D-Wave
The Quantum Computing Company™

Developer Guide for C

09-1034A-F

Notice and Disclaimer

D-Wave Systems Inc. ("D-Wave"), its subsidiaries and affiliates, makes commercially reasonable efforts to ensure that the information in this document is accurate and up to date, but errors may occur. NONE OF D-WAVE SYSTEMS INC., its subsidiaries and affiliates, OR ANY OF ITS RESPECTIVE DIRECTORS, EMPLOYEES, AGENTS, OR OTHER REPRESENTATIVES WILL BE LIABLE FOR DAMAGES, CLAIMS, EXPENSES OR OTHER COSTS (INCLUDING WITHOUT LIMITATION LEGAL FEES) ARISING OUT OF OR IN CONNECTION WITH THE USE OF THIS DOCUMENT OR ANY INFORMATION CONTAINED OR REFERRED TO IN IT. THIS IS A COMPREHENSIVE LIMITATION OF LIABILITY THAT APPLIES TO ALL DAMAGES OF ANY KIND, INCLUDING (WITHOUT LIMITATION) COMPENSATORY, DIRECT, INDIRECT, EXEMPLARY, PUNITIVE AND CONSEQUENTIAL DAMAGES, LOSS OF PROGRAMS OR DATA, INCOME OR PROFIT, LOSS OR DAMAGE TO PROPERTY, AND CLAIMS OF THIRD PARTIES.

D-Wave reserves the right to alter this document and other referenced documents without notice from time to time and at its sole discretion. D-Wave reserves its intellectual property rights in and to this document and its proprietary technology, including copyright, trademark rights, industrial design rights, and patent rights. D-Wave trademarks used herein include D-WAVE®, D-WAVE 2X™, D-WAVE 2000Q™, and the D-Wave logo (the "D-Wave Marks"). Other marks used in this document are the property of their respective owners. D-Wave does not grant any license, assignment, or other grant of interest in or to the copyright of this document, the D-Wave Marks, any other marks used in this document, or any other intellectual property rights used or referred to herein, except as D-Wave may expressly provide in a written agreement. This document may refer to other documents, including documents subject to the rights of third parties. Nothing in this document constitutes a grant by D-Wave of any license, assignment, or any other interest in the copyright or other intellectual property rights of such other documents. Any use of such other documents is subject to the rights of D-Wave and/or any applicable third parties in those documents.

All installation, service, support, and maintenance of and for the D-Wave System must be performed by qualified factory-trained D-Wave personnel. Do not move, repair, alter, modify, or change the D-Wave System. If the equipment is used in a manner not specified by D-Wave, the protection provided by the equipment may be impaired. Do not provide access to the customer site to anyone other than authorized and qualified personnel. Failure to follow these guidelines may result in disruption of service, extended downtime, damage to equipment (customer's, D-Wave's, and/or third parties'), injury, loss of life, or loss of property.

CONTENTS

CHAPTER

ONE

INTRODUCTION

The D-Wave QPU is based on a physical lattice of qubits and the couplers that connect them. Together, these qubits and couplers are referred to as the Chimera graph. The lattice structure is a set of connected unit cells, each comprising four horizontal qubits connected to four vertical qubits via couplers. Unit cells are tiled vertically and horizontally with adjacent qubits connected, creating a lattice of sparsely connected qubits. Within a given system, certain qubits or couplers may not function as desired. In such cases, the devices are eliminated from the programmable fabric available. The subgraph available to solve a problem is called the *working graph*.

A given *logical problem* defined on a general graph can be mapped to a *physical problem* defined on the working graph using *chains*. A chain is a collection of qubits bound together to represent a single logical node. The association between the logical problem and the physical problem is carried out by minor *embedding*.

The qubits, denoted q_i, implement the Ising spins. Their physical connectivity determines which couplings, $J_{i,j}$, can be set to nonzero values. The allowed connectivity is described with a Chimera graph; see Fig 1.1. An $M \times N \times L$ Chimera graph consists of an $M \times N$ two-dimensional lattice of blocks, with each block consisting of $2L$ variables, for a total of $2MNL$ variables.

Any discrete optimization problem can be cast as a Chimera-structured Ising problem given a large enough Chimera lattice. Methods are available to reduce higher-order interactions in the optimization objective to pairwise, and to address the connectivity mismatches between the problem and the fixed qubit connectivity of Chimera. In this document, we provide an overview of software tools (available in C packs) that solve these problems using hardware or software simulations and make formulating and solving QUBO and Ising problems simpler.

The functionalities of the utility packs include:

1. Managing connections to solvers, and in particular, solving Ising/QUBO problems by quantum annealing or simulated quantum annealing.
2. Simplifying Ising/QUBO problems to equivalent, but easier to solve problems.
3. Solving non-Chimera-structured problems in solvers using embeddings (special mappings of problem variables to qubits) and find embeddings.
4. Reducing objective functions with high-order interactions to QUBO problems.
5. Using QSage quantum accelerator tool.

Note: The results shown in the examples of this document may differ from the results the user would get from running the same examples depending on the solver being used and its actual parameters.

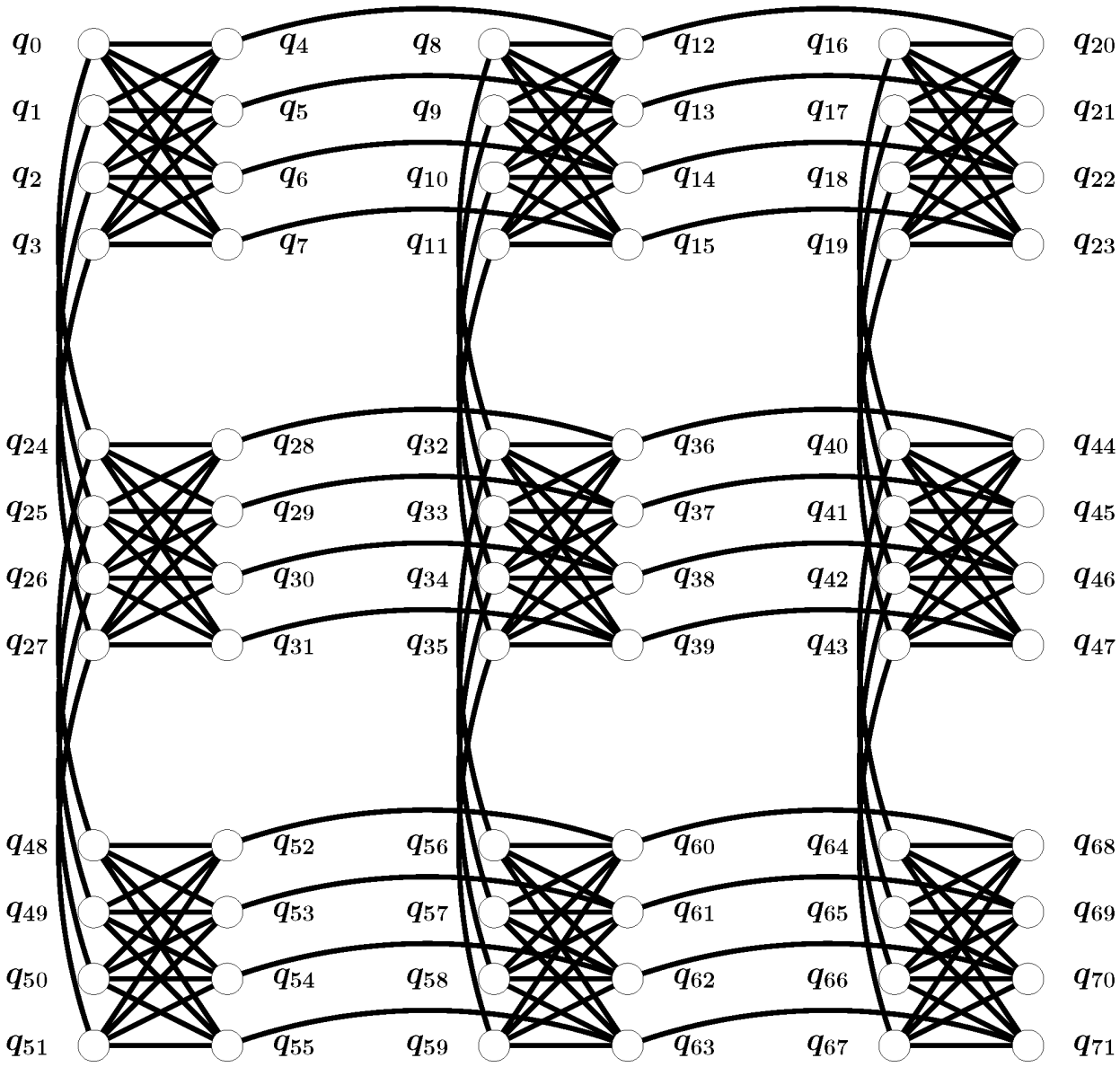

Fig. 1.1: A $3 \times 3 \times 4$ Chimera graph. Nodes in an $M \times N \times L$ Chimera graph represent each of the $2MNL$ qubits, labeled as q_i. Edges (connections between nodes) in the graph, $J_{i,j}$, indicate couplings that may be nonzero. As an example, $J_{3,4}$ may be nonzero because an edge connects qubits 3 and 4, but $J_{2,3}$ must always be zero because no edge connects qubits 2 and 3. The basic repeating block of Chimera (a block of $2L$ variables with complete bipartite connectivity) may be tiled into an $M \times N$ lattice. The left-side variables within each block connect vertically; the right-side variables, horizontally.

