varp python API documentation

Instance functions

```
varpy.new(options)
```

Create a new varp instance from a dict of options

- {'size': size | 'default'}
  Initial variable table size (default 1024)
- {'qtype': 'lifo'|'fifo'|'recursive'} (default = **recursive**)
  Use lifo/fifo strategy in bcp
- {'xref': x} (default = **False**)
  Use cross references if x is **True**
- {'vsids': x} (default = **True**)
  Use variable decaying sum variable selection ifg x is **True**
- {'init_phase': x} (default = **True**)
  The initial phase to start with, x is boolean. Set to **None** if
  random value is requested.
- {'use_phase': x} (default = **False**)
  Use saved phase in decide iff x is **True**
- {'all_used': x} (default = **False**)
  All variables are "used" if 'all_used' is **True**,
  that is all variables created are backtracked
  and turn up in calls to next_unbound. If all_\used is **False** then
  varpy.isused and literal degree controls when variabels are used.
- {'hash': x} (default = **False**)
  use hash table for clauses iff x is **True**
- {'seed': x} (default = 0)
  random seed used (64-bit unsigned number). Use seed = 0 when a,
  kind of non deterministc, number is wanted as seed.

```
varpy.clone(vp, options)
```

Clone the varp instance using setting options from varpy.new with the
follow additions:

- {'level': l}
  clone bindings up until level **l**
- {'set': **clauseset** | [**clauseset**]}
  clone clauseset DELTA, GAMMA, BETA and/or ALPHA
- {'queue', x}
  clone bcp queue only if x is True.

where **clauseset** = 'delta'|'gamma'|'beta'|'alpha'

```
varpy.info(vp,  item)
```

Get varp information

- version

- 'bcp_counter'
- 'level'
- 'conflict_counter'
- 'max_conflicting'
- 'num_conflicting'
- 'number_of_variables'
- 'number_of_clauses'
- 'number_of_edges'
- 'number_of_dead_clauses'
- 'number_of_dead_edges'
- 'number_of_learnt_clauses'
- 'number_of_bound_variables'
- 'number_of_subst_variables'
- 'number_of_unbound_variables'
- 'clause_n_counter'
- 'clause_2_counter'
- 'clause_3_counter'
- 'clause_d_counter'
- 'size'
- 'qtype'
- 'max_level'
- 'min_level'
- 'max_bound'
- 'literal_size'
- 'literal_integer'
- 'value_packing'
- 'edge'
- 'xref'
- 'hash'
- 'init_phase'
- 'use_phase'
- 'seed'

```
varpy.config(vp, item, value)
```

Set configurable items in varp

- 'max_conflicting'
  set max number of conflicts during bcp (<= MAX_CONFLICTING=1024)
- 'xref'
  turn on (**True**) or off (**False**) cross reference handling
- 'vsids'
  enable (**True**) or disable (**False**) the use of VSIDS,
  variable decaying sum variable selection.
- 'hash'
  turn on (**True**) or off (**False**) hash table handling
- 'qtype'
  set style of literal queueing in bcp
  where value is one of
  'lifo' | 'fifo' | 'recursive'
- 'seed'
  random seed used (64-bit unsigned number). Use seed = 0 when a,

kind of non deterministc, number is wanted as seed.

```
varpy.add_variable(vp)
varpy.add_variable(vp, is_atom)
varpy.add_variable(vp, is_atom, is_used)
```

Create a new variable. The variable is return as an index to the
next available variable in the variable table. Mark the new variable
as atom if is_atom is **True**. The atom status may later be queried with
variable info. is_atom defaults to **True**.

exception: system_limit (too many variables)

```
varpy.add_variables(vp, num)
varpy.add_variables(vp, num, is_atom)
varpy.add_variables(vp, num, is_atom, is_used)
```

Create **num** new variables. The variables are return as a tuple
(**firstindex**, **lastindex**). If is_atom is **True** then the variables
are marked as atom.

exception: system_limit (too many variables)

```
varpy.level(vp)
```

Return current binding level.

```
varpy.value(vp, x)
```

Return **True** | **False** | **None**,
Value, None is return if variables undefined.

exception: literal (x is not a literal)

```
varpy.bound(vp, x)
```

Return **True** | **False** | **None** | literal

None is return if variable x is unbound,
if x is bound to literal y then y is returned.

