ELSEVIER

Artificial Intelligence 126 (2001) 43–62

Artificial Intelligence

www.elsevier.com/locate/artint

Algorithm portfolios

Carla P. Gomes *, Bart Selman

Department of Computer Science, Cornell University, 4130 Upson Hall, Ithaca, NY 14853-7501, USA

Received 19 November 1999

Abstract

Stochastic algorithms are among the best methods for solving computationally hard search and reasoning problems. The run time of such procedures can vary significantly from instance to instance and, when using different random seeds, on the same instance. One can take advantage of such differences by combining several algorithms into a portfolio, and running them in parallel or interleaving them on a single processor. We provide an evaluation of the portfolio approach on distributions of hard combinatorial search problems. We show under what conditions the portfolio approach can have a dramatic computational advantage over the best traditional methods. In particular, we will see how, in a portfolio setting, it can be advantageous to use a more "risk-seeking" strategy with a high variance in run time, such as a randomized depth-first search approach in mixed integer programming versus the more traditional best-bound approach. We hope these insights will stimulate the development of novel randomized combinatorial search methods. © 2001 Published by Elsevier Science B.V.

Keywords: Algorithm portfolios; Randomized algorithms; Cost profiles; Anytime algorithms; Empirical evaluation

1. Introduction

Randomized algorithms are among the best current algorithms for solving computationally hard problem. Most local search methods for solving combinatorial optimization problems have a stochastic component, both to generate an initial candidate solution, as well as to choose among good local improvements during the search. One can also incorporate an element of randomness in the value and variable selection strategies of complete backtrack-style search methods. The run time of such randomized algorithms varies from

* Corresponding author.
E-mail addresses: gomes@cs.cornell.edu (C.P. Gomes), selman@cs.cornell.edu (B. Selman).

0004-3702/01/$ – see front matter
PII: S0004-3702(00)00081-3

run to run on the same problem instance, and therefore can be characterized by a probability distribution. The performance of algorithms can also vary dramatically among different problem instances. In this case, we want to consider the performance profile of the algorithms over a spectrum of problem instances.

Given the diversity in performance profiles among algorithms, various approaches have been developed to optimize the overall performance of algorithms, taking into account computational resource constraints. These considerations led to the development of anytime algorithms [6], decision theoretic metareasoning, and related approaches [15, 23].

Despite the numerous formal results obtained in these areas, so far they have not been exploited much by the traditional communities that study hard computational problems, such as operations research (OR), constraint satisfaction (CSP), theorem proving, and the experimental algorithms community. In order to bridge this gap, we analyze the performance profiles of several of the state-of-the-art search methods on distributions of hard search problems encoded as Constraint Satisfaction problems (CSP), and as Mixed Integer Programming problems (MIP). Our study reveals several interesting problem classes where a portfolio approach gives a dramatic improvement in terms of overall performance, compared to a single algorithm approach. In addition, we also show that a good strategy for designing a portfolio is to combine many short runs of the *same* algorithm. The effectiveness of such portfolios explains the common practice of "restarts" for stochastic procedures, where the same algorithm is run repeatedly with different initial seeds for the random number generator.

Our results also provide new directions for designing good heuristic algorithmic strategies. For example, using portfolios in our MIP domain, we find that a depth-first search strategy is preferable over the more standard best-bound strategy. When running a single process, the best-bound strategy is more robust than a depth-first strategy. That is, both the expected run time and variance of best-bound approaches tend to be smaller than the corresponding values for depth-first strategies. However, when running several processes interleaved or in parallel on a compute cluster, a collection of depth-first runs outperforms a set of best-bound runs. The key is that depth-first is in a sense a more "audacious" strategy. It has a much larger variance than the best-bound strategy and has a non-negligible chance of finding solutions on very short runs. By running an ensemble of such "risky" strategy, one can outperform the more conservative best-bound strategy. In fact, one obtains a smaller expected overall run time and a smaller variance when using a portfolio of depth-first runs. These insights suggest that portfolio approaches can have a concrete practical pay off, when one combines high-variance search methods.

Our focus is on an empirical validation of the portfolio approach. We will also provide several general guidelines for designing portfolio strategies and discuss various theoretical results concerning optimal portfolios. However, as we will see, the actual performance of a portfolio strategy is quite sensitive to the underlying probability distributions. It therefore appears unlikely that one can devise general practical portfolio strategies with a guaranteed pay off that are relatively independent of the underlying run time distributions of the algorithms involved. A more practical approach will need to involve some mechanism for estimating the underlying run time distributions in order to design effective portfolios.

Overall, our results suggest that the various ideas on flexible computation can indeed play a significant role in algorithm design, complementing the more traditional methods for solving computationally hard search and reasoning problems. We hope that this work will push these ideas closer to practical applications. For related work, see [10,16,24].

The paper is structured as follows. In the next section, we introduce our search procedures and problem domains. In Section 3, we present the empirical computational cost profiles obtained by running on several prototypical problem instances. Section 4 introduces the notion of portfolios with a basic formal analysis. The next section gives the overall performance profiles of a series of portfolios. We consider three types of portfolios:

(1) running on a parallel machine,
(2) running interleaved on a single processor, and
(3) running an algorithm "restart" strategy.

A restart strategy consists of a series of short runs of the same algorithm, scheduled sequentially on a single processor; each run uses a different random initial seed. From these three types of portfolios, the restart strategy has so far found the most widespread use, presumably because such an approach is straightforward to implement and is often surprisingly effective. Our results provide further evidence for the effectiveness of the restart strategy, but also reveal the potential advantage of using more sophisticated portfolio strategies.

2. Search procedures and problem domains

2.1. Randomization of backtrack search

We consider a general technique for adding randomization to complete, systematic, backtrack search procedures. One can incorporate randomization in the value or variable selection heuristics of a backtrack search method. In particular, one can randomize tie-breaking among equally good choices. Even this simple modification can dramatically change the behavior of a search algorithm, as we will see below. However, if the heuristic function is particularly powerful, it may rarely assign the highest score to more than one choice. To handle this situation, we can introduce a "heuristic equivalence" parameter to the algorithm. Setting the parameter to a value H greater than zero means that the top H-percent choices are considered equally good. This expands the choice set for random tie-breaking.

With these changes, each run of the backtrack search algorithm on a particular instance differs in the order in which choices are made and potentially in time to solution. We should note that introducing randomness in the branching variable selection does not affect the completeness of the backtrack search. Some basic bookkeeping ensures that the procedures do not revisit any previously explored part of the search space, which means that we can still determine inconsistencies, in contrast to local search methods. The bookkeeping mechanism involves some additional information, where for each variable on the stack, we keep track of which assignments have been tried so far.

We randomized the Ilog constraint solver engine for our experiments on constraint satisfaction formulations. Ilog provides a powerful C++ constraint programming library [22].

We randomized the first-fail heuristic and various variants of the Brelaz selection rule, which has been shown to be effective on graph coloring style problem domains [4].