CHAPTER

TWO

LIBRARY INITIALIZATION AND CLEANUP

The SAPI library maintains some internal global state that you must initialize and clean up.

2.1 sapi_globalInit

```
sapi_Code sapi_globalInit();
```

SAPI global initialization function.

Note: Call this function before creating connections with the *sapi_localConnection* or *sapi_remoteConnection* functions.

2.1.1 Return Value

SAPI error code

2.2 sapi_globalCleanup

```
void sapi_globalCleanup();
```

SAPI global cleanup function.

Note: Call this function after you are done with all solvers and connections.

CHAPTER

THREE

CONNECTING TO THE SOLVER

3.1 Data Types, Enums and Macros

This section lists the data types, enums, and macros in alphabetical order.

3.1.1 sapi_AnnealOffsetProperties

Note: Annealing offsets are not supported on D-Wave 2X and earlier systems.

```
typedef struct sapi_AnnealOffsetProperties
{
  const sapi_AnnealOffsetRange *ranges;
  size_t ranges_len;
  double step;
  double step_phi0;
} sapi_AnnealOffsetProperties;
```

SAPI anneal offsets properties struct.

Fields:

ranges — Array of ranges of valid anneal offset values, in normalized offset units, for each qubit. The negative values represent the largest number of normalized offset units by which a qubit's anneal path may be delayed. The positive values represent the largest number of normalized offset units by which a qubit's anneal path may be advanced.

ranges_len — Length of the ranges property. This value is the total number of qubits as returned by the num_qubits property.

step — Quantization step size of anneal offset values in normalized units.

step_phi0 — Quantization step size in physical units (annealing flux bias units).

3.1.2 sapi_AnnealOffsetRange

Note: Annealing offsets are not supported on D-Wave 2X and earlier systems.

```
typedef struct sapi_AnnealOffsetRange
{
  double min;
```

```
  double max;
} sapi_AnnealOffsetRange;
```

SAPI anneal offsets struct.

Fields:

min — Negative numeric value representing the largest number of normalized offset units by which a qubit's anneal path may be delayed.

max — Positive numeric value representing the largest number of normalized offset units by which a qubit's anneal path may be advanced.

3.1.3 sapi_AnnealOffsets

Note: Annealing offsets are not supported on D-Wave 2X and earlier systems.

```
typedef struct sapi_AnnealOffsets
{
  double *elements;
  size_t len;
} sapi_AnnealOffsets;
```

SAPI anneal offsets struct.

Fields:

elements — Annealing offset elements.

> Provide an array of annealing offset values, in normalized offset units, for all qubits, working or not. Use 0 for no offset. Negative values produce a negative offset (qubits are annealed *after* the standard annealing trajectory); positive values produce a positive offset (qubits are annealed *before* the standard trajectory). Before using this parameter, query the solver properties to determine whether `anneal_offsets` exists in the `parameters` property. If so, retrieve `anneal_offset_ranges` to obtain the permitted offset values per qubit.
>
> Use the *sapi_getSolverProperties* call to retrieve solver properties. If the solver does not support annealing offsets, the anneal_offset field is NULL. If it does support annealing offsets, the `anneal_offset -> ranges` property shows the range of permitted offset values, per qubit.

len — Length of the elements array. The number of entries in the array must equal the number of qubits (num_qubits).

3.1.4 sapi_Chains

```
typedef struct sapi_Chains
{
    int *elements;
    size_t len;
} sapi_Chains;
```

SAPI chains struct.

elements — Array of integers, the value of elements[i] means chain elements[i] contains qubit i; -1 means a singleton chain.

len — Length of the elements array.

3.1.5 sapi_Code

```
typedef enum sapi_Code
{
    SAPI_OK = 0,
    SAPI_INVALID_PARAMETER,
    SAPI_SOLVE_ERROR,
    SAPI_AUTHENTICATION_ERROR,
    SAPI_NETWORK_ERROR,
    SAPI_COMMUNICATION_ERROR,
    SAPI_ASYNC_NOT_DONE,
    SAPI_PROBLEM_CANCELLED,
    SAPI_OUT_OF_MEMORY
} sapi_Code;
```

SAPI error code enum.

SAPI_OK — Function call is successful, no error occurred.

SAPI_INVALID_PARAMETER — Invalid parameter.

SAPI_SOLVE_ERROR — Error occurred during the solve.

SAPI_AUTHENTICATION_ERROR — Authentication failed.

SAPI_NETWORK_ERROR — Network error occurred.

SAPI_COMMUNICATION_ERROR — Network communication failed.

SAPI_ASYNC_NOT_DONE — Problem not done.

SAPI_PROBLEM_CANCELLED — Problem cancelled.

SAPI_OUT_OF_MEMORY — Not enough memory.

3.1.6 sapi_Connection

```
typedef struct sapi_Connection sapi_Connection;
```

Opaque SAPI connection type.

Returned by *sapi_remoteConnection* or *sapi_localConnection* function.

Note: The *sapi_Connection* pointer that is returned by *sapi_remoteConnection* function needs to be freed by using *sapi_freeConnection* function. The *sapi_Connection* pointer that is returned by *sapi_localConnection* function does not need to be freed by using *sapi_freeConnection* function. It will be automatically freed when *sapi_globalCleanup* function is called.

3.1.7 sapi_Coupler

```
typedef struct sapi_Coupler
{
    int q1;
    int q2;
} sapi_Coupler;
```

SAPI solver property's coupler struct.

Fields:

q1, q2 — [q1, q2] represents a coupler (q1 < q2).

3.1.8 SAPI_ERROR_MESSAGE_MAX_SIZE

```
#define SAPI_ERROR_MESSAGE_MAX_SIZE 512
```

SAPI error message maximum size.

Note: when using dwave_sapi c library function with char *err_msg parameter, need to provide a char buffer whose size is SAPI_ERROR_MESSAGE_MAX_SIZE, i.e., char err_msg[SAPI_ERROR_MESSAGE_MAX_SIZE].

3.1.9 sapi_IsingRangeProperties

```
typedef struct sapi_IsingRangeProperties
{
    double h_min;
    double h_max;
    double j_min;
    double j_max;
} sapi_IsingRangeProperties;
```

SAPI Ising range property struct.

Fields:

h_min — Minimum value h can have.

h_max — Maximum value h can have.

j_min — Minimum value J can have.

j_max — Maximum value J can have.

3.1.10 sapi_IsingResult

```
typedef struct sapi_IsingResult
{
    int *solutions;
    size_t solution_len;
    size_t num_solutions;
    double *energies;
    int *num_occurrences;
    sapi_Timing timing;
} sapi_IsingResult;
```

SAPI Ising/QUBO result struct.

Note: Use *sapi_freeIsingResult* function to free *sapi_IsingResult* pointer.

Fields:

solutions — Array of integers. The solutions contain either -1/1 for Ising problem or 0/1 for QUBO problem. Note that its length is: solution_len * num_solutions. If answer_mode is *SAPI_ANSWER_MODE_HISTOGRAM*, the states (entries) are unique and sorted in increasing-energy order; If answer_mode is *SAPI_ANSWER_MODE_RAW*, all the output states are in the order that they were generated.

solution_len — Length of each solution.

num_solutions — Number of solutions.

energies — Array of doubles representing the corresponding energy for each solution. Note that energies' length is: num_solutions.

num_occurrences — Array of integers representing the number of occurrences for each solution. This field will be NULL if answer_mode is *SAPI_ANSWER_MODE_RAW*. Note that if num_occurrences pointer is not NULL, its length is: num_solutions.

timing — Structure containing the time taken (in microseconds) at each step of the routine such as *qpu_anneal_time_per_sample*, *preprocessing_time*, etc. (Optional, only hardware solvers return the *timing* structure.)

> **Note:** Prior to Release 2.4 of the Solver API, the timing field names were different. For more information about the timing structure, see *Measuring Computation Time on D-Wave Systems,* available for download on the Qubist web interface.

3.1.11 sapi_ParametersProperty

```
typedef struct sapi_ParametersProperty
{
  const char *const *elements;
  size_t len;
} sapi_ParametersProperty;
```

SAPI parameters property struct.

