International Journal of Applied Engineering Research ISSN 0973-4562 Volume 12, Number 4 (2017) pp. 540-548
© Research India Publications. http://www.ripublication.com

AI Planning Search In Direct Access Bit Array Based Representation

Joseph I. Zalaket[1] and Joe M. Khalil

Faculty of Engineering, dept. of Computer Engineering,
Holy Spirit University of Kaslik,
Jounieh, Lebanon.

[1]ORCID 6506327855

Abstract

Artificial Intelligence planning problem consists of finding a sequence of applicable actions that allow a planner to move from an initial state – defined by some state variables in a given representation – to a goal state. This can be done using different planning algorithms which may be based on direct brute-force search of in more advanced heuristic search. The main restrictions in AI planning problems are space and time complexities which depend in general from the data representation and the used planning algorithm.

In this paper we present a novel data representation for AI panning problems that can serve the most of the existing planning algorithms. The proposed data representation is based on a variety of bit arrays used to represent the planning search space and to allow a speed search through it. The objectives of this representation are: in one hand, to decrease the memory space consumption which is critical for all AI algorithms in order to solve bigger problems and in the other hand, to allow an in-memory quasi-direct access in search space which is crucial for reducing the time complexity of planning algorithms. Using bit arrays decreases the space consumption due to the allocation of bits instead of strings for all states and state variables; it also allows accessing data through quasi-direct array indexes instead of string matching search technics. To test the efficiency of the proposed representation, a depth first search planning algorithm is implemented and evaluated.

Keywords: Artificial Intelligence; Planning; Search Space; Bit Array

INTRODUCTION

Nowadays Artificial Intelligence planning is used in a many fields, such as robotics, medicine, aviation, industry etc. [1] It consists of choosing a sequence of actions to be done in a specific order to let the planner move from an initial state, to a goal state. A state is defined by a set of variables' values that can be propositional as in some representations and they are called fluents or atoms, or multivalued called state variables. An action is an operation performed that changes the value of one or more variables, and changes consequently the state, it has pre-conditions and post-conditions that help the planner to choose the right action or actions to be applied at a given state. Since AI planning was introduced in many fields that need a very fast, accurate and efficient decision making and doesn't tolerate significant delays and results looseness (e.g. medicine, military, industries and gaming etc...), for these reasons speed and precision are critical in all AI planners, and they require efficient algorithms, with as low as possible space and time complexities, to be applied on data saved in primary memory. Data cannot be saved in the secondary memory, because despite its bigger capacity, the latency that results when reading from static memory is not tolerable in a planning problem. Normally, these algorithms should be domain independent to work in many situations independently from the problem that needs to be solved. For instance the same algorithm should work in blocks world, logistics, or any other problem giving a right and precise answer within the same complexity concerning time and space. As a consequence, variables should all be saved on main memory, as well as possible actions each with its preconditions and effects, and the planner should keep track of the information of every state he evaluated with the corresponding action. This will make a huge amount of continuously increasing data when dealing with planning problems, and will limit planners to solve relatively small problems. There have been many attempts to decrease the time and space complexity by changing the data representation from the use of straight-forward propositional variables to finite domain state-variables that decrease the size of the state and accelerate access to it in search and update operations [2]

Introducing new heuristic methods and different planning algorithms such as HSP [4],and FF [5] and this research area is still open for any innovation that can help solving bigger problems faster and more efficiently. To make improvements in AI planning, two things must be taken into consideration: reduction of space and time complexities. Reducing space complexity means solving bigger problems some attempts have been done in this mean by reducing the search space like with GraphPlan [6], others by proposition new data representation like with SAS+ [7]. Reducing time complexity means solving more complicated problems many attempts have been done in this mean especially through the use of heuristic search like in HSP [4], FF [5] and the Fast Downward [8]. Consequently, the problem of space and time complexities can at some level be solved by using of bigger memory and more powerful processors by applying some heuristics and advanced algorithms, and. However, the increase in the amount of data is huge as well as the increase of the complexity of computations through time. So finding a better and more adapted representation for data makes space complexity shrink and time complexity affected by the way to access and update data. In this paper we propose the usage of a bit array based representation to store data, allocating one bit for each value of the finite-domain variables; so we will gain in space using this compact representation. Although, since the representation is array based, the access to data in search and compare and update operations becomes faster and thus, the time complexity of these operations decreases compared to string manipulations through string matching and replacing. In the rest of this paper, we will start by stating some variable representations that were initiated through time, we will present our proposed bit array representation, we will present a breadth-first search algorithm running on the proposed bit array representation and we will discuss our results before concluding our work.

