Search–based techniques for optimizing software project resource allocation

Anonymous as per GECCO double blind submission rules

January 19, 2004

Abstract

We present a search–based approach for planning resource allocation in large software projects, which aims to find an optimal or near optimal order in which to allocate work packages to programming teams, in order to minimize the project duration.

The approach is validated by an empirical study of a large, commercial Y2K massive maintenance project, comparing random scheduling, hill climbing, simulating annealing and genetic algorithms, applied to two different problem encodings.

Results show that a genome encoding the work package ordering, and a fitness function obtained by queuing simulation constitute the best choice, both in terms of quality of results and number of fitness evaluations required to achieve them.

Keywords: Software Project Management, Genetic Algorithms, Queuing Networks

1 Introduction

In software development, testing and maintenance, as in other large scale engineering activities, effective project planning is essential. Failure to plan and/or poor planning can cause delays and costs that, given timing and budget constraints, are often unacceptable, leading to business–critical failures. Traditional tools such as the Project Evaluation and Review Technique (PERT), the Critical Path Method (CPM), Gantt diagrams and Earned Value Analysis help to plan and track project milestones. While these tools and techniques are important, they cannot assist with the identification of optimal scheduling assignment in the presence of configurable resource allocation.

However, most large scale software projects involve several teams of programmers and many individual project work packages [16]. As such, the optimal allocation of teams of programmers (the primary resource cost drivers) to Work Packages (WPs) is an important problem which cannot be overlooked.

In this paper we study this problem from the perspective of a massive software maintenance project. The term 'massive' is used to refer to those maintenance interventions, such as Y2K remediation, Euro conversion or phone numbering change, involving a large number of applications simultaneously [11]. Such maintenance activities present particularly acute problems for managers, since they have fixed hard deadlines and cut right across an entire software portfolio, touching almost every software asset possessed by the organisation.

When a massive maintenance request arrives, it is split in WPs according to the project work-breakdown structure. An analogy estimate [17] can be used to determine the effort required to maintain each WP. Having obtained estimates for effort, the next task is to determine the order in which WPs flow into the queuing system to be dealt with by the next available team of programmers.

The order of presentation of WPs is a way of describing the allocation of programmer teams to WPs. Such a resource allocation problem is an example of a bin packing problem, the solution of which is NP-hard [8] and, for which, search based techniques are known to be effective [10]. This paper presents the results of an empirical study into the applicability of search–based techniques to software resource allocation problems. The approach is validated by an empirical study, using historical data from a real–world massive Y2K maintenance intervention, conducted on a financial system for a European company.

The primary contributions of this paper are as follows:

- This is the first paper to consider the problem of software project resource allocation using Search–Based Software Engineering.
- The paper presents results from an empirical study which compares two different encoding strategies. For each strategy, results are reported for implementations of four algorithms: genetic algorithm, simulated annealing, hill climbing and random search. The empirical study's experiments are conducted on real-world data from a massive Y2K intervention.
- The paper also presents results of a study into the effect of changing the staffing levels on the overall time required.

The remainder of the paper is organized as follows. After a brief overview of existing scheduling approaches and application of heuristic approaches to software project management, Section 3 describes the search techniques used in the present paper. Section 4 lists the research questions this paper aims to answer, while Section 5 reports and discusses the results from the empirical study. Section 6 concludes.

2 Related Work

One of the first examples of search–based scheduling was due to Davis [6]. A survey of the application of genetic algorithms to solve scheduling problems has been presented by Husbands [10]. The mathematical problem encountered is, as described by the author, the classical, NP-hard, bin packing or shop-bag problem. A survey of approximated approaches for the bin packing problem is presented in [5].

Search heuristics have been applied in the past to solve some related software project management problems. In particular, Kirsopp et al. reported a comparison of random search, hill climbing and forward sequential selection (FSS) to select the optimal set of project attributes to use in a search–based approach to estimating project effort [13].

A comparison of approaches (both analytical and evolutionary) for prioritizing software requirements is proposed in [12], while Greer and Ruhe proposed a GA-based approach for planning software releases [9].

