of candidate solutions. He employed the perturbation mechanisms of *mutation*, *replacement* and *crossover* to evolve better solutions. Holland's original algorithm was quite simple, yet remarkably robust, in finding optimal solutions to a wide variety of problems. Many *custom GAs* exist today to solve very large and complex real-world problems using methods only different from those of Holland. In a earlier paper [Chang and Christensen 1999] we reported research work that utilized genetic algorithms to solve resource allocation and scheduling problems, based on the original work of Chao [1995]. Concurrently, Wall [1996] created a package of GA classes with hierarchical structures. Wall's implementation allows users to incorporate their own interface and other functionality. We fully utilized Wall's GA library to attack the job assignment problem.

Among the questions addressed in this paper are:

- What are the assumptions used in project management?
- What are realistic objective or goal functions that can be used in project management?
- Why the new GA reported in this paper is better?
- How to validate our GA solutions?
- What is the computational complexity of the GA solution compared to exhaustive search?
- How do different parameter setting impact the performance of GA solution?

Since this work was derived from that of [Chao 1995], we include table 1 highlighting similarities and differences between [Chao 1995] and our present work.

Table 1
The comparison between new GA and Chao's (1995) GA solution.

New GA	Chao's GA
Genetic representation is orthogonal 2D array.	*Genetic representation are 2 sets of linked lists.*
A 2D array is used, with one dimension for tasks, the other for employees.	Two lists are used to represent a schedule, one for tasks, the other for employees. Duplicate cross-references between the two lists representing the schedule, which is redundant.
The 2D array stores all needed information, speeds up memory access and avoids dynamic memory allocation.	Memory access is sequential and slow.
New GA supports one to many or many to one assignment.	*Earlier GA can only support one to one assignment.*
2D array can represent schedules for multiple, concurrent projects, either one-to-one (one employee can do one job at a time), one-to-many (one can do many jobs at a time), many-to-one (many employees can do one job together), or many-to-many.	Linked list only supports one to one employee/task assignments.
New GA supports partial commitment.	*Earlier GA supports binary commitment.*
Employees are assigned partial, discrete allocation of time, currently from {0, 0.25, 0.5, 0.75, 1} percent.	Employees are assigned either 0% or 100% percentage of commitment for one task.
New GA prefers post-checking.	*Earlier GA prefers pre-checking.*
The validity of a solution is checked after assignment;	The validity of solution is checked before the genetic evolving process;
The objective function is isolated from genetic operating process;	The objective function is integrated into the genetic operators;
Effect: Only objective function needs to be modified for different projects.	Effect: Both operators and objective functions need to be modified for different projects.

Table 1
(Continued).

New GA	Chao's GA
New GA uses normalized objective values.	*Earlier GA uses absolute objective values.*
Because different objectives can have different scales, it would be more meaningful to normalize their values before evaluation and comparison.	Unnormalized values used in single objective function.
All objective values (cost, time) are now normalized into the range of {0, 1}	Weighting was not effective for two objective values on two totally different scales.
A composite objective can be used, by summing weighted objective values.	
New GA prefers population diversity.	*Earlier GA prefers population validity.*
Diversity improves the speed and breadth of the search. Perturbation of an invalid solution may create valid solutions. The new GA allows a portion of population to consist of invalid solutions.	Operators emphasize creating valid genes, which decreases diversity. Potential that the program will fixate on a local-optimum (i.e., premature, sub-optimal convergence).
GALIB provides multiple mechanism to create a diverse population; for instance, the mutation operator has "flip", "destruction" and "swap" options to increase the diversity.	
New GA supports multiple-project scheduling.	*Earlier GA only supports single-project scheduling.*
New GA has much higher complexities.	*Earlier GA has lower complexity.*
The worst-case complexity is $N^{(\text{Num\_Employee x Num\_Task})}$, where N is the number of possible time increments of a particular employee, in terms of percentage of work time.	The worst-case complexity is $\text{Num\_Employee}^{\text{Num\_Task}}$ or $\text{Num\_Task}^{\text{Num\_Employee}}$
Partially compensated for by improved memory access speed and diversity.	

2. Genetic algorithms

2.1. The concept of genetic algorithms

Genetic algorithms mimic natural evolution, by acting on a population to favor the creation of new individuals that 'perform' better than their predecessors, as evaluated using some criteria, such as an objective function. At any given generation (that is, population), the algorithm has a pool of trial solutions. A population can consist of from as low as 20 to several hundred individuals. These individuals compete for an opportunity to reproduce. Reproduction will propagate some of an individual's characteristics (traits) into the next generation. Candidates for reproduction are chosen probabilistically, but in a manner that should favor individuals whose offspring will perform well.

Reproduction is a critical step in exploring the solution space because it creates new candidate solutions. For example, reproduction can consist of two substeps: *se-*

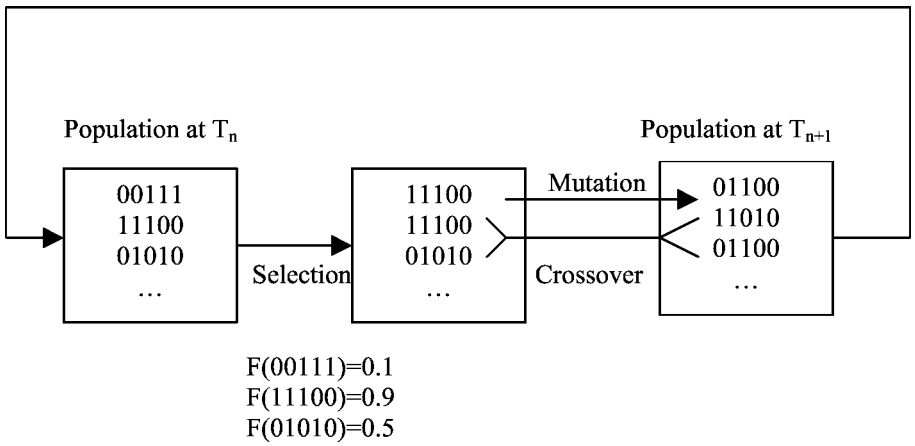


Figure 1. A simplified GA example.

lection of the parents and *crossover* (sometimes combined with *mutation*), which is the construction of a child solution from components of the parent solutions. The selection process should give preference to individuals with better performance. A selection algorithm that gives little weight to performance will tend to search widely but usually will not converge quickly. On the other hand, an algorithm that overemphasizes performance as a selection criterion tends to converge quickly but to a suboptimal solution.

The specific mechanisms used in the crossover step depend on the problem and the internal representation chosen. Finally, to further expand the search GA implementations incorporate a low-probability random process called *mutation*. Mutation acts to randomly perturb some of the solutions in the population. In the absence of mutation, no child could ever acquire parameter value that was not already present in the population.

2.2. The operation of a genetic algorithm

Figure 1 illustrates the operations performed by genetic algorithms. A population of three individuals is shown as binary strings. Each is assigned a fitness value by the function F. On the basis of these fitness values; the selection phase eliminates the worst case 00111 and replaces it by the best individual 11100. After selection, the genetic operators are applied probabilistically. The first individual has its first bit *mutated* from a '1' to '0', and *crossover* combines the other two individuals into two new ones. The resulting population is shown in the box labeled T_{n+1}.

Typically a genetic algorithm has no obvious stopping criterion. Often, the number of generations is used as the stopping criteria. Genetic algorithms have been the subject of extensive research since their creation in 1975. As stated by Forrest, "the researches on genetic algorithm have abstracted out much of the richness of biology, and more elaborate representation techniques can be expected . . ." [Forrest 1993].

2.3. GAlib: A C++ *library of genetic algorithm components*

A variety of researchers and practictionares have implemented genetic algorithms. The implementations available include GA_{UCSD} [Schraudolph 1992], GALOPPS [Goodman 1996], IlliGAL [Knjazew 2000], and GAlib [Wall 1996]. As the objective of the current research was to apply genetic algorithms to the problems of project scheduling an extensive search of existing GA packages was conducted. We decided to adopt the GA library in C++ created by Wall [Wall 1996] (available as a free download from `http://lancet.mit.edu/galib-2.4`). Wall's *GAlib* provides rich types of Genomes and Operators. Each type can be customized to meet more complicated requirements, for instance, deterministic crowding, traveling salesman, DeJong, and Royal Road problems. Also, new genetic algorithms can be derived from base genetic algorithms class in the library. We hereby give a very brief introduction to *GAlib* for the readers who are not familiar with this work.

2.3.1. Four classes of genetic algorithms

- *Simple GA* – Uses non-overlapping populations and optional elitism. For each generation the algorithm creates an entirely new population of individuals.
- *Steady-state GA* – Uses overlapping population. User can specify how much of the population should be replaced in each generation.
- *Incremental GA* – Allows user-defined replacement methods for the integration of the new generation into the population.
- *Deme genetic algorithm* – Allows for the evolution of multiple populations in parallel using a steady-state algorithm. The algorithm migrates some of the individuals from each population to one of the other populations.

In addition to four major classes of genetic algorithms, GAlib also supports a variety of representations of Genome, such as lists, binary strings and arrays. Choosing a representation that is minimal and sufficiently expressive is critical to solving any optimization problem. The right representation of the genome should be able to represent any point in the search space. Although it is attractive to use a genome containing "extra" gene information, the complexity of the algorithms will increase and thus hinder the performance.

Three primary operators can be applied to genomes: initialization, mutation and crossover. In addition to these three primary operators, objective function that is used to evaluate the fitness of genomes, together with a comparison operator, must be supplied. The comparator is used to determine how one genome is different from another. Conceptually, it serves as a *distance function*.

