ELSEVIER

Blocks World revisited

John Slaney [a,*,1], Sylvie Thiébaux [b,2]

[a] *Computer Sciences Lab, Australian National University, Canberra, Australia*
[b] *CSIRO Mathematical & Information Sciences, PO Box 664, Canberra, Australia*

Abstract

Contemporary AI shows a healthy trend away from artificial problems towards real-world applications. Less healthy, however, is the fashionable disparagement of "toy" domains: when properly approached, these domains can at the very least support meaningful systematic experiments, and allow features relevant to many kinds of reasoning to be abstracted and studied. A major reason why they have fallen into disrepute is that superficial understanding of them has resulted in poor experimental methodology and consequent failure to extract useful information. This paper presents a sustained investigation of one such toy: the (in)famous Blocks World planning problem, and provides the level of understanding required for its effective use as a benchmark. Our results include methods for generating random problems for systematic experimentation, the best domain-specific planning algorithms against which AI planners can be compared, and observations establishing the average plan quality of near-optimal methods. We also study the distribution of hard/easy instances, and identify the structure that AI planners must be able to exploit in order to approach Blocks World successfully. © 2001 Elsevier Science B.V. All rights reserved.

Keywords: Blocks World; Planning benchmarks; Random/hard problems; Approximation algorithms

1. Introduction

1.1. Blocks World

The Blocks World (BW) consists of a finite number of blocks stacked into towers on a table large enough to hold them all. The positioning of the towers on the table is irrelevant.
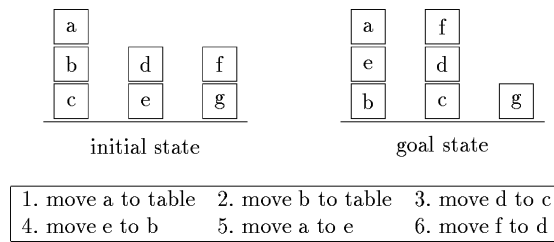

Fig. 1. BW planning problem and optimal plan.

The BW planning problem is to turn an initial state of the blocks into a goal state, by moving one block at a time from the top of a tower onto another tower or to the table (see Fig. 1). The optimal BW planning problem is to do so in a minimal number of moves.

BW planning turns out to be tractable and optimal BW planning NP-hard [6,13,14]. Further, optimal BW planning is Max-SNP hard, meaning that there exists some fixed performance ratio under which it cannot be approximated tractably [26]. Near-optimal planning within a ratio of 2 is tractable [14], but the question of whether a better ratio is achievable in polynomial time remains open [26]. For a number of extensions of the basic BW above (table with a limited capacity, blocks with identical names, . . .) similar complexity results can be shown [14], except for approximation within a constant ratio, which is not always tractable [26].

1.2. Motivations behind this paper

Artificial domains such as the Blocks World, the Traveling Salesman and the Queens, are in themselves of little practical interest. Despite this, they have remained staples of the AI literature over 30 years, because they are hard for general purpose AI systems and, at least in principle, can support meaningful, systematic and affordable experiments. More recently, BW seems to have fallen into disrepute. In part, this is due to the efforts of the AI planning community to address domains of more practical relevance. More importantly however, it is due to superficial understanding of the domain, leading to poor experimental methodology and consequent failure to extract useful results from it.

Three major gaps in existing knowledge of BW prevent it from being effectively used as a benchmark. The first is the lack of knowledge about how to construct problem instances that are suitable for systematic experimentation: for example hard instances, or uniformly distributed random ones. This seriously lowers the value of experiments. Results are commonly obtained on isolated BW instances, often unspecified except for the number of blocks involved, and presumably hand-crafted to show off the good points of a particular system (here we refrain from pointing the finger at any particular example in the literature). Matters have started to improve with attempts to create some benchmarks for planning. [3] In particular Kautz and Selman have introduced a series of four BW problems called 'bw_large.a' to 'bw_large.d', which have now attained that status [17,18]. This is

---

[3] The AIPS planning competition (http://www.cs.yale.edu/~mcdermott.html and http://www.cs.toronto. edu/aips2000) is a symptom of the perceived need for standards.

good at least in that everyone's system is solving the same problem, but as we shall show, these problems are atypical in several respects, not least in that they have abnormally short plans, and are easy to solve optimally. The alternative to experimenting on particular instances is to use random problems, but when this has been done, it has usually been done poorly. BW optimisation problems of a given size vary so much in difficulty that "five random problems" are hardly a statistically adequate base for strong conclusions. Moreover, no attention has been paid to the distribution of the random instances, although it greatly affects the results. Even in the AIPS 2000 competition, perhaps the most carefully constructed testbed for planners, the largest BW problems are again atypical and easy to solve optimally. We believe that problem selection will improve only when the features making instances hard, easy or average are identified, and when software for generating random instances with meaningful distributions is used.

The second important gap lies in the lack of knowledge about the performance in time and solution quality of BW-specific planning methods. It was not until the 1990s that the *worst-case* complexity of optimal BW planning was studied by Chenoweth [6], Gupta and Nau [13,14] and Selman [26], and much remains to be done. Very little is known about the *average* time complexity of optimal BW planning. No results are available about average performance ratios for near-optimal planning. Also, the exact time complexity of near-optimal BW planning has not been carefully analysed, [14] and [1,6] reporting respectively cubic-time and quadratic-time upper bounds in the number of blocks. As a consequence, no proper BW "gold standard" exists, and this makes it hard to assess the effectiveness of approaches to planning on the basis of results reported in the literature. For instance, [1,2,7,18] show how specific methods for near-optimal BW planning can be fruitfully encoded in a general system, while [17] exhibits domain-independent techniques that dramatically improve performance for BW planning. These facts should not be interpreted as showing that these systems are really effective for domains like BW unless they match the best domain-specific ones, both in time complexity and in solution quality. As long as little is known about the behaviour of BW-specific methods, misinterpretation is easy.

Only recently have planning systems been able to deal with more than a few blocks— a fact which has hardly enhanced the reputation of the field within the wider research community. This was largely due to a third gap: ignorance of BW-specific features that general systems should be able to exploit if they are to do well with the domain. The papers by Bacchus and Kabanza [1,2] clearly show that attempting to bridge this gap greatly improves not only planner performance but also the ability to interpret the experimental results. Indeed, without a precise understanding of which information is relevant for a domain and which is not, the reasons why a given general technique performs well on this domain cannot be adequately analysed, nor is it possible to know whether the domain is a suitable testbed for analysing the merits of that technique. Recent improvements notwithstanding, the performance of general planners, especially on optimal BW problems, remains far from comparable with domain-specific ones. This indicates that a more detailed examination of the structure of BW problems is likely to reveal features which will further improve planner performance and experimental methodology.

1.3. Contributions of the paper

In this paper we undertake a sustained investigation of BW, with a view to filling all three gaps. Much of the material presented is compiled from a series of conference papers [31,32] and technical reports [27–30] written over a number of years.

In Section 2 we look at BW itself, giving expressions for the number of BW states as a function of the number of blocks and showing how to use this function to generate random states with uniform distribution. A supply of such states is essential to experiments on average-case behaviour, yet the problem of generating them is not trivial and has not previously been addressed.

In Section 3 we consider algorithms both for optimal and for near-optimal BW planning. We detail several methods for finding near-optimal plans in linear time, thus closing the complexity question for this problem. We also outline an optimal solver capable of dealing with arbitrary problems of up to 150 blocks. The resulting programs are offered as a reference point for assessing the effectiveness of planning systems on this domain.

In Section 4 we first examine the average performance of the algorithms in terms of speed and, more importantly, of solution quality. Having linear-time implementations means that we can find near-optimal BW plans for problems on the order of a million blocks in a matter of seconds, enabling us to consider much larger problems than was previously possible. We present evidence that the average performance ratios of these algorithms are everywhere much better than the worst case of 2, and may even approach 1 in the limit. Since the difference between plans produced by the crudest algorithm and optimal plans is so small, claims that a system produces suboptimal plans of "high quality" must be interpreted carefully.

Using our optimal algorithm, we then turn to discovering which optimisation problems are hard and which are easy. It emerges that the hard problems are those in a critically constrained range for which the number of towers in the initial and goal states is a reasonably good predictor. We conclude Section 4 by reconsidering the structure of BW planning problems in the light of these experiments, with a view to determining how they should be approached by domain-independent planning systems.

Finally, we discuss the future of BW as a testbed for comparing planners and as an object of study of relevance to general-purpose planning. Naturally, it would not do to study *only* artificial domains such as the blocks, but we present this paper as evidence that there is still much to learn from them.

2. The states of Blocks World

2.1. Definitions

We shall first enter some definitions. We assume a finite set B of blocks, with TABLE as a special member of B which is not on anything. Our favourite presentation of BW takes as primitive not the binary relation ON but the unary "support" function S which picks out, for block x, the block which x is on. Thus S is a partial function from $B \setminus \{\text{TABLE}\}$ to B, injective except possibly at TABLE and such that its transitive closure is irreflexive. We

refer to the pair $\langle B, S \rangle$ as a *part-state*, and identify a *state* of BW with such a part-state in which S is a total function. Clearly, in the latter case, a specification of the support function completely determines the disposition of the blocks.

For a part-state $\sigma = \langle B, S \rangle$ and for any a and b in B, we define: $\text{ON}_\sigma(a, b)$ iff $S(a) = b$, $\text{CLEAR}_\sigma(a)$ iff either $a = \text{TABLE}$ or $\neg \exists b \, (\text{ON}_\sigma(b, a))$, ABOVE_σ as the transitive closure of ON_σ, and finally $\text{POSITION}_\sigma(a)$ as the sequence $\langle a :: \text{POSITION}_\sigma(S(a)) \rangle$ if $S(a)$ exists and $\langle a \rangle$ otherwise. That is, the *position* of a block is the sequence of blocks at or below it. We refer to the position of a clear block as a *tower*. A tower is *grounded* iff it ends with the table. Note that in a state (as opposed to a mere part-state) every tower is grounded.

2.2. Number of Blocks World states

One of our first questions on encountering BW was how many different states it has, as a function of the number of blocks. Answering this question turns out to be more useful than it may appear at first glance. For instance, Schoppers uses an estimate of the number of states to argue about the compactness of universal BW plans [25]. As another example, we shall see later how such a derivation provides important insight into the generation of random BW states. At least to judge by the number of people who have asked us for our magic formula, there does not seem to be any analytical definition published in the literature. To our knowledge, only algorithmic descriptions of the computation of the number of states as well as exact or approximate figures for small number of blocks have been reported, see, e.g., [18,25].

Instead of merely counting the number of BW states of a given size, we shall in fact consider a slightly more general problem. We will count the number of states that extend a part-state in which there are k grounded towers and n ungrounded ones. Let this be $g(n, k)$. As a special case, the number of BW states of size n is $f(n) = g(n, 0)$. We will give two definitions of g: a recursive one, and an iterative one, each of which has its uses.

The recursive definition of g is quite simple and gives insight into the generation of random states. First $g(0, k) = 1$ for all k, since if $n = 0$ then every tower is already grounded and the part-state is already a state. Now consider $g(n + 1, k)$. To extend the part-state, the first ungrounded tower must go on something: either on the table or on one of the $n + k$ other towers. If it goes on the table, that gives a part-state with n ungrounded towers and $k + 1$ grounded ones, which has then $g(n, k + 1)$ possible extensions. Otherwise, there are $n + k$ ways of placing it on one of the $n + k$ other towers, each of which leaves a part-state with n ungrounded towers and k grounded ones, which has then $g(n, k)$ possible extensions. In sum:

$$g(0, k) = 1,$$
$$g(n + 1, k) = g(n, k + 1) + (n + k)g(n, k).$$

Table 1 gives the value of $g(n, k)$ for small n and k. Since the value of $g(n, k)$ is produced by the values of the $g(x, y)$ for $\langle x, y \rangle$ in the triangle bounded by the vertical column above $\langle n, k \rangle$ and by the diagonal going up and right from $\langle n, k \rangle$, Table 1 is reminiscent of Pascal's triangle, but with multiplication thrown in. It is worth noting that these numbers rapidly become big. For instance $f(30) = 1979874012955717189150065982397968511$.

