Brothers in Arms? On AI Planning and Cellular Automata

Jörg Hoffmann[1] and **Nazim Fatès**[2] and **Hector Palacios**[3]

Abstract. AI Planning is concerned with the selection of actions towards achieving a goal. Research on cellular automata (CA) is concerned with the question how global behaviours arise from local updating rules relating a cell to its direct neighbours. While these two areas are disparate at first glance, we herein identify a problem that is interesting to both: How to reach a fixed point in an asynchronous CA where cells are updated one-by-one? Considering a particular local updating rule, we encode this problem into PDDL and show that the resulting benchmark is an interesting challenge for AI Planning. For example, our experiments determine that, very atypically, an optimal SAT-based planner outperforms state-of-the-art satisficing heuristic search planners. This points to a severe weakness of current heuristics because, as we prove herein, plans for this problem can always be constructed in time linear in the size of the automaton. Our proof of this starts from a high-level argument and then relies on using a planner for flexible case enumeration within localised parts of the argument. Besides the formal result itself, this establishes a new proof technique for CAs and thus demonstrates that the potential benefit of research crossing the two fields is mutual.

1 Introduction

Cellular automata (CA) are discrete dynamical systems which are frequently used as a model of massively parallel computation [12, 20]. The CAs we consider consist of a collection of *cells*, arranged on a two-dimensional $L \times L$ grid with toroidal boundary conditions, i.e., the grid is the cartesian product of two cycles of length L. Each cell may assume one of two possible contents, `0` or `1`. The contents evolve at discrete time steps according to a *local transition rule*. This rule defines how a cell updates its content according to its current content and the contents of its neighbours.

CAs are generally defined with synchronous updating, i.e., all cells are updated simultaneously at each time step, assuming perfect synchrony. That assumption can be relaxed in many ways, for example by considering that the cells are updated in groups. Such models, called *asynchronous CA*, are yet only poorly understood comparatively with their synchronous counterparts. Asynchronous updating has two advantages: (a) a "noise" in the ordering of the updates may produce new interesting behaviours of the CA (e.g., phase transitions [5]), (b) if the model has to be transposed to a real computing device, this device does not necessarily need a central clock to perform the computations. In the latter case, a simple way to encode the "end" of a computation is to say that the system has converged, i.e., reached a fixed point state in which all the cells are stable.

[1] INRIA, Nancy, France, email: joerg.hoffmann@inria.fr
[2] INRIA, Nancy, France, email: nazim.fates@inria.fr
[3] Universidad Simón Bolívar, Caracas, Venezuela, email: hlp@ldc.usb.ve

Herein, we focus on the case where the updated groups are singletons, i.e., cells are updated one by one. First results on convergence time for such CAs were obtained [7, 6]. In particular, it was noted that, if the cells to update are chosen at random, there exist some rules for which convergence time increases exponentially with size. Now, consider an experiment with random updates that doesn't converge after a given time. What can be concluded? Nothing, because convergence may just need more time. This is where AI Planning comes into the play. For discriminating between convergence and non-convergence, it is sufficient to consider the setting where cells are updated in a controlled way. Hence the question becomes:

Is it possible to *choose* a sequence of updates that drives the system to a fixed point?

Clearly, this question corresponds to a planning problem, hence constituting an interesting new application of AI Planning. Specifically, we propose to apply planners in two ways: (I) deliver information about what kinds of fixed points may be reached using what kinds of strategies; (II) enumerate local cases within human-designed high-level proofs inspired by the outcome of (I). We will refer to (I) and (II) as *global* vs. *local* use of planners. Note that this application requires the flexibility of AI Planning, because many different transition rules are of interest. As a starting point, we focus on one particular rule, called the "binary totalistic CA rule $\mathbb{T}10$".

It has been hypothesised [6] that $\mathbb{T}10$ converges. Our first technical contribution lies in proving this. We prove that, from any given start state, there is a linear-size converging sequence (and hence random convergence is at most exponential). The proof is constructive and shows how to find such a sequence in linear time, hence clarifying also the domain-specific planning complexity of this problem. More importantly, our proof establishes a new *proof method* for the investigation of reachability in CAs, and in particular a proof method that crucially relies on AI planning systems. Our proof starts from a high-level argument decomposing the proof into localised sub-problems. Addressing each of these sub-problems comes down to an enumeration of cases. The number of cases (up to 2^{20} in our proof) makes manual exploration exceedingly difficult. Our key observation is that we can formulate this enumeration as a kind of planning problem, corresponding to application (II) outlined above. In our current proof, we simply wrap FF [11] calls into an explicit enumerator. More generally, this use of planners poses novel requirements, that we will discuss along with the proof.

Our second contribution is the encoding of, and experimentation with, application (I) under $\mathbb{T}10$ in PDDL. The PDDL and a problem generator are publicly available, as a new benchmark for planning.[4] The basic part of the encoding is straightforward: the cell contents are the state variables, and the transition rule yields the action set. The

[4] `http://www.loria.fr/~hoffmanj/CA-Planning.zip`

more subtle question is how to encode the goal of reaching a fixed point. We devise two alternative encodings, **CApddl** using only the STRIPS fragment of PDDL, and **CApddl-dp** using derived predicates [19, 9]. The latter is more concise, but also less accessible as a benchmark because many planners do not handle derived predicates.