Fields:

elements — Array of valid solver parameter names, sorted in ascending order.

len — Length of the elements array.

3.1.12 sapi_Postprocess

```
typedef enum sapi_Postprocess
{
    SAPI_POSTPROCESS_NONE,
    SAPI_POSTPROCESS_SAMPLING,
    SAPI_POSTPROCESS_OPTIMIZATION
} sapi_Postprocess;
```

SAPI_POSTPROCESS_NONE: no postprocessing.

SAPI_POSTPROCESS_SAMPLING: sampling.

SAPI_POSTPROCESS_OPTIMIZATION: optimization.

3.1.13 sapi_Problem

```
typedef struct sapi_Problem
{
    sapi_ProblemEntry *elements;
    size_t len;
} sapi_Problem;
```

SAPI problem struct.

It is used as a parameter by functions *sapi_solveIsing*, *sapi_solveQubo*, *sapi_asyncSolveIsing*, *sapi_asyncSolveQubo*, *sapi_findEmbedding*, *sapi_getChimeraAdjacency*, *sapi_getHardwareAdjacency* and *sapi_makeQuadratic*.

Note: When user uses functions *sapi_solveIsing*, *sapi_solveQubo*, *sapi_asyncSolveIsing*, *sapi_asyncSolveQubo* and *sapi_findEmbedding*, if they create *sapi_Problem* pointer using memory allocation function (such as malloc), the *sapi_Problem* pointer needs to be freed using the corresponding memory deallocation function (such as free). But *sapi_Problem* pointer returned by function *sapi_getChimeraAdjacency*, *sapi_getHardwareAdjacency* or *sapi_makeQuadratic* needs to be freed by using *sapi_freeProblem* function.

Fields:

elements — Array of *sapi_ProblemEntry* struct.

len — Length of the elements array.

3.1.14 sapi_ProblemEntry

```
typedef struct sapi_ProblemEntry
{
    int i;
    int j;
    double value;
} sapi_ProblemEntry;
```

SAPI problem entry struct.

Note: When i == j, it represents a linear term.

Fields:

i, j — [i, j] represents an edge.

value — Weight of the [i, j] edge.

3.1.15 sapi_ProblemStatus

```
typedef struct sapi_ProblemStatus
{
    char problem_id[64];
    char time_received[64];
    char time_solved[64];
    sapi_SubmittedState state;
    sapi_SubmittedState last_good_state;
```

```
    sapi_RemoteStatus remote_status;
    sapi_Code error_code;
    char error_message[SAPI_ERROR_MESSAGE_MAX_SIZE];
} sapi_ProblemStatus;
```

Status of an asynchronous submitted problem Use *sapi_asyncStatus* to fill in this structure.

Fields:

problem_id — Remote problem ID (null terminated). Will be empty until problem is submitted or if using a local solver.

time_received — Time at which the server received the problem (ISO-8601 format). Will be empty until problem is submitted or if using a local solver.

time_solved — Time at which the problem was completed (ISO-8601 format). Will be empty until problem is completed or if using a local solver.

state — State of the problem, as seen by the client library. One of:

- *SAPI_STATE_SUBMITTING* — Problem is still being submitted
- *SAPI_STATE_SUBMITTED* — Problem has been submitted but isn't done yet
- *SAPI_STATE_DONE* — Problem is done (completed, failed, or cancelled)
- *SAPI_STATE_FAILED* — Network communication error occurred while submitting the problem or checking its status and polling the server has stopped. This does not indicate that solving the problem has failed!
- *SAPI_STATE_RETRYING* — Network communication error occurred but submission/polling is being retried (either automatically or by a call to sapi_asyncRetry)

last_good_state — Last "good" value of state, that is, the last value of state other than *SAPI_STATE_FAILED* or *SAPI_STATE_RETRYING*.

remote_status — Status of the problem as reported by the server. One of:

- *SAPI_STATUS_UNKNOWN* — No server response yet (still submitting)
- *SAPI_STATUS_PENDING* — Problem is waiting in a queue
- *SAPI_STATUS_IN_PROGRESS* — Problem is being solved (or will be solved shortly)
- *SAPI_STATUS_COMPLETED* — Solving succeeded
- *SAPI_STATUS_FAILED* — Solving failed
- *SAPI_STATUS_CANCELED* — Problem cancelled by user

error_code — Error type when in any kind of failed state:

- State is either *SAPI_STATE_RETRYING* or *SAPI_STATE_FAILED*
- Remote_status is either *SAPI_STATUS_FAILED* or *SAPI_STATUS_CANCELED*

Otherwise this value will be *SAPI_OK*.

error_message — Error message when error_code is not *SAPI_OK* (blank otherwise).

This structure isn't meaningful for problems running locally. Field values will be:

- *problem_id* — blank
- *time_received* — blank
- *time_solved* — blank
- *state* — *SAPI_STATE_DONE*

- *last_good_state* — *SAPI_STATE_DONE*
- *remote_status* — *SAPI_STATUS_COMPLETED*
- *error_code* — *SAPI_OK*
- *error_message* — blank

3.1.16 sapi_QuantumSolverParameters

```
typedef struct sapi_QuantumSolverParameters
{
    const int parameter_unique_id;
    int annealing_time;
    sapi_SolverParameterAnswerMode answer_mode;
    int auto_scale;
    double beta;
    const sapi_Chains *chains;
    int max_answers;
    int num_reads;
    int num_spin_reversal_transforms;
    sapi_Postprocess postprocess;
    int programming_thermalization;
    int readout_thermalization;
    const sapi_AnnealOffsets *anneal_offsets;
} sapi_QuantumSolverParameters;
```

SAPI quantum solver parameters struct.

Fields:

parameter_unique_id — Unique ID for *sapi_QuantumSolverParameters* struct. Trying to modify it will cause undefined behaviour. Initialize from SAPI_QUANTUM_SOLVER_DEFAULT_PARAMETERS.

annealing_time — Positive integer that sets the duration (in microseconds) of quantum annealing time. This value populates the *qpu_anneal_time_per_sample* field returned in the *timing* structure. (Must be an integer > 0, default is hardware specific.)

> **Note:** For more information about the timing structure, see *Measuring Computation Time on D-Wave Systems,* available for download on the Qubist web interface.

answer_mode — Indicates whether to return a histogram of answers, sorted in order of energy (*SAPI_ANSWER_MODE_HISTOGRAM*); or to return all answers individually in the order they were read (*SAPI_ANSWER_MODE_RAW*). (must be *SAPI_ANSWER_MODE_HISTOGRAM* or *SAPI_ANSWER_MODE_RAW*, default = *SAPI_ANSWER_MODE_HISTOGRAM*)

auto_scale — Indicates whether h and J values will be rescaled to use as much of the range of h and the range of J as possible, or be used as is. When enabled, h and J values need not lie within the range of h and the range of J (but must still be finite). Must be an integer [0 1]; default is 1 (enabled).

beta — Boltzmann distribution parameter for sampling postprocessing. (Any finite value; default is hardware-specific)

chains — Postprocessing chains. (default = NULL)

max_answers — Maximum number of answers returned from the solver in histogram mode (which sorts the returned states in order of increasing energy); this is the total number of distinct answers. In raw mode, this limits the returned values to the first max_answers of num_reads samples. Thus, in this mode, max_answers should never be more than num_reads. (must be an integer > 0, default = num_reads)

num_reads — Positive integer that indicates the number of states (output solutions) to read from the solver. (must be an integer > 0, default = 1)

num_spin_reversal_transforms — Number of spin-reversal transforms.

> Use this parameter to specify how many spin-reversal transforms to perform on the problem. Valid values range from 0 (do not transform the problem; the default value) to a value equal to but no larger than the *num-reads* specified. If you specify a nonzero value, the system divides the number of reads by the number of spin-reversal transforms to determine how many reads to take for each transform. For example, if the number of reads is 10 and the number of transforms is 2, then 5 reads use the first transform and 5 use the second.

postprocess — Type of postprocessing to enable (default = *SAPI_POSTPROCESS_NONE*).

> **Note:** For problems that use the VFYC solver, postprocessing always runs. As with other solvers, users can choose either sampling or optimization postprocessing; however, if this parameter is left blank for a problem submitted to the VFYC solver, optimization postprocessing runs.

programming_thermalization — Integer that gives the time (in microseconds) to wait after programming the processor in order for it to cool back to base temperature (i.e., post-programming thermalization time). Lower values will speed up solving at the expense of solution quality. (must be an integer > 0, default is hardware specific)

readout_thermalization — Integer that gives the time (in microseconds) to wait after each state is read from the processor in order for it to cool back to base temperature (i.e., post-readout thermalization time). This value contributes to the *qpu_delay_time_per_sample* field returned in the *timing* structure. (Must be an integer; default is system specific: contact dwsupport@dwavesys.com for more information on the default and permitted ranges for your system.)

