

A Case Base Seeding for Case-Based Planning Systems

Flavio Tonidandel and Márcio Rillo

Centro Universitário da FEI - UniFEI
Av. Humberto de A. Castelo Branco, 3972
09850-901 - São Bernardo do Campo - SP - Brazil
e-mails: flaviot@fei.edu.br; rillo@lsi.usp.br

Abstract. This paper describes a Case Base Seeding system (CBS) that can be used to seed a case base with some random cases in order to provide minimal conditions for the empirical tests of a Case-Based Planning System (CBP). Random case bases are necessary to guarantee that the results of the tests are not manipulated. Although these kind of case bases are important, there are no references about CBS systems in the literature even from those CBP systems that claim to use some similar systems. Therefore, this paper tries to overcome this deficiency by modeling and implementing a complete random Case Base Seeding process.

1. Introduction

There are many case-based planning (CBP) systems in the literature [4],[5],[7],[12],[13]. A CBP system is known as a planner that retrieves potential cases from a case base that can become solutions to problems. Any case-based planner needs a case base with suitable cases that allows the system to work properly and to have some advantages over generative planning systems.

Differently of the CBP literature, Case Base Seeding systems are rare in the literature, although they are important for CBP system. In fact, most CBP systems seed their case bases in order to perform their tests. However, these seeding processes are either composed by hand or they are specific for some domains. In addition, there are no available explanations about them in the literature

This lack of a generic and random seeding process in the literature can become an obstacle to CBP systems researches to perform their empirical tests and, consequently, to analyze their systems performance suitably.

In this paper, a Case Base Seeding system is presented by focusing on how to create random and suitable states in planning domains. The seeding system creates a random problem by creating consistent initial and goal states. This random problem is then solved by a generative planner, called FF [3], which produces a solution (a plan), and consequently, a new case for the case base.

This paper is a detailed extension of the CBS System used in [12] and it is structured as follow: in the section 2, some case bases used by some CBP systems are described. In section 3, the main part of the CBS system is detailed followed by the section 4 that describes the complete CBS system. Section 5 discusses some issues of CBS system and, finally, section 6 concludes this paper.

2. Case bases in CBP systems

A Case-Based Planning system is a planner that uses past experience, stored in cases, to solve the problems. Cases in a CBP system are previous plans or traces of planning solutions. In order to be an efficient system, a Case-Based Planner must use a case base with a large variety of cases. This is because the performance of a CBP system depends on the cases in the case base since the adaptation of a similar case can be computational expensive in general. Therefore, to retrieve a very similar case or even a complete-solution case improves the performance of the CBP system by decreasing the time spent by the adaptation phase. Obviously, anyone can design and create, by hand, a case base with many very similar cases to some specific problems in order to increase the performance of the CBP system.

However, to perform any empirical test in CBP systems, a case base with random cases must be available. This random case base would avoid any kind of manipulation that permits a system to work with many very similar cases and that consequently turns the tests not completely valid. The problem is that there is no schema or a formal process to create case bases with random cases available in the literature.

Some CBP systems use pre-defined or given examples to fill their case bases up or to perform their tests, e.g., DerSNLP+EBL [4]. Others systems like CAPER [5] and Caplan/CBC [7] use automatically creation of cases, but they need either a start pivot (Caplan/CBC) or were designed for some specific domains (CAPER – UM Translog Domain).

The Prodigy/Analogy [13] is another CBP system that uses an automatic creation process of cases. This process creates random problems and, consequently, random cases. However, it is specific for some domains, like logistic transportation, one-way-rocket and machine-shop domains, and Veloso [13] does not let available any detail of how this seeding system really works.

To summarize, the cases creation processes used by CBP systems are either domain specific or no specification of how these processes work can be found. In other words, there is no generic process that researches can use to create random cases for CBP systems.

This paper intends to overcome this deficiency by defining a Case Base Seeding system that can be used for any Case-Based Planner to create their own random case base. It is a detailed and extended version of the seeding system presented and used in the Far-Off system [12].