Table 1
The function g and the triangle for $g(3, 1)$

k	0	1	2	3	4	5
n						
0	1	1	1	1	1	1
1	1	2	3	4	5	6
2	3	7	13	21	31	43
3	13	34	73	136	229	358
4	73	209	501	1045	1961	3393
5	501	1546	4051	9276	19081	36046
6	4051	13327	37633	93289	207775	424051
7	37633	130922	394353	1047376	2501801	5470158
8	394353	1441729	4596553	12975561	32989969	76751233
9	4596553	17572114	58941091	175721140	472630861	1163391958

The iterative definition of g is less intuitive, but forms a better basis from which to derive average-case figures for BW states. Let us decide that we are going to extend the part-state by putting i of the n ungrounded towers on the table and the remaining $n - i$ on other towers. There are $\binom{n}{i}$ ways of choosing the i towers. For the first of the remaining $n - i$ towers, there are $n + k - 1$ ways of choosing the tower it is going to be on. For the second, there are $n + k - 2$ ways, and for the last one, $i + k$ ways. So altogether, there are $(n + k - 1)!/(i + k - 1)!$ ways of placing the $n - i$ towers on others. Summing for all i yields:

$$g(n, k) = \sum_{i=0}^{n} \binom{n}{i} \frac{(n + k - 1)!}{(i + k - 1)!}.$$

Naturally, the two definitions of g can be proven equivalent (see [28, pp. 5–6]). The proof resembles the derivation of the combinatorial interpretation of binomial coefficients from the binomial addition formula. In fact, it is legitimate to ask whether the values of g correspond to any well-known sequences of numbers, such as binomial coefficients, Stirling numbers, and the like. The answer is that they nearly do. There exists a simple expression of g in terms of a Laguerre polynomial: [4]

$$g(n, k) = n! L_n^{k-1}(-1)$$

the advantage being that we can help ourselves to the known mathematics of these polynomials. In [29, pp. 23–26] we do so to show that the average number of towers in a BW state of size n tends to $\sqrt{n}$.

---

[4] For an overview of these polynomials, see [21, pp. 76–97]. The Laguerre polynomial $L_n^k(x)$ is defined as $\sum_{i=0}^{n} \binom{n}{i} \frac{(n+k)!}{n!(i+k)!} (-x)^i$.

The iterative version of g is also well suited to derive a simple expression for the number of states with exactly t towers, or more generally for the number of states with exactly t towers extending a part state with k grounded towers and n ungrounded ones. Let us call this $h(n, k, t)$. If k towers are already grounded, we must put exactly $i = t - k$ towers on the table, and the rest onto other towers. So it suffices to omit the sum from the iterative definition of g, and to replace i with $t - k$:

$$h(n, k, t) = \binom{n}{t - k} \frac{(n + k - 1)!}{(t - 1)!} \text{ if } t \geqslant k \text{ and } 0 \text{ otherwise.}$$

We are indebted to an anonymous reviewer of this paper for yet another definition of the number of states f, which does not require a generalisation to the binary function g, and which lends itself to more efficient algorithms for generating states. Let $c(n)$ be the number of states of n blocks in which a given block (say, the nth) is clear. Obviously $c(1) = 1$. Now for the recursive definition, suppose we know $f(n)$ and $c(n)$ and consider adding one more block. The states of $n + 1$ blocks comprise those in which the extra block is under one of the blocks in a state of n blocks ($n \cdot f(n)$ of those), those in which the extra block is clear and on the table ($f(n)$ of those) and those in which the extra block is placed on a clear block in a state of n blocks. In the last case, note that there are n blocks on which the extra one could be, and each is clear in $c(n)$ states. The extra block is clear in all states except for the first $n \cdot f(n)$. In sum:

$$f(n + 1) = f(n) + n \cdot c(n) + n \cdot f(n),$$

$$c(n + 1) = f(n) + n \cdot c(n).$$

Again it is not difficult to generalise the definition from that of all states to that of states with exactly t towers, providing an alternative to h, but we refrain from going into details here because that generalisation will not be used in what follows.

2.3. Random Blocks World states

Armed with the g function, or with f and c, we can now generate uniformly distributed random BW states and problems (pairs of random states), and with h we can generate random states with a given number of towers. The latter is particularly useful, because as we shall see in Section 4, the difficulty of optimal BW planning varies with the number of towers.

Surprisingly few papers mention experiments on random BW problems, and in those which do, the underlying distribution is left obscure (see, e.g., [22, p. 180], [1,9,15,23]). Perhaps the reason is that generating uniformly distributed random BW problems is not as trivial as it may appear. Naïvely incorporating a random number generator into an algorithm for producing the states will not work: typically, it makes some states more probable than others by a factor exponential in the number of blocks. As a result, the sample is skewed and experiments on the average case may be biased. We invite the skeptical reader to try by hand the case $n = 2$ with a view of generating the three possible states with

a probability of $1/3$. Most methods, including the random BW states generator included in the distribution of UCPOP,[5] give probabilities $1/2$, $1/4$ and $1/4$.

The UCPOP generator, for instance, uses the following method to produce a BW state of size n:

(1) start with an empty table and n blocks to be placed,
(2) repeat until all n blocks have been placed:
   (2a) select with probability $1/\phi$ one of the ϕ blocks that have not yet been placed,
   (2b) select with probability $1/(\tau + 1)$ the table or one of the τ towers being grounded at this stage, and place the selected block on it.

So with two blocks for example, the probability of both blocks being in a single tower in ascending order is $1/4$, and that of both blocks being on the table is $1/2$. With n blocks, the probability of all blocks being on the table is $1/n!$, while that of all blocks being in one tower in ascending order is $1/(2^{(n-1)}n!)$, that is, exponentially smaller. It can be shown that the average number of towers in a state generated this way tends to $\sqrt{2n}$ and not $\sqrt{n}$. So since the number of towers affects the difficulty of problems, average-case experiments on the performance of algorithms will be biased.

The skew in the above method comes from the fact that placement on the table should not have the same probability as placement on another block. That is, among the states extending a given part-state, the proportion in which the selected block is on the table is not the same as the proportion in which it is on a given block. Therefore, the key to getting the probabilities right is to use the g function for counting the possible extensions of the part-state. We build a BW state of size n as follows:

(1) start with an empty table and n ungrounded towers each consisting of a single block,
(2) repeat until all towers are grounded:
   (2a) arbitrarily select one of the ϕ yet ungrounded towers,
   (2b) select the table with probability $g(\phi - 1, \tau + 1)/g(\phi, \tau)$ or one of the other towers (grounded or not) each with probability $g(\phi - 1, \tau)/g(\phi, \tau)$, and place the selected ungrounded tower onto it.

In fact, it is hard to work with g directly, as the numbers rapidly become too large. It is better to work with the ratio $R(\phi, \tau) = g(\phi - 1, \tau + 1)/g(\phi - 1, \tau)$ between the probabilities of the two types of placements. This is always a fairly small real number lying roughly in the range $[1, \sqrt{\phi}]$.[6] Elementary calculation shows that $R(1, \tau) = 1$ for all τ and that:

$$R(\phi + 1, \tau) = \frac{R(\phi, \tau)(\phi + \tau + R(\phi, \tau + 1))}{\phi - 1 + \tau + R(\phi, \tau)}$$

so no value of g need ever be computed.

This method is implemented in the BWSTATES program[7] [27]. BWSTATES first calculates in quadratic time all the values of R relevant to the size of the problems being generated and stores them using quadratic space. Once this is done, generating a random state is a linear-time problem. However, the space requirements impose a limit of the order

---

[5] http://www.cs.washington.edu/research/projects/ai/www/ucpop.html.

[6] In the limit, as ϕ becomes much larger than τ it converges to $\sqrt{\phi}$. On the other hand, where τ is much larger than ϕ the limiting value is 1 [29, pp. 23–25].

[7] BWSTATES is available from http://arp.anu.edu.au/~jks/bwstates.html.

Begin with n ungrounded blocks (i.e. $\phi = n$) and an empty table

Repeat until all towers are grounded

    Arbitrarily select an ungrounded tower

    With probability $c(\phi)/f(\phi)$ do:

        /* the tower constitutes the top of a grounded tower */

        Repeat until the given tower is grounded:

            Select the table with probability $f(\phi)/(f(\phi) + \phi c(\phi))$
            or one of the other ungrounded towers each with equal
            probability, and put the given tower on it.

            Decrement ϕ

    else (that is, with probability $1 - c(\phi)/f(\phi)$) do:

        /* the tower goes below another tower */

        Select one of the other ungrounded towers, each with
        equal probability, and put it on the given tower.

        Decrement ϕ

Fig. 2. Algorithm to generate random states using the c and f functions.

of 10000 blocks. The alternative, i.e., generating states without precomputing and storing R values, would be even more limiting since it would require a time cubic in the number of blocks.

The same principles may be applied to generate uniformly distributed states with exactly t towers, except that the h function should be used in place of g. So step (2b) reads: select the table with probability $h(\phi - 1, \tau + 1, t)/h(\phi, \tau, t)$ or one of the other towers each with probability $h(\phi - 1, \tau, t)/h(\phi, \tau, t)$, and place the selected ungrounded tower onto it. Here calculations are made much easier, because the probability of a placement on the table simplifies to $(t - \tau)/\phi$, and that of the other placements to $(\phi + \tau - t)/(\phi(\phi + \tau - 1))$. So these may be computed directly, without going through the ratio R, and the generation of a state takes linear time in the number of blocks.

Although the g function makes it easy to explain the skew in other random generators such as that in UCPOP and although it is used in the 1995 release of BWSTATES, it is not the most efficient basis for producing random states with uniform distribution. For certain of the experiments reported below, we required a supply of uniformly distributed states with up to a million blocks, which are beyond what can be done using g. Fortunately, the c and f functions lead naturally to a linear-time algorithm as shown in Fig. 2.

As in the case of the algorithm based on the g function, it pays to pre-compute and store the ratios required, in this case $P(\phi) = c(\phi)/f(\phi)$, for all $\phi < n$. In fact, $P(\phi)$ is close to $1/\sqrt{\phi}$, so again these are convenient-sized real numbers and it is much easier to work with them than with f and c directly. The recursive calculation of P is very easy: $P(1) = 1$ and

$$P(\phi + 1) = \frac{\phi P(\phi) + 1}{\phi(P(\phi) + 1) + 1}.$$

What makes this algorithm more efficient than that based on g is just that computing and storing P takes linear resources instead of the quadratic ones required for R.

We now leave the topic of random BW states. We shall return to it in Section 4 when making empirical observations on the average-case behaviour of planning algorithms, but for now we shall focus on BW planning itself.

3. Blocks World planning

3.1. Definitions

We first define the classical notions of problems,[8] moves, and plans. A BW planning *problem* over B is a pair of states $\langle\langle B, S_1\rangle, \langle B, S_2\rangle\rangle$. In problem $\langle I, G\rangle$, I is the *initial* state and G is the *goal* state. Here we consider only problems with completely specified goal states. A *move* in state $\sigma = \langle B, S\rangle$ is a pair of blocks $m = \langle a, b\rangle$ with $a \in B\backslash\{\text{TABLE}\}$ and $b \in B\backslash\{a\}$, such that $\text{CLEAR}_\sigma(a)$, $\text{CLEAR}_\sigma(b)$ and $\neg\text{ON}_\sigma(a, b)$. The result of m in σ is the state $\text{RES}(m, \sigma) = \langle B, S'\rangle$ where $S'(a) = b$ and $S'(x) = S(x)$ for $x \in B\backslash\{\text{TABLE}, a\}$. A *plan* for BW problem $\langle I, G\rangle$ is a finite sequence $\langle m_1, \ldots, m_p\rangle$ of moves such that either $I = G$ and $p = 0$ or else m_1 is a move in I and $\langle m_2, \ldots, m_p\rangle$ is a plan for $\langle\text{RES}(m_1, I), G\rangle$ in virtue of this definition. A plan for a given problem is *optimal* iff there is no shorter plan for that problem.

