Local Search and Backtracking vs Non-Systematic Backtracking

Steven Prestwich
Department of Computer Science
National University of Ireland at Cork
s.prestwich@cs.ucc.ie

Abstract
This paper addresses the following question: what is the essential difference between stochastic local search (LS) and systematic backtracking (BT) that sometimes causes LS to scale far better than BT? We propose that LS’s main advantage is simply that it does not commit to any variable assignment. We modify three BT algorithms to have the same feature, retaining constraint handling though not completeness: a forward checker for n-queens, the DSATUR algorithm for graph colouring, and a Davis-Loveland procedure for satisfiability. In each case the modified algorithm sometimes outperforms LS or BT or both.

1 Introduction
A variety of algorithms have been devised to solve combinatorial problems, two contrasting approaches being systematic backtracking and stochastic local search. Backtrackers are complete and can therefore prove unsolvability or find all solutions. They can also prove the optimality of a solution to an optimisation problem by failing to find a better solution. They often use techniques such as constraint propagation, value and variable ordering heuristics, branch-and-bound and intelligent backtracking. Backtrackers are sometimes called constructive because they construct solutions from partial consistent assignments of domain values to variables. In contrast, local search algorithms are typically non-constructive, searching a space of total but inconsistent assignments. They attempt to minimise constraint violations, and when all violations have been removed a solution has been found. Two drawbacks of local search are incompleteness and an inability to exploit constraint processing techniques. However, it often has much better scalability than backtracking.

Neither backtracking nor local search is seen as adequate for all problems. Real-world problems often have a great deal of structure, giving them different properties than randomly generated problems. Such problems are often solved effectively by backtrackers, which are able to exploit structure via constraint propagation. Unfortunately, backtrackers do not always scale well to very large problems, even when augmented with powerful constraint propagation and intelligent backtracking. For these problems it is often more efficient to use local search. However, non-constructive local search algorithms cannot fully exploit problem structure. This situation has motivated research into hybrid approaches, with the aim of combining features of both classes of algorithm to solve currently intractable problems.

What is the essential difference between local search and backtracking, that sometimes enables local search to scale far better than backtracking? Answering this question may lead to new algorithms that outperform both. The issue was debated in [Freuder et al., 1995] with little consensus, but some suggestions were local search’s locality, randomness, incompleteness, different search space, and ability to follow gradients. We suggest that the essential difference is backtrackers’ strong commitment to variable assignments, whereas local search can in principle (given sufficient noise) reassign any variable at any point during search. This view has led to a new class of algorithms that sometimes out-performs both backtracking and local search.

Implementations of the algorithm have already been described [Prestwich, 1998, Prestwich, 2000a, Prestwich, 2000b, Prestwich, 2001] but it is summarised here. It begins like any backtracker, assigning values to variables and optionally interleaving constraint propagation, and on reaching a dead-end it backtracks by unassigning variables, then tries to assign again. The novel feature is the choice of backtracking variable: the variables selected for unassignment are chosen randomly, or using a heuristic. As in Dynamic Backtracking [Ginsberg, 1993] only the selected variables are unassigned, without losing later assignments. Because of its similarity to Dynamic Backtracking we shall call this approach Incomplete Dynamic Backtracking (IDB).

The search is no longer complete, so we must sometimes force the unassigning of more than one variable. We do this by adding an integer parameter $b \geq 1$ to the
algorithm, and unassign b variables at each dead-end. A value ordering heuristic is also used, but not based on constraint violations as is usual, but based on previous assignments. It slightly improves performance and was described in previous papers.

