PSVN API Manual (June 13, 2014)

Robert Holte and **Neil Burch**
Computing Science Department
University of Alberta
Edmonton, AB Canada T6G 2E8
(rholte,nburch@ualberta.ca)

Beat Hänger and **Florian Pommerening**
University of Basel
Department of Mathematics and Computer Science
Bernoullistrasse 16
CH - 4056 Basel, Switzerland
(beat.haenger,florian.pommerening@unibas.ch)

1 Introduction

This document describes the 2014 `PSVN` API, the programming interface for performing fundamental operations underlying state-space search. This version of the API differs slightly from the previous one. The changes became effective June 1, 2014, and were triggered by the collaborative effort of the authors to implement the API in two very different settings. In Holte and Burch's implementation, the functions implementing the API are produced by the `psvn2c` compiler for the `PSVN` language, are written in C, and are fully defined at the time the user's code is compiled. Pommerening and Hänger's implementation builds on the Fast Downward (Helmert 2006) code base, which loads the $\texttt{SAS}^+$ state space definition at runtime and is written in C++.

This manual is best read as a companion to the `PSVN` Manual even if the reader is not using the `PSVN` language. The `PSVN` Manual contains descriptions of the functionality this API supports, illustrative snippets of code using the API described here, and tutorials with full implementations of several standard search algorithms using the `PSVN` API.

As in the `PSVN` Manual, the word "rule" is synonymous with "operator" and "successors" is synonymous with "children".

The `PSVN` API is divided into the following components:

`CORE`: API for forward state-space search, basic input/output, and simple state manipulation.

`BWD_MOVES`: API for backward state-space search.

`FWD_MOVE_PRUNING`: API for move pruning in forward state-space search. "Move pruning" here refers to any method, based on the sequence of operators by which a state has been generated, for deciding not to permit a rule to be applied to a state even though the state satisfies the rule's preconditions. The most fully developed such method is due to Burch and Holte (2011; 2012) (Holte and Burch 2014), but implementations are not required to provide exactly that method.

`BWD_MOVE_PRUNING`: API for move pruning in backward state-space search.

`ABSTRACTION`: API for state-space abstraction.

All implementations must implement `CORE`, all other components of the `PSVN` API are optional. An implementation declares which components it has by including the appropriate `#define` statements from this list:

```
#define HAVE_BWD_MOVES
#define HAVE_FWD_MOVE_PRUNING
#define HAVE_BWD_MOVE_PRUNING
#define HAVE_ABSTRACTION
```

There is no `#define HAVE_CORE` because the `CORE` API component is not optional.

This manual has one section for each of these components. Additional `PSVN` API components are under development and will be added as they become finalized.

Implementation Variations

Ideally, an API would define exactly what each of its elements does in complete detail, but we decided to allow some precisely defined scope for variation. This was motivated by getting the most efficient possible code in the compile-time implementation (Holte and Burch) while at the same time making it possible to write a run-time implementation (Pommerening and Hänger). The variations permitted are as follows:

- All the constants in the `PSVN` API should be treated as immutable by user code even though they are described below as variables that can be assigned a value at run-time. Implementations are encouraged to use `#define` or `const` variables to define the constants, but are permitted to use run-time changeable variables for them. The constant `num_fwd_rules`, for example, is described below as

  ```
  static int num_fwd_rules
  ```

  but it could actually be implemented as

  `#define num_fwd_rules` *value*

  or

  `const static int num_fwd_rules =` *value*`;`

- Anything described as a function in the documentation below could be implemented either as a function or as a `#define` macro. `is_goal`, for example, is described below as a function that returns 1 if the state it is given is a goal state, 0 otherwise, but it could be implemented as macro with the functionality described. Because a `#define` macro can be used to implement any function, all the usual warnings about macros should be heeded by user code. For example, it is advisable to always use constants or simple variable names, not expressions, to specify the parameters passed to a function.