Regardless of the encoding, the CA benchmark is interesting because it has a puzzle-like structure where changes made to one part of the puzzle may have detrimental side effects on other parts – changing the content of a cell also changes the neighbourhood of the surrounding cells. In this puzzle-like nature, the new benchmark is similar to well-established benchmarks like Rubic's Cube and the 15-puzzle. In difference to these benchmarks, the CA benchmark encompasses not one but a whole family of problems (one for each different transition rule), and its solution is actually of interest to somebody (the CA community). For the particular rule $\mathbb{T}10$ we consider here, there exists a linear-time domain-specific algorithm (cf. above). As argued in [10], existence of a polynomial-time domain-specific algorithm is a desirable property for a planning benchmark, provided that algorithm is non-trivial: the benchmark then tests whether the planner is clever enough to uncover the relevant structure.

Another interesting aspect of the CA benchmark is that it is suitable alike to challenge: (1) classical planning, where the start configuration of the automaton is known; (2) conformant/contingent planning, where the planner needs to generalise over several possible start configurations; (3) generalised planning parametrised by L, where the ultimate research goal is to automatically construct a domain-specific updating strategy that guarantees to reach a fixed point.

We run large-scale experiments with SATPLAN [13], FF [11], LAMA [17], and T0 [15]. We make a number of interesting observations, in particular that SATPLAN outperforms FF and LAMA by many orders of magnitude. We provide insights into how planner performance depends on the number of update operations required, etc.

Section 2 describes the particular form of cellular automata we consider. Section 3 contains our proof of linear-time convergence. Section 4 explains our PDDL encodings. Section 5 presents our experiments in classical planning. Section 6 summarises them for planning under uncertainty. Section 7 concludes with a brief discussion.

2 Asynchronous Cellular Automata

Let Λ be the two-dimensional square grid of size L^2, with toroidal boundary conditions. That is, we identify Λ with $(\mathbb{Z}/L.\mathbb{Z}) \times (\mathbb{Z}/L.\mathbb{Z})$, where $\mathbb{Z}/L.\mathbb{Z}$ denotes the quotient group of L, i.e., the set $\{0, \ldots, L-1\}$ with addition modulo L. Each cell may be in one of two cell states, 0 or 1. Hence a state of the overall automaton is a tuple $s \in \{0, 1\}^\Lambda$. Figure 1 depicts three states of a grid of size 6, taken by screenshot from a CA simulation tool.[5]

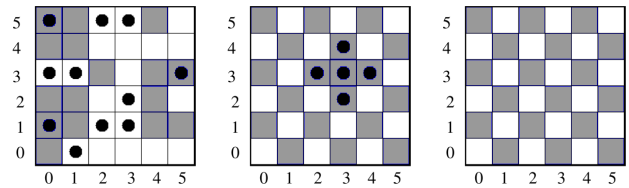

Figure 1. 3 states for $L = 6$; white 0, blue 1; unstable cells have circles.

In what follows, let s be a state. We denote by s_c the cell state of a cell $c \in \Lambda$ in s. Starting from s, and choosing one cell c to update, the system evolves to the state $s' = F(s, c)$ where s' is identical to s except for s'_c. The latter is defined as follows. Let $n = (0, 1)$ and $e = (1, 0)$ denote the north and east vectors, so that $c+n$ denotes c's neighbour to the north, etc. Further, denote $\Sigma_c := s_c + s_{c+n} + s_{c+e} + s_{c-n} + s_{c-e}$. Then $s'_c := f(\Sigma_c)$, where the function f expresses the local transition rule given by:[6]

s	0	1	2	3	4	5
$f(s)$	0	1	0	1	0	0

For example, by updating cell $(3, 3)$ in Figure 1 (b), we evolve to the state shown in Figure 1 (c) because the four neighbours of $(3, 3)$ are all set to 1 so $\Sigma_{(3,3)} = 5$ and $f(\Sigma_{(3,3)}) = 0$.

A cell $c \in \Lambda$ is *stable* if $F(s, c) = s$. A state s is a *fixed point* if all $c \in \Lambda$ are stable. Figure 1 (c) is an example of such a state. We consider the problem of finding a sequence of updates that reaches a fixed point (from a single start state for classical planning, and from several start states for planning under uncertainty). Note that changing the value of a cell c changes the value of $\Sigma_{c'}$ for all neighbours c' of c. Hence this problem is a kind of "puzzle" where implementing a desired effect may have undesired side-effects. We next show that this puzzle can be solved efficiently.

3 Solving the Puzzle

We prove that a classical plan can always be constructed in time $O(L^2)$, i.e., linear in the size of the automaton. The proof is constructive, defining an algorithm that works for all states. Hence, the classical planning problem has an efficient domain-specific solver, and planning under uncertainty can, provided it allows the required constructs (observations and loops), in principle also be effective.