3. Creating Random Consistent States

A Case Base Seeding system can be designed by using a generative planning system to produce sound plan-solutions and, consequently, suitable cases. However, this generative planner just performs its task if an initial state and a goal state are available. In fact, the most important piece of a CBS system is a process that creates correct and consistent states in the application's domain. A state in a planning domain is a set of instantiated predicates [12].

In order to generate random and consistent states in a certain domain, some additional information must be added by the user. Usually, the available information for a planning system is only the domain and the problem features both in PDDL language [6]. The domain features describe the actions and the predicates, besides the types of the elements that the predicates can handle. The problem features describe the initial state, the Goal State and the elements for each type in the domain.

Each predicate can handle some specific types of elements, which are the possible values for the predicate's free variables. A predicate can become a fact when its all free variables are instantiated, i.e., a fact is a grounded predicate.

Definition 1 (a fact): A fact is a grounded (instantiated) predicate.

Definition 2 (the predicate of a fact): The predicate of a fact is the not-grounded format of the fact.

However, these information about domain, predicates, actions and problem are not enough for a CBS system, i.e., it is not possible to create consistent states with this information only.

Therefore, additional information, containing the semantic of each predicate and their relations with others predicates, is necessary. For example, in the Blocks World domain, this additional information must define that *holding* and *handempty* predicates cannot appear together in a state. In fact, this additional knowledge is difficult to be obtained from the information described in PDDL language.

This additional information, from now on simply denominated domain semantic, encapsulates semantic features of the domain in terms of predicates relations that can not be extracted from actions, but only from consistent states features.

To describe the relations among predicates, some definitions are stated. These definitions must encapsulate negative and positive interactions:

Definition 3: (Positive Existence) *The Positive Existence of a predicate p is the set of predicates and facts that must be in the state where p is true.*

Definition 4: (Negative Existence) *The Negative Existence of a predicate p is the set of predicates and facts that can not be in the state where p is true.*

Definition 5: (Positive Absence) *The Positive Absence of a predicate p is the set of predicates and facts that must be in the state where p is not true.*

Definition 6: (Negative Absence) *The negative absence of a predicate p is a set of predicates and facts that can not be in a state where p is not true.*

In order to clarify the use and the importance of the definitions above, the Blocks World domain with 4 blocks (A, B, C and D) can be considered. In this domain, the facts $on(A, C)$, $holding(A)$ and $clear(B)$ are the predicates that are in the negative existence of the fact $on(A, B)$. It means that the facts $on(A, C)$, $holding(A)$ and $clear(B)$ can not be together, with the fact $on(A, B)$ in the same state. In fact, not only $on(A, C)$ can not exist in the same state of $on(A, B)$, but also any predicate of the form $on(A, x)$.

The seeding system permits a generic specification of negative and positive existence and absence. For example, for a $on(x,y)$ predicate, the following predicates can not be in the same state: $holding(x)$, $clear(y)$ and $on(x,_)$. The others definitions encapsulate the complementary three possibilities of Existence and Absence relations of a specific predicate. There are some facts, however, that must be in all states of the domain. They are called fixed facts:

Definition 7 (Fixed Facts) *The Fixed Facts is a set of facts that must be true in all states of the domain.*

In a typed domain, some predicates restrict their variables to some values. An example can be illustrated in Logistic domain. The predicate $at(x,y)$ means that x is at y , where x is a *package* or a *vehicle* and y must be a *location* like *city* or *airport*. However, the semantic of this predicate in a domain is restricted to: $at(airplane,airport)$ and $at(package,y)$, i.e, when the x variable is instantiated with an *airplane*, the *location* denoted by y must necessarily be an *airport*. On the other hand, the type *package* does not require any kind of restriction for the y variable. This information is not explicit in the domain's definition. To describe such feature the following definition is stated:

Definition 8 (Restricted Facts): *The Restricted Facts is a set of facts of a specific predicate that can really exist in a certain domain.*

The purpose of the above definition is to restrict the facts that can be created between the predicate definition and all elements encapsulated in each type. These restricted facts must be provided by the user because there is no information about it in problem or domain features. They can also be automatically extracted from the domain specification through some state invariant extractor, like TIM [1].