exception: literal (x is not a literal)

```
varpy.bind(vp, x)
```

Bind variable x to True.

```
varpy.decide(vp, x)
```

Use the decision parameters to decide and bind variable x
also mark x as a decision variable.
The value used for x is initially the value of 'init_phase' as
setup in varpy.new. If 'use_phase' is true the last propagated
value is used as next initial decision value.

```
varpy.subst(vp, x, y)
```

Substitute one literal for an other. Apply the substitution [**x**/**y**]
that is, all instances of **y** are replaced by **x** in all clauses.
The literal **y** is then linked to **x** so it will keep the
same status and value as that of **x**.
NOTE that cross references must be enabled before varpy.subst can be
used, that is a call to varpy.config(vp, 'xref', True) must have been made
prior a call to varpy.subst.

exception: literal (if x or y are not literals)
exception: level (when level != 0)
exception: xref (when xref is not turned on)

```
varpy.implication_clause(vp, x)
```

Return the clause index for the clause where literal **x** became a unit.

exception: variable (x is not a variable)

```
varpy.implication_level(vp, x)
```

Return the bind level for variable **x**, where it was assigned during bcp.

exception: variable (x is not a variable)

```
varpy.conflicting_clause(vp, i)
```

Return the i'th conflicting clause found during the last bcp. The number
of conflicting clauses that can be returned is 'num_conflicting'.

```
varpy.is_variable(vp, x)
```

Return **True** if literal **x** is unbound. Return **False** otherwise

exception: variable (x is not a variable)

```
varpy.is_bound(vp, x)
```

Return **True** if literal **x** is bound, through substition,
to an other variable. Return **False** otherwise.

exception: variable (x is not a variable)

```
varpy.is_equal(vp, x, y)
```

Check if literal **x** and literal **y** are the equal, that is bound to the
same variable or are bound to the same constant.

exception: literal (x or y are not literals)

```
varpy.isused(vp, x)
varpy.isused(vp, x, value)
```

Check if literal **x** is used in any clause or is forced to be
in use by a previous setting of value. This makes "free" variables
be included in fist and next_unbound calls.

exception: variable (x is not a variable)

```
varpy.isatom(vp, x)
varpy.isatom(vp, x value)
```

Check if literal **x** is an **atom**, that is
marked as atomic formula when adding new variables
or set using is_atom(vp, x, True)

exception: variable (x is not a variable)

```
varpy.phase(vp, x)
```

Lookup the stored phase of variable **x** and return
-1 if negative 1 if positive and **None** if not set.

```
varpy.set_phase(vp, x)
```

Set the phase of a variable given literal x,
if x is negative the the phase is set to -1 otherwise
the phase is set to 1. The previous phase value for
variable x is returned.

```
varpy.push(vp)
```

Push binding level stack and move on to the next,
return the level number before the push.

```
varpy.pop(vp)
```

Undo all bindings on current level and move to the
previous level. Return the level number after pop.

```
varpy.pop(vp, l)
```

Undo and pop all bindings until level l, but NOT including level l.

```
varpy.undo(vp)
```

Undo bindings typically after an nbind. Undo will undo all bindings
until a decision and flip the variable if not already flipped.

```
varpy.bcp(vp)

varpy.bcp(vp [x1,..,xn])
varpy.bcp(vp [x1,..,xn], all)
```

Run value propagation. Return **True** if no
contradiction is found, **False** otherwise.

[EXPERIMENTAL]

If literals **x1**..**xn** are given they are checked for
"turbo" rule, that is, if all clauses that xi
is a part of are true regardless of the value of xi.

If 'all' is **True** then all xi's must be true for the
rule to hold. If turbo rule is successful then
'turbo' is returned.

[EXPERIMENTAL]

```
varpy.nbcp(vp)
```

Repeatedly decide/bind next unbound variable, on a new level and
run bcp until there are no more variables to bind or a contradiction
is found. If a contraduction is readed them return **False**
otherwise there is a model, and **True** is returned.
nbcp can be use together with undo to implement
simple backtracking using a tight loop:

```
    def bt(vp):
        while not varpy.nbcp(vp):
          if varpy.undo(vp) == False:
            return False # contradiction
        return True # model
```

```
varpy.vbcp(vp, [x1,...,xn])
varpy.vbcp(vp, [x1,...,xn], single_level)
```

Vector bcp.