More important perhaps than this result itself is that we obtain it with a novel *proof method*. We employ a planning system for enumeration of cases within a high-level proof argument. While such computer-aided proof design is of course not new in general, to our knowledge it has never yet been applied in the CA area, and certainly it has never yet been done using an AI Planning system as the case enumerator. Our proof works by tackling all 2x2 sub-squares iteratively, bringing them into a desired fixed point pattern by local actions only. The "local" part here, i.e., the moves inside a 2x2 sub-square, is done by a planner – application (II) from the introduction. This technical trick was instrumental for being able to obtain the result. In the proofs for the 2x2 sub-squares, it does not suffice to consider only the 4 cells directly involved. We must also reason about all possible values of their direct neighbours, pushing the number up to 12 cells and hence $2^{12} = 4096$ possible configurations. In fact, for a number of particularly intricate cases, this did not suffice and we had to reason also about the possible configurations of a neighbouring 2x2 sub-square, along with that square's direct neighbour cells, yielding 20 cells in total. Playing through all these combinations by hand, without ever making a mistake, is impossible or at least requires an excessive amount of time and patience. Our proof shows how one can very conveniently leave this task up to a planner instead. We now explain this in detail.

Theorem 1 *Assume the local transition rule* $\mathbb{T}10$. *There exists an algorithm* **fix[**$\mathbb{T}10$**]** *so that, for any grid* Λ *of size* L^2, *and for any state* $s \in \{0, 1\}^\Lambda$, **fix[**$\mathbb{T}10$**]** *run on* Λ *and* s *terminates in time* $O(L^2)$, *and returns a sequence of* $O(L^2)$ *updates reaching a fixed point.*

[5] FiatLux `http://webloria.loria.fr/~fates/fiatlux.html`

[6] This rule is called "totalistic" because it depends only on the *sum* of the cell states of the cell's neighbours. It corresponds, in the notation of Wolfram [20], to the totalistic rule $\mathbb{T}10$; see [6] for more details.

We prove the theorem by defining a suitable algorithm **fix[**$\mathbb{T}10$**]**. **The full proof details, including the entire machinery for case enumeration, are available in the zip file specified in Footnote 4.** In what follows we provide a high-level description for even $L > 2$. For $L = 2, 3$ the theorem is trivial. For odd $L > 3$, **fix[**$\mathbb{T}10$**]** can be suitably extended; we will outline that extension.

As indicated, **fix[**$\mathbb{T}10$**]** works iteratively on 2x2 sub-squares. This is done in rows bottom-to-top, left-to-right within each row. Accordingly, denote in what follows by $Q_0, \ldots, Q_{n-1}$, where $n = (L/2)^2$, the 2x2 sub-squares in that order. We will also use the notation $Q_i = Q_{y*L/2+x}$ where $0 \leq x, y < L/2$. E.g., $Q_4 = Q_{1*L/2+1}$ is the middle square in Figures 1 (a–c), and $Q_8 = Q_{2*L/2+2}$ is the top right square. **fix[**$\mathbb{T}10$**]** selects updates to achieve the particular fixed point where in each Q_i the top left and bottom right cells are set to `1` whereas the other cells are set to `0` – the kind of fixed point as in Figure 1 (c).[7] We will refer to this setting of Q_i as the *checkerboard pattern*. The high-level structure of **fix[**$\mathbb{T}10$**]** is given in Figure 2.

Input: Grid Λ of even size $L > 2$, state s
Output: Sequence of cell updates reaching a fixed point

(1) **if** s is a fixed point **then** stop **endif**
for $0 \leq i < n - 1$ **do**
(2) In case Q_i is stable, propagate instability into Q_i without affecting any Q_k, $k < i$
(3) Acting only on Q_i and a neighbouring square Q_j, $j > i$, bring Q_i into the checkerboard pattern
endfor
(4) Acting only on Q_{n-1} and its direct neighbour cells, bring Q_{n-1} into the checkerboard pattern, and undo any changes to the affected neighbours

Figure 2. High-level structure of **fix[**$\mathbb{T}10$**]** algorithm for even $L > 2$.

First, two trivial but necessary observations: (a) we can act only on unstable cells, since updating stable cells has no effect; (b) any updating action changes only the content of the updated cell. Due to (a), we need the sanity test (1) as well as step (2). Due to (b), we can make our changes *locally*. If we act on a square Q_i then all other squares retain their content (although their stability may be affected).

Step (2) is trivial based on the observation that, if any cell c is currently stable but a neighbour cell c' is not, then updating c' leads to a state in which c is unstable. Hence, to propagate instability into Q_i, we can simply start at an unstable cell c' to the top and/or right of Q_i, and connect c' to Q_i by moving horizontally to Q_i's x position, then down to Q_i's y position. This does not affect any Q_k, $k < i$.

Step (3) is much more intricate. How to bring Q_i into the checkerboard pattern without affecting any Q_k, $k < i$? We need to determine appropriate updating sequences for every possible state of Q_i and its neighbour cells. Figure 3 (a) depicts this localised sub-problem.