Some predicates are even more restricted. They can appear in a state one time at most and others can appear one time at least. This feature is then defined:

Definition 9 (At-Least-One predicate): *The At-Least-One predicate is that predicate which facts can be true in a state at least one time.*

Definition 10 (At-Most-One predicate): *The At-Most-One predicate is that predicate which a fact can be true in a state at most one time.*

With all above definitions, a process to create a consistent state is defined in Figure. 1. As stated before, a process to create a consistent state is necessary to define a process that creates a case base. In order to produce a consistent state, a set of all facts of the domain is necessary.

The process to create this set, called *Available and not Allocate* predicates set (AnA), follows an initial seed example and the restrictions configured by the user.

This initial seed is an example of a problem, including an initial state and all the elements of each type of the domain. This information can also be given by the user, like the Prodigy/Analogy system [13].

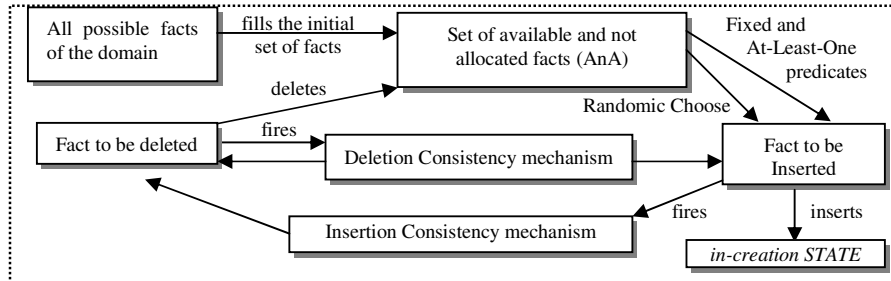


Fig. 1 – The diagram of the consistent state creation process

The most important contribution of the seed example is the determination of the values that can instantiate a fixed predicate. Since the fixed facts are defined as those predicates that must be in all states, they will also be in the initial state of the seed example. Therefore, the possible facts of each fixed predicate can be easily extracted from the seed example.

Besides the number and specification of the elements and the determination of fixed facts, the seed example is also necessary to provide important information about the consistency between the initial state and the goal state. Some predicates accept only some values for their variables, and these values are defined in an example of a possible initial state. For example, in the logistic domain, the predicate $at(truck, city)$ can not be instantiated with any *truck* and any *city*, because some *trucks* are specific of some *cities* and they can not travel to other *cities*. So, the CBS system must consider this information.

In order to complete the restriction specified in definition 8, a specific and special variable must be used to extract information from the initial state of the seed problem example. This variable, called *InSt*, restricts the values of an specific variable to those presented in the seed example.

Observe that the seed example avoids the user to specify all the restricted, fixed facts, and all elements of the domain by hand. Therefore, the process to create the AnA set follows the rules specified below:

- ◆ Extract from the seed example all elements of each type
- ◆ For each predicate, all possible combination of elements in their variables are generated considering:
 - ◆ Each element must instantiate one and only one variable of the predicate, eliminating any fact that has repeated element in their variables.
 - ◆ If the predicate is a fixed fact, then consider only the facts that exist in the seed example.
 - ◆ If an element will instantiate a variable labeled as *InSt*, then consider only the facts that match any other fact in the seed example.

During the AnA set creation, all facts that are grounded At-Least-One predicates compose a specific sub-set and all facts that configure as Fixed Fact compose another specific sub-set.

The second step is to choose one fact from AnA to compose the *in-creation* state. This choosing process follows a heuristic that chooses the most restricted predicates first. These restricted predicates are those classified as Fixed and At-Least-One predicates. This is because both predicates must be in all states and they can probably prune other facts.

The process chooses the Fixed Facts first and then selects At-Least-One predicates randomly. Each selection fires the Consistency Mechanism that applies one of their rules. When all Fixed and any different At-Least-One predicates from the chosen ones are not available in AnA set, the process starts the generic random process that can choose any fact present in AnA.

The CBS system uses a set of consistency rules that guarantee the creation of a consistent state (*in-creation* State). The consistency rules are allocated in two Consistency Mechanisms: Insertion and Deletion Consistency Mechanisms.

