MANAGEMENT SCIENCE
Vol. 54, No. 1, January 2008, pp. 194–207
ISSN 0025-1909 | EISSN 1526-5501 | 08 | 5401 | 0194
informs®
DOI 10.1287/mnsc.1070.0766
© 2008 INFORMS

Maximum Commonality Problems: Applications and Analysis

Milind Dawande
School of Management, University of Texas at Dallas, Richardson, Texas 75083, milind@utdallas.edu

Subodha Kumar
Michael G. Foster School of Business, University of Washington, Seattle, Washington 98195, subodha@u.washington.edu

Vijay Mookerjee, Chelliah Sriskandarajah
School of Management, University of Texas at Dallas, Richardson, Texas 75083
{vijaym@utdallas.edu, chelliah@utdallas.edu}

Recently, an agile software development technique called *extreme programming* has caught the attention of practitioners and researchers in the software industry. A core practice of extreme programming is *pair programming*, where two developers work on the same piece of code. We introduce the problem of assigning pairs of developers to modules so as to maximize the commonality—a measure of the extent to which common developers work on related modules—subject to a load-balancing constraint that is motivated by the need to control the completion time of the project. We consider two variants of this problem. In MCAP^n, a developer is teamed up with exactly one other developer to form a pair that works together for the entire duration of the project. In MCAP^s, we allow a developer to pair with more than one other developer during the project. This "pair-splitting" version of the problem facilitates knowledge dissemination among developers, but can increase the effort needed for a developer to adjust to the work habits of several partners.

The difference between the commonality achieved with and without pair splitting crucially depends on the underlying structure of the problem. For trees, we show that the value of the maximum commonality is the same for both MCAP^n and MCAP^s. Additionally, we obtain polynomial-time algorithms for both of these variants. For general graphs, both problems MCAP^n and MCAP^s are shown to be strongly NP-complete. We prove that the maximum commonality for MCAP^s is at most $\frac{3}{2}$ times the maximum commonality of MCAP^n. We also provide polynomial-time algorithms and approximation results for a number of special cases of these problems.

Key words: analysis of algorithms; computational complexity; integer programming; applications; suboptimal algorithms
History: Accepted by Dorit Hochbaum, optimization and modeling; received November 14, 2006. This paper was with the authors 2 months for 2 revisions.

1. Introduction—The Practice of Pair Programming

Both software developers and users often view traditional software development techniques as too slow (Astels et al. 2002). The need to deliver software under increasingly tightening deadlines has motivated industry experts to explore the extremes of different development practices, resulting in a novel software development paradigm called *extreme programming* (XP) (Beck 2000). In the short span of time since its formal introduction in the late 1990s, XP has gained popularity as a lightweight, low-risk, flexible, predictable, scientific, and fun way to develop software.

A key aspect of XP is pair programming—a practice in which two developers jointly work on the same task: design creation, algorithm development, coding, or testing (Astels et al. 2002, Wood and Kleb 2002). One member of the pair, called the driver, actively works on the task (e.g., writes down a design, codes a specific functionality, etc.). The other member, called the navigator, observes the work of the driver and looks for tactical or strategic defects and recommends improvements.

The benefits of pair programming include higher software quality and the consequent reduction in the amount of testing required, reduction in the effort required to integrate the various modules of a system, higher developer morale, better trust and teamwork, and enhanced knowledge transfer and learning (Ambler 2002, Johar et al. 2003, Canfora et al. 2005). Although pair programming is almost always used in XP, this practice is not limited to extreme programming. Even before the XP development methodology was introduced in 1996, Constantine (1995) observed pairs of developers producing code faster and more free of bugs than code developed using traditional methods. XP is now practiced by developers worldwide, and its impressive successes have aroused the curiosity of researchers and practitioners alike in the

software engineering area. The founders of XP credit many of these successes to the use of pair programming (Williams et al. 2000).

1.1. The Notion of Commonality

A software system consists of a set of modules that need to be individually created and later integrated and tested to work properly as a whole. As a consequence, the construction activities to produce a software system are usually partitioned into direct activities (e.g., design, code, unit test, etc.) and coordination activities (e.g., system integration, system testing, etc.). The total system integration and testing effort can be calculated as the sum of the effort needed for each pair of modules that needs to be tested and integrated. Thus, a software system can be conceptualized as a set of modules with a link between certain pairs of modules to indicate that their development activities are related.

When two modules are connected by a link, using (one or more) common developers can reduce the integration and testing effort for these modules. In other words, the activities associated with producing a connected pair of modules exhibit scope economies. Such scope economies mainly accrue from a reduction in coordination-related activities. For example, the two modules may share code (e.g., one module calls the other; a module inherits methods or attributes from the other, etc.), and having common developers can be expected to reduce the effort needed to integrate these modules. In other situations, the output of one module may be used by the other to produce a common functionality. Once again, assigning common developers to the pair of modules should reduce the integration and testing time. Thus, in our conceptualization of a software system, a link between two modules indicates that some economy of effort accrues from assigning common developers to create, integrate, and test the modules.

For a system containing several connected modules, we consider the following notion of *commonality*:

Given an assignment of two developers to each module, the commonality for a pair of connected modules is the number of common developers assigned to the two modules. The total commonality (*for the system*) *is the sum of the commonalities for each pair of connected modules.*

Although commonality is a direct consequence of the assignment of developers to modules, this assignment problem has lacked formal study. Moreover, practical implementations of pair programming demonstrate that many of the benefits of pair programming amplify as the number of common developers for pairs of connected modules increases (Astels et al. 2002, Erdogmus and Williams 2003, Kuppuswami et al. 2003, Wood and Kleb 2002). Several XP practitioners believe that the benefits of pair programming are mitigated due to the absence of a formal methodology for assigning developer pairs to modules; see private communications.[1]

A subtle feature of pair programming is pair splitting—a practice in which a developer pairs with several different developers during the project. Thus, a developer may work with one partner to develop a module, and with a different partner when developing another module. At a high level, pair splitting facilitates the dissemination of knowledge within the team, breaks down communication barriers between teammates, and reduces the training time needed to assimilate new members (Shukla 2002, Wood and Kleb 2002). Specifically, by pairing with many different partners, developers learn more about the system's architecture, and share programming skills, style, tools, techniques, tricks, etc., with other developers. A recent practitioner survey (Williams et al. 2004) indicates that pair splitting is widely used.

Despite its apparent benefits, the practice of pair splitting can be inefficient because it requires developers to adjust to different partners with (possibly) different programming styles (Srikanth et al. 2004). Thus, the indiscriminate use of pair splitting may be counterproductive. When pair splitting is done, its impact on commonality becomes an especially important metric for software managers. If the use of pair splitting can increase commonality, the extra overhead involved in forming additional pairs may be justifiable. On the other hand, if for some software projects it is possible to achieve maximum commonality without splitting pairs it may be beneficial to match certain developer pairs and allow them to work together throughout the project. Motivated by these considerations, we consider two variants of the developer assignment problem—with and without pair splitting—described below.

The formal definitions of these problems are provided in §2. Informally, the problem can be described as that of finding an assignment of modules to developers to maximize commonality such that the following conditions are met:

1. Every module is assigned to exactly two developers.
2. No developer is assigned to more than a prespecified number T of modules.

The requirement that a developer can be assigned to at most T modules—a load-balancing constraint—is motivated by the need to control the completion

[1] Private communications: E. Herman, Product Sight Corporation, Bellevue, WA; P. Kedia, Microsoft Corp., Redmond, WA; R. C. Martin, CEO, President and Founder, Object Mentor Inc., Gurnee, IL; C. Poole, Founder and Principal, Poole Consulting, Poulsbo, WA; B. Ramsdell, Co-founder, Brute Squad Labs., Emeryville, CA; R. Rangan, CTO, Product Sight Corporation, Bellevue, WA; A. Ridlehoover, SBI Group, Bellevue, WA; S. Sidhartha, Microsoft Corp., Redmond, WA; R. Venkatapathy, Microsoft Corp., Redmond,WA.

time of the project. In the absence of this constraint, the assignment problem becomes trivial because maximum commonality can be achieved by assigning a single pair of developers to all the modules in the system. Most projects, however, need to be completed by a deadline, and having a single pair of developers work on the entire system is infeasible in most practical situations.

1.2. Summary of Results

We consider two variants of the maximum commonality problem: one in which pair splitting is not allowed (first variant), and the other in which pair splitting is allowed (second variant). We summarize our results below.

(1) Due to its prevalence in practice, we pay special attention to the case when the underlying graph of the problem is a tree. On trees,

(a) We obtain polynomial-time algorithms for both variants of the maximum commonality problem.

(b) We show that the maximum commonality is the same for both variants. In other words, allowing pair splitting does not affect the maximum commonality in trees.

(2) For a general graph,

(a) We show that the maximum commonality with pair splitting is at most $\frac{3}{2}$ times the maximum commonality without pair splitting. We also show that this bound is tight.

(b) When the load-balancing upper bound is $T = 2$, there is a precise correspondence between the two variants and integer and linear versions of the classical matching problem. As a consequence, we provide polynomial-time algorithms for both of the variants for $T = 2$.