- An implementation is permitted to have a function called `initialize_XXX_psvn_api` that must be called prior to using anything in the `PSVN` API. `XXX` in the name identifies the implementation (see below). The arguments of this function, and its exact behaviour, may vary arbitrarily from one implementation to another, and an implementation may have no such function. It is likely that user code will have to change this one line of code when moving from one implementation to another.
- Except as specified below, there is no requirement for an implementation to check the validity of the parameter values the user's code passes to the functions in the `PSVN` API and any behaviour, including crashing unceremoniously, is permitted if invalid values are passed to the functions.
- An implementation may offer functions in addition to those specified in the `PSVN` API. A typical example of this would be a data structure whose implementation is specific to the internal details of the state data type, as opposed to being a generic implementation of the data structure. `state_map`, in the `psvn2c` implementation of the API, is an example of such a data structure.
- Implementations may vary in where the responsibility lies for allocating and freeing the memory needed for variables whose types are defined in the `PSVN` API (e.g. variables of type `state_t` or `ruleid_iterator_t`). Some implementations may do all of the memory allocation and freeing themselves, internally, while others might require the user code to do some or all of the allocation and/or freeing.
- The preceding are general rules for implementation variation. If a particular function permits some other kind of variation, it will be described with the function definition below (e.g. `next_ruleid`).

Each implementation will have thorough documentation explaining which of the permitted variations it used for all the constants, functions, etc. in the `PSVN` API.

Each implementation will also have a `#define` that uniquely identifies it. For example, the implementation in the Fast Downward system identifies itself in this way:

```
#define FASTDOWNWARD_PSVN_API
```

This allows users to write their code with `#ifdef`'s to work with all the implementations, as illustrated in the following.

```
#ifdef FASTDOWNWARD_PSVN_API
    initialize_fastdownward_psvn_api();
#elif SOMEOTHER_PSVN_API
    initialize_someother_psvn_api(x,y,z);
#endif
```

2 CORE

This section describes the `CORE` component of the `PSVN` API. The following `#include`'s are part of this component.

`#include <stdlib.h>` (needed for `size_t`)
`#include <stdio.h>` (needed for `FILE`)
`#include <stdint.h>` (needed for `uint64_t`)
`#include <limits.h>` (needed for `INT_MAX`)

`typedef state_t` Type for states.

`typedef ruleid_iterator_t` Type for the iterators that generate, one at a time, the ids of the rules that are applicable to a state. A rule's id is a non-negative integer that each implementation is free to assign as it chooses.

`static int num_fwd_rules` The number of operators for forward search. This may not be exactly the same number as in the state space definition. Some implementations might eliminate useless operators (e.g. ones that do not change a state, or that are entirely redundant with other operators), and some implementations might partially ground an operator, thereby creating several operators when initially there was only one.

`static int cost_of_cheapest_fwd_rule` Cost of the cheapest operator for forward search.

`static int cost_of_costliest_fwd_rule` Cost of the most expensive operator for forward search.

`static int max_fwd_children` Upper bound on the number of successors any state can have. Not guaranteed to be tight (i.e. there may be no state that has this number of successors).

`static int is_goal( const state_t *state_ptr )` Returns 1 if the state that `state_ptr` points at is a goal state, 0 otherwise.

`static void init_fwd_iter( ruleid_iterator_t *iter, const state_t *state_ptr )` Given a pointer to a ruleid iterator and a pointer to a state, initializes the iterator so that it will iterate through the rules whose preconditions are satisfied by the state.

`static int next_ruleid( ruleid_iterator_t *iter )` Given a pointer to a ruleid iterator, returns the id of the next rule whose preconditions are satisfied by the state associated with the given iterator. The direction (forward or backward) is determined by how the iterator was initialized (`init_fwd_iter` or `init_bwd_iter`). The order in which rules are returned can vary from implementation to implementation, it is not necessarily the order in which the rules were given in the original state space definition. Returns -1 if there are no further applicable rules.

`static void apply_fwd_rule( int ruleid, const state_t *state_ptr, state_t *result_ptr )` Applies the operator identified by `ruleid` to the state pointed at by `state_ptr`. The output parameter `result_ptr` points at the resulting state.

`static int get_fwd_rule_cost( int ruleid )` Given a rule id, returns the cost of the corresponding rule.

`static const char* get_fwd_rule_label( int ruleid )` Given a rule id, returns the label associated with the corresponding rule.

`static void copy_state( state_t *dest_ptr, const state_t *src_ptr )` A copy of the state pointed at by `src_ptr` is made. The output parameter `dest_ptr` points at the result.

`static int compare_states( const state_t *a, const state_t *b )` Returns 0 if the states pointed at by `a` and `b` are equal, non-zero otherwise.

`static uint64_t hash_state( const state_t *state_ptr )` A hash function for states. Can vary from implementation to implementation.