In order to discuss particular BW planning algorithms in the remainder of this paper we need a few further notions. We say that a block whose position in I (as defined earlier) is different from its position in G is *misplaced* in $\langle I, G\rangle$, and that one which is not misplaced is *in position*. Only misplaced blocks have to be moved in order to solve a problem. Next, we say that a move $\langle a, b\rangle$ is *constructive* in $\langle I, G\rangle$ iff a is in position in $\langle\text{RES}(\langle a, b\rangle, I), G\rangle$. That is, iff a is put into position by being moved to b. Once a block has been moved constructively, it need not move again in the course of solving the problem.

If no constructive move is possible in a given problem we say that the problem is *deadlocked*. In that case, for any misplaced block b_1 there is some block b_2 which must be moved before a constructive move with b_1 is possible. Since the number of blocks is finite the sequence $\langle b_1, b_2 \ldots\rangle$ must eventually loop. The concept of a deadlock, adapted from that given in [14], makes this idea precise. A deadlock for BW problem $\langle I, G\rangle$ over B is a nonempty subset of B that can be ordered $\langle d_1, \ldots, d_k\rangle$ in such a way that:

$$\begin{cases} \text{for all } i, 1 \leqslant i < k, & N_{\langle I,G\rangle}(d_i, d_{i+1}) \\ \text{and also} & N_{\langle I,G\rangle}(d_k, d_1), \end{cases}$$

where

$$N_{\langle I,G\rangle}(a, b) \equiv \text{POSITION}_I(a) \neq \text{POSITION}_G(a) \wedge \text{POSITION}_I(b) \neq \text{POSITION}_G(b) \wedge$$
$$\exists x \neq \text{TABLE} \ (\text{ABOVE}_I(b, x) \wedge \text{ABOVE}_G(a, x)).$$

Note that it is possible for a single block to constitute a deadlock. For instance, the problem in Fig. 1 is deadlocked, the deadlocks being {a} and {a, d}. To see this, note that $N_{\langle I,G\rangle}(\text{a, d})$ taking the third block in the definition to be $x = \text{e}$, that $N_{\langle I,G\rangle}(\text{d, a})$ taking $x = \text{c}$, and that $N_{\langle I,G\rangle}(\text{a, a})$ taking $x = \text{b}$.

---

[8] We follow [14] in using the expression *problem* for what some authors call a *problem instance*. This should not cause any confusion.

It is easy to see that if $N_{\langle I,G\rangle}(a,b)$ then in any plan for $\langle I,G\rangle$, the first time b is moved must precede the last time a is moved.[9] A deadlock being a loop of the $N_{\langle I,G\rangle}$ relation, at least one block in each deadlock must be moved twice. The first move of this block should be to the table, so as to break deadlocks without risking introducing new ones. This already sets constraints on the sort of plans we need to consider. They will consist of exactly one constructive move and at most one non-constructive move to the table per misplaced block. There must be enough of the latter type to break all the deadlocks. In other terms, the set of blocks moved this way must be a *hitting set* for the deadlocks (i.e., intersect all the deadlocks).[10] In order to produce an optimal plan, this hitting set must be of minimal size, and as it turns out, finding a minimal-size hitting set is notoriously NP-hard [10, p. 222]. This is the sub-problem which makes optimal BW planning difficult [14].

3.2. Strategies for near-optimal BW planning

On the other hand, merely finding some plan or other is easy. Unless we start moving blocks that are already in position or choose deliberately to introduce new deadlocks, we can even easily produce plans that are at most twice as long as the optimal. Various strategies for near-optimal BW planning within a factor of 2 have been described in the literature, for which we shall take the papers by Gupta and Nau as sources [13,14]. However, we find that these methods need to be clarified, and that their time complexity has not carefully been analysed. We shall therefore reformulate some of them and show that they can all be implemented to run in time linear in the number of blocks.

US. The first and simplest strategy we shall consider is one we have dubbed US (Unstack-Stack) [13, p. 630]. It amounts to putting all misplaced blocks on the table (the 'unstack' phase) and then building the goal state by constructive moves (the 'stack' phase). No block is moved by US unless it is misplaced, and no block is moved more than twice. Every misplaced block must be moved at least once even in an optimal plan. Hence the total number of moves in a US plan is at worst twice the optimal.

GN1. Another algorithm which is usually better in terms of plan length than US (and never worse) is given on pp. 229–230 of [14]. We call it GN1 for Gupta and Nau. It amounts to a loop, executed until broken by entering case (1):
  (1) If all blocks are in position, stop.
  (2) Else if a constructive move $\langle a,b\rangle$ exists, move a to b.
  (3) Else arbitrarily choose a misplaced clear block not yet on the table and move it to the table.

---

[9] The case in which $a=b$ is trivial. If $a \neq b$, then in the state immediately following its last move, a is clear and at the top of a tower containing x. If b has not yet moved, since it was above x initially, it is still above x and hence below a, so a must be moved again in order to clear b, contradicting the supposition that it had made its last move.

[10] For our problem example in Fig. 1, all blocks except g are misplaced and have to be moved constructively. In addition, the two deadlocks {a, d} and {a} have to be broken, which is achieved by moving a to the table.

Compared to US, GN1 never (non-constructively) moves a block to the table if it can be moved constructively. This avoids moving twice when once is enough.

GN2. Some of the remarks in [13,14] suggest yet another reading, though with no details of how it may be achieved. This one uses the concept of the deadlock. We call it GN2 and it is the same as GN1 except that the misplaced clear block to be moved to the table is chosen not completely arbitrarily but in such a way as to break at least one deadlock. That is, clause (3) reads:

(3) Else arbitrarily choose a clear block which is in a deadlock and move it to the table.

Gupta and Nau [14, p. 229, step 7 of their algorithm] say that in a deadlocked problem every misplaced clear block is in at least one deadlock. If this were true, GN1 and GN2 would be identical. It is false, however, as may be seen from the example in Fig. 1. Block f is not in any deadlock, and so can be chosen for a move to the table by GN1 but not by GN2. It is possible for GN1 to produce a shorter plan than GN2 (indeed to produce an optimal plan) in any given case, though on average GN2 performs better because it never completely wastes a non-constructive move by failing to break a deadlock.

In order for GN2 to be complete, in every deadlocked problem there must be at least one clear block which is in a deadlock. In fact, we can prove the stronger result that in every deadlocked problem there exists a deadlock consisting entirely of clear blocks. We now sketch the proof, since it makes use of the notion of a Δ sequence which will be needed in implementing GN2.

Let $\sigma = \langle B, S \rangle$ be a state that occurs during the attempt to solve a problem with goal state $G = \langle B, S_G \rangle$ and suppose the problem $\pi = \langle \sigma, G \rangle$ is deadlocked. Let b be misplaced and clear in σ. Consider $\text{POSITION}_G(b)$, the sequence of blocks which in G will lead from b down to the table. Let c be the first (highest) block in this sequence which is already in position in σ (c may be the table, or $S_G(b)$, or somewhere in between). In the goal state, either b or some block below b will be on c. Let us call this block d. What we need to do, in order to advance towards a constructive move with b, is to put d on c. This is not immediately possible because there is no constructive move in σ, so either c is not clear or else d is not clear. If c is not clear, we must move the blocks above it, starting with the one at the top of the tower which contains c in σ. If c is clear, we should next move the clear block above d in σ. This is how the function δ_π is defined: $\delta_\pi(b) = x$ such that

$$\text{CLEAR}_\sigma(x) \quad \text{and} \quad \begin{cases} \text{ABOVE}_\sigma(x, d) & \text{if } \text{CLEAR}_\sigma(c), \\ \text{ABOVE}_\sigma(x, c) & \text{if } \neg\text{CLEAR}_\sigma(c). \end{cases}$$

If b is in position or not clear or if $S_G(b)$ is in position and clear, let $\delta_\pi(b)$ be undefined. Now let $\Delta_\pi(b)$ be the sequence of blocks obtained from b by chasing the function δ_π:

$$\Delta_\pi(b) = \begin{cases} \langle b :: \Delta_\pi(\delta_\pi(b)) \rangle & \text{if } \delta_\pi(b) \text{ exists,} \\ \langle b \rangle & \text{otherwise.} \end{cases}$$

The point of the construction is that if $\delta_\pi(b)$ exists, then $\text{CLEAR}_\sigma(\delta_\pi(b))$ and $N_\pi(b, \delta_\pi(b))$. To see why the latter holds, note that either c or d is below $\delta_\pi(b)$ in σ and below b in the goal. Now for any misplaced clear b, if $\Delta_\pi(b)$ is finite then there is a constructive move in σ using the last block in $\Delta_\pi(b)$, while if it is infinite then it loops and the loop is a

deadlock consisting of clear blocks. This loop need not contain b of course: again Fig. 1 shows an example, where Δ_π(d) is $\langle$d, a, a, ...$\rangle$.

Our suggestion for a way of implementing GN2, therefore, is to replace the original clause (3) with:

(3) Else arbitrarily choose a misplaced clear block not on the table; compute its Δ sequence until this loops; detect the loop when for some x in Δ_π, $\delta_\pi(x)$ occurs earlier in Δ_π; move x to the table.

It is worth noting that our GN2 is not quite the same as the pure GN2 given above, because we always break a Δ sequence deadlock rather than an arbitrary deadlock. So there may be deadlocked blocks (e.g., d in Fig. 1) which cannot be chosen for a move to the table because they are not part of a loop in any Δ sequence. This appears to improve the average-case behaviour, but we have not investigated the matter in detail.

We now show how to implement all of US, GN1 and GN2 to run in time linear in the number n of blocks. This improves on the known complexity of these algorithms. The original [14] did not mention any bound better than O(n^3) for near-optimal BW planning though O(n^2) implementations have been described by other authors [1,6].

3.3. Linear-time algorithm for US

The key to making US a linear-time algorithm is to find a way to compute which blocks are in position in O(n), and to execute this computation only once in the course of the problem solution. We do this by means of a combination of recursion and iteration,

```
procedure US ()                          function INPOS (b : block) : boolean
   INIT()                                   if b = TABLE then
   for each b ∈ B\{TABLE} do                  return true
      if Clear_b then                         if not Examined_b then
         UNSTACK(b)                              Examined_b ← true
   for each b ∈ B\{TABLE} do                     if Si_b ≠ Sg_b then
      STACK(b)                                      InPosition_b ← false
                                                 else InPosition_b ← INPOS(Si_b)
procedure INIT ()                            return InPosition_b
   Plan ← ⟨ ⟩
   for each b ∈ B\{TABLE} do              procedure MOVE (⟨a, b⟩ : move)
      Clear_b ← true                         Plan ← ⟨⟨a, b⟩ :: Plan⟩
      Examined_b ← false                     if Si_a ≠ TABLE then
   for each b ∈ B\{TABLE} do                    Clear_{Si_a} ← true
      INPOS(b)                                if b ≠ TABLE then
      if Si_b ≠ TABLE then                       Clear_b ← false
         Clear_{Si_b} ← false                     InPosition_a ← (Sg_a = b) and InPosition_b
                                                else InPosition_a ← (Sg_a = TABLE)
procedure UNSTACK (b : block)                Si_a ← b
   if (not InPosition_b) and
      (Si_b ≠ TABLE) then                 procedure STACK (b : block)
      (local) c ← Si_b                       if not InPosition_b then
      MOVE(⟨b, TABLE⟩)                          STACK(Sg_b)
      UNSTACK(c)                               MOVE(⟨b, Sg_b⟩)
```

Fig. 3. The US algorithm.

as shown in Fig. 3. The algorithm makes use of several variables associated with each block b:

Clear_b	True iff b is clear in the current state.
$\mathsf{InPosition}_b$	True iff b is already in position.
$\mathsf{Examined}_b$	True iff $\mathsf{InPosition}_b$ has been determined.
Si_b	Block currently below b.
Sg_b	Block below b in the goal.