2.3.2. The role of the objective function

The objective function provides a measure of how good an individual is but can be considered for either an individual in isolation or within the context of the entire population. The objective score is a measure used to evaluate the performance of the genome. The

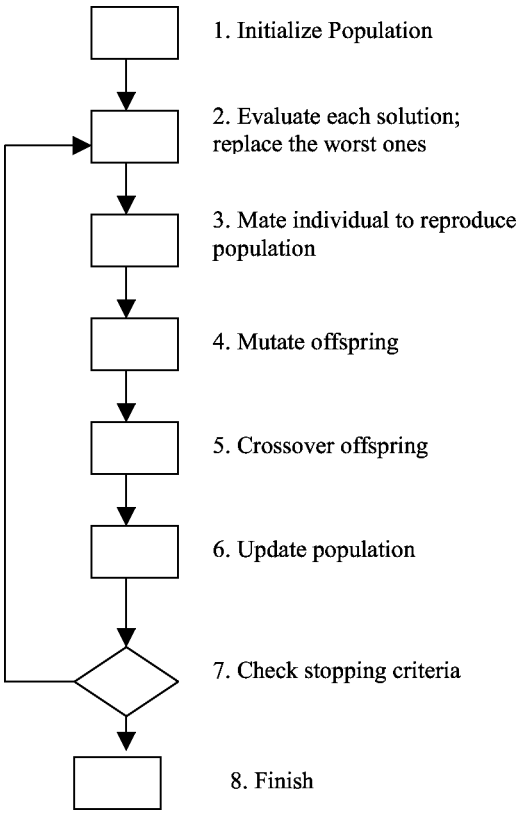


Figure 2. Flow chart of GAlib (from [Wall 1996]).

fitness score is computed from the objective score using a scaling strategy, such as those introduced by Goldberg [1989]. Figure 2 shows the operation of GAlib and, in particular, shows the use of the objective function in the second step.

3. Project scheduling

3.1. Definition of the scheduling problem

There are a variety of representations that can be used when scheduling projects including PERT (program evaluation and review technique), and CPM (critical path method) methodologies. Since all of these methods are attempting to solve a single problem they share a number of features. These include:

I. A technique for describing tasks and their requirements.

II. A method of specifying the relationships between the tasks.

III. A description of the resources available to perform the tasks.

IV. A set of objectives that will be used to evaluate the schedule.

V. A specification of any constraints that the project must satisfy.

A schedule is a specific, time-phased assignment of resources to tasks that satisfies the requirements and constraints. The goal of the project manager is to produce a schedule that *optimizes* the objectives [Davis 1971]. In general, if only precedence relationships constraint the schedule, it would require only polynomial-time computation, as commonly employed in most project management tools.

This paper treats project scheduling as a resource allocation or assignment function by presenting a time schedule, which belongs to a class of NP-hard problems, also known as the resource-constrained project scheduling problem [Blazewicz 1983]. That is, as "Who does what and when", where 'who' stands for employees, 'what' stands for tasks, and 'when' means the time schedule.

A project is best represented as a Task Precedence Graph (TPG). A TPG is an acyclic directed graph consisting of a set of tasks $V = \{T_1, T_2, \ldots, T_n\}$ and a set of precedence relationships $P = \{(P_{ij});\ i \neq j,\ 1 \leqslant i \leqslant n,\ 1 \leqslant j \leqslant n\}$, where $P_{ij} = 1$ if task i must be first completed, with no other intervening tasks, before task j can start, and zero if not. Associated with each task T_k are the estimated effort and required skills. We assume that such effort estimation can be obtained from any known estimation method such as COCOMO [Boehm 1981].

The description of a task should include what skills are needed to complete it, along with what level of effort and staffing is required, and any constraints the task is subject to. The resources that are required to perform tasks include personnel, usually described by numbers of hours, equipment, and facilities. Performance attributes can be specified for resources. For example, the required skills of personnel can be specified, as can the throughput of computing resources. The relationship between resources and tasks is a many-to-many one. That is, many resources can be assigned to multiple concurrent tasks.

The assignment of resources to tasks is usually subject to rules and constraints. In the current research prototype the following simplified rules for assigning personnel to tasks. First, the time required to perform a task is inversely proportional to the number of resources (primarily appropriately skilled people) assigned to it. It is well known that this is usually not the case. In addition, this assumption serves to increase the span of the search space and slows down the algorithm. Likewise, the labor costs were assumed to be constant for a given individual. Thus, all personnel are compensated for all overtime at their normal rate. In some companies no overtime is paid to some employees, while others receive a premium. Finally, no penalty was incurred for under utilization of personnel. Some companies strive for 100% utilization, while others deliberately reserve a nominal amount (say 10%) for administrative functions.

Improving the realism of the task assignment and labor cost functions of the algorithm were not seen as essential at this time. We will discuss how to improve the realism of the system in these areas later in this paper.

Finally, the assignment of personnel to tasks was constrained to those areas that were essential to the prototype. Thus the percent of an employee's labor that can be committed to any give task was constrained to a discrete set of values. This commitment quota was constrained to the set {0%, 25%, 50%, 75%, 100%} for each individual's effort that could be assigned to any single task. Thus if person P_i is assigned a task T_j with 0.25 quota they will expend 25% of the their normal, 40-hour work week working on that task while it is active. Increasing the granularity of the allowable values (say to the set {0%, 10%, 20%, . . . , 90%, 100%}) is not a structural change to the algorithm but does degrade performance.

In addition, a constraint was applied to the overtime that an employee could work. In our experiments this limit was set to 75%. Thus an employee committed to work on task A 100% is also allowed to work on other tasks with a maximum of 75% commitment quota. For protracted periods this would be excessive but the limit can easily be adjusted downward or even made dependent on the duration of the overtime. Project management commonly makes such adjustments.

Lastly, the objective function must be specified. Objective functions must satisfy the following conditions:

- The objective function must depend upon the entire schedule. That is, the objective must be dependent upon every task.
- The ultimate goal is to minimize or maximize the objective function.
- Objective functions may be composite. That is, they can be dependent upon component objectives such as cost, schedule, and overtime commitment or any other properties of the network. However, the objective function must return a scalar value.
- Any component objectives must be normalized and weighted. Since different component objectives have different units and scales, it is necessary to normalize each objective such that they all have comparability. Also different component objectives can have different weights, or priorities.

In the research reported in this paper, four component objectives were considered.

Validity of job assignments (Validity). If the job assignments of a schedule are valid they must satisfy the following constraints [Chao 1995]: the precedence relations among tasks must be observed; the employee must posses the skills required for each task; all the skill needs of the tasks must be satisfied; all the tasks must appear in the schedule (*completeness*). Validity is usually scored on a 0/1 basis, 0 if the assignments are invalid, 1 if they are valid.

Minimum level of overtime (OverLoad). The amount of time worked beyond the individual over time limits is summed over all employees, and it is treated as a global objective for a project. This was done to further reduce the amount of overtime worked by employees. The alternative would be to impose a cost penalty for overtime, such as an overtime premium.

Minimum cost (CostMoney). The total labor cost of performing the project, computed using the labor rates of each resource and the hours applied to the tasks.

Minimum of time span (CostTime). The total time span required to finish the project, from the start of the first task until the end of the last, can be used as a component objective.

The simplest composite objective value is the summation of weighted component objective values. Since genetic algorithms search for the solution with the highest fitness values, the composite objective will be maximized. The simplest form of a composite objective function is:

$$\begin{aligned}&\text{Composite objective function}\\&\quad = \text{Validity} \cdot (W_1/\text{OverLoad} + W_2/\text{CostMoney} + W_3/\text{CostTime})\end{aligned} \tag{1}$$

In this form the Validity is used to impose a large penalty in the event the schedule assignments are incomplete. The alternative would be to reject the genome out of hand.

Often the objectives conflict with each other. For example, it may be possible to shorten a project's duration by assigning more experienced employee to critical tasks, but the cost will usually increase. As more components are encompassed by the objective function the possibility for conflict between the components increases. This can result in extended execution time of the genetic algorithm. However, it is more realistic, as making tradeoffs between competing objectives is a common project management task.

3.2. Search by genetic algorithm

Our approach to project scheduling, extended from that of [Chang and Christensen 1999], requires that the user describe the problem in the following form:

I. A task precedence graph, TPG $= (V, E)$,

II. An employee database D_{emp} including skills and salary,

III. An objective function.

The parameters describing the GA search are:

$$\begin{aligned}N_{\text{pop}} &= \text{size of population},\\ \text{Prob}_{\text{cross}} &= \text{probability of crossover},\\ \text{Prob}_{\text{mut}} &= \text{probability of mutation},\\ \text{Prob}_{\text{replace}} &= \text{probability of replacement}.\end{aligned}$$

The goal is to produce near optimal (in the sense of the specified objective function) schedule $S_{\text{near-opt}}$.

Consider the case of two employees A and B with the same (sufficient to all tasks) skills, with tasks 1, 2, 3, 4 having precedence shown in figure 3.

One possible solution to the problem of scheduling this simple project would be:

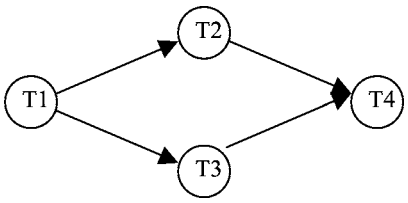


Figure 3. An example for task precedence graph.

"Employee A can be assigned to do task 1 with 50% commitment and task 2 with 25% commitment. Employee B does task 2 with 75% commitment and task 3 with 50% commitment."

The intuitive representation for the above example is a two dimensional array with Employee enumerated along the rows, and the tasks along the columns:

	T1	T2	T3	T4
Emp. 1	0.5	0.25	0	0.25
Emp. 2	0	0.75	0.5	0.25

GAlib provides 2DArrayGenome 2DArrayAlleleGenome classes, simplifying the task of implementing a GA search for problems of this type. Both individuals are assigned to task 4 at the 25% level.

4. The GA operators

Our algorithm used the default genetic operators provided by GAlib:

Genetic operators	GAlib methods
Initialization	GA2DarrayAlleleGenome⟨⟩::UniformInitializer
Comparison	GA2DarrayGenome⟨⟩::ElementComparator
Mutation	GA2DarrayAlleleGenome⟨⟩::FlipMutator
Crossover	GA2DarrayGenome⟨⟩::OnePointCrossover

4.1. Initialization

Genetic algorithms are an example of a search method with weak memory. Indeed, the only memory is the population itself. Initialization may have little effect on the performance of the solution provided that the evolution history is long enough. As a result we used the UniformInitializer, which initializes all individuals in the population with uniform values. This results in a population that initially occupies only a small region of the total solution space but, if the GA propagation parameters are selected

at all well, it will rapidly produce populations containing individuals with high figures of merit (as measured by the objective function). The properties of individuals with high figures of merit will be propagated throughout the whole population very quickly. Customization of the initializer was not necessary in our experiments.

For the example shown in figure 3, the UniformInitializer assigns each individual the same value, for example, 0.25.