> **Note:** While still supported in SAPI Release 2.10, the `readout_thermalization` parameter is deprecated and will eventually be removed from the API. Plan code updates accordingly.

anneal_offsets — Amount to offset annealing paths, per qubit. This value populates the fields returned in the *sapi_AnnealOffsets* structure.

> Provide an array of annealing offset values, in normalized offset units, for all qubits, working or not. Use 0 for no offset. Negative values produce a negative offset (qubits are annealed *after* the standard annealing trajectory); positive values produce a positive offset (qubits are annealed *before* the standard trajectory). Before using this parameter, query the solver properties to determine whether `anneal_offsets` exists in the `parameters` property. If so, retrieve `anneal_offset_ranges` to obtain the permitted offset values per qubit.
>
> The `anneal_offset -> ranges` property shows the range of permitted offset values, per qubit, for the solver.

3.1.17 SAPI_QUANTUM_SOLVER_DEFAULT_PARAMETERS

```
extern const sapi_QuantumSolverParameters SAPI_QUANTUM_SOLVER_DEFAULT_PARAMETERS;
```

sapi_QuantumSolverParameters default value.

3.1.18 sapi_QuantumSolverProperties

```
typedef struct sapi_QuantumSolverProperties
{
    int num_qubits;
    const int *qubits;
    size_t qubits_len;
    const sapi_Coupler *couplers;
    size_t couplers_len;
} sapi_QuantumSolverProperties;
```

SAPI quantum solver property struct.

Fields:

num_qubits — Total number of qubits.

qubits — Array of working qubits.

qubits_len — Length of the qubits array.

couplers — Array of working couplers.

couplers_len — Length of the couplers array.

3.1.19 sapi_RemoteStatus

```
typedef enum sapi_RemoteStatus
{
  SAPI_STATUS_UNKNOWN,
  SAPI_STATUS_PENDING,
  SAPI_STATUS_IN_PROGRESS,
  SAPI_STATUS_COMPLETED,
  SAPI_STATUS_FAILED,
  SAPI_STATUS_CANCELED
} sapi_RemoteStatus;
```

See *sapi_ProblemStatus*.

3.1.20 sapi_Solver

```
typedef struct sapi_Solver sapi_Solver;
```

Opaque SAPI solver type.

Returned by *sapi_getSolver* function.

Note: Use *sapi_freeSolver* function to free sapi_Solver pointer.

3.1.21 sapi_SolverParameterAnswerMode

```
typedef enum sapi_SolverParameterAnswerMode
{
    SAPI_ANSWER_MODE_HISTOGRAM,
    SAPI_ANSWER_MODE_RAW
} sapi_SolverParameterAnswerMode;
```

SAPI solver answer mode parameter enum.

SAPI_ANSWER_MODE_HISTOGRAM — Histogram mode, return a histogram of answers, sorted in order of energy.

SAPI_ANSWER_MODE_RAW — Raw mode, return all answers individually in the order they were read.

3.1.22 sapi_SolverParameters

```
typedef struct sapi_SolverParameters
{
    const int parameter_unique_id;
} sapi_SolverParameters;
```

General SAPI solver parameters.

Note: The *sapi_SolverParameters* is not directly used by any dwave_sapi functions (use the parameters listed below for specific solvers instead). It is used by the internal implementation codes.

Fields:

parameter_unique_id — Unique id for *sapi_SolverParameters* struct. Trying to modify it will cause undefined behaviour.

3.1.23 sapi_SolverProperties

```
typedef struct sapi_SolverProperties
{
    const sapi_SupportedProblemTypeProperty *supported_problem_types;
    const sapi_QuantumSolverProperties *quantum_solver;
    const sapi_IsingRangeProperties *ising_ranges;
    const sapi_AnnealOffsetProperties *anneal_offset;
    const sapi_ParametersProperty *parameters;
} sapi_SolverProperties;
```

SAPI solver property struct.

Note: If any property does not exist, it will be a NULL pointer.

Fields:

supported_problem_types — Pointer to sapi_SupportedProblemTypeProperty.

quantum_solver — Pointer to sapi_QuantumSolverProperties.

ising_ranges — Pointer to sapi_IsingRangeProperties.

anneal_offset — Pointer to *sapi_AnnealOffsets*. If the solver does not support annealing offsets, this field is NULL.

> **Note:** Annealing offsets are not supported on D-Wave 2X and earlier systems.

parameters — Pointer to *sapi_ParametersProperty*.

3.1.24 sapi_SubmittedProblem

```
typedef struct sapi_SubmittedProblem sapi_SubmittedProblem;
```

SAPI asynchronous submitted problem.

Note: Use *sapi_freeSubmittedProblem* function to free *sapi_SubmittedProblem* pointer.

3.1.25 sapi_SubmittedState

```
typedef enum sapi_SubmittedState
{
  SAPI_STATE_SUBMITTING,
  SAPI_STATE_SUBMITTED,
  SAPI_STATE_DONE,
  SAPI_STATE_RETRYING,
  SAPI_STATE_FAILED
} sapi_SubmittedState;
```

See *sapi_ProblemStatus*.

3.1.26 sapi_SupportedProblemTypeProperty

```
typedef struct sapi_SupportedProblemTypeProperty
{
    const char * const *elements;
    size_t len;
} sapi_SupportedProblemTypeProperty;
```

SAPI solver supported problem type property struct.

Fields:

elements — Array of strings representing supported problem types.

len — Length of the elements array.

3.1.27 sapi_SwSampleSolverParameters

```
typedef struct sapi_SwSampleSolverParameters
{
    const int parameter_unique_id;
    sapi_SolverParameterAnswerMode answer_mode;
    double beta;
    int max_answers;
    int num_reads;
    int use_random_seed;
    unsigned int random_seed;
} sapi_SwSampleSolverParameters;
```

SAPI local solver (sample) parameters struct.

Fields:

parameter_unique_id — Unique ID for *sapi_SwSampleSolverParameters* struct. Trying to modify it will cause undefined behaviour. Initialize from SAPI_SW_SAMPLE_SOLVER_DEFAULT_PARAMETERS.

answer_mode — Indicates whether to return a histogram of answers, sorted in order of energy (*SAPI_ANSWER_MODE_HISTOGRAM*); or to return all answers individually in the order they were read (*SAPI_ANSWER_MODE_RAW*). (must be *SAPI_ANSWER_MODE_HISTOGRAM* or *SAPI_ANSWER_MODE_RAW*, default = *SAPI_ANSWER_MODE_HISTOGRAM*)

beta — Boltzmann distribution parameter. The unnormalized probability of a sample is proportional to exp(-beta * E) where E is its energy. (Any finite value; default = 3.0)

max_answers — Maximum number of answers returned from the solver in histogram mode (which sorts the returned states in order of increasing energy); this is the total number of distinct answers. In raw mode, this limits the returned values to the first max_answers of num_reads samples. Thus, in this mode, max_answers should never be more than num_reads. (must be an integer > 0, default = num_reads)

num_reads — Positive integer that indicates the number of states (output solutions) to read from the solver in each programming cycle. (must be an integer > 0, default = 1)

use_random_seed — Indicates whether to use the random_seed field or not. (must be an integer [0 1], default = 0)

random_seed — Random number generator seed. When a value is provided, solving the same problem with the same parameters will produce the same results every time. If no value is provided, a time-based seed is selected. (must be an integer >= 0, default is a time-based seed)

3.1.28 SAPI_SW_SAMPLE_SOLVER_DEFAULT_PARAMETERS

```
extern const sapi_SwSampleSolverParameters SAPI_SW_SAMPLE_SOLVER_DEFAULT_PARAMETERS;
```

sapi_SwSampleSolverParameters default value.

3.1.29 sapi_SwOptimizeSolverParameters

```
typedef struct sapi_SwOptimizeSolverParameters
{
    const int parameter_unique_id;
    sapi_SolverParameterAnswerMode answer_mode;
    int max_answers;
    int num_reads;
} sapi_SwOptimizeSolverParameters;
```

sapi local solver (optimize) parameters struct.

