Linear Time Near-Optimal Planning in the Blocks World

John Slaney
Automated Reasoning Project
Australian National University
Canberra, ACT 0200, Australia
John.Slaney@anu.edu.au

Sylvie Thiébaux
IRISA
Campus de Beaulieu
35042 Rennes Cedex, France
Sylvie.Thiebaux@irisa.fr

Abstract

This paper reports an analysis of near-optimal Blocks World planning. Various methods are clarified, and their time complexity is shown to be linear in the number of blocks, which improves their known complexity bounds. The speed of the implemented programs (ten thousand blocks are handled in a second) enables us to make empirical observations on large problems. These suggest that the above methods have very close average performance ratios, and yield a rough upper bound on those ratios well below the worst case of 2. Further, they lead to the conjecture that in the limit the simplest linear time algorithm could be just as good on average as the optimal one.

Motivation

The Blocks World (BW) is an artificial planning domain, of little practical interest. Nonetheless, we see at least two reasons for examining it in more detail.

In the first place, for good or ill, BW is by far the most extensively used example in the planning literature. It often serves for demonstrating the merit of domain-independent techniques, paradigms and planners. See (Bacchus & Kabanza 1995; Kautz & Selman 1992; 1996; Schoppers 1994) for recent examples. In order to assess the benefits of these approaches and the significance of the claims formulated in the literature, it is therefore necessary to know certain basic facts about BW, such as what makes optimal[1] BW planning hard (Gupta & Nau 1992), how it may best be approximated and what BW-specific information our systems must be able to represent and use in order to cope with it. In the cited papers, for example, Bacchus and Kabanza show how specific methods for near-optimal BW planning can be encoded in a general system, while Kautz and Selman exhibit domain-independent techniques that dramatically improve performance for BW. These facts should not be misinterpreted as showing that such systems are really effective for problems like BW unless they match the best domain-specific ones, both in time complexity and in solution quality. We do not suppose that the cited authors are themselves confused on this point, but as long as little is known about the behavior of BW-specific methods, such misinterpretation of their claims is dangerously easy.

The second motivation for studying BW arises from research on identifying tractable classes of planning problems. We note that within some restricted classes of domain-independent formalisms, such as SAS$^+$-US or the restriction of STRIPS to ground literals and operators with positive preconditions and one postcondition, planning is tractable while optimal planning and even near-optimal planning are not (Bäckström & Nebel 1995; Bylander 1994; Selman 1994).[2] However, near-optimal planning is tractable for certain domains that are too sophisticated to be encoded within such classes. BW is one such domain. This suggests that there is more to learn by focusing first on tractable near-optimal planning in the domain-dependent setting, where the specific features responsible for intractability are more easily identified and coped with. Indeed, the identification of tractable subclasses of SAS$^+$ originated from the careful examination of a simple problem in sequential control. BW appears then as a good candidate for identifying in a similar way a class of planning problems for which near-optimal planning is tractable. Again, this requires that we first acquire detailed knowledge of near-optimal BW planning, keeping in mind that it has many properties that are not necessarily shared by other applications.

Although we hope that our investigations will help towards this second goal, our direct concern in this paper is with the first one, i.e. improving the current knowledge of BW to be used for assessment purposes. We shall focus on the performance in time complexity and average solution quality of polynomial time near-optimal algorithms for BW planning. Various methods for near-optimal BW planning within a factor of 2 exist, for which we shall take (Gupta & Nau 1991; 1992) as sources. However, we find that these methods are

[1] In the following, optimal planning denotes the problem of finding a plan of minimal length, and near-optimal planning the problem of finding a plan of length at most k times the minimal, for some constant factor k.

[2] The intractability of near-optimal planning for SAS$^+$-US follows directly from the corresponding intractability result for the mentioned subclass of STRIPS (Selman 1994) and from the inclusion of this latter subclass in SAS$^+$-US.

nowhere clearly formulated and that little is known about their performance.

The paper makes the following contributions. The first part formulates those methods and shows that they can all be implemented to run in time linear in the number of blocks. This improves the cubic upper bound given in (Gupta & Nau 1992). The speed of the implemented programs (10000 blocks in under a second) also makes it possible to look at the plans produced by the algorithms on *large* problems.