	T1	T2	T3	T4
Emp. 1	0.25	0.25	0.25	0.25
Emp. 2	0.25	0.25	0.25	0.25

4.2. Comparator

The comparator is used to determine how one genome is different from another, as measured in the space of all solutions. Thus, the comparison operation generates a distance measure between two points in the search space as an intrinsic property of the space itself. Thus, given two solutions A and B, one definition of the distance between A and B is given by the equation

$$d = \sqrt{\sum_{i=1}^{n}\sum_{j=1}^{m}(a_{ij} - b_{ij})^2}, \tag{2}$$

where a_{ij} – ith row and jth column of genome A, b_{ij} – ith row and jth column of genome B, m – number of tasks, n – number of employees.

For example, given two solutions A and B for project scheduling:

A

	T1	T2	T3	T4
Emp. 1	0.25	0.25	0.25	0.25
Emp. 2	0.25	0.25	0.25	0.25

B

	T1	T2	T3	T4
Emp. 1	0	1	0.5	0
Emp. 2	0.5	0	0.25	0.5

$$d = \big((0.25-0)^2 + (0.25-1)^2 + (0.25-0.5)^2 + (0.25-0)^2 + (0.25-0.5)^2 + (0.25-0)^2 + (0.25-0.25)^2 + (0.25-0.5)^2\big)^{1/2}$$

The comparator thus provides a measure of how diverse a population is, as distinct from how desirable it is. The objective function is used to evaluate that property.

4.3. Crossover

The crossover operator mimics the way in which bisexual reproduction passes along each parent's "good genes" to the next generation. Figure 4, reproduced from [Wall 1996], illustrates how a crossover operator operates on 2D array genomes.

In this example the two parent solutions create two new "offspring" solutions by combining their "genes" through a one-point crossover operation. Crossover uses both inheritance and variation to improve the performance of the population (as measured by the objective function) while retaining a diverse (as measured by the comparison function) population.

4.4. Mutation

Following the crossover operation the offspring may (at a rate determined by Prob_{mut}) be mutated by the mutation operator. This is done by modifying one of the values used to encode the offspring. Similar to random mutations in the biological world, this function is intended to preserve the diversity of the population, thereby expanding the search space into regions that may contain better solutions.

GAlib provides three types of mutation operators, destructive, element flip and element swap shown in figure 5, again, reproduced from [Wall 1996]. Each can be assigned their own probabilities.

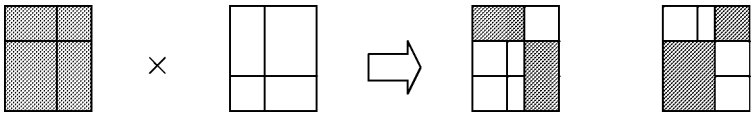


Figure 4. One-point crossover operator for 2D array genome (from [Wall 1996]).

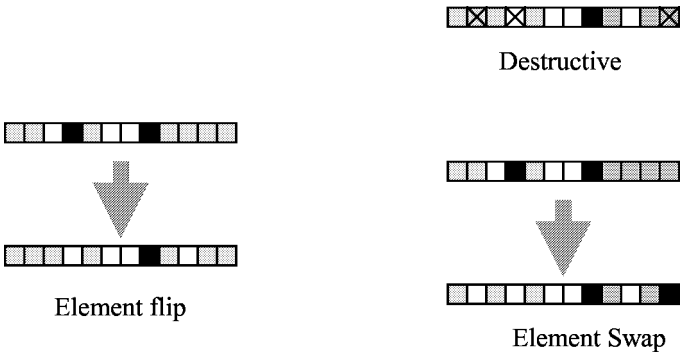


Figure 5. Three types of mutation operators for array genome.

5. Details of the objective function

The objective function is used to determine the fitness value for each solution of the problem. The fitness is, in turn, used during the selection process to determine the pool of solutions used to create the next generation. In our approach to the project-scheduling problem, it is performed in two stages, each based on the schedule the genome represents. The first stage evaluates the extent to which the genome satisfies the constraints, the second stage evaluates the schedule performance of the genome. The 2D array genome contains all the data needed to directly calculate the value of the objective function.

Of the two stages of evaluation, the constraint satisfaction is the easier one to perform. Two validations are performed sequentially, task completeness and resource skill match.

Task completeness. At least one employee must be assigned to work on each task T_j. In other words the summation of any column of the employee-to-task assignment array S must be greater than zero.

$$\sum_{i=1}^{n} S_{ij} > 0. \tag{3}$$

Resource skill match. The combined skill set of the employees assigned to task i (SK_EMP(i)) should include all the skills required by the task (SK_TASK(i)). That is,

$$\text{SK\_EMP}(i) \supseteq \text{SK\_TASK}(i). \tag{4}$$

In the second stage of the objective function evaluation process the cost, duration, and overtime percentages are evaluated and combined. The cost is computed for each task by summing the cost contribution of each individual to the task. The total cost is then computed by summing the costs of all the

$$\text{CostMoney}_{\text{norm}} = \frac{\text{CostMoney}}{\text{CostMoney}_{\text{max}}} \tag{5}$$

tasks. The duration is computed by finding the longest (in calendar time) path through the precedence graph. These components are computed for each member of the current population and then normalized using the maximum values found for that population, as shown in equations (5) and (6). Finally, the two components are combined, using user-selected weights (W_{time} and W_{money}), as shown in equation (7).

$$\text{CostTime}_{\text{Norm}} = \frac{\text{CostTime}}{\text{CostTime}_{\text{max}}}, \tag{6}$$

$$\text{FF} = \frac{W_{\text{time}}}{\text{CostTime}_{\text{norm}}} + \frac{W_{\text{money}}}{\text{CostMoney}_{\text{norm}}}. \tag{7}$$

The final component of the objective function is the amount of effort expended beyond the over time limits. Each employee can be separately assigned a limit to the amount

of work load they can be assigned. The total effort expended beyond the limits is then totaled across all tasks and all employees. In our work any over-limit is considered a reason for rejection of the schedule. However, the metric could be normalized and combined with the other components, as shown in equation (1).

6. The test problems and results

The project used in the tests consists of 18 tasks. The TPG for the project is shown in figure 6. There are 10 employees available to work on this project. The skill proficiency (0–5) and salary for each employee is shown in table 2.

We evaluated the algorithms using a variety of objective functions, focusing on the time required to find satisfactory solutions. GA performance is tested using both single objective and composite objectives.

Test 1. "*Find the schedule that utilizes all employees with full loading and no restriction of skill match, money cost and overtime working*."

Test 1 is definitely not realistic, it was used only for evaluation purposes. It has one expected solution – a 2D array with full of '1's, which means that every employee should do each of the task with 100% commitment. Figure 7 shows its evolution history. This curve shows that the job loading increases with each generation, and eventually

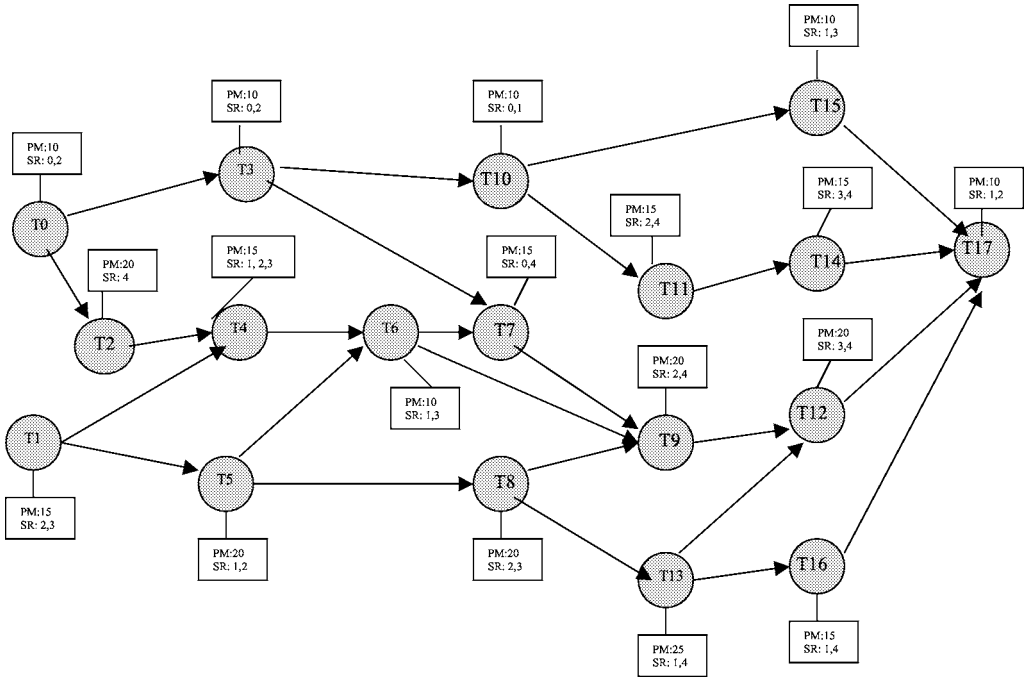


Figure 6. Task precedence graph of the experiment project (PM: person month, SR: skill required).

Table 2
The employee table.

Emp. ID	Skill 0	Skill 1	Skill 2	Skill 3	Skill 4	Salary
P1	–	–	2	4	–	5000.00
P2	1	–	2	–	–	4000.00
P3	–	4	–	–	–	3000.00
P4	–	–	2	4	3	5000.00
P5	–	3	–	–	–	3000.00
P6	–	3	2	4	–	6000.00
P7	–	2	2	–	–	6000.00
P8	–	3	3	4	–	5000.00
P9	–	1	2	–	–	8000.00
P10	3	2	–	–	3	9000.00

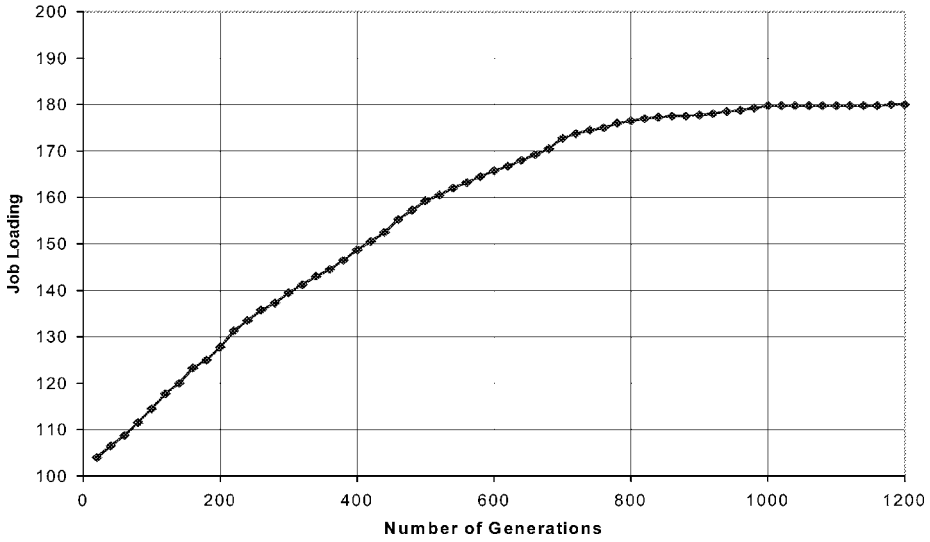


Figure 7. The evolutionary history for test 1 with job loading as objective.

reaches its maximum level of 180. Recall that the solution array is 10×18, so 180 is the summation of all the commitments in the array. The optimal solution for test 1 is shown in table 4.

