PSVN Manual (June 20, 2014)

Robert C. Holte
Computing Science Department
University of Alberta
Edmonton, AB Canada T6G 2E8
(rholte@ualberta.ca)

Broderick Arneson

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

1 Introduction

This manual describes PSVN, a language for describing state space search problems using multi-valued variables that lends itself to efficient calculation of a state's successors and predecessors without having to fully ground the operators. It also describes the PSVN compiler, psvn2c, which accepts PSVN as input and compiles it into efficient C code that can then be incorporated into the search or planning code of one's choice. Finally, it includes a set of tutorial lessons to help you learn how to use PSVN.

To researchers or practitioners who are focused on a specific search domain, PSVN will be useful as a rapid prototyping tool, since the code psvn2c generates is efficient and its move pruning analysis automatically detects and removes redundant operator sequences that are typical of the domain-specific optimizations done when writing code by hand.

To researchers or practitioners interested in domain-independent search or planning, PSVN offers an attractive middle ground between first-order languages for specifying state spaces, such as PDDL (Ghallab *et al.* 1998), and fully grounded languages such as FDR (Helmert 2009) and SAS$^+$ (Bäckström and Nebel 1995). PSVN is very similar to the latter, but has a simple, efficient mechanism enabling the value of a state variable to be referenced and copied without grounding. By avoiding the exponential blowup that occurs in some search domains when their operators are grounded, PSVN extends the set of state spaces that can be handled in practice and adds to the efficiency of reasoning about a state space by reducing the number of operators and precondition tests.

PSVN is also designed to fully support the following commonly needed functions:

- backward search (predecessor calculation); psvn2c automatically infers inverses of the given operators;
- state space abstraction (in particular, "projection" and "domain abstraction").

2 The PSVN Language

PSVN is a slight extension of the language with the same name introduced by Hernádvölgyi and Holte (1999). One overarching principle has guided the design of the language: efficiency of execution. Features are not added to the language to make it more convenient for humans to use or read; they are only added if they will result in efficiency gains. PSVN is therefore more akin to an assembly language than a high-level programming language, and one can imagine an entirely separate language specifically designed for ease of modelling that is then compiled into PSVN. Expressiveness of the language has not been a primary concern up to this point. Our aim has been to make an efficient language expressive enough that the standard testbed problems could be compactly encoded in it.

2.1 States and Variable Domains

In PSVN a state is a vector of fixed length, n. The entry in position i is drawn from a finite set of possible values called its domain, D_i. In many state spaces every position of the vector has the same domain, but in principle they could all be different. Different domains in PSVN are entirely distinct; the same symbols can appear in different domains, but they will internally be considered distinct.

Figure 1 illustrates how domains are defined and assigned to each position of the state vector. The keyword DOMAIN is followed by a user-given name for the domain, the number of values in the domain, and then the values in the domain, which can be any legal tokens. The domain definitions are followed by the length of the state vector (n = 7 in this example). Then the domain for each position in the vector is declared (the line size gender size height 3 2N 4). In addition to user-defined domains, there are two kinds of built-

```
DOMAIN gender 2  FEMALE MALE
DOMAIN size   5  SMALL MEDIUM LARGE XL XXL
DOMAIN height 3  SHORT MEDIUM TALL

7

size gender size height 3 2N 4
```

Figure 1: Domain Definitions and Declarations.

in numerical domains. A number by itself denotes the domain containing that many integer values starting at 0 (e.g. `3` denotes the domain $\{0,1,2\}$), a number followed by `N` denotes the domain containing that many integer values starting at 1 (e.g. `2N` denotes the domain $\{1,2\}$). The token `MEDIUM` occurs in two of the user-defined domains in this example, but the two occurrences are treated by `PSVN` as entirely distinct values because they are in different domains. Likewise, the domains `3`, `2N`, and `4` all have some numbers in common but, because the domains are distinct, the numbers in one domain are considered to be different than the numbers in the other two domains.

The `PSVN` language is entirely insensitive to case, so the keyword `DOMAIN` could also be entered as `domain`, `Domain`, etc., and constants such as `MEDIUM`, `medium`, and `MeDiUm` are all indistinguishable from each other.

2.2 Operators

The transitions in the state space are specified by a set of operators (also called rules). Each operator has a left-hand side (LHS) specifying the operator's preconditions and a right-hand side (RHS) specifying the operator's effects. The LHS and RHS are each a vector of length n. In both the LHS and the RHS, position i is either a constant from D_i or a variable symbol. Any number of positions in these vectors (LHS and RHS) can contain the same variable symbol providing their domains are all the same.

State $s = \langle s_1 ... s_n \rangle$ matches $LHS = \langle L_1 ... L_n \rangle$ if and only if $s_i{=}L_i$ for every L_i that is a constant and $s_i{=}s_j$ for every i and j such that L_i and L_j are the same variable symbol.

An operator is "deterministic" if every variable symbol in its RHS is also in its LHS. The effect of a deterministic operator when it is applied to state $s = \langle s_1 ... s_n \rangle$ matching its LHS is to create a state $s' = \langle s'_1 ... s'_n \rangle$ such that: (i) if position j of the RHS is the constant $c \in D_j$ then $s'_j{=}c$; (ii) if position j of the RHS is the variable symbol that occurs in the position i of the LHS then $s'_j{=}s_i$.

An operator is "non-deterministic" if one or more of the variable symbols in its RHS do not occur in its LHS. We call such variable symbols "unbound". The effect of a non-deterministic operator when it is applied to state $s = \langle s_1 ... s_n \rangle$ matching its LHS is to create a set of successor states. There is one successor for every possible combination of values of the unbound variables (if the unbound variable is in position i, its values are drawn from D_i). Each of the other positions of these successors will be the same in all the successors and are determined by the rules for calculating the effects of deterministic operators. For example, if $n{=}4$ and all positions have domain $\{1,2\}$ then the rule[1]

```
1 A B C  =>  E 1 D E
```

would create four successors when applied to state[2] $\langle 1,2,1,2 \rangle$, namely, $\langle 1,1,1,1 \rangle$, $\langle 2,1,1,2 \rangle$, $\langle 1,1,2,1 \rangle$, and $\langle 2,1,2,2 \rangle$.

Optionally, an operator may be given a label or a cost or both. In the current version, if both are given the label must be given first. These are specified by adding after the RHS the keyword `LABEL` followed by a token or the keyword `COST` followed by a non-negative integer. Zero-cost operators are allowed. The default cost of an operator is 1. For example, specifying the label `example` and a cost of 7 for the rule above would be done as follows:

```
1 A B C  =>  E 1 D E  LABEL example COST 7
```

The same label can be given to multiple rules. For example, in the sliding-tile puzzle one could give the label `UP` to all operators that exchange a tile with the blank by moving the blank up. In the absence of user-assigned labels, the operators are labelled `rule_1`, `rule_2`, etc.

2.3 Special Symbols

Comments can be included anywhere in a `PSVN` description using the symbols `#` or `;`. On an input line containing one of these symbols everything after the symbol is ignored.

As a concession to human readability `PSVN` allows one additional symbol, the dash ("-"), to be used in any position in an LHS or RHS. The aim is to allow positions that "don't matter" to be marked with a special symbol so that the constants and variable symbols that do occur in an operator are ones that "matter". More precisely, a dash in position i of the LHS means that the value in position i is not tested in a precondition in the LHS or assigned to a different position in the RHS, and a dash in position i of the RHS means that the value in position i does not change when the operator is applied. The dash does not add any expressiveness to the `PSVN` language since its meaning can be defined in terms of the other elements of the language.