The second part then, is devoted to experimental results. We first introduce a technique for producing truly random BW problems, which is a nontrivial task. Experiments on these random problems give us a rough upper bound of around 1.2 on the average performance ratios of the near-optimal algorithms, and suggest that when the number of blocks is large, it makes little difference which of these algorithms (more sophisticated or trivial) is used, because all produce plans of length close to twice the number of blocks on average. Further, though optimal BW planning is NP-equivalent (Gupta & Nau 1992) and though there is a hard lower bound on *absolute* performance ratios tractably achievable (Selman 1994), the experiments lead to the conjecture that on average and in the limit, linear time algorithms could be just as good as the optimal one.

Definitions

Before presenting the algorithms, we shall enter some definitions. We assume a finite set $\mathcal{B}$ of blocks, with TABLE as a special member of $\mathcal{B}$ which is not on anything. Noting that the relation ON is really a function, we write it as a unary S (for 'support'), where $S(x)$ picks out, for block x, the block which x is on. Thus S is a partial function from $\mathcal{B}\backslash\{\textsc{table}\}$ to $\mathcal{B}$, injective except possibly at TABLE and such that its transitive closure is irreflexive. We refer to the pair $\langle\mathcal{B}, S\rangle$ as a *part-state*, and identify a *state* of BW with such a part-state in which S is a total function.

For a part-state $\sigma = \langle\mathcal{B}, S\rangle$ and for any a and b in $\mathcal{B}$, we define: $\textsc{on}_\sigma(a, b)$ iff $S(a) = b$, $\textsc{clear}_\sigma(a)$ iff either $a = \textsc{table}$ or $\neg\exists b\,(\textsc{on}_\sigma(b, a))$, $\textsc{above}_\sigma$ as the transitive closure of $\textsc{on}_\sigma$, and $\textsc{position}_\sigma(a)$ as the sequence $\langle a :: \textsc{position}_\sigma(S(a))\rangle$ 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.

A BW planning *problem* over $\mathcal{B}$ is a pair of states $\langle\langle\mathcal{B}, S_1\rangle, \langle\mathcal{B}, S_2\rangle\rangle$. 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 $\sigma = \langle\mathcal{B}, S\rangle$ is a pair $m = \langle a, b\rangle$ with $a \in \mathcal{B}\backslash\{\textsc{table}\}$ and $b \in \mathcal{B}$, such that $\textsc{clear}_\sigma(a)$, $\textsc{clear}_\sigma(b)$ and $\neg\textsc{on}_\sigma(a, b)$. The result of m in σ is the state $\textsc{res}(m, \sigma) = \langle\mathcal{B}, S'\rangle$ where $S'(a) = b$ and $S'(x) = S(x)$ for $x \in \mathcal{B}\backslash\{\textsc{table}, a\}$.

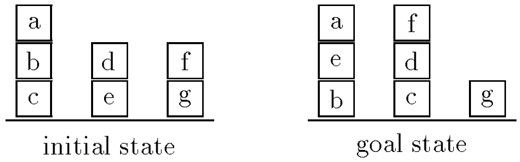

Figure 1: BW Planning Problem

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 $\langle\, \textsc{res}(m_1, I), G\,\rangle$ in virtue of this definition.

We say that a block whose position in I is different from its position in G is *misplaced* in $\langle I, G\rangle$, and that one which is not misplaced is *in position*. Next, we say that a move $\langle a, b\rangle$ is *constructive* in $\langle I, G\rangle$ iff a is in position in $\langle\textsc{res}(\langle a, b\rangle, I), G\rangle$. That is, iff a is put into position by being moved to b. Once a block has been moved constructively it need never be moved 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*.[3] 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 (Gupta & Nau 1992), makes this idea precise. A deadlock for BW problem $\langle I, G\rangle$ over $\mathcal{B}$ is a nonempty subset of $\mathcal{B}$ that can be ordered $\langle d_1, \ldots, d_k\rangle$ in such a way that:

$$\begin{cases} \text{for all } i,\ 1 \leq i < k,\ \ B_{\langle I,G\rangle}(d_i, d_{i+1}) \\ B_{\langle I,G\rangle}(d_k, d_1) \end{cases}$$

where

$$\begin{aligned} B_{\langle I,G\rangle}(a, b) \quad \equiv \quad & \textsc{position}_I(a) \neq \textsc{position}_G(a)\ \wedge \\ & \textsc{position}_I(b) \neq \textsc{position}_G(b)\ \wedge \\ & \exists x \neq \textsc{table}\ (\textsc{above}_I(b, x) \wedge \textsc{above}_G(a, x)) \end{aligned}$$