(c) We prove that the decision problems corresponding to both of variants are strongly NP-complete for $T = 3$.

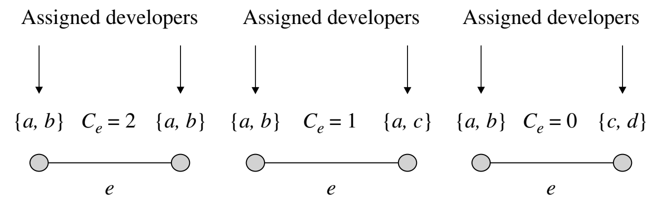
(d) For $T = 3$, we provide two polynomial-time approximations—a $\frac{2}{3}$-approximation for the first variant and a $\frac{1}{2}$-approximation for the second variant.

The remainder of this paper is organized as follows. Section 2 defines the two variants of the commonality problem. For trees, §3 provides polynomial-time algorithms for both of the variants, and proves that the maximum commonality of a graph remains the same, with or without pair splitting. In §4, we discuss pair programming on general graphs using a few examples, and present some insights and bounds. Sections 5 and 6 present, respectively, our results for the first and second variants on general graphs. Finally, §7 concludes this paper and provides directions for future research.

2. Problem Description

We start by defining the general version of the maximum commonality problem.

Figure 1 The Commonality of an Edge



Maximum Commonality Assignment Problem (MCAP).

Instance. n modules; an undirected graph $G(V, E)$, $|V| = n$, in which the nodes represent the modules and the edge set defines the connectivity of the modules; positive integers $T \leq n$ and $R \leq 2n$.

Solution. An assignment of two distinct developers to each node of G such that (i) each developer is assigned to at most T nodes, and (ii) the total number of developers used in the assignment is at most R.

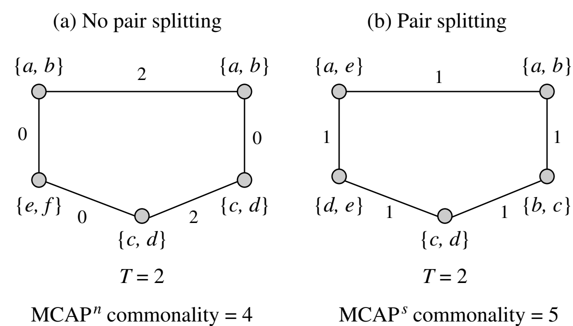
Objective Function. Maximize the commonality, $C(G)$, defined as follows: For an edge $e = (i, j)$, its commonality C_e is the number of common developers between those assigned at i and j (see Figure 1). The commonality of G, $C(G) = \sum_{e \in E} C_e$.

Note that any feasible assignment to problem MCAP uses at most $2n$ distinct developers. In this paper, we assume $R = 2n$. In other words, we assume that there is no restriction on the number of developers available for assignment. Two specific versions of MCAP are considered in this paper:

1. **Maximizing Commonality with No Pair Splitting:** The variant of MCAP where pair splitting is not allowed. We refer to this problem as MCAP^n. Thus, a developer can pair with only one partner in a feasible solution to MCAP^n. It follows that $C_e \in \{0, 2\}\ \forall e \in E$ in any feasible solution to problem MCAP^n. For $T = 2$, Figure 2(a) illustrates an assignment for an odd cycle. A total of six developers (a, b, c, d, e, and f) are used in this solution. Developer a is exclusively paired with developer b and vice versa; other such pairs are (c, d) and (e, f).

2. **Maximizing Commonality with Pair Splitting Allowed:** In this variant, a developer is allowed to be paired with more than one partner. We refer to

Figure 2 Effect of Pair Splitting on Commonality



this variant as MCAP^s. Thus, $C_e \in \{0, 1, 2\}$ $\forall e \in E$. An example is shown in Figure 2(b). Here, developer a is paired with developer b at one node and with developer e at another node. Similarly, developers b, c, d, and e are each paired with two distinct developers. A total of five developers are used in this solution.

Before we start our analysis, a few comments are in order.

- The common upper bound T—the maximum number of different modules a developer can be assigned to—is used as a surrogate for the makespan of the project. When pair splitting is not allowed, the developer pairs are disjoint. Thus, the developer pairs all work in parallel and uninterrupted. In other words, each developer pair can work on its assigned modules without ever needing to wait for any other developer or for the completion of any module not assigned to it. The number of modules assigned to each developer pair is at most T. Assuming modules of equal complexity (in keeping with the recommended practice in extreme programming of partitioning functionality requirements into *user stories* and the development schedule into *iterations*; (see Beck and Fowler 2001, Astels et al. 2002), each module can be completed, without loss of generality, in unit time. Thus, the makespan of the entire system is at most T. This argument is clearly not valid when pair splitting is allowed. In this case, however, it is easy to see that the makespan is at most $2T - 1$. Thus, when pair splitting is allowed, T is a weak surrogate for the makespan. Also, following the (extreme programming) principles of *coding standards* and *collective code ownership* (Beck 2000), we assume that a module can be assigned to any developer in the team.
- A complete specification of the developer assignments at each node will, of course, determine the total number of developers required by that assignment. However, there are no a priori bounds on the number of developers that can be used. Nevertheless, some trivial bounds are immediate; e.g., because each node $i \in V$ is assigned two distinct developers, the maximum number of developers in any feasible solution is $2n$; any feasible solution that uses exactly $2n$ developers has a commonality of zero. Similarly, because each developer can be assigned to a maximum of T nodes, the minimum number of developers required is $\lceil 2n/T \rceil$.
- Problems MCAP^n and MCAP^s are trivial for extreme values of T. When $T = 1$, we have $2n$ developers and a corresponding commonality of zero. When $T = n$, an optimum assignment to both the problems consists of two developers, say a and b, that are assigned to each node of G; the maximum commonality in this case is $2|E|$.

3. Developer Assignments on Trees

In many situations, it is reasonable to depict the underlying graph of a software system as a tree (Stewart 2004). The artifact being produced (i.e., the software system) is itself a model of real-world business processes that are typically organized in a hierarchical manner. Thus, the structural properties of the system being produced make the tree structure especially important for the maximum commonality problems being studied in this paper.

3.1. Problem MCAP^n on Trees

Consider an arbitrary feasible solution to MCAP^n. Because pair splitting is not allowed, the commonality of each edge $e \in E$ is either zero or two. Thus, $C_e \in \{0, 2\}$. For an edge $e = (i, j)$, $C_e = 2$ iff the same pair of developers is assigned to nodes i and j. Consider the subgraph $\bar{G}(V, \bar{E})$, where $\bar{E} = \{e \in E\colon C_e = 2\}$. In other words, $\bar{G}$ is a subgraph of G induced by those edges $e \in E$ with $C_e = 2$. It is straightforward to see that $\bar{G}$ is a forest with each connected component corresponding to the nodes that have been assigned the same pair of developers. Moreover, because each developer can be used at most T times, the number of nodes in each component of $\bar{G}$ is at most T. Finally, we note that because $C_e \in \{0, 2\}$ $\forall e \in E$, maximizing commonality is equivalent to minimizing the number of edges with a commonality of zero.

The observations above imply that, on trees, problem MCAP^n is equivalent to the following problem:

Tree-Partitioning Problem (TPP). Partition a given tree $H(V, E)$ into a minimum number of subtrees such that the number of nodes in any subtree is at most T.

Problem TPP, applied to the module connectivity graph $G(V, E)$, seeks a minimum cardinality set $\tilde{E}$ of edges that, if deleted, partitions G into subtrees with the property that each subtree has at most T nodes. Given a solution to problem TPP with, say, k subtrees, a corresponding solution to MCAP^n is obvious: We use k disjoint pairs of developers $\{a^i, b^i\}$, $i = 1, \ldots, k$. All the nodes in subtree i are assigned the same pair of developers (a^i, b^i). Note that $C_e = 0$ for $e \in \tilde{E}$ and $C_e = 2$ for $e \in E \backslash \tilde{E}$. The optimality of this assignment for MCAP^n is immediate.

Hadlock (1974) and Kundu and Misra (1977) provide polynomial-time algorithms for problem TPP. In Hadlock (1974), a positive weight is associated to each node of the tree, and the objective is to minimize the number of subtrees such that the sum of node weights in any subtree does not exceed a prespecified upper bound T. Clearly, problem TPP is a special case of this problem where the weight of every node is one. Kundu and Misra (1977) provide an $O(|V|)$ algorithm for the same problem as in Hadlock (1974),

for a rooted tree. This algorithm also solves problem TPP because every partition of a rooted tree induces a partition, satisfying the same conditions, for the corresponding unrooted tree (and vice versa) (Becker et al. 1982). We therefore have the following result:

Theorem 1 (Hadlock 1974, Kundu and Misra 1977). *Problem TPP, and hence problem MCAP^n, can be solved optimally in time $O(|V|)$.*

3.2. Problem MCAP^s on Trees

Problem MCAP^s allows pair splitting, and is therefore a relaxation of problem MCAP^n. Thus, it is reasonable to expect the maximum commonality from allowing pair splitting to be higher than that without pair splitting. However, we now show that the maximum commonality for both problems MCAP^s and MCAP^n is the same for trees. An immediate consequence of this result is the polynomial solvability of problem MCAP^s.