Finally, the problem of staffing a software maintenance project using queuing networks and discrete-event simulation was addressed by Antoniol et al. [3]. Given an (ordered) distribution of incoming maintenance requests, the goal of Antoniol et al. was to determine the staffing levels for each team. This paper aims to augment this approach, using search–based techniques to determine the optimal WP ordering.

3 The Different Approaches

At a first level of approximation, the maintenance task is considered as a monolithic step task (i.e., the single node model of Antoniol et al. [3]). This section explains how we formulated the problem as a search–based problem, using two different encodings, GAs, hill climbing and simulated annealing. A random search was also implemented for each encoding, to provide base line (worst case) data.

3.1 The encodings used

The search approaches applied in this paper were implemented for two different schemas of genome encoding and fitness function: the *pigeon hole* genome and the *ordering* genome. The overall implementation, combining the search-based heuristics (GA, hill climbing, simulated annealing) and queuing simulation is shown in Figure 1-c.

3.1.1 The pigeon hole genome

The pigeon hole genome describes the genome as an array of N integers, where N is the number of WPs. Each value of the array indicates the team the WP is assigned to. The genome schema is shown in Figure 1-a. The fitness function (which is minimised) is simply the value of the project's overall deadline. That is, for a single-step/multi-server maintenance process, the maximum completion time among the different servers.

To implement a GA for this encoding, the mutation operator randomly selects a WP and randomly changes its team. The crossover operator is the standard *single point* crossover.

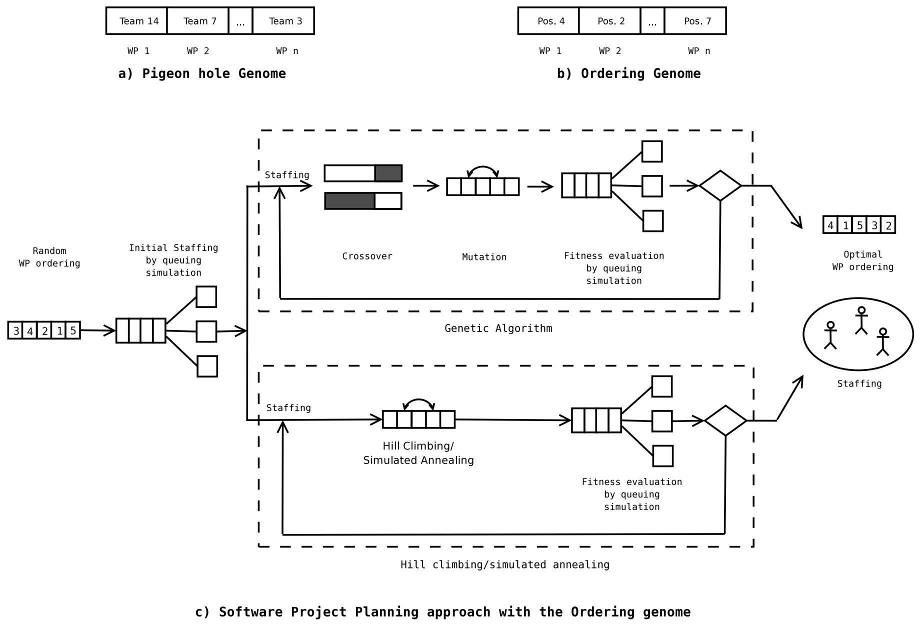
Team 14
Team 7
...
Team 3
WP 1
WP 2
WP n
a) Pigeon hole Genome
Pos. 4
Pos. 2
...
Pos. 7
WP 1
WP 2
WP n
b) Ordering Genome
Random WP ordering
3 4 2 1 5
Initial Staffing by queuing simulation
Staffing
Crossover
Mutation
Fitness evaluation by queuing simulation
Genetic Algorithm
4 1 5 3 2
Optimal WP ordering
Staffing
Staffing
Hill Climbing/ Simulated Annealing
Fitness evaluation by queuing simulation
Hill climbing/simulated annealing
c) Software Project Planning approach with the Ordering genome

Figure 1: a) The Pigeon Hole Genome - b) The Ordering Genome - c) The proposed approach

3.1.2 The ordering genome

The ordering genome also represents the problem as an N-sized array, but the value of a genome element indicates the position of the WP in the incoming queue, for a single-queue/multi-server queuing system. The genome schema is shown in Figure 1-b.