As an example, consider the 4-peg Towers of Hanoi problem. There are many ways to encode this state space in `PSVN`, here we shall consider one of the most compact. For each disk there will be four binary state variables indicating which peg the disk is on (if disk d is on peg i, the i^{th} state variable for d will be 1 and the others will be 0). For each disk d and for each pair of pegs i and $j \neq i$ one `PSVN` operator is needed for moving d from the top of peg i to the top of peg j. For example, if there are just three disks, the operator for moving the largest disk from peg 2 to peg 3 is:

```
      - 0 0 -  - 0 0 -  - 1 - -
=>    - - - -  - - - -  - 0 1 -
```

The `0`s among the first eight positions of the LHS encode the requirement that the smaller disks not be on pegs 2 and 3, so they are not blocking this move. The `1`

[1] Unless otherwise stated, in all rules in this manual domain values are integers and variable symbols are upper case letters.

[2] The angle brackets around vectors and commas separating the vector positions are not part of `PSVN`, they are in the text for readability.

among the last four positions in the LHS tests that the largest disk is indeed on peg 2. The RHS only changes the state variables indicating which peg the largest disk is on.

This operator works perfectly well in the forward direction (for successor generation) but, for reasons that will be discussed in detail below (Section 3.2), it does not work as intended when used in the backward direction (for predecessor generation). This is not a bug in PSVN, it is inappropriate modelling by the person who wrote the rule if the intent was to use the rule in both directions. The appropriate way to write the rule so it behaves as intended in both directions is:[3]

```
      - 0 0 -  - 0 0 -  - 1 0 -
=>    - 0 0 -  - 0 0 -  - 0 1 -
```

From an efficiency point of view, this way of writing the rule is less than ideal because the 0 in the last four positions of the LHS will have to be tested when, in fact, it is guaranteed to be 0 in a legitimate Towers of Hanoi state given that the 1 occurs as shown. The extra testing is a waste of time. PSVN allows the user to indicate that a constant need not be tested by putting an asterisk in front of it. The most efficient encoding of this Towers of Hanoi operator that works properly in both directions uses this as follows:

```
      - 0 0 -  - 0 0 -  - 1  *0 -
=>    - 0 0 -  - 0 0 -  - *0 1  -
```

Asterisks in the RHS have no effect when the rule is used in the forward direction, but avoid needless testing when the rule is used in the backwards direction, since the RHS specifies the preconditions of the backwards application of a rule (see Section 3.2).

An asterisk can also be used when a state variable is involved in an equality test. For example, consider the following LHS, which tests if the first three state variables all have the same value:

```
X  X  X ...  =>  ...
```

psvn2c would generate two equality tests to test this precondition: the first and second state variables would be compared and, if they were equal, the first and third state variables would be compared. If, however, you knew that the first state variable had to be the same as the other two if they were equal, you could use this knowledge to write the LHS in a more efficient form:

```
*X  X  X ...  =>  ...
```

For this LHS psvn2c would generate only one equality test: only the second and third state variables would be compared. If the rule was applied backwards all three variables still would be guaranteed to be assigned the same value. The general meaning of the asterisk on a particular state variable in the LHS (or the RHS if the rule is being used backwards) is that it removes that state variable from any involvement in precondition testing. The rule

```
*X *X  X ...  =>  ...
```

therefore does no testing, since there is no variable without an asterisk for the third state variable to be compared to. It is equivalent to:

```
*X *X *X ...  =>  ...
```

[3]In this rule it is not necessary to specify a 0 for pegs 1 and 4 of the largest disk because these positions "don't matter" at all, there is a dash for them in both the LHS and the RHS.

```
# 4-Pancake Puzzle
4              # 4 vector positions
4 4 4 4    # all domains = {0,1,2,3}
A B - -  =>  B A - -  LABEL reverse2
A - C -  =>  C - A -  LABEL reverse3
A B C D  =>  D C B A  LABEL reverse4
GOAL 0 1 2 3
```

Figure 2: Complete PSVN for the 4-Pancake Puzzle.

2.4 Goal Conditions

The final feature of PSVN is that the goal of a search is not required to be a single state. Goal conditions are specified by writing one or more special operators of the form "GOAL Condition", where Condition takes the same form as the LHS of a normal operator. An example is shown in the last line of Figure 2. If there are several such goal statements they are interpreted disjunctively. GOAL statements can be placed before or after the operators, or even intermingled with them.

2.5 Discussion

Although the PSVN language is extremely simple, it gains enormous power from the way variables in its rules get bound to values in the state to which the rule is being applied and then copied into arbitrary locations in the successor state. This is a very efficient operation that allows replicating and/or swapping of values in a state without having to enumerate all the different ways the rule could be grounded. Many planning and search problems involve shifting objects among a set of locations (e.g. driving a truck from one location to another). This power is most evident in pure "permutation" puzzles such as the Pancake Puzzle (Dweighter 1975). In the k-Pancake Puzzle, a state is a permutation of the numbers 1 to k and has $k-1$ successors, with the l^{th} successor formed by reversing the order of the first $l+1$ positions of the permutation ($1 \leq l \leq k-1$). In PSVN these rules are trivial to encode: the complete PSVN for the 4-Pancake Puzzle is shown in Figure 2.

3 psvn2c, the PSVN Compiler

The primary purpose of psvn2c, the PSVN compiler, is to read a PSVN description and produce a corresponding set of functions and type definitions, in the C programming language that can be used in a user's search or planning code. These include functions for testing if a state satisfies the goal conditions and for

```
state_t state, child;
ruleid_iterator_t iter;
int move_cost, ruleid;
...
init_fwd_iter( &iter, &state );
while((ruleid=next_ruleid( &iter )) >= 0)
{
   apply_fwd_rule( ruleid,&state,&child );
   move_cost = get_fwd_rule_cost( ruleid );
   if( is_goal( &child ) )
    ...
}
```

Figure 3: C code for iterating through a state's successors.

generating the successors of a state one by one. Figure 3 shows a code fragment for iterating through the successors of the state stored in variable state and testing if each is a goal state. The code also shows how to get the cost of the rule used to generate each child. state_t and ruleid_iterator_t are types defined by psvn2c for states and iterators, respectively, and init_fwd_iter, next_ruleid, apply_fwd_rule, get_fwd_rule_cost, and is_goal are functions psvn2c has defined. Note that applying a rule to a state creates an entirely new state (child in the figure), it does not modify the given state. Also note that the order in which successors are generated is determined internally by psvn2c and is not necessarily the order in which the operators are given in the PSVN description.

The type state_t is the internal representation of the state that is used by most PSVN functions. It is not advisable for user code to make any assumptions about the internal details of state_t since they might not always be the same. To convert back and forth between the internal representation and a string representation that is human readable and easy for user code to manipulate, psvn2c provides three functions, sprint_state, print_state, and read_state, whose use is illustrated in Figure 4. Note that sprint_state takes a maximum length parameter (256 in the example) and sprint_state will fail if the string representation exceeds this limit.

```
state_t state;
char string[256];
ssize_t len;
...
/* convert a state to a string*/
len = sprint_state(string,256,&state);
...
/* convert a string to a state */
len = read_state(string,&state);
...
```

Figure 4: C code for converting a state between internal and string representations.

The string representation of a state contains the domain values separated by white space. For example, the string representation of a typical state based on the definitions in Figure 1 would be

"SMALL MALE XXL TALL 0 2 3".

3.1 How to Run the psvn2c Compiler

In this section, we briefly describe how to actually run psvn2c on a PSVN source file and include the C code that it generates when the user's search code is compiled.

psvn2c was developed and tested on Linux using gcc and g++. The code requires only the STL libraries, so it should be easily portable to other systems.