For a graph, consider the problem of maximizing the commonality when only one developer (instead of two) is assigned to each node. We refer to this problem as problem ONED. We make two simple observations about the relationship between problem ONED and problems MCAP^n and MCAP^s:

1. For any graph, the value of the optimum solution to problem MCAP^n equals twice the value of the optimum solution to problem ONED.

2. For a tree, the value of the optimum solution to problem MCAP^s equals twice the value of the optimum solution to problem ONED. Consider a feasible solution, S, of problem MCAP^s and order the pairs of developers at each node such that each developer is either always the first member of a pair or always the second member of a pair. It is easy to see that such an ordering is always possible for a tree and can be obtained, for example, by a depth-first traversal of the tree. Then consider two feasible solutions S_i, $i = 1, 2$, of problem ONED corresponding, respectively, to the first and second members of the developer pairs. Clearly, the commonality of S equals the sum of the commonalities of S_1 and S_2. The result follows.

An immediate consequence of these two observations is the following result.

Theorem 2. *On trees, the optimum solution value of problem MCAP^n equals that of problem MCAP^s.*

Corollary 1. *Problem MCAP^s can be solved optimally in time $O(|V|)$.*

Corollary 2. *Allowing for pair splitting does not increase the maximum commonality in trees when there is a common upper bound T on the number of modules each developer can be assigned.*

The corollary above has useful implications for software developers and managers. Each time a developer pairs with a new partner, a pair formation effort is incurred by the pair of developers to establish the mutual understanding needed to work effectively as a team. This effort, also referred to as the *pair-jelling* effort, may vary depending on the characteristics of the developers and the structure of the organization (Erdogmus and Williams 2003). If the costs for developers having to readjust with new partners and their programming styles is very high, then the benefits of pair splitting are often mitigated (Srikanth et al. 2004). Therefore, in spite of its benefits, pair splitting may not always be preferable. In this context, Corollary 2 provides useful information: if the underlying activity structure of the modules for a particular project is a tree, then there is no gain in terms of commonality from splitting developer pairs.

Note that the primary assumption for Corollary 2 is that the upper bound T is developer independent. An interesting variant of the commonality problem is where we have T *classes* of developers, C_i, $i = 1, \ldots, T$. A developer in Class C_i, if used, must be assigned to *exactly* i modules. That is, we have developer-dependent limits τ_i, $i = 1, 2, \ldots, T$. For the developers in Class C_i, we have $\tau_i = i$. We assume that each class C_i has enough developers; without loss of generality, we may assume that each class C_i has $\lceil 2n/i \rceil$ developers. For example, when $T = 4$, we have four classes of developers C_1, C_2, C_3, and C_4. A developer in Class C_4 (if used) must be assigned to exactly four modules. A developer in Class C_3 (if used) must be assigned to three modules, and so on. For this variant, the usages τ_i, $i = 1, 2, \ldots, T$, are developer dependent (or more correctly, the usage of each developer depends on her class). However, from an algorithmic viewpoint, this variant is equivalent to the commonality problems considered in this paper. If the number of modules assigned to a developer in an optimum solution to a commonality problem considered in this paper is i, then that developer can be considered to belong to Class C_i. Proceeding in this manner, each developer can be accommodated in its unique class because each class has enough developers (i.e., $\lceil 2n/i \rceil$ developers).

The comments above should not be misconstrued to imply that commonality problems with arbitrary developer-dependent limits on the number of modules to which they can be assigned are equivalent to the commonality problems considered in this paper (i.e., ones where we have a common limit T for each developer). The variant mentioned above is equivalent because the following special conditions hold: (i) there are exactly T *classes* of developers, C_i, $i = 1, \ldots, T$, and (ii) each class C_i has $\lceil 2n/i \rceil$ developers.

Another interesting variant is where each node has an associated positive weight and we need a pair-programming assignment such that the following condition holds for each developer: the total weight of the nodes where the developer is assigned is at most T. As mentioned earlier, Hadlock (1974) and Kundu and Misra (1977) solve the weighted version of problem TPP. It is straightforward to verify that Theorem 2 remains valid for the weighted versions of problems MCAP^n and MCAP^s.

4. Pair Programming on a Generalized Structure

As discussed in §3, there are analytical benefits that accrue when the underlying graph is a tree. However, in large software systems, general interaction between the modules could take the form of functional calls, message passing, and updating common data stores, making the tree structure restrictive for such systems. Additionally, when the structure of a software system is a general graph, it is not always possible to represent the relationship between the activities that produce the system in the form of a tree. Thus, our analysis of problems MCAP^s and MCAP^n would be incomplete without analyzing the practice of pair splitting for general graphs.

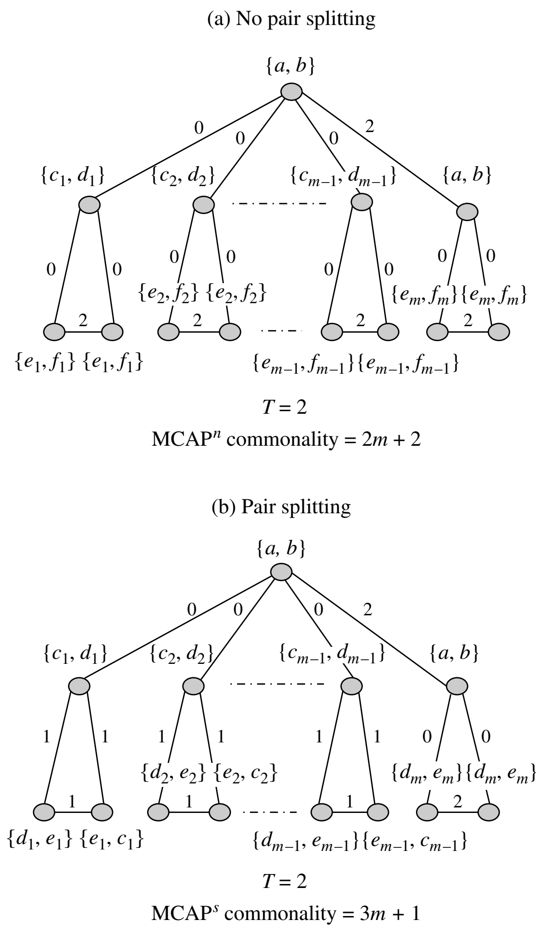
In this section and the next, we discuss problems MCAP^n and MCAP^s on general graphs. This section begins by providing a few examples and insights to illustrate the effect of pair splitting and the nature of the generalized problem. Next, we bound the maximum possible increase in commonality from allowing pair splitting. We end this section by proving an upper bound on the commonality for the two problems.

4.1. Examples and Insights

In §3, we proved that the maximum commonality is the same on trees, with or without pair splitting. We first observe that this result does not hold for general graphs. Consider the odd cycle shown in Figure 2: Figure 2(a) shows an optimum assignment without any pair splitting (i.e., problem MCAP^n), whereas an optimum assignment with pair splitting (i.e., MCAP^s) is shown in Figure 2(b).

In light of the above example, an immediate question that arises is What is the magnitude of the increase in commonality from allowing pair splitting? Consider the system of $3m+1$ modules, $m \geq 1$, shown in Figure 3. For $T=2$, the maximum commonality for problem MCAP^n is $2m+2$ (Figure 3(a)), whereas the maximum commonality for problem MCAP^s is $3m+1$ (Figure 3(b)). Thus, in this case, allowing pair splitting increases the maximum commonality by $m-1$; clearly, this increase is unbounded as $m \to \infty$. We record these two observations in the remark below.

Figure 3 **Increase in Commonality Due to Pair Splitting**

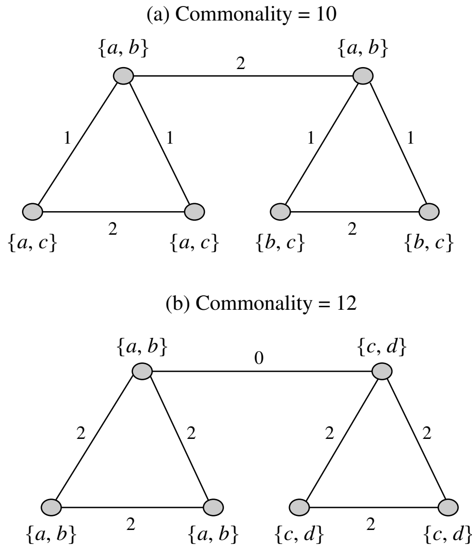


Remark 1. On general graphs, the maximum commonality with pair splitting can be strictly greater than that without pair splitting. Furthermore, the difference in the optimum values of MCAP^s and MCAP^n can grow without bound with the size of the graph. However, as will be shown in Theorem 3 later, the ratio of the optimum value of MCAP^s to that of MCAP^n is bounded above by $\frac{3}{2}$.