E.g., the problem in Figure 1 is deadlocked, the deadlocks being $\{\mathrm{a}\}$ and $\{\mathrm{a}, \mathrm{d}\}$.[4] It is easy to see that if $B_{\langle I,G\rangle}(a, b)$ then in any plan for $\langle I, G\rangle$, the first time b is moved must precede the last time a is moved. A deadlock being a loop of the $B_{\langle I,G\rangle}$ relation, at least one block in each deadlock must be moved twice. The first move of this block may as well always be to the table, so as to break deadlocks without introducing new ones. What makes optimal BW planning hard is to choose those deadlock-breaking moves so that it pays in the long term (Gupta & Nau 1992).

[3] By slight abuse of notation, we allow ourselves to speak of moves as constructive in a state, rather than in a problem, of a state rather than a problem as deadlocked and so forth, leaving mention of the goal to be understood.

[4] To see this, note that $B_{\langle I,G\rangle}(\mathrm{a}, \mathrm{d})$ taking the third block in the definition to be $x = \mathrm{e}$, that $B_{\langle I,G\rangle}(\mathrm{d}, \mathrm{a})$ taking $x = \mathrm{c}$, and that $B_{\langle I,G\rangle}(\mathrm{a}, \mathrm{a})$ taking $x = \mathrm{b}$.

Near-Optimal BW Planning

There is a nondeterministic algorithm which solves BW problems optimally in polynomial time (Gupta & Nau 1991; 1992). It consists basically of 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 nondeterministically choose a misplaced clear block not yet on the table and move it to the table.

In the course of their discussion, Gupta and Nau also note three deterministic polynomial time algorithms which approximate optimality within a factor of 2.

US The first and simplest is one we have dubbed US (Unstack-Stack). 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 the simple deterministic version of the above optimal one given on pages 229–230 of (Gupta & Nau 1992). It differs from the nondeterministic version just in choosing *arbitrarily* some move of a misplaced clear block to the table whenever no constructive move is available. We call it GN1 for Gupta and Nau.

GN2 Yet another algorithm is suggested (Gupta & Nau 1991) though with no details of how it may be achieved. This one uses the concept of a 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:

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

Gupta and Nau (1992, p. 229, step 7 of their algorithm) say that in a deadlocked state 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 Figure 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 state 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 state 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 $\sigma = \langle \mathcal{B}, S\rangle$ be a state that occurs during the attempt to solve a problem with goal state $G = \langle \mathcal{B}, S_G\rangle$ and suppose the problem $\pi = \langle \sigma, G\rangle$ is deadlocked. Let b be misplaced and clear in σ. Consider $\text{POSITION}_G(\text{b})$, 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

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

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 $B_\pi(b, \delta_\pi(b))$. 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 Figure 1 shows an example, where $\Delta_\pi(\text{d})$ is $\langle \text{d, a, a}, \ldots\rangle$.

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.

Linear time algorithms

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 (Gupta & Nau 1992) 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 (Chenoweth 1991; Bacchus & Kabanza 1995).

```
function INPOS (b : block) : boolean
    if b = TABLE then
        return true
    if not Examined_b then
        Examined_b ← true
        if Si_b ≠ Sg_b then
            InPosition_b ← false
        else InPosition_b ← INPOS(Si_b)
    return InPosition_b

procedure US ()
    INIT()
    for each b ∈ B\{TABLE} do
        if Clear_b then
            UNSTACK(b)
    for each b ∈ B\{TABLE} do
        STACK(b)

procedure INIT ()
    Plan ← ⟨ ⟩
    for each b ∈ B\{TABLE} do
        Clear_b ← true
        Examined_b ← false
    for each b ∈ B\{TABLE} do
        INPOS(b)
        if Si_b ≠ TABLE then
            Clear_{Si_b} ← false

procedure UNSTACK (b : block)
    if (not InPosition_b) and
        (Si_b ≠ TABLE) then
        (local) c ← Si_b
        MOVE(⟨b, TABLE⟩)
        UNSTACK(c)

procedure MOVE (⟨a, b⟩ : move)
    Plan ← ⟨⟨a, b⟩ :: Plan⟩
    if Si_a ≠ TABLE then
        Clear_{Si_a} ← true
    if b ≠ TABLE then
        Clear_b ← false
        InPosition_a ← (Sg_a = b) and
                        InPosition_b
    else InPosition_a ← (Sg_a = TABLE)
    Si_a ← b

procedure STACK (b : block)
    if not InPosition_b then
        STACK(Sg_b)
        MOVE(⟨b, Sg_b⟩)
```

Figure 2: The US Algorithm

How to make US linear

The key to making US a linear time algorithm is to find a way to compute which blocks are in position in $\mathcal{O}(n)$, 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, as shown in Figure 2. 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.