2.2. Randomization of branch-and-bound

The standard approach used by the OR community to solve mixed integer programming problems (MIP) is branch-and-bound search. First, a linear program (LP) relaxation of the problem instance is considered. In such a relaxation, all variables of the problem are treated as continuous variables. If the solution to the LP relaxation problem has non-integer values for some of the integer variables, we have to branch on one of those variables. This way we create two new subproblems (nodes of the search tree), one with the floor of the fractional value and one with the ceiling. (For the case of binary $(0/1)$ variables, we create a node with the variable set to 0 and another with the variable set to 1.) The standard heuristic for deciding which variable to branch on is based on the degree of infeasibility of variables ("max infeasibility variable selection"). That is, we select the variable whose non-integer part in the solution of the LP relaxation is closest to 0.5. Informally, we pick the variable whose value is "least integer".

Following the strategy of repeatedly fixing integer variables to integer values will lead at some point to a subproblem with an overall integer solution (provided we are dealing with a feasible problem instance). (Note we call any solution where all the integer variables have integer values an "integer solution".) In practice, it often happens that the solution of the LP relaxation of a subproblem already is an integer solution, in which case we do not have to branch further from this node.

Once we have found an integer solution, its objective function value can be used to prune other nodes in the tree, whose relaxations have worse values. This is because the LP relaxation bounds the optimal solution of the problem. For example, for a minimization problem, the LP relaxation of a node provides a lower-bound on the best possible integer solution.

A critical issue that determines the performance of branch-and-bound is the way in which the next node to expand is selected. The standard approach, in OR, is to use a best-bound selection strategy. That is, from the list of nodes (subproblems) to be considered, we select the one with the best LP bound. (This approach is analogous to an $A^\star$ style search. The LP relaxation provides an admissible search heuristic.)

The best-bound node selection strategy is particularly well-suited for reaching an optimal solution (because of the greedy guidance), which has been the traditional focus of much of the research in OR. One significant drawback of this approach is that it may take a long time before the procedure finds an integer solution, because of the breadth first flavor of the search. Also, the approach has serious memory requirements because the full fringe of the tree has to be stored.

Given problems that have a difficult feasibility part, the best-bound approach may take too long before reaching an integer solution. (Note that an integer solution is required before any nodes can be pruned.) Therefore, we also considered a depth-first node selection strategy. Such a strategy often quickly reaches an integer solution, but may take longer to produce an overall optimal value.

In our experiments, we used a state-of-the-art MIP programming package, called CPLEX. CPLEX provides a set of libraries that allows one to customize the branch-and-bound search strategy. For example, one can vary node selection, variable selection, variable setting strategies, the LP solver, etc. We used the default settings for the LP solver, which is for the first node primal-simplex and for subsequent nodes dual-simplex. We modified the search strategies to include some level of randomization. We randomized the variable selection strategy by introducing noise in the ranking of the variables, based on maximum infeasibility. Note that the completeness of the search method is maintained. We also experimented with several other randomization strategies. For example, in CPLEX one can assign an apriori variable ranking, which is fixed throughout branch-and-bound. We experimented by randomizing this apriori ranking. We found, however, that the dynamic randomized variable selection strategy, as described above, is more effective.

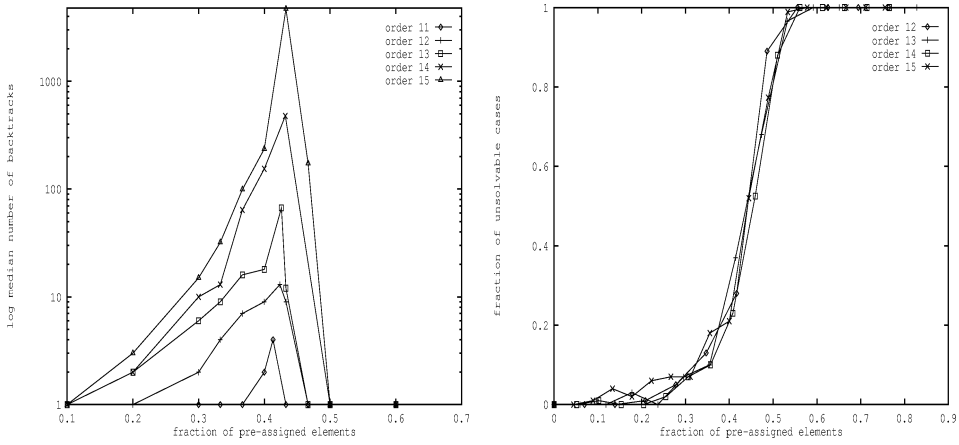
2.3. Problem domains

In order to study the performance profile of different search strategies, we derive generic distributions of hard combinatorial search problems from the domain of finite algebra. In particular, we consider the quasigroup domain. A quasigroup is an ordered pair $(Q, \cdot)$, where Q is a set and $(\cdot)$ is a binary operation on Q such that the equations $a \cdot x = b$ and $y \cdot a = b$ are uniquely solvable for every pair of elements a, b in Q. The *order* N of the quasigroup is the cardinality of the set Q. The best way to understand the structure of a quasigroup is to consider its N by N multiplication table as defined by its binary operation. The constraints on a quasigroup are such that its multiplication table defines a *Latin square*. This means that in each row of the table, each element of the set Q occurs exactly once; similarly, in each column, each element occurs exactly once [7].

An *incomplete* or *partial latin square* P is a partially filled N by N table such that no symbol occurs twice in a row or a column. The *Quasigroup Completion Problem* [11] is the problem of determining whether the remaining entries of the table can be filled in such a way that we obtain a complete latin square, that is, a full multiplication table of a quasigroup. We view the pre-assigned values of the latin square as a *perturbation* to the original problem of finding an arbitrary latin square. Another way to look at these pre-assigned values is as a set of additional problem constraints to the basic structure of the quasigroup.

There is a natural formulation of the problem as a Constraint Satisfaction Problem. We have a variable for each of the N^2 entries in the multiplication table of the quasigroup, and we use constraints to capture the requirement of having no repeated values in any row or column. All variables have the same domain, namely the set of elements Q of the quasigroup. Pre-assigned values are captured by fixing the value of some of the variables.

Colbourn [5] showed the quasigroup completion problem to be NP-complete. In previous work, we identified a clear phase transition phenomenon for the quasigroup completion problem [11]. See Fig. 1. From the figures, we observe that the costs peak roughly around the same ratio (approximately 42% pre-assignment) for different values of N. (Each data point is generated using 1000 problem instances. The pre-assigned values



(a) (b)

Fig. 1. (a) Cost profile, and (b) phase transition for the quasigroup completion problem (up to order 15).

were randomly generated.) This phase transition with the corresponding cost profile allows us to tune the difficulty of our problem class by varying the percentage of pre-assigned values.

An interesting application area of latin squares is the design of statistical experiments. The purpose of latin squares is to eliminate the effect of certain systematic dependency among the data [7]. Another interesting application is in scheduling and timetabling. For example, latin squares are useful in determining intricate schedules involving pairwise meetings among the members of a group [3]. The natural perturbation of this problem is the problem of completing a schedule given a set of pre-assigned meetings.

The quasigroup domain has also been extensively used in the area of automated theorem proving. In this community, the main interest in this domain has been driven by questions regarding the existence and nonexistence of quasigroups with additional mathematical properties [9,19].