The two examples above hint at a relationship between the two commonality problems for $T=2$ and the classical 1-matching problem. Later, in §§5.1 and 6.1, we define this relationship precisely.

Another interesting question relates to the number of developers required by an assignment that achieves the maximum commonality. Because (i) the two developers assigned to a node have to be distinct, and (ii) an individual developer can be assigned to at most T nodes, the minimum number of developers required by *any* feasible solution is $\lceil 2n/T \rceil$. Intuitively, because commonality is a measure of the number of common developers at pairs of nodes that share an edge, the maximum commonality is expected to be achieved by some assignment that uses the min-

Figure 4 Effect of Increasing the Number of Developers on Commonality



imum number of developers. The graph in Figure 4 has six nodes (i.e., $n = 6$); with $T = 4$, we require a minimum of $\lceil 2n/T \rceil = 3$ developers. An assignment of maximum commonality with three developers is shown in Figure 4(a). A better assignment that uses four developers is shown in Figure 4(b). Thus, commonality can increase by using more than the minimum number of developers required. Similar examples can be shown to illustrate that commonality is not necessarily maximized by using the minimum number of distinct developer *pairs*. In Figure 2(a), the number of distinct developer pairs is three, whereas this number in Figure 2(b) is five. Similarly, the number of distinct developer pairs in Figure 3(a) is $2m$, whereas this number is $3m-1$ in Figure 3(b). We summarize this discussion in our next remark.

Remark 2. In general, the maximum commonality is not necessarily achieved by using the minimum possible number of distinct developers or developer pairs. However, for a complete graph, the maximum commonality is achievable by an assignment that uses the minimum possible number of distinct developers.

The behavior of the marginal increase in maximum commonality corresponding to a unit increase in the value of the load-balancing constant T provides a useful input in examining the trade-off between commonality and administrative and scheduling issues related to an increased team size. Intuitively, one would expect this marginal increase to be a nonincreasing function of T. This, however, is not always the case. In other words, commonality is not always a concave function of T. To illustrate, consider problem MCAP^n on the graph of Figure 2. For $T = 2, 3, 4$, and 5, it is easy to verify that the respective maximum commonalities are 4, 6, 6, and 10. For problem MCAP^s, these values are 5, 6, 7, and 10, respectively. Depending on the structure of G, such "sudden jumps" in the maximum commonality may be much more pronounced.

Remark 3. In general, the marginal increase in the maximum commonality corresponding to a unit increase in T is neither a nonincreasing nor a nondecreasing function of T.

Given the correspondence between T and the completion time of the project (§2), Remark 3 offers the following managerial insight: In some cases, a project can be tightened without sacrificing any commonality. In other cases, a project can be tightened with only a small reduction in commonality.

4.2. A Tight Bound on the Ratio of the Commonalities With and Without Pair Splitting

As illustrated in §4.1, the magnitude of the difference between the optimum values of problems MCAP^s and MCAP^n can grow without bound as the number of modules increases. A natural question, therefore, is whether there is an upper bound on the *ratio* of the optimum values of these two problems. For trees, this ratio is one (§3). An answer to this question for general graphs is of practical benefit to managers because it bounds the percentage increase in commonality from allowing pair splitting. As discussed in §3, practitioners strive to balance the benefits of pair splitting with the corresponding increase in pair formation effort. Thus, Theorem 3 below should be helpful in trading-off the costs and benefits of pair splitting.

As before, we denote the optimum values of problems MCAP^s and MCAP^n on an arbitrary graph $G(V, E)$ by $\text{MCAP}^{s*}_{\text{gen}}$ and $\text{MCAP}^{n*}_{\text{gen}}$, respectively. As in the proof of Theorem 2, we will need the one-developer problem (i.e., problem ONED; see §3.2) in an intermediate step; the optimum value of this problem is denoted by $\text{ONED}^*_{\text{gen}}$. We now state and prove our main result of this section.

Theorem 3. *$\text{MCAP}^{s*}_{\text{gen}} \le 1.5\text{MCAP}^{n*}_{\text{gen}}$. In other words, allowing for pair splitting can increase the maximum commonality in general graphs by at most* 50%. *This bound is tight.*

Proof. We begin the proof by stating a straightforward relationship between problems MCAP^n and ONED. A feasible solution to problem MCAP^n can be converted into a feasible solution to problem ONED by replacing each disjoint pair of developers with a single developer, and vice versa. Thus,

$$\text{MCAP}^{n*}_{\text{gen}} = 2\text{ONED}^*_{\text{gen}}. \tag{1}$$

We now show that

$$\text{MCAP}^{s*}_{\text{gen}} \le 3\text{ONED}^*_{\text{gen}}. \tag{2}$$

The main idea behind the proof is easy to explain: The commonality of an arbitrary solution to problem MCAP^s on G can be expressed as the sum of the commonalities of three feasible solutions of problem ONED on G. To show this, we use three distinct copies G_1, G_2, and G_3, of G.

Consider an optimum solution x^* of MCAP^s. We will express, via the construction below, the total commonality of x^* as the sum of the commonalities in G_i, $i = 1, 2, 3$; the assignments in G_i, $i = 1, 2, 3$, are each feasible for problem ONED. To analyze x^*, we need the following notation:

- $D = \{d_i, i = 1, \ldots, k\}$: the set of developers used in x^*.
- $S_{d_i}(V^{d_i}, E^{d_i})$, $i = 1, 2, \ldots, k$: the subgraph of developer d_i, $i \in D$. Thus, $V^{d_i} \subseteq V$ is the set of nodes that have d_i as one of the assigned developers; $E^{d_i} \subseteq E$ is the set of edges that have d_i assigned to both their endpoints.
- $B(i)$: set of developers assigned to node i, $i \in V$; note that $|B(i)| = 2$.
- $E^{d_i d_j}$, $i \neq j$: $\{(p, q): p \in V^{d_i}, q \in V^{d_j}\}$.
- $F^{d_i d_j}$, $i \neq j$: $\{(p, q): p \in V^{d_i}, q \in V^{d_j}, B(p) \cap B(q) = \{r\}, r \neq i, r \neq j\}$. Thus, if $(p, q) \in F^{d_i d_j}$, then the developer assignments at nodes p and q are $\{d_i, d_r\}$ and $\{d_j, d_r\}$, respectively. It follows that $F^{d_i d_j} \subseteq E^{d_i d_j}$.

If x^* uses three (or fewer) developers, say d_i, $i = 1, 2, 3$, then the result is trivial: assign developer d_i to the nodes V^{d_i} in G_i, $i = 1, 2, 3$. We therefore assume $|D| \geq 4$. As explained earlier in the proof of Theorem 2, we assume without loss of generality that each subgraph $S_{d_i}(V^{d_i}, E^{d_i})$, $i = 1, 2, \ldots, k$, is connected.

Initialization: Consider the subgraphs $S_{d_1}(V^{d_1}, E^{d_1})$ and $S_{d_2}(V^{d_2}, E^{d_2})$ of developers d_1 and d_2, respectively. We make the following assignments:

(a) Assign developer d_1 to all nodes in V^{d_1} in G_1.

(b) Assign developer d_2 to all nodes in V^{d_2} in G_2.

(c) If $F^{d_1 d_2} \neq \varnothing$, then do the following for each edge in $F^{d_1 d_2}$: Consider $(p, q) \in F^{d_1 d_2}$. In particular, let $B(p) \cap B(q) = \{r\}$, $r \neq 1$, $r \neq 2$. Assign developer d_r to nodes p and q in G_3.

Iterative step for $k \geq 3$: The idea is to first recover the commonality of edges in $F^{d_j d_k}$, $j = 1, 2, \ldots, k-1$. We then address the edges in E^{d_k} whose commonality has not been recovered by previous steps.

- Assignment for edges in $F^{d_j d_k}$, $j = 1, 2, \ldots, k-1$: consider $(p, q) \in F^{d_j d_k}$. Thus, $B(p) \cap B(q) = \{r\}$. If $r \in \{1, 2, \ldots, j-1, j+1, \ldots, k-1\}$, then the commonality of this edge was already recovered in the earlier steps. Otherwise, $r \geq k+1$. Furthermore, there are no assignments to nodes p and q in G_3. Assign developer d_r to nodes p and q in G_3.
- Consider the subgraph $S_k(V^{d_k}, E^{d_k})$ of developer d_k. Thus, each node in V^{d_k} has been assigned developer d_k.

1. Remove from E^{d_k} all the edges whose commonality has been completely accounted for in earlier steps. For example, the commonality of edges $(p, q) \in E^{d_k}$ with $B(p) = \{d_i, d_3\}$ and $B(q) = \{d_j, d_3\}$, $i \leq k-1$, $j \leq k-1$, $i \neq j$ was already recovered, and therefore can be removed.