Test 2. "*Find the schedule that has shortest execution time, assuming no restriction of skill match, money cost and overtime working*."

According to the constraints used, the time spent to fulfill a task is inversely proportional to the number of employees assigned to it. The expected result is a schedule that assigns as many employees as possible to each task with full commitment quotas. Recall that the commitment quotas take the form of {0%, 25%, 50%, 75%, 100%}. This test was used to evaluate the correctness of the implementation.

Table 3
TPG matrix represents the precedence relations between tasks.

T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13	T14	T15	T16	T17
0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	1	0	0	1	0	0	0	0	0	0	0
0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 4
The optimum solution for test 1.

	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13	T14	T15	T16	T17
P1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
P2	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
P3	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
P4	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
P5	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
P6	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
P7	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
P8	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
P9	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
P10	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Figure 8 shows the genetic evolutionary history to find the solution for test 2. Each generation produces a population that provides better and better performance. Table 5 shows the optimal solution when the generation number is over 500. The solution exhibits the following expected behaviors.

(a) The solution array is dominated by full loading commitments. It achieves the stated objective that more personnel can complete a task more quickly.

(b) The matrix also contains a small number of assignments with partial commitment quota. The reason is that the schedule duration is only determined by the critical path of the TPG. For example, the following graph shows a project consisting of

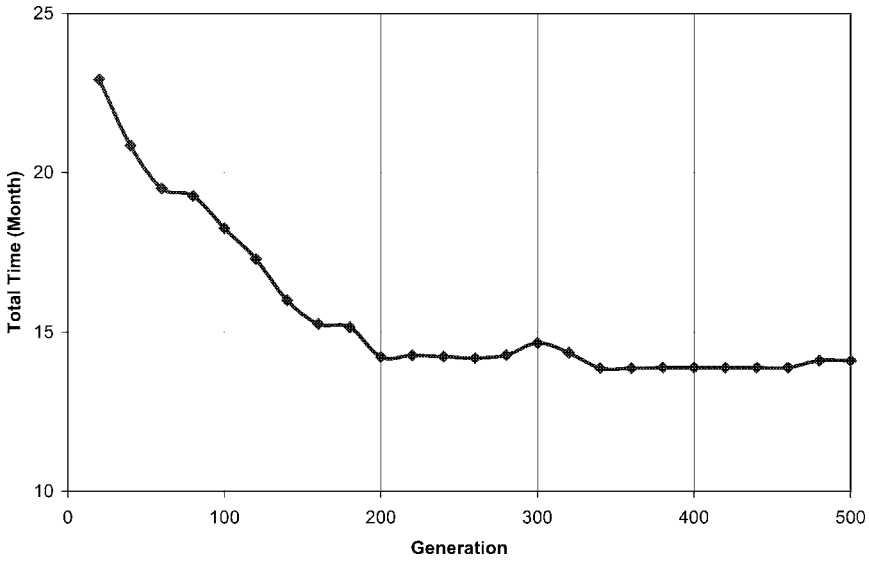
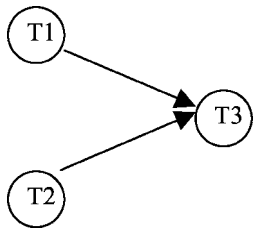


Figure 8. The genetic evolutionary history of test 2 with time cost as primary objective.

Table 5
The optimum solution for test 2.

	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13	T14	T15	T16	T17
P1	1	0.5	1	1	1	1	1	1	0.75	1	1	1	1	0.75	1	1	1	1
P2	1	1	1	1	1	0.25	0.75	1	1	1	1	1	1	0.75	1	1	1	1
P3	1	0.5	1	1	1	1	1	1	1	1	0.5	1	1	1	1	1	1	1
P4	1	1	1	1	1	1	1	1	1	1	1	1	1	0.75	1	1	1	1
P5	1	1	1	1	1	1	1	1	0.5	1	0.5	1	1	1	1	1	1	1
P6	1	0.5	1	1	1	0.75	1	1	0.75	0.75	0.5	1	1	0.5	1	1	1	1
P7	1	1	1	1	1	0.25	1	1	0.5	1	0.75	1	1	0.75	1	1	1	1
P8	1	0.75	1	1	1	0.75	1	1	0.5	1	1	1	1	1	1	1	1	1
P9	1	1	1	1	1	1	1	1	0.75	1	0.75	1	1	1	1	1	1	1
P10	1	0.25	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

three tasks. If the minimum execution time for T1 is 20 months, T2 is 5 months, T3 is 10 months, the minimum time cost to full T1, T2, T3 is $20 + 10 = 30$ months. The critical path is from T1 to T3. As long as the assignment for T1 and T3 are optimized, T2 need not to be fully loaded.



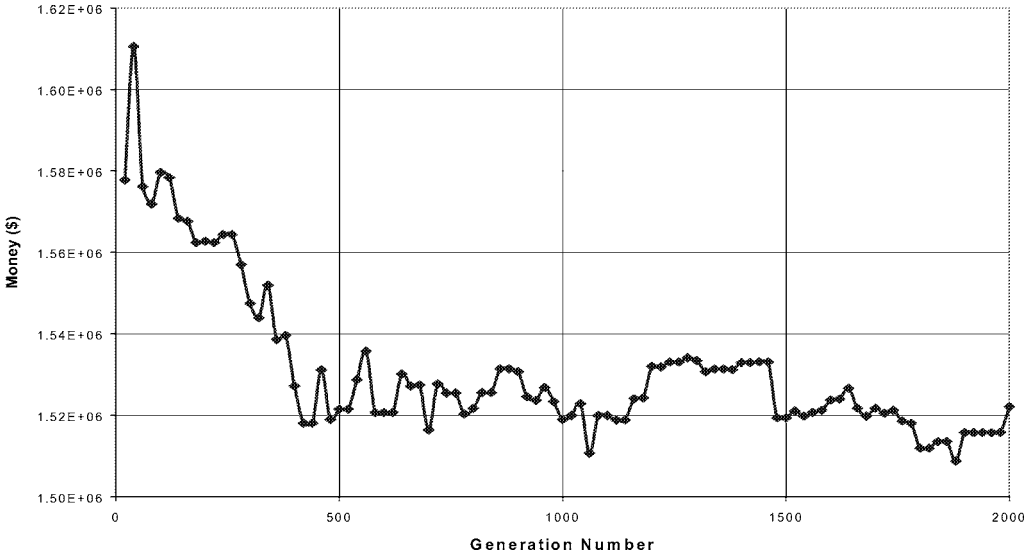


Figure 9. The genetic evolutionary history of test 3 with money cost as primary objective.

Table 6
The near-optimum solution for test 3 with lowest money cost.

	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13	T14	T15	T16	T17	T18
P1	1	0.5	0.5	0	0	0.25	0.25	0.75	0	0	0.5	0.75	0.75	0.75	0.5	0	0.25	1
P2	1	0.5	0.75	1	0.5	0.25	1	0.5	0	0.75	1	0.75	0.75	0.75	1	0.75	0.5	1
P3	0.75	0.75	0.75	1	0.75	1	1	1	0.75	0.75	0.25	1	1	1	1	1	0.5	0.75
P4	1	0.5	0.5	0.5	0	0.25	0.5	0.75	0.25	0.75	1	0.25	0	0	1	0.25	0.5	0.75
P5	0.75	1	0.5	1	0.75	0.75	0.75	1	1	0.75	1	1	1	1	0.75	0.75	1	0.5
P6	0.25	0.25	0.25	0	0.25	0	0.5	0.75	0	0	0.25	0.25	0	0	0.5	0	0	0
P7	0	0.25	0.5	0.25	0	0	0	0	0	0.25	0.5	0	0	0.25	0.25	0	0.75	0.75
P8	0.75	0.25	0	0.5	0	0.25	0.25	0.75	0	0	0.5	0.25	0.75	0	0	0.5	0.5	0.5
P9	0	0	0	0	0	0	0	0.25	0	0	0.5	0	0	0	0	0	0	0
P10	0.25	0	0	0	0	0	0.25	0	0	0	0.25	0	0.25	0.25	0.25	0	0.25	0.25

Test 3. "*Find a valid schedule that has lowest money cost, irrespective of time cost and overtime working*."

Compared to tests 1 and 2, the schedule produced by test 3 is not predictable. According to our algorithm described above, the money cost needed to complete the project is based on the execution time for each task, the number of employees assigned to that task and their salaries.

Figure 9 shows how genetic algorithm "evolves" an optimum solution for test 3. The evolutionary curve clearly demonstrates a pattern of long term cost reductions with short-term fluctuations. Table 6 shows the near-optimum solution.

Test 4. "*Find the optimum valid schedule, satisfying a composite objective function, including money cost, time cost and overt time limits for loading*."

The objective for test 4 is composite. Three objectives: time, money, overloading are considered at different levels. Since these individual objectives use different scales, they are normalized to the range of 0–1. The definition of the composite objective is shown in equation (1). Figure 10 shows the evolution of the solution for test 4. The evolution process reached its optimum level after around 6,000 generations. Figures 11 and 12 show the behavior of the individual components of money and time.