REPRESENTATIONS OVERVIEW

There exist many data representations, the commonly used ones are STRIPS [9], PDDL [11] and SAS+ [7]. They can all be used for representing planning problems, but they differ by the form of presenting, accessing, analyzing and updating data, and each algorithm is done based on some specific representation, and optimized to be used with data stored following the accordingly designed representation.

STRIPS

In STRIPS representation [9], the problem is represented by a tuple $\Psi = (P, O, I, G)$:

- *P:* is a set that represents all propositional facts (also called conditions and ground atomic formulas).
- *O:* is a set that groups all operators or actions, each o $\in O$ defined by a triple (pre(o), add(o), del(o)) with pre(o), add(o) and del(o) $\subseteq P$ and where pre(o) is a set that defines preconditions of the operator, add(o) and del(o) define positive and negative post-conditions respectively.
- *I* and *G* are both states $\subseteq P$, where I represents the initial state and *G* the desired state also called goal state.
- Three sets of actions exist in this representation in addition to previously defined ones, PRE(p), DEL(p) and ADD(p), they define actions that have p as one of their precondition, delete or add effect respectively [10].

To apply an operator to a state, the operator precondition must be included in this state, and after it is performed, we apply delete and add to move to another state.

In STRIPS the state is represented as a set of positive literals, containing all elements considered in the problem. For example in the logistics domain where we have 4 positions A, B, C and D and 2 packets P1, P2 and one Truck T; the state must include the positive literals of all the considered elements (P1, P2, T):

Suppose T is at A, P1 at B and P2 at D; the state will be described as follows:

At (P1; B) ^ At (P2; D) ^ At (T, A)

The goal state is a partial condition state, it takes the same form like a state but can include only some information of the only considered variables:

Example:

At (P1; A) ^ At (P2; A)

In this example 3 actions are possible: pack (P) unpack (P) and move (T). And will be represented as follows:

Action (MoveT (from; to);

PRECOND: At (T; from)

EFFECT: ┐At (T; from) ^ At (T; to))

Action (Pack (P, from);

PRECOND: At (P; from) ^ At (T, from)

EFFECT: ┐At (P; from) ^ At (P; T))

Action(Unpack (p; to);

PRECOND: At (p; T) ^ At(T, to)

EFFECT: ┐ At (P; T) ^ At (P; to))

Despite the fact that STRIPS has no negated preconditions and doesn’t specify any type for objects, it is accepted as input to the majority of planners, but the need for more advanced features has resulted in creating the PDDL.

PDDL

PDDL (Planning Domain Definition Language) [11] is defined as a standard to represent planning problems, it includes internally the STRIPS representation and adds more features to it, like negated preconditions and specifying object types. It also has conditional effects and safety constraint specifications as well as definition for subactions and subgoals. In its recent version it allows numeric values for variables and Manages many problems in different fields by differing language features subsets.

Unlike STRIPS, PDDL doesn’t have only positive preconditions and two types of effects (add & delete), PDDL only has preconditions and effects (post-conditions), and both contain positive and negative terms.

PDDL has appeared in many versions: PDDL 1.2 added checking for actions expansions existence [11] PDDL 2.1 [12] added numeric fluents, plan-metrics that allow plan optimization, and continuous actions or durable actions that allow more realistic problem analysis. PDDL 2.2 [13] added derived predicates to evaluate dependency between conditions, and timed initial literals in order to detect actions done independently from the plan. PDDL 3.0 [14] added state-trajectory constraints and preferences. And finally PDDL 3.1 joined object-fluents to previous features which are analogous to the numeric fluents introduced in PDDL 2.1. Where numeric fluents map a tuple of objects to a number, object fluents map a tuple of objects to an object of the problem.