2. The remaining edges in E^{d_k} are of the following four types.

Case A: (p, q) with $B(p) = \{d_i, d_k\}$, $B(q) = \{d_i, d_k\}$, $i \leq k-1$. Then, developer d_i was assigned (by an earlier step) to both nodes p and q in G_3 or exactly one of G_1 or G_2, say G_1. Then, one of the following two cases must occur: (i) There are no developer assignments yet to nodes p and q in G_2. In this case, we assign developer d_k to nodes p and q in G_2. (ii) d_k is assigned either at nodes p or q or at both of these nodes in G_2. If d_k has been assigned at both nodes p and q of G_2, then we have already recovered the commonality of (p, q) due to developer d_k. Otherwise, if d_k has been assigned at, say, node p of G_2, then there are two possibilities: If there are no developer assignments yet at node q of G_2, then we assign d_k at node q of G_2. Otherwise, observe that there are no assignments yet to both nodes p and q in G_3. We therefore assign d_k at nodes p and q in G_3.

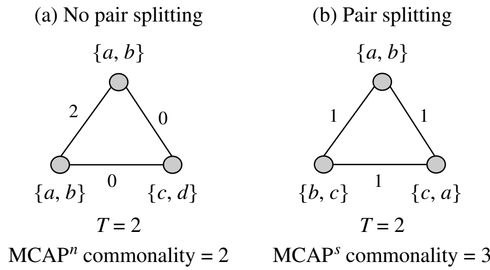
Case B: (p, q) with $B(p) = \{d_i, d_k\}$, $B(q) = \{d_j, d_k\}$, $i \leq k-1$, $j \geq k+1$. Then, developer d_i was assigned (by an earlier step) to node p in exactly one of G_1 or G_2, say G_1. Then, one of the following two cases must occur: (i) There are no developer assignments yet to nodes p and q in G_2. In this case, we assign developer d_k to nodes p and q in G_2. (ii) d_k is assigned either at nodes p or q or at both of these nodes in G_2. The argument is now similar to that in (ii) of Case A above.

Case C: (p, q) with $B(p) = \{d_r, d_k\}$, $B(q) = \{d_r, d_k\}$, $r \geq k+1$. Then, one of the following cases must occur: (i) There are no developer assignments yet to both nodes p and q in G_1 and G_2. In this case, we arbitrarily assign d_k to nodes p and q in G_1. (ii) In exactly one of G_1 or G_2, say G_1, developer d_k has been assigned either at nodes p or q or at both of these nodes. If d_k has been assigned at both nodes p and q of G_1, then we have already recovered the commonality of (p, q) due to developer d_k. Otherwise, if d_k has been assigned at, say, node p of G_1, then note that there are no developer assignments yet at node q of G_1. We therefore assign d_k at node q of G_1.

Case D: (p, q) with $B(p) = \{d_w, d_k\}$, $B(q) = \{d_r, d_k\}$, $w \geq k+1$, $r \geq k+1$, $w \neq r$. The argument is similar to that in Case (C) above.

It follows from (1) and (2) that $\text{MCAP}^{s*}_{\text{gen}} \leq 1.5 \cdot \text{MCAP}^{n*}_{\text{gen}}$. To show that this bound is tight, we consider the triangle in Figure 5. Here, we have $\text{MCAP}^{s*}_{\text{gen}} = 3$ and $\text{MCAP}^{n*}_{\text{gen}} = 2$; therefore, $\text{MCAP}^{s*}_{\text{gen}} = 1.5\text{MCAP}^{n*}_{\text{gen}}$. The example in Figure 3 also proves the asymptotic tightness of the bound; here we have

Figure 5 An Example Showing That the Bound in Theorem 3 Is Tight



$\text{MCAP}^{s*}_{\text{gen}}/\text{MCAP}^{n*}_{\text{gen}} = (3m+1)/(2m+2) \to 1.5$ as $m \to \infty$.

4.3. An Upper Bound on Commonality

To be able to assess the quality of feasible solutions to problems MCAP^n and MCAP^s on general graphs, we need an upper bound on their maximum commonality. From our discussion thus far, we have

$$\text{MCAP}^{n*}_{\text{gen}} \le \text{MCAP}^{s*}_{\text{gen}} \le 1.5\text{MCAP}^{n*}_{\text{gen}}. \quad (3)$$

Depending on the structure of $G(V, E)$ and the value of the load-balancing bound T, we can have equality at either of the two inequalities in (3). We state a common upper bound below.

Lemma 1. *For either problem MCAP^n or problem MCAP^s, an upper bound on the maximum commonality of G, $C(G)$ is given by*

$$C(G) \le \min\left\{2|E|, \sum_{i\in V} \min\{(T-1), |\delta_i|\}\right\},$$

where $\delta_i = \{e \in E\colon e = (i, j), j \in V\}$.

Proof. Because $C_e \le 2\ \forall e \in E$, we have

$$C(G) = \sum_{e\in E} C_e \le 2|E|. \quad (4)$$

Because a developer can be assigned to at most T modules, the total commonality of all the edges incident on a node i can be at most $2(T-1)$. Clearly, this commonality is also bounded by twice the number of edges incident on node i. Therefore,

$$\sum_{e\in\delta(i)} C_e \le \min\{2(T-1), 2|\delta_i|\}. \quad (5)$$

Because

$$C(G) = \tfrac{1}{2}\sum_{i\in V}\sum_{e\in\delta(i)} C_e,$$

we have

$$C(G) \le \sum_{i\in V} \min\{(T-1), |\delta_i|\}. \quad (6)$$

Inequalities (4) and (6) together imply the desired upper bound. □

We now turn our attention to analyzing problems MCAP^n and MCAP^s on general graphs. When $T = 2$, these problems are closely related to the classical matching problem on general graphs—Problem MCAP^n to the integer version of the matching problem, and problem MCAP^s to the corresponding linear programming (LP) relaxation.

5. Solving MCAP^n on General Graphs

We begin by observing that problem MCAP^n is polynomially solvable for $T = 2$. Then, we show that the corresponding decision problem is strongly NP-complete for $T = 3$. Finally, we derive a $\frac{2}{3}$-approximation algorithm for $T = 3$. In the following, we let $C^*(G)$ denote the maximum commonality of G.

For $T = 2$, problem ONED (§3.2) is equivalent to the classical maximum cardinality 1-matching problem. Also, recall from §3.2 that the value of the optimum solution of problem MCAP^n equals twice that of problem ONED. Thus, for $T = 2$, the value of the optimum solution of problem MCAP^n equals $2|M^*|$, where M^* is a maximum cardinality matching in G. Given M^*, an optimum assignment can be obtained by assigning a developer pair $\{a_e, b_e\}$ at the two endpoints of an edge $e \in M^*$. Thus, problem MCAP^n is solvable in time $O(\sqrt{|V|}|E|)$ for $T = 2$ because a maximum cardinality matching can be obtained in time $O(\sqrt{|V|}|E|)$ (Micali and Vazirani 1980).

5.1. MCAP^n with $T \ge 3$

We first show the hardness of problem MCAP^n for $T = 3$, and then describe a polynomial-time $\frac{2}{3}$-approximation algorithm.

Theorem 4. *The decision problem corresponding to MCAP^n is strongly NP-complete for $T = 3$.*

For our reduction in the proof of Theorem 4, we choose the three-dimensional matching problem, which is a well-known strongly NP-complete problem (Garey and Johnson 1979, Karp 1972). For brevity, the details of the proof are given in the e-companion to this paper.[2]

5.1.1. A $\frac{2}{3}$-Approximation Algorithm for $T = 3$. Consider an arbitrary feasible assignment, x, of pairs of developers to modules. Then, from the proof of Lemma 1, this assignment satisfies $\sum_{e\in\delta(i)} C_e \le 2(T-1) = 4\ \forall i \in V$. Also, because $C_e \in \{0, 2\}$, we have that $C_e/2 \in \{0, 1\}$. The vector $\langle C_e/2, e \in E\rangle$ is, therefore,

[2] An electronic companion to this paper is available as part of the online version that can be found at http://mansci.journal.informs.org/.

a feasible solution to the classical *2-matching problem* defined below.

$$D^*(G) = \max \sum_{e \in E} d_e$$
$$\sum_{e \in \delta(i)} d_e \le 2 \quad \forall\, i \in V$$
$$d_e \in \{0, 1\} \quad \forall\, e \in E. \tag{7}$$

Because x is an arbitrary assignment, it follows that

$$C^*(G) \le 2D^*(G). \tag{8}$$

Consider an optimum solution $\langle d_e^*, e \in E\rangle$ to the 2-matching problem (7) above, and let $E' \subseteq E$ be defined by $E' = \{e \in E\colon\ d_e^* = 1\}$. Thus, $D^*(G) = |E'|$. From (8), we have

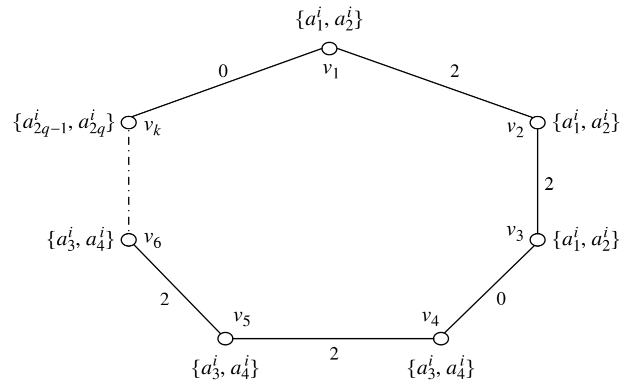
$$C^*(G) \le 2|E'|. \tag{9}$$

We now obtain a feasible solution to problem MCAP^n with a commonality of at least $\frac{2}{3}(2|E'|) - 2 = \frac{4}{3}|E'| - 2$. We consider the subgraph S_G (of G) induced by E'. The degree of each node in S_G is at most two. Therefore, the nodes of S_G can be partitioned into a set of disjoint paths, say P_i, $i = 1, \ldots, p$, and a set of disjoint cycles, say $C_1, \ldots, C_l$. Let $|P_i|$ (respectively, $|C_j|$) denote the number of nodes in path P_i (respectively, cycle C_j). Following this path-cycle decomposition, we propose a specific assignment of developers for the paths and cycles in the decomposition.