Assume there is a file called pancake4.psvn containing the PSVN shown in Figure 2. If it is in the same directory as the psvn2c executable (called psvn2c) then the first step is to apply psvn2c to the PSVN source file as follows:

```
./psvn2c < pancake4.psvn > pancake4.c
```

The resulting file (pancake4.c) contains C language definitions of the types, constants, and functions that psvn2c produces. A full list of these is given in the PSVN API Manual. The user's search/planning code should be written in C or C++ using these, as illustrated by the code fragments throughout this document (e.g. Figure 3). The integration of the psvn2c-generated code with the user's code happens when the user's code is compiled. If the user's code, mysearch.cpp, is in C++, for example, it should be compiled as follows:

```
g++ mysearch.cpp -include pancake4.c -o
        mysearch.pancake4
```

This manual is accompanied by a series of tutorial lessons to help you learn the practical aspects of using the PSVN system. The lessons can be found at the end of this manual, and the code and other files for each of them is found in a directory called LessonNN (where NN is the lesson number, e.g. Lesson01 for Lesson #1). Lesson #1 is an introduction to writing your own PSVN state space definitions, running the psvn2c compiler, and integrating the code it creates with the search code you have written.

> You are now ready to do Tutorial Lessons #1, #1A, and #2.

3.2 Applying Operators Backwards

Figure 3 and the related text described the "forward" application of the operators, whereby the successors of a state are computed by testing if the state matches each operator's LHS and, if it does, using the RHS to determine the successor state(s). Techniques such as regression planning (Rintanen 2008), bidirectional search (Kaindl and Kainz 1997), and pattern databases (Culberson and Schaeffer 1998) require computing the predecessors of a state s, i.e., the set $P(s)$ containing all and only the states p such that $r(p) = s$

for some rule r that can be applied to be p. Fortunately, it is very easy to generate the backwards rule for a given PSVN rule.

Consider a rule $r = \langle t_1, ..., t_n \rangle \rightarrow \langle a_1, ..., a_n \rangle$. The backwards rule $b = \langle tb_1, ..., tb_n \rangle \rightarrow \langle ab_1, ..., ab_n \rangle$ can be constructed one test/action pair at a time. There are three cases. First, if neither t_i nor a_i is a dash $(-)$, then we can just switch their roles: $tb_i = a_i$ and $ab_i = t_i$. If a_i is a dash, then $tb_i = t_i$ and $ab_i = a_i$. Finally, if a_i is a variable or constant and t_i is a dash, then $tb_i = a_i$ and ab_i is assigned a variable name which does not occur anywhere else in the backwards rule. For example, $\langle -, X, 0, -, -, X, 2 \rangle \rightarrow \langle -, -, -, 1, 1, 3, X \rangle$ becomes $\langle -, X, 0, 1, 1, 3, X \rangle \rightarrow \langle -, -, -, T1, T2, X, 2 \rangle$.

A special circumstance to note is when a rule, applied in the forward direction, loses information about one or more of the values of the state to which it was applied. This will happen when t_i is a variable symbol that does not occur in the rule's RHS or, equivalently, if t_i is a dash and a_i is not. When such a rule is applied to a state s, s_i can be any value, but its value is lost, overwritten by a_i. That is, we have a rule which maps many different states to the same state. We must satisfy $s'_i = a_i$, so going backwards we have $tb_i = a_i$. Since there is no way to recover the specific value of s_i, we must generate all possible values for it, so we make ab_i an unbound variable.

This can produce some unexpected results. If the state space is encoded in a way where only some of the possible state vectors are considered to be valid, and the rules make implicit use of this, the backwards rules may generate invalid states from valid states. For example, most encodings of permutation puzzles will have n variables with a domain of size n. There are $n!$ valid states out of n^n possible states. A rule like $\langle 1, 2, - \rangle \rightarrow \langle 3, 1, 2 \rangle$ is quite reasonable. There is no reason to test that the third variable is 3, because we know it must be 3 to be a valid state. Applying this rule backwards we get $\langle 1, 2, 3 \rangle$, but we also get the undesirable states $\langle 1, 2, 1 \rangle$ and $\langle 1, 2, 2 \rangle$, because there is nothing explicitly enforcing the constraints of a valid state. It is for this reason that the first version of the Towers of Hanoi rule given as an example in Section 2.3 does not work correctly when used in the backward direction.

The semantics of applying an operator backwards is such that if the operator is applicable to state s in the forward direction and doing so produces state s' (not necessarily deterministically), then the operator, when used in the backwards mode, is guaranteed to be applicable to state s' and is guaranteed to produce state s when it is applied to s' (again, not necessarily deterministically). Note that an operator that is deterministic in one direction might be non-deterministic in the other.

A code fragment illustrating how to generate all the predecessors of a state is given in Figure 5. It is very similar to the code for generating the successors of a state (Figure 3), except that `init_fwd_iter` and `apply_fwd_ruleid` have been replaced by `init_bwd_iter` and `apply_bwd_ruleid`. Note

```
state_t state, child;
ruleid_iterator_t iter;
int ruleid;
...
init_bwd_iter( &iter, &state );
while( (ruleid=next_ruleid( &iter )) >= 0)
{
   apply_bwd_rule(ruleid,&state,&child);
   ...
}
```

Figure 5: C code for iterating through a state's predecessors.

that `next_ruleid` has not changed, since the direction (forward, backward) of the iterator is fixed when it is initialized.

It is optional whether `psvn2c` generates backwards rules or not. The command line option `--no_backwards_moves` specifies that they should not be generated, the command line option `--backwards_moves` specifies that they should. A full list of the command line options for `psvn2c` is given in the Appendix.

3.3 Searching Backwards from the Goal

As mentioned in Section 2.4, in PSVN the goal is not necessarily a single, fully specified state, it is one or more goal conditions that may specify a large set of states. Most searches that occur backwards start at the goal. Ideally, this would be done by reasoning "symbolically" about the goal conditions without grounding them. This is possible in PSVN but the current implementation does not implement it. Instead, it provides a mechanism for iterating through all the states that match the goal conditions one at a time. The code for a loop that generates all the states that match the goal conditions is shown in Figure 6.

In Figure 6, `goal_num` stores an id for the current goal condition, starting at 0 and incrementing over goal conditions in the same order as they appear in the PSVN description. When `next_goal_state` has enumerated all states for the current condition, `goal_num` is automatically incremented and `next_goal_state` will then enumerate states for the next condition, and so on. This is all handled internally, the user need only copy the code

```
state_t state;
int goal_num;  /* does not need to be initialized */
    ...
first_goal_state( &state, &goal_num );
do {
    ...
   } while( next_goal_state( &state, &goal_num ) );
...
```

Figure 6: C code for iterating through all the states that match the goal conditions.

in the example to obtain this functionality.

Note that the code generated by `psvn2c` generates all the states that match each goal condition, so if a state matches more than one goal condition it will be generate more than once (exactly once for each goal condition it matches). If you want to eliminate duplicates of this kind, you will have to write that code yourself.

> You are now ready to do Tutorial Lessons #3 and #4.

3.4 Move Pruning

In most state spaces, there are multiple paths leading from one state to another, and any effort beyond exploring one of the cheapest such paths is wasted. Taylor and Korf (1993) introduced a method for identifying redundant move sequences, in a preprocessing step.

We have generalized their method (Burch and Holte 2011; 2012; Holte 2013; Holte and Burch 2014) to work on any `PSVN` state space definition. As it is compiling your `PSVN` file, `psvn2c` can perform this analysis and create a function you can use to avoid executing redundant operator sequences (this is illustrated in Figure 7 and discussed below). The basic idea behind `psvn2c`'s analysis is that it builds a complete search tree of a user-specified (small) depth from a generic state, and checks for duplicate states within this search tree. If a sequence of moves from a generic state is a transposition, then this sequence of moves will also be a transposition for any real state, and the final move of the sequence should be pruned. Because there are often many rules in a `PSVN` state space description, the analysis should be limited to short sequences of moves, but this is sufficient to do eliminate the costliest cycles and transpositions.