In our experiments we were also interested in problems that combine a hard combinatorial component with numerical information. Integrating numerical information into standard AI formalism is becoming of increasing importance. For example, in planning, one would like to incorporate resource constraints or a measure of overall plan quality. We considered examples based on logistics planning problems, formulated as mixed integer programming problems. These formulations extend the traditional AI planning approach by combining the hard constraints of the planning operators, initial state, and goal state, with a series of soft constraints capturing resource utilization. Such formulations have been shown to be very promising for modeling AI planning problems [18,27].

Experimentation with the CPLEX MIP solver showed that these problem instances are characterized by a non-trivial feasibility component. [1]

[1] We thank Henry Kautz and Joachim Walser for providing us with MIP formulations of the logistic planning problems.

3. Computational cost profiles

We start by considering the computational cost of solving the quasigroup completion problem for different search strategies. As our basic search procedure, we use a complete backtrack-style search method. The performance of such procedures can vary dramatically depending on the way one selects the next variable to branch on (the "variable selection strategy") and in what order the possible values are assigned to a variable (the "value selection strategy"). There is a large body of work in both the CSP and OR communities exploring different search strategies.

One of the most effective strategies is the so-called First-Fail heuristic. [2] In the First-Fail heuristic, the next variable to branch on is the one with the smallest remaining domain (i.e., in choosing a value for the variable during the backtrack search, the search procedure has the fewest possible options left to explore—leading to the smallest branching factor). We consider a popular extension of the First-Fail heuristic, called the Brelaz heuristics [4]. The Brelaz heuristic was originally introduced for graph coloring procedures. It is one of the most powerful heuristics for graph-coloring and general CSP [26].

The Brelaz heuristic specifies a way for breaking ties in the First-fail rule: If two variables have equally small remaining domains, the Brelaz heuristic proposes to select the variable that shares constraints with the largest number of the remaining unassigned variables. A natural variation on this tie-breaking rule is what we call the "reverse Brelaz" heuristic, in which preference is given to the variable that shares constraints with the *smallest* number of unassigned variables. Any remaining ties after the (reverse) Brelaz rule are resolved randomly. One final issue left to specify in our search procedure is the order in which the values are assigned to a variable. In the standard Brelaz, value assignment is done in lexicographical order (i.e., systematic). In our experiments, we consider four strategies:

- *Brelaz-S*—Brelaz with systematic value selection,
- *Brelaz-R*—Brelaz with random value selection,
- *R-Brelaz-S*—Reverse Brelaz with systematic value selection, and
- *R-Brelaz-R*—Reverse Brelaz with random value selection.

Fig. 2 shows the performance profile of our four strategies on an instance of the quasigroup completion problem (order 20, 10% preassigned). Each curve gives the cumulative distribution obtained for each strategy by solving the problem 10,000 times. The cost (horizontal axis) is measured in number of backtracks, which is directly proportional to the total run time of our strategies. For example, the figure shows that R-Brelaz-R, finished roughly 80% of the 10,000 runs in 15 backtracks or less. The left panel of the figure shows the overall profile; the right panel gives the initial part of the profile.

Note that that R-Brelaz-R dominates R-Brelaz-S over the full profile. In other words, the cumulative relative frequency curve for R-Brelaz-R lies above that of R-Brelaz-S at every point along the x-axis. Brelaz-S, in turn, strictly dominates Brelaz-R.

[2] It is really a prerequisite for any reasonable backtrack-style search method. In theorem proving and Boolean satisfiability, the rule underlies to the powerful unit-propagation heuristic.

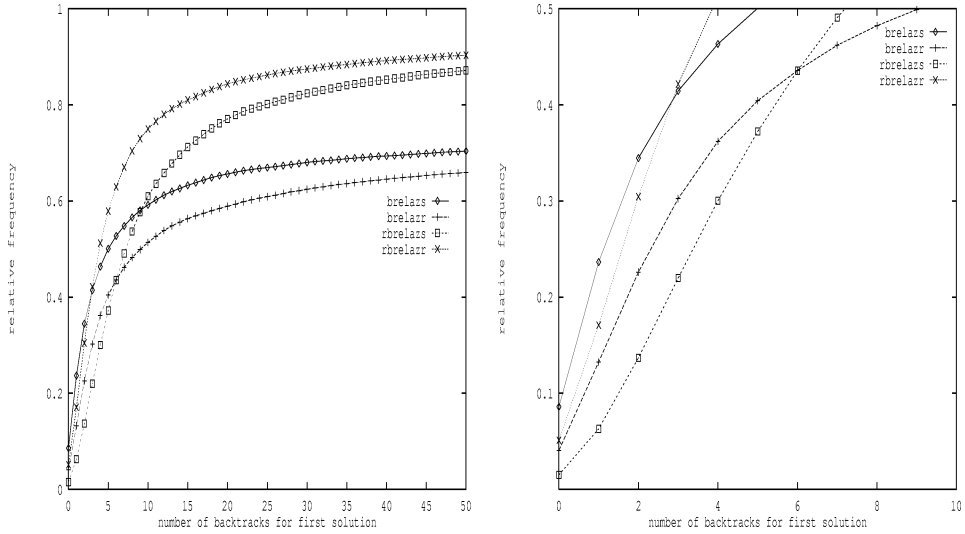


Fig. 2. Cost profiles for the quasigroup completion problem for a range of search heuristics.

From the perspective of combining algorithms, what is most interesting, however, is that in the initial part of the profile (see right panel of Fig. 2), Brelaz-S dominates R-Brelaz-R. Intuitively, Brelaz-S is better than R-Brelaz-R at finding solutions *quickly*. However, in the latter part of the cumulative distribution, R-Brelaz-R dominates Brelaz-S. In a sense, R-Brelaz-R gets relatively better when the search gets harder. As we will see in the next section, we can exploit this in our algorithm portfolio design.

In Fig. 3, we compare the run time profile of a depth-first strategy with a best-bound strategy to solve a hard feasibility problem in the logistics domain, formulated as a mixed integer programming problem. The search is terminated when a high quality solution is found, but without the requirement of proving optimality.[3] The figure shows the cumulative distribution of solution time (in number of expanded nodes). For example, with 500 or less nodes, the depth-first search finds a solution on approximately 50% of the runs. Each run had a time limit of 5000 seconds. As we see from the figure, the depth-first search initially outperforms the best-bound search. However, after more than 1500 node expansions, the best-bound becomes more effective. For example, best-bound finds a solution on approximately 75% of the runs with 2000 node expansions or less. In contrast, depth-first search can only find solutions on 55% of the runs with the same number of node expansions. This data is consistent with the observation above that best-bound may take some time to find an initial integer solution. However, once such an initial integer solution is found, best-bound becomes more effective.

[3] One should be careful to distinguish between finding an optimal integer solution and proving that this is indeed *the* optimal solution. Our interest lies in problems where the proof of optimality can be beyond reach of any procedure; however, we can often still find a good quality solution. For the particular problem instance used in our experiment, we first determined a good solution value by using a very long run of best-bound.