Step A (Paths). Let $P_i = v_1 - v_2 - v_3 - \cdots - v_k$. We use $q = \lceil k/3 \rceil$ disjoint pairs of developers: $\{a_1^i, a_2^i\}$, $\{a_3^i, a_4^i\}, \ldots, \{a_{2q-1}^i, a_{2q}^i\}$. Each developer pair is assigned to three consecutive nodes in the path. Thus, the developer pair $\{a_1^i, a_2^i\}$ is assigned to modules v_1, v_2, and v_3; the pair $\{a_3^i, a_4^i\}$ is assigned to nodes v_4, v_5, and v_6, and so on. Figure 6 demonstrates this assignment. Because P_i contains k nodes (and, hence, $k-1$ edges), the commonality of exactly $\lfloor (k-1)/3 \rfloor$ edges of P_i is zero, and the commonality of the remaining (i.e., $k - 1 - \lfloor (k-1)/3 \rfloor$) edges is two. The total commonality of the edges of P_i is, therefore, $2(k-1) - 2\lfloor (k-1)/3 \rfloor$.

Step B (Cycles). Let $C_j = v_1 - v_2 - v_3 - \cdots - v_k - v_1$. Thus, C_j contains k nodes (and k edges). We again use $q = \lceil k/3 \rceil$ disjoint pairs of developers: $\{a_1^i, a_2^i\}$, $\{a_3^i, a_4^i\}, \ldots, \{a_{2q-1}^i, a_{2q}^i\}$. As with paths, each developer pair is assigned to three consecutive nodes in the

Figure 7 An Assignment of Developers for a Cycle C_j in the Path-Cycle Decomposition of S_G



cycle. Thus, the developer pair $\{a_1^i, a_2^i\}$ is assigned to modules v_1, v_2, and v_3; the pair $\{a_3^i, a_4^i\}$ is assigned to nodes v_4, v_5, and v_6, and so on. Figure 7 demonstrates this assignment. With this assignment, it is easy to verify that (i) for $k = 3$, all the edges have commonality two, and (ii) for $k \ge 4$, the commonality of exactly $\lceil k/3 \rceil$ edges is zero; the commonality of the remaining (i.e., $k - \lceil k/3 \rceil$) edges is two. The total commonality of the edges of C_j is, therefore, at least $2k - 2\lceil k/3 \rceil$.

We now evaluate the quality of this assignment. Let $C(G)$ denote the commonality achieved by this assignment. Our discussion above implies that

$$C(G) \ge 2|E'| - 2\left\lceil \frac{|E'|}{3} \right\rceil \ge \frac{4}{3}|E'| - 2.$$

This, and (9), imply that $C(G) \ge \frac{2}{3}C^*(G) - 2$. A formal description of the algorithm, referred to as Algorithm Approx, is presented below.

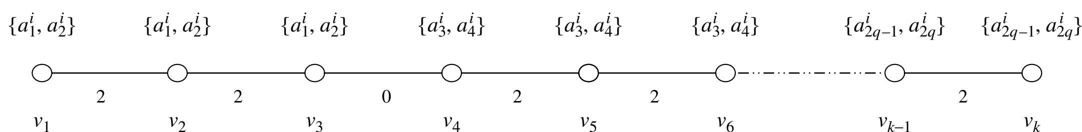
Algorithm Approx

Input. An undirected graph $G(V, E)$, where V represents the modules and the edge set E represents the connections between the modules.

Step 1. Solve the maximum 2-matching problem (7) on G, and let $\langle d_e^*, e \in E\rangle$ denote an optimum solution to this problem. Let $E' \subseteq E$ be defined by $E' = \{e \in E\colon d_e^* = 1\}$, and consider the subgraph S_G induced by E'.

Step 2. Construct a path-cycle decomposition of S_G. Let P_i, $i = 1, 2, \ldots p$, be the paths, and C_j, $j = 1, 2, \ldots, l$, be the cycles in this decomposition.

Figure 6 An Assignment of Developers for a Path P_i in the Path-Cycle Decomposition of S_G



Step 3. For each path P_i, $i = 1, 2, \ldots, p$, assign developers to the nodes in P_i as described in Step A above. For each cycle C_j, $j = 1, 2, \ldots, l$, assign the developers as described in Step B above.

Our discussion preceding the description of Algorithm Approx implies the following result.

Theorem 5. *Algorithm* Approx *is a* $\frac{2}{3}$*-approximation for problem MCAP*^{n} *with* $T = 3$.

6. Solving MCAP^s on General Graphs

As mentioned at the end of §4.3, the special case of problem MCAP^s with $T = 2$ is closely related to the LP relaxation of the matching problem. We start by examining this relationship, and the consequent polynomial-time algorithm. Proving the hardness of MCAP^s for $T = 3$ turns out to be a nontrivial exercise; §6.2 provides a proof. Finally, in §6.2.1, we obtain a polynomial-time $\frac{1}{2}$-approximation for MCAP^s with $T = 3$.

6.1. MCAP^s with $T = 2$

We let MCAP$^s_{T=2}$ denote the special case of MCAP^s with $T = 2$. Given $G(V, E)$, consider the following problem defined on variables b_e, $e \in E$.

$$B^*(G) = \max \sum_{e\in E} b_e$$
$$\sum_{e\in\delta(i)} b_e \le 1 \quad \forall\, i \in V$$
$$0 \le b_e \le 1 \quad \forall\, e \in E. \qquad (10)$$

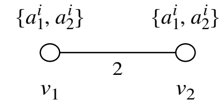
Problem (10) is the LP relaxation of the classical 1-matching problem on G. Let b^*_e, $e \in E$ denote an optimum solution of problem (10), and $C^*(G)$ be the optimum value of MCAP^s. For $T = 2$, (5) gives $\sum_{e\in\delta(i)} C_e \le 2\ \forall\, i \in V$. Therefore, $\sum_{e\in\delta(i)} C_e/2 \le 1$. Thus, if vector $\langle C_e, e \in E\rangle$ is feasible for MCAP^s, then vector $\langle b_e = C_e/2,\ e \in E\rangle$ is feasible for problem (10). Therefore,

$$C^*(G) \le 2B^*(G) = \sum_{e\in E} 2b^*_e. \qquad (11)$$

A well-known property of the LP relaxation (10) (see, e.g., Garfinkel and Nemhauser 1972, Nemhauser and Wolsey 1988) is that $b^*_e \in \{0, \frac{1}{2}, 1\}\ \forall\, e \in E$; thus, $2b^*_e \in \{0, 1, 2\}$. We will now demonstrate a feasible assignment for MCAP^s with $C^*(G) = 2B^*(G)$. The optimality of this assignment follows immediately from (11).

Consider the induced subgraph G_{b^*} of G corresponding to $\tilde{E} = \{e \in E{:}\ b^*_e = \frac{1}{2} \text{ or } b^*_e = 1\}$. The degree of each node in G_{b^*} is at most two, and therefore $\tilde{E}$ can be partitioned into a set of disjoint maximal paths, say P_i, $i = 1, \ldots, p$, and a set of disjoint cycles, say C_j, $j = 1, \ldots, l$. The assignments for paths and cycles are now described below.

Figure 8 Assignment of Developers for a Path with One Edge in G_{b^*}



Step A (Paths with one edge). Let $P_i = v_1 - v_2$, and $e = (v_1, v_2)$. The optimality of b^* implies that $b^*_e = 1$. As shown in Figure 8, we assign the same pair of developers to both nodes v_1 and v_2 of P_i. Thus, the commonality of e equals 2 ($=2b^*_e$).