Fields:

parameter_unique_id — Unique ID for *sapi_SwOptimizeSolverParameters* structure. Trying to modify it will cause undefined behaviour. Initialize from SAPI_SW_OPTIMIZE_SOLVER_DEFAULT_PARAMETERS.

answer_mode — Indicates whether to return a histogram of answers, sorted in order of energy (*SAPI_ANSWER_MODE_HISTOGRAM*); or to return all answers individually in the order they were read (*SAPI_ANSWER_MODE_RAW*). (must be *SAPI_ANSWER_MODE_HISTOGRAM* or *SAPI_ANSWER_MODE_RAW*, default = *SAPI_ANSWER_MODE_HISTOGRAM*)

max_answers — Maximum number of answers returned from the solver in histogram mode (which sorts the returned states in order of increasing energy); this is the total number of distinct answers. In raw mode, this limits the returned values to the first max_answers of num_reads samples. Thus, in this mode, max_answers should never be more than num_reads. (must be an integer > 0, default = num_reads)

num_reads — Positive integer that indicates the number of states (output solutions) to read from the solver in each programming cycle. (must be an integer > 0, default = 1)

3.1.30 SAPI_SW_OPTIMIZE_SOLVER_DEFAULT_PARAMETERS

```
extern const sapi_SwOptimizeSolverParameters SAPI_SW_OPTIMIZE_SOLVER_DEFAULT_
↪PARAMETERS;
```

sapi_SwOptimizeSolverParameters default value.

3.1.31 sapi_SwHeuristicSolverParameters

```
typedef struct sapi_SwHeuristicSolverParameters
{
    const int parameter_unique_id;
    int iteration_limit;
    double max_bit_flip_prob;
    int max_local_complexity;
    double min_bit_flip_prob;
    int local_stuck_limit;
    int num_perturbed_copies;
    int num_variables;
    int use_random_seed;
    unsigned int random_seed;
    double time_limit_seconds;
} sapi_SwHeuristicSolverParameters;
```

SAPI local solver (heuristic) parameters struct.

Fields:

parameter_unique_id — Unique id for *sapi_SwHeuristicSolverParameters* structure. Trying to modify it will cause undefined behaviour. Initialize from SAPI_SW_HEURISTIC_SOLVER_DEFAULT_PARAMETERS.

iteration_limit — Maximum number of solver iterations. This does not include the initial local search. (must be an integer >= 0, default = 10)

min_bit_flip_prob, *max_bit_flip_prob* — Bit flip probability range. The probability of flipping each bit is constant for each perturbed solution copy but varies across copies. The probabilities used are linearly interpolated between min_bit_flip_prob and max_bit_flip_prob. Larger values allow more exploration of the solution space and easier escapes from local minima but may also discard nearly-optimal solutions. (must be a number [0.0 1.0] and min_bit_flip_prob <= max_bit_flip_prob, default min_bit_flip_prob = 1.0 / 32.0, default max_bit_flip_prob = 1.0 / 8.0)

max_local_complexity — Maximum complexity of subgraphs used during local search. The run time and memory requirements of each step in the local search are exponential in this parameter. Larger values allow larger subgraphs (which can improve solution quality) but require much more time and space. Subgraph "complexity" here means treewidth + 1. (must be an integer > 0, default = 9)

local_stuck_limit — Number of consecutive local search steps that do not improve solution quality to allow before determining a solution to be a local optimum. Larger values produce more thorough local searches but increase run time. (must be an integer > 0, default = 8)

num_perturbed_copies — Number of perturbed solution copies created at each iteration. Run time is linear in this value. (must be an integer > 0, default = 4)

num_variables — Lower bound on the number of variables. This solver can accept problems of arbitrary structure and the size of the solution returned is determined by the maximum variable index in the problem. The size of the solution can be increased by setting this parameter. (must be an integer >= 0, default = 0)

use_random_seed — Indicates whether to use the random_seed field or not. (must be an integer [0 1], default = 0)

random_seed — (Optional.) Random number generator seed. When a value is provided, solving the same problem with the same parameters will produce the same results every time. If no value is provided, a time-based seed is selected. The use of a wall clock-based timeout may in fact cause different results with the same random_seed value. If the same problem is run under different CPU load conditions (or on computers with different performance), the amount of work completed may vary despite the fact that the algorithm is deterministic. If repeatability of results is important, rely on the iteration_limit parameter rather than the time_limit_seconds parameter to set the stopping criterion. (must be an integer >= 0, default is a time-based seed)

time_limit_seconds — Maximum wall clock time in seconds. Actual run times will exceed this value slightly. (must be a number >= 0.0, default = 5.0)

3.1.32 SAPI_SW_HEURISTIC_SOLVER_DEFAULT_PARAMETERS

```
extern const sapi_SwHeuristicSolverParameters SAPI_SW_HEURISTIC_SOLVER_DEFAULT_
↪PARAMETERS;
```

sapi_SwHeuristicSolverParameters default value.

3.1.33 sapi_Timing

```
typedef struct sapi_Timing
{
  long long qpu_access_time;
  long long qpu_programming_time;
  long long qpu_sampling_time;
  long long qpu_anneal_time_per_sample;
  long long qpu_readout_time_per_sample;
  long long qpu_delay_time_per_sample;
  long long total_post_processing_time;
  long long post_processing_overhead_time;

  <-- the following fields were deprecated in Release 2.4 of the Solver API:

  long long run_time_chip;
  long long anneal_time_per_run;
  long long readout_time_per_run;
  long long total_real_time;
} sapi_Timing;
```

SAPI timing entry struct.

Fields:

All times are in microseconds.

qpu_access_time — Total time in the QPU

qpu_programming_time — Time to program the QPU

qpu_sampling_time – Total time for R samples, where R is the number of reads/samples

qpu_anneal_time_per_sample – Time for one anneal

qpu_readout_time_per_sample – Time for one read

qpu_delay_time_per_sample – Rethermalization time between anneals

total_post_processing_time — Total time spent in postprocessing (including energy calculations and histogramming)

post_processing_overhead_time — Part of the total postprocessing time that is not concurrent with QPU

Note: For more information about the timing structure, see *Measuring Computation Time on D-Wave Systems,* available for download on the Qubist web interface.

3.2 sapi_asyncDone

```
int sapi_asyncDone(const sapi_SubmittedProblem *submitted_problem);
```

Check if an asynchronously submitted problem is done. Don't use this function to wait for problems to complete, use *sapi_awaitCompletion* instead.

Note: Once the problem is done, you can retrieve the answer with *sapi_asyncResult* function.

3.2.1 Parameters

submitted_problem — Pointer to *sapi_SubmittedProblem*, returned by *sapi_asyncSolveIsing* or *sapi_asyncSolveQubo* function.

3.2.2 Return Value

Returns 1 if the problem has been solved, 0 if the problem hasn't been solved.

3.3 sapi_asyncResult

```
sapi_Code sapi_asyncResult(const sapi_SubmittedProblem *submitted_problem,
                           sapi_IsingResult **result, char *err_msg);
```

Retrieve the answer from an asynchronously submitted problem.

Note: Attempting to retrieve the answer to a problem that has been cancelled will trigger a *SAPI_PROBLEM_CANCELLED* error. Attempting to retrieve the answer to a problem that is not done will trigger a *SAPI_ASYNC_NOT_DONE* error. Use *sapi_asyncDone* function to check whether or not the problem is done. Use *sapi_freeIsingResult* function to free the result pointer.

3.3.1 Parameters

submitted_problem: — Pointer to *sapi_SubmittedProblem*, returned by *sapi_asyncSolveIsing* or *sapi_asyncSolveQubo* function.

result — Answer to the problem. The format will be identical to answers returned by the synchronous solving functions *sapi_solveIsing* or *sapi_solveQubo*.

err_msg — Error message.

3.3.2 Return Value

SAPI error code.

3.4 sapi_asyncRetry

```
void sapi_asyncRetry(const sapi_SubmittedProblem *submitted_problem);
```

Retry a submitted problem that has encountered a network, communication, or authentication error.

This function has no effect on problems that:

- are still in progress
- have been cancelled
- failed while solving (e.g. due to an invalid parameter)

Its purpose is to recover from intermittent network failures (SAPI_ERR_NETWORK but occasionally SAPI_ERR_COMMUNICATION) without resubmitting a problem that may have completed successfully. It can also recover from authentication errors caused by disabling a token (of course, the token must be re-enabled first).

3.4.1 Parameters

submitted_problem — Problem to retry.

3.5 sapi_asyncSolveIsing

```
sapi_Code sapi_asyncSolveIsing(const sapi_Solver *solver, const sapi_Problem *problem,
                               const sapi_SolverParameters *solver_params,
                               sapi_SubmittedProblem **submitted_problem, char *err_
↪msg);
```

When submitting a large number of problems, it can often take a long time to solve all the problems. *sapi_asyncSolveIsing* lets the user submit Ising problems and continue working on other tasks.

Note: When solver is a local solver, the *sapi_asyncDone* function will return 1 immediately, and the problem is actually solved when the answer is requested when using *sapi_asyncResult*.

Solve an Ising problem asynchronously. The pointer returned by *sapi_asyncSolveIsing* can be used by *sapi_asyncDone* which is used to check if the problem has been solved and *sapi_asyncResult* which retrieves the solution to the

problem, can also be used by *sapi_cancelSubmittedProblem* to cancel the submitted problem, can also be used by *sapi_awaitCompletion* to wait for problems to complete.