As an example to PDDL representation, we're going to take the Gripper task example where there are 2 rooms and 4 balls and a robot that can move balls between these rooms with the use of any of his hands. (the example is taken from a presentation done by Malte Helmert)

- Objects declaration:
- (:objects firstRoom secondRoom ballA ballB ballC ballD leftHand rightHand)
- Predicates:
- (:predicates (IsARoom ?x) (IsABall ?x) (IsAGripper ?x) (robotAt ?x) (ballAt ?x ?y) (freeGripper ?x) (carryingGripper ?x ?y))
- Initial state:
- (:init (IsARoom firstRoom) (IsARoom secondRoom) (IsABall ballA) (IsABall ballB) (IsABall ballC) (IsABall ballD) (IsAGripper leftHand) (IsAGripper rightHand) (freeGripper leftHand) (freeGripper rightHand) (robotAt firstRoom) (ballAt ballA firstRoom) (ballAt ballB firstRoom) (ballAt ballC firstRoom) (ballAt ballD firstRoom))
- Goal State:
- (:goal (and (ballAt ballA secondRoom) (ballAt ballB secondRoom) (ballAt ballC secondRoom) (ballAt ballD secondRoom)))
- Actions (move, pickup, drop)

  (:action move :parameters (?x ?y)

  :precondition (and (IsARoom ?x) (IsARoom ?y) (robotAt ?x))

  :effect (and (robotAt ?y) (not (robotAt ?x))))

  (:action pick-up :parameters (?x ?y ?z)

  :precondition (and (IsABall ?x) (IsARoom ?y) (IsAGripper ?z) (ballAt ?x ?y) (robotAt ?y) (freeGripper ?z))

  :effect (and (carryingGripper ?z ?x) (not (ballAt ?x ?y)) (not (freeGripper ?z))))

  (:action drop :parameters (?x ?y ?z)

  :precondition (and (IsABall ?x) (IsARoom ?y) (IsAGripper ?z) (carryingGripper ?z ?x) (robotAt ?y))

  :effect (and (ballAt ?x ?y) (freeGripper ?z) (not (carryingGripper ?z ?x))))

SAS+

SAS+ [7] is a light modification on STRIPS, but there exist some variations between them. There are two main differences between these representations: on the first hand, SAS+ uses multi-valued state variable to represent facts instead of using propositional fluents, so a number of mutually exclusive propositional atoms can be replaced by one multi-valued variable. So we can describe problems in a natural way, we can also reduce the complexity of some problems solved with restrictions using STRIPS and be able to remove these restrictions, and generalizing state variables to other domains will take smaller steps [3]. On the second hand, for the actions representation, each action has a precondition, post-condition and prevail-condition:

Pre and Post-conditions define the old and new value of a state variable after performing an action, and prevail-condition is a special kind of precondition; it only includes conditions that will remain the same after execution.

SAS+ representation is suitable for many algorithms, but the most fitting one with it is the algorithm for the fast downward planner.

By using the multivalued state variables, in SAS+ the representation of states becomes easier and faster, the variables are always present in every comparison (now always defined), so the procedures of this algorithm can access the required field directly by counting the separators, with no need to compare the value with every portion of the string.

As an example, we'll take the same like in PDDL but with one Gripper:

- State variables

Index	Description	Domain
1	RobotAt	{First, Second}
2	BallAAt	{FirstRoom, SecondRoom, Robot }
3	BallBAt	{FirstRoom, SecondRoom, Robot }
4	BallCAt	{FirstRoom, SecondRoom, Robot }
5	BallDAt	{FirstRoom, SecondRoom, Robot }
6	Gripper	{Free, Carrying}