Table 1, from (Burch and Holte 2012; Holte and Burch 2014), demonstrates the power of move pruning in a range of state spaces, including several in which the operators have complex preconditions. For each state space, we compared the number of nodes generated, and execution time, of three variations of depth-first search (DFS) with a depth bound: (1) DFS with parent pruning (length 2 cycle detection), (2) DFS with our move pruning method applied to sequences of length $L = 2$ or less, and (3) DFS with our move pruning method applied to sequences of length $L = 3$ or less. All experiments were run on a machine with a 2.83GHz Core2 Q9550 CPU and 8GB of RAM). Node counts (in thousands of nodes) and computation times (in seconds) are totals (not averages) across 100 randomly generated start states.

Move pruning is an optional feature of `psvn2c`. The command line option `--history_len=`H specifies that move pruning analysis should be performed on all sequences of length $L = H + 1$ or less. `--history_len=0` effectively turns move pruning off, since it will only look fors redundancy among the operators themselves (such redundancy is not uncommon in `PSVN` files created automatically by the abstraction methods described in Section 3.5 below). If backwards rules are being generated by `psvn2c`, move pruning for the backwards rules can be specified separately from move pruning for the forward rules by using command line options `--fwd_history_len=`H_f and `--bwd_history_len=`H_b instead of `--history_len=`H. With these options, move pruning analysis for the forward rules will be applied to sequences of length $H_f + 1$ or less, and for the backwards rules to sequences of length $H_b + 1$ or less.

Figure 7 shows code for generating the children of a state when move pruning is being used. The function `fwd_rule_valid_for_history` returns 0 if the move pruning analysis done by `psvn2c` determines that `ruleid` should not be applied given the preceding sequence of moves (which is what the variable `hist` records). The `PSVN` API Manual gives a full description of the functions supporting move pruning.

State Space	d	DFS+PP	DFS+MP L=2	DFS+MP L=3
16-Arrow puzzle	15	? >3600s	3,277 0.39s 0.07s	3,277 0.39s 18.58s
10 Blocks World	11	352,028 25.02s	352,028 12.23s 5.97s	352,028 12.53s 8m 27s
8-puzzle	25	368,357 24.77s	368,357 10.40s 0.01s	368,357 10.40s 0.08s
Pancake puzzle	9	5,380,481 246.49s	5,380,481 115.22s 0.01s	5,288,231 111.18s 0.02s
Towers of Hanoi	10	1,422,419 97.02s	31,673 1.45s 0.30s	9,060 0.49s 3m 43s
2x2x2 Rubik's Cube	6	2,715,477 132.74s	833,111 20.00s 0.02s	515,614 13.35s 1.10s
TopSpin	9	? >3600s	2,165,977 73.80s 0.00s	316,437 12.59s 1.11s
Work or Golf	13	? >3600s	209,501 16.44s 2.98s	58,712 5.14s 15m 4s
Gripper	14	9,794,961 544.85s	590,870 17.22s 0.08s	25,982 0.95s 0.85s

Table 1: The first two columns indicate the state space and the depth bound used. The other columns give results for each DFS variation. In each results cell the top number is the total number of nodes generated, in thousands, to solve all the test problems. The number below that is the total time, in seconds, to solve all the test problems. In the DFS+MP columns the bottom number is the time ("m" for minutes, "s" for seconds) needed for the move pruning analysis.

```
int hist, child_hist;
hist = init_history;
...
init_fwd_iter( &iter, &state );
while((ruleid=next_ruleid( &iter )) >= 0)
{
  if (!fwd_rule_valid_for_history(hist,ruleid))
     continue;
  child_hist = next_fwd_history(hist,ruleid);
  apply_fwd_rule(ruleid,&state,&child);
  ...
}
```

Figure 7: `C` code for iterating through a state's successors with move pruning. The history variable passed to `fwd_rule_valid_for_history` (`hist` in this example) must always have a valid history value. This can be done by using `init_history` to initialize it, as shown.

It is important to note that move pruning is not, in general, safe to use with any form of transposition table (Akagi *et al.* 2010), not even something as simple as the duplicate detection in A*. Interactions between move pruning and these other methods creates the risk of eliminating all possible optimal solutions to a problem (Burch and Holte 2012; Holte 2013; Holte and Burch 2014).

3.5 State Space Abstraction

State space abstraction is a process by which the definition of a state space is changed to create the definition of a state space that is (ideally) smaller. There are numerous kinds of state space abstractions but at present `PSVN` supports just two, namely, projection and domain abstraction.

In projection, the length of the state vector is effectively reduced by eliminating from consideration a specified subset of vector positions. The domains of the remaining vector positions are unchanged.

The way projection is currently implemented, the variables that are projected away are not actually eliminated from the state vector. Instead, they remain in memory and the length of the state vector is unchanged, but the operators are changed so no operator ever reads or writes to these vector positions, and the goal conditions are changed to ignore these vector positions. In addition, their domain is changed to be `1`, which means their value cannot change, it must always be `0`.

For example, Figure 8 shows the `PSVN` representing the 4-pancake puzzle when the last two vector positions have been projected away. This definition is created by taking the definition of the 4-pancake puzzle in Figure 2 and effectively removing the last two positions from the LHS and RHS of every rule and from the goal condition. Note that there are still 4 distinct constants (representing the 4 different pancakes) in the domain of the unprojected vector positions. Note also that some of the rules have become nondeterministic (in the last two rules there are variables in the RHS that do not occur in the rule's LHS).

```
# 4-Pancake Puzzle
4             # 4 vector positions
4 4 1 1
A B - - =>  B A - - LABEL reverse2
A - - - =>  C - - - LABEL reverse3
A B - - =>  D C - - LABEL reverse4
GOAL 0 1 - -
```

Figure 8: `PSVN` for the 4-Pancake Puzzle with the last two vector positions projected away.

```
# 4-Pancake Puzzle
4             # 4 vector positions
4 4 4 4
A B - -  =>  B A - -  LABEL reverse2
A - C -  =>  C - A -  LABEL reverse3
A B C D  =>  D C B A  LABEL reverse4
GOAL 0 0 0 3
```

Figure 9: Resulting `PSVN` when the domain abstraction in the text is applied to the 4-Pancake Puzzle.

Domain abstraction allows each element of a domain to be mapped to a different element of that domain (note that the current implementation does not allow elements to map to different domains). Generally, several elements are made to map to a single element, effectively reducing the size of the domain. For example, an abstraction of the domain in the 4-pancake puzzle might map constants `0`, `1` and `2` to `0` and map `3` to itself. This abstraction would be applied to states to create the corresponding abstract states, and to the `PSVN` operators and goal conditions to create their abstract counterparts. The resulting `PSVN` is shown in Figure 9. Of course, projection and domain abstraction can be used together to create an abstract space in which the state vectors are effectively shorter and one or more of the domains has been abstracted.

Figures 8 and 9 show the `PSVN` encoding of the resulting abstracted spaces after the abstraction has been performed, but the abstraction itself needs to be encoded somehow so that `PSVN` knows how to map states from the original state space to the abstract state space at runtime. The way an abstraction is represented is shown in Figure 10. The format consists of an `abstraction` block enclosed with curly braces. This block contains a list of abstraction elements, which can

```
abstraction {
  4 { 0 0 0 3 }
  projection { K K P P }
}
```

Figure 10: Abstraction encoding both example abstractions on the 4-pancake puzzle.

be either projection abstractions or domain abstractions. A projection abstraction is the token `projection` followed by a block containing a letter for each variable, where `K` means keep the variable and `P` means project the variable away. A projection of the form "`K K P P`" as in Figure 10 will keep the first two variables and project away the last two.

A domain abstraction is the domain name followed by a value mapping. The value mapping is a block of domain values, one for each element of the domain, the value of which is the name of the element it will map to. A domain abstraction of "`4 { 0 0 0 3 }`", as in Figure 10, specifies a mapping for domain `4` (the standard zero-based four element domain and the only domain in the 4-pancake puzzle). The mapping "`0 0 0 3`" means `0` maps to `0`, `1` maps to `0`, `2` maps to `0`, and `3` maps to `3`.