How to make GN1 linear

For GN1, we add organizational 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 cannot move constructively because its target is either misplaced or not clear.
3. neither of the above.

More variables are now associated with each block. One records the status of this block, while others denote the blocks (if any) which are on this one currently and in the goal. Initialising and updating these does not upset the $O(n)$ running time. To make it possible to select moves in constant time, the blocks of status 1 and 2 are organized into doubly linked lists, 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 the 'ready' list unless that list is empty, in which case it is the block at the head of the 'stuck' list. If both lists are empty, the goal is reached.

When a block a moves, it changes its status as well as its position. At most three other blocks may also change status as a result of the move: the block currently below a (if any), the block that will be on a in the goal (if any), and the block which in the goal will be on the block currently below a (if any). Hence the number of delete-insert list operations is at worst linear in the number of moves in the plan, and so in the number of blocks. Nothing else stands in the way of a linear time implementation of GN1.

How to make GN2 linear

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 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.

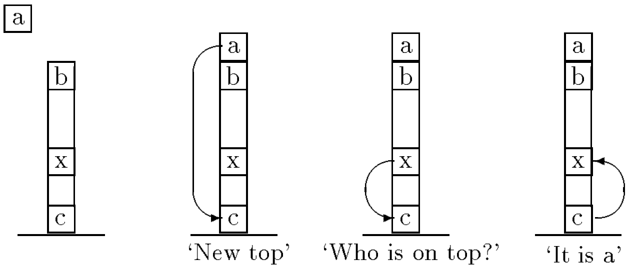

Figure 3: 'Who lives at the top of the tower?'

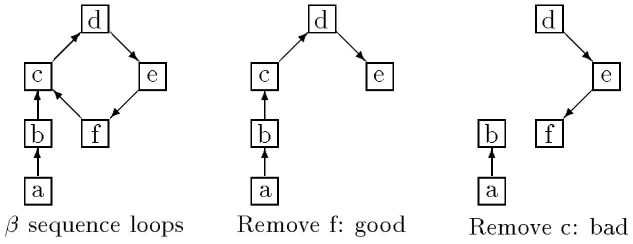

Figure 4: How [not] to break a deadlock

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. 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 Figure 3). 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. In case b is a concierge, there are more variables denoting the clear block above it, and the highest block 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 B_π relation is indestructible except by moving the blocks involved. That is, if $B_{\langle\sigma,G\rangle}(x,y)$ then that relationship persists in the sense that in all future states θ, $B_{\langle\theta,G\rangle}(x,y)$ unless x or y has moved in getting from σ to θ. Moreover, if $B_\pi(x,y)$ 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 B_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 B_π is indestructible β remains linked by B_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 'stuck' list, 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 Figure 4). 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.

Pseudo-code for our implementations of both GN1 and GN2 is given in the technical report (Slaney & Thiébaux 1995).

Generating Random BW-Problems

In designing the experiments to be presented in the next section, we needed a supply of uniformly distributed random BW problems (pairs of BW states). Every state of the chosen size must have the same probability of being generated, otherwise the sample will be skewed and experiments on the average case may be biased. Note that, contrary to what might have been expected, generating such random BW states is not entirely trivial. Naïvely incorporating a random number generator into an algorithm for producing the states does not work: typically, it makes some states more probable than others by a factor exponential in the number of blocks. The unconvinced reader is invited to try by hand the case n=2 with a view to generating the three possible states each with a probability of 1/3 (most methods give probabilities 1/2, 1/4 and 1/4).

To build a BW state of size n, we start with the part-state containing n ungrounded towers each consisting of a single block, and extend this part-state progressively by selecting at each step an arbitrary ungrounded tower and placing it on top of another tower (already grounded or not) or putting it on the table. The difficulty is that the probabilities of these placements are not all the same.