- Initial state

  <F, F, F, F, F, F>

- Goal state

<u, S, S, S,S,F >

- Actions

Action	Pre	Post	Prv

MoveRobot(X, Y)	<X, u, u, u, u, u>	<Y, u, u, u, u, u>	<u, u, u, u, u, u>
PickUpA(X)	<u, X, u, u, u, F>	<u, R, u, u, u, C>	<X, u, u, u, u, u>
PickUpB(X)	<u, u, X, u, u, F>	<u, u, R, u, u, C>	<X, u, u, u, u, u>
PickUpC(X)	<u, u, u, X, u, F>	<u, u, u, R, u, C>	<X, u, u, u, u, u>
PickUpD(X)	<u, u, u, u, X, F>	<u, u, u, u, R, C>	<X, u, u, u, u, u>
DropA(X)	<u, R, u, u, u, C>	<u, X, u, u, u, F>	<X, u, u, u, u, u>
DropB(X)	<u, u, R, u, u, C>	<u, u, X, u, u, F>	<X, u, u, u, u, u>
DropC(X)	<u, u, u, R, u, C>	<u, u, u, X, u, F>	<X, u, u, u, u, u>
DropD(X)	<u, u, u, u, R, C>	<u, u, u, u, X, F>	<X, u, u, u, u, u>

BIT ARRAY BASED REPRESENTATION FOR PLANNING PROBLEMS

Problem

The major deficiencies in AI planning are the huge space and time consumptions that limit the solvable problems to relatively restricted and small sized problems. As mentioned in the previous chapter, restrictions must be applied on data representation and manipulation, in order to limit the complexity to a determined level. For instance, the restrictions in SAS+ can affect the number of state variables changed by each operation, the number of values in the domain of each multivalued variable, the number of operators that can lead to an effect and obliging all preconditions not changed after the execution to have the same value [3]. All these restrictions are caused by the high complexity, and once the complexity is better in terms of space and time, this increases the capacity of reaching the solution of a bigger and more complex problem with less and less restrictions depending on the complexity amelioration. There have been many attempts to reduce the complexity, but available enhancements on data representations didn't succeed to reduce these complexities enormously, and they still get close results compared to the representations that they intended to improve. For example, using multivalued discrete state variables instead of fluents in SAS and SAS+ has relatively decreased space and time consumptions compared to STRIPS, but computationally when both representations are used for a planning algorithm, they give relatively close results in terms of space and time complexities. As a result, we still can't solve as big and complex problems as we need in the fields where Artificial Intelligence planning is introduced despite the advanced search algorithms mainly based on heuristics, so here comes the need to do domain dependent algorithm in some cases, but researchers are still looking for a better generic and domain independent algorithm that can solve problems from different domains with no restrictions on the chosen domain. This initiates the need for a new data representation beside the current and future advanced algorithms that have made remarkable improvements in terms of time complexity but even this achievement can be enhanced and improved by changing the data representation and the way to access data which are two main causes of the high complexity. A new challenge raises here, the problematic in this research field is doing whatever it takes to improve space and time complexity, so the best way to do it, is to find a better model to store data than in traditional representations (STRIPS, PDDL, SAS+ etc…), and a better way to access data rather than string matching to search and compare preconditions, intermediate states, results etc… So we have to find a lighter easily manageable structure to represent states and a faster – less complex- method to access needed information such as for searching for preconditions and applying post-conditions. Our proposed solution is described in the following paragraph.

Proposed Solution

The most important requirement in our method is to decrease space and time consumptions. Space consumption can be decreased by using a light and compact data representation model, and time complexity reduction should be based on improving the way the algorithms accesses, compares and updates data in the given representation.

Data representation and space reduction

To decrease space consumption, we used the main advantage of SAS+ over other representations to store data, which is the multivalued aspect of variables, so each state variable will be represented in an array of bits (a BitSet), where each bit represents one value of the domain that this variable could have, as a consequence, each array of bit will contain one or many values set to 1 and the others filled with 0. For example: suppose we have a state variable representing the number of packets in a truck, and the truck capacity is 5 packets.

In STRIPS and PDDL (multiple propositional variables):

- zeroInTruck = True/False, oneInTruck = True/False, twoInTruck = True/False, threeInTruck = True/False, fourInTruck = True/False, fiveInTruck = True/False

In SAS+:

- numOfPackets ∈ {0, 1, 2, 3, 4, 5}
- In our new proposition:
- numOfPackets

0	1	2	3	4	5
0/1	0/1	0/1	0/1	0/1	0/1

In this specific example, we have consumed one pointer size for each variable instead of 6 in propositional representation, and 1 bit for each value in the domain instead of 32 bits (size of integer) for each value in SAS+.