A complication arises when we wish to combine IDB with forward checking. Say we have reached a dead-end after assigning variables $v_1, \ldots, v_k$, and $v_{k+1}, \ldots, v_b$ remain unassigned. We would like to unassign some arbitrary variable $v_k$ where $1 \leq u \leq k$, leaving the domains in the state they would have been in had we assigned only $v_1, \ldots, v_{u-1}, v_{u+1}, \ldots, v_k$. How can we do this efficiently? One way to characterise forward checking is as follows: a variable $x$ is in the domain of a currently unassigned variable $v$ if and only if adding the assignment $v = x$ would not cause a constraint violation. In backtracking algorithms this is often used to update the domains of unassigned variables as assignments are added and removed. We generalise this idea by associating with each value $x$ in the domain of each variable $v$ a conflict count $C_{v,x}$. The value of $C_{v,x}$ is the number of constraints that would be violated if the assignment $v = x$ were added. If $C_{v,x} = 0$ then $x$ is currently in the domain of $v$. We make a further generalisation by maintaining the conflict counts of all variables, assigned and unassigned. The meaning of a conflict count for a currently assigned variable is: $C_{v,x}$ is the number of constraints that would be violated if the variable $v$ were reassigned to $x$. Now on assigning or unassigning a variable, we incrementally update conflict counts in all other variable domains. This is clearly more expensive than standard forward checking, which only updates the domains of unassigned variables. However, the state of each variable domain is now independent of the order in which assignments were made, and we can unassign variables arbitrarily. Moreover, it applies to both binary and non-binary constraints, and has been used to combine IDB with unit resolution for satisfiability problems [Prestwich, 2000a, Prestwich, 2000b].

Section 2 evaluates the IDB on some well-known combinatorial problems, and Section 3 discusses the results and related work.

2 Experiments

In this section IDB is applied to three problems: $n$-queens, graph colouring and satisfiability. The experiments are performed on a 300MHz DEC AlphaServer 1000A 5/300 under Unix. For readers wishing to normalise our execution times to other platforms, the DL-MACS [Johnson and Trick, 1996] benchmark program $dfmax$ runs 0.5 seconds on the Alphaserver.

2.1 The $n$-queens problem

We first evaluate IDB on the $n$-queens problem. Though fashionable several years ago $n$-queens is no longer considered a challenging problem. However, large instances still defeat most backtrackers and it is therefore of interest. Consider a generalised chess board, which is a square divided into $n \times n$ smaller squares. Place $n$ queens on it in such a way that no queen attacks any other. (This is often called an $n$-queens attack another if it is on the same row, column or diagonal (in which case both attacks each other). We can model this problem using $n$ variables each with domain $D_i = \{1, \ldots, n\}$. A variable $v_i$ corresponds to a queen on row $i$ (there is one queen per row), and the assignment $v_i = j$ denotes that the queen on row $i$ is placed in column $j$, where $j \in D_i$. The constraint is that $v_i = v_j$ would not cause a constraint violation. In backtracking algorithms this is often used to update the domains of unassigned variables as assignments are added and removed. We generalise this idea by associating with each value $x$ in the domain of each variable $v$ a conflict count $C_{v,x}$. The value of $C_{v,x}$ is the number of constraints that would be violated if the assignment $v = x$ were added. If $C_{v,x} = 0$ then $x$ is currently in the domain of $v$. We make a further generalisation by maintaining the conflict counts of all variables, assigned and unassigned. The meaning of a conflict count for a currently assigned variable is: $C_{v,x}$ is the number of constraints that would be violated if the variable $v$ were reassigned to $x$. Now on assigning or unassigning a variable, we incrementally update conflict counts in all other variable domains. This is clearly more expensive than standard forward checking, which only updates the domains of unassigned variables. However, the state of each variable domain is now independent of the order in which assignments were made, and we can unassign variables arbitrarily. Moreover, it applies to both binary and non-binary constraints, and has been used to combine IDB with unit resolution for satisfiability problems [Prestwich, 2000a, Prestwich, 2000b].

Section 2 evaluates the IDB on some well-known combinatorial problems, and Section 3 discusses the results and related work.

2 Experiments

In this section IDB is applied to three problems: $n$-queens, graph colouring and satisfiability. The experiments are performed on a 300MHz DEC AlphaServer 1000A 5/300 under Unix. For readers wishing to normalise our execution times to other platforms, the DL-MACS [Johnson and Trick, 1996] benchmark program $dfmax$ runs 462 seconds on the Alphaserver.