Step B (Paths with more than one edge). Let $P_i = v_1 - v_2 - v_3 - \cdots - v_k$. Again, the optimality of b^* implies that $b^*_e = \frac{1}{2}$ for each edge $e = (v_j, v_{j+1})$, $j = 1, 2, \ldots, k-1$, of P_i. We use $q = k + 1$ pairs of developers: $\{a^i_1, a^i_2\}, \{a^i_2, a^i_3\}, \{a^i_3, a^i_4\}, \ldots, \{a^i_{q-1}, a^i_q\}$. The developer pair $\{a^i_r, a^i_{r+1}\}$, $r = 1, \ldots, k$, is assigned to node v_r of P_i. This assignment is shown in Figure 9. It is easy to see that the commonality of each edge $e = (v_j, v_{j+1})$, $j = 1, 2, \ldots, k-1$, of P_i equals 1 ($=2b^*_e$).

Step C (Cycles). Let $C_j = v_1 - v_2 - v_3 - \cdots - v_k - v_1$. Thus, C_j contains k nodes (and k edges). We use $q = k$ pairs of developers: $\{a^i_1, a^i_2\}, \{a^i_2, a^i_3\}, \{a^i_3, a^i_4\}, \ldots, \{a^i_q, a^i_1\}$. The developer pair $\{a^i_r, a^i_{r+1}\}$, $r = 1, \ldots, k-1$ is assigned to node v_r, and the pair $\{a^i_k, a^i_1\}$ is assigned to node v_k. Figure 10 demonstrates this assignment. Again, the commonality of each edge $e \in \{(v_r, v_{r+1}),\ r = 1, 2, \ldots, k-1\} \cup \{(v_k, v_1)\}$ of C_j is 1 ($=2b^*_e$).

Thus, by construction, the total commonality of the above assignment for MCAP^s satisfies $C(G) = 2B^*(G)$, and is therefore optimal. A formal description of the algorithm, referred to as Algorithm MCAP$^s_{T=2}$, is now straightforward.

Algorithm MCAP$^s_{T=2}$

Input. An undirected graph $G(V, E)$, where V represents the modules and the edge set E represents the connections between the modules.

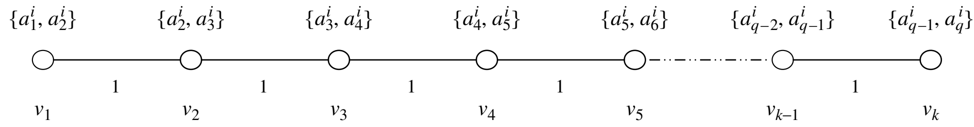
Step 1. Solve the LP relaxation of 1-matching problem (problem (10)) on G, and let b^*_e, $e \in E$ denote an optimum solution. Let $\tilde{E} \subseteq E$ be defined by $\tilde{E} = \{e \in E{:}\ b^*_e = \frac{1}{2} \text{ or } b^*_e = 1\}$, and consider the subgraph G_{b^*} induced by $\tilde{E}$.

Step 2. Construct a path-cycle decomposition of G_{b^*}. Let P_i, $i = 1, 2, \ldots, p$, denote the paths, and C_j, $j = 1, 2, \ldots, l$, denote the cycles in this decomposition.

Step 3. For each path P_i, $i = 1, 2, \ldots, p$, assign developers to the nodes in P_i as described in Step A or Step B above, depending, respectively, on whether the path has one or more edges. For each cycle C_j, $j = 1, 2, \ldots, l$, assign developers to the nodes as described in Step C above.

The following theorem formally states the result obtained above.

Figure 9 Assignment of Developers for a Path with Multiple Edges in G_{b^*}



Theorem 6. *Algorithm $MCAP^s_{T=2}$ provides an optimum solution to problem $MCAP^s_{T=2}$.*

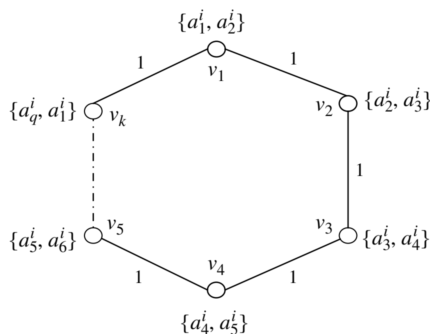
Let $\text{MCAP}^{s*}_{T=2}$ (respectively, $\text{MCAP}^{n*}_{T=2}$) denote the optimum value of problem $\text{MCAP}^s_{T=2}$ (respectively, problem $\text{MCAP}^n_{T=2}$). Our analysis in §§5 and 6.1 illustrates some interesting connections with the classical matching problem: For $T = 2$, the maximum commonality of problem MCAP^n equals twice the cardinality of a maximum matching in G; for problem MCAP^s, the maximum commonality equals twice the optimum value of the LP relaxation (10) of the matching problem. Thus, the increase in commonality from allowing pair splitting is exactly twice the integrality gap (i.e., the difference in the optimum values of the LP (10) and its integer counterpart) of the matching problem. If G is bipartite, then this integrality gap is zero.

Corollary 3. *For $T = 2$, the difference in the maximum commonalities of problems $MCAP^s$ and $MCAP^n$ is exactly twice the difference in the optimum value of the LP* (10) *and the maximum cardinality of a matching in G. In particular, if G is bipartite, then this difference is zero.*

6.2. Problem MCAP^s with $T \geq 3$

Recall from the proof of Theorem 4 that it was straightforward to reduce an arbitrary instance of a three-dimensional matching problem to a specific instance of problem MCAP^n with $T = 3$. Much of the simplicity in that proof was due to the fact that $C_e \in \{0, 2\}$. Thus, the absence of pair splitting implies that all nonzero commonalities have value of two. For MCAP^s, however, $C_e \in \{0, 1, 2\}$, and it takes a nontrivial construction to ensure that a solution of MCAP^s results in a three-dimensional matching.

Figure 10 Assignment of Developers for a Cycle in G_{b^*}



We first prove the hardness of problem MCAP^s for $T \geq 3$, and then discuss two polynomial-time approximations for $T = 3$.

Theorem 7. *The decision problem corresponding to $MCAP^s$ is strongly NP-complete for $T = 3$.*

We again choose the three-dimensional matching problem (§5.1) for our reduction in the proof of Theorem 7. The details of the proof are again given in the e-companion to this paper.

6.2.1. Approximation Algorithms for $T = 3$.

A $\frac{4}{9}$-approximation algorithm is easy to obtain from our earlier results. Algorithm Approx (see §5.1.1) is a $\frac{2}{3}$-approximation for MCAP^n with $T = 3$ (Theorem 5). If we denote the commonality given by Algorithm Approx as $\text{MCAP}^{\text{approx}}_{\text{gen}}$, then $\text{MCAP}^{\text{approx}}_{\text{gen}} \geq \frac{2}{3}\text{MCAP}^{n*}_{\text{gen}}$. Moreover, Theorem 3 in §4.2 implies that $\text{MCAP}^{n*}_{\text{gen}} \geq \frac{2}{3}\text{MCAP}^{s*}_{\text{gen}}$. Therefore, $\text{MCAP}^{\text{approx}}_{\text{gen}} \geq \frac{4}{9}\text{MCAP}^{s*}_{\text{gen}}$. We note the result below.

Theorem 8. *Algorithm* Approx *is a $\frac{4}{9}$-approximation for problem $MCAP^s$ with $T = 3$.*

Next, we present a $\frac{1}{2}$-approximation algorithm.

Theorem 9. *Algorithm $MCAP^s_{T=2}$ is a $\frac{1}{2}$-approximation for $MCAP^s$ with $T = 3$.*

Proof. For $T = 3$, (5) implies that $\sum_{e\in\delta(i)} C_e \leq 4$ $\forall\, i \in V$ in an optimum solution of MCAP^s. Thus, if vector $\langle C_e^*, e \in E\rangle$ is an optimum solution for MCAP^s, then the vector $\langle b_e = C_e^*/4, e \in E\rangle$ is feasible for problem (10). Denote the optimum value of MCAP^s for $T = 2$ (respectively, $T = 3$) by $\text{MCAP}^{s*}_{T=2}$ (respectively, $\text{MCAP}^{s*}_{T=3}$). We therefore have

$$\frac{\text{MCAP}^{s*}_{T=3}}{4} \leq B^*(G).$$

Our result in §6.1 implies that $\text{MCAP}^{s*}_{T=2} = 2B^*(G)$. Then, we have

$$\text{MCAP}^{s*}_{T=2} \geq \frac{\text{MCAP}^{s*}_{T=3}}{2}.$$

Recall that Algorithm $\text{MCAP}^s_{T=2}$ provides an optimum solution for MCAP^s with $T = 2$ (see Theorem 6). Because this solution is feasible for $T = 3$, the result follows. □

Corollary 4. *For problem $MCAP^s$, commonality can increase by at most* 100% *when T increases from two to three.*

The bound in Corollary 4 is tight: If G is a triangle, then $\text{MCAP}^{s*}_{T=2} = 3$ and $\text{MCAP}^{s*}_{T=3} = 6$. Thus, in addition to providing an approximation for MCAP^s with $T = 3$, Theorem 9 also presents a bound on the increase in commonality when T increases from two to three. As mentioned earlier in §4.1, such results on a marginal increase in commonality are useful tools for managers as they decide the team size and incorporate load-balancing considerations.