Definition 11 (Rules of the Consistency Mechanism)

*In Insertion Consistency Mechanism, there are three rules: For each insertion of a fact f in *in-creation* State, where f is a grounded predicate of the general predicate p :*

- ◆ **Consistency Rule 1 (ICM-R1):**
*All facts that fit in Positive Existence predicates of p are included in the *in-creation* State and deleted from AnA set.*
- ◆ **Consistency Rule 2 (ICM-R2):**
All facts that fit in Negative Existence predicates of p are deleted from AnA set.
- ◆ **Consistency Rule 3 (ICM-R3):**
If the predicate p is configured as The-Most-One predicate, all facts of predicate p are deleted from AnA

In the same way, the Deletion Consistency Mechanism has two rules: For each fact f deleted from AnA, where f is a grounded predicate of the general predicate p :

- ◆ **Consistency Rule 1 (DCM-R1):**
Each fact that fits in the Negative Absence of p must be also deleted from AnA.
- ◆ **Consistency Rule 2 (DCM-R2):**
*Each fact that fits in the Positive Absence of p must be inserted in *in-creation* State and deleted from AnA.*

Therefore, the CBS system performs a repetition of insertion and the application of the Consistency rules. When the AnA becomes empty, the process stops and the *in-creation* State becomes a potential consistent state following the user configuration.

During the application of the consistent rules, it is possible to detect that fail occurred and consistent state can not be created. For example, the rule DCM-R2 can detect there is no predicate that it must insert. Therefore, if any rule fails to accomplish their tasks no consistent state is guaranteed to be created.

The process of deletion and insertion performed by the consistent rules are fired when a fact is deleted from AnA or inserted in the *in-creation* State. Since this process is not cyclic, it can not falls in a infinite and recursive looping caused by a wrong configuration, which would probably accuse fail.

4. Creating a random case base

With a random state creation process defined, the entire process of a case base creation can be designed. As mentioned before, a case is made by a plan generated by a planner. This planner will be the generative planner called Fast-Forward (FF) [3] that is a heuristic search planner. The FF system uses a heuristic (FF-heuristic) to perform its planning task: to find a plan from a given initial state to another state where the goal is satisfied.

4.1. Creating a Random Initial and Goal States

The process to create random and consistent states, described before, is used to create the initial and the goal states. Although the initial state is a complete and consistent state that is straightforward released by the state creation process, the goal state is not. In fact, the difference between initial state and the Goal State is that the latter can not be a complete state. In theory, a goal state may be a complete state, however, as verified in many available planning problems for different domains, only some predicates are allowed to constitute a Goal State. For example, in the Blocks World domain, only the predicate $on(x,y)$ is relevant to the goal. The same observation can be made for $at(x,y)$ predicate in Logistic domain and the $served(x)$ in Miconic domain.

Therefore, to let the Goal State similar to those presented in the problems, it is defined the relevant predicates:

Definition 12 (Relevant Predicates): *The relevant Predicates are those predicates that can be part of a goal state*

The relevant predicates of a domain must be configured by the user or extract from a seed example. The goal state has another restriction: the number of the relevant predicates is also random. For example, in the Blocks World domain with 10 blocks, a goal can be only a composition of two $on(X,Y)$ predicates, like $on(A,B)$ and $on(B,C)$, and not a composition of all possible and consistent combination of $on(x,y)$ predicate.

Therefore, the process to create a goal state can be more than the straightforward use of a complete random state. It requires two filters: The Relevant Predicates Filter and the Random Number of Facts Filter. These filters are applied in a random and consistent state, called first-stage goal state, created by the process described above.

Both filters prune the predicates in the Goal State. The Relevant Predicate Filter will delete any fact from the first-stage goal state that is not a relevant predicate, creating the second-stage goal state with only relevant predicates.

The second filter, the Random Number of Facts Filter, is now applied. Its first step is to choose randomly the number of facts, denominated G_n , which will be left in the state. This number has the range from 1 to the number of facts in the second stage state. The second step is to choose randomly G_n facts of the second stage, creating the third (the final stage) of the goal state.

The final stage of the goal state is then created by G_n relevant facts from a consistent and complete state resulted by the creation process.