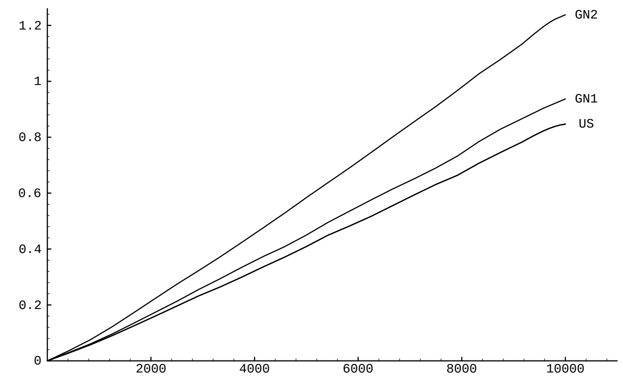

Figure 5: Average CPU time (in seconds)

The solution is to count the states that extend a part-state in which there are τ grounded towers and $\phi+1$ ungrounded ones. Let this be $g(\phi+1,\tau)$.[5] At the next step in the construction process, there will be ϕ ungrounded towers, since one ungrounded tower will have been placed on something. The probability that the selected tower will be on the table in an arbitrary reachable state is $g(\phi,\tau+1)/g(\phi+1,\tau)$ while each of the other possible placements has probability $g(\phi,\tau)/g(\phi+1,\tau)$.

The recursive definition of g is quite simple. First $g(0,\tau)=1$ for all τ, since if $\phi=0$ then every tower is already grounded. Now consider $g(\phi+1,\tau)$. In any reachable state, the first ungrounded tower is either on the table or on one of the $\phi+\tau$ other towers. If it goes on the table, that gives a part-state with ϕ ungrounded towers and with $\tau+1$ grounded ones. If it goes on another tower, that leaves ϕ ungrounded towers and τ grounded ones. In sum:

$$g(\phi+1,\tau)=g(\phi,\tau)(\phi+\tau)+g(\phi,\tau+1)$$

An equivalent iterative definition is:

$$g(\phi,\tau)=\sum_{i=0}^{\phi}\binom{\phi}{i}\frac{(\phi+\tau-1)!}{(i+\tau-1)!}\qquad(\text{for } \phi+\tau>0)$$

In fact, for present purposes it is hard to work with g directly, as the numbers rapidly become too large. For example, $g(100,0)\simeq 2.4\ 10^{164}$. It is better to work with the ratio $\mathcal{R}(\phi,\tau)=g(\phi,\tau+1)\,/\,g(\phi,\tau)$ since this is always a fairly small real number lying roughly in the range $1\ldots\sqrt{\phi}$.[6] Elementary calculation shows that $\mathcal{R}(0,\tau)=1$ for all τ and that

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

[5] As a special case, note that $g(n,0)$ is the number of BW states of size n.

[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 (Slaney & Thiébaux 1995). A corollary of these results is that the average number of towers in a state of n blocks converges to $\sqrt{n}$.

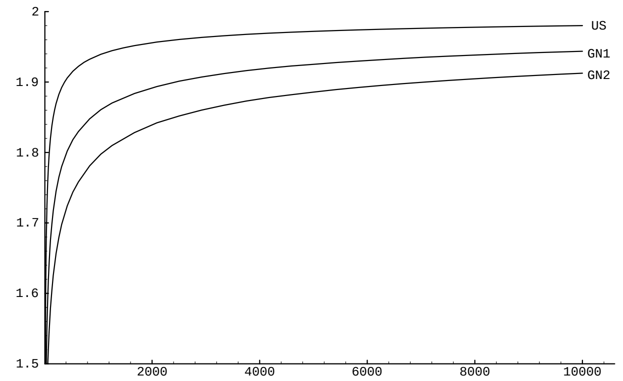

Figure 6: Average plan length: $\frac{\text{number of moves}}{\text{number of blocks}}$

Experimental Results

With a random BW states generator using $\mathcal{R}$ to calculate the relevant probabilities,[7] we generated between 1000 and 10000 random BW problems for each of the 160 sizes we considered, up to $n=10000$ blocks. With this test set of some 735000 problems, we observed the average performance of US, GN1 and GN2. The graphs in this section are continuous lines passing through all data points.

Average Run Times

Figure 5 shows the average runtimes of the near-optimal algorithms as a function of n. As will be clear from the next experiment, there is no significant difference betweeen the average and worst cases runtimes. Times were obtained on a Sun 670 MP under Solaris 2.3. 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.