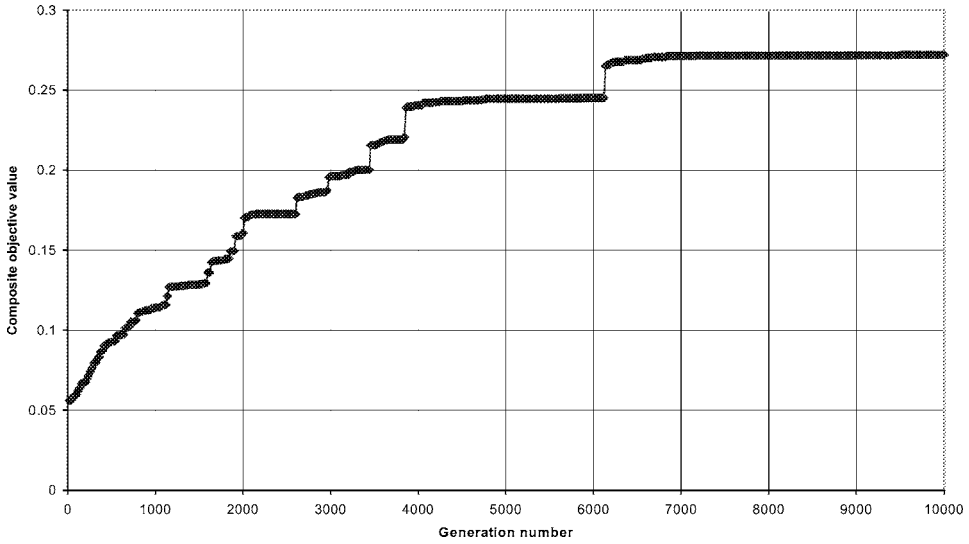


Figure 10. The genetic evolutionary history of test 4 with composite objective.

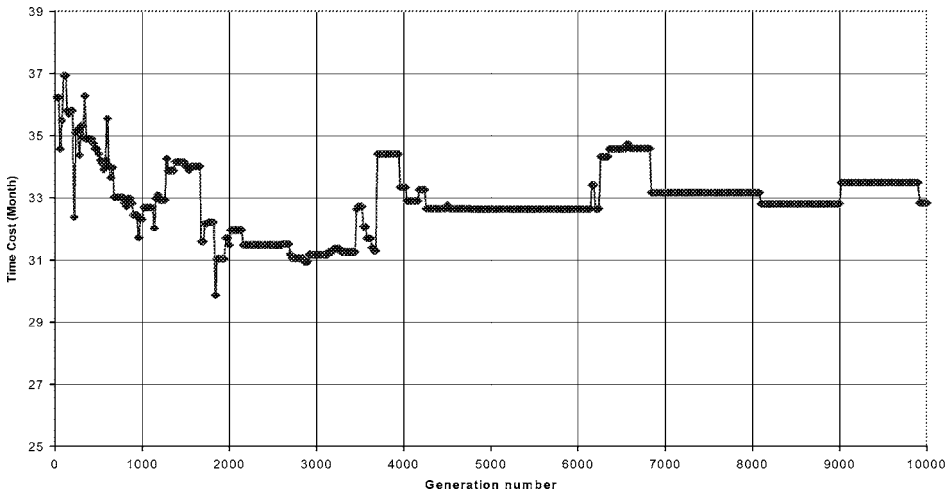


Figure 11. The genetic evolutionary history for time cost of test 4.

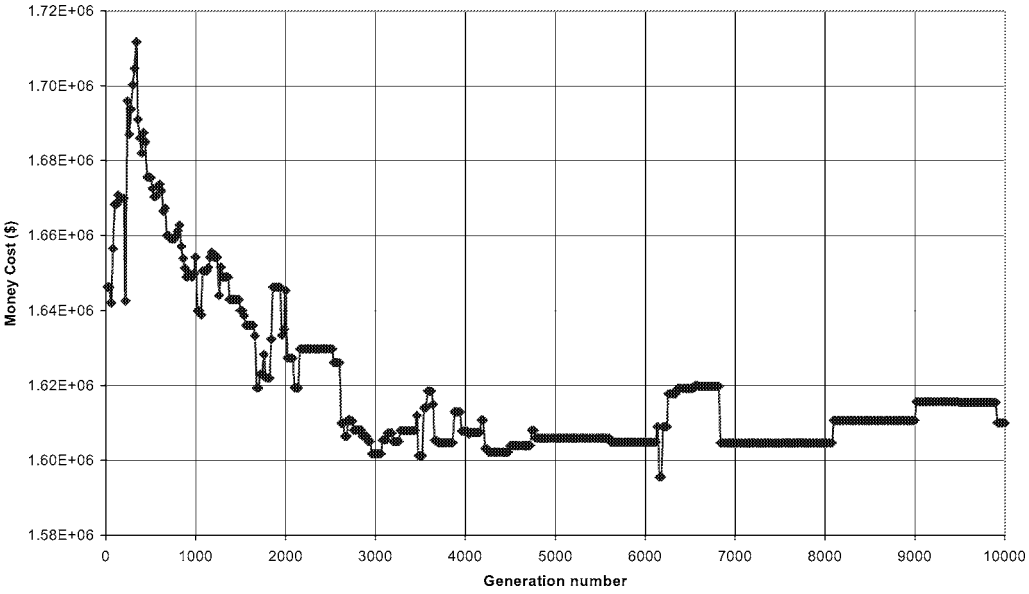


Figure 12. The genetic evolutionary history for money cost of test 4.

Table 7
The GA solution that considers real time overloading.

	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13	T14	T15	T16	T17	T18
P1	0	1	0.25	0.25	0.5	1	0	0.5	0.5	1	0	0.75	1	0	0.75	0.25	1	0.5
P2	0	1	0.25	0	0.25	0.5	0.25	0.75	0.75	1	0.5	0	1	0.75	0.25	0	1	0.75
P3	0.5	0.5	0.25	0	0	0.5	0	0	0.5	1	0	0.75	1	0.75	0.75	0.25	0.75	0
P4	0.75	0.5	0	0.5	0	0.75	0.50	1	0	1	0.75	1	1	0.25	0.5	0.25	0	1
P5	0.5	0.5	0.25	0	0	0.5	0.75	0	0.5	1	0	0.75	1	0.75	0.75	0.25	0.75	0
P6	0.5	0.5	0	0.25	0	1	0	0.25	0.75	1	1	1	1	1	0.5	0	0.25	0.25
P7	0.25	0.5	0	0	0.25	0.75	1	0.25	0.75	1	0.5	0.75	1	0.5	0	0	0	0.25
P8	0	1	0.25	0.25	0.5	1	1	0.5	0.5	1	0	0.75	1	0	0.75	0.25	1	0.5
P9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
P10	0	0	0.25	0	0	0	0.25	0	0	0	0	0	0	0.25	0.5	0.25	1	0

Both “Money” and “Time” evolution show decreasing trends but they are not correlated due to heuristic behavior of genetic algorithm and the complex relationship between time cost and money cost. Table 7 shows the optimum schedule for test 4. Notice that, employee 9 isn’t assigned any work, and employee 10 is only lightly loaded. Both of these employees have higher salaries and few applicable skills. The schedule shown by table 7 appears reasonable and also is valid in the sense described earlier.

Finally, figure 13 shows the loading for every employee in test 4. The loading is set as 100%. From the graph we can see, the time over loading is minimized to zero. The limit for loading can be changed as a user input to the algorithm.

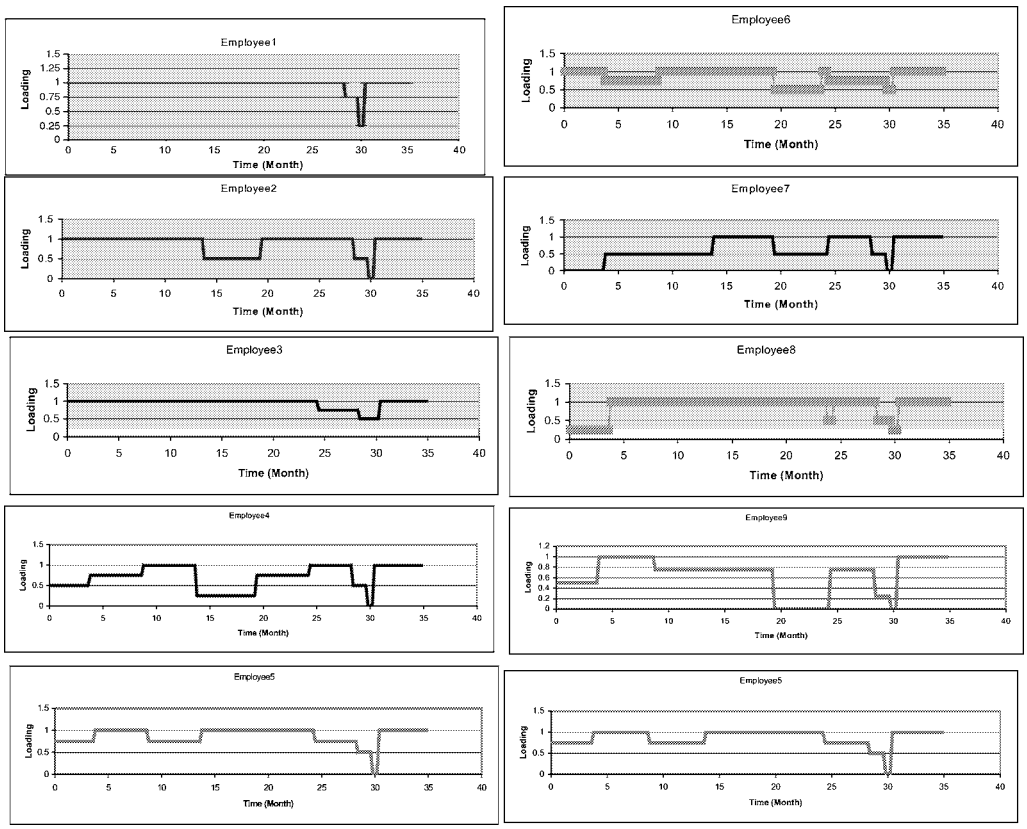


Figure 13. The employee's loading charts. The loading limit is set as 100%.

To illustrate this, test 4 was re-run using various loading limits. In the following test each employee was assigned a different loading limit according to the following table:

Employee	Loading limit
1	1.0
2	1.0
3	1.25
4	1.25
5	1.5
6	1.5
7	1.75
8	1.75
9	2.0
10	2.0

The GA produced a valid schedule for this case, resulting in employee task assignments shown in table 8 and loading charts given in figure 14.

Table 8
The GA solution.