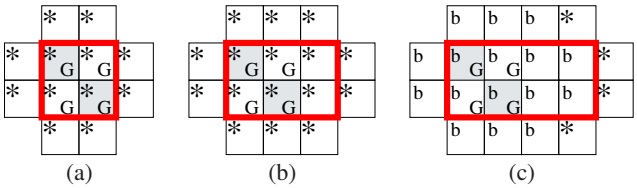

Figure 3. Local sub-problems tackled in the proof of Theorem 1.

The "*" in the cells of Figure 3 (a) mean that each of the two cell states `0` and `1` is possible. The area enclosed by the (red) boldface rectangle is the one we may act on, i.e., these are cells for which we may apply updates. The cells marked with "G" are the goal cells, i.e., we wish to reach a state where these have the values indicated by their colour (white `0` vs. grey `1`).

To design a proof that we can tackle the localised sub-problem of Figure 3 (a), we need to distinguish $2^{12} = 4096$ cases. Our initial attempts to do this by hand were not fruitful. However, note that Figure 3 (a) is essentially a planning problem. For any of the possible initial states of the cells, can we reach the depicted goal state by acting only on the cells within the boldface rectangle? This is a planning problem with deterministic operators, uncertain initial state, and full observability. This profile is not a match for many planners, but it is e.g. for MBP [4]. However, there is another feature we require:

(*) We want to obtain results also for tasks with unsolvable initial states, and we want those states to be listed in the planner output.

It will become clear shortly why this is important. For the following proof, we simply implemented a generator that enumerates all initial states and, for each, creates a classical planning task (based on the PDDL encoding described in Section 4). We run FF on these tasks; any other complete and terminating classical planner would do.

Running FF in the described way on the Figure 3 (a) sub-problem, we determined that 659 of the initial states are actually *not* solvable in this way. It does not suffice to act only on the cells of the 2x2 square Q_i in question. We hence consider the extended sub-problem of Figure 3 (b). This sub-problem allows to act also on the left half of Q_j with $j = i + 1$, c.f. step (3) in Figure 2. This is admissible because **fix[**$\mathbb{T}10$**]** tackles the Q_i by increasing i, so $i + 1$ is still unaddressed and can be modified without destroying prior work. Now, denoting $Q_i = Q_{y*L/2+x}$ as explained above, if $x = L/2 - 1$, then $x + 1 = L/2$ so $Q_{i+1} = Q_{(y+1)*L/2+0}$ is in the row above Q_i; this case will be handled below. For now, we consider $x < L/2 - 1$: then, Q_{i+1} is the square directly to the right of Q_i.

The Figure 3 (b) sub-problem has 16 cells and thus close to 65536 initial states (remember that we exclude stable ones). Running FF on those determines that all but 48 of them are solvable. The remaining 48 unsolved cases lead to the sub-problem of Figure 3 (c). Here, we allow to act on *all* cells of Q_{i+1}. We generate the initial states by starting from the 48 cases left unsolved in Figure 3 (b) – illustrated by "b" in the cells in Figure 3 (c) – and extending those with all possible settings of the remaining 4 cells, yielding 768 cases. Note that this is the point where we exploit feature (*), or else we would need to consider 2^{20} cases.[8] Running FF on the Figure 3 (c) tasks determines that they are all solvable, which concludes the argument for $x < L/2 - 1$. For $x = L/2 - 1$, we proceed in exactly the same way, except that now the square Q_j is the one directly above Q_i, i.e., $j = (y + 1)L/2 + x$. Modifying the planning tasks accordingly, like before we obtain 48 unsolved cases for the case corresponding to (b), and only solved cases for the case corresponding to (c).

Step (4) of **fix[**$\mathbb{T}10$**]** addresses $Q_i = Q_{n-1}$. There, we cannot act on any other square Q_j because all those squares have already been dealt with. The latter, however, is also an advantage: we know exactly what the surroundings of Q_{n-1} will be. Designing according planning tasks determines that all cases are solvable, except 2 cases where Q_{n-1} is already stable. For those cases, we designed tasks that allow to act also on the surrounding cells, provided those are brought back to their initial value. Both these tasks are solvable.

[7] There are many other possible fixed points, e.g. the one where all cells are empty. Our choice of the checkerboard is an informed one, based on manual experimentation including runs of FF along the lines of application (I).

[8] We could of course have done a similar reduction already in the step from (a) to (b). This did not occur to us at this point when we first lead the proof, and we have left it that way here since we think it provides a nice step-by-step presentation of the proof method.

For odd L, **fix[**$\mathbb{T}10$**]** needs to consider an additional 3x2 pattern (one per horizontal pass over the grid). Acting only on the pattern itself yields 9195 unsolved cases; extending those to the next 2 cells on the right yields 1056 unsolved cases; extending those to a further 2 cells on the right yields only solved cases. Extending the 9195 unsolved cases of the basic pattern with the 3 cells above immediately yields only solved cases. Finally, the last pattern will be a 3x2 one for which we know exactly the surroundings, and which is solvable in all possible 64 configurations. This concludes the proof of Theorem 1.