When multiple variables have similar representations, we can group them into one bit matrix. So that instead of the name of the variable, it will have an index in the matrix. For example if we have 4 trucks (named Truck 0 – 3), and can contain 5 packets each, they will be represented in a 4x6 matrix.

	0	1	2	3	4	5
Truck 0	0/1	0/1	0/1	0/1	0/1	0/1
Truck 1	0/1	0/1	0/1	0/1	0/1	0/1
Truck 2	0/1	0/1	0/1	0/1	0/1	0/1
Truck 3	0/1	0/1	0/1	0/1	0/1	0/1

Knowing that each row must contain values set to 1 or 0. Sometimes we need to make some modifications in the structure of each variable, so it can fit in the matrix with others. As a result, there will be some restrictions on the matrix like fields that cannot be set. For this reason we sometimes use restriction helpful matrix that has the same size as the original one and specifies the fields that cannot be set. In blocks world for instance, in order to use the same index to access a given block in any row, we need to add a field for each block pointing to itself:

Consider blocks world example for 3 blocks.

- On(A) ∈ {B, C, T}
- On(B) ∈ {A, C, T}
- On(C) ∈ {A, B, T}

In order to put them in the same matrix, we add A to On(A), B to On(B) and C to On(C).

	0 (A)	1(B)	2(C)	3(T)
On(A)	0	0/1	0/1	0/1
On(B)	0/1	0	0/1	0/1
On(C)	0/1	0/1	0	0/1

The restriction helpful matrix in this case will be:

1	0	0	0
0	1	0	0
0	0	1	0

This structure might not look as compact as possible, it is quite sure that we can replace these 6 fields of 1 bit each (which makes 6 bits), with $\log_2(6) = 2.5849 \Rightarrow 3$ bits. Theoretically we can, but it would increase the time complexity and complicate data manipulation operations. So we cannot work on decreasing space without considering time consumption, besides that our goal includes time amelioration.

In some specific cases, the vectors may contain multiple variables where some variables domains is only {0,1} and are closely related to each other, so it will result in a one-field matrix that can be considered as a bit array, but with more than one field that could be set. In blocks world for example, one variable is needed per block to check if a block is clear (can be moved) or has another block put over it. Suppose we have 3 blocks A, B and C put on the table as shown in figure 1.

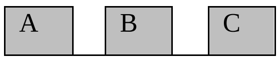

Figure 1

Clear A	0	1
Clear B	1	1
Clear C	2	1

A state can be defined now as a set of matrices and/or bit arrays; it could contain multiple separated matrices and bit sets that will be combined on runtime in one bit set. The major consumption of space happens on runtime. Each state the planner has been through has to be saved for comparisons in order to prevent the planner from getting back to an already visited state and slows down the search or even enters an infinite loop. So at runtime, when we want to save the current state, we flatten it, put it in one bit set and save it, so instead of saving a structure or an object (as in an object oriented programming language), we will be saving one variable that contains exactly the needed number of bits without wasting more space on the structure description and multiple pointers, and we save all the passed through states in an array so we can profit more and more, this will have an advantage on time as well and will be explained in the next step. In fact, the flattening consists of concatenating all rows in matrices and adding bit arrays to them at the end. For example: Suppose a state that contains 3 matrices and 2 bit sets:

Main structure: (random values are used to clarify the flattening process)

Matrix 1	0	1	2	3	4
0	0	0	1	0	0
1	1	0	0	0	0
2	0	0	0	0	1

Matrix 2	0	1	2	3
0	1	0	0	0
1	0	0	1	0

2	0	1	0	0
3	1	0	0	0
4	1	0	0	0

Array 1	0	0	1	0	0	0	0

Array 2	1	0	0	0	0	0

In this case the Flat State will look like this sequence of bits:

|0|0|1|0|0|1|0|0|0|0|0|0|0|0|1|1|0|0|0|0|0|1|0|0|1|0|0|1|0|0|0|1|0|0|0|
0|0|1|0|0|0|0|1|0|0|0|0|0|

This structure will consume less space in the intermediate steps and will facilitate states comparison with each other to cut cycles and prevent infinite loops and with the goal state to define if the goal is reached or not.