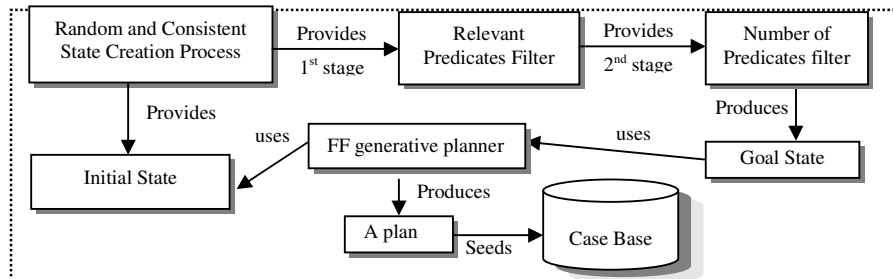


Fig. 2 – The complete model of the Case Base Seeding Process.

4.2. Producing a Plan

With an initial and goal states established, the process to produce a plan is about to start. However, a last verification must be necessary. Until now, consistent initial and goal states are created, but nothing can guarantee that both states are consistent with each other. As a last verification, the FF-Heuristic is used to estimate a number of action between the initial and goal states. The FF-heuristic is very useful for many case base planning purposes [10], because it is a good estimation of the possible number of actions between a state and the goal.

In order to guarantee the consistency between the initial and goal states, the FF-heuristic result, applied for both states, must be a finite number. If the estimation result is infinite, then there is no possible solution between those states. In this case, the process restarts and others consistent states are generated as initial and goal states. Otherwise, if the result estimation is a finite number, then the FF planner is fired to produce a plan between initial and goal states. This plan is transformed to a new case.

A generative planner is necessary because cases are composed by plans or traces of them. Given a sound plan as a solution for a problem, a case can be created. The Figure 2 summarizes the case base creation process in a more general view.

4.3. Maintenance Process for Case bases

As a post-seeding process, a maintenance policy must be applied in the case base in order to let it representative and not redundant. For that, any maintenance policy found in the literature, like min-injury [11], Type-based [9] and case-addition policy [14] can be applied to the case base in order to improve its competence. The competence of a case is the range of problems that it can solve [8].

The maintenance policies will reduce the redundant cases and others cases that do not contribute to increase the competence, but only to create ‘hot spots’ – spots of concentration of cases. After the use of any maintenance process, a short case base will be created, keeping its original competence and an uniform distribution of cases.

If the short case base becomes smaller than necessary, the seeding process can be called to produce more cases to fill the case base up again. This will create a cycle of seeding and maintenance process that will increase the quality and the representativeness of the case base.

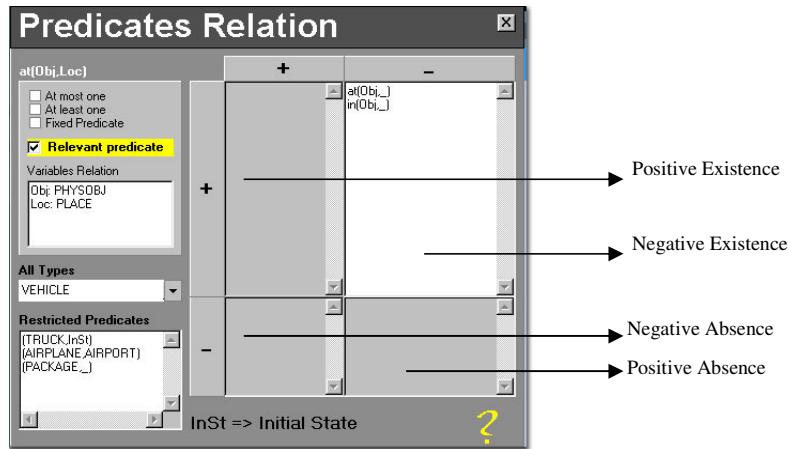


Fig. 3 – Example of the $at(obj,Loc)$ predicate configuration in the CBS.

5. Empirical tests and discussion

The Case Base Seeding process presented in this paper was implemented and used in the Far-Off system [12]. The figure 3 shows the configuration window of the CBS seeding used by the Far-Off system.