Clearly, the proof we have just conducted points out a general proof method for analysing reachability in cellular automata. The method applies whenever the target state can be expressed as a combination of fixed-size local patterns. The human conducting the proof uses planners, as we did, to examine the solvability of each pattern. Assembling the overall proof from this is perhaps not easy in general (although it has been in our case), but certainly much less cumbersome than doing the entire proof by hand. We are currently developing a proof environment for this purpose, allowing the user to conveniently specify the local sub-problems in a manner similar to the notations in Figure 3. Note that the flexibility of AI Planning is instrumental for this proof method and environment, because CA reachability is interesting for many classes of updating rules and target patterns. AI planners allow to exchange those effortlessly.

In our proof here, none of the sub-problems had a prohibitive number of initial states so we were able to enumerate those. Of course, this may not be the case in general, so more clever techniques, such as MBP [4], may pay off. Note however that feature (*) may be instrumental for scalability. It is very natural to consecutively filter out solvable cases, as we did in the step from Figure 3 (b) to Figure 3 (c). Recall that this reduced the number of cases for (c) from 2^{20} to 768. Detecting solvable/unsolvable initial states is not a match for MBP off-the-shelf but could be accomplished by easy modifications (symbolic backward chaining with BDDs). For planners returning policies, provided the planner can handle dead-ends, one could potentially extract the solved states from the policy.[9]

4 PDDL Encoding

In the rest of the paper, we focus on CA fixed point reachability at the global level, corresponding to application (I) from the introduction. As outlined, this question is interesting to the CA community because, run in this way, planners provide information about what fixed points are reachable and how to reach them. This is convenient for manual understanding, and gives valuable input for setting up a proof like the one we just lead (e.g. for choosing the goal pattern).

We encode the transition rule $\mathbb{T}10$ into PDDL. This yields a new benchmark for AI Planning, with a puzzle-like nature that can be solved effectively in principle (Theorem 1), but that is not captured by existing heuristics. In the PDDL, a predicate `on(c)` encodes whether cell c contains a "1". The transition rule yields `update` actions that change the value of a cell, depending on the cell's neighbours. For classical planning, this dependency is expressed by preconditions, for planning under uncertainty we use effect conditions so that updates can be done without full knowledge of the state.

How to encode the goal of reaching a fixed point? PDDL allows us to formulate this using a quantified goal formula ("all cells are in a stable state"). However, that formula is rather complex. Grounding and transformation to DNF entails enumerating all fixed point states. This is not a good idea in theory, because the number of such states may be large. It is an even worse idea in practice, because planner implementations tend to not be effective for this kind of enumeration. FF's pre-process runs out of memory already for $L = 3$.

We herein devise two alternative encodings, **CApddl** and **CApddl-dp**. **CApddl** uses only the STRIPS fragment of PDDL and is hence accessible to the widest possible class of planners. It separates an "updating" and a "fixing" phase, of which the former allows only cell updates, and the latter allows only to prove cells to be stable. The goal is to have `stable(c)` for all cells c. The task is initially in the updating phase, and an explicit `switch` action is used to move to the fixing phase; once that is done, there is no way back. While this may at first appear unnecessarily strict, it is of great advantage for planners whose heuristics are based on ignoring delete lists:

Proposition 1 *For* **CApddl***, if s is a state in the fixing phase, then $h^+(s) = h^{\mathrm{add}}(s) = h^{\mathrm{FF}}(s)$ is the real goal distance of s.*

Here, as usual, h^+ denotes the length of the optimal solution to the relaxed problem (no deletes); h^{add} is HSP's [2] additive approximation of h^+; h^{FF} is FF's [11] relaxed plan based approximation of h^+. Proposition 1 follows from the particular form of the actions applicable in the fixing phase. We have one action for each cell c, whose only effect is `stable(c)`. The precondition refers only to the value of the `on(c)` predicate for c and its neighbours. Hence the actions have only positive effects, and do not influence each other's preconditions; each goal fact corresponds to exactly one of them. This implies that either $h^+(s) = h^{\mathrm{add}}(s) = h^{\mathrm{FF}}(s) = \infty$ (in case there exists a cell that is not stable) or $h^+(s) = h^{\mathrm{add}}(s) = h^{\mathrm{FF}}(s) =$ the number of cells whose action has not yet been applied.

The most important aspect of Proposition 1 is that, if the state is not a fixed point, then applying `switch` leads to a state whose relaxed plan heuristic is ∞. This would not be the case for a more liberal encoding allowing update and fixing actions to be mixed. For other kinds of planners, it is less clear whether this encoding is the most favourable one. Note, however, that all fixing actions can be applied within a single parallel time step.

CApddl-dp differs from **CApddl** in that `stable(c)` is encoded as a *derived predicate* [19, 9]. In a nutshell, derived predicates allow to extend STRIPS-style descriptions with predicates that are not affected by the operators, and whose value in each state is instead defined via evaluating a stratified set of logic programming derivation rules. In our case, this simply means to turn the fixing actions into such rules. The `stable(c)` predicate is then evaluated directly in every state, and the fixing phase can be dropped completely. On the downside, many planners do not handle derived predicates.