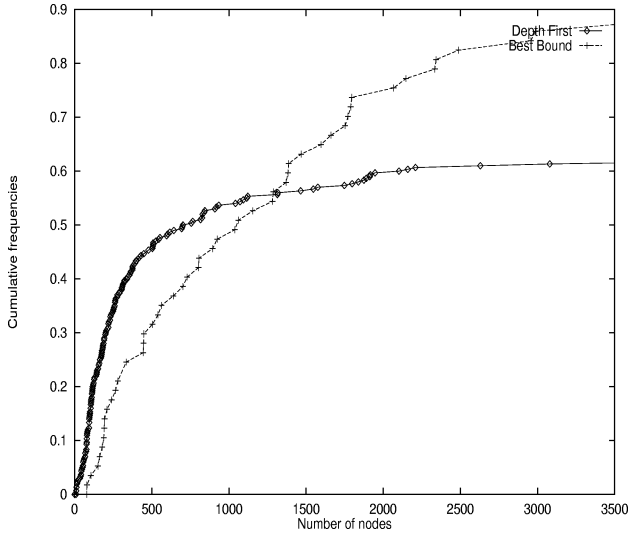


Fig. 3. Cost profiles for a logistics planning problems for depth-first and best-bound search strategies.

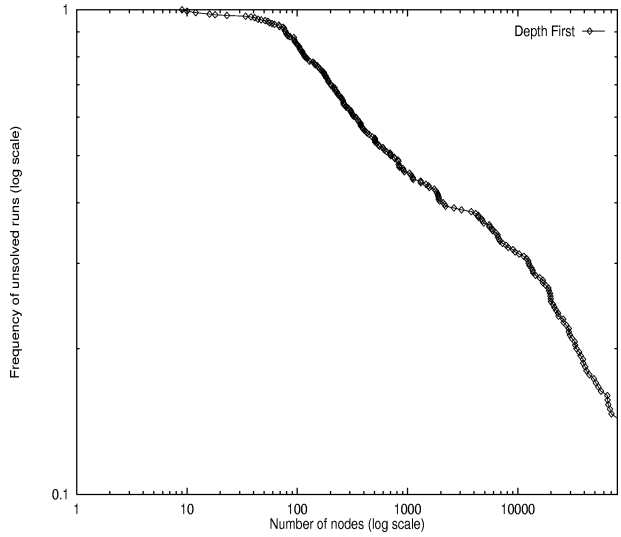


Fig. 4. Heavy-tailed behavior of depth-first search.

We now consider the run time distributions more closely. Fig. 4 gives a log-log plot of the complement of the cumulative distribution for the depth-first procedure. For example, from this plot, we see that after 10,000 nodes, approximately 30% of the runs have not yet found the solution. The figure shows a near linear behavior over several orders of magnitude. This is an indication of so-called heavy-tailed behavior which often characterizes complete

search methods [12]. In a sense, the time till solution behaves in a very erratic manner: very long runs occur much more frequently than one might expect.

Heavy-tailed distributions are formally characterized by tails that have a *power-law* (polynomial) decay, i.e., distributions which asymptotically have "heavy tails"—also called tails of the Pareto–Lévy form, viz.

$$P[X > x] \sim Cx^{-\alpha}, \quad x > 0,$$

where $0 < \alpha < 2$ and $C > 0$ are constants. Some of the moments of heavy-tailed distributions are infinite (e.g., some heavy-tailed distributions have infinite mean and infinite variance, others just infinite variance).

In the next section, we show how the large variance in search methods, as characterized by heavy-tailed behavior, can be exploited by combining algorithms into portfolios or running multiple copies of the same algorithm.

4. Portfolio design

A *portfolio of algorithms* is a collection of different algorithms and/or different copies of the same algorithm running on different processors [10,16]. [4] Here we consider the case of independent runs without interprocess communication.

We are considering Las Vegas type algorithms, i.e., stochastic algorithms that always return a model satisfying the constraints of the search problem or demonstrate that no such model exists [21]. The computational cost of the portfolio is therefore a random variable. The *expected* computational cost of the portfolio is simply the expected value of the random variable associated with the portfolio and its *standard deviation* is a measure of the "dispersion" of the computational cost obtained when using the portfolio of algorithms. In this sense, the standard deviation is a measure of the risk inherent to the portfolio.

The main motivation to combine different algorithms into a portfolio is to improve on the performance of the component algorithms, mainly in terms of expected computational cost but also in terms of the overall *risk*. As we will show, some portfolios are strictly preferable to others, in the sense that they provide a lower risk and also a lower expected computational cost. However, in some cases, we cannot identify any portfolio within a set that is the best, both in terms of expected value and risk. This set of portfolios corresponds to the *efficient set* or *efficient frontier*, following terminology used in the theory of mathematical finance. Within this set, in order to minimize the risk, one has to deteriorate the expected value or, in order to improve the expected value of the portfolio, one has to increase the risk.

Let us consider a set of two algorithms, *algorithm* 1 and *algorithm* 2. Let us associate a random variable with each algorithm: A1—the number of backtracks that algorithm 1 takes to find the first solution or to prove that a solution does not exist; A2—the number of backtracks that algorithm 2 takes to find the first solution or to prove that a solution does not exist.

[4] Later on, we will also consider the the case of interleaving the execution of algorithms on one or more processors.

Let us assume that we have N processors and that we design a portfolio using $n1$ processors with *algorithm* 1 and $n2$ processors with *algorithm* 2. So, $N = n1 + n2$. Let us define the random variable associated with this portfolio: X—the number of backtracks that the portfolio takes to find the first solution or to prove that a solution does not exist.

The probability distribution of X is a "weighted" probability distribution of the probability distributions of *algorithm* 1 and *algorithm* 2. More precisely, the probability that $X = x$ is given by the probability that one processor takes exactly x backtracks and all the other ones take x or more backtracks to find a solution or to prove that a solution does not exist.

Let us assume that we have N processors and our portfolio consists of N copies of *algorithm* 1. In this case, $P[X = x]$ is given by the probability that one processor take exactly x backtracks and the other $N - 1$ take more than x backtracks, plus the probability that two processors take exactly x backtracks and the other $(N - 2)$ take more than x backtracks, etc., plus the probability that all the processors take exactly x backtracks to find a solution or to prove that a solution does not exist. The following expression gives the probability function for such a portfolio.

Given N processors, and let $n1 = N$ and $n2 = 0$. $P[X = x]$ is given by

$$\sum_{i=1}^{N} \binom{N}{i} P[A1 = x]^i P[A1 > x]^{(N-i)}.$$

To consider two algorithms, we have to generalize the above expression, considering that $X = x$ can occur just within the processors that use *algorithm* 1, or just within the processors that use *algorithm* 2 or within both. As a result, the probability function for a portfolio with two algorithms, is given by the following expression:

Given N processors, $n1$ such that $0 <= n1 <= N$, and $n2 = N - n1$, $P[X = x]$ is given by

$$\sum_{i=1}^{N} \sum_{i'=0}^{n1} \binom{n1}{i'} P[A1 = x]^{i'} P[A1 > x]^{(n1-i')} \times \binom{n2}{i''} P[A2 = x]^{i''} P[A2 > x]^{(n2-i'')}.$$

The value of i'' is given by $i'' = i - i'$, and the term in the summation is 0 whenever $i'' < 0$ or $i'' > n2$.

In the case of a portfolio involving two algorithms the probability distribution of the portfolio is a summation of a product of two expressions, each one corresponding to one algorithm. In the case of a portfolio comprising M different algorithms, this probability function can be easily generalized, by having a summation of a product of M expressions, each corresponding to an algorithm.

Once we obtain the probability distribution for the random variable associated with the portfolio, we can compute its *expected value* and *standard deviation*.