Average Plan Length

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. Figure 6 shows that the average case approaches this worst case quite closely. 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 (and maybe identical) 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 and if so whether this limiting figure is $2n$. For what our opinion is worth, we conjecture positive answers to both questions.

[7] The generator is made available by the first author at `http://arp.anu.edu.au/arp/jks/bwstates.html`.

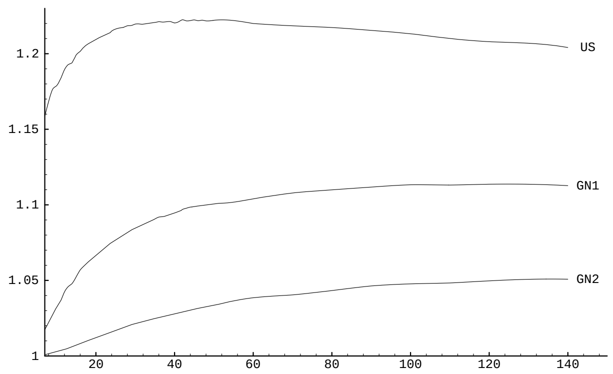

Figure 7: Average perf. ratio: $\frac{\text{plan length}}{\text{optimal plan length}}$

Average Performance Ratio

The *absolute* performance ratio of the near-optimal algorithms is 2 in the limit (for a description of the worst-case instances, see (Slaney & Thiébaux 1995)). Figure 7 shows the *average* performance ratios. The optimal plan lengths were determined by running an optimal BW planner (Slaney & Thiébaux 1995).

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 seems to be that the plan lengths for US quickly approach the ceiling of $2n$, at which point the optimal plan lengths are increasing more quickly than the US ones because the latter are nearly as bad as they can get. We expect 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, Figures 6 and 7 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 of unstacking *all* blocks to the table before building the goal, are optimal on average 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 $0.4n$.

Conclusion and Future Work

By presenting linear time algorithms for near-optimal BW planning within a factor of 2, this paper has closed the question of its time complexity. We hope that both this result and the algorithms we have presented will contribute to clarification and thus to the understanding of BW planning needed for assessment purposes, and that the present paper will not be seen merely as a report on how to stack blocks fast.

While the time complexity questions have been closed, a number of open questions remain concerning the average solution quality of these algorithms. We do not believe that further experimentation alone will answer those questions, because this would require generating optimal BW plans for very large problems, which is impossible on complexity grounds. On the other hand, the theoretical investigation does not appear easy either. Indeed, even to prove (Slaney & Thiébaux 1995) 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.

Therefore, we consider that we have stopped at a convenient point. It appears likely that extensions of the present investigations will belong less directly to planning and increasingly to pure number theory. The most important future direction is to address the second goal mentioned in the introduction of this paper: to exploit our investigations in identifying a class of problems for which near-optimal planning is tractable, and to generalize them to related phenomena in planning domains of greater practical interest.

References

Bacchus, F., and Kabanza, F. 1995. Using temporal logic to control search in a forward chaining planner. In *Proc. EWSP-95*, 157–169.

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

Bylander, T. 1994. The computational complexity of propositional STRIPS planning. *Artificial Intelligence* 69:165–204.

Chenoweth, S. 1991. On the NP-hardness of blocks world. In *Proc. AAAI-91*, 623–628.

Gupta, N., and Nau, D. 1991. Complexity results for blocks world planning. In *Proc. AAAI-91*, 629–633.

Gupta, N., and Nau, D. 1992. On the complexity of blocks world planning. *Artificial Intelligence* 56:223–254.

Kautz, H., and Selman, B. 1992. Planning as satisfiability. In *Proc. ECAI-92*, 359–363.

Kautz, H., and Selman, B. 1996. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proc. AAAI-96*. In this issue.

Schoppers, M. 1994. Estimating reaction plan size. In *Proc. AAAI-94*, 1238–1244.

Selman, B. 1994. Near-optimal plans, tractability, and reactivity. In *Proc. KR-94*, 521–529.

Slaney, J., and Thiébaux, S. 1995. Blocks world tamed: Ten thousand blocks in under a second. Technical Report TR-ARP-17-95, Automated Reasoning Project, Australian National University. `ftp://arp.anu.edu.au/pub/techreports`.