Adaptive Search

A Library to Solve CSPs
Edition 1.1, for Adaptive Search version 1.2.0
March 14, 2018

by Daniel Diaz, Philippe Codognet and Salvador Abreu

1 Introduction

The Adaptive Search library provides a set of functions to solve CSPs by a local search method. For more information consult [1] The current release only works for problems that can be stated as permutation problems. More precisely, all n variables have a same domain x_1 .. x_n and are subject to an implicit *all-different* constraint. Several problems fall into this category and some examples are provided with the library.

2 Installation

Please refer to the file called `INSTALL` located in the src subdirectory.

3 The Adaptive Search API

3.1 Overall usage

The typical use of the API is as follows:

- Initialize a structure with the input data needed by the solver. This includes problem data (e.g. size, domain,...) together with parameters to tune the solver (e.g. tebu tenure,...).
- Define a set of functions needed by the solver (e.g. to compute the cost of a configuration). Some functions are optional meaning that the solver performs an implicit treatment in the absence of such a function. Most of the times, providing an optional function speeds up the execution.
- Call the solver.
- Exploit the data provided by the solver (the solution, various counters giving information about the resolutions).

To use the API a C file should include the header file `ad_solver.h`:

```
#include "ad_solver.h"
```

Obviously the C compiler must be invoked with the adequate option to ensure the header file can be found by the preprocessor.

At link time, the library called `libad_solver.a` must be passed. Here also, some options might have to be passed to the C compiler to allow the linker to locate the library.

If both the include file and the library are in the same directory as the user C file (for instance `problem.c`), then the following Unix command line (using gcc) suffices:

```
gcc -o problem problem.c libad_solver.a
```

If the include file is in `/usr/adaptive/include` and the library in `/usr/adaptive/lib`, a possible invocation could be:

```
gcc -I/usr/adaptive/include -L/usr/adaptive/lib -o problem problem.c -lad_solver
```

A structure (C type `AdData`) is used to communicate with the solver. Fields in this structure can be decomposed in: input or output data (or input-output). Input parameters are given to the solver and should be initialized before calling the solver. Output parameters are provided by the solver.

Please look at the header file for more information about the fields in the `AdData` type. We here detail the most important parameters.

3.2 Input parameters

The following input variables control the basic data and have to be initialized before calling the resolution function.

- `int size`: size of the problem (number of variables).
- `int *sol`: the array of variables. It is an output parameter but it can also be used to pass the initial configuration if `int do_not_init` is set.
- `int do_not_init`: if set to true (a value != 0) the initial configuration used is the one present in `sol` (else a random configuration is computed).
- `int base_value`: base offset for the domain of each variable (each vaiable can then take a value in `base_value` .. `base_value` + `size` − 1.
- `int *actual_value`: if not `NULL` it contains the array of values (domain) variables can take. If `base_value` is given, it is added to each value of `actual_value` to form the domain.
- `int break_nl`: when the solver displays a solution a new line is inserted every `break_nl` values (0 if no break is wanted). This makes it possible to display matrix in a more readable form.
- `int debug`: debug level (0: none, 1: trace, 2: interactive). This requires the library is compiled with debugging support (see INSTALL).
- `char *log_file`: name of the log file (or `NULL` if none). This requires the library is compiled with log file support (see INSTALL).

The following input parameters make it possible to tune the solver and should be initialized before calling the resolution function.

- `int exhaustive`: if true the solver always evaluate (the cost of) all possible swaps to chose the best swap. If false a projection of the error on each variable is used to first select the “worst” variable ([1] for more information).

- `int first_best`: when looking for the next configuration, the solver stops as soon as a better move is found (instead of continuing to find the best move).
- `int prob_select_loc_min`: this is a percentage to force a local minimum (i.e. when the 2 selected variables to swap are the same) instead of staying on a plateau (a swap involves 2 different variables but the overall cost will remain the same). If a value > 100 is given, this option is not used.
- `int freeze_loc_min`: number of swaps a variable is frozen when a local minimum is encountered (i.e. the 2 variables to swap are the same).
- `int freeze_swap`: number of swaps the 2 variables that have been selected (and thus swapped) to improve the solution are frozen.
- `int reset_limit`: number of frozen varables before a reset is triggered.
- `int nb_var_to_reset`: number of variables to randomly reset.
- `int restart_limit`: maximum number of iterations before restarting from scratch (give a big number to avoid a restart).
- `int restart_max`: maximal number of restarts to perform before giving up. To avoid a too long computation the parameters `int restart_limit` and `int restart_max` can be defined.
- `int reinit_after_if_swap`: see the defintion of the user function `Cost_If_Swap()` for more information.