5. Empirical results for portfolio design

5.1. Constraint satisfaction

We now derive different portfolios based on the run time profiles given in Fig. 2 (Section 3). This is an interesting case from the portfolio design perspective because Brelaz-S dominates in the initial part of the distribution, whereas R-Brelaz-R dominates in the latter part.

It should be noted that in real-world applications one may not be able to obtain detailed run time distributions for evaluating possible portfolios. In such cases, run time distributions obtained from experiments on scaled down problem instances should still provide useful guidance for finding effective portfolios. Moreover, within a single application domain, a search technique often behaves similarly on instances with similar structural characteristics (e.g., in the quasigroup domain, the level of preassignment is a good predictor of run time behavior). In such cases, the run distributions of a few prototypical instances also provide good approximations for portfolio design. [5]

Fig. 5 gives the expected run time values and the standard deviations of portfolios for 2, 5, 10, and 20 processors. (Results derived using the expressions given above.) We see that for two processors, the portfolio consisting of two copies of R-Brelaz-R has the lowest expected run time *and* the lowest standard deviation. This portfolio dominates the two other 2-processor portfolios.

When we increase the number of processors, we observe an interesting shift in the optimal portfolio mix. For example, for 5 processors, using 2 copies of Brelaz-S and 3 copies of R-Brelaz-R gives a better expected value at only a slight increase in the risk (standard deviation), compared to a portfolio consisting of 5 copies of R-Brelaz-R. In the case of five processors, the efficient set comprises four portfolios: one with 5 R-Brelaz-R, one with 1 Brelaz-S and 4 R-Brelaz-R, one with 2 Brelaz-S and 3 R-Brelaz-R, and one with 3 Brelaz-S and 2 R-Brelaz-R. There is no clear dominant portfolio among those four. In this set, one has to trade a decrease in expected run time for an increase in variance: in order to minimize the expected run time, the best portfolio is 3 Brelaz-S and 2 R-Brelaz-R; in order to minimize the risk (variance) the best portfolio corresponds to 5 R-Brelaz-R.

The situation changes even more dramatically if we increase the number of processors. In particular, with 20 processors (Fig. 5(d)), the best portfolio corresponds to using only copies of the Brelaz-S strategy on all processors, obtaining the lowest expected value and the lowest standard deviation. The intuitive explanation for this is that by running many copies of Brelaz-S, we have a good chance that at least one of them will find a solution quickly. Overall we have the somewhat counter-intuitive result that, even when given two stochastic algorithms where neither strictly dominates the other, running multiple copies of a single algorithm can be preferable to a mix of algorithms. One might be tempted to conclude that with the number of processors growing arbitrarily large, the optimal portfolio would always consist solely of copies of the stochastic method that dominates early on (as

[5] For example, the Blackbox planner incorporates a series of portfolio strategies [17]. In general, for a given planning domain, our experience has been that it is often sufficient to tune the strategy on a small set of planning instances. See also [24].

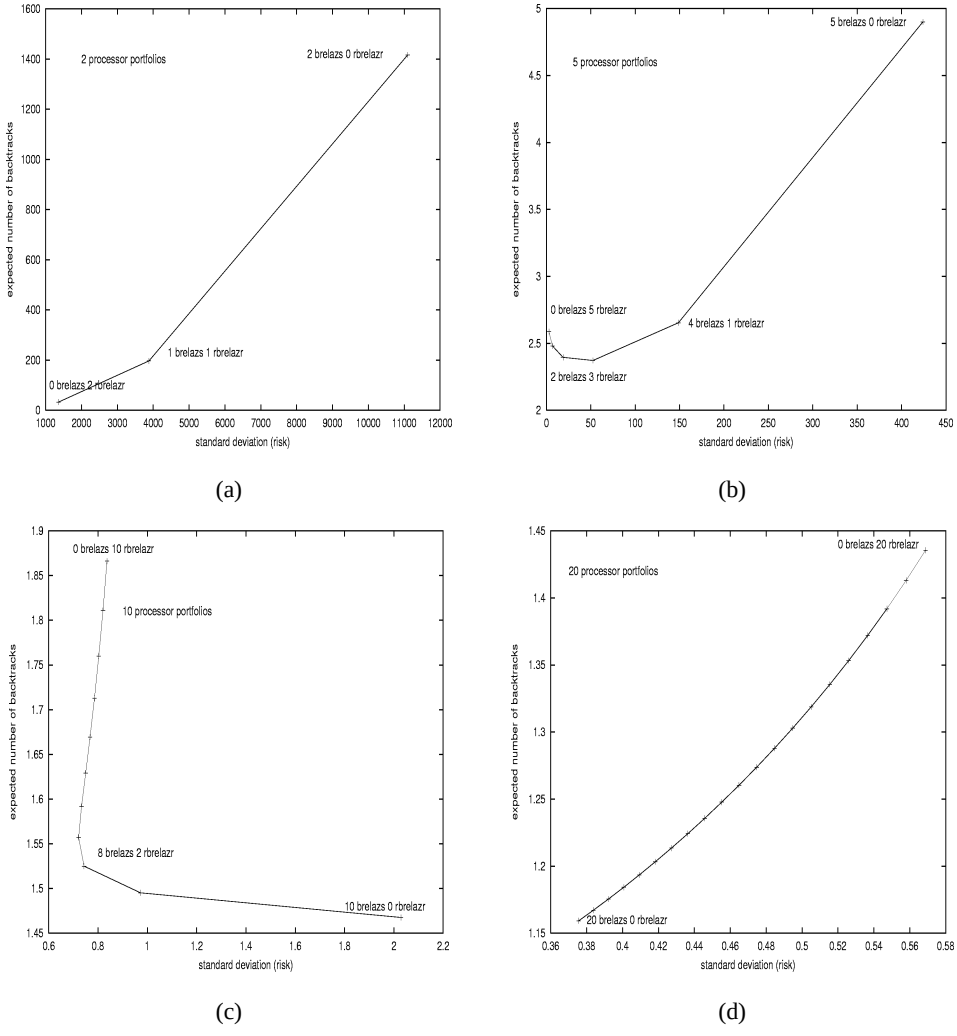


Fig. 5. Portfolio combining Brelaz-S and R-Brelaz-R for solving a quasigroup completion instance using two (a), five (b), ten (c), and twenty processors (d).

in Fig. 5(d)). However, one can find pairs of run time distributions for which the optimal strategy in the limit is still a mixed portfolio. In general, it appears that the optimal portfolio design is quite sensitive to the details of the underlying run time distributions.

Fig. 6 combines the results from Fig. 5 into a single figure to show the relative positions of the performance profiles more clearly. We see how increasing the number of processors leads to a continued improvement in performance both in terms of the expected value and variance of the run time, as one would expect for parallel runs.