Note: Use *sapi_freeSubmittedProblem* function to free the *submitted_problem* pointer.

3.5.1 Parameters

solver — *sapi_Solver* pointer to use to solve the problem..

problem — Pointer to *sapi_Problem*.

Note: The *problem* pointer needs to be valid until the problem is solved since *sapi_asyncSolveIsing* function doesn't copy the problem data.

solver_params — Parameters for *solver*. If the solver is a quantum solver, the *solver_params* must be a pointer to type *sapi_QuantumSolverParameters*; if the solver is a sofware sampling solver, the *solver_params* must be a pointer to type *sapi_SwSampleSolverParameters*; if the solver is a software optimizing solver, the *solver_params* must be a pointer to type *sapi_SwOptimizeSolverParameters*.

submitted_problem — Pointer to a pointer to *sapi_SubmittedProblem*.

err_msg — Error message.

3.5.2 Return Value

SAPI error code.

3.6 sapi_asyncSolveQubo

```
sapi_Code sapi_asyncSolveQubo(const sapi_Solver *solver, const sapi_Problem *problem,
                              const sapi_SolverParameters *solver_params,
                              sapi_SubmittedProblem **submitted_problem, char *err_
↪msg);
```

When submitting a large number of problems, it can often take a long time to solve all the problems. *sapi_asyncSolveQubo* lets the user submit QUBO problems and continue working on other tasks.

Note: When solver is a local solver, the *sapi_asyncDone* function will return 1 immediately, and the problem is actually solved when the answer is requested when using *sapi_asyncResult*.

Solve a QUBO problem asynchronously. The pointer returned by *sapi_asyncSolveQubo* can be used by *sapi_asyncDone* which is used to check if the problem has been solved and *sapi_asyncResult* which retrieves the solution to the problem, can also be used by *sapi_cancelSubmittedProblem* to cancel the submitted problem, can also be used by *sapi_awaitCompletion* to wait for problems to complete.

Note: Use *sapi_freeSubmittedProblem* function to free the *submitted_problem* pointer.

3.6.1 Parameters

solver — *sapi_Solver* pointer to use to solve the problem..

problem — Pointer to *sapi_Problem*.

Note: The *problem* pointer needs to be valid until the problem is solved since *sapi_asyncSolveQubo* function doesn't copy the problem data.

---

solver_params — Parameters for *solver*. If the solver is a quantum solver, the *solver_params* must be a pointer to type *sapi_QuantumSolverParameters*; if the solver is a sofware sampling solver, the *solver_params* must be a pointer to type *sapi_SwSampleSolverParameters*; if the solver is a software optimizing solver, the *solver_params* must be a pointer to type *sapi_SwOptimizeSolverParameters*.

submitted_problem — Pointer to a pointer to *sapi_SubmittedProblem*.

err_msg — Error message.

3.6.2 Return Value

SAPI error code.

3.7 sapi_asyncStatus

```
sapi_Code sapi_asyncStatus(const sapi_SubmittedProblem *submitted_problem,
                           sapi_ProblemStatus *status);
```

Retrieve information about an asynchronously-submitted problem. See *sapi_ProblemStatus* for a description of the information provided.

3.7.1 Parameters

submitted_problem — Problem whose status will be retrieved.

status — Pointer to an existing *sapi_ProblemStatus* structure that will be filled in.

3.7.2 Return Value

SAPI error code. On failure, the *status* structure will not be modified.

3.8 sapi_awaitCompletion

```
int sapi_awaitCompletion(const sapi_SubmittedProblem **submitted_problems,
                         int num_submitted_problems, int min_done, double timeout);
```

Waits for problems to complete.

3.8.1 Parameters

submitted_problems — Array of submitted problems, each of the submitted problems returned by *sapi_asyncSolveIsing* or *sapi_asyncSolveQubo* function.

num_submitted_problems — Length of the submitted_problems array.

min_done — Minimum number of problems that must be completed before returning (without timeout).

timeout — Maximum time to wait (in seconds).

3.8.2 Return Value

Returns 1 if returning because enough problems completed, 0 if returning because of timeout.

3.9 sapi_cancelSubmittedProblem

```
void sapi_cancelSubmittedProblem(sapi_SubmittedProblem *submitted_problem);
```

Cancel a submitted problem. Cancellation is not guaranteed; problems may still complete successfully.

3.9.1 Parameters

submitted_problem — Pointer to *sapi_SubmittedProblem*, returned by the *sapi_asyncSolveIsing* or *sapi_asyncSolveQubo* function.

3.10 sapi_freeConnection

```
void sapi_freeConnection(sapi_Connection *connection);
```

Free *sapi_Connection* pointer.

Note: The sapi_Connection pointer that is returned by *sapi_localConnection* does not need to be freed by using this function. It will be automatically freed when *sapi_globalCleanup* function is called.

3.10.1 Parameters

connection — Returned by the *sapi_remoteConnection* function.

3.11 sapi_freeIsingResult

```
void sapi_freeIsingResult(sapi_IsingResult *result);
```

Free *sapi_IsingResult* pointer.

3.11.1 Parameters

result — Returned by *sapi_solveIsing* or *sapi_solveQubo* or *sapi_asyncResult*

3.12 sapi_freeSolver

```
void sapi_freeSolver(sapi_Solver *solver);
```

Free *sapi_Solver* pointer.

3.12.1 Parameters

solver — Pointer to *sapi_Solver*.

3.13 sapi_freeSubmittedProblem

```
void sapi_freeSubmittedProblem(sapi_SubmittedProblem *submitted_problem);
```

Free *sapi_SubmittedProblem* pointer.

3.13.1 Parameters

submitted_problem — Returned by the *sapi_asyncSolveIsing* or *sapi_asyncSolveQubo* function.

3.14 sapi_getSolver

```
sapi_Solver *sapi_getSolver(const sapi_Connection *connection, const char *solver_
↪name);
```

Creates a solver pointer available through *connection*.

The returned solver pointer can be used by sapi_getSolverProperties to get the properties of the solver. It can also be used by *sapi_solveIsing*, *sapi_solveQubo* to solve Ising/QUBO problems synchronously and *sapi_asyncSolveIsing*, *sapi_asyncSolveQubo* to solve Ising/QUBO problems asynchronously.

Note: Use *sapi_freeSolver* function to free the *sapi_Solver* pointer that this function returns.

3.14.1 Parameters

connection — Pointer to *sapi_Connection* returned by the *sapi_remoteConnection* or *sapi_localConnection* function.

solver_name — String of the requested solver's name. Must be listed in *sapi_listSolvers*.

3.14.2 Return Value

The requested solver pointer.

The function returns NULL if no solver's name is *solver_name*.

3.15 sapi_getSolverProperties

```
const sapi_SolverProperties *sapi_getSolverProperties(const sapi_Solver *solver);
```

Get solver properties.

All solvers have a "supported_problem_types" property whose value is an array of problem type strings.

Note: A complete list of solver-specific properties is in *SAPI Solvers*. The returned *sapi_SolverProperties* pointer is freed automatically when the solver is freed by calling *sapi_freeSolver*.

3.15.1 Parameters

solver — Solver pointer.

3.15.2 Return Value

A *sapi_SolverProperties* pointer.

3.16 sapi_listSolvers

```
const char **sapi_listSolvers(const sapi_Connection *connection);
```

Retrieves all the available solvers in *connection*.

Note: The returned string array will be freed automatically after the *connection* is freed by calling *sapi_freeConnection*.

3.16.1 Parameters

connection: — Pointer to *sapi_Connection* returned by *sapi_remoteConnection* or *sapi_localConnection*.

3.16.2 Return Value

A string array which contains all available solvers' names, the last element will be NULL.

The function returns NULL if some error happens.

3.17 sapi_localConnection

```
sapi_Connection *sapi_localConnection();
```

If you choose to use a local solver instead of a remote solver as in *sapi_remoteConnection*, a connection to the local solver should be established through *sapi_localConnection*.

The returned *sapi_Connection* pointer is used to retrieve available solver names as well as to create a solver pointer which will perform most subsequent communications with the solver. As such, there are two functions that can use *sapi_Connection* pointer as a parameter: *sapi_listSolvers* and *sapi_getSolver*.

Note: The *sapi_Connection* pointer that this function returns does not need to be freed by *sapi_freeConnection* function. It will be automatically freed when *sapi_globalCleanup* function is called.

3.17.1 Return Value

sapi_Connection pointer.

The function returns NULL if some error happens.

3.18 sapi_remoteConnection

```
sapi_Code sapi_remoteConnection(const char *url, const char *token,
                                const char *proxy_url,
                                sapi_Connection **remote_connection, char *err_msg);
```

As a first step, a connection to the solver must be made. The type of solvers available to the user is shown in *Solver tree diagram*.