Regardless whether **CApddl** or **CApddl-dp** is used, the puzzle-like nature of the problem implies that ignoring delete lists is a rather harmful relaxation. Assume that, in the definition of h^+, the values of derived predicates are derived per-state (per relaxed state, that is) from the derivation rules, exactly as in the original problem – an idealised definition more accurate than known approximations of h^+ in the presence of derived predicates.[10] Even then, we have:

Proposition 2 *For both* **CApddl** *and* **CApddl-dp***, the exit distance from local minima under h^+ is unbounded.*

This exit distance [8] measures the maximal number of actions required to escape a local minimum, and thus gives a measure of how

[9] Note here that a weaker version of (*), where the planner only guarantees to deliver a superset of the unsolvable initial states (i.e., states marked as "solved" are indeed solvable), would suffice for the filtering to be valid.

[10] FF's heuristic [19] assumes that all derived predicates are already true when relaxed planning begins, so on **CApddl-dp** the heuristic is constant 0. In LAMA [17], the relaxed plan heuristic treats derivation rules like actions with 0 cost. Thus (if arbitrary choices are the same) it is identical to **CApddl** in the updating phase, minus the constant summand $L^2 + 1$.

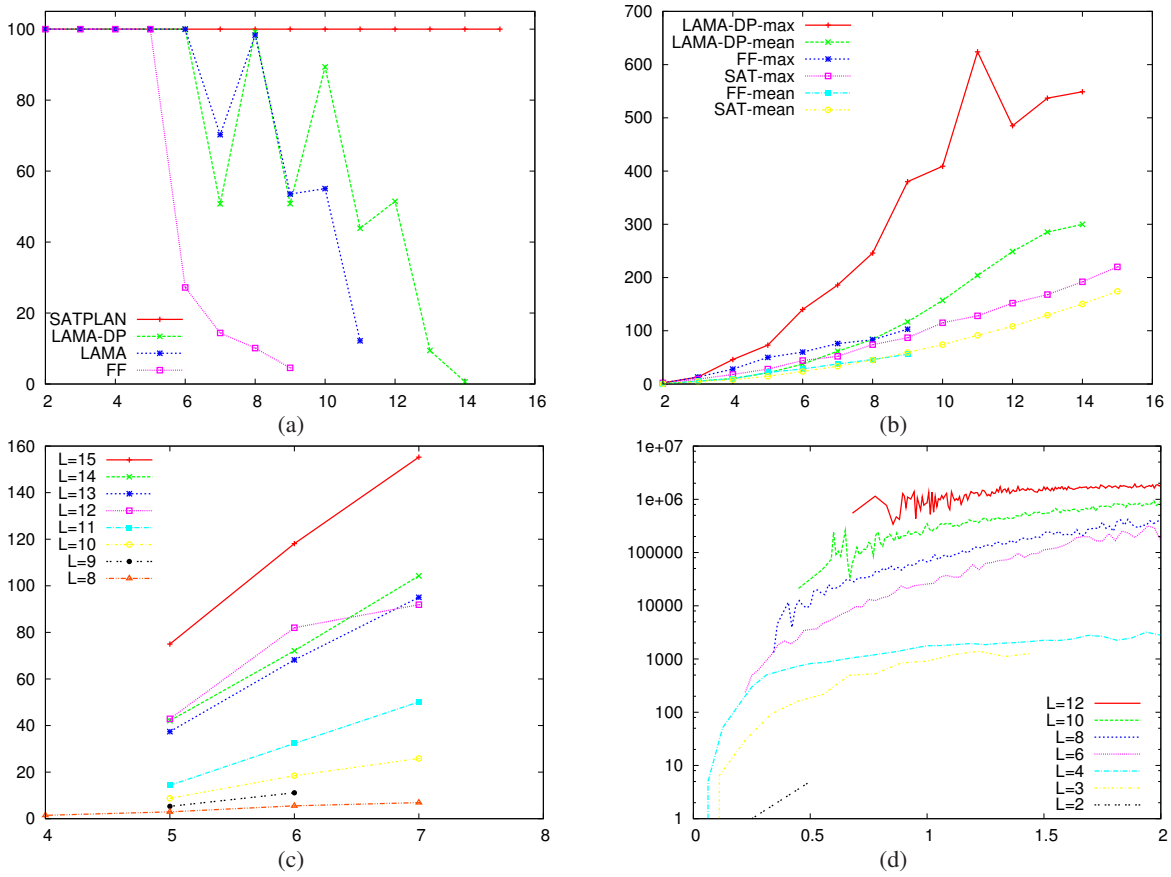

Figure 4. Experiment results in classical planning. (a) Coverage over L, (b) number of updating actions over L, (c) SATPLAN mean runtime over parallel plan layers, (d) LAMA-DP mean number of generated states over $U^* :=$ (number-of-update-actions)$/L^2$. Ordering of keys corresponds (roughly) to relative height of curves. In (c) and (d), for readability, some curves are not shown.

hard it is to "correct" the heuristic estimation error by search.[11] Consider, e.g., a state where only a single cell is `on`. Without delete lists, one only needs to turn the 4 neighbours of that cell `on` as well. However, in the real problem, doing so makes many other cells unstable, and reaching a fixed point requires $\Omega(L^2)$ updates.