The `PSVN` software suite provides a tool (`abstractor.cpp`) to interactively create the abstraction and `PSVN` definition files for an abstract space from a given `PSVN` definition. The `PSVN` definition of the abstracted space can then be compiled using `psvn2c` and integrated with the user's search code (the API for manipulating abstractions is given in the `PSVN` API Manual). For example, a pattern database (PDB) would be built and used during search by executing the following six steps:

1. Given the `PSVN` definition for a state space, `abstractor.cpp` is used to create an abstraction file and the `PSVN` definition of the abstract space;
2. The `PSVN` for the abstract space is compiled by `psvn2c`;
3. The user's search code for enumerating the abstract space and recording distances to the goal (e.g. `distSummary.c` from Lesson #3) is compiled and run on the abstract space to create the PDB;
4. The `PSVN` for the original space is compiled by `psvn2c`;
5. The user's heuristic search code is compiled together with the `C` code for the original state space created by `psvn2c` in the preceding step. In the user's code, states are abstracted to the abstract space using the abstraction description, and the value for the abstracted state is looked up in the PDB and used as a heuristic;
6. The user's heuristic search code, compiled in the previous step, is executed on one or more start states.

This sequence of steps is implemented in the `Makefile` associated with Tutorial Lesson #5.

IMPORTANT: Be sure move pruning is not used when the abstract space is being enumerated (step 3).

You are now ready for Tutorial Lesson #5.

4 Acknowledgements

Thanks to Rong Zhou for alpha testing early versions of the `PSVN` compiler, to Shahab Jabbari Arfaee, Zhaoxing Bu, Levi Lelis, Fan Xie, and Dylan Cassidy for their contributions to the `PSVN` software suite, and to Alberta's Informatics Circle of Research Excellence (iCORE), the Alberta Innovates Centre for Machine Learning (AICML), and Canada's Natural Sciences and Engineering Research Council (NSERC) for their research funding.

Appendix. Command Line Options for `psvn2c`

`psvn2c` reads the `PSVN` definition from standard input and outputs the `C` code for the `PSVN` API to stdout.

`--help` or `-h` Print out the command line options.

`--backwards_moves` Create the backwards rules and generate the API code for using them.

`--no_backwards_moves` Don't create the backwards rules or generate the API code for using them.

`--history_len=len` Use move histories of length `len` for pruning forward and backwards moves. Note that `--history_len=len` is equivalent to having `--fwd_history_len=len` and `--bwd_history_len=len`. If `len` is 0, no move pruning is done.

`--fwd_history_len=len` Use move histories of length `len` for pruning forward moves.

`--bwd_history_len=len` Use move histories of length `len` for pruning backwards moves.

`--rule_group_size=num` In an attempt to be as efficient as possible in determining (during search) which operators are applicable to a given state, `psvn2c` reasons about all the preconditions that occur in groups of rules, not about individual rules. If two rules in the same group have exactly the same preconditions, for example, the preconditions will just be tested once to determine whether or not both rules apply. `num` determines the size of the rule groups that `psvn2c` analyzes. If `num` is 0, all the rules in the `PSVN` definition will be considered as a single group. This maximizes the efficiency of precondition testing during search, but can make `psvn2c`'s analysis very time-consuming. If `num` is 1, rules will be considered individually. Larger values of `num` specify the size of the rule groups to be used. `psvn2c` will automatically decrease the size of the rule groups if too much time is being taken for this analysis.

`--keep_duplicate_rules` Keep rules with identical preconditions and effects.

`--remove_duplicate_rules` If two or more rules have identical preconditions and effects, only one copy is kept. Testing this is quadratic in the number of rules, so is only recommended if you are fairly sure there are identical copies of the same rule in the `PSVN` definition (e.g. because it was generated by projecting out many variables).

`--abstraction` Generate the code needed for state space abstraction.

`--no_abstraction` Don't generate the code for state space abstraction.

`--state_map` Generate the code for the state_map data structure.

`--no_state_map` Don't generate the state_map code.

`--verbosity=val` Set the verbosity level to `val`. `val`=0 suppresses all of the step-by-step output about `psvn2c`'s processing of a `PSVN` file. `val`=1 would be used if `psvn2c` is slow and you want to know which parts of `psvn2c`'s processing are responsible (e.g. so that you could change command line parameters accordingly). Higher values are not intended for users (unless they are curious about `psvn2c`'s inner workings), they are for debugging the `psvn2c` code.

References

Yuima Akagi, Akihiro Kishimoto, and Alex Fukunaga. On transposition tables for single-agent search and planning: Summary of results. In *Proceeding of the Third Annual Symposium on Combinatorial Search (SOCS-10)*, 2010.

Christer Bäckström and Bernhard Nebel. Complexity results for SAS+ planning. *Computational Intelligence*, 11:625–656, 1995.

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.

Joseph Culberson and Jonathan Schaeffer. Pattern databases. *Computational Intelligence*, 14(3):318–334, 1998.

Harry Dweighter. Problem E2569. *American Mathematical Monthly*, 82:1010, 1975.

Malik Ghallab, Craig K. Isi, Scott Penberthy, David E. Smith, Ying Sun, and Daniel Weld. PDDL - the planning domain definition language. Technical report, CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, 1998.

Malte Helmert. Concise finite-domain representations for PDDL planning tasks. *Artificial Intelligence*, 173:503–535, 2009.

István Hernádvölgyi and Robert Holte. PSVN: A vector representation for production systems. Technical Report TR-99-04, Department of Computer Science, University of Ottawa, 1999.

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

Robert C. Holte. Move pruning and duplicate detection. In *Proceedings of the 26th Canadian Conference on Artificial Intelligence*, pages 40–51, 2013.

Hermann Kaindl and Gerhard Kainz. Bidirectional heuristic search reconsidered. *Journal of Artificial Intelligence Research*, 7:283–317, 1997.

Jussi Rintanen. Regression for classical and nondeterministic planning. In *Proceeding of the 18th European Conference on Artificial Intelligence*, pages 568–572. IOS Press, 2008.

Larry A. Taylor and Richard E. Korf. Pruning duplicate nodes in depth-first search. In *AAAI*, pages 756–761, 1993.

PSVN Tutorial Lesson #1

Abstract

This lesson is to get you familiar with the PSVN language, some of the basic elements of the PSVN API, and how to compile and execute psvn2c.

Before Starting this Lesson

1. Read up to the end of Section 3.1 in the PSVN Manual.
2. Login to a Unix system, change to the directory where you placed the PSVN home directory.
3. If you haven't done so before, type `make psvn2c` in the PSVN home directory. This creates a working executable of the psvn2c compiler.
4. Change to the Lesson01 directory.

succ.c

File succ.c contains a C program that reads a state from stdin and prints out the state's successors (children) and the cost and label of the operator that generates each successor. It illustrates several of the most important elements in the PSVN API.

The most important thing to note is that this program is completely independent of the state space to which it is applied. This is because its manipulation of states is done entirely through the PSVN API.

sliding_tile1x3.psvn

File sliding_tile1x3.psvn contains a PSVN definition for a 1x3 sliding tile puzzle. It uses the symbol b to represent the blank and numbers 1 and 2 to represent the tiles. Each operator has a label and a cost. Although very small, it illustrates almost all the elements of the PSVN language.

Putting the pieces together

How do we create a version of succ.c that is specific to the 1x3 sliding tile puzzle as defined in sliding_tile1x3.psvn? The answer is `make sliding_tile1x3.succ`. This will create an executable called sliding_tile1x3.succ that will perform the function defined by succ.c specifically for the 1x3 sliding tile puzzle as defined in sliding_tile1x3.psvn.