During initialization, INPOS can be called at most twice with any particular block b as parameter: once as part of the iteration through the blocks and at most once recursively from a call with the block above b. Hence the number of calls to INPOS is bounded above by $2n$. Similar considerations apply to the recursive STACK and UNSTACK procedures. The stored information is updated in constant time by MOVE.

3.4. Linear-time algorithm for GN1

For GN1, we need to recognise constructive moves from tower to tower as well as those from the table. To achieve this, we add further structure to the problem representation. At any given time, each block has a status. It may be:

(1) ready to move constructively. That is, it is misplaced but clear and its target is in position and clear.

(2) stuck on a tower. That is, it is misplaced, clear and not on the table, but unable to move constructively because its target is either misplaced or not clear.

(3) neither of the above.

More variables are now associated with each block b: Status_b records the status (READY, STUCK, or OTHER) of this block, while Pi_b and Pg_b denote the blocks (if any) which are on this one currently and in the goal. Evidently, initialising, setting and updating these will not upset the $O(n)$ running time. To make it possible to select moves in constant time, the blocks of status READY and STUCK are organised into doubly linked lists, ReadyList and StuckList, one such list containing the blocks of each status. Inserting a block in a list and deleting it from a list are constant-time operations as is familiar. The next block to move is that at the head of ReadyList unless that list is empty, in which case it is the block at the head of StuckList. If both lists are empty, the goal is reached.

When a block a moves, it changes its status as well as its position. Certain other blocks may also change status as a result of the move. When they exist, these are: the block currently below a (Si_a) if any, the block that will be on a in the goal (Pg_a) if any, and the block which in the goal will be on the block currently below a ($\mathsf{Pg}_{\mathsf{Si}_a}$) if any. Those, however, are all of the possible changes: a constant number (4) for each move. Therefore the number of delete-insert list operations on blocks is at worst linear in the number of moves in the plan, Nothing else stands in the way of the linear-time implementation of GN1 shown in Fig. 4. [11]

---

[11] The function INPOS is the same as in the US implementation and is therefore not repeated in Fig. 4.

```
procedure GN1 ()
    INIT()
    while not (EMPTY(ReadyList) and
                EMPTY(StuckList)) do
      if not EMPTY(ReadyList) then
        (local) b ← HEAD(ReadyList)
        MOVE(⟨b, Sg_b⟩)
      else (local) b ← HEAD(StuckList)
           MOVE(⟨b, TABLE⟩)

procedure INIT ()
    Plan ← ⟨ ⟩
    EMPTYLIST(ReadyList)
    EMPTYLIST(StuckList)
    for each b ∈ B\{TABLE} do
      Clear_b ← true
      Examined_b ← false
      Status_b ← OTHER
      Pi_b ← SKY
      Pg_b ← SKY
    for each b ∈ B\{TABLE} do
      INPOS(b)
      if Si_b ≠ TABLE then
        Clear_{Si_b} ← false
        Pi_{Si_b} ← b
      if Sg_b ≠ TABLE then
        Pg_{Sg_b} ← b
    for each b ∈ B\{TABLE} do
      STATUS(b)

procedure STATUS (b : block)
    if (not InPosition_b) and Clear_b then
      if Sg_b = TABLE then
        STAT(b,READY)
      else if InPosition_{Sg_b} and Clear_{Sg_b} then
           STAT(b,READY)
           else if Si_b = TABLE then
                STAT(b,OTHER)
           else STAT(b,STUCK)
    else STAT(b,OTHER)
```

```
procedure STAT (b : block, x : status)
    case Status_b of
      READY : DELETE(b,ReadyList)
      STUCK : DELETE(b,StuckList)
      OTHER :
    case x of
      READY : INSERT(b,ReadyList)
      STUCK : INSERT(b,StuckList)
      OTHER :
    Status_b ← x


procedure MOVE (⟨a, b⟩ : move)
    Plan ← ⟨⟨a, b⟩ :: Plan⟩
    (local) c1 ← Si_a
    (local) c2 ← Pg_a
    if Si_a ≠ TABLE then
      (local) c3 ← Pg_{Si_a}
    else (local) c3 ← TABLE
    if Si_a ≠ TABLE then
      Clear_{Si_a} ← true
      Pi_{Si_a} ← SKY
    if b ≠ TABLE then
      Clear_b ← false
      Pi_b ← a
      InPosition_a ← (Sg_a = b) and InPosition_b
    else InPosition_a ← (Sg_a = TABLE)
    Si_a ← b
    STATUS(a)
    if c1 ≠ TABLE then
      STATUS(c1)
    if c2 ≠ SKY then
      STATUS(c2)
    if c3 ≠ TABLE then
      STATUS(c3)
```

Fig. 4. The GN1 algorithm.

3.5. Linear-time algorithm for GN2

GN2, however, is a different matter. To implement GN2 via Δ sequences, it is necessary to compute $\delta_\pi(b)$ for various blocks b, and to achieve linear time there must be both a way to do this in constant time and a way to limit the number of δ calculations to a constant number per block. On the face of it, neither of these is easy. To find $\delta_\pi(b)$ it is necessary to know which is the highest block in position in the goal tower of b and to know which is the clear block above a given one. These items of information change as moves are made, and each time such an item changes for one block it changes for all the $O(n)$ blocks in a tower, so how can those changes be propagated in constant time? Moreover, when a deadlock is
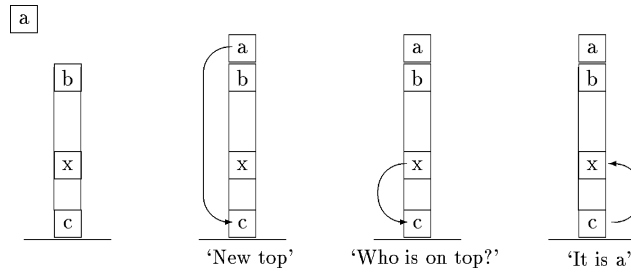
Fig. 5. Asking the concierge 'Who lives at the top of the tower?'

to be broken, a new Δ sequence has to be computed, as many blocks may have moved since the last one was computed, thus changing δ. Computing a Δ sequence appears to be irreducibly an O(n) problem, and since O(n) such sequences may be needed, this appears to require GN2 to be of O(n^2) even if $\delta_\pi(b)$ can somehow be found in constant time.

The first trick that begins to address these difficulties is to note that whatever changes in a tower of blocks, one thing does not change: the block on the table at the bottom of the tower. If that block moves, the tower no longer exists. We call the bottom block in a tower the *concierge* for that tower. Now if we want to know who lives at the top of the tower, we can ask the concierge. When a block comes or goes at the top of the tower, only the concierge need be informed (in constant time) and then since every block knows which is its concierge, there is a constant-time route from any given block to the information as to which block above it is clear (see Fig. 5). Not only the towers in the initial (or current) state have concierges, but so do the towers in the goal state. These keep track of which block in their tower is the highest already in position. Additional variables associated with each block b denote its initial and goal concierges, Ci_b and Cg_b. In case b is a concierge, there are more variables denoting the clear block Top_b above it, and the highest block Hin_b already in position in its goal tower. Through the concierges, there is a a constant-time route from b to the c and d required to define $\delta_\pi(b)$. The procedure for initialising the additional variables is closely analogous to that for determining which blocks are in position and can be executed in linear time for the same reason. Updating them when a move is made takes constant time.

Next, the key to managing the Δ sequences is that although δ may change the N_π relation is indestructible except by moving the blocks involved. That is, if $N_{\langle\sigma,G\rangle}(x,y)$
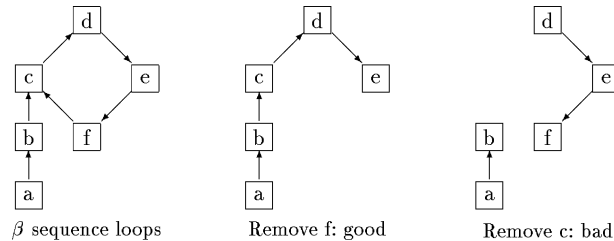

Fig. 6. How [not] to break a deadlock.

```
procedure GN2 ()                          function BETA () : block
    INIT()                                    (local) f ← false
    while not (EMPTY(ReadyList) and          while not (EMPTY(Beta) or f) do
            EMPTY(StuckList)) do                 (local) b ← TOP(Beta)
        if not EMPTY(ReadyList) then             if Status_b ≠ STUCK then
            (local) b ← HEAD(ReadyList)              POP(Beta)
            MOVE(⟨b, Sg_b⟩)                      else f ← true
        else (local) b ← BETA()              if EMPTY(Beta) then
            MOVE(⟨b, TABLE⟩)                     (local) b ← HEAD(StuckList)
                                                 Examined_b ← true
procedure INIT ()                            (local) f ← false
    Plan ← ⟨ ⟩                               while not f do
    EMPTYLIST(ReadyList)                         (local) b ← DELTA(TOP(Beta))
    EMPTYLIST(StuckList)                         if Examined_b then
    EMPTYSTACK(Beta)                                (local) f ← true
    for each b ∈ B\{TABLE} do                    else Examined_b ← true
        Clear_b ← true                               PUSH(b,Beta)
        Examined_b ← false                      return TOP(Beta)
        Status_b ← OTHER
        Pi_b ← SKY                           function DELTA (b : block) : block
        Pg_b ← SKY                              if Status_b ≠ stuck then
        Hin_b ← TABLE                               return undefined
    for each b ∈ B\{TABLE} do                    (local) c ← Hin_{Cg_b}
        INPOS(b)                                 if c ≠ TABLE then
        if Si_b ≠ TABLE then                         (local) d ← Pg_c
            Clear_{Si_b} ← false                 else (local) d ← Cg_b
            Pi_{Si_b} ← b                       if c = TABLE then
        if Sg_b ≠ TABLE then                         return Top_{Ci_d}
            Pg_{Sg_b} ← b                       if Clear_c then
    for each b ∈ B\{TABLE} do                        return Top_{Ci_d}
        STATUS(b)                                return Top_{Ci_c}
    for each b ∈ B\{TABLE} do
        Examined_b ← false                   procedure MOVE (⟨a, b⟩ : move)
    for each b ∈ B\{TABLE} do                    Plan ← ⟨⟨a, b⟩ :: Plan⟩
        CONCIERGE(b)                             (local) c1 ← Si_a
    for each b ∈ B\{TABLE} do                    (local) c2 ← Pg_a
        Examined_b ← false                       if Si_a ≠ TABLE then
    for each b ∈ B\{TABLE} do                        (local) c3 ← Pg_{Si_a}
        GCONCIERGE(b)                            else (local) c3 ← TABLE
    for each b ∈ B\{TABLE} do                    if Si_a ≠ TABLE then
        Examined_b ← false                           Clear_{Si_a} ← true
                                                     Pi_{Si_a} ← SKY
function CONCIERGE (b : block) : block               Top_{Ci_a} ← Si_a
    if not Examined_b then                       if b ≠ TABLE then
        Examined_b ← true                            Clear_b ← false
        if Si_b ≠ TABLE then                         Pi_b ← a
            Ci_b ← CONCIERGE(Si_b)                   Ci_a ← Ci_b
        else Ci_b ← b                                InPosition_a ← (Sg_a = b) and InPosition_b
        Top_{Ci_b} ← b                           else Ci_a ← a
    return Ci_b                                       InPosition_a ← (Sg_a = TABLE)
                                                 Top_{Ci_a} ← a
                                                 if InPosition_a then
function GCONCIERGE (b : block) : block              Hin_{Cg_a} ← a
    if not Examined_b then                       Si_a ← b
        Examined_b ← true                        STATUS(a)
        if Sg_b ≠ TABLE then                     if c1 ≠ TABLE then
            Cg_b ← CONCIERGE(Sg_b)                   STATUS(c1)
        else Cg_b ← b                            if c2 ≠ SKY then
        if InPosition_b then                         STATUS(c2)
            Hin_{Cg_b} ← b                       if c3 ≠ TABLE then
    return Cg_b                                      STATUS(c3)
```