7. Conclusions and Future Research Directions

Many of the benefits of pair programming—better quality of the code and the consequent reduction in the amount of testing required, reduction in the effort required to integrate the various modules of the system, enhanced knowledge transfer and learning—improve with the number of common developers for pairs of connected modules in a software system. This paper introduces the notion of commonality, and analyzes two variants of the problem of obtaining a pair-programming assignment of developers to modules with the objective of maximizing the commonality. The two variants, MCAP^n and MCAP^s, differ in whether or not pair splitting—a practice in which a developer is allowed to pair with different partners during the project—is allowed. For both of the variants, a load-balancing constraint imposes that each individual developer can be assigned to at most T different modules. This constraint approximately controls the completion time of the project.

When the underlying graph is a tree, we show that the value of the maximum commonality is identical for problems MCAP^n and MCAP^s. As a by-product, we also obtain polynomial-time algorithms for both of these problems. For general graphs, we obtain a tight upper bound on the percentage increase in commonality from allowing pair splitting by showing that the value of the maximum commonality for MCAP^s is at most $\frac{3}{2}$ times that of MCAP^n. For $T = 2$, we exploit the precise relationship of MCAP^s and MCAP^n with the linear and integer versions, respectively, of the classical 1-matching problem to provide polynomial-time algorithms. We then show that the decision problems corresponding to problems MCAP^n and MCAP^s are strongly NP-complete for $T = 3$. For $T = 3$, we use the 1-matching and 2-matching problems to provide two polynomial-time approximations—a $\frac{2}{3}$-approximation for MCAP^n, and a $\frac{1}{2}$-approximation for MCAP^s.

To the best of our knowledge, this paper is the first attempt at an algorithmic analysis of developer assignments in pair programming. There are a number of directions in which future work can proceed. We offer the following suggestions:

- As a natural extension of our results on trees, analyzing problems MCAP^n and MCAP^s on bipartite graphs is an interesting and challenging direction. Theorem 2 does not hold for bipartite graphs. A simple counterexample is a cycle of length 4 with $T = 3$; here, the optimum solution value of problem MCAP^n is four and that of problem MCAP^s is five. Problem MCAP^n belongs to a larger class of "edge-deletion" problems for hereditary and monotone graph properties (Yannakakis 1978). Whereas the general class of edge-deletion problems on bipartite graphs is NP-complete (Yannakakis 1981, Natanzon et al. 2001), the complexity status of problem MCAP^n is open. However, a recent result of Alon et al. (2005) implies a linear-time algorithm that approximates the optimum solution to problem MCAP^n within a small additive error.
- The notion of commonality in this paper is derived from pairwise interactions between modules. That is, the total commonality of the system is obtained from the commonality between pairs of interacting modules. A relevant generalization is to consider the interaction between *sets* of modules. Thus, an "edge" in the underlying graph corresponds to a subset of modules; commonality for a subset is achieved when all the modules in that subset share one or more common developers. It is likely that such a generalization is related to matchings on hypergraphs (Berge 1973, Schrijver 2003), although an exact correspondence seems difficult to characterize.
- Using the developer assignments to obtain a daily schedule for the developers over a (given) planning horizon is an interesting problem. Note that the two developers assigned to a module have to work *together* as a team. Because a developer cannot simultaneously work on two modules, the question of obtaining a minimum-length feasible schedule becomes relevant. Apart from the developer assignments, a complete description of such a problem should also include the expected development time for each module.
- Neither of the variants of the commonality problem discussed in this paper impose any bound on the total number of developers used in an assignment. In practice, there might be situations where the size of the development team is a constant, is bounded from above, or is allowed to vary between a lower and an upper bound. Introducing such a constraint fundamentally alters the analysis, and is a direction worth exploring.
- Corollary 4 in §6.2.1 provides a simple example of the marginal increase in commonality resulting from an increase in the load-balancing bound from two to three. Generalizing this sensitivity analysis for

problems MCAP^s and MCAP^n will help to understand the structural behavior of commonality, and will help managers to incorporate the load-balancing considerations.

8. Electronic Companion

An electronic companion to this paper is available as part of the online version that can be found at http://mansci.journal.informs.org/.

Acknowledgments

The authors thank the three anonymous reviewers, the associate editor, and the department editor for their valuable comments and suggestions.

References

Alon, N., A. Shapira, B. Sudakov. 2005. Additive approximation for edge-deletion problems. *Proc. 46th Annual IEEE Sympos. Foundations Comput. Sci.* (*FOCS*), IEEE Computer Society, Washington, D.C., 419–428.

Ambler, S. W. 2002. *Agile Modeling*: *Effective Practices for Extreme Programming and the Unified Process*. John Wiley & Sons, New York.

Astels, D., G. Miller, M. Novak. 2002. *A Practical Guide to Extreme Programming.* Prentice Hall, Upper Saddle River, NJ.

Beck, K. 2000. *Extreme Programming Explained*: *Embrace Change*. Addison-Wesley, New York.

Beck, K., M. Fowler. 2001. *Planning Extreme Programming*. Addison-Wesley, New York.

Becker, R. I., S. R. Schach, Y. Perl. 1982. A shifting algorithm for min-max tree partitioning. *J. ACM* **29**(1) 58–67.

Berge, C. 1973. *Graphs and Hypergraphs*. North-Holland, Amsterdam.

Canfora, G., A. Cimitile, C. A. Visaggio. 2005. An empirical study on the productivity of pair programming. *Proc. 6th Internat. Conf. Extreme Programming and Agile Processes Software Engrg.*, Springer, New York, 92–99.

Constantine, L. L. 1995. *Constantine on Peopleware*. Yourdon Press, Englewood Cliffs, NJ.

Erdogmus, H., L. Williams. 2003. The economics of software development by pair programmers. *Engrg. Economist* **48**(4) 283–319.

Garey, M. R., D. S. Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York.

Garfinkel, R., G. L. Nemhauser. 1972. *Integer Programming*. John Wiley & Sons, New York.

Hadlock, F. 1974. Minimum spanning forests of bounded trees. *Proc. 5th Southeast Conf. Combinatorics, Graph Theory, and Comput., Boca Raton, FL,* 449–460.

Johar, M., S. Kumar, M. Dawande, V. Mookerjee. 2003. Optimizing the rotation of developers in extreme programming: A model and comparison. *Proc. 13th Annual Workshop Inform. Tech. Systems, Seattle, WA,* 97–102.

Karp, R. M. 1972. Reducibility among combinatorial computations. R. E. Miller, J. W. Thatcher, eds. *Complexity of Computer Computations*. Plenum Press, New York, 85–103.

Kundu, S., J. Misra. 1977. A linear tree partitioning algorithm. *SIAM J. Comput.* **6**(1) 151–154.

Kuppuswami, S., K. Vivekandanam, P. Ramaswamy, P. Rodrigues. 2003. The effects of individual XP practices on software development effort. *ACM SIGSOFT Software Engrg. Notes* **28**(6) 6–13.

Micali, S., V. Vazirani. 1980. An $O(\sqrt{|V|}.|E|)$ algorithm for finding maximum matching in general graphs. *Proc. 21st Annual Sympos. Foundations Comput. Sci., Syracuse, NY,* 21–27.

Natanzon, A., R. Shamir, R. Sharan. 2001. Complexity classification of some edge modification problems. *Discrete Appl. Math.* **113**(1) 109–128.

Nemhauser, G. L., L. A. Wolsey. 1988. *Integer Programming and Combinatorial Optimization*. John Wiley & Sons, New York.

Schrijver, A. 2003. *Combinatorial Optimization*: *Polyhedra and Efficiency*, Vol. C. Springer-Verlag, Berlin.

Shukla, A. 2002. Pair programming and the factors affecting Brooks' law. Master's thesis, North Carolina State University, Raleigh, NC.

Srikanth, H., L. Williams, E. Wiebe, C. Miller, S. Balik. 2004. On pair rotation in the computer science course. *Proc. 17th Conf. Software Engrg. Ed. Training*, IEEE Computer Society, Washington, D.C., 144–149.

Stewart, D. B. 2004. Twenty-five most common mistakes with real-time software development. *Proc. Embedded Systems Conf.* (*ESC*), *San Francisco, CA*, 1–13.

Williams, L., A. Shukla, A. I. Anton. 2004. An initial exploration of the relationship between pair programming and Brooks' law. *Proc. Agile Development Conf.*, IEEE Computer Society, Washington, D.C., 11–20.

Williams, L., R. Kessler, W. Cunningham, R. Jeffries. 2000. Strengthening the case for pair-programming. *IEEE Software* **17**(4) 19–25.

Wood, A., W. Kleb. 2002. Extreme programming in a research environment. *Proc. 2nd XP Universe and 1st Agile Conf. Extreme Programming and Agile Methods,* Springer-Verlag, London, 89–99.

Yannakakis, M. 1978. Node- and edge-deletion NP-complete problems. *Proc. 10th Annual ACM Sympos. Theory Comput.* (*STOC*), ACM Press, New York, 253–264.

Yannakakis, M. 1981. Edge-deletion problems. *SIAM J. Comput.* **10** 297–309.