The user can connect to either a remote solver or a local solver. A remote solver can be a hardware or a software solver. Hardware solver implies the remotely located quantum processor while software solver is a remotely located software solver. Local solver is a software solver running on the local machine of the user. However, local solvers and remote software solvers run the same algorithms. If using a remote solver, a connection to the solver must be established through *sapi_remoteConnection*. A comprehensive description of different solvers that can be used is given in *SAPI Solvers*.

code = sapi_remoteConnection(url, token, NULL, &connection, err_msg) creates a remote connection pointer to the available software solvers and hardware solvers (check your local documentation for the *url* of the solvers) under the token *token* (check your local documentation/User Interface on how to obtain a *token*).

code = sapi_remoteConnection(url, token, proxy_url, &connection, err_msg) creates a remote connection pointer to the available software solvers and hardware solvers using an additional URL *proxy_url* that can be retrieved from your system administrator.

Error conditions will be raised if any of the inputs are invalid. If the code is not *SAPI_OK*, user can print out the error message from *err_msg*.

The *remote_connection* pointer is used to retrieve available solver names as well as to create a solver pointer which will perform most subsequent communications with the solver. As such, there are two functions that can use *remote_connection* as a parameter: *sapi_listSolvers* and *sapi_getSolver*.

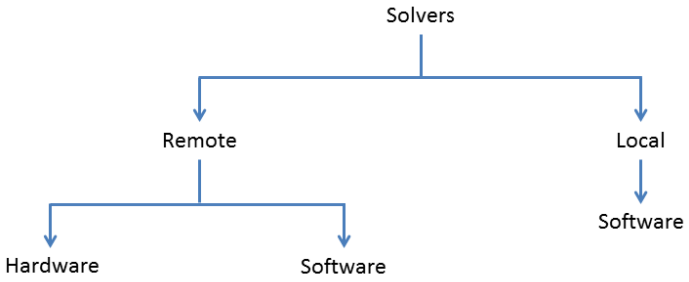

Fig. 3.1: Solver tree diagram

Note: Use *sapi_freeConnection* function to free the *remote_connection* pointer.

3.18.1 Parameters

url — String for URL.

token — String for token.

proxy_url — String for proxy url. (If do not want to use proxy, set proxy_url as NULL.) proxy_url causes requests to go through a proxy. If proxy is given, it must be a string url of proxy. The default is to read the list of proxies from the environment variables <protocol>_proxy. If no proxy environment variables are set, in a Windows environment, proxy settings are obtained from the registry's Internet Settings section and in a Mac OS X environment, proxy information is retrieved from the OS X System Configuration Framework. (To disable autodetected proxy pass an empty string.)

remote_connection — Pointer to a pointer to *sapi_Connection*.

err_msg — Error message.

3.18.2 Return Value

SAPI error code.

3.19 sapi_solveIsing

```
sapi_solveIsing(const sapi_Solver *solver, const sapi_Problem *problem,
                const sapi_SolverParameters *solver_params,
                sapi_IsingResult **result, char *err_msg);
```

Solve an Ising problem synchronously.

Note: Use *sapi_freeIsingResult* function to free the *result* pointer.

3.19.1 Parameters

solver — *sapi_Solver* pointer to use to solve the problem.

problem — Pointer to *sapi_Problem*.

Note: The *problem* pointer needs to be valid until the problem is solved since *sapi_solveIsing* function doesn't copy the problem data.

solver_params — Parameters for *solver*. If the solver is a quantum solver, the *solver_params* must be a pointer to type *sapi_QuantumSolverParameters*; if the solver is a software sampling solver, the *solver_params* must be a pointer to type *sapi_SwSampleSolverParameters*; if the solver is a software optimizing solver, the *solver_params* must be a pointer to type *sapi_SwOptimizeSolverParameters*.

result — Pointer to a pointer to *sapi_IsingResult*.

err_msg — Error message.

3.19.2 Return Value

SAPI error code.

3.20 sapi_solveQubo

```
sapi_solveQubo(const sapi_Solver *solver, const sapi_Problem *problem,
               const sapi_SolverParameters *solver_params,
               sapi_IsingResult **result, char *err_msg);
```

Solve a QUBO problem synchronously.

Note: Use *sapi_freeIsingResult* function to free the *result* pointer.

3.20.1 Parameters

solver — *sapi_Solver* pointer to use to solve the problem.

problem — Pointer to *sapi_Problem*.

Note: The *problem* pointer needs to be valid until the problem is solved since *sapi_solveQubo* function doesn't copy the problem data.

solver_params — Parameters for *solver*. If the solver is a quantum solver, the *solver_params* must be a pointer to type *sapi_QuantumSolverParameters*; if the solver is a sofware sampling solver, the *solver_params* must be a pointer to type *sapi_SwSampleSolverParameters*; if the solver is a software optimizing solver, the *solver_params* must be a pointer to type *sapi_SwOptimizeSolverParameters*.

result — Pointer to a pointer to *sapi_IsingResult*.

err_msg — Error message.

3.20.2 Return Value

SAPI error code.

3.21 sapi_version

```
const char *sapi_version();
```

3.21.1 Return Value

The SAPI version string.

CHAPTER

FOUR

POSTPROCESSING

4.1 Postprocessing Overview

4.1.1 Available Methods

The D-Wave system enables users to run postprocessing optimization and sampling algorithms on solutions obtained through the quantum processing unit (QPU). Postprocessing provides local improvements to these solutions with minimal overhead.

When submitting a problem to the QPU, users choose from:

- No postprocessing (default)
- Optimization postprocessing
- Sampling postprocessing

For optimization problems, the goal is to find the state vector with the lowest energy. For sampling problems, the goal is to produce samples from a specific probability distribution. In both cases, a logical graph structure is defined and embedded into the QPU's Chimera topology. Postprocessing methods are applied to solutions defined on this logical graph structure.

For more information about the postprocessing methods available, see *Postprocessing Methods on D-Wave Systems*.

4.1.2 Trade-offs

Mapping of most real-world problems onto a Chimera graph requires increasing the connectivity on the QPU, which is currently done by introducing the so-called chains: groups of qubits that are strongly coupled together and represent a single problem variable. In the perfect world, when infinite precision on h and J values is available, one could enforce the qubits on the chain to get the same spin by assigning a large enough (problem-dependent) coupling strength between the qubits on the chain. In reality, however, chains could (and in most cases do) break.

When breakage on the chain happens, the corresponding sample could either be thrown away or be mapped to a close feasible state — the state with no breakage. The former choice could lead to wasting a lot of samples before (if ever) a feasible state is achieved. The latter option introduces some overhead on the user side to postprocess the samples. Currently, majority voting on the chain is performed to map the broken chains to their closest (in terms of Hamming distance) feasible state.

Moreover, for optimization problems, where we are looking for the global optima or at least good local optima, states that are not locally optimum are not interesting before further postprocessings. The simplest postprocessing to map a non-locally optimum state to a candidate solution is to run a local search to find a close local optimum state. In practice, some of the samples returned by the QPU are not locally optimum and like broken chains leave two options to treat them, discard them, or run a local search to fix them. Again, the trade-off is between spending some time sampling new states or spending some time running local search on such samples.

4.2 Parameters

Use the following fields of the *sapi_QuantumSolverParameters* struct to control postprocessing:

- *beta*
- *chains*
- *postprocess*

See also:

sapi_QuantumSolverParameters

CHAPTER

FIVE

SIMPLIFYING OPTIMIZATION PROBLEMS

For some Ising/QUBO problems, we can infer (in polynomial time) the value that certain variables take in the lowest energy states. If certain variables can be set, this reduces the size of the problem that needs to be sent to the processor, and may help alleviate precision problems. For certain problems (submodular problems where all $J_{ij} < 0$ or $Q_{ij} < 0$) all variables may be inferred. For other problems, no variables may be inferred.

5.1 Types and Enums

5.1.1 sapi_FixVariablesMethod

```
typedef enum sapi_FixVariablesMethod
{
    SAPI_FIX_VARIABLES_METHOD_OPTIMIZED,
    SAPI_FIX_VARIABLES_METHOD_STANDARD
} sapi_FixVariablesMethod;
```

SAPI fix variables method enum.

SAPI_FIX_VARIABLES_METHOD_OPTIMIZED: uses roof-duality & strongly connected components.

SAPI_FIX_VARIABLES_METHOD_STANDARD: uses roof-duality only.

Fix variables function uses maximum flow in the implication network to correctly fix variables (that is, one can find an assignment for the other variables that attains the optimal value). The variables that roof duality fixes will take the same values in all optimal solutions.

Using strongly connected components can fix more variables, but in some optimal solutions these variables may take different values.