Fig. 7. The GN2 algorithm.

then that relationship persists in the sense that in all future states θ, $N_{\langle \theta, G \rangle}(x, y)$ unless x or y has moved in getting from σ to θ. Moreover, as noted above, if $N_\pi(x, y)$ then x cannot move constructively until y has moved at least once. Now, let $\beta = \langle b_1, \ldots, b_k \rangle$ be a non-looping sequence of clear blocks which are stuck on towers, each except the last linked to its successor by the relation N_π. At some point, b_k may cease to be stuck and become ready to move, but no other block in β can change its status until b_k actually moves. Thus the β sequence may dwindle, and even become null, as moves are made, but it always remains a single sequence—it never falls into two pieces as would happen if a block from the middle of it changed status—and because N_π is indestructible β remains linked by N_π. For the algorithm, then, we maintain such a sequence β, constructed from parts of Δ sequences as follows. Initially, β is null. If the problem becomes deadlocked, first if β is null then it is set to consist just of the block at the head of the StuckList, and then it is extended by adding $\delta_\pi(b_k)$ to the end of it. This is done repeatedly until the sequence threatens to loop because δ_π of the last block b_m is already in β. At that point b_m is chosen to break the deadlock. It is important not to choose $\delta_\pi(b_m)$ for this purpose, since that could result in breaking β into two pieces (see Fig. 6). Each addition to β takes only constant time, and any given block can be added to the sequence at most once. Therefore maintenance of the β sequence requires only linear time. This completes the description of the principles underlying the algorithm, which is shown in Fig. 7. [12]

3.6. Algorithm for optimal Blocks-World planning

Some of the experiments in Section 4 below require the ability to generate optimal BW plans. For instance, in order to gather experimental results concerning the average performance ratios of the various near-optimal algorithms, we need to know the optimal plan lengths. A good BW-specific optimal planning method is also needed to determine how hard optimal BW planning really is, and get an idea of the size of (average) BW problems that should be regarded as truly challenging. Moreover, it is a prerequisite to the study of the distribution of hard and easy instances among problems of a given size, to the identification of a pattern in this distribution, and to the investigation of the relationship between hard problems and average ones.

Unfortunately, the literature seems to lack any algorithm suitable for these purposes. The non-deterministic algorithm given in [14] is useful for complexity analyses but not as a practical method, while most papers reporting experiments on BW use a domain-independent planner for generating optimal plans and drastically limit the size of the problems considered. We shall therefore describe the method we used.

Recall from Section 3.1 above that the key to optimal BW planning is finding a hitting set of minimal cardinality for the set of deadlocks in the problem. At the heart of our algorithm, therefore, is a backtracking search for such a hitting set. One complication is that we do not know at the outset which sets of blocks constitute deadlocks; nor do we know of an efficient method of enumerating the deadlocks; nor in fact do we ever know that we have hit them all until we can finally produce a plan.

---

[12] The functions INPOS, STATUS and STAT are the same as in the implementation of GN1, and are not repeated in Fig. 7. The β sequence is implemented as a stack.

Our algorithm PERFECT overcomes this problem as follows. It constructs a set K of known deadlocks. Initially, only the singleton deadlocks are known; other deadlocks are computed as needed. At each iteration it finds a minimal size hitting set H for the deadlocks currently in K and tests this to discover whether it hits all deadlocks in the problem. If it does, it constitutes a solution and the algorithm halts. If not, a deadlock disjoint from H is found and added to K:

> */* initialisation */*
> $K \leftarrow \{\{b\} : \{b\}$ is a deadlock$\}$
> */* main loop */*
>   Repeat until plan returned:
>   Generate H a minimal size hitting set for K
>   TEST(H)
>   if H solves the problem then
>   return PLAN(H)
>   else find a deadlock D such that $D \cap H = \emptyset$
> $K \leftarrow K \cup \{D\}$

This requires five procedures to solve sub-problems:
  (1) The set of singleton deadlocks must be found to initialise K.
  (2) A minimal size hitting set H for K must be generated.
  (3) H must be tested to see whether it hits all deadlocks in the problem.
  (4) A deadlock D disjoint from H must be found.
  (5) H must be used to produce a plan.
Of these, (2) is the NP-complete sub-problem which will not be detailed further here except to note that it is solved by a backtracking search, requiring exponential time in the worst case. It may be speeded up somewhat by incorporating various search heuristics which, however, are not especially germane to the present paper. The other four sub-problems are all tractable and the corresponding procedures all make use of our linear-time implementation of the near-optimal BW planner GN1.

It is easy to modify GN1 by giving it a set H of blocks as a parameter and requiring (in clause (3) of the GN1 algorithm) that only blocks in H be used to break deadlocks. This results in a plan if H is a hitting set for the deadlocks in the problem, and in failure to find a plan otherwise. That is, it yields a (linear-time) decision procedure for whether H is a hitting set or not. We call GN1 thus modified GN1H. Evidently, GN1H suffices for both (3) and (5) above.

GN1H may also be used to achieve (1). It is clear that GN1H($B \setminus \{b\}$) results in a plan if and only if $\{b\}$ is not a (singleton) deadlock. Therefore, by running GN1H($B \setminus \{b\}$) for each block $b \in B$ in turn, we obtain in quadratic time the set of all singleton deadlocks. [13]

---

[13] This is not the most efficient way of detecting singleton deadlocks—the experiments on singleton deadlocks reported in Appendix A below use a better one—but for small problems such as those for which optimal planning is feasible, it is adequate.

A similar technique suffices for (4). Suppose that H is not a hitting set for all deadlocks in the problem. Then proceed as follows:

```
for each b ∈ B \ TABLE do
    TEST(H ∪ {b})
    if H ∪ {b} is not a hitting set then
        H ← H ∪ {b}
```

Obviously, at the end of this (quadratic-time) procedure, H is a non-hitting set and is maximal in the sense that every proper superset of it is a hitting set. Let D be its complement with respect to $B \setminus \{\text{TABLE}\}$. D is a deadlock: it contains a deadlock, since otherwise H would be a hitting set, and it contains nothing else, since otherwise H would not be maximal.

Our linear-time implementation of GN1 therefore serves both as a decision procedure for potential hitting sets and as a plan generator once a hitting set has been found. It enables our optimal procedure to cope with well over 100 blocks in reasonable time, as may be seen in Fig. 9 below.

4. Experimental observations

Using our random BW problem generator and the above algorithms, we are able to make a number of interesting observations on the structure of large BW problems and on the performance in time and average solution quality of BW-specific planning methods.

Unsurprisingly, our near-optimal planners are fast. They all solve problems of a million blocks in a matter of seconds. More interestingly, we find that the plans they produce are closer to the optimal than we had expected. On the basis of the experimental results, it may be conjectured that their average performance ratios tend in the limit to 1, in which case, despite the NP-hardness of optimal planning, it is possible in linear time to find plans in which, on average, the proportion of unnecessary moves is vanishingly small.

By close examination of random BW problems, we identify the structural features which planning systems should use if they are to do well with the domain. Proving the optimality of plans, of course, is more complicated. Our experiments show a clear easy-hard-easy distribution of optimisation problems, using the numbers of towers in the initial and goal states as the order parameter. This is related to a tradeoff between the number of deadlocks and their size.

In the interest of readability, details of the data used for these experiments have been relegated to Appendix A.

4.1. Time performances

We start with the average runtimes of the near-optimal algorithms as a function of the number n of blocks. As will be clear from the experiments in Section 4.2 below, there is no significant difference between the average and worst cases runtimes for these algorithms. As shown in Fig. 8, the time for all three programs is linear in the number of blocks. There is very little difference between GN1 and US. Since some aspects of the computation are closely related to the plan length, GN1 sometimes even outperforms US because it produces
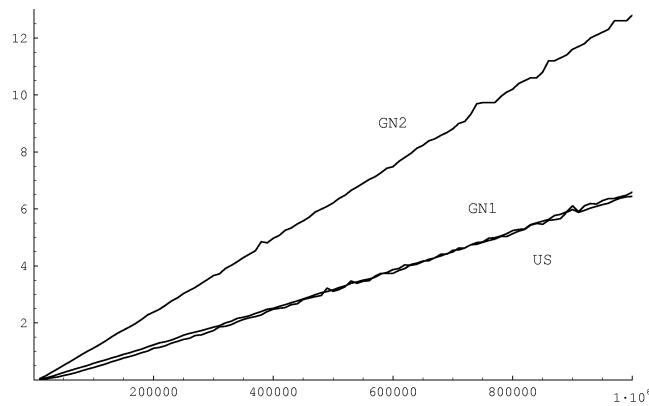
Fig. 8. Average runtimes (in secs) as a function of n for the near-optimal algorithms.
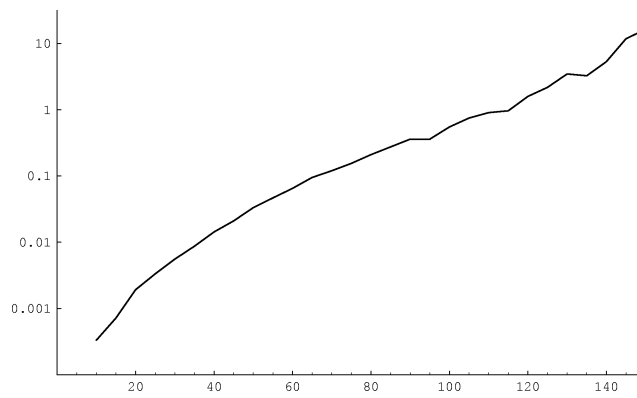

Fig. 9. Median runtime (in secs) as a function of n for the optimal algorithm.

shorter plans. Naturally, since nobody really wants to convert one BW state into another, the program speeds are not important in themselves. The point of this experiment was just to confirm empirically the theoretical claims that linear execution time may be attained, and that domain-independent planners are orders of magnitude away from BW-specific methods. [14]

In Fig. 9, we also measured the runtime of our optimal algorithm. As will become apparent from the experiments in Subsection 4.3.2, the average case is again similar to the worst case, namely here exponential in n. The main purpose of the experiment was to decide what number of blocks we could expect to use for other experiments if these were to be completed within the time available without sacrificing the information which can be gleaned from relatively large examples. We find that it is reasonable to go up to $n = 150$, at

---

[14] To the best of our knowledge, the current state of the art is represented by TALPLANNER [7,19], a variant of the TLPLAN system [2] which solves BW problems near-optimally in quadratic time. 500 blocks problems take about a second.
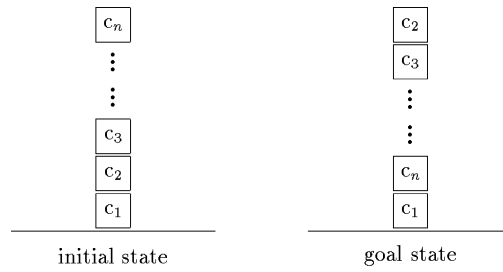
Fig. 10. Problem leading to the worst optimal plan length.
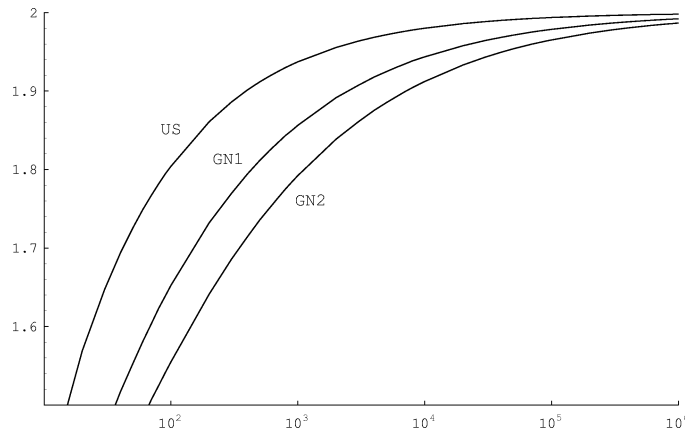
Fig. 11. Average plan length as a function of n: (number of moves)$/n$.

which size the average problem is several orders of magnitude harder than those that have been considered in the domain-independent setting. [15]

4.2. Plan length and solution quality

After the performance in time of the algorithms, we look at their performance in solution quality. The length of the plan produced by all near-optimal algorithms, as well as the optimal plan length is $2n - 2$ in the *worst* case. As can be seen from Fig. 10, it is possible for every block but one on the table to constitute a singleton deadlock and to have to move twice. The *average* length of the plan produced by the near-optimal algorithms approaches this worst case quite closely, as may be observed from the graph in Fig. 11. As expected, US gives the longest plans on average and GN2 the shortest, but for large numbers of blocks it makes little difference which algorithm is used. In particular, it is evident from the graphs that the algorithms will give very close average plan lengths in the limit.