2.1 The $n$-queens problem

We first evaluate IDB on the $n$-queens problem. Though fashionable several years ago $n$-queens is no longer considered a challenging problem. However, large instances still defeat most backtrackers and it is therefore of interest. Consider a generalised chess board, which is a square divided into $n \times n$ smaller squares. Place $n$ queens on it in such a way that no queen attacks any other. (This is often called an $n$-queens attack another if it is on the same row, column or diagonal (in which case both attacks each other). We can model this problem using $n$ variables each with domain $D_i = \{1, \ldots, n\}$. A variable $v_i$ corresponds to a queen on row $i$ (there is one queen per row), and the assignment $v_i = j$ denotes that the queen on row $i$ is placed in column $j$, where $j \in D_i$. The constraint is that $v_i = v_j$ would not cause a constraint violation. In backtracking algorithms this is often used to update the domains of unassigned variables as assignments are added and removed. We generalise this idea by associating with each value $x$ in the domain of each variable $v$ a conflict count $C_{v,x}$. The value of $C_{v,x}$ is the number of constraints that would be violated if the assignment $v = x$ were added. If $C_{v,x} = 0$ then $x$ is currently in the domain of $v$. We make a further generalisation by maintaining the conflict counts of all variables, assigned and unassigned. The meaning of a conflict count for a currently assigned variable is: $C_{v,x}$ is the number of constraints that would be violated if the variable $v$ were reassigned to $x$. Now on assigning or unassigning a variable, we incrementally update conflict counts in all other variable domains. This is clearly more expensive than standard forward checking, which only updates the domains of unassigned variables. However, the state of each variable domain is now independent of the order in which assignments were made, and we can unassign variables arbitrarily. Moreover, it applies to both binary and non-binary constraints, and has been used to combine IDB with unit resolution for satisfiability problems [Prestwich, 2000a, Prestwich, 2000b].

Section 2 evaluates the IDB on some well-known combinatorial problems, and Section 3 discusses the results and related work.

2 Experiments

In this section IDB is applied to three problems: $n$-queens, graph colouring and satisfiability. The experiments are performed on a 300MHz DEC AlphaServer 1000A 5/300 under Unix. For readers wishing to normalise our execution times to other platforms, the DL-MACS [Johnson and Trick, 1996] benchmark program $dfmax$ runs 462 seconds on the Alphaserver.

2.1 The $n$-queens problem

We first evaluate IDB on the $n$-queens problem. Though fashionable several years ago $n$-queens is no longer considered a challenging problem. However, large instances still defeat most backtrackers and it is therefore of interest. Consider a generalised chess board, which is a square divided into $n \times n$ smaller squares. Place $n$ queens on it in such a way that no queen attacks any other. (This is often called an $n$-queens attack another if it is on the same row, column or diagonal (in which case both attacks each other). We can model this problem using $n$ variables each with domain $D_i = \{1, \ldots, n\}$. A variable $v_i$ corresponds to a queen on row $i$ (there is one queen per row), and the assignment $v_i = j$ denotes that the queen on row $i$ is placed in column $j$, where $j \in D_i$. The constraint is that $v_i = v_j$ would not cause a constraint violation. In backtracking algorithms this is often used to update the domains of unassigned variables as assignments are added and removed. We generalise this idea by associating with each value $x$ in the domain of each variable $v$ a conflict count $C_{v,x}$. The value of $C_{v,x}$ is the number of constraints that would be violated if the assignment $v = x$ were added. If $C_{v,x} = 0$ then $x$ is currently in the domain of $v$. We make a further generalisation by maintaining the conflict counts of all variables, assigned and unassigned. The meaning of a conflict count for a currently assigned variable is: $C_{v,x}$ is the number of constraints that would be violated if the variable $v$ were reassigned to $x$. Now on assigning or unassigning a variable, we incrementally update conflict counts in all other variable domains. This is clearly more expensive than standard forward checking, which only updates the domains of unassigned variables. However, the state of each variable domain is now independent of the order in which assignments were made, and we can unassign variables arbitrarily. Moreover, it applies to both binary and non-binary constraints, and has been used to combine IDB with unit resolution for satisfiability problems [Prestwich, 2000a, Prestwich, 2000b].

Section 2 evaluates the IDB on some well-known combinatorial problems, and Section 3 discusses the results and related work.

2 Experiments

In this section IDB is applied to three problems: $n$-queens, graph colouring and satisfiability. The experiments are performed on a 300MHz DEC AlphaServer 1000A 5/300 under Unix. For readers wishing to normalise our execution times to other platforms, the DL-MACS [Johnson and Trick, 1996] benchmark program $dfmax$ runs 462 seconds on the Alphaserver.
time linear in the problem size.