	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13	T14	T15	T16	T17	T18
E1	0.25	0.5	0	0	0.25	0.75	1	1	0.75	1	0.5	0.75	1	0.5	0	0	0	0.25
E2	0	1	0.25	0	0.25	0.5	1	1	0.75	1	0.5	0	1	0.75	0.25	0	1	0.75
E3	0.5	0.5	0.5	0.25	0.25	0.5	0.5	0.75	0.25	1	0.75	1	1	0.5	0.25	0.5	1	1
E4	0.75	0.5	0	0.5	0	0.75	1	1	0	1	0.75	1	1	0.25	0.5	0.25	0	1
E5	0.5	0.5	0.25	0	0	0.5	1	1	0.5	1	0	0.75	1	0.75	0.75	0.25	0.75	0
E6	0.5	0.5	0	0.25	0	1	1	1	0.75	1	1	1	1	1	0.5	0	0.25	0.25
E7	0.5	0.75	0	0	0	1	1	1	0.25	1	0.75	1	1	1	0.25	0.75	0.75	1
E8	0	1	0.25	0.25	0.5	1	1	1	0.5	1	0	0.75	1	0	0.75	0.25	1	0.5
E9	0.75	0.75	0.75	0.25	0.75	0.25	1	1	1	1	0.75	1	1	0	0.5	0.75	0.75	0.5
E10	1	1	0.25	0.75	0	0.75	1	1	1	1	1	1	1	0.5	0.5	0.25	1	0

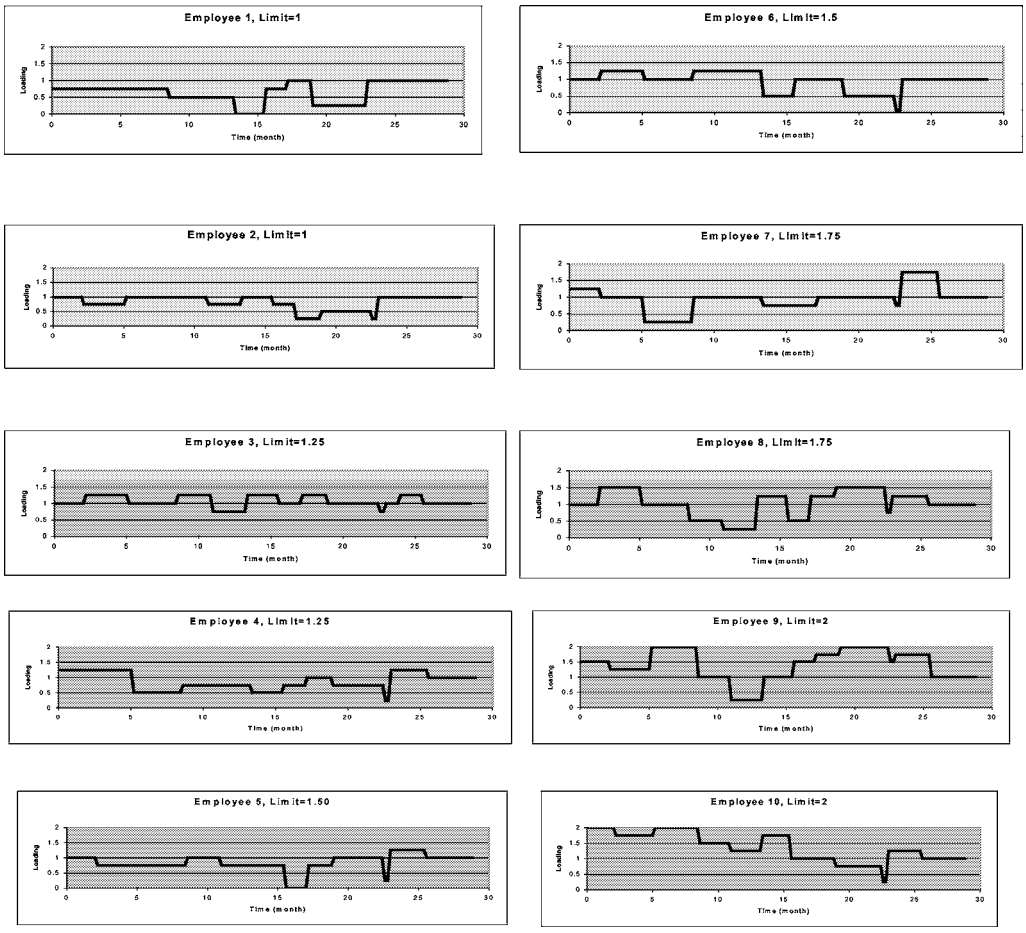


Figure 14. The loading history for employees having different loading limits.

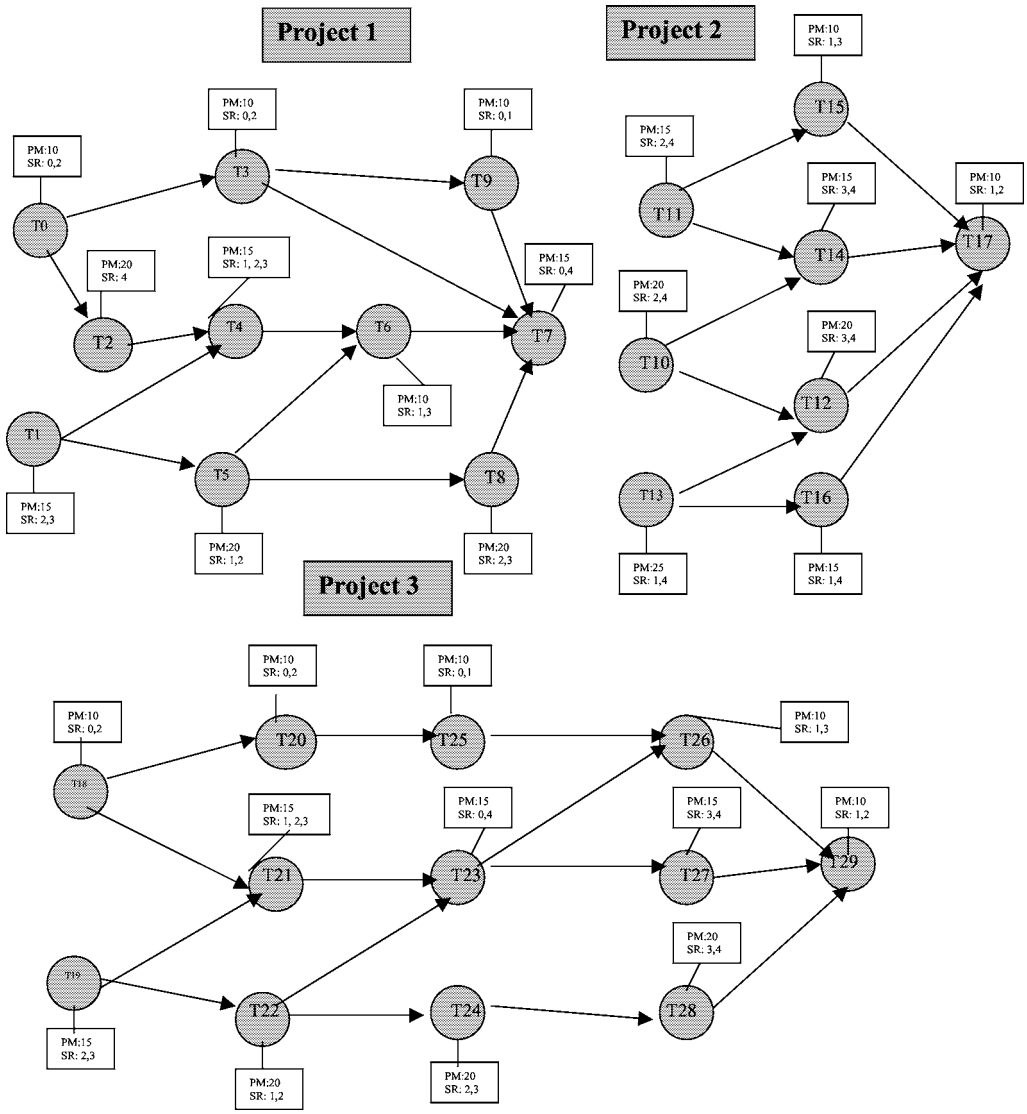


Figure 15. The task precedence graph for multiple projects used in test 5.

Test 5. "*Multiple project scheduling* ($N = 3$) *without loading limit*."

Project scheduling based on 2D-array genetic representation can be easily extended to multiple projects. With our representation, multiple projects do not necessarily have higher complexity. Figure 15 shows the TPG for three projects. Table 9 gives the corresponding solution.

Table 9
The optimum solution for multiple project of test 5.

Project 1										
	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9
P1	0.5	0.5	0.5	0	0	1	0.75	0.25	1	0
P2	0.75	0.75	0.5	0	0	0.75	0	0.75	1	0.25
P3	0.75	1	0.75	0	0	0	0	1	1	0.5
P4	0.5	1	0	0	0	0	0	0.25	0.75	0.5
P5	0.75	1	1	0	0.25	0.5	0	0	0.5	0.5
P6	0.25	1	0.25	1	0.25	0	0	0	0.5	0.5
P7	0	0.25	0	0.25	0.75	0	0	0	0	0.75
P8	0.25	0	0.5	1	1	0.5	0	0	0.5	0
P9	0	0.75	0	1	0	0.5	0	0	0.25	0
P10	0	0	0	0	1	0	1	0	0	0
P11	0	0.75	0	0	0	0	0	0	1	0
P12	0.25	0.75	0	0	1	0.5	0	0.25	0.25	0
P13	0	0.75	0	0.25	1	0	0	0	0	0.75
P14	1	0.75	0.75	1	0.75	0	0	0	0	0.25
P15	0.25	0.25	0.5	1	0.75	0	0.75	1	1	0

Project 2								
	T10	T11	T12	T13	T14	T15	T16	T17
P1	1	0.75	0.25	0.25	0	0.25	0.75	0
P2	0.75	0	0.75	0	0	0.25	0.25	0
P3	1	0	1	1	0.75	1	1	0.75
P4	0	0	0.25	1	0	0.25	0	0
P5	0.5	0	0	1	0.25	0.5	1	0.5
P6	0	0	1	0.25	0.25	0	0.5	0
P7	0	0	1	0.25	0	0	0	0
P8	0.5	0	0.75	0.5	0.5	0	0	0.25
P9	0.5	0	0	0.75	0.25	0	0	0
P10	0	0	1	0.25	0	0	0	0
P11	1	1	0	0	0	0	0	0
P12	0.5	0.75	0	0.75	0.25	0.75	0.5	0
P13	0	0	0	1	1	0.75	0.75	1
P14	1	1	0	1	0.75	0.75	0.75	0.25
P15	1	0.75	1	1	0.75	0.5	0.25	1

7. GA performance analysis

All the results of tests were achieved using a steady-state genetic algorithm. No specific attempt was made to tune the genetic algorithm, all tests were run for a fixed number of generations with reasonable mutation and crossover rates, population size, and replacement rate. Tables 4–9 summarize the parameters used in the genetic algorithm runs.