The immediate questions raised by the present results are whether all of the algorithms converge to the same average plan length (as a proportion of n) and if so whether

---

[15] Neither TLPLAN [2] nor SATPLAN [18] copes with random optimisation problems of more than 20 blocks.

this limiting figure is $2n$. Positive answers to both questions are strongly suggested by the data—indeed, they seem obvious—but obviousness is not proof. The theoretical investigation does not appear easy: even to prove that the average length of the plans produced by the 'baby' algorithm converges to $2(n - \sqrt{n})$ we needed nontrivial mathematics involving complex analysis and the theory of Laguerre polynomials [29, pp. 23–26]. This algorithm, which is not near-optimal, simply puts *all* blocks (misplaced or not) on the table before building the goal position.

The *absolute* performance ratio of the near-optimal algorithms is 2 in the limit. The worst-case problem is shown in Fig. 12. The deadlocks are the pairs $\{a, c_i\}$ for $1 \leqslant i \leqslant k$, so the optimal plan, of length $k + 2$, consists in breaking all the deadlocks by moving a to the table and then moving the c_i and a constructively. US will first move all of the c_i (and a) to the table. In the worst case, GN1 and GN2 may also move all of the c_i to the table before moving a constructively, giving a plan of length $2k + 1$ and therefore a performance ratio of $(2k + 1)/(k + 2)$ which tends quickly to 2.

Fig. 13 shows the *average* performance ratios. This graph contains a real surprise: the average performance of US does not degrade monotonically but turns around at about $n = 50$ and begins to improve thereafter. The explanation is the plan lengths for US quickly approach the ceiling of $2n$, at which point the optimal plan lengths are increasing more
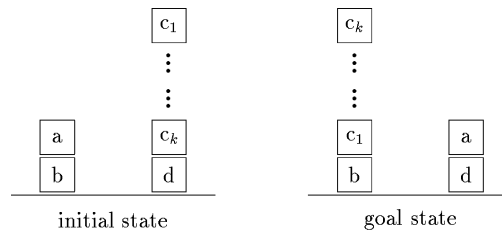

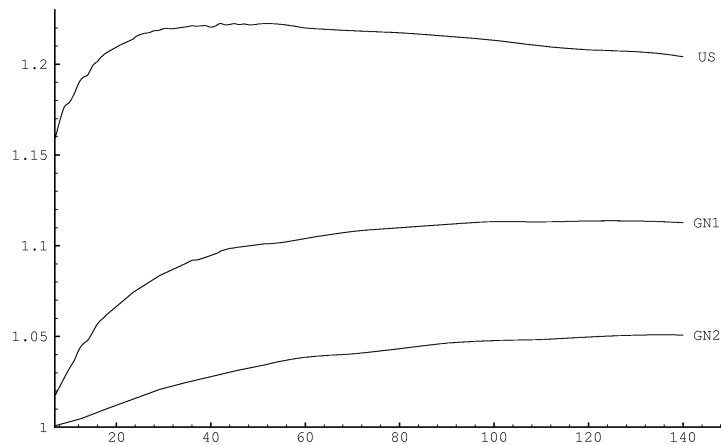Fig. 12. Problem leading to performance ratios of 2.

Fig. 13. Average performance ratio as a function of n: (plan length)/(optimal plan length).

quickly than the US ones because the latter are nearly as bad as they can get. It is clear from this figure and Fig. 11 that the other near-optimal algorithms would exhibit similar curves if it were possible to observe the length of optimal plans for high enough values of n.

One result readily available from the graphs is an upper bound around 1.23 for average performance ratios. However, Figs. 11 and 13 together suggest that the limiting value will be well below this rough upper bound. Our open question following this experiment is whether the optimal plans tend to a length of $2n$ in the limit. If they do, then not only the near-optimal algorithms but even the 'baby' algorithm have the perfect average performance ratio of 1 in the limit. A positive answer would be implied if the number of singleton deadlocks tended to n, but on investigating this figure experimentally we found that it appears to be only around $0.4n$ (see experiments in Section 4.3.1).

4.3. The structure of Blocks World planning problems

According to the above experiments, the gap in time performance between domain-specific and domain-independent methods is huge, in both the optimal and near-optimal cases. Moreover, in the latter, domain-specific methods achieve higher solution quality. Why is this? What structural properties of the domain are the specific solvers using which the generic planners are missing?

4.3.1. Composition of the average BW problem

As a preliminary step, we examine the composition of the average BW problem. Some of the blocks are misplaced and have to move, and some do not. Then, of the misplaced blocks, some occur in deadlocks, and some do not. Finally, of the deadlocked blocks, some constitute singleton deadlocks and so must obviously move twice, while others do not. We call the blocks in the last category *live* blocks. The non-trivial part of a BW planning problem is to decide which of the live blocks should move twice. This is demonstrated in Fig. 14, which represents the average hardness of optimal BW planning problems of $n = 100$ blocks as a function of the number of live blocks in these problems. The positive correlation is obvious.

Quantitatively, what is the proportion of blocks in each category? The first observation is that almost all of the n blocks are misplaced. Fig. 15 shows the distribution of problems with a given number of misplaced blocks. This has been obtained with random problems of $n = 1000$ blocks, but we find that the distribution does not vary significantly with n and similar results are obtained with as few as 20 blocks. In particular the probability that all blocks are misplaced seems to converge quickly to e^{-1}.[16] Instances with more than about 5 blocks in position are extremely rare, even for problems with 10000 blocks. Except in pathological cases, this already imposes a lower bound of roughly n on the plan length.

Clearly any block which starts or finishes on the table cannot be involved in any deadlock. As already noted, $\sqrt{n}$ blocks are on the table in the average state, so there should be at least $2\sqrt{n}$ deadlock-free blocks on average. This gives us an upper bound

---

[16] An analytical expression for this probability as a function of n is given in [28].
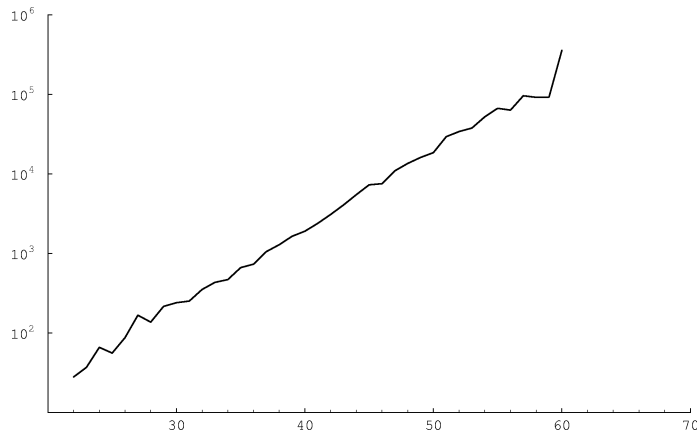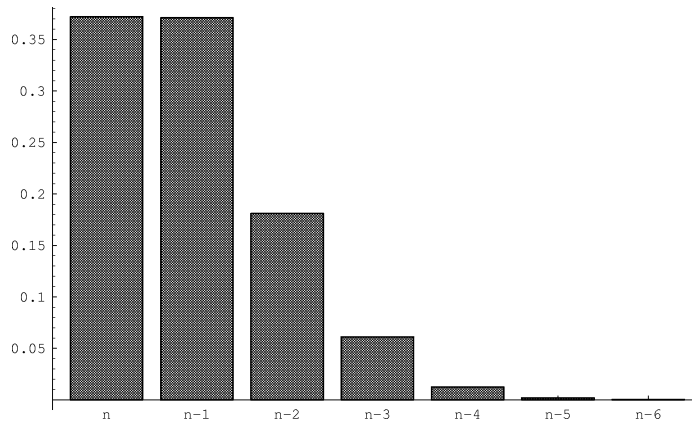
Fig. 14. Median hardness as a function of the number of live blocks.

Fig. 15. Probability of exactly m blocks being misplaced for $m = n, \dots, n - 6$.

around $2(n - \sqrt{n})$ on the average plan length. Experimentally (see Fig. 16) we find that the blocks on the table are almost the only deadlock-free ones: on average less than 2 others are deadlock-free. It follows that the upper bound cannot be significantly improved by reasoning about which blocks are in deadlocks, and that there is no need for a planner to spend a lot of time in such reasoning.

On the other hand, reasoning about special kinds of deadlocks may be valuable (for instance GN2 gains by concentrating on Δ sequence deadlocks). Most importantly, we observe in Fig. 17 that on average nearly 40% of the blocks are singleton deadlocks. This limiting figure is approached quite closely for problems around $n = 100$ blocks. In sum, the optimal plan length is almost always between $1.4n$ and $2(n - \sqrt{n})$, and so on average less than 30% of the moves are non-trivial.
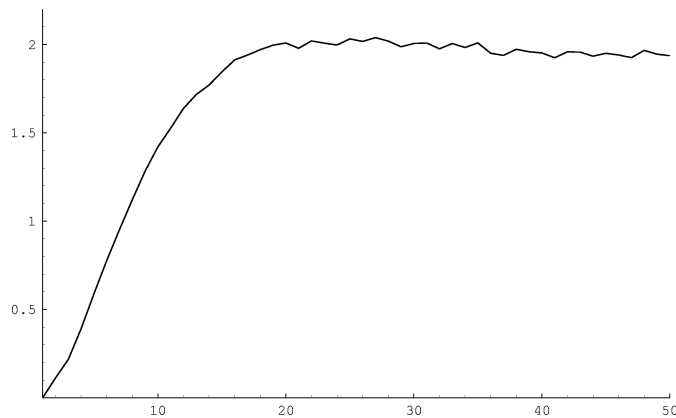
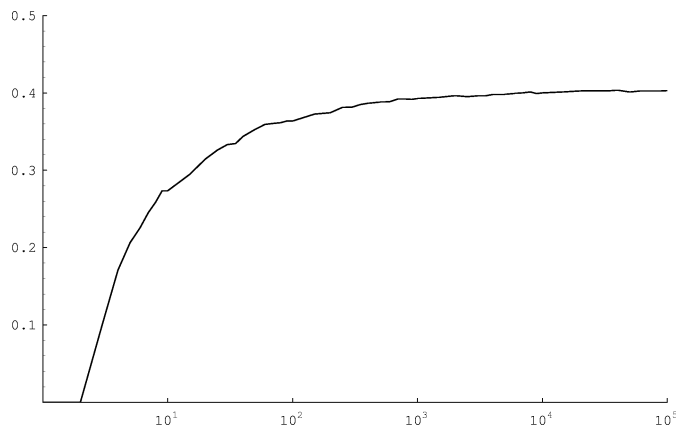Fig. 16. Average number of deadlock-free blocks not on the table as a function of n.