It should be noted that IDB is not the only backtracking heuristic to perform local search on n-queens. Similar results were obtained by Minton et al.’s MCBT algorithm (see Figure 1), as well as others. Such algorithms rely on good value ordering heuristics. In MCBT an initial total assignment is generated by the MC heuristic and used to guide DFS in two ways. Firstly, variables are selected for assignment on the basis of how many violations they cause in I. Secondly, values are tried in ascending order of number of violations with currently unassigned variables, an example of a value ordering heuristic. This informed backtracking algorithm performs almost identically to MCHC on n-queens. However, MCBT is still prone to the same drawback as most backtrackers: a poor choice of assignment high in the search tree will still take a very long time to recover from. IDB is able to modify earlier choices, as long as the b parameter is sufficiently high, so it can recover from poor early decisions. This difference is not apparent on the n-queens problem, but it will be significant in problems for which value ordering heuristics are of little help.

2.2 Graph colouring

Graph colouring is a combinatorial optimisation problem with real-world applications such as timetabling, scheduling, frequency assignment, computer register allocation, printed circuit board testing and pattern matching. A graph \( G = (V,E) \) consists of a set \( V \) of vertices and a set \( E \) of edges between vertices. Two vertices connected by an edge are said to be adjacent. The aim is to assign a colour to each vertex in such a way that no two adjacent vertices have the same colour, using as few colours as possible.

The backtracking heuristic used for n-queens causes the search to stagnate on some colouring problems, so we use a modified heuristic: instead of selecting the assigned variable with largest domain, we randomly select a variable with domain size greater than one, if possible, otherwise randomly select an assigned variable. This weaker heuristic allows greater flexibility in the choice of backtracking variable, and gives better results. We also use forward checking but a different variable ordering heuristic: the Brežan heuristic [Brežan, 1979]. This selects an unassigned variable with minimum domain size, breaking ties by considering the degree in the backtracked part of the graph, and breaking further ties randomly. The Brežan heuristic is used in the well-known DSA TUR colouring algorithm (as well as in other constraint applications) and our IDB algorithm is a modified DSA TUR.

A common graph colouring benchmark is 3-colourable random graphs. [Pothen and Richards, 1995] compared the performance of MCHC with that of Weak Commitment Search (WCS) on these graphs, with 60 vertices and average degree between 3 and 7. Using a maximum of 2000 steps both algorithms had a lower success rate around the phase transition near degree 5: WCS (without Brežan heuristics or forward checking) dipped to about 80% while MCHC dipped to less than 5%. A modified MCHC allowing uphill moves performed much like WCS. We generate such graphs in the same way and apply IDB to them under the same conditions: over 1000 runs with tuned values of the parameter \( b \) (increasing from 1 to 9 near the phase transition) its success rate dips to 98.3% at degree 4.57. Hence IDB beats MCHC on at least some colouring problems.

Minton et al found that DSA TUR outperformed MCHC on these graphs. Jonsson and Ginsberg, 1993 found that depth-first search and Dynamic Backtracking both beat MCHC and the GSAT local search algorithm on similar graphs. To compare IDB with DSA TUR we therefore need harder benchmarks. In a previous paper [Prestwich, 1998] we used several graph types described in [Culberson et al, 1995], and compared IDB with a combination of IDB with DIMACS graphs, and compared IDB with M. Trick’s DSA TUR implementation which also performs some preprocessing of graphs by finding a clique, an enhancement which does not have. We use two types of graph: random and geometric. Random graphs are a standard benchmark and have little structure for constraint algorithms to exploit. Geometric graphs tend to have large cliques, which should give DSA TUR an advantage. Even without the clique enhancement, DSA TUR works quite well on geometric graphs [Culberson et al, 1995], so this is a challenging test for IDB.

Figure 2 shows the results. IDB results are means over 10 runs and started from the colouring shown plus

\[\text{http://mat.gsia.cmu.edu/COLOR/color.html}\]
On finding each colouring IDB was restarted with one less colour available, and tried to reuse previous assignments. DSA TUR’s results are obtained from a single run because its heuristics are deterministic. We shall also mention results for five other algorithms, whose results are contained in [Joslin and Clements, 1999] and not shown here for space reasons. They are Squeaky Wheel Optimization, TABU search (both general-purpose combinatorial optimisation algorithms), Iterative Greedy, distributed IMP ASSE and parallel IMP ASSE (all pure colouring algorithms).