In the Far-Off system empirical tests, this CBS was used to create case bases over more than 3000 cases for each problem and domain. The creation of those case bases allowed diversified tests in many STRIPS domains, as showed in [12]. It is important to highlight that the CBS described in this paper concerns about STRIPS-model domains. Therefore, the CBS system is not suitable to describe the semantics of all existed planning domains. This is because the definitions in this paper were extracted only from STRIPS domains, and most of them were in planning competition.

The CBS system can be improved to work with more complex domains that handle numerical parameters and resources. In addition, it can be also improved by using a learning system that can learn and extract information from states, problems and domain defined in PDDL language. An algorithm, called DISCOPLAN [2] discovers state constraints (invariants) of a domain automatically. It uses this information to improve the planning efficiency. It can discover, for example, some information about type constraint, like those defined in definition 8, or also extract some relation as those defined from definition 3 to definition 6. In the future, the CBS system can use DICOPLAN or TIM [1] to extract information automatically.

6. Conclusion

This paper describes a Case Base Seeding system (CBS) that can be used to construct a case base with random cases. It is suitable for empirical tests of Case-Based Planning Systems (CBP).

Seeding systems are not easily found in the literature, although many CBP systems use some random case bases in their tests. This paper just describes a complete and generic CBS system that can be applied in STRIPS-model domains.

The CBS system can be improved in the future, by incorporating more complex features and by using some automatic processes to extract states constraints and invariant automatically.

References

1. Fox, M.; Long, D. The Automatic Inference of State Invariants in TIM. *Journal of Artificial Intelligence Research*, 9 , 1998. 367-421.
2. Gerevini A.; Schubert, L. Discovering State Constraints in DISCOPLAN: Some New Results. In: Proc. of AAAI-2000. AAAI Press, 2000.
3. Hoffmann, J.; Nebel, B.. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research*.14, 2001. 253 – 302.
4. Ihrig, L.H.; Kambhampati,S. Storing and Indexing Plan Derivations through Explanation-based Analysis of Retrieval Failures. *Journal of Artificial Intelligence Research*, 1 , 1991. 1-15
5. Kettler, B. P. *Case-Based Planning with a High-Performance Parallel Memory*. 1995. PhD Thesis, University of Maryland Park, Maryland, 1995.
6. Long, D.; Fox, M. *The 3rd International Planning Competition - IPC'2002*. Available in <<http://www.dur.ac.uk/d.p.long/competition.html>>.
7. Muñoz-Avila, H. ; Weberskirch, F. Planning for Manufacturing Workpieces by Storing, Indexing and Replaying Planning Decisions. In: Proc. of AIPS-96, AAAI Press, 1996.
8. Smyth, B., McKenna, E. Building Compact Competent Case-Bases. In: Proc. of ICCBR'99. Althouff, K., Bergmann, R., Branting, K. (Eds.), Lecture Notes in Artificial Intelligence, v. 1650. Springer-Verlag,1999.
9. Smyth, B., Keane, M. Remembering to Forget: A Competence-preserving Case-deletion Policy for Case-based Reasoning Systems. In: Proc. of IJCAI'95, 14., Morgan Kaufmann, 1995.
10. Tonidandel, F.; Rillo, M. An Accurate Adaptation-Guided Similarity Metric for Case-Based Planning In: Proc. of ICCBR-2001. Aha, D., Watson, I. (Eds.), Lecture Notes in Artificial Intelligence. v.2080, Springer-Verlag, 2001.
11. Tonidandel, F.; Rillo, M. Releasing Memory Space through a Case-deletion policy with a Lower bound for Residual Competence. In: Proc. of ICCBR-2001. Aha, D., Watson, I. (Eds.), Lecture Notes in Artificial Intelligence. v.2080, Springer-Verlag, 2001.
12. Tonidandel, F.; Rillo, M. The Far-Off system: A Heuristic Search Case-Based Planning. In: Proc. of aips'02, AAAI Press, 2002.
13. Veloso, M. Planning and Learning by Analogical Reasoning. *Lecture Notes in Artificial Intelligence*, v.886. Springer-Verlag, 1994.
14. Zhu J., Yang Q. Remembering to Add: Competence-preserving Case-Addition Policies for Case-Base Maintenance. In: Proc. of IJCAI'99. M.Kaufmann, 1999.