In the scenario we considered so far, each process runs on its own dedicated processor. In practice, one may often need to run the portfolio interleaved on one or a limited number

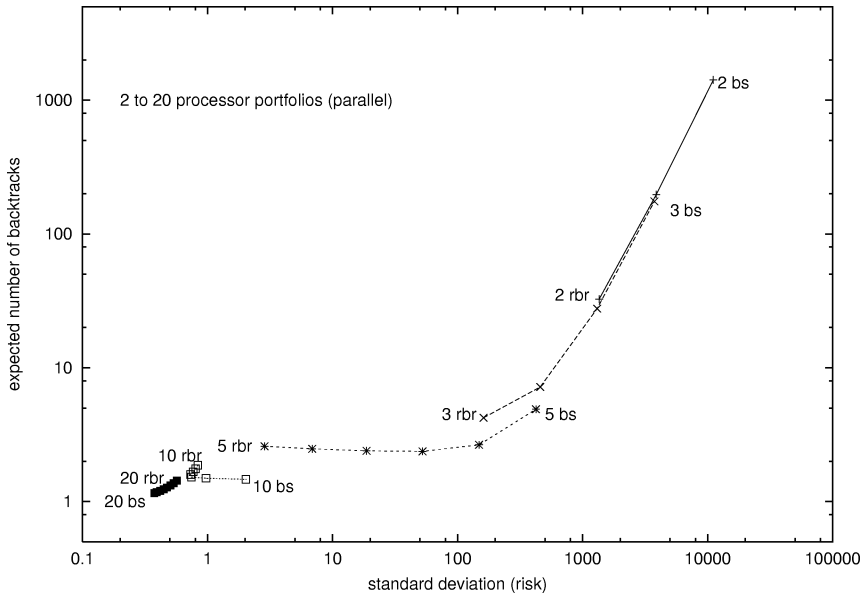


Fig. 6. Portfolios combining Brelaz-S (bs) and R-Brelaz-R (rbr) for 2 to 20 processors.

of processors. In that case, the expected *total* CPU use becomes relevant. Fig. 7 gives the results of running interleaved processes on a single processor.

We see again an initial reduction in run time and variance when increasing the number of processes running interleaved. However, at some point the approach becomes counter-productive and the average and mean start increasing. See, for example, the case for 20 interleaved processes. Intuitively, this happens when the benefit of an increased probability of reaching a solution early on by having many independent runs does not counterbalance the increased cost of running many copies interleaved. In this case, the efficient set includes, e.g., a portfolio with 8 Brelaz-S and 2 R-Brelaz-R and a 5 process portfolio, running only R-Brelaz-R.

Finally, we consider one more portfolio strategy, consisting of a "restart" approach running on a single processor. The idea behind restarts is to run a randomized algorithm for a fixed amount of time. If no solution is found, the algorithm is restarted with a new random seed. This process is repeated until a solution is found or one reaches a preset resource limit. Restart strategies are often used in practice. They are straightforward to implement and can be surprisingly effective. For related work on restarts, see e.g., [1,2,8, 14,20,25].

Restarts were originally proposed as a way of escaping from local minima in local search methods. More recently, we have shown that, because of the heavy-tailed distributions underlying backtrack style search methods, restarts can also be effective when dealing with complete search procedures. In fact, a restart strategy provably eliminates heavy-tailed behavior. This can be seen from the following argument [13,14].

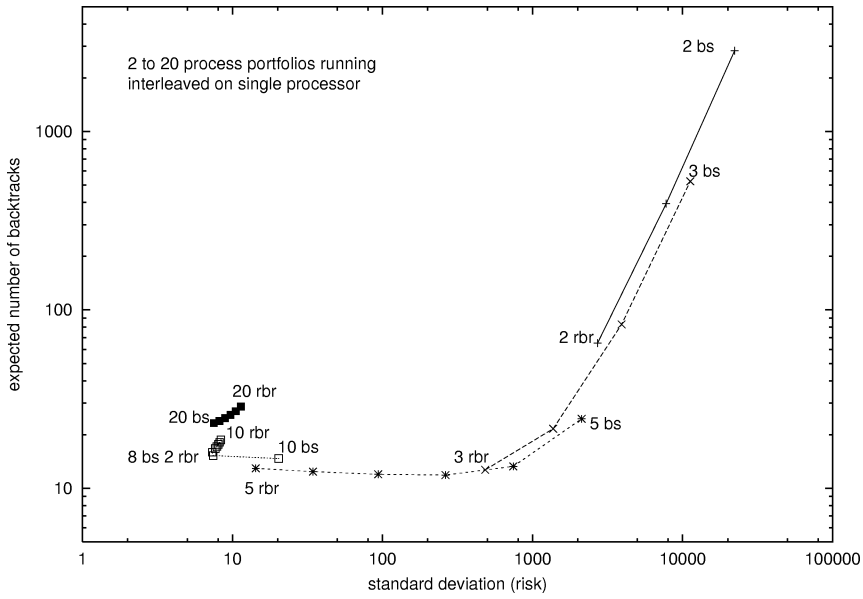


Fig. 7. Portfolios combining Brelaz-S (bs) and R-Brelaz-R (rbr) running interleaved on a single processor.

The runs executed in a restart portfolio are independent (no information is carried over between runs, and each run uses a new random seed) and therefore constitute a sequence of Bernoulli trials. The success of each trial corresponds to finding a solution, or proving that one does not exist, during a run. Its probability is given by $P[B \leqslant c]$, where the random variable B gives the number backtracks needed by the original randomized backtrack search procedure, and c is the cutoff value of the restart strategy. Therefore, the number of runs executed by the restart strategy follows a *geometric distribution* with parameter $p = P[B \leqslant c]$. Let S be the random variable representing the total number of backtracks for the restart strategy until a solution is found or infeasibility is proven. The probability of the tail of S, $P[S > s]$, corresponds to the probability of not finding the solution in the first $\lfloor s/c \rfloor$ runs, and finding it with more than (s mod c) choice points in the next run. We obtain the following expression for the tail distribution:

$$P[S > s] = P[B > c]^{\lfloor s/c \rfloor}\ P[B > s \bmod c].$$

Given the exponential decay of the tail of the distribution, it follows that the restart portfolio is not heavy-tailed. (For details, see [14].)

Fig. 8 shows the performance of restart portfolios for R-Brelaz-R and Brelaz-S.[6] The data points in Fig. 8 correspond to different restart rates (i.e., different cutoff values). For example, Brelaz-S, restarted every 15 backtracks, gives an average time till solution of around 11.5 backtracks with a standard deviation of around 13. From the figure, we see that

[6] One could also consider combined restart approaches, for example where one alternates between running Brelaz-S and R-Brelaz-R. Such scenarios do not change the overall results in a significant manner.

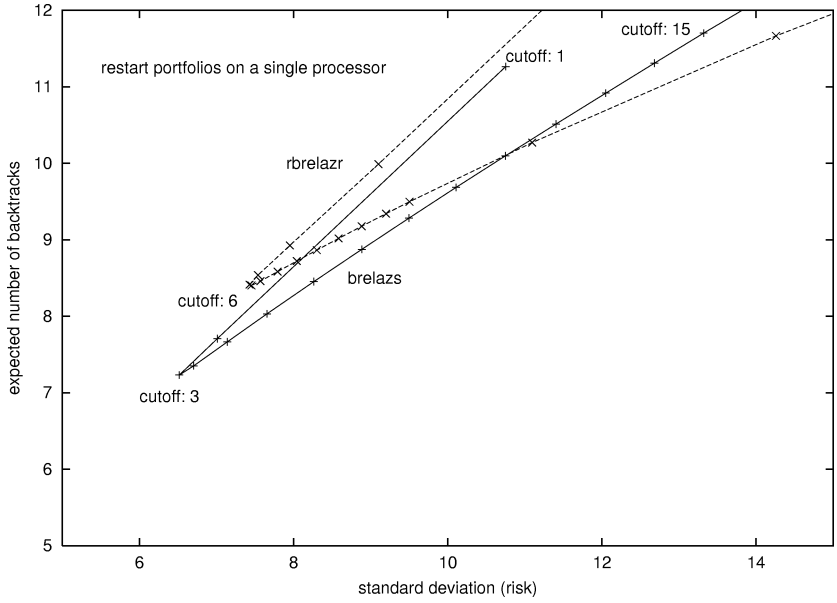


Fig. 8. Comparing restart strategies for different cutoff values.

restarting every 6 backtracks is optimal for the R-Brelaz-R strategy, while an even better performance can be obtained by running Brelaz-S with a cutoff value of 3 backtracks.