On the sparse geometric graphs (\( \text{DSJC}^{*\cdot 1} \)) IDB and DSA TUR are both very fast, often requiring no backtracks at all. The other five algorithms also found these problems easy. However, on the denser geometric graphs (\( \text{DSJC}^{*\cdot 2} \)) IDB consistently finds the same or better colourings in shorter times than DSA TUR or any of the other algorithms. On two of the graphs IDB found a better colouring than any of the other algorithms: the previous best for DSJR500.5 was distributed IMP ASSE with 123 in 94.1 seconds (time normalised to our platform), and that for R1000.5 was Squeaky Wheel Optimization with 239 in 309 seconds. In some runs IDB finds a 234-colouring for R1000.5. IDB clearly inherits DSA TUR’s efficient handling of cliques but has local search-like scalability.

On the random graphs (DSJC^{*\cdot 5}) IDB also finds consistently better colouring than DSA TUR. Note that though DSA TUR’s times are sometimes shorter, it was given 5 minutes to find a better colouring but did not; it tended to find a reasonable colouring quickly, then made little further progress. However, IDB is not the best algorithm on the random graphs, beating only TABU and performing roughly like Iterative Greedy.

<table>
<thead>
<tr>
<th>graph</th>
<th>DSA TUR</th>
<th>IDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSJC^{*\cdot 1}</td>
<td>5 0.0</td>
<td>5 0.01 1</td>
</tr>
<tr>
<td>DSJC^{*\cdot 5}</td>
<td>65 2.9</td>
<td>65 0.33 2</td>
</tr>
<tr>
<td>DSJC1000.5</td>
<td>100 15.6</td>
<td>122 1.99 7</td>
</tr>
<tr>
<td>DSJC1250.5</td>
<td>209 58.6</td>
<td>235 0.61 6</td>
</tr>
<tr>
<td>DSJC2500.5</td>
<td>35 198</td>
<td>32 0.31 1</td>
</tr>
<tr>
<td>DSJC500.5</td>
<td>63 5.8</td>
<td>59 0.41 1</td>
</tr>
<tr>
<td>DSJC1000.5</td>
<td>114 5.0</td>
<td>107 13.2 1</td>
</tr>
</tbody>
</table>

Figure 2: Results on selected DIMACS benchmarks

2.3 Satisfiability

The satisfiability (SAT) problem is of both practical and theoretical importance. It was the first problem shown to be NP-complete, has been the subject of a great deal of recent research and has well-known benchmarks and algorithms, making it ideal for evaluating new approaches. The SAT problem is to determine whether a Boolean expression has a satisfying set of truth assignments. Modern systematic SAT solvers are still based on this scheme and include TABLEAU [Crawford and Auton, 1996], RELSAT [Bayardo and Schrag, 1997], POSIT [Freeman, 1994], SATO [Zhang, 1997], GRASP [Siva and Sakallah, 1996] and SATZ [Li and Ambulagun, 1997]. The main difference between these solvers is their variable ordering heuristics, though SATZ also has a preprocessing phase and RELSAT uses look-back techniques. There are also several local search SAT algorithms. Early examples are GSAT [Selman et al, 1992] and Gu’s algorithms [Gu, 1992]. GSAT has since been enhanced in various ways [Selman et al, 1994] giving several WSAT variants. Other local search SAT algorithms include DLM [Wu and Wah, 2000] and GRASP [Resende and Foscope, 1990]. DLM is based on Lagrange multipliers, while GRASP is a greedy, randomised, adaptive algorithm.

IDB has already been applied to SAT by modifying a simple Davis-Logemann-Loveland procedure. New heuristics are now described, which improve performance on some problems and allow a convergence property to be proved [Probabilistic Approximate Completions [Hocs, 1998], proof omitted for space reasons]. The new variable ordering heuristic randomly selects a variable with domain size 1. If there are none then it selects the variable that was unassignable at the last dead-end, and subsequently (until the next dead-end) the most recently unassigned variable. For backtracking the variable with lowest conflict counts is selected, breaking ties randomly. A probabilistic value ordering rule is used as follows. Each variable records its last assigned value (initially set to a random truth value). For each assignment, with probability \( 1 - 1/\mu \) the negated value is preferred, and with probability \( 1 - (1/\mu) \) the value itself, where \( \mu \) is the number of currently unassigned variables. Finally, the integer parameter \( b \) is replaced by a parameter \( p \) (0 \( \leq p < 1 \))
which is used as follows. At each dead-end a variable is unassigned as before, and with probability \( p \) further variables are iteratively unassigned.