If **single_level** is **false** (default) then each
literal xi in x1...xn xi is used as decision followed
by a bcp. if bcp generates a conflict then **false** is returned,
If xj in xi+1...xn is inconsistent then (j,xj) is returned.
else if xi is not last then the level is pushed and xi+1 is
processed.

If **single_level** is **true** then all literals xi in x1...xn,
are assigned, followed by a bcp. if assignment of xi is inconsistent
then (i, xi) is returned, otherwise the return value of bcp is returned.

```
varpy.add_clause(vp, [x1,...,xn])
varpy.add_clause(vp, (x1,...,xn))
varpy.add_clause(vp, [x1,...,xn], clause_set)
varpy.add_clause(vp, (x1,...,xn), clause_set)
```

Create a new clause, given as a literal list and return the
new clause index. All varables indices must already have been
created by calling add_variable. The clause create is installed
in one of four clause sets: 'delta', 'gamma', 'alpha', 'beta'
given by **clause_set**
The 'delta' clauseset is use to store the "problem" formula
clauses while 'gamma' is used for storing learnt clauses.
However the conflict clause(s) created by varpy.conflict are created in
'alpha' and may then, by user, moved into 'gamma'.

```
varpy.find_clause(vp, [x1,...,xn])
varpy.find_clause(vp, (x1,...,xn))
```

Check if the clause [**x1**,...,**xn**] exist among the clausesets.
return clause index if found, return **False** otherwise.

```
varpy.compress_clause(vp,  cix)
varpy.compress_clause(vp,  [x1,...,xn])
```

Return a compressed version of the clause [**x1**,...,**xn**],
or the clause given by the clause index cix.
It writes a utf8 like code with 0x80 bit for continuation bit and
7-bits per byte for integer value. The LSB is coded as the literal sign.
Note that when argument is given as an integer list the elements
do not need to created as variables/literals.

-1000 is translated to unsigned by

```
2*1000 + 1 = 0xb11111010001
```

which is divided in groups of 7 bits starting from LSB

```
(1)1010001,(0)0001111
```

while 1000 is translated to

```
2*1000 = 0xb11111010000
```

which is divided in groups of 7 bits starting from LSB

```
(1)1010000,(0)0001111
```

Then sequence of coded literals are then terminated with a zero.

```
varpy.clause_info(vp, cix, item)
```

Get information about clause given by clause index cix

- 'length'
- 'jump'
- 'status'
- 'watch0'
- 'watch1'
- 'watch'

```
varpy.variable_info(vp, x, item)
```

Get information about variable x

- 'implication'
- 'implication_clause'
- 'level'
- 'phase'
- 'is_atom'
- 'is_used'
- 'degree'
- 'symbol'

exception: variable (x is not a variable)

```
varpy.literal_info(vp, x, item)
```

Get information about literal x

- 'degree'
- 'mark'
- 'xref'
- 'symbol'

```
varpy.del_clause(vp, cix)
varpy.del_clause(vp, [x1,...,xn])
varpy.del_clause(vp, (x1,...,xn))
```

Delete clause **cix** or [**x1**,...,**xn**] from clausesets.
NOTE: When deleting clauses by giving it as a list,
then hashing may be enable (varpy.config(vp, 'hash', True))
to gain reasonable speed.

exception: level (level != 0)

```
varpy.clean_clause(vp, cix)
```

Cleanup clause by removing all false literals on level 0.
if clause is contradictory then exception is raised

exception: level (level != 0)

```
varpy.clean_edges(vp, x)
```

Remove **x** edges, that is clauses on form [-x,y] where y
is constant.

exception: literal (x is not a literal)

```
varpy.get_clause(vp, cix)
varpy.get_clause(vp, cix, skip)
varpy.get_clause(vp, cix, skip, raw)
varpy.get_clause(vp, cix, skip, raw, as_tuple)
```

Return a list of literals given by clause index **cix**.
Remove the literal **skip** from the returned list.
Also remove literals on level 0 if **raw** is **False**.

```
varpy.get_decision(vp, l)
```

Get decision literal on level **l**.

```
varpy.get_undo_state(vp, l)
```

DEBUG

Return the undo state on level **l**

- 'set'
- 'toggle'
- 'done'

- 'undef'

```
varpy.get_bindings(vp)
varpy.get_bindings(vp, l)
varpy.get_bindings(vp, l, as_trail)
varpy.get_bindings(vp, l, as_trail, as_tuple)
```

Return all bindings on level **l**. Return them in order of when binding where made if **as_trail** is **True** otherwise the bidnings are returned as latest binding first (default).
if **as_tuple** is **True** (default) then bindings are returned as a tuple otherwise a list is returned.