ALGORITHM AND TIME REDUCTION

In this paragraph, our goal is not to define or explain a specific algorithm, because we are aiming to initiate a new data representation that fits with many algorithms and should help any applied algorithm to gain more space and time. Despite the differences between planning algorithms, the main functionalities are always the same, and proving that the bit array based data representation helps decreasing complexity in the access to data in searching, updating and states comparison in one algorithm should be enough to generalize this amelioration to cover all algorithms that work based on this representation and access data in the same way.

As a prototype, we used the most basic planning algorithm to prove our proposition, which is a depth-first search based algorithm. The search will be applied in a dynamically constructed tree structure, where at each state we apply all possible actions so we have child states, and each child leads to new children states, and so on until reach the goal state.

The first step in this algorithm is getting the initial and the goal state, flattening then comparing them to see if it needs to search for a plan or if the initial state is already the required one. Flattening the steps will help comparing them as a bit set instead of comparing objects. This makes it easier and faster. Once a bit is detected to be different between the first and the second bit set, the comparison stops. Even though both operations should have at least linear complexities O(n) in the worst case and a constant complexities O(1) in the best case. They have the same order of growth, but the bit set comparison takes less time than the comparison of objects since n in bit set (number of bits) is always less than n in an object holding the same information as the bit set. So in this step, we reduced the time of the operation for sure but not the time complexity.

If the initial state is different than the goal state, the second step of the algorithm is to test all actions' preconditions and to execute applicable functions – knowing that actions are translated into functions with preconditions as if statements, and effects as instructions to do if the conditions are satisfied.

First we will compare the validation process of preconditions. Satisfaction of preconditions is based generally on a defined number of fields, the same in both cases, and should give the same result as well. In the existing data representations, there have to be Strings manipulation on data to find the variable we need to compare with the considered precondition; and here is the major strength of our representation, since we used bit arrays and bit matrices, preconditions will be based on indices. Accessing and comparing a value through its index will have constant time complexity in O(1), differently from String manipulation that has a complexity of O(n*k) in the best case, where n is the number of facts and k is the longest fact length as String. This action will be repeated for every new child node (state) before reaching the goal state. Same thing in applying post conditions: string manipulations are executed once again to find the field and update value.

For example: in the 3 block example used before: to put A on B after it was on C we have to execute the action putOn(x, y) in PDDL where x = A, y = B and z=C;

Pre: clear(x) = true, clear(y) = true;

Post: clear(y)=false, on(x,y)=true, on(x,z)=false

In SAS+

Pre: clear(y)=true;

Prevail: clear(x)=true;

Post: clear(y)=false, on(x)=y.

This is how the action is stored in both cases, so there will be string matching to find variables to compare and edit.

Instead, in our new representation actions will be like:

Pre:

clear(X)[x] = true, clear(X)[y] = true; --these are translated to an "if" condition.

Post:

clear(X)[y]=false, clear(X)[z]=false, on(X,Y)[x][y]=true, on(X,Y)[x][z]=false. --these are instructions to do.

Where x, y and z are the indices for A, B, and C (0, 1 and 2). By this representation, we don't have to search for the variable and set it to a specific value, we use the matrix on(X,Y) with direct index or the bit set clear also with direct index to the wanted field.