5 Experiments in Classical Planning

In the classical setting, for any given L, there are 2^{L*L} possible start states, i.e., possible instances. We consider all these instances for $L = 2, 3, 4$; for $L > 4$ that is not feasible. We consider 10000 instances for each $L > 4$. We run FF [11] as a baseline, LAMA [17] since it performed best in IPC 2008, and SATPLAN [13] since (in this domain) STAPLAN is actually more effective than FF and LAMA. All experiments were run on a CPU running at 2.66 GHz, with a 30 minute runtime cut-off, and with 1 GB memory limit. SATPLAN does not handle derived predicates so we run it only on **CApddl**. FF does handle derived predicates but its *pre-processor* takes $>$ 30 minutes already for $L = 4$ so we also run it only on **CApddl**. LAMA is run on both **CApddl** and **CApddl-dp**, and we will denote the different versions by LAMA and LAMA-DP respectively. Due to performance differences, we run the planners up to different maximal L: 9 for FF, 11 for LAMA, 14 for LAMA-DP, 15 for SATPLAN. This results in a total number of > 550000 test runs. We measured the usual plan length and runtime/search space parameters, as output by the planners. An overview of the results is in Figure 4.

Consider first Figure 4 (a), which shows coverage data – percentage of solved instances – over L. FF exhibits a dramatic performance decline as we reach $L = 6$. LAMA does better, but also fails to scale beyond $L = 12$ with derived predicates, and $L = 10$ without.[12] SATPLAN, on the other hand, scales to $L = 15$ quite comfortably. If not anything else, this certainly shows that FF and LAMA are very bad indeed at uncovering the relevant structure of this domain. Importantly, SATPLAN is far away from a "definite answer" to the domain. For $L = 16$, SATPLAN does not solve a single instance: the formulas become too big to fit into (even 2GB) memory.

The difference between FF and LAMA appears to be mostly due to their respective search procedures. LAMA profits enormously from deferred evaluation [16]: the number of generated states is typically around 2 (and sometimes even 3) orders of magnitude higher than the number of expanded ones. On the other hand, as can be seen in Figure 4 (b), it seems that LAMA's search efficiency comes at the price of overlong plans. (For readability, we show only LAMA-DP here; the behaviour without derived predicates is similar.) Note that even the mean length of LAMA-DP's solutions is larger than the maximum lengths for SATPLAN.

Figure 4 (c) offers a partial explanation of SATPLAN's efficiency. As one would expect, SATPLAN's runtime grows steeply with the number of parallel plan layers. However, the maximum number of

[11] FF's "enforced hill-climbing" search method, e.g., attempts to escape local minima by breadth-first search, which is exponential in exit distance.

[12] Note the zig-zag pattern: even L is easier for LAMA than odd L.

such layers is nearly *constant* over L – the little variance in our data likely arises due to the (incomplete) sampling of start states.

We remark that short plans alone are not the "key" to SATPLAN's performance. We ran IPP [14] to double-check whether the use of planning graphs and parallel plans suffices, or whether the power of modern SAT solvers is required. The result is very affirmative of the latter: IPP solves less than 1% of the instances even for $L = 5$.

In Figure 4 (d), each data point arises from considering the set of instances identified by U^* and the respective L; the y value then is the mean number of generated states, over these instances. U^* is defined, for each individual plan, as the number of updates in the plan divided by L^2, i.e., the number of updates normalised against the grid size. Intuitively, U^* is a measure of how "dense" the puzzle is. For $U^* > 1$, there are more updates than grid cells, which can only be due to harmful interactions within the puzzle (unless the planner includes actions that are completely superfluous). We can see that, with some exceptions, LAMA-TD tends to find instances with larger U^* more difficult.[13] LAMA and FF behave similarly.

6 Planning under Uncertainty

The CA benchmark is also suitable to challenge planning under uncertainty (several possible start states), and generalised planning (value of L not fixed). Consider first the conformant planning problem of obtaining a sequence of actions that achieves a fixed point for any possible valuation of the `on(c)` predicate. We do not know whether there exists a polynomial-time domain-specific conformant planner (our algorithm **fix[**$\mathbb{T}10$**]** involves observations). Note that, for this problem, each value of L yields a single planning instance only.

We performed experiments using the planner T_0 [15], that translates a conformant planning problem into classical one. A key problem is that T_0's clever encoding techniques do not help in this domain (conformant width is equal to the number of cells). The generated PDDL tasks are huge. FF solves the classical problem corresponding to instance 2×2 in 0.07 seconds, obtaining a plan with 33 steps. FF runs out of 14 GB memory while trying to solve the 3×3 instance. LAMA finds a solution to that instance, a conformant plan of 984 steps. The conversion from PDDL to SAS takes 63 minutes, the search pre-processing takes 152 minutes, and the search itself takes 8.26 minutes. We remark that the sum of the updating actions in all of LAMA's classical plans for $L = 3$ is 2786, hence the conformant plan returned by LAMA contains significant generalisations.

One can extend the conformant problem by allowing to observe whether a cell is `on` or not, obtaining a contingent planning problem. We tried the contingent planner CLG [1], an extension of T_0, on this encoding. We were not able to obtain any result.