A list of literals are returned. A negative literal means that the variable is bound to **False** a positive literal means that the variable is bound to **True**.

```
varpy.get_nbindings(vp, count)
varpy.get_nbindings(vp, count, as_trail)
varpy.get_nbindings(vp, count, as_trail, as_tuple)
```

Return a maximum of **count** bindings.
Return them in order of when binding where made, if **as_trail** is **True**, otherwise the bidnings are returned as latest binding first.
if **as_tuple** is **True** (default) then bindings are returned as a tuple otherwise a list is returned.

A list of literals are returned. A negative literal means that the variable is bound to **False** a positive literal means that the variable is bound to **True**.

```
varpy.get_number_of_bindings(vp, l)
```

Return number of bindings on binding level **l**.

```
varpy.order_sort(vp, key)
varpy.order_sort(vp, key, arg)
varpy.order_sort(vp, key1, key2)
varpy.order_sort(vp, key1, key2, arg)
```

Order variables according to **key1** and then **key2**, an optional unsigned integer **arg** may be supplied when needed by sorting.
In the case of 'random' sort the **arg** is the random seed
(use **arg** = 0 to set an arbitrary seed)

The sort keys available are:

- 'identity'
  Sort according to when the variable number, this mostly corresponds to when the variable was created.
- 'random'
  Sort variables using a uniform distribution, an integer seed may be given as **arg**.
- 'degree'

Sort literals according to the number of time they occur in the clauses.

- 'rank'
  Sort literals according to the sum of ranks for all occurences in all clauses. The rank for literal x is defined as the sum of 1/|ci| for all clauses ci where x is a member.

If the sort key is prefixed with a '+' then sorting is ascending. If prefix is '-' then the sort is descending, wich is also the default.

exception: level (when level != 0)

```
varpy.order_first(vp, [x1,..,xn])
varpy.order_first(vp, [x1,..,xn], set_phase)
```

Update current sort order so that literals **x1**..**xn** are placed first.
If **set_phase** is True then the sign of the literal is used to update the phase, for later decision. Otherwise the sign of the literal is ignored.

exception: level (when level != 0)

```
varpy.order_last(vp, [x1,..,xn])
varpy.order_last(vp, [x1,..,xn], set_phase)
```

Update current sort order so that literals **x1**..**xn** are placed last.
If **set_phase** is True then the sign of the literal is used to update the phase, for later decision. Otherwise the sign of the literal is ignored.

exception: level (when level != 0)

```
varpy.next_unbound(varp)
varpy.next_unbound(varp, previous)
```

Return the next unbound literal in the current variable order.
If **previous** is given then start looking for unbound literals from that point.

exception: variable (previous is not a variable)

```
varpy.queue_first(varp)
```

DEBUG Return the first literal on the bcp queue.

```
varpy.queue_next(vp, x)
```

DEBUG Return the next literal on the bcp queue, following literal **x**, that must previously being returned from varpy.queue_first of varpy.queue_next.

```
varpy.queue_clear(varp)
```

Remove all literals enqueued on the bcp queue by calls to varpy.bind or varpy.decide.

```
varpy.add_symbol(vp, x, string)
varpy.add_symbol(vp, x, term)
varpy.add_symbol(vp, xs, string)
varpy.add_symbol(vp, xs, term)
```

Associate a term or string to to a variable **x** or variables **xs**,
for example the name of the variable.
The term or string must not be assoicated with
other variables or an exception will occur.
If the variable part is a list **xs** then the symbol refer to a
list of variables, integer encoding or bit vector.
Integer encoding should store least significant bit first (at index 0)

```
varpy.del_symbol(vp, string)
varpy.del_symbol(vp, term)
```

Remove the symbol from the symbol table.

```
varpy.find_symbol(vp, symbol)
```

Given a symbol return the variable assoicated with it.
If no assoication is found **False** is returned.

```
varpy.first_symbol(vp)
```

Find first symbol in the symbol table. Return **False** if
not found. Try not use **False** as a symbol.

```
varpy.next_symbol(vp, symbol)
```