Fig. 17. Proportion of blocks constituting singleton deadlocks as a function of n.

4.3.2. *Hard and easy* BW *problems*

The next question is under what conditions is it difficult to plan these non-trivial moves in an optimal way—that is, when is it hard to find a minimal size hitting set for the deadlocks? Well, it is easy if there are few deadlocks, and it is easy if all the deadlocks are small. Unfortunately, these two conditions tend to work against each other. For few deadlocks to exist, the N relation should be sparse, which typically occurs where there are many short towers. Small deadlocks occur where the N relation loops very easily, which happens when most of the blocks are found in a very few tall towers. The hard problems are therefore squeezed between the "under-constrained" area in which there are many towers and the "over-constrained" area in which there are few.

This analysis is confirmed by Fig. 18, which represents the median hardness (as the number of times PERFECT backtracks) of random problems of 100 blocks against the parameters of the number of towers initially and in the goal. It can be seen that optimal
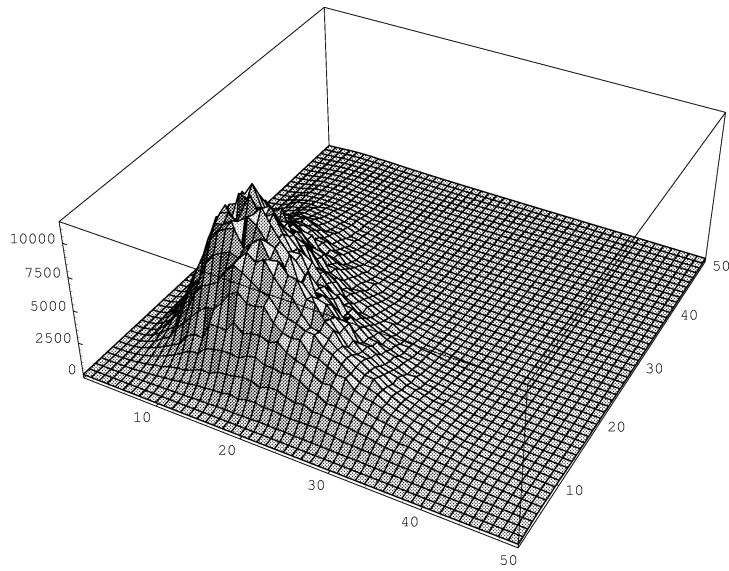
Fig. 18. Median hardness by number of towers in the initial and goal states.

planning is easy for a wide *range* of these parameters: for problems of this size the hard region is confined between 8 and 18 towers. However, it is important to note that most random problems fall in the hard region: random problems of this size cluster strongly around $\sqrt{100} = 10$ towers. So on average, BW problems are reasonably hard. Random problems are however not the hardest. For instance, problems of this size produced by the UCPOP generator cluster around $\sqrt{200} = 14$ towers and are harder.

4.3.3. What are the relevant features?

In the light of the above, we may begin to answer the questions at the head of this section. At least we may observe:

(1) As noted for example in [1], it is crucial that planners classify blocks as in position or misplaced rather than attempting to carry out long chains of reasoning purely in terms of ON and CLEAR. Being able to recognise constructive moves and make them whenever the opportunity exists already suffices for more than half of the average plan.

(2) More than half of the remainder are moves which break singleton deadlocks. Being able to recognise and make these moves is therefore sufficient to allow a planner to come close to completing plans without any search or backtracking.

(3) The fact that all non-constructive moves may be to the table is easily coded (as noted, e.g., by Kautz and Selman in [18]) and greatly reduces the set of possible plans and therefore the search. In Kautz and Selman's terminology, it is a *safe simplifying assumption* based on a completeness theorem which could in principle be found automatically.

Once the constructive moves have been identified, and the blocks in singleton deadlocks removed from consideration, there remains only the problem of finding a hitting set for the remaining deadlocks. For near-optimal BW planning, no more reasoning is required: any choice whatever, even the worst, yields a plan very close to the optimal on average. For optimal BW planning, it is necessary to generate a minimal size hitting set, but *this* problem is not specific to the field of planning and is best approached using techniques available off the shelf. In either case, therefore, the above features constitute the structure which we expect domain-independent planners to exploit if they are to deal adequately with the domain.

4.3.4. Existing benchmark instances

Another question that can be addressed following the above observations is that of the representativity of existing benchmark instances, such as the 'bw_large.a' to 'bw_large.d' commonly used in discussing the effectiveness of SAT-based planners (see, e.g., [8,11,12, 16–18]).

As can be seen from Fig. 19, the first obvious feature of these four problems is that the larger ones are built up from the smallest by progressively adding blocks. In bw_large.a there are no deadlocks at all, so even GN1 solves it optimally. In the others, there are two deadlocks (the same two in every case) and it is obvious that putting block 11 on the table is sufficient to break them both, after which again constructive moves suffice. Hence the reasoning in all three cases is essentially the same and is hardly challenging.
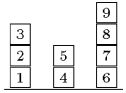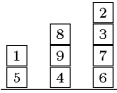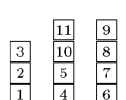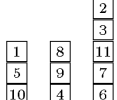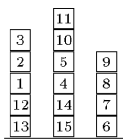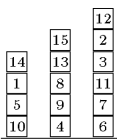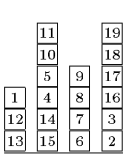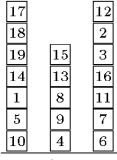
problem		blocks	misplaced	deadlocks	moves
initial state goal state		9	6	none	6
initial state goal state		11	8	$\{8, 11\}$ $\{9, 11\}$	9
initial state goal state		15	13	$\{8, 11\}$ $\{9, 11\}$	14
initial state goal state		19	17	$\{8, 11\}$ $\{9, 11\}$	18

Fig. 19. Problems bw_large.a . . . bw_large.d from [17].

Problem	Blocks in position	Number of towers	Singleton deadlocks	Optimal planlength	Time to solve optimally (sec)
100.0	1	3 + 5	79	182	0.04
100.1	1	2 + 5	79	182	0.05
200.0	1	10 + 9	129	348	2.92
200.1	2	9 + 6	135	346	0.59
300.0	1	5 + 2	278	577	0.21
300.1	2	5 + 6	264	567	0.47
400.0	1	4 + 4	368	773	0.94
400.1	2	8 + 5	325	740	5.21
500.0	1	5 + 6	462	963	0.83
500.1	1	4 + 5	461	969	1.84

Fig. 20. Problems probblocks-100 . . . probblocks-500 from AIPS-2000.

A less obvious but important feature is that they are quite unlike the average problem of their size in that they have many non-deadlocked blocks and no singleton deadlocks at all. This gives them all unusually short plans: the average plan length for 19-block problems is 24.5, yet bw_large.d requires only 18 moves. Comparisons based on these problems tend to favour planners which prefer short plans, SAT-based planners for instance. This feature is therefore highly relevant to their use as a benchmark.

Another benchmark set calling for comment is that from the planning competition of AIPS-2000, and in particular the "additional" problems from the "hand tailored systems track" (see Fig. 20). These have from 100 to 500 blocks. For their size, they all have very small numbers of towers in both initial and goal states, leading to abnormally large proportions of self-deadlocked blocks (over 90% in some cases). As a consequence, their optimal plans are long but extremely easy to find. PERFECT, which finds random problems of 160 blocks challenging, solves them almost instantly, as shown in the last column of Fig. 20. Problem 300.0 is even solved without backtracking. In the AIPS competition, optimality was not required, so the extreme skew in the sample does not invalidate the results. However, it should be noted that because they are so easy to solve optimally and because their optimal plans are so long, these problems reveal little or nothing about the quality of solutions produced by competing suboptimal planners.

5. Conclusion: What future for Blocks World?

BW has traditionally been the "Hello World" of planning: often, a trivial instance of it is used to illustrate what an action is, how some notation represents postconditions, how execution failures can be handled or whatever. To this practice we have no objection, save that the example becomes boring eventually. However, as we have been at pains to point out, the use of it as an benchmark has been less satisfactory. We believe that its

future can be a great improvement on its past. Following the investigations reported in this paper, we conclude with our expectations as to what this future might be and a few last recommendations about how it should be achieved.

5.1. Comparing planners

It seems clear that planning systems worth comparing on the domain will soon all achieve quadratic-time performance—some have already done so [1,7]—or even linear time, after which all that can be compared on the basis of speed is the efficiency of the coding. So we do *not* think that comparing time performances in solving BW problems near-optimally will be found particularly useful. This is probably why, according to reports of the AIPS 2000 competition, "one comment that was heard at the conference . . . was that now the Blocks World is no longer an interesting problem even as a benchmark." [17]

We expect that much more worthwhile comparisons will be made on the basis of plan length, and for anytime planners, on the basis of their rate of improvement. It will be important to bear in mind that the difference between poor plans such as those produced by US and reasonably good ones such as those of GN2 is not a large proportion of the moves, so apparently small improvements in solution quality may represent significant advances in planning technology. Hence, while we agree that (non-optimal) BW planning is becoming a less interesting benchmark as regards solution time, we strongly disagree with the view that it is no longer a worthwhile benchmark at all.

We believe that another important facet of planning for which BW will prove a suitable testbed is the ease with which formalisms and systems allow domain-specific control knowledge to be specified or automatically learned. For example, the effect of TLPLAN's goal control rule 3 [2, p. 11] is to disallow any plan that is not at least as good as a US plan (i.e., the moves produced are a subset of US moves). This together with the "trigger" rule noted in [1, p. 165] yields an implementation of GN1. The obvious next step is to see how easily more restrictive conditions could be encoded, for instance to disallow non-GN2 moves and recognise singleton deadlocks.

5.2. Using the right problem instances

In making comparisons, care will have to be taken over the choice of problem instances used. For experiments in which optimal plans are sought, it is best to use instances likely to be hard on average, from the region around the peak in Fig. 18 for example. However, it is important to be careful as to what is being tested. The difficulty of the hardest instances is dominated by that of finding minimal cardinality hitting sets, and a planner's ability to do this may well not be seen as the most important thing to assess. Hence, for example, if 100-block problems are used to compare optimal planners, it may be valuable to concentrate on moderately hard instances whose initial and goal states have 10 towers, or 20, rather than on the hardest ones which have 14. This will give more weight to other aspects of reasoning.

---

[17] http://www.ida.liu.sc/ext/witas/achiev/aipscom/page.html.

For experiments in which optimality is not an issue, it is clearly good to use uniformly distributed random instances in order to obtain meaningful results on average behaviour. In any case, there is no point in seeking especially hard instances for that purpose, since there is no significant difference in difficulty or plan length between the average problems and the hardest.

5.3. Beyond experiments

BW's place as an experimentation benchmark need not exhaust its future contribution. Like the Traveling Salesman [20], it may constitute a basis for investigating other problems of more practical interest. The abstract problem of which BW is a instance is the following: sets of actions producing the goal conditions (constructive actions) cannot be consistently ordered so as to meet all of their preconditions (that is, deadlocks occur). To resolve each deadlock, a number of additional actions have to be introduced (non-constructive actions). Since deadlocks are not independent, there is a need to reason about how to resolve them all using as few additional actions as possible. This core problem is present in other more realistic situations. For instance, moving blocks is not very different from moving more exciting objects such as packages, trucks and planes. Again, the version of BW in which the table has a limited capacity captures the essence of the container loading problem, a problem that is crucial to efficiency of freight operations [24,33].[18] Therefore, finding the right generalisations of strategies that are effective for BW appears promising as an approach to more sophisticated problems.