If you look inside the Makefile you will see this happens in two steps. First, the psvn2c compiler is applied to sliding_tile1x3.psvn to create a file called sliding_tile1x3.c. You should see this file in the Lesson01 directory once the make has finished. This is a C program that implements the PSVN API for the 1x3 sliding tile puzzle (as defined in sliding_tile1x3.psvn). You will need to be delete this file manually if you make any changes to the PSVN file from which it was created. Once sliding_tile1x3.c has been created, program succ.c is compiled with sliding_tile1x3.c included in it. The result of this compilation is sliding_tile1x3.succ.

Go ahead and make sliding_tile1x3.succ and try executing it.

Exercises

Note: In all the tutorial lessons, the time required to do an exercise can vary from a few minutes to a day or two. You are not required to do all the exercises in one lesson before proceeding to the next lesson. The exercises, especially the larger ones, are meant to illustrate what you should be capable of doing if you have fully understood the lesson.

1. How many successors does the state `b b b` have ? Note that they are not produced in the order that the rules occur in sliding_tile1x3.psvn.
2. Modify succ.c so that it also prints out if a successor is a goal state or not. You will need additional things from the PSVN API for this.
3. Modify succ.c so that it reads in two states and determines if the second state is a child of the first one. You will need additional things from the PSVN API for this.
4. Modify succ.c so that it prints out a random path of length 4 starting at the given state.
5. Create a PSVN file for the 2x2 sliding tile puzzle (call it sliding_tile2x2.psvn.), then create and test out sliding_tile2x2.succ.
6. Create PSVN files and the corresponding .succ programs for a variety of small state spaces (the Arrow puzzle, Towers of Hanoi, the Pancake puzzle, etc.).
7. You will find it tedious and error-prone to create PSVN files by hand. The solution is to write a program, called a PSVN generator, that creates the PSVN for a problem domain. This idea is explained in detail in Lesson #1A and numerous PSVN generators can be found in the directory ProblemDomains.

PSVN Tutorial Lesson #1A

Abstract

This lesson is about writing PSVN generators for problem domains.

Before Starting this Lesson

1. Complete Tutorial #1.
2. Change to the ProblemDomains directory.

What is a PSVN Generator?

Most problem domains have one or more parameters associated with them, and different settings of those parameters define a different version of the problem. For example, in the Towers of Hanoi the number of disks is the most natural parameter. The number of pegs could be another parameter. The idea of a PSVN generator is to write a program (in any language you like) that will accept the values of the problem domain's parameters and generate the PSVN representing that specific version of the problem domain.

Alternative Encodings

An "encoding" of a problem domain is a choice of state variables to represent the states in the problem domain. Usually there are several different encodings that are more or less equally natural and compact. The encoding is known to have a significant effect on several aspects of search algorithms. For example, the Blocks World can be encoded with a "hand" to pick up and put down a block, or without a hand, in which case a block on top of a stack can be moved directly to the table or to the top of any other stack. Solutions using the "hand" encoding are twice as deep as solutions using the "no hand" encoding, but the branching factor in the "no hand" encoding is roughly the square of the branching factor in the "hand" encoding.

A situation in which there are always two different encodings is when states are permutations, as in the Sliding Tile puzzle, the Pancake Puzzle, TopSpin, and Scanalyzer. In permutation problems, the number of "objects" (e.g. pancakes in the Pancake puzzle) is always exactly the same as the number of "locations" where objects can be placed. In the "standard" encoding, there is one state variable for each location and the value of a variable indicates which object is in that location in the current state. In the "dual" encoding, there is one state variable for each object and the value of a variable indicates the location of that object in the current state. The two encodings can produce very different results. In some cases, operator preconditions are much easier to express in one encoding than in the other (this is why the dual encoding is not supported in the Pancake Puzzle generator, for example).

In order to study the effects of different encodings, you are encouraged to support multiple encodings when you write a generator for a problem domain.

Cost Models

A cost model defines how operator costs are determined. The simplest cost model has all operators costing one. This is called the "unit cost" model. Another common cost model in experiments is to assign operator costs randomly in a given range (e.g. between 1 and 100). There can also be cost models specific to a problem domain. In the Towers of Hanoi, for example, it is natural to think of a larger disk being "heavier" than a smaller disk and therefore more costly to move.

A PSVN generator ideally supports several cost models. In most cases there are parameters associated with the cost model in addition to the parameters associated with the problem domain. For example, to generate random costs in a given range, one needs parameters specifying the lower and upper values of the range and the random number seed.

Using Operators Backwards

Lesson #3 is all about using operators backwards. It points out that it is possible to write operators that work correctly in the forward direction but don't work as intended if used backwards. PSVN provides a solution to this that does not sacrifice efficiency (the special symbol "*"). This is important to keep in mind when writing a PSVN generator, since it is quite likely that the PSVN created will, for one reason or another, be used in the backwards direction. All generators should create PSVN that works correctly and efficiently in both the forward and backwards directions.

Exercises

1. Study the given PSVN generators. Try adding a different cost model to one or more of them, or a different encoding.
2. Try writing your own generator for a new state space. Support multiple cost models and, if appropriate, multiple encodings.

References

The following papers investigate the effect of different encodings on the performance of search or planning systems.

1. Sandra Zilles and Robert C. Holte (2010), "The Computational Complexity of Avoiding Spurious States in State Space Abstraction", *Artificial Intelligence*, 174:1072-1092.
2. Pat J. Riddle, Robert C. Holte, and Michael W. Barley (2011), "Does Representation Matter in the Planning Competition?", Symposium on Abstraction, Reformulation, and Approximation (SARA).
3. Levi Lelis, Sandra Zilles and Robert C. Holte (2013), "Stratified Tree Search: A Novel Suboptimal Heuristic Search Algorithm", Conference on Autonomous Agents and Multiagent Systems (AAMAS).

PSVN Tutorial Lesson #2

Abstract

This lesson is to study a complete search algorithm using PSVN. There is very little new, PSVN-wise, compared to Tutorial #1, but the search algorithm (DFID) is more complex than the successor program used to get you started.

Before Starting this Lesson

1. Read up to the end of Section 3.1 in the PSVN Manual and complete Lesson #1.
2. If you aren't familiar with the search algorithm called "depth-first iterative deepening" (DFID), learn about it.
3. Go to the Lesson02 directory.

dfid.c

File `dfid.c` contains a `C` program that reads a start state from `stdin` and uses depth-first iterative deepening (DFID) to find a minimum length path from start to goal. As with `succ.c` from Lesson #1, the most important thing to note is that this program is completely independent of the state space to which it is applied. This is because its manipulation of states is done entirely through the PSVN API.

slidejump07.psvn

File `slidejump07.psvn` contains a PSVN definition for a "slide-jump" peg solitaire puzzle. The puzzle is described in the comments at the beginning of the file. It was chosen as a test domain for DFID because it contains deadend states (ones with no children) at various depths, a good way to test that the code for updating the depth bound is working correctly. It also has a fairly deep solution (15 moves) for the standard start state, and its state space is a finite directed acyclic graph, so DFID will always terminate on it.

Putting the pieces together

Similar to Lesson #1 type `make slidejump07.dfid`. This illustrates a convention we are going to follow throughout these lessons. If we have a program called `xxx.c` (or `xxx.cpp`) which we wish to apply to a state space define by `ss.psvn`, we will create a Makefile that has `%.xxx` as a target so that we can put the pieces together by typing `make ss.xxx`. Although the details of the Makefiles might differ depending on the target, they will often look very much like the Makefile in Lesson #1. For example, the Makefile for this lesson is identical to the one for Lesson #1 except that `succ` has been replaced everywhere by `dfid`. Another convention we will follow is to not delete the `ss.c` file that gets created as an intermediate step in making `ss.xxx`. For example, you should see `slidejump07.c` in the Lesson02 directory after executing `make slidejump07.dfid` the first time. We keep this file because the `psvn2c` process that creates it can sometimes be quite time-consuming. It needs to be deleted (manually, by you) if you make any changes to the PSVN file from which it was created.