In [Prestwich, 2000a] IDB was shown to scale almost identically to WSAT on hard random 3-SAT problems, and in [Prestwich, 2000b] it was shown to efficiently solve three classes of problem considered hard for local search: all-interval series, parity learning and some AIM problems (though DLM has recently been shown to perform efficiently on these problems [Wu and Wah, 2003]). It also beat backtracking, local search and some hybrids on a set of geometric graph colouring problems. Figure 3 shows the performance of the new IDB algorithm for SAT on benchmarks from three sources: the 2nd DIMACS Implementation Challenge, 2 the SATLIB problem repository, 3 and the International Competition and Symposium on Satisfiability Testing in Beijing. 4 It has similar performance to the previous implementation on most of the problems already tested, but is several times faster on these problems. We believe that it can be further improved by using variable ordering heuristics taken from backtrackers.

Random 3-SAT is a standard benchmark for SAT algorithms, the hard problems being found near the phase transition. Backtracking can efficiently solve such problems only up to a few hundred variables [Crawford and Auton, 1999] and we were unable to solve 1600 with SATZ, but local search algorithms can solve much larger problems [Selman et al., 1994, Wu and Wah, 2003]. The results verify that IDB has the scalability of local search, though WSAT is faster and the latest DLM implementation can solve 2000 in 19.2 seconds (on a 400MHz Pentium-II processor).

Circuit fault analysis SAT problems are harder for backtracking than for local search. The IDB results are means over 20 runs for all four DMACS problems denoted by [hanoi/4]. It has similar performance to WSAT, is slower than DLM, and faster than the GRASP local search algorithm and the backtrackers, thus ranking second of eight algorithms.

Circuit synthesis SAT problems are also hard for backtrackers. The IDB times are means over 100 runs. On the benchmarks 16*1 and 132*1 it is slower than DLM and GSAT with random walk but faster than the GRASP local search algorithm, hence performing like a reasonably efficient local search algorithm. On the benchmarks 2bitadd/1 etc., IDB times are means over 20 runs. It has similar performance to that of GSAT with random walk and is many times faster than the backtrackers SATZ, GRASP, POSIT and RELSAT on the 3bitadd problems. However, WSAT is several times faster.

Figure 3: IDB on selected SAT benchmarks

<table>
<thead>
<tr>
<th>Problem</th>
<th>Time (s)</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000</td>
<td>4.35</td>
<td>0.01</td>
</tr>
<tr>
<td>1000</td>
<td>84.1</td>
<td>0.01</td>
</tr>
<tr>
<td>2000</td>
<td>686</td>
<td>0.01</td>
</tr>
<tr>
<td>2bitcomp</td>
<td>0.002</td>
<td>0.98</td>
</tr>
<tr>
<td>3bitadd</td>
<td>0.025</td>
<td>0.98</td>
</tr>
<tr>
<td>2bitadd</td>
<td>0.028</td>
<td>0.98</td>
</tr>
<tr>
<td>2bitadd</td>
<td>11.0</td>
<td>0.98</td>
</tr>
<tr>
<td>3bitadd</td>
<td>8.6</td>
<td>0.98</td>
</tr>
<tr>
<td>bLarge</td>
<td>0.425</td>
<td>0.9</td>
</tr>
<tr>
<td>bLarge</td>
<td>7.75</td>
<td>0.95</td>
</tr>
<tr>
<td>bLarge</td>
<td>48.9</td>
<td>0.98</td>
</tr>
<tr>
<td>bLarge</td>
<td>29</td>
<td>0.97</td>
</tr>
<tr>
<td>hanoi/4</td>
<td>41.66</td>
<td>0.98</td>
</tr>
</tbody>
</table>

The only algorithm apart from IDB currently able to solve the hanoi/4 problem is DLM, which takes a mean of 615 seconds over 10 runs on a Pentium-III 500MHz with Solaris 7. Hence IDB is currently the fastest known algorithm on hanoi/4.