The effectiveness of restarts on these problem instances follows from the fact that they exploit a non-negligible probability of finding a solution early on, while at the same time they avoid the heavy-tailed component of the run time distributions, by doing full restarts. In fact, we see that a restart approach on a single processor actually outperforms running interleaved (Fig. 7) for these run time distributions. Note that running interleaved can still lead to an advantage in case short successful runs are very rare. Moreover, interleaving gives more flexibility in combining run time distributions with very different characteristics.

5.2. Mixed Integer Programming

We now consider our MIP problem domain. In Section 3, we saw that there are several interesting trade-offs between depth-first branch-and-bound versus best-bound branch-and-bound. In particular, depth-first search performs better early on in the search, whereas best-bound is better on longer runs. A portfolio approach can again be used to effectively combine the best features of each search strategy.

In Fig. 9, we consider a range of portfolios for solving our feasibility problem for the logistics domain, considering situations from two processors to twenty processors (the same instance as the one considered in Fig. 3). The plot gives the expected run time and standard deviation for different ways of combining a branch-and-bound search procedure using depth-first search and best-bound search. From this plot we see that the

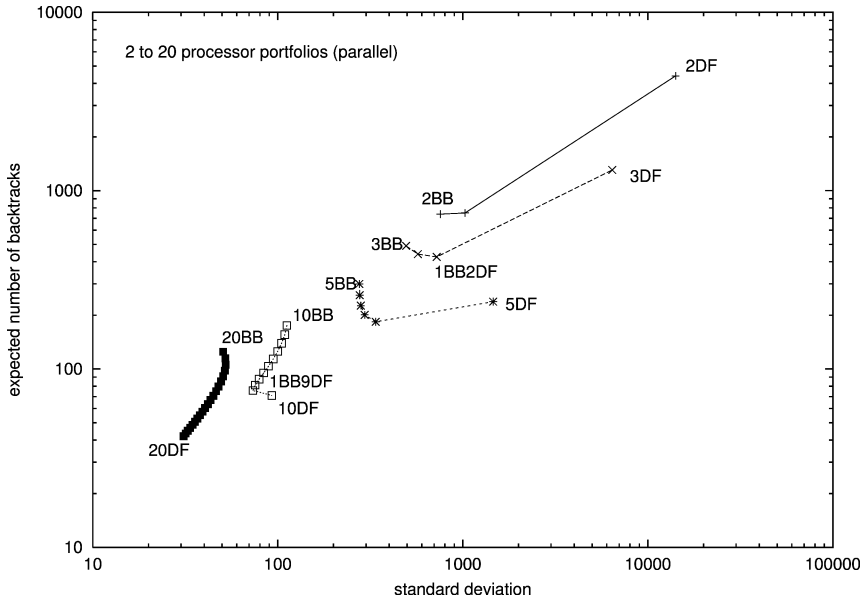


Fig. 9. Portfolios for the MIP formulation of logistics planning.

best choice, when using two processors, in terms of minimizing expected run time and standard deviation, consists of running branch-and-bound with best-bound only.

The mixing strategy changes as we increase the number of processors or the amount of interleaving. For example, for the case of ten processors, the best strategies are 9DF/1BB and 10DF/0BB. (We use the notation xDF/yBB to mean running x depth-first processes and y best-bound processes.) These strategies give both a low expected run time and a low standard deviation. There is no clear dominant strategy among those two (efficient set). In this set, one has to trade a decrease in expected run time for an increase in variance: in order to minimize the expected run time, the best portfolio is 10DF/0BB; in order to minimize the risk (variance) the best portfolio corresponds to 9DF/1BB. For the case of twenty processors, the dominating portfolio again becomes a uniform one—running only depth-first search (i.e., 20DF/0BB).

In Fig. 10, we give the performance of MIP portfolios running interleaved on a single processor. As we saw earlier, we now pay a cost for running too many processes. The best overall performance in terms of expected is obtained by running 10 processes. The performance of portfolios 9DF/1BB and 10DF/0BB is comparable and near optimal.

In Fig. 11, we consider the restart portfolio strategy. The figure shows the effect of different cutoff values in terms of the restart strategy. (The lowest cutoff we consider for best-bound is 100 backtracks, since the probability of finding a solution with less backtracks is negligible.) We see that the overall performance of the restart portfolio is a function of the cutoff value for both best-bound and depth-first search. However, the performance of depth-first is much more sensitive to the cutoff value than that of best-bound. This is due to the fact that the run time behavior of the depth-first MIP strategy

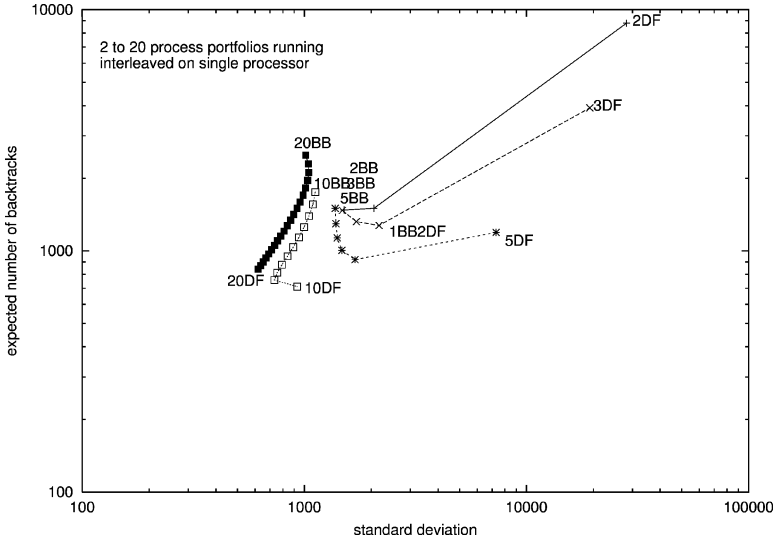


Fig. 10. Portfolios for the MIP formulation of logistics planning running interleaved on a single processor.