In addition, using a flat format bit set representing each state for saving went through states and states comparison is surely less complex than saving and comparing Strings representing states or saving the whole objects with all references and structure information. Besides that comparing a bit set is quite faster and less complex than comparing Strings or objects. Here's another advantage of the new representation that can also be used in applying heuristics in future implementations. Even though we cannot retrieve all information from a flat state and also we cannot know the number of matrices and bit sets contained in this state, but we don't need to; we only need to compare and define if the latest state is similar to one visited before or not. All other information will be defined by 2 more arrays used instead of representing the heavy tree structure; these features are specific for this basic prototype and can differ from a planner to another; but no planning algorithm will have to retrieve information about a state after it is flat, although it is possible, but it causes more delays. The 2 parallel arrays are one for the actions that led to each state and one for the parent of each state; so that when the goal state is reached, we immediately know its ancestors and each action that corresponds to each transition:

As a conclusion, the new data representation is proved to have better space and time complexities:

- In space complexity:

Using one bit for each value in the domain of state variables and grouping relatively close variables in matrices and bit sets –instead of separate propositional atoms (STRIPS and PDDL) and integer or string multivalued variables (SAS+)– is a more compact structure and helps gaining in space. In addition to the flat format used to store and compare visited states consumes less space than saving the object or the structure and the most important aspect is that it decreases the continuously increasing space on the runtime.

- In time complexity:

Two aspects are taken into consideration, the speed of search and the speed of update. The speed of search concerns checking preconditions and visited states as well as retrieving solution at the end, which are all improved with the bit array based representation and we proved that these could be done in less time than if we were working with other applications. The speed of update consists of negation and setting fields as a respond to post conditions also known as operations effects. Time complexity and consumption has improved thanks to the direct and easy access to data through indexes.

RESULTS AND DISCUSSION

It is sure that the depth-first search algorithm works on our new bit array based representation, and gives right sequences of actions so we can always assume having right results. However, giving the right answer is not enough for us; we have to prove that it takes less time. As shown in the previous paragraph, the complexity is better, but we have to prove it; so we're going to represent the same state in STRIPS, SAS+ and our representation, and try to search for some conditions and update some information, and compare time consumption for each of them.

Strips: Init (On (A, D) ∧ On (B, C) ∧ On (C, T) ∧ On (D, T) ∧ Clear (A) ∧ Clear (B) ∧ ┐ Clear (C) ∧ ┐ Clear (D))

SAS+: <D, C, T, T, Y, Y, N, N>

Bit Array based representation:

0	0	0	1	0
0	0	1	0	0
0	0	0	0	1
0	0	0	0	1
1	1	0	0	1

And we're going to put A on B:

[1] check if Clear A and Clear B 3 iterations
[2] Put Clear B to false
[3] Put A on B (or put On(A, B) true)

Average results (ns)	STRIPS	SAS+	Bit Array
[1]	55514	18194	10263
[1]	79305	23325	13528
[1]	53181	19127	13062
[2]	4198965	1236691	27524
[2]	5960003	1921979	25657
[2]	3660158	1249287	20992
[3]	4558637	453904	4199
[3]	3663891	508485	5598
[3]	11611650	506619	4665

Searching and comparing preconditions in addition to applying effects are always present and very important in planning problems, so this example, even if it is basic and simple, doesn't only prove that we are gaining in time, but also that this improvement is important, the difference between results is considerable. The experiments have clearly showed how SAS+ has made an improvement to STRIPS in using multivalued variables, the time delays have decreased passing from STRIPS to SAS+. And finally tests have revealed that our new method takes less time than both STRIPS and SAS+, especially in the update operation. These results cannot be generalized and taken as a reference to prove that our representation is better, since it is done on one case and a little number of iterations is done; but it can clearly support our opinion already proven theoretically.

Besides time improvements; the first String (STRIPS took 113 bytes) the second one (SAS+ took 22 bytes) and the Bit set (Clear has 4 bits, but it might take 1 byte due to memory allocation constraints) and the matrix is an array of 4 bit sets of 5 bits each, suppose the bit set size is 1 byte, it took at most 4 bytes.

Our next step in experiments is to enlarge the problem as much as we can, and take time records for each size of problem starting with 4 cubes.