3.3 Output parameters

In addition to the array containing the solution, the solver maintains counters that can be consulted by the user to obtain some information about the resolution.

- `int *sol`: the current configuration. When the solver teminates, it normally contains a solution. If the solver has finished because it reached the maximum number of iterations and restarts, the `sol` array contains a pseudo-solution (an aprroximation of the solution).
- `int total_cost`: cost of the current configuration (0 means a solution).
- `int nb_restart`: number of restart performed.
- `int nb_iter`, `int nb_iter_tot`: number of iterations performed in the current pass and across restarts.
- `int nb_swap`, `int nb_swap_tot`: number of swaps performed.
- `int nb_same_var`, `int nb_same_var_tot`: number of variables with (the same) highest cost.

- `int nb_reset`, textttint nb_reset_tot: number of reset swaps performed.
- `int nb_local_min`, `int nb_local_min_tot`: number of local minimum encountered.

3.4 Miscellaneous parameters

The following variables are not used by the solver. They simply convey values for the user. It is particularly useful for multithreading. It also contains some information related to the default `main()` function.

- `int param`: the parameter handled by the default `main()` function.
- `int seed`: the seed set by a command-line option of the default `main()` function (or -1 if any).
- `int reset_percent`: -1 or the % of variables to reset defined by a command-line option of the default `main()` function. If it is -1, the `Init_Parameters()` function should either set it to a percentage or directly set the `nb_var_reset` parameter.
- `int data32[4]`: some values to store 32-bits user information.
- `int data64[42`: some values to store 64-bits user information.

3.5 Functions

Here is the set of functions provided by the library:

- `int Ad_Solve(AdData *p_ad)`: this function invokes the Adaptive solver to find a solution to the problem. This function calls in turn user functions (e.g. to compute the cost of a solution or to project this cost on a given variable). This function returns the `total_cost` at then end of the resolution (i.e. 0 if a solution has been found).
- `void Ad_Display(int *t, AdData *p_ad, unsigned *mark)`: this function displays an array `t` (generally `sol`) and also displays a 'X' for marked variables (if `mark != NULL`). This function is generally only used by the solver.

3.6 User functions

The function `Ad_Solve()` calls some user functions to guide its resolution. Some functions are MANDATORY while others are OPTIONAL. Here is the set of user functions:

- `int Cost_Of_Solution(int should_be_recorded)`: [MANDATORY] this function returns the cost of the current solution (the user code should keep a pointer to `sol` it needed). The argument `should_be_recorded` is passed

by the solver, if true the solver will continue with this cost (so maybe the user code needs to register some information), if false the solver simply wants to know the cost of a possible move (but without electing it).

- `int Cost_On_Variable(int i)`: [OPTIONAL] this function returns the projection of the current cost on the i*th* variable (from 0 to `size`-1). If not present then the resolution must be exhausitive (see `exhausitive`).
- `int Cost_If_Swap(int current_cost, int i, int j)`: [OPTIONAL] this function evaluates the cost of a swap (the swap is not performed and should not be performed by the function). Passed arguments are the cost of the solution, the indexes `i` and`j` of the 2 candidates for a swap.. If this function is not present a default function is used which:
  - performs the swap,
  - calls `Cost_Of_Solution()`,
  - undoes the swap,
  - if the variable `int reinit_after_if_swap` is true then `Cost_Of_Solution()` is also called another time. This is useful if `Cost_Of_Solution()` updates some global information to ensure this information is reset.
- `void Executed_Swap(int i, int j)`: [OPTIONAL] this function is called to inform the user code a swap has been done. This is useful if the user code maintains some global information.
- `int Next_I(int i)`: [OPTIONAL] this function is called in case of an exhaustive search (see `exhaustive`). It is used to enumerate the first variable. This functions receives the current `i` (initially it is -1) and returns the next value (or something $>$ `size` at the end). In case this function is not defined, `i` takes the values 0 .. `size` $-$ 1.
- `int Next_J(int i, int j)`: [OPTIONAL] this function is called in case of an exhaustive search (see `exhaustive`). It is used to enumerate the second variable. This functions receives the current `i` and the current `j` (for each new `i` it is -1) and returns the next value for `j` (or something $>$ `size` at the end). In case this function is not defined, `j` takes the values `i` $+ 1$ .. `size` $- 1$ for each new `i`.
- `void Display_Solution(AdData *p_ad)`: [OPTIONAL] this function is called to display a solution (stored inside `sol`). This allows the user to customize the output (useful if modelisation of the problem needs a decoding to be understood). The default version simply displays the values in `sol`.