The fitness function takes as input, the genome (i.e. the WP sequence) and computes the finishing deadline using the queuing simulator of Antoniol et al. [3].

To implement a GA for this encoding, the mutation operator randomly selects two WPs (i.e. two array items) and exchanges their position in the queue. The crossover operator is somewhat more complex. Two offspring (o_1 and o_2) are formed from two parents (p_1 and p_2), as follows:

1. A random position k, is selected in the genome.
2. The first k elements of p_1 become the first k elements of o_1.
3. The last N-k elements of o_1 are the sequence of N-k elements which remain when the k elements selected from p_1 are removed from p_2.
4. o_2 is obtained similarly, composed of the first N-k elements of p_2 and the remaining elements of p_1 (when the first N-k elements of p_2 are removed).

For instance, if $k = 2$ and $p_1 \equiv \{4, 2, 3, 6\}$ and $p_2 \equiv \{4, 6, 3, 2\}$, then $o_1 \equiv \{4, 2, 6, 3\}$ and $o_2 \equiv \{4, 6, 2, 3\}$.

This approach to crossover has the advantage that it guarantees that each offspring contains precisely one position in the sequence per WP. It therefore avoids the need for repair, or some other mechanism which might be required to deal with duplication of sequence numbers in a more simple-minded crossover operator.

3.2 The hill climbing approach

The hill climbing approach works as follows:

1. It starts with a initial solution where each WP is randomly assigned to a team;
2. It randomly takes a WP and assigns it to a new, randomly selected, team.
3. The new configuration is accepted if and only if it leads to a finishing deadline smaller than the one obtained with the previous configuration.
4. The algorithm iterates through step 2 for a given number of times, or until, for a given number of steps, the objective function does not improve.

3.3 The simulated annealing approach

It is well known [15] that one of the main weaknesses of the hill climbing approach is that it suffers of the problem of local optima. To overcome such a problem, a simulating annealing approach [15] can be applied instead.

Simulating annealing randomly selects a WP and tries to randomly assign it to a new team. Unlike hill climbing, however, if the latter causes an increase of the objective function (which we seek to minimize), the new configuration *can* be accepted if:

$$p < m \tag{1}$$

where:

- p is a random number in the range $[0 \ldots 1]$ and
- $m = e^{\Delta fitness/t}$

where t was chosen as

$$t = \frac{\alpha}{log(x + \beta)} \tag{2}$$

α and β are constants of the same order of magnitude of $\Delta fitness$, and x is the current number of iterations.

4 Research Questions

The research questions this paper aims to answer are the following:

- For a fixed staffing level, what is the optimal order in which to present the WPs for action?
- How do the results vary with team size and distribution?
- What is the difference between GA, hill climbing and simulated annealing, both in terms of result quality and number of required fitness evaluations?
- Which is the best genome representation fitness function?

5 Empirical Study

5.1 Empirical study description

The empirical study proposed in this paper aims at defining a near optimal scheduling for maintenance activities of work packages coming from a massive maintenance project, related to fixing the Y2K problem in a large financial software system from a European commercial financial organisation.

According to its Work Breakdown Structure (WBS), the application was decomposed into WPs, i.e. loosely coupled, elementary units (from one to nine for each application) subject to maintenance activities; each WP was managed by a WP leader and assigned to a maintenance team (the average team size was approximately four programmers). Overall, the entire system was decomposed in 84 WPs, each one composed, on average, of 300 COBOL and JCL files.