Number of cubes	Number of visited states	Time (micro seconds)
4	52	67108.87
5	444	335544.3
6	3696	1140850.8
7	30098	2.1139292E7
8	143498	3.70340256E8
9	*Memory saturation*	

We can easily notice the fast expansion of the problem, and the fast augmentation in space and time consumptions, knowing that these results are get with all the ameliorations and enhancements of speed and space economy by using bit arrays. It is normal for the algorithm to stop through full memory at some point; stopping at the existence of 9 blocks is not bad as a result since the algorithm used is a basic one, neither heuristics nor advanced methods are used. This proves that the use of our new representation based on bit arrays is better and would be more helpful and useful if we use it with more advanced algorithms.

CONCLUSION

Artificial Intelligence planning problems have severe space consumption and time complexities. The space complexity problem is problematic, and should be reduced to the minimum in order to have the capacity to solve bigger problems, and the time complexity problem is critical, and should also be reduced to keep the maximum delay at runtime tolerable.

To decrease space consumption, we used the main advantage of SAS+ over other representations to store data, which is the multivalued aspect of variables, so each state variable will be represented in an array of bits, where each bit represents one value of the domain that this variable could have, as a consequence, each array of bit will contain one or many values set to 1 and the others filled with 0.

We have also proved that the bit array based data representation helps decreasing time complexity in accessing data in searching, updating and states comparison in all algorithms that work based on this representation and access data in the same way.

Our light weight array based representation follows a successful time and space economic strategy. In terms of comparison it is better, faster and easier; since comparing bits is much simpler than comparing Strings.

This method, by the use of the array structure, enhances the speed of search and update by the use of indexes pointing to 1 bit. Experimental results have shown the advantage of the new method in finding, comparing and updating results, what brings its advantage over other data access methods used in other planner that have to match strings. String matching complexity is at least linear; data access and comparison complexity through the index is constant.

For the new researchers who would like to enter this field, there is a lot of perspective work: first there is need to implement advanced algorithms and used some good heuristics based on this new representation, like FF [5] , fast downward [8] and others.

REFERENCES

[1] S. S. Shukla and V. Jaiswal, 2013, "Applicability of Artificial Intelligence in Different Fields of Life," International Journal of Scientific Engineering and Research (IJSER), pp. 28-35.

[2] M. Helmert, 2009, "Concise finite-domain representations for PDDL planning tasks," Artificial Intelligence 173, p. 503–535.

[3] C. Bäckström and B. Nebel, 1995, "Complexity Results for SAS+ Planning" Computational Intelligence, Volume 11, Issue 4, pp. 625–655.

[4] B. Bonet and H. Geffner, 2001, "Planning as heuristic search," Artificial Intelligence 129, p. 5–33.

[5] J. Hoffman and B. Nebel, 2001, "The FF Planning System: Fast Plan Generation Through Heuristic Search," Journal of Artificial Intelligence Research, 14, pp. 253-302.

[6] A. L. Blum and M. L. Furst, 1997, "Fast Planning Through Planning Graph Analysis," Artificial Intelligence, 90, pp. 281-300.

[7] R. Huang, Y. Chen and W. Zhang, 2012, "SAS+ Planning as Satisfability," Journal of Artificial Intelligence Research 43, pp. 293-328.

[8] M. Helmert, 2006, "The Fast Downward Planning System," pp. 191-246

[9] R. E. Fikes and N. J. Nilsson, 1971, "STRIPS: a new approach to the application of theorem proving to problem solving," Artificial Intelligence 2, pp. 189-208.

[10] T. Bylander, "The Computational Complexity of Propositional STRIPS planning," Artificial intelligence, 7 March 1994.

[11] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld and D. Wilkins, 1998, "PDDL — The Planning Domain Definition Language," Yale Center for Computational Vision and Control Tech Report CVC TR-98-003/DCS TR-1165.

[12] M. Fox and D. Long, 2003, "PDDL2.1: An Extension to pddl for Expressing Temporal Planning Domains," Journal of Artificial Intelligence Research 20, pp. 61-124.

[13] S. Edelkamp and J. Hoffman, 2004, "PDDL2.2: The Language for the Classical Part of the 4th International Planning Competition," Technical Report No. 195.

[14] A. Gerevini and D. Long, 2006, "Preferences and Soft Constraints in PDDL3," Proceedings of the ICAPS-2006 Workshop on Preferences and Soft Constraints in

Planning, p. 46–54.