3 Discussion

The results in this and previous papers show that IDB can solve problems considered hard for backtrackers or for local search, and sometimes out-performs both. N-queens is easy for local search and IDB but hard for backtracking. Some SAT problems are hard for local search and others for backtracking, but IDB is able to solve both. On some graph colouring problems IDB beats both backtracking and local search. A modified constraint solver scaled better to large Golomb Ruler problems than its systematic counterpart and also beat a genetic algorithm [Prestwich, 2001]. A modified branch-and-bound algorithm found optimal solutions to a hard optimisation problem faster than...
the systematic version, whereas a variety of stochastic search algorithms were unable to find optimal solutions [Prestwich, 2000b]. Very competitive results were also obtained for maximum clique benchmarks [Prestwich, 2001]. IDB’s strength is large, structured, solvable problems that are too large for backtracking yet unsuitable for local search.

Should IDB be classified as a backtracker, as local search, as both or as a hybrid? It may appear to be simply an inferior version of Dynamic Backtracking (DB), sacrificing the important property of completeness to no good purpose. A counter-example to this view is the n-queens problem, on which DB performs like chronological backtracking [Jonsson and Ginsberg, 1993], whereas IDB performs like local search. Another is the 3-SAT problem, where DB is slower than depth-first search (though a modified DB is similar) [Baker, 1994] while IDB again performs like local search. We claim that IDB is in fact a local search algorithm, and in previous papers it has been referred to as Constrained Local Search. Our view is that it stochastically explores a space of consistent partial assignments while trying to minimise an objective function: the number of unsatisfied variables. The forward local moves are variable assignments, while the backward local moves are unassignments (backtracks). IDB can also be viewed as a hybrid, applying a local search strategy to the space of states normally explored by backtracking, or as a way of adding constraint handling to local search. However, to some extent the question is academic: even if IDB is not local search, results show that it captures its essence.

Other researchers have designed algorithms using backtracking techniques but with improved scalability. Iterative Sampling [Langley, 1992] restarts a constructive search every time a dead-end is reached. Bounded Backtrack Search [Harvey, 1995] is a hybrid of Iterative Sampling and chronological backtracking, alternating a limited amount of chronological backtracking with random restarts. [Cromes et al., 1998] periodically restart chronological or intelligent backtracking with slightly randomised heuristics. [Ginsberg and McAllester, 1994] describe a hybrid of the GSAT local search algorithm and Dynamic Backtracking that increases the flexibility in choice of backtracking variable. However, they note that to achieve total flexibility while preserving completeness would require exponential memory, and they recommend a less flexible version using only polynomial memory. Weak Commitment Search [Yokoo, 1994] builds consistent partial assignments, using greedy search (the min-conflict heuristic) to guide value selection. On reaching a dead-end it restarts and uses learning to avoid redundant search. Local Changes [Verfaillie and Schiex, 1994] is a complete backtracking algorithm that uses conflict analysis to unassign variables leading to constraint violation, and a heuristic similar to our value ordering heuristic that restores assignments after backtracking. Limited Discrepancy Search [Harvey, 1995] searches the neighbourhood of a consistent partial assignment, trying neighbours in increasing order of distance from the partial assignment. It is shown theoretically, and experimentally on job shop scheduling problems, to be superior to Iterative Sampling and chronological backtracking.

There are also many hybrid approaches. Schaefer. 1997 describes an algorithm that searches the space of all partial assignments, which is larger than the space searched by IDB (the consistent partial assignments). The objective function to be minimised includes a measure of constraint violation, whereas IDB never violates a constraint. The Path-Repair Algorithm [Jussien and Lhomme, 1999] is a generalisation of Schaefer’s approach that includes learning, allowing complete versions to be devised. Two-phase algorithm of [Zhang and Zhang, 1996] searches a space of partial assignments, alternating backtracking search with local search. [Ye et al., 1997] generate partial assignments to key variables by local search, then pass them to a constraint solver that checks consistency. [Pessant and Gendreau, 1998] use branch-and-bound to efficiently explore local search neighbourhoods. Large Neighbourhood Search [Shaw, 1998] performs local search and uses backtracking to test the legality of moves. [Crawford, 1993] uses local search within a complete SAT solver to select the best branching variable.

Future work will include investigating whether IDB can be made complete, thus combining the advantages of systematic and non-systematic search. This could be achieved by learning techniques, but memory requirements make this approach impractical for very large problems.

References