Input/Output of States

There are two formats for input and output of states, "raw binary" (`dump_state` and `load_state`), which is meant for rapid reading/writing of states from/to files, and "user-readable" (`print_state`, `sprint_state`, and `read_state`), which is intended to be as meaningful to the user as possible. The exact format can vary from implementation to implementation, but it must be the case that the format produced by `dump_state` can be read by `load_state` to reproduce exactly the state that was dumped, and that the format(s) produced by `print_state` and `sprint_state` can be read by `read_state` to reproduce exactly the state that was printed.

`static ssize_t print_state( FILE *file, const state_t *state_ptr )` The state pointed at by `state_ptr` is written to `file` in user-readable format. Returns the number of characters written on success, -1 on failure.

`static ssize_t sprint_state( char *string, const size_t max_len, const state_t *state_ptr )` The state pointed at by `state_ptr` is written to the `char` array `string` in user-readable format. Fails if `max_len` or more characters are required. Returns the number of characters written on success, -1 on failure.

`static ssize_t read_state( const char *string, state_t *state_ptr )` Reads a state in user-readable format from the `char` array `string`. The output parameter `state_ptr` points at the result. Returns the number of characters consumed on success, -1 on failure.

`static void dump_state( FILE *file, const state_t *state_ptr )` Writes the state pointed at by `state_ptr` to `file` in raw binary format. Returns 1 on success, 0 on failure.

`static void load_state( FILE *file, state_t *state_ptr )` Reads a state from `file` in raw binary format. The output parameter `state_ptr` points at the result. Returns 1 on success, 0 on failure.

3 BWD_MOVES

The `BWD_MOVES` component of the `PSVN` API has counterparts for all the constants, functions, etc. in the `CORE` component that have "`fwd`" in their names, namely:

```
static int num_bwd_rules
static int cost_of_cheapest_bwd_rule
static int cost_of_costliest_bwd_rule
static int max_bwd_children
```

```
static void init_bwd_iter( ruleid_iterator_t
*iter, const state_t *state_ptr )
```

```
static void apply_bwd_rule( int ruleid, const
state_t *state_ptr, state_t *result_ptr )
```

```
static int get_bwd_rule_cost( int ruleid )
static const char* get_bwd_rule_label( int
ruleid )
```

The descriptions of these are identical to their `fwd` counterparts except, of course, these apply to backwards search. Note that the same function, `next_ruleid( ruleid_iterator_t *iter )`, is used to iterate through the successors (forward direction) and predecessors (backward direction) of a state. The direction (forward or backward) is determined by how the iterator was initialized (`init_fwd_iter` or `init_bwd_iter`).

In addition, the `BWD_MOVES` component of the `PSVN` API has the functions below for enumerating all the goal states. It is assumed that the goal is specified in disjunctive normal form, i.e. as a disjunction of conjunctions. We refer to a conjunction as a `GOAL` condition. Each `GOAL` condition is associated with a unique integer identifier, which an implementation may assign any way it chooses.

`static void first_goal_state( state_t *state_ptr, int *goal_num )` This sets `goal_num` to the id of one of the `GOAL` conditions, creates a state that matches that `GOAL` condition, and returns a pointer to it in the output parameter `state_ptr`. Since there must be at least one `GOAL` condition, this function always succeeds.

`static int8_t next_goal_state( state_t *state_ptr, int *goal_num )` Creates a state that matches the `GOAL` condition identified by `goal_num` that is different than any previous state returned for this `GOAL` condition. If there are no such states for the current `GOAL` condition, `goal_num` is changed to an id for a `GOAL` condition whose states have not yet been enumerated and creates a state that matches that `GOAL` condition. In either of these cases, the function returns 1 and the output parameter `state_ptr` points at the newly created state. If all states matching all `GOAL` conditions have already been returned, the function returns 0. An implementation may produce the same state multiple times, but at most once for each `GOAL` condition the state matches.

4 `FWD_MOVE_PRUNING`, `BWD_MOVE_PRUNING`

Move pruning, whether in the forward or backward direction, depends on the path (sequence of operators) that leads from the start state to the current state. This path is summarized by an integer called the "history". Everything in the `PSVN` API for move pruning involves the history and it is vital that the user's code properly updates the history (with the function `next_fwd_history`, for forward search, and `next_bwd_history` for backward search) when an operator is applied to a state.

`static int init_history` An integer value that represents "no history", i.e. the empty operator sequence. A variable representing a history can be initialized by assigning this value to it. The same initial history is used for both forward and backward search.