There is more. Existing classes of tractable planning problems do not seem to capture the properties of domains like BW. In SAS$^+$-US [4] or the restriction of STRIPS to ground literals and operators with positive preconditions and one postcondition [5], planning is tractable while optimal planning and even near-optimal planning are not [26].[19] Yet, near-optimal planning *is* tractable for certain domains like BW which are too sophisticated to be encoded within such classes.[20] The fact that the identification of tractable subclasses of SAS$^+$ planning originated in the careful examination of a toy problem in sequential control [3, p. 29] suggests that BW is a good candidate for identifying in a similar way a new class of planning problems for which near-optimal planning is tractable.

5.4. Summary

We take the present paper to have cleared the way to the effective use of BW as a benchmark. Firstly, our methods for generating random instances with meaningful distributions enable systematic experiments to be performed. Secondly, by presenting

---

[18] At the minimum, the "classical" version considered in this paper, in which the table capacity is unlimited, is a useful relaxation of the limited-capacity version. An optimal plan for the former yields an underestimate of the plan length for the latter and thus an admissible heuristic.

[19] The intractability of near-optimal planning for SAS$^+$-US follows directly from the corresponding intractability result for the mentioned subclass of STRIPS [26] and from the inclusion of this latter subclass in SAS$^+$-US.

[20] BW planning is approximable within a constant because (1) a constant number of non-constructive actions suffices to resolve all the deadlocks involving a given constructive action, and (2) the non-constructive actions can be introduced in such a way as not to create new deadlocks.

linear-time algorithms for near-optimal BW planning within a factor of 2, we have closed the question of its time complexity. These algorithms and our optimal solver provide a reference point for future experiments. Thirdly, we have empirically established the average plan quality of near-optimal solution methods, thus setting the parameters for evaluation of suboptimal BW planners. Finally, by careful experimentation, we have identified the distribution of hard and easy instances, and isolated the features most relevant to efficient planning for the domain.

While most of our observations and results are aimed at enhancing the understanding of BW needed for assessment purposes, the same understanding is a prerequisite for the larger project of extending insights gained from BW to a wider class of domains. We expect that our investigations will contribute to that project too, and hope that this paper will not be seen merely as a report on how to stack blocks fast.

Acknowledgements

Appendix A. Experimental data

This appendix details the setup for each of the experiments in Section 4. Note that the 2-dimensional graphs in that section are series of lines joining all the data points.

Time Performances (Figs. 8 and 9). The programs were written in C and compiled with gcc using optimisation to level O2. Times were obtained using the C library function `times()`. The experiments were run on a Sun Enterprise 450 under Solaris. The system has four processors and 2Gb of memory. For the near-optimal algorithms, 100 random problems were generated for each size n a multiple of 10000, up to $n = 1$ million. For the optimal algorithm, 100 random problems were used for each size n a multiple of 5 up to $n = 150$. In the optimal case, the median rather than the mean is plotted, because the distribution of times is expected to be heavy-tailed, so that the observed mean depends on the sample size.

Plan Length and Plan Quality (Figs. 11 and 13). For the experiments on plan length, between 100 and 10000 random BW problems were used for each of the 55 sizes we considered, up to $n = 1$ million blocks, forming a test set of some 250000 problems. More specifically, we generated 10000 problems for each size up to $n = 10$ and each size a multiple of 10 up to $n = 100$, 5000 problems for each size n a multiple of 100 from $n = 200$ to $n = 1000$, 1000 problems for each size n a multiple of 1000 from $n = 2000$ to $n = 10000$, 500 problems for each size n a multiple of 10000 from $n = 20000$ to

$n = 100000$, and 100 problems for each size n a multiple of 100000 from $n = 200000$ to $n = 1000000$. For the experiments on plan quality, we used 10000 problems for each size up to $n = 50$ and 3000 problems for each size n a multiple of 10 from $n = 60$ to $n = 140$.

Hardness and Live Blocks (Fig. 14). The figure has been produced by solving optimally 50000 random problems of $n = 100$ blocks, and recording for each of them the number of times PERFECT backtracks, as well as the number of live blocks. The curve shows the median of the hardness cases for each number of live blocks. The live blocks are those which are involved in deadlocks but not as singletons. A block x is involved in a deadlock iff $N^*(x, x)$, where N^* is the transitive closure of the N relation. So the set of deadlocked blocks can be computed in $O(n^3)$ at worst, which is the time taken to calculate both N and its transitive closure. The singleton deadlocks are computed in $O(n^2)$ using the procedure described in Section 3.6, and are removed from the set.

Misplaced Blocks (Fig. 15). The figure has been produced by analysing 10000 random problems of size $n = 1000$ blocks. We also conducted the same experiment on 1000 problems of sizes $n = 20$, 100 and 10000, respectively, and obtained similar figures.

Deadlock-Free Blocks (Fig. 16). The figure has been produced using 10000 random problems for each size up to $n = 50$. We also conducted the same experiment on 100 problems for each size n a multiple of 50 from $n = 50$ up to $n = 400$. This confirmed that the average number of deadlock-free blocks not on the table does not exceed 2 and even tends to decrease slowly with n (1.87 for $n = 400$). The deadlock-free blocks are computed in $O(n^3)$ as indicated above (see live blocks).

Singleton Deadlocks (Fig. 17). The data set for this experiment consisted of 1000 random problems with n blocks for each $n \leqslant 10$ and for each of the following values of n: 15, 20, 25, 30, 35, 40, 50, 60, 70, 80, 90, 100, 150, 200, 250, 300, 350, 400, 500, 600, 700, 800, 900, 1000, 1500, 2000, 2500, 3000, 3500, 4000, 5000, 6000, 7000, 8000, 9000, 10000, 15000, 20000, 25000, 30000, 35000, 40000, 50000, 60000, 70000, 80000, 90000, 100000. For each problem in the set, if $n < 2000$, all blocks were selected, otherwise 1000 blocks were selected at random. Each selected block was tested to see whether any block below it initially was also below it in the goal. For each block selected, this computation takes time linear in the number of blocks below it initially, so we could only afford to consider problems of size up to $n = 10^5$ blocks instead of $n = 10^6$ as in the above experiment on plan length.

Hardness and Towers (Fig. 18). For each configuration (number of towers initially and in the goal from 1 to 50), 1000 random problems of $n = 100$ blocks were used. Since the graph is symmetric about the diagonal, only half of the cases needed to be considered. Even so, altogether more than a million problems with 100 blocks were solved optimally using PERFECT, and the number of backtracks recorded. Experiments for smaller n showed the same easy-hard-easy pattern, the location of the peak increasing with n (see [30]).

References

[1] F. Bacchus, F. Kabanza, Using temporal logic to control search in a forward chaining planner, in: Proc. EWSP-95, Assisi, Italy, 1995, pp. 157–169.

[2] F. Bacchus, F. Kabanza, Using temporal logic to express search control knowledge for planning, Artificial Intelligence 116 (1–2) (2000) 123–191.

[3] C. Bäckström, Five years of tractable planning, in: Proc. EWSP-95, Assisi, Italy, 1995, pp. 19–33.

[4] C. Bäckström, B. Nebel, Complexity results for SAS$^+$ planning, Computational Intelligence 11 (4) (1995).

[5] T. Bylander, The computational complexity of propositional STRIPS planning, Artificial Intelligence 69 (1994) 165–204.

[6] S.V. Chenoweth, On the NP-hardness of Blocks World, in: Proc. AAAI-91, Anaheim, CA, 1991, pp. 623–628.

[7] P. Doherty, J. Kvarnström, TALplanner: An empirical investigation of a temporal logic-based forward chaining planner, in: Proc. 6th International Workshop on Temporal Representation and Reasoning (TIME-99), 1999.

[8] M.D. Ernst, T.D. Millstein, D.S. Weld, Automatic SAT-compilation of planning problems, in: Proc. IJCAI-99, Stockholm, Sweden, 1999, pp. 1169–1176.

[9] T.A. Estlin, R.J. Mooney, Hybrid learning of search control rules for plan-space planners, in: Proc. EWSP-95, Assisi, Italy, 1995, pp. 145–156.

[10] M.R. Garey, D.S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, Freeman, San Francisco, CA, 1979.

[11] A. Gerevini, L. Schubert, Inferring state constraints for domain-independent planning, in: Proc. AAAI-98, Madison, WI, 1998, pp. 905–912.

[12] E. Giunchiglia, A. Massarotto, R. Sebastiani, Act, and the rest will follow: Exploiting determinism in planning as satisfiability, in: Proc. AAAI-98, Madison, WI, 1998, pp. 948–953.

[13] N. Gupta, D.S. Nau, Complexity results for Blocks World planning, in: Proc. AAAI-91, Anaheim, CA, 1991, pp. 629–633.

[14] N. Gupta, D.S. Nau, On the complexity of Blocks World planning, Artificial Intelligence 56 (1992) 223–254.

[15] S. Kambhampati, B. Srivastava, Universal classical planner: An algorithm for unifying state-space and plan-space planning, in: Proc. EWSP-95, Assisi, Italy, 1995, pp. 81–94.

[16] H. Kautz, B. Selman, Planning as satisfiability, in: Proc. ECAI-92, Vienna, Austria, 1992, pp. 359–363.

[17] H. Kautz, B. Selman, Pushing the envelope: Planning, propositional logic, and stochastic search, in: Proc. AAAI-96, Portland, OR, 1996, pp. 1194–1201.

[18] H. Kautz, B. Selman, The role of domain-specific knowledge in the planning as satisfiability framework, in: Proc. AIPS-98, 1998, pp. 181–189.

[19] J. Kvarnström, P. Doherty, P. Haslum, Extending TALplanner with concurrency and resources, in: Proc. ECAI-2000, 2000, pp. 501–505.

[20] E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, D.B. Shmoys, The Traveling Salesman Problem, Wiley, Chichester, 1992.

[21] N.N. Lebedev, Special Functions and Their Applications, Dover Publications, New York, 1972.

[22] S. Minton, Learning effective search control knowledge: An explanation-based approach, Ph.D. Thesis, Carnegie-Mellon University, Pittsburgh, PA, 1988.

[23] Y. Qu, S. Kambhampati, Learning search control rules for plan-space planners: Factors affecting the performance, in: Proc. EWSP-95, Assisi, Italy, 1995, pp. 133–144.

[24] A.E. Rizzoli, L.M. Gambardella, M. Zaffalon, M. Mastrolilli, Simulation for the evaluation of optimised operations policies in a container terminal, in: Proc. HMS99, Maritime and Industrial Logistics Modelling and Simulation, Genoa, Italy, 1999.

[25] M.J. Schoppers, Estimating reaction plan size, in: Proc. AAAI-94, Seattle, WA, 1994, pp. 1238–1244.

[26] B. Selman, Near-optimal plans, tractability, and reactivity, in: Proc. Internat. Conference on the Principles of Knowledge Representation and Reasoning (KR-94), Bonn, Germany, 1994, pp. 521–529.

[27] J. Slaney, Generating random states of blocks world, Technical Report TR-ARP-18-95, Automated Reasoning Project, Australian National University, Canberra, 1995.

[28] J. Slaney, S. Thiébaux, Adventures in blocks world, Technical Report TR-ARP-7-94, Australian National University, Canberra, 1994.

[29] J. Slaney, S. Thiébaux, Blocks World tamed: Ten thousand blocks in under a second, Technical Report TR-ARP-17-95, Automated Reasoning Project, Australian National University, Canberra, 1995.

[30] J. Slaney, S. Thiébaux, How best to put things on top of other things, Technical Report TR-ARP-6-96, Automated Reasoning Project, Australian National University, Canberra, 1996.

[31] J. Slaney, S. Thiébaux, Linear time near-optimal planning in the Blocks World, in: Proc. AAAI-96, Portland, OR, 1996, pp. 1208–1214.

[32] J. Slaney, S. Thiébaux, On the hardness of decision and optimisation problems, in: Proc. ECAI-98, Brighton, England, 1998, pp. 244–248.

[33] T. Slavin, Virtual port of call, in: New Scientist, June 1996, pp. 40–43.