Finally, the "grand challenge" is to generalise over different values of L. The ultimate goal would be to automatically construct an algorithm with properties like **fix[**$\mathbb{T}10$**]**. Note that this requires, apart from constructing loops and branches, a generalisation over the concrete objects (the cells) available in any given instance. This goes well beyond the capabilities of, e.g., recent work on the automatic construction of finite-state controllers [3].

7 Discussion

We identified an interesting new application of AI Planning, in the investigation of fixed point behaviour of cellular automata. The application is performed at two levels: (I) *global* in order to provide insights about which fixed points can be reached and how, (II) *local* in order to enumerate cases within human-made high-level proofs of convergence. The CA community gains a new tool for performing research. The AI Planning community gains a new application, and a new family of benchmarks exhibiting interesting structure.

In planning under uncertainty, our results indicate a profound lack of performance. Possibly, methods more targeted at learning from experience, e.g. [18], could work better than the purely search-based methods we tested so far. We remark that, in classical planning, we have observed much improved performance when asking the planner to achieve a specific state, rather than any fixed point. This variant is not relevant in our application (where planners are supposed to provide information about possible fixed points in the first place), but it may form a more feasible, and hence possibly more suitable, benchmark for planning under uncertainty.

Our main line of current work is the integration of our planning techniques into a CA simulation tool, which we will make available to both communities. We hope that this new connection will inspire other researchers as well.

[13] For large L, both the higher variance in the curves and their starting at larger U^* likely arise because we sample smaller fractions of the set of possible instances. We do not show curves for $L = 13, 14$ because, there, the number of instances solved is too small for this plot to be meaningful.

REFERENCES

[1] Alexandre Albore, Hector Palacios, and Hector Geffner, 'A translation-based approach to contingent planning.', in *IJCAI*, (2009).

[2] Blai Bonet and Hector Geffner, 'Planning as heuristic search', *AI*, **129**(1–2), 5–33, (2001).

[3] Blai Bonet, Hector Palacios, and Hector Geffner, 'Automatic derivation of memoryless policies and finite-state controllers using classical planners', in *ICAPS'09*, (2009).

[4] Alessandro Cimatti, Marco Pistore, Marco Roveri, and Paolo Traverso, 'Weak, strong, and strong cyclic planning via symbolic model checking', *Artificial Intelligence*, **147**(1-2), 35–84, (2003).

[5] Nazim Fatès, 'Asynchronism induces second order phase transitions in elementary cellular automata', *Journal of Cellular Automata*, **4**(1), 21–38, (2009).

[6] Nazim Fatès and Lucas Gerin, 'Examples of fast and slow convergence of 2D asynchronous cellular systems', in *8th International Conference on Cellular Automata for Research and Industry, ACRI'08*, (2008).

[7] Nazim Fatès, Michel Morvan, Nicolas Schabanel, and Eric Thierry, 'Fully asynchronous behavior of double-quiescent elementary cellular automata', *Theoretical Computer Science*, **362**, 1–16, (2006).

[8] Jörg Hoffmann, 'Where 'ignoring delete lists' works: Local search topology in planning benchmarks', *JAIR*, **24**, 685–758, (2005).

[9] Jörg Hoffmann and Stefan Edelkamp, 'The deterministic part of IPC-4: An overview', *JAIR*, **24**, 519–579, (2005).

[10] Jörg Hoffmann, Stefan Edelkamp, Sylvie Thiébaux, Roman Englert, Frederico Liporace, and Sebastian Trüg, 'Engineering benchmarks for planning: the domains used in the deterministic part of IPC-4', *JAIR*, **26**, 453–541, (2006).

[11] Jörg Hoffmann and Bernhard Nebel, 'The FF planning system: Fast plan generation through heuristic search', *JAIR*, **14**, 253–302, (2001).

[12] Andrew Illachinski, *Cellular Automata - A discrete universe*, World Scientific, 2001.

[13] Henry Kautz and Bart Selman, 'Unifying SAT-based and graph-based planning', in *IJCAI'99*, (1999).

[14] Jana Koehler, Bernhard Nebel, Jörg Hoffmann, and Yannis Dimopoulos, 'Extending planning graphs to an ADL subset', in *ECP'97*, (1997).

[15] Hector Palacios and Hector Geffner, 'Compiling uncertainty away in conformant planning problems with bounded width', *JAIR*, **35**, 623–675, (2009).

[16] Silvia Richter and Malte Helmert, 'Preferred operators and deferred evaluation in satisficing planning', in *ICAPS'09*, (2009).

[17] Silvia Richter, Malte Helmert, and Matthias Westphal, 'Landmarks revisited', in *AAAI'08*, (2008).

[18] Siddharth Srivastava, Neil Immerman, and Shlomo Zilberstein, 'Learning generalized plans using abstract counting', in *AAAI'08*, (2008).

[19] Sylvie Thiebaux, Jörg Hoffmann, and Bernhard Nebel, 'In defense of PDDL axioms', *AI*, **168**(1–2), 38–69, (2005).

[20] Stephen Wolfram, *A new kind of science*, Wolfram Media Inc., 2002.