The project followed a phased maintenance process (similar to that defined by the IEEE maintenance standard [1]), encompassing five macro-phases:

1. *Inventory*: deals with the decomposition of the application portfolio into independent applications, and successive decomposition of each application into WPs;
2. *Assessment*: identifies, for each WP, candidate impacted items, using automatic tools;
3. *Technical Analysis (TA)*: deals with the analysis of impacted items and identifies a candidate solution among a set of pre-defined solution patterns;
4. *Enactment (Enact)*: an automatic tool was used to apply patches to the problems identified and analyzed in the previous phases. The solution was usually based on windowing [14];
5. *Unit Testing (UT)* was performed on each impacted WP; tools were used for automatic generation of test cases.

Because the intervention was performed almost semi-automatically and involved highly standardized activities, it is possible to make an assumption that it is valid to interchange between people and months. That is, given a maintenance team size, s and the effort required e, the time t necessary to perform the task is:

$$t \simeq \frac{e}{n} \tag{3}$$

Due to Brooks' law [4], this could be an overly optimistic assumption. However, as other authors have noted [2], given the small team sizes (fewer than eight people) and the standard (training–free) nature of the maintenance task, this approximation was considered reasonable. The model can be generalized to situations in which Brooks' law does apply by the simple introduction of a non-linearity factor.

A further simplification introduced with our study is the absence of dependencies between WPs (i.e., there is no constraint on possible orderings which can be selected).

5.2 Case study results

First and foremost, we will analyze the difference i) between the two types of genome, i.e. the ordering genome and the pigeon hole genome, and ii) between all the different approaches described in Section 3. Results for all eight possible combinations are shown in Figure 2. For robustness, the average over ten runs for each encoding/search algorithm was used to produce the results reported in Figure 2. For the GA, the following parameters were chosen:

- Population size: 100 individuals;
- Type of GA: elective;
- Crossover probability: 0.6; and
- Mutation probability: 0.1.

The figure clearly shows that the ordering genome encoding outperforms, in general, the pigeon hole genome encoding. This supports the usefulness of the proposed approach that combines search–based heuristics with queuing simulation. The ordering genome is effective not only for speed of convergence, but also for the possibility it gives of modeling more complex maintenance tasks. A queuing simulator allows modeling multi–stage maintenance processes, even accounting for rework or abandonment after a given phase, as well as for priority queues and for dependencies between WPs.

The results also highlight the fact that applying search–based heuristics is a sensible approach because they significantly outperform a randomly-generated staffing. Only the GA–based pigeon hole algorithm reaches results comparable with the ordering genome, although requiring a high number of generations. The comparison of the different approaches, implemented using the ordering

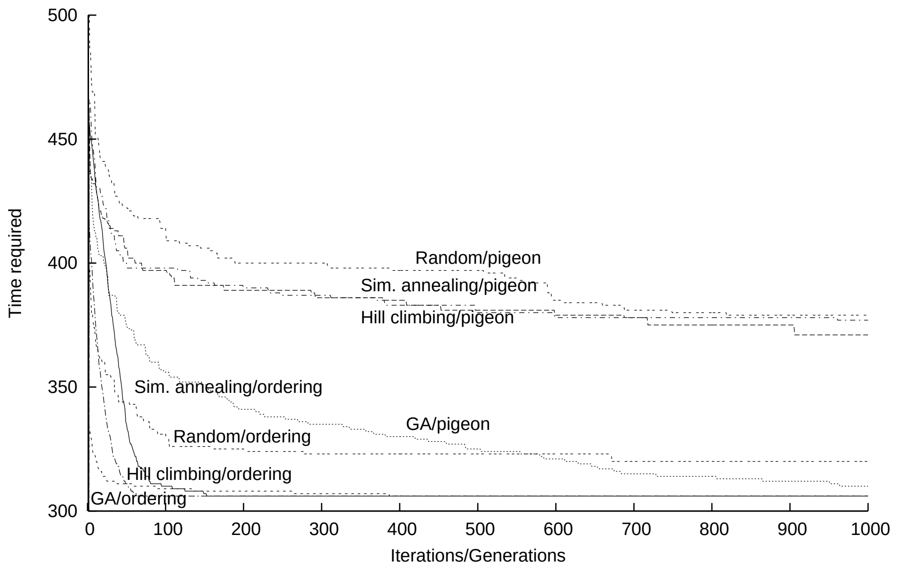

Figure 2: Comparison of different genomes and different approaches

genome, highlights that the GA seems to exhibit a faster start (due to its intrinsic parallelism), although it is then overtaken by the hill climber. After 400/500 generations/iterations the results are exactly the same. Also the simulated annealing approach reaches, after about 100 generations, the same result of GA and hill climbing. It exhibits a slower start due to the likelihood (higher for the first generations) of accepting a worsening solution.