4 Other utility functions

To use this functions the user C code should include the file `tools.h`.

- `long Real_Time(void)`: returns the real elapsed time since the start of the process (wall time).
- `long User_Time(void)`: returns the user time since the start of the process.
- `unsigned Randomize_Seed(unsigned seed)`: intializes the random generator with a given `seed`.
- `unsigned Randomize(void)`: randomly initilizes the random generator.
- `unsigned Random(unsigned n)`: returns a random integer $>= 0$ and $<$ `n`.
- `void Random_Permut(int *vec, int size, const int *actual_value, int base_value)`: initializes the `size` elements of the vector `vec` with a random permutation. If `actual_value` is `NULL`, values are taken in *base_value* .. `size` $- 1 +$ *base_value*. If `actual_value` is given, values are take from this array (each element of the array is added to `base_value` to form an element of the permutation).
- `int Random_Permut_Check(int *vec, int size, const int *actual_value, int base_value)`: checks if the values of `vec` forms a valid permutation (returns true or false).
- `void Random_Permut_(int *vec, int size, const int *actual_value, int base_value)`: repairs the permutation stored in `vec` so that it now contains a valid permutation (trying to keep untouched as much as possible good values).

5 Using the default `main()` function

The user is obviously free to write his own `main()` function. In order to have a same command-line options for all bechmarks a default `main()` is included in the library (it is then used if no user `main()` is found at link-time). The default function act as follows:

- it parses the command-line to retrieve tuning options, the running mode (number of executions,...), and the parameter if it is expected (e.g. the chessboard size in the queens). NB: if a parameter is expected the variable `param_needed` must be declared and initialized to 1 in the user code. Each tuning option can be set via a command-line option and the corresponding variable (see input variables) is set. The only exception is for for `nb_var_reset` which can be specified indirectly as a percentage (instead of an absolute value) inside the variable `reset_percent`.
- it invokes a user function `void Init_Parameters(AdData *p_ad)` which must initialises all input variables (e.g. `size`, allocate `sol`,...). This function should only initialises tuning variable that are not set via command-line options. In this case the value of the corresponding variable is -1.

- it invokes the user defined `void Solve(AdData *p_ad)` function (which in turn should invoke the Adaptive solver `Ad_Solve()`.
- it displays the result or a summary of the counters (in benchmark mode).

In addition to the variables described above, the following parameters are available when using the default `main()`:

- `int param`: the parameter (if `param_needed` is true).
- `int reset_percent`: -1 or the % of variables to reset defined by a command-line option. If it is -1, the `Initializations()` function should either set it to a percentage or directly set `nb_var_reset`.
- `int seed`: -1 or the seed defined by a command-line option. The `main()` function initialises the random number generator in both cases so the `Initializations()` does not need to do it.

The default default `main()` function needs an additional function to check the validity of a solution. The user must then provide a function `int Check_Solution(AdData *p_ad)` which returns 1 if the solution passed in `p_ad` is valid. A Simple definition for this function could be to simply test if the cost of the solution is 0 (not very precise as verification).

Please look at the examples for more details.

References

[1] P. Codognet and D. Diaz. Yet Another Local Search Method for Constraint Solving. In *Proc. SAGA01*, 1st International Symposim on Stochastic Algorithms : Foundations and Applications, LNCS 2246, Springer Verlag 2001