The genetic algorithm required no modifications to switch between any of these test problems. All of the tests used the same data structures and genetic strategies. Only the objective function was modified for the different tests.

Table 9
(Continued).

Project 3												
	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11
P1	0.25	0.75	0	1	0.75	0	0.25	0.25	0.75	0.75	0.75	1
P2	0.25	0.25	0	0.75	0.25	1	0.5	0	0.25	0	0	0.75
P3	1	1	0.75	0.75	0.75	0	0	0	1	0.5	1	1
P4	0.25	0	0	0.75	0.25	0	0	0.5	0.25	0	0.5	0
P5	0.5	1	0.5	0	0	0	1	0.75	1	0.75	0.75	1
P6	0	0.5	0	0.25	1	0	0.75	0.25	0	0	0	0.75
P7	0	0	0	0	0	1	0	0	0	0.75	0	0.75
P8	0	0	0.25	0	0.5	0.5	0.75	0	0.25	0	0.25	1
P9	0	0	0	0.25	0	0	0	0	0	0	0	0.25
P10	0	0	0	0.5	0	0.25	0	0	0.25	0.75	0	0.75
P11	0	0	0	0.25	0	0.5	0.25	0	0.25	0.75	0.75	0.5
P12	0.75	0.5	0	1	0.5	0.5	0.5	0.5	0.25	0.5	1	1
P13	0.75	0.75	1	1	1	0.75	1	0.25	0	0	0	0.75
P14	0.75	0.75	0.25	0.75	0	1	1	1	1	0.75	1	1
P15	0.5	0.25	0	0	0	0.75	0.75	1	0.25	0.75	0.5	0.75

Table 10
The summery of parameters used in GA tests.

	Genetic algorithm	Number of generation	Mutation probability (%)	Crossover probability (%)	Replacement (%)	Population size
Test 1	Steady-state	1200	0.15	65	15	30
Test 2	Steady-state	500	0.15	65	15	60
Test 3	Steady-state	2000	0.25	90	15	30
Test 4	Steady-state	10000	0.3	75	15	30
Test 5	Steady-state	8000	0.2	65	15	30

7.1. GA versus exhaustive search

To establish a benchmark by which to evaluate performance of the genetic algorithm an exhaustive search approach was also implemented. Obviously, an exhaustive search is computable, as it simply enumerates every possible combination in the solution space, evaluating each and keeping the record for the best solution seen to date. However, the computational complexity of exhaustive search is extremely high. Given the example shown in figure 3, this complexity can be estimated. This example consists of four tasks and two employees. The solution is represented as a 2-dimensional array with height as employee dimension and width as task dimension. Each cell in the array is assigned one value from the set of $\{0, 0.25, 0.5, 0.75, 1\}$. So it has a total number of 5^8 different combinations including invalid cases. For a general case, an $n \times m$ array will have $5^{n \times m}$ combinations. Using a machine with a 1000 MHz CPU, assuming it can enumerate 10 million combinations in one second, the following chart shows the estimation of real computing time with respect of size of the array.

Table 11
The complexity of exhaustive searching with 2D array representation (worst case).

Size of array	Number of combinations	Number of combinations	Computing time (s)	Computing time (year)
4	5^4	625	6.25E−5	1.98E−12
8	5^8	390,625	3.9E−2	1.24E−9
16	5^{16}	1.526E11	152.6E+2	4.84E−4
32	5^{32}	2.328E22	2.32E+15	7.4E+7
64	5^{64}	5.421E44	5.42E+37	1.72E+30
128	5^{128}	2.939E89	2.94E+82	9.32E+75

Table 12
Experiment results from exhaustive search and GA-Scheduling algorithm.

Proj. #	# Tasks	# Emp.	Array size	Num. comb.	Time (ES)	Time (GA) (seconds)	#pop/#Conv./#Gen.
(1)	3	2	6	5^6	0.5 s	0.5	40/100/1000
(2)	3	3	9	5^9	17.2 s	0.61	
(2)	4	4	16	5^{16}	7.25 h	0.98	40/330/1000
(3)	8	4	32	5^{32}	N/A	1.4	40/400/1000
(4)	8	8	64	5^{64}	N/A	2.8	40/560/1000
(5)	10	10	100	5^{100}	N/A	4.2	40/470/1000
(6)	15	10	150	5^{150}	N/A	6.3	40/580/1000
(7)	20	10	200	5^{200}	N/A	14.0	40/780/1000
(8)	20	15	300	5^{300}	N/A	18.2	40/778/1000
(9)	20	20	400	5^{400}	N/A	21.6	40/890/1000
(10)	30	20	600	5^{600}	N/A	22.4	40/760/1000
(11)	30	25	750	5^{750}	N/A	24.5	40/810/1000
(12)	30	30	900	5^{900}	N/A	27.7	40/860/1000
(13)	40	30	1200	5^{1200}	N/A	35.0	40/960/1000
(14)	50	30	1500	5^{1500}	N/A	52.5	40/860/1000

The exponential complexity of exhaustive search makes it impossible to find optimal solutions for projects of any realistic size or complexity.

In order to contrast the performance of exhaustive search and GA approaches an experiment was performed on a Pentium-based, 300 MHz-class computer.

In the experiment we randomly created a set of experimental projects. Each project has different precedence relationships, different constraints for each task, and various skills for the programmers. In each project, we use the exhaustive search to find the optimal solution, and then the GA-Scheduling was applied to the same project to find the optimal (or near-optimal) solution. Table 12 summarizes the experimental results from exhaustive search and genetic algorithm.

As was estimated in table 11, the computational complexity is extremely high for exhaustive search.

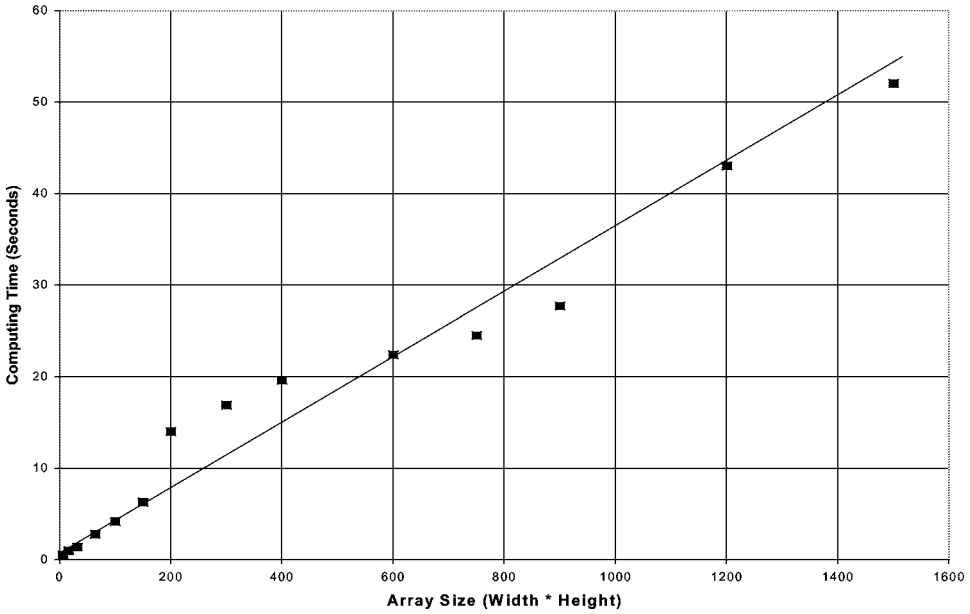


Figure 16. The time expense for GA with respect to the size of gene array.

Table 13
The description of three different groups of projects.

Group number	Number of projects	Number of tasks	Number of employees	Combination
Group (1)	10	10	10	5^{100}
Group (2)	10	30	15	5^{450}
Group (3)	10	50	30	5^{1500}

Figure 16 shows that the complexity of GA is not proportional to the number of combinations, which is $5^{\text{array size}}$, but rather grows linear with the size of the 2D array. This is because of the fixed population size used in each generation.

7.2. Parameter tuning

In order to investigate the impact of different parameters, including converge numbers, crossover probability, mutation probability and replacement probability, on the performance of GA-Scheduling algorithm, we conducted a comprehensive experiment on a number of projects. Test data for 30 projects with different precedence relationships, different constraints for each task, and various skills for the employees were randomly generated. The projects are divided into three different groups as shown in table 13 according to the computational complexity.

Table 14
The range of different parameters.

Parameters	Parameter's range
Population number	10, 20, ..., 100
Generation number	100, 200, ..., 1000
Crossover probability	10%, 20%, ..., 100%
Mutation probability	10%, 20%, ..., 100%

Table 15
The relationship between generation number and statistical behavior of GA.

	Group 1			Group 2			Group 3		
Gen. num.	Min	Mean	Max	Min	Mean	Max	Min	Mean	Max
100	83.73	83.83	84.79	8.57	8.58	8.60	1.80	1.80	1.81
200	90.90	91.04	91.25	8.88	8.92	8.98	1.83	1.83	1.83
300	101.26	102.15	108.49	9.30	9.32	9.41	1.86	1.87	1.87
400	116.92	116.93	117.06	9.55	9.55	9.58	1.89	1.90	1.90
500	125.76	125.76	125.82	9.66	9.68	9.71	1.93	1.93	1.93
600	127.48	128.00	137.59	9.80	9.83	9.92	1.96	1.96	1.98
700	149.32	149.33	149.46	10.13	10.20	10.33	2.01	2.01	2.01
800	151.28	151.84	153.38	10.48	10.50	10.53	2.04	2.04	2.04
900	166.38	166.45	167.23	10.90	10.93	10.96	2.05	2.05	2.06
1000	170.96	171.14	171.23	11.22	11.24	11.27	2.08	2.08	2.08

Table 16
The relationship between population size and statistical behavior of GA.