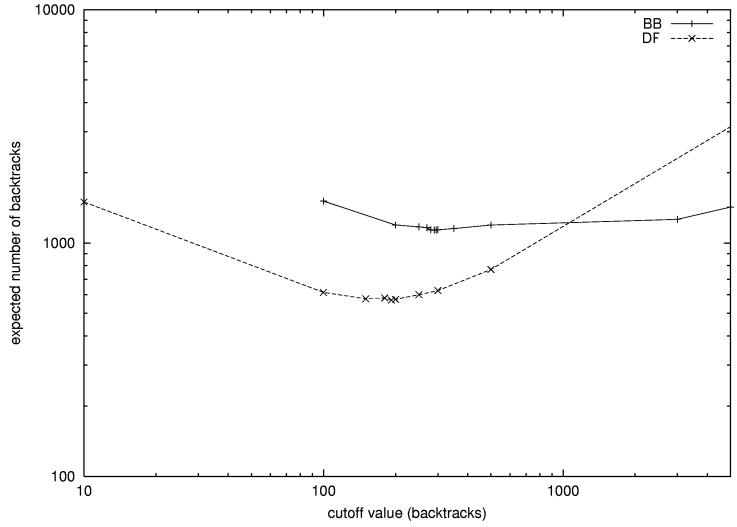


Fig. 11. Expected cost as a function of restart cutoff for depth-first (DF) and best-bound (BB) MIP.

is strongly heavy-tailed. By choosing a cutoff rate of around 200 backtracks, depth-first gives the overall best performance. This is a clear example of how a more "risk-seeking" approach can improve overall performance when using a portfolio strategy. In this case, the best performance of the interleaving strategy (Fig. 10) approaches that of the restart approach.

6. Conclusions

We have provided results showing the computational advantage of a portfolio approach for dealing with hard combinatorial search and reasoning problems. Our results considered two predominant representation paradigms for combinatorial problems: Constraint Satisfaction formulations and Mixed Integer Programming formulations. Our analysis shows that one can exploit the large variance in certain randomized search methods by running them in a portfolio strategy and obtaining a superior overall performance, compared to more conservative algorithmic strategies. As our experiments show, the portfolio approach suggests new algorithm design strategies. In particular, there is an advantage in developing methods that have some chance of finding a solution early on, even though, on a single processor, this may lead to a much larger overall expected run time and variance than that of other more traditional search techniques. Finally, if only a single processor is available, random restarts of a stochastic method is often the optimal strategy.

References

[1] D. Aldous, U. Vazirani, Go with the winners algorithms, in: Proc. 35th Symposium on the Found. of Comp. Sci., IEEE Press, 1994, pp. 492–501.
[2] H. Alt, L. Guibas, K. Mehlhorn, R. Karp, A. Wigderson, A method for obtaining randomized algorithms with small tail probabilities, Algorithmica 16 (4–5) (1996) 543–547.
[3] L. Anderson, Completing partial latin squares, Mathematisk Fysiske Meddelelser 41 (1985) 23–69.
[4] D. Brelaz, New methods to color the vertices of a graph, Comm. ACM 22 (4) (1979) 251–256.
[5] C. Colbourn, Embedding partial Steiner triple systems is NP-complete, J. Combin. Theory A 35 (1983) 100–105.
[6] T. Dean, M. Boddy, An analysis of time-dependent planning, in: Proc. AAAI-88, St. Paul, MN, 1988, pp. 49–54.
[7] J. Denes, A. Keedwell, Latin Squares and their Applications, Akademiai Kiado, Budapest/English Universities Press, London, 1974.
[8] W. Ertel, M. Luby, Optimal parallelization of Las Vegas algorithms, in: Proc. Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Computer Science, Vol. 775, Springer, Berlin, 1994, pp. 463–475.
[9] M. Fujita, J. Slaney, F. Bennett, Automatic generation of some results in finite algebra, in: Proc. IJCAI-93, Chambéry, France, AAAI Press, Menlo Park, CA, 1993, pp. 52–57.
[10] C.P. Gomes, B. Selman, Algorithm portfolio design: Theory vs. practice, in: Proc. 13th Conference on Uncertainty in Artificial Intelligence (UAI-97), Providence, RI, Morgan Kaufmann, San Mateo, CA, 1997, pp. 190–197.
[11] C.P. Gomes, B. Selman, Problem structure in the presence of perturbations, in: Proc. AAAI-97, Providence, RI, AAAI Press, Menlo Park, CA, 1997, pp. 221–227.
[12] C.P. Gomes, B. Selman, N. Crato, Heavy-tailed distributions in combinatorial search, in: G. Smolka (Ed.), Principles and Practice of Constraint Programming (CP'97), Linz, Austria, Lecture Notes in Computer Science, Springer, Berlin, 1997, pp. 121–135.
[13] C.P. Gomes, B. Selman, H. Kautz, Boosting combinatorial search through randomization, in: Proc. AAAI-98, Providence, RI, AAAI Press, Menlo Park, CA, 1998, pp. 431–438.
[14] C.P. Gomes, B. Selman, N. Crato, H. Kautz, Heavy-tailed phenomena in satisfiability and constraint satisfaction problems, J. Automat. Reason. 24 (1–2) (2000) 67–100.
[15] E. Horvitz, S.Z. (Eds.), Proceedings of Flexible Computation, Technical Report, AAAI Fall Symposium, 1995.
[16] B. Huberman, R. Lukose, T. Hogg, An economics approach to hard computational problems, Science 275 (1997) 51–54.

[17] H. Kautz, B. Selman, Unifying SAT-based and graph-based planning, in: T. Dean (Ed.), Proc. IJCAI-99, Stockholm, Sweden, Morgan Kaufmann, San Mateo, CA, 1999, pp. 318–325.
[18] H. Kautz, J. Walser, State-space planning by integer optimization, in: Proc. AAAI-99, Orlando, FL, AAAI Press, Menlo Park, CA, 1999, pp. 526–533.
[19] C. Lam, L. Thiel, S. Swiercz, The non-existence of finite projective planes of order 10, Can. J. Math. XLI (6) (1994) 1117–1123.
[20] M. Luby, A. Sinclair, D. Zuckerman, Optimal speedup of Las Vegas algorithms, Inform. Process. Lett. 47 (1993) 173–180.
[21] R. Motwani, P. Raghavan, Randomized Algorithms, Cambridge University Press, Cambridge, England, 1995.
[22] J.F. Puget, M. Leconte, Beyond the black box: Constraints as objects, in: Proc. ILPS'95, Lisbon, Portugal, MIT Press, Cambridge, MA, 1995, pp. 513–527.
[23] S. Russell, P. Norvig, Artificial Intelligence: A Modern Approach, Prentice Hall, Englewood Cliffs, NJ, 1995.
[24] F.S. Salman, J.R. Kalagnanam, A. Davenport, Cooperative strategies for solving the bicriteria sparse multiple knapsack problem, in: Proc. Congress on Evolutionary Computation, 1999, pp. 53–60.
[25] B. Selman, S. Kirkpatrick, Finite-size scaling of the computational cost of systematic search, Artificial Intelligence 81 (1–2) (1996) 273–295.
[26] M. Trick, D.J. (Eds.), Proc. DIMACS Challenge on Satisfiability Testing, Graph Coloring, and Cliques, DIMACS Series on Discr. Math., Am. Math. Soc., Providence, RI, 1996.
[27] T. Vossen, M. Ball, A. Lotem, D. Nau, Integer programming models in AI planning, Technical Report, Workshop on Planning as Combinatorial Search (held in conjunction with AIPS-98), Pittsburgh, PA, 1999.