Go ahead and make `slidejump07.dfid` and try executing it. The first thing to try is the standard start state: `r r r e y y y`.

Exercises

1. Change `slidejump07.psvn` so that the goal is to get all the yellow pegs adjacent to one another but not at either end of the puzzle **and** there is at least one red peg to the right of all the yellows (such a state can be reached from the standard state state).
   Reminder: If you change `slidejump07.psvn` (or any other `.psvn` file), you will need to delete the corresponding `.c` file manually, the `Makefile` is not set up to do this for you.
2. Change DFID so that it takes into account operator costs. i.e. instead of finding the minimum length solution, it finds the least-cost solution in spaces where different operators have different costs. This is an important change to make before getting to Lesson #5 on heuristics, because once we have heuristics, it will be essential to have a cost-based version of DFID (its official name is IDA*). Be sure your code correctly handles 0-cost operators. A good test for your code would be a cost-based version of the Pancake puzzle or TopSpin in which the cost of an operator was a function of the tokens moved by the operator.
3. Read Section 3.4 of the PSVN Manual. Modify `dfid.c` and the Makefile to use move pruning and run an experiment like the one described in Section 3.4 (note: the `hist_len` parameter is 1 less than L in the table).

PSVN Tutorial Lesson #3

Abstract

This lesson looks at searching backwards.

Before Starting this Lesson

1. Read up to the end of Section 3.2 in the PSVN Manual and complete Lessons #1 and #2.
2. Read about the state_map data structure in the psvn2c Appendix to the PSVN API Manual.
3. If you aren't familiar with the search algorithm called Dijkstra's algorithm, learn about it.
4. Go to the Lesson03 directory.

dist.cpp, distSummary.cpp

File dist.cpp contains a C++ program (not C) that searches backwards from the goal in order to calculate distances to goal[4] of all the states from which the goal can be reached. It uses Dijkstra's algorithm in conjunction with the backward operators that psvn2c generates automatically from the operators given in a PSVN file. As with all the programs in these tutorials, dist.cpp is completely independent of the state space to which it is applied because its manipulation of states is done entirely through the PSVN API. distSummary.cpp is exactly the same as dist.cpp except that it prints a summary of distances to goal to standard output instead of the distance for each individual state.

This is an important program. When it is applied to an abstraction of a state space ("abstraction" is the topic of Lesson #5) the state_map it creates (and, optionally, writes to a file) is a pattern database (PDB) that will be used by our heuristic search algorithms. It is also important because one of the classic search algorithms, A*, is very closely related to it.[5]

hanoi4p03d.psvn, hanoi4p03d_bad.psvn

Files hanoi4p03d.psvn and hanoi4p03d_bad.psvn each contain a PSVN definition for the 4-peg Towers of Hanoi puzzle with 3 disks. Both use the encoding discussed in Section 2 of the PSVN manual. They are identical except for the operators that move the smallest disk. As discussed in Section 2.3 of the PSVN manual, file hanoi4p03d_bad.psvn is missing an implicit precondition for these operators. The first operator, for example, moves the smallest disk from peg 1 to peg 2. In hanoi4p03d_bad.psvn it tests if the smallest disk is on peg 1, but it does not check if the smallest disk is not on peg 2. In hanoi4p03d.psvn both conditions are checked. This makes no difference if the operators are used in the forward direction starting at a state in which the smallest disk is on exactly one peg but, as explained in Section 3.2 of the PSVN manual, this does affect what happens when the operators are used in the backwards direction.

[4]Some authors use "distance to goal" of state s to mean the number of edges in a path from s to the goal with the fewest edges regardless of their cost. In the PSVN documents and code the "distance to goal" of state s is the cost of a least-cost path from s to the goal regardless of how many edges are in the path.

[5]The priority queue used by dist.cpp is designed to be used by A*. That is why it has two keys (the primary key is "f", in A* terminology, the secondary key is "g"), even though Dijkstra's algorithm only requires one key ("g").

Putting the pieces together

Following our convention, type make hanoi4p03d.dist and make hanoi4p03d_bad.dist. Note that these are C++ programs, not C. When they are executed you will see they produce different results. This is because the backwards operators generated from hanoi4p03d_bad.psvn are missing implicit preconditions and therefore generate more states than they should.

Exercises

1. Modify succ.c from Lesson #1 so that it computes the predecessors of a state instead of the successors (called the resulting program pred.c).
2. Make a version of dist.cpp called dijkstra.cpp that reads in a start state and applies Dijkstra's algorithm forward from that state until the distances from that state to all reachable states have been calculated. Change the makefile so that it can make ss.dijkstra given a state space defined by ss.psvn. When applied to any start state hanoi4p03d.dijkstra and hanoi4p03d_bad.dijkstra should produce identical output—this shows that the "bad" operators work correctly in the forward direction. When applied to the start state in which all the disks are on peg 1 the number of states at each distance should be the same for hanoi4p03d.dijkstra and hanoi4p03d.dist, but not for hanoi4p03d_bad.dijkstra and hanoi4p03d_bad.dist, showing that the backwards operators based on hanoi4p03d.psvn are working correctly but those based on hanoi4p03d_bad.psvn are not.
3. Section 2.3 of the PSVN manual described the special symbol "*". Add it to the **appropriate** places in hanoi4p03d.psvn so that the operators have exactly the same behaviour, forwards and backwards, as in hanoi4p03d.psvn, but are more efficient (because fewer preconditions are tested).
4. If you have written any PSVN generators of your own (Lesson #1A), check them to make sure the PSVN they generate works as intended when used backwards and that the special symbol "*" is generated everywhere that it is appropriate. Test the PSVN with your pred.c program.

PSVN Tutorial Lesson #4

Abstract

This lesson looks at one way of testing a search algorithm.

Before Starting this Lesson

1. Complete Lessons #2 and #3.

dfid.c and dijkstra.cpp

File `dfid.c` is identical to the `dfid.c` in Lesson #2 except that a command line option has been added. If you don't specify the command line option it behaves exactly like `dfid.c` in Lesson #2. If you specify the command line option `--test`, instead of prompting for input and reading one state from `stdin`, `dfid.c` does not prompt for input, it keeps reading one line at a time from `stdin` until there are no more input lines, and each input line should contain a distance from goal followed by a state. It calls the `dfid` function to get a solution length for the state and then compares that solution length to the distance that was read in. This is called "testing mode", and it is useful if you have a file of states whose distances to goal are known, or you can pipe the output of `dist.c` into `dfid.c --test` and test if `dfid.c` correctly calculates the distances to goal of all the states from which the goal can be reached.

`dijkstra.cpp` is Dijkstra's algorithm, the "forward" version of `dist.cpp` from Lesson #3 (except that `dist.cpp` finds all distances to goal, whereas `dijkstra.cpp` finds the cost of the least-cost path from a given start state to goal). It has the same `--test` command line option as `dfid.c` and is included here to show that the code for this option can, and should, be included in all the search functions that you write.

Makefile

Testing search algorithms thoroughly is important. This Makefile provides a very easy way for you to test a search algorithm in which you have implemented the `--test` command line option described above. This Makefile has two command line arguments that must be provided, `p` and `dir`. Argument `p` specifies which program you want to test. Argument `dir` specifies a directory containing the test files to use for testing program `p`.

To illustrate this, there is a directory in the `PSVN` home directory called `TEST` that contains two subdirectories, `Small` and `SmallCosts`. Exactly what they contain is described below, but the key difference between them is that one of them (`Small`) is for testing programs that do not take operator costs into account, whereas the other one (`SmallCosts`) is for testing programs that do take operator costs into account (of course, those programs should also work correctly when all operator costs are one).

To test program `dfid.c` on all the files in `TEST/Small` simply type `make test p=dfid.c dir=../TEST/Small`.