	Group 1			Group 2			Group 3		
Pop. num.	Min	Mean	Max	Min	Mean	Max	Min	Mean	Max
10	127.06	127.06	127.06	8.78	8.79	8.80	1.90	1.90	1.90
20	155.87	155.87	155.87	10.45	10.46	10.47	2.10	2.10	2.12
30	187.18	187.18	187.18	11.18	11.23	11.33	2.09	2.09	2.10
40	271.70	272.65	276.28	11.17	11.18	11.28	2.13	2.13	2.13
50	**296.85**	**296.90**	**297.70**	11.48	11.49	11.54	2.12	2.12	2.12
60	289.30	289.45	290.91	**12.37**	**12.39**	**12.54**	2.17	2.17	2.18
70	238.80	238.88	239.52	10.99	11.02	11.16	2.17	2.17	2.17
80	253.76	254.43	255.08	12.06	12.07	12.10	**2.18**	**2.19**	**2.19**
90	206.69	207.51	208.43	11.25	11.27	11.30	2.15	2.15	2.16
100	200.12	200.14	200.19	11.64	11.66	11.69	2.14	2.14	2.14

For each project, we applied the GA-Scheduling algorithm with different parameter settings. The range for each parameter is shown in table 14. Each project has been executed with a combination of parameters.

For each set of parameters, the behavior of the genetic population was evaluated, using the minimum (Min), the maximum (Max), and average (Mean) values of the fitness value seen in the population.

Table 15 shows the relationship between the number of solutions in the population and the behavior of these three statistics of the GA solution. The relationship between the

Table 17
The relationship between crossover probability and statistical behavior of GA.

	Group 1			Group 2			Group 3		
CrossOver %	Min	Mean	Max	Min	Mean	Max	Min	Mean	Max
10	101.90	101.90	101.90	9.61	9.61	9.61	1.94	1.94	1.94
20	116.80	117.10	117.10	9.40	9.41	9.46	1.97	1.97	1.97
30	124.75	124.78	125.51	10.27	10.28	10.30	2.02	2.02	2.02
40	138.52	138.52	138.60	**10.80**	**10.81**	**10.83**	2.04	2.04	2.04
50	209.56	209.56	209.58	10.34	10.35	10.36	2.07	2.07	2.08
60	110.51	110.51	110.51	10.64	10.65	10.67	2.05	2.06	2.06
70	**216.41**	**216.41**	**216.41**	10.47	10.48	10.51	2.07	2.07	2.08
80	165.91	165.91	165.93	10.28	10.29	10.31	**2.11**	**2.12**	**2.12**
90	179.38	179.90	180.76	10.44	10.45	10.47	2.07	2.08	2.08
100	144.76	145.30	145.38	10.05	10.06	10.08	2.10	2.10	2.11

Table 18
The relationship between mutation probability and statistical behavior of GA.

	Group 1			Group 2			Group 3		
Mutation %	Min	Mean	Max	Min	Mean	Max	Min	Mean	Max
10	147.40	147.40	147.40	8.44	8.44	8.45	**1.87**	**1.87**	**1.87**
20	**779.80**	**787.60**	**804.10**	8.41	8.41	8.41	1.81	1.81	1.81
30	328.66	329.41	339.88	8.29	8.29	8.29	1.83	1.83	1.83
40	78.70	78.70	78.70	**8.46**	**8.46**	**8.46**	1.80	1.80	1.80
50	423.47	539.14	543.13	8.17	8.17	8.17	1.79	1.19	1.80
60	109.65	110.88	123.32	8.03	8.03	8.03	1.79	1.79	1.79
70	172.63	172.63	172.63	8.14	8.14	8.14	1.80	1.80	1.80
80	527.84	528.44	533.91	8.15	8.15	8.15	1.80	1.80	1.80
90	737.48	743.34	802.20	8.12	8.12	8.12	1.81	1.81	1.81
100	345.50	381.45	505.40	8.39	8.39	8.39	1.81	1.81	1.81

generation numbers and GA statistics is depicted in table 16. Table 17 shows the impacts of crossover probability to GA statistics. Table 18 shows the mutation probability effect.

Table 15 shows that statistically the fitness values will increase with the generation numbers. The variance of each population will decrease while the populations converge to a steady state.

We were interested in whether or not there is an optimal combination of parameters that improves the performance of the Genetic algorithm. Tables 16–18 suggests that when the population size is around 50–80, crossover around 40–80%, and mutation around 10–40%, the genetic algorithm perform best.

8. Conclusions

This paper reports new research results as a major extension to the thesis work done by Chao [Chao 1995] that was inspired by Chang's message [Chang 1993] to encourage the

software engineering researchers to bridge the gap between *soft computing* and *software engineering.*

Borrowing the wisdom of *soft computing*, this research applied an existing GA software library to the problems of project scheduling. The representation chosen for this work was PM-Net [Chang and Christensen 1999]. We introduced a new internal representation of PM-Net and implementation for application of genetic algorithms to project scheduling. This technique is able to determine the near-optimal resource allocation and automatically schedule the resources. Validation of our techniques has been conducted by five test problems with progressively increased complexity. Results show that the GA scheduling approach provides an attractive tool to project management.

The major advantage of GAs lies in their ability to generate near-optimal solutions rapidly. All five tests demonstrated superior performance compared to exhaustive search. Another advantage of the GA approach to project management is the ease with which it can be adapted to varying constraints and objectives as weighted components of the fitness function (or objective function). This makes it easy to adapt the GA scheduler to the particular requirements of projects.

The tuning of the GA's free parameters is currently rather *ad hoc*. Indeed, most of the research into GA seams to be centered on providing minor improvements to efficiency without much theoretical justification. A clearer theoretical framework is needed for more thorough understanding of the subject. It is conceivable that a quantitative statistical basis for GA may be found in Bayesian inference [Backus 1988; Gallagher *et al.* 1991].

Genetic algorithms are conceptually simple and well suited to problems with a mix of continuous and discrete variables. They frequently outperform other more direct methods such as gradient descent on difficult problems, i.e., those involving highly nonlinear, high dimensional, discrete, multi-modal or noisy functions. Many empirical simulations have demonstrated the efficiency and robustness of GA on different optimization problems. However, GA are not guaranteed to find the global functional optima simply because the search process is not guaranteed to cover the entire state space. However, we are encouraged by our results in the sense that currently most project management tools failed even to attempt to tackle the difficult NP-hard scheduling problems.

9. Limitations and future work

We discovered several limitations of our current design and implementation. Some of the most important issues are discussed below.

9.1. Integrating the COCOMO model with the constraint

One assumption made in implementing the constraints was that the time to complete a task is inversely proportional to the number of persons involving in it. This is not an essential issue but did simplify the calculation of cost. However, it is well known that this is not realistic. The COCOMO model [Boehm 1981] provides a simple mechanism

that we will use to eliminate this flaw. It gives the relationship between the time needed to complete a project and the number of employees involved. The empirical relationship between cost (total labor hours in person-months (PM)) for development and length of duration is

$$\text{TDEV} = 2.5 \cdot (\text{MM})^{0.38}.$$

So the average, normal staffing on a task is

$$\text{Staff} = \frac{\text{MM}}{2.5 \cdot \text{MM}^{0.38}} = 0.4 \cdot \text{MM}^{0.62}.$$

The staffing number is thus nonlinear with the total labor content of a task. The COCOMO equation can be used as a filter to restrict on the number of employees working on same tasks. In other words, we can constrain the distribution of employee such that execution of tasks follows COCOMO pattern. Of course, not every task in a project would be subject to this constraint.

An alternative way to integrate COCOMO or other cost model would be to introduce an efficiency function into the fitness function. This function would penalize unrealistic job loading and would give feedback to the genetic algorithm to generate more efficient schedules.

9.2. More constraints on people

Modern project management practices should encourage individual career growth. Thus job assignments could also consider an employee's experience needs. This would require that information of this type would need to be added to the employee's current skill set. Including this factor will, of course, increase the complexity of the scheduling problem and may conflict with other project objectives. In addition, if employees are assigned tasks that they have little or no experience in performing, some form of "learning curve" must be included in the model.

In addition, for reasons of efficiency, it is common practice to assign a person to no more than a small numbers of tasks at any one time. The rule of thumb we suggest is to assign at most 1.5–2 tasks per person. This can easily be included in the model and will, in fact, serve to reduce the number of possible solutions.

Finally, the limit of overloading may vary from person and person. For instance, lead programmers or team leaders may have higher limits than others.

References

Blazewicz, J., J.K. Lenstra and A.H.G. Rinnooy Kan (1983), "Scheduling Subject to Resource Constraints: Classification and Complexity," *Discrete Applied Mathematics 5*, 11–24.

Boehm, B. (1981), *Software Engineering Economics*, Prentice-Hall, Englewood Cliffs, NJ.

Chang, C.K. (1993), "Is existing software engineering obsolete?" *IEEE Software*, September, 4–5.

Chao, C. (1995), "SPMNET: A New Methodology for Software Management", Ph.D. Thesis, The University of Illinois at Chicago.

Chang, C.K. and M. Christensen (1999), "A Net Practice for Software Project Management," *IEEE Software*, November/December, 80–88.

Davis, E.W. and G.E. Heidorn (1971), "An Algorithm for Optimal Project Scheduling under Multiple Resource Constraints," *Management Science 17*, 12, B803–B817.

Forrest, S. (1993), "Genetic Algorithms: Principles of Natural Selection Applied to Computation," *Science 261*, 872–878.

Gallagher, K., M. Sambridge and G. Drijkoningen (1991), "Genetic Algorithms: An Evolution from Monte Carlo Methods for Strongly Non-Linear Geophysical Optimization Problems," *Geophysical Research Letter 18*, 12, 2177–2180.

Goldberg, D.E. (1989), *Genetic Algorithm in Search, Optimization and Machine* Learning, Addison-Wesley, Reading, MA.

Goodman, E.D. (1996). "An Introduction to GALOPPS Release 3.2 (July 31, 1996)," Technical Report No. 96-07-01, Genetic Algorithms Research and Applications Group (GARAGe), Michigan State University.

Holland, J.H. (1992), "Genetic Algorithms," *Scientific American*, 66–72.

Knjazew, D. (2000), "Ordering Messy Genetic Algorithm in C++," IlliGAL Report No. 2000034.

Ozdamar, L. and G. Ulusoy (1995), "A Survey on the Resource-Constrained Project Scheduling Problem," *IIE Transactions 27*, 574–586.

Schraudolph, N.N. and J.J.Grefenstette, (1992), "A User's Guide to GA_{UCSD} 1.4", Technical Report CS92-249, CSE Department, UC San Diego.

Wall, B.M. (1996), "A Genetic Algorithm for Resource-Constrained Scheduling", Ph.D. Thesis, MIT; `http://lancet.mit.edu/ga`.