Since hill climbing with the ordering genome is identified as the best search technique/encoding, according to the research questions stated in Section 4, the question now becomes:

1. How does the estimated finishing time vary with the number of people available? and

2. What happens if we consider groups of different size? That is, Like De Penta et al. [7], we may have some *fast lanes* in the queuing model. However, while in [7] the authors dedicated some servers to the shortest maintenance requests (similar to the fast lane checkout in a supermarket), here we can differently distribute the available people, having a given percentage of double–sized teams, i.e. *fast servers* to which the longest requests will be dispatched. As before, this assumes that Brooke's law is avoided because of the nature of the project (See Equation 3).

Results for different numbers of people available in total, and for different percentages of double–sized teams are shown in Figure 3. In particular, each line represents a different number (from 10 to 40) of people involved, the Y axis represents the result of the staffing (i.e., the estimated finishing time), while the X axis represents the percentage of double–sized teams. For instance, a percentage of 10% for a total of 10 people available means that 10% of the teams will be composed of two persons instead of one only. In this case we will have a total of 9 teams, 8 composed of one person and one (about 10% of 9) composed of two people. As explained above, such a team can be used to handle (in less time) the longest requests; the fitness function will therefore tend to award the assignment of a long task to a double–sized team, in that the resulting overall finishing time will be shorter. On the other hand, assigning short tasks to such a team will be almost useless, and could prevent the correct assignment of the longest tasks (due to the busy state of the resource).

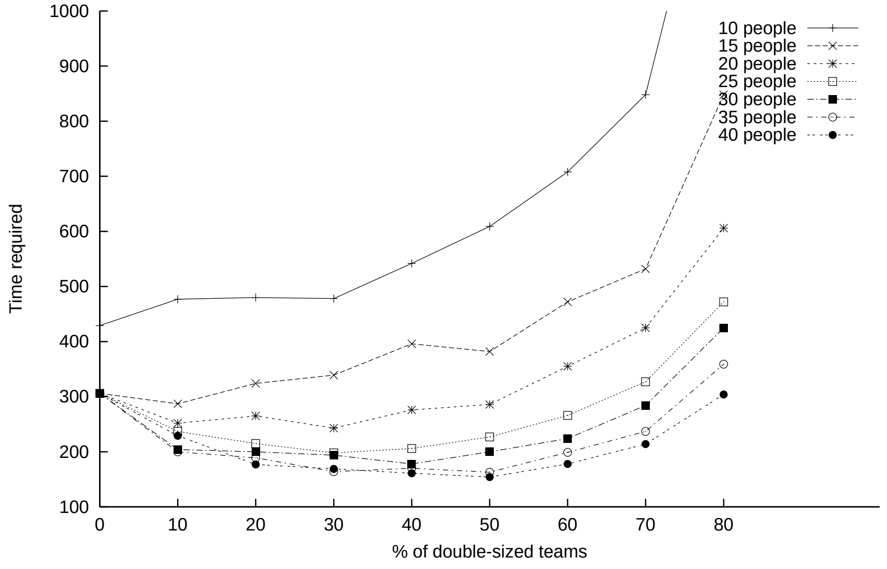

Figure 3: Performance obtained with different numbers of people available overall for the project and different percentages of double–sized teams

The figure shows that, for a small number of people available (say 10 to 15), it is not convenient to have any *fast servers*, since it implies a reduction of the total number of servers; for such a staffing level this is unacceptable. However, when the staffing increases, instead of excessively increasing the number of servers it turns out to be useful to have at least a small percentage of *fast servers*.