Given a symbol in the symbol table (must be present),
find next symbol in the symbol table. Return **False** if
not found. Try not use **False** as a symbol.
If symbol table is updated while calling next_symbol then
the behaviour is undefined.

```
varpy.use_clause(vp, cix)
```

Update the clause **cix** timestamp to the current bcp_counter,
the number of bcp's that has been run since **vp** instance was created.

```
varpy.bump(vp, x, n)
```

"bump" a variable **x** to move it in the dynamic variable order.
Either bump value **n** is one of

- 'next', move variable to the top position, next variable to be assigned
- 'log2', bump value is calculated to log2('**number-of-variables**')
- 'log10', bump value is calculated to log10('**number-of-variables**')
- 'rank', bump value is set to the length of the implication clause.

if **n** is a floating point value between 0 and 1 then the
bump value will be computed to the relative to the number of variables.
For example a value of, 0.1 means move 10% in number of variables.

If **n** is an integer then x is moved that exact number of steps.

```
varpy.subscribe(vp, flag)
varpy.subscribe(vp, [flag])
```

Flags

- 'variable'
- 'atom'
- 'number_of_variables'
- 'number_of_bound_variables'
- 'number_of_subst_variables'
- 'number_of_clauses'
- 'number_of_dead_clauses'
- 'max_level'
- 'max_bound'
- 'min_level'

```
varpy.clauseset_size(vp, s)
```

Where **s** is one of 'delta', 'gamma', 'alpha', 'beta'

```
varpy.clauseset_offset(vp, s)
```

Get offset where **s** is one of 'delta', 'gamma', 'alpha', 'beta'

```
varpy.clauseset_offset(vp, s, offset)
```

Set offset where **s** is one of 'delta', 'gamma', 'alpha', 'beta'

```
varpy.clauseset_sort(vp, s)
```

Sort clauses in the clause set **s**, where **s** is one of
'delta', 'gamma', 'alpha', 'beta'. The clauses are sorted
according to the internal use counter set by varpy.use_clause.

```
varpy.clauseset_first(vp, s)
```

get clause index of first clause in clauseset **s**

```
varpy.clauseset_next(vp, s)
```

get clause index to the next clause in clauseset **s**

```
varpy.conflict(vp, bump, cix)
varpy.conflict(vp, bump, cix, xi)
varpy.conflict(vp, bump, [x1..xn])
varpy.conflict(vp, bump, [x1..xn], xi)
```

Do conflict analysis on the conflict clause cix or
explict clause [x1,...,xn], the unit literal xi must
be supplied or **false** (default)

The **bump** factor is applied to variables involved in the conflict.
Returned value is a clause index in clauseset 'alpha'. This
clause may then be minimized and later moved to 'gamma'.

if **None** is returned then the conflict clause was a copy of an existing clause.

```
varpy.minimize(vp, cix)
varpy.minimize(vp, cix, 'local'|'global'|'recursive')
```

Minimize clause, may be called after varpy.conflict and requires that literals and levels are set like after the conflict.
Return updated length of clause if successful, or **None** if clause, after minimization, already exists.

```
varpy.move_clause(vp, cix, set)
```

Move clause **cix** to clauseset **set**.
NOTE that this function is currently limited to clause in 'alpha' and set must be 'gamma'

```
varpy.unmark(vp)
```

Clear all marks

```
varpy.mark(vp, l | [x1,...,xn] | (x1,...,xn), [clear])
```

Mark variables **x1**..**xn** or all bindind on level **l**.
Optionally if **clear** is **False** then marks are concatinated to the previous marks, otherwise the marks are cleared before adding new ones.

```
varpy.intersect_marks(vp, l | [x1,...,xn] | (x1,...,xn))
```

Keep all marks that are present in bindings on level **l** or the literals **x1**...**xn**. Clear the other marks.

```
varpy.intersect_var(vp, x, l | [x1,...,xn]|(x1,...,xn), as_tuple)
```

Return all marks that are present in bindings on level **l** or the literals **x1**..**xn**. Return the marks in a list if *_as_tuple* if **False** or as a tuple if **as_tuple** is **True**.

```
varpy.get_marked(vp, as_tuple)
```

Return the marks in a list if *_as_tuple* if **False** or as a tuple if **as_tuple** is **True**.