To test program `dijkstra.cpp` on all the files in `TEST/SmallCosts` simply type `make test p=dijkstra.cpp dir=../TEST/SmallCosts`.

What is in the test directories?

Have a look in directory `TEST/Small`. What you will see is a number of `.psvn` files together with the corresponding `.c` and `.dist` files created using the Makefile in Lesson #3. To add more problem domains for testing, you would first create the `.psvn` file (either manually or using a generator, as described in Lesson #1A), use the Makefile in Lesson #3 to create the corresponding `.c` and `.dist` files, and then move these into the test directory (`TEST/Small` or `TEST/SmallCosts` or a test directory of your own making).

Exercises

1. Add one or more new test files to `TEST/Small` and test `dfid.c` using the Makefile in this lesson. Make sure you see in the output that is produced that `dfid.c` has been applied to your new test files and worked properly.
2. Add one or more new test files to `TEST/SmallCosts` and test `dijkstra.cpp` using the Makefile in this lesson. Make sure you see in the output that is produced that `dijkstra.cpp` has been applied to your new test files and worked properly.

PSVN Tutorial Lesson #5

Abstract

This lesson looks at using abstraction to create a pattern database.

Before Starting this Lesson

1. Read up to the end of Section 3.5 of the PSVN Manual and complete Lessons #2 and #3.
2. If you aren't familiar with domain abstraction, projection, pattern databases (PDBs), and the IDA* search algorithm, learn about them.
3. Go to the Lesson05 directory.

abstractor.cpp and hanoi3_5d.psvn

Program abstractor.cpp is briefly described in Section 3.5 of the PSVN Manual. A PSVN file is specified as a command line argument and abstractor.cpp then reads a series of commands from stdin, which can include commands specifying projection and/or domain abstractions to apply to the PSVN file. The set of possible commands is printed when abstractor.cpp begins execution. When there are no more commands, abstractor.cpp writes out two files: (1) a file containing the new PSVN definition named absname.psvn, where absname is given as the second command line argument to abstractor.cpp; and (2) a file named absname.abst containing the representation of the abstraction (as illustrated in Figure 10 of the PSVN Manual). The code for abstractor.cpp illustrates how to use the PSVN class, which is how a PSVN definition is represented internally. The definition of the PSVN class is in the file psvn.hpp in the main PSVN directory.

hanoi3_5d.psvn is the PSVN for the 3-peg, 5-disk Towers of Hanoi, in the special encoding for the 3-disk version of the problem. It is useful for illustrating abstraction because projection has an intuitive meaning (each variable represents the status of a disk, so when you project out a variable you are eliminating a disk) and it is multi-valued, so domain abstraction is also possible. The meaning of domain abstraction is **not** intuitive; you should think about exactly what it means to map values 1 and 2 to the same constant (say, 1).

Before proceeding further in the tutorial, compile abstractor.cpp (make abstractor) and test it on hanoi3_5d.psvn with a variety of projections and domain abstractions, first separately and then together. Look carefully at the resulting PSVN file to understand exactly what is being done.

Makefile

The six-step process for creating a PDB described at the end of Section 3.5 of the PSVN Manual is coded in the pdb target of this Makefile. There are two arguments that must be specified: absname, as described above, and ss, the prefix of the .psvn file of the original, unabstracted state space (e.g. if we wish to create a PDB for hanoi3_5d.psvn and name the resulting abstraction files with the prefix foo, we would type make pdb ss=hanoi3_5d absname=foo). If absname is not specified its default is absname. During this make process abstractor.cpp is compiled and executed, so user input, as described above, is required. Optionally, if a file called foo.txt exists it will be used as the source of commands for abstractor.cpp instead of stdin.

The PDB is created by the first three steps of the process, the subsequent steps compile and run a search algorithm using the PDB created by the first three steps. In this lesson IDA* is the search algorithm used (see ida.c below for more about it). The final step is also interactive, the user is asked to enter start states. Optionally, if a file called baz.test exists (where baz is the string assigned to the ss argument) it will be used as the source of start states for this step instead of stdin.

Five files will remain after this make process has finished: (1) foo.abst (created by abstractor.cpp), (2) the PDB is in foo.state_map (created by running distSummary.cpp from Lesson #3 on foo.psvn), (3) ss.ida is an executable for ida.c specialized to search the state space defined by ss.psvn, (4) and (5) are the .c files created when psvn2c is run on ss.psvn and foo.psvn.

If the PDB foo.state_map has been created by this make process and you wish to run IDA* with the PDB on additional start states, there is no need to go through the make process again, you can just run ./ss.ida foo (in our example ./hanoi3_5d.ida foo).

ida.c

ida.c is a straightforward, efficient implementation of IDA*, complete with timers and counters for running actual experiments.

The key thing to study in this code are the two components of the abstraction_data_t structure. One of these (abst) is of type abstraction_t. This is the data structure used to store an abstraction mapping, as illustrated in Figure 10 of the PSVN Manual. This type is defined by psvn2c when it compiles a PSVN file. The functions that manipulate this type (e.g. the ones used in ida.c) are defined in the section on ABSTRACTION in the PSVN API Manual.

The other component (map) is the actual PDB stored in a data structure called a state_map; its type is state_map_t. This type, and the functions that manipulate it, were first used in Lesson #3 and are described in the Additional Functions section of the psvn2c Appendix to the PSVN API Manual.

It is important to know how to use the state_map_t type because the PDBs in all your heuristic search algorithms will be of this type (unless you write your own PDB data structure). The state_map is not the ideal data structure for a PDB because it can be very wasteful of memory. A new data structure for PDBs, based on the BDZ public domain minimal perfect hash function code at http://cmph.sourceforge.net/bdz.html, is currently being incorporated into the PSVN system. It

has excellent memory performance and its lookup time is adequate, although not as fast as the lookup time for a state_map.

The other feature of the `PSVN` API that is illustrated in `ida.c` is the use of `#ifdef`'s to change how IDA* behaves depending on the command line options specified for `psvn2c` when the `PSVN` description was compiled. As described in the `PSVN` API Manual, `psvn2c` will generate `#define HAVE_FWD_MOVE_PRUNING` if and only if the forward search move pruning part of the API is generated (by specifying a `len` greater than zero in the `psvn2c` command line option `--fwd_history_len=len`). `ida.c` tests if `HAVE_FWD_MOVE_PRUNING` is defined. If it is, then the code in `ida.c` that uses move pruning is activated (by `#ifdef`'s). If it is not, the code `ida.c` that does parent pruning is activated.

Exercises

1. Test IDA* on various abstractions of `hanoi3_5d.psvn`. Notice how its run time changes depending on whether it is a given a strong or weak heuristic.[6] Also test IDA* on a variety of abstractions of other state spaces.
2. Modify the given `ida.c` so that it uses two PDBs instead of one (to compute the heuristic value of state s it should take the maximum of the values returned for s by each PDB).

Do Lesson #4 prior the following exercises.

3. Create an implementation of A* by modifying the `dijkstra.cpp` code in Lesson #4 so that it uses a PDB as a heuristic function. Ignore the testing mode for now (see next exercise). Note that once a PDB has been built using Lesson #5's Makefile, it can be used with any heuristic search algorithm.
4. To test a search algorithm that uses a PDB, you should try it on a variety of problem domains, as discussed in Lesson #4, and for each problem domain you should test it with a variety of PDBs. This means adding two or more `.state_map` files, and the corresponding `.abst` files, for each domain in `TEST/Small` and/or `TEST/SmallCosts`, and changing the makefile given in Lesson #4 so that it iterates over all the PDBs for a given domain. Do this. Test your A* code. Add testing mode code to `ida.c` and test it.

[6] In the 3-peg Towers of Hanoi the distance between states in which all the disks are on one peg grow exponentially with the number of disks ($2^d - 1$ when there are d disks). Even with 5 disks, this can be challenge for IDA* unless it is given quite a good heuristic.