These fast servers are able to avoid that the longest task duration determines the overall finishing time. For example, let us suppose that the longest task requires 60 days, while all the remaining ones can be accomplished in 30 days. In that case, the total duration of the project cannot be smaller than 60 days. Instead, if we double the size of the team working on that task, the total finishing time could be, for example, 40 days (because the longest task now completes in 30 days and that the reduction in the number of people available for the other tasks causes them to finish in 40 days).

A further increase of staffing tends to increase the optimal percentage of *fast servers*, moving the minimum point of the 'bathtub curve' to the right.

6 Conclusions

This paper has demonstrated that search–based techniques can be applied to optimise resource allocation in a software engineering project. Three search based techniques were evaluated. Each was applied to two very different encoding strategies. Each encoding represents the way in which the work packages of the overall project are to be allocated to teams of programmers.

The ordering encoding, which combines the search–based approach with a queuing simulation model, was found to outperform the other approaches.

For the less optimal encoding the GA performed significantly better than the other approaches. For the optimal encoding, though GA starts better simulated annealing and hill climbing approaches soon catch up, so that the overall difference between the three approaches appears to be small, compared to the problem of establishing an effective encoding.

Finally, the paper reports the results of experiments that alter the size of the project teams. While for a small overall staffing level, double-sized teams do not improve performance, increasing the overall staffing level is sufficiently high, it proved effective to have double–sized teams.

7 Acknowledgments

References

[1] *IEEE std 1219: Standard for Software maintenance.* 1998.

[2] T. Abdel-Hamid. The dynamics of software project staffing: a system dynamics based simulation approach. *IEEE Transactions on Software Engineering*, 15(2):109–119, 1989.

[3] G. Antoniol, A. Cimitile, G. A. Di Lucca, and M. Di Penta. Assessing staffing needs for a software maintenance project through queuing simulation. *IEEE Transactions on Software Engineering*, 30(1):43–58, Jan 2004.

[4] F. Brooks. *The Mythical Man-Month 20th anniversary edition.* Addison-Wesley Publishing Company, Reading, MA, 1995.

[5] E. J. Coffman, M. Garey, and D. Johnson. Approximation algorithms for bin-packing. *In Algorithm Design for Computer System Design*, 1984.

[6] L. Davis. Job-shop scheduling with genetic algorithms. In *International Conference on GAs*, pages 136–140. Lawrence Erlbaum, 1985.

[7] M. Di Penta, G. Casazza, G. Antoniol, and E. Merlo. Modeling web maintenance centers through queue models. In *European Conference on Software Maintenance and Reengineering*, pages 131–138, Lisbon, Portugal, March 2001. IEEE Society Press.

[8] M. Garey and D. Johnson. *Computers and Intractability: a Guide to the Theory of NP-Completeness*. W.H. Freeman, 1979.

[9] D. Greer and G. Ruhe. Software release planning: an evolutionary and iterative approach. *Information and Software Technology*, (to appear).

[10] P. Husbands. Genetic algorithms for scheduling.

[11] C. Jones. Mass-updates and software project management in is organizations-*http://www.artemis.it/artemis/artemis/lang_en/libreria/mass-updates.htm*, 1999. Accessed on Aug, 25 2003.

[12] J. Karlsson, C. Wohlin, and B. Regnell. An evaluation of methods for priorizing software requirements. *Information and Software Technology*, 39:939–947, 1998.

[13] C. Kirsopp, M. Sheppard, and J. Hart. Search heuristics, case-based reasoning and software project effort prediction. In *Genetic and Evolutionary Computation Conference*. Springer-Verlag, 2002.

[14] E. Lynd. Living with the 2-digit year, year 2000 maintenance using a procedural solution. In *Proceedings of IEEE International Conference on Software Maintenance*, pages 206–212, Bari, Italy, 1997.

[15] M. McLaughlin. Simulated annealing. *Dr. Dobb's Journal*, pages 26–37, September 1989.

[16] R. S. Pressman. *Software Engineering: A Practitioner's Approach 3rd edition*. McGraw-Hill, 1992.

[17] M. Shepperd and C. Schofield. Estimating software project effort using analogies. *IEEE Transactions on Software Engineering*, 23(11):736–743, 1997.