The following are for move pruning during forward search.

`static int fwd_rule_valid_for_history( int history, int ruleid )` Returns 0 if move pruning analysis has determined that the rule identified by `ruleid` should not be applied after the operator sequence represented by `history`; returns a non-zero value otherwise.

`static int next_fwd_history( int history, int ruleid )` Returns the history representing the operator sequence produced by applying the rule identified by `ruleid` after the operator sequence represented by `history`.

The counterparts for these, for backward search, are:

`static int bwd_rule_valid_for_history( int history, int ruleid )`

`static int next_bwd_history( int history, int ruleid )`

5 `ABSTRACTION`

State space abstraction is described in Section 3.5 of the `PSVN` Manual. Abstractions are created, and then written to a file, using a program such as `abstractor.cpp` (see Tutorial Lesson #5). The `PSVN` API for using an abstraction once it has been created in this way is as follows.

`abstraction_t` Type for abstractions.

`static abstraction_t *read_abstraction_from_file( const char *filename )` Reads an abstraction from the given file and returns a pointer to it. The exact format for the input may vary from one implementation to another. The memory required to store the abstraction is allocated by this function. It is the user's responsibility to free that memory, by calling `destroy_abstraction`, when the abstraction is no longer needed.

`static void print_abstraction( const abstraction_t* abst )` Prints the given abstraction on `stdout`. The exact format may vary from one implementation to another, but has the property that if an abstraction is printed using `print_abstraction` it will be readable by `read_abstraction_from_file` and reproduce the abstraction that was printed.

`static void destroy_abstraction( abstraction_t* abst )` Destroys the given abstraction. Frees all associated memory.

`static void abstract_state( const abstraction_t *abst, const state_t *state_ptr, state_t* abst_state_ptr)` Creates the abstract state corresponding to the state pointed at by `state_ptr` using the abstraction pointed at by `abst`. The result is pointed at by `abst_state_ptr`.

6 Acknowledgements

Thanks to Malte Helmert and the entire Fast Downward research group for hosting the visit by Holte that spawned the collaboration on the `PSVN` API, to Switzerland's SNF for funding that visit, and to Canada's Natural Sciences and Engineering Research Council (NSERC) and iCORE (Alberta Innovates) for their funding of the `PSVN` research.

Appendix. psvn2c Implementation of the PSVN API

This appendix specifies details relevant to user code concerning how the psvn2c implementation of the 2014 PSVN API conforms to the variation that is permitted within the API definition.

The psvn2c implementation of the API is generated by the program psvn2c, which compiles state-space definitions in the PSVN language (see the PSVN Manual for a description) to C code that implements the API. This code is then included with the user's code when the latter is compiled.

The unique identifier for this implementation is

```
#define psvn2c_PSVN_API
```

The psvn2c implementation of the API has no function that must be called prior to using anything in the PSVN API (cf. the discussion of initialize_XXX_psvn_api in the API Manual).

All the constants and functions in the PSVN API are implemented with #define except the following, which are all functions exactly as described in the API Manual:

- CORE: is_goal, hash_state, and all the state input/output functions.
- BWD_MOVES: next_goal_state
- ABSTRACTION: all functions in this API component are implemented as functions.

In this implementation, none of the functions do any checking of the validity of the parameter values they are passed beyond the minimum required by the API.

The user is not responsible for allocating or freeing the memory needed for states (variables of type state_t) or ruleid iterators (variables of type ruleid_iterator_t), that is entirely taken care of by the implementation.

7 CORE

Internally, the implementation of the CORE component of the API uses the following additional #include's and #define's:

```
#include <inttypes.h>
#include <string.h>
#include <ctype.h>
#include <assert.h>
#define __STDC_FORMAT_MACROS
#define __STDC_LIMIT_MACROS
```

The order in which rules are returned by next_ruleid is determined internally by psvn2c and can bear no relation to the order in which they appeared in the PSVN file. The aim is to return the first rule that applies to a state as quickly as possible, which is beneficial for searches, such as depth-first search, which do not necessarily generate all the children of a state.

The output produced by print_state and sprint_state use the variable ordering and domain values that were specified in the PSVN file that defined the state space. There are no leading or trailing spaces, and adjacent values have one space between them. read_state reads this format, but allows leading and trailing whitespace, and any non-zero amount of whitespace between adjacent values. In the string passed to read_state, the part representing the state must be followed either by the end of string marker or by whitespace (which is consumed up to the first non-whitespace character or the end of string marker).

For all the functions that have a file as a parameter (print_state, dump_state, and load_state), the file must be opened prior to calling the function and it is the user's responsibility to close it.

8 BWD_MOVES

psvn2c's automatic calculation of operators for searching backwards is discussed in detail in the PSVN Manual, which also discusses challenges in modelling operators so that they work as intended in the backwards direction. In the psvn2c implementation an entirely separate set of rules is produced for backwards search, but the rule identifiers for backwards and forward rules cannot be distinguished from one another. The user is therefore responsible for ensuring that the functions called to apply a rule to a state, to get a rule cost or label, etc. are for the same direction (fwd or bwd) as the iterator that created the ruleid that is being passed to these functions.

In iterating through the goal states, the GOAL conditions are considered in the order in which they appeared in the PSVN description defining the state space (the first one to appear is goal_num=0). A state that matches multiple GOAL conditions is generated once for each GOAL condition it matches.

9 FWD_MOVE_PRUNING, BWD_MOVE_PRUNING

psvn2c implements full move pruning as defined by Burch and Holte (2011; 2012) (Holte and Burch 2014) using a length-lexicographic order on operator sequences. Via a command line option to psvn2c, the user specifies a "history length", H, and all operator sequences of length $H+1$ or less are evaluated to see which are redundant with other sequences of length $H+1$ or less. This creates a move pruning table that says which operators are allowed after each history. The memory needed for this table is usually negligible. Because the number of possible sequences is often exponential in H, the time required for psvn2c to do the move pruning analysis will usually be prohibitive for values of H much beyond 2. Separate values of H can be set for pruning in the forwards and backwards direction.

10 ABSTRACTION

psvn2c, and the abstractor.cpp that comes with it, currently support two kinds of state space abstraction, namely, projection and domain abstraction. These, and the format for reading/printing abstractions, is described in Section 3.5 of the PSVN Manual. For projection, `psvn2c` uses the method described in the `PSVN` Manual of not actually removing the variables that are projected out, but just rendering them irrelevant. This is done so that the same `state_t` can be used for all states.

`read_abstraction_from_file` opens the file whose name is given and closes it when it is done.

11 Additional Functions

In addition to the required functions, the `psvn2c` implementation of the `PSVN` API provides an implementation of a data structure called a `state_map`. This is a standard hash table in which the keys are states and the value associated with a state is an `int`. The API associated with `state_map` is as follows.

`typedef state_map_t` Type for a `state_map`.

`static state_map_t *new_state_map()` Creates a new `state_map` and returns a pointer to it. Allocates memory.

`static void destroy_state_map(state_map_t *map_ptr)` Destroys the given `state_map`. Frees all associated memory.

`static int *state_map_get( const state_map_t *map_ptr, const state_t *state_ptr )` Returns a pointer to the integer value associated with the given state in the given `state_map`. Returns `NULL` if the state is not in the `state_map`.

`static void state_map_add( state_map_t *map_ptr, const state_t *state_ptr, const int value )` If the given state is not in the given `state_map`, creates a new entry in the `state_map` associating the given value with the state. If the state is already in the `state_map` the new value replaces the existing one.

`static void write_state_map( FILE *file, const state_map_t *map )` Dumps the given `state_map` to the given file in binary format. The file must be opened prior to calling this function.

`static state_map_t *read_state_map( FILE *file )` Reads a `state_map` from the given file and returns a pointer to it. In the file, the `state_map` must be in the same binary format that is written by `write_state_map`. The file must be opened prior to calling this function and it is the user's responsibility to close it. The memory required to store the state_map is allocated by this function. It is the user's responsibility to free that memory, by calling `destroy_state_map`, when the state_map is no longer needed.

References

Neil Burch and Robert C. Holte. Automatic move pruning in general single-player games. In *Proceedings of the 4th Symposium on Combinatorial Search (SoCS)*, 2011.

Neil Burch and Robert C. Holte. Automatic move pruning revisted. In *Proceedings of the 5th Symposium on Combinatorial Search (SoCS)*, 2012.

Malte Helmert. The Fast Downward planning system. *Journal of Artificial Intelligence Research (JAIR)*, 26:191–246, 2006.

Robert C. Holte and Neil Burch. Automatic move pruning for single-agent search. *AI Communications*, 2014. (to appear).