In summary:

- All the variables fixed by *SAPI_FIX_VARIABLES_METHOD_STANDARD* will also be fixed by *SAPI_FIX_VARIABLES_METHOD_OPTIMIZED* (reverse is not true).
- All the variables fixed by *SAPI_FIX_VARIABLES_METHOD_STANDARD* will take the same value in every optimal solution.
- There exists at least one optimal solution that has the fixed values as given by *SAPI_FIX_VARIABLES_METHOD_OPTIMIZED*.

Thus, *SAPI_FIX_VARIABLES_METHOD_STANDARD* is a subset of *SAPI_FIX_VARIABLES_METHOD_OPTIMIZED* as any variable that is fixed by *SAPI_FIX_VARIABLES_METHOD_STANDARD* will also be fixed by *SAPI_FIX_VARIABLES_METHOD_OPTIMIZED* and additionally, *SAPI_FIX_VARIABLES_METHOD_OPTIMIZED* may fix some variables that *SAPI_FIX_VARIABLES_METHOD_STANDARD* could not. For

this reason, *SAPI_FIX_VARIABLES_METHOD_OPTIMIZED* takes longer than *SAPI_FIX_VARIABLES_METHOD_STANDARD*.

5.1.2 sapi_FixedVariable

```
typedef struct sapi_FixedVariable {
  int var;
  int value;
} sapi_FixedVariable;
```

Represents a single fixed variable. Variable index *var* is fixed to value *value*.

5.1.3 sapi_FixVariablesResult

```
typedef struct sapi_FixVariablesResult
{
  sapi_FixedVariable *fixed_variables;
  size_t fixed_variables_len;
  double offset;
  sapi_Problem new_problem;
} sapi_FixVariablesResult;
```

- *fixed_variables*: array of fixed variables.
- *fixed_variables_len*: length of *fixed_variables*.
- *offset*: energy difference from *new_problem* to the original problem.
- *new_problem*: simplified problem. No fixed variables appear in it.

5.2 sapi_fixVariables

```
sapi_Code sapi_fixVariables(const sapi_Problem *problem,
                            sapi_FixVariablesMethod method,
                            sapi_FixVariablesResult **result,
                            char *err_msg);
```

Fix variables for a QUBO problem.

Note: Use *sapi_freeFixVariablesResult* function to free the result pointer.

5.2.1 Parameters

problem: pointer to a QUBO problem.

method: the method for fix variables algorithms, refer to sapi_FixVariablesMethod.

result: output parameter; **result* will point to a newly-created *sapi_FixVariablesResult*.

err_msg: error message.

5.2.2 Return Value

SAPI error code.

5.3 sapi_freeFixVariablesResult

```
void sapi_freeFixVariablesResult(sapi_FixVariablesResult* result);
```

5.3.1 Parameters

result: fix variables result returned by *sapi_fixVariables*.

CHAPTER

SIX

SOLVING NON-CHIMERA STRUCTURED PROBLEMS

In this section, we consider Ising/QUBO problems with variable interactions that do not match those of the current working Chimera graph. It is assumed that:

1. These problems have fewer variables than the current working Chimera graph, and
2. The user provides an *embedding* of the problem variables into the current working Chimera graph.

Suppose that $G = (V, E)$ is the current working Chimera graph, where V is the set of vertices (i.e., working qubits) and E is the set edges (i.e., working couplers). Consider the Ising problem P defined by $min_x(h' \times x + x' \times J \times x)$, where the dimension of x (i.e., the number of variables) is t. We assume that $t \leq |V|$. Our goal is to define a problem in the current working Chimera graph whose solution will result in a solution to the original problem P.

An *embedding* of the Ising problem P into graph G is a mapping that assigns to each variable x_i a subset of nodes $T(x_i) \subset V$ of G such that:

- The subsets $T(x_i)$ are disjoint, that is, $T(x_i) \cap T(x_j) = \emptyset$ for $i \neq j$,
- For each i the subset $T(x_i)$ is connected (usually a path),
- If there is an interaction between x_i and x_j, that is, $J_{ij} \neq 0$, then there is at least one edge $e \in E$ (i.e., a working coupler) between the subsets $T(x_i)$ and $T(x_j)$.

The next step is to solve the problem defined by h_0 and a J_0 in the Chimera graph. However, we must also make sure that for each i, all the qubits in $T(x_i)$ are aligned (i.e., all the variables in $T(x_i)$ take the same value). We enforce these constraints by penalizing the qubit configurations that violate them using a penalty term JF_m (called *ferromagnetic coupling*). Essentially we use the following. Suppose that qubits q_a and q_b have to take the same value and have a common coupler. We can accomplish this by setting $(JF_m)_{ab}$ to be -1 and scaling up JF_m as necessary.

Finally, to solve the original problem P, we solve $\min_y(h'_0 \times y + y' \times J_0 \times y + \lambda\ y' \times JF_m \times y)$ in the hardware graph, where $\lambda > 0$ needs to be adjusted to make sure that all the qubit constraints are satisfied.

6.1 Types

6.1.1 sapi_EmbedProblemResult

```
typedef struct sapi_EmbedProblemResult
{
  sapi_Problem problem;
  sapi_Problem jc;
  sapi_Embeddings embeddings;
} sapi_EmbedProblemResult;
```

SAPI embed problem result.

Fields:

problem: embedded original problem.

jc: chain edges, i.e. J values coupling vertices representing the same logical variable.

embeddings: original embeddings, possibly modified by cleaning or smearing.

6.1.2 sapi_BrokenChains

```
typedef enum sapi_BrokenChains
{
    SAPI_BROKEN_CHAINS_MINIMIZE_ENERGY,
    SAPI_BROKEN_CHAINS_VOTE,
    SAPI_BROKEN_CHAINS_DISCARD,
    SAPI_BROKEN_CHAINS_WEIGHTED_RANDOM
} sapi_BrokenChains;
```

SAPI unembed answer broken chains enum.

SAPI_BROKEN_CHAINS_MINIMIZE_ENERGY: greedy descent on original problem. Intact chains aren't changed. Minimize in variable order (default).

SAPI_BROKEN_CHAINS_VOTE: use majority value of each chain. In case of tie, choose -1/1 with equal probability.

SAPI_BROKEN_CHAINS_DISCARD: discard any answer with broken chains.

SAPI_BROKEN_CHAINS_WEIGHTED_RANDOM: choose value randomly, with P(+1) = number of +1 in chain / chain size.

6.1.3 sapi_FindEmbeddingParameters

```
typedef struct sapi_FindEmbeddingParameters
{
    int fast_embedding;
    int max_no_improvement;
    int use_random_seed;
    unsigned int random_seed;
    double timeout;
    int tries;
    int verbose;
} sapi_FindEmbeddingParameters;
```

SAPI find embedding parameters struct.

Fields:

fast_embedding: tries to get an embedding quickly, without worrying about chain length. (must be an integer [0 1], default = 0)

max_no_improvement: number of rounds of the algorithm to try from the current solution with no improvement. Each round consists of an attempt to find an embedding for each variable of S such that it is adjacent to all its neighbours. (must be an integer >= 0, default = 10)

use_random_seed: indicates whether to use the random_seed field or not. (must be an integer [0 1], default = 0)

random_seed: seed for random number generator that *sapi_findEmbedding* uses. (must be an integer >= 0, default is a time-based seed)

timeout: the algorithm gives up after timeout seconds. (must be a number >= 0.0, default is approximately 1000.0 seconds)

tries: the algorithm stops after this number of restart attempts. (must be an integer >= 0, default = 10)

Note: The algorithm stops when either of *timeout* or *tries* is reached, whichever comes first.

verbose: control the output information. (must be an integer [0 1], default = 0) When verbose is 1, the output information will be like:

component ..., try ...:

max overfill = ..., num max overfills = ...

Embedding found. Minimizing chains...

max chain size = ..., num max chains = ..., qubits used = ...

Detailed explanation of the output information:

- "component": process ith (0-based) component, the algorithm tries to embed larger strongly connected components first, then smaller components
- "try": jth (0-based) try
- "max overfill": largest number of variables represented in a qubit
- "num max overfills": the number of qubits that has max overfill
- "max chain size": largest number of qubits representing a single variable
- "num max chains": the number of variables that has max chain size
- "qubits used": the total number of qubits used to represent variables

6.1.4 SAPI_FIND_EMBEDDING_DEFAULT_PARAMETERS

```
extern const sapi_FindEmbeddingParameters SAPI_FIND_EMBEDDING_DEFAULT_PARAMETERS;
```

sapi_FindEmbeddingParameters default value.

6.1.5 sapi_Embeddings

```
typedef struct sapi_Embeddings
{
    int *elements;
    size_t len;
} sapi_Embeddings;
```

SAPI embeddings struct.

Fields:

elements: an array of integers, the value of elements[i] means logical variable elements[i] maps to physical qubit i; -1 means the physical qubit i is not used for embedding.

len: the length of the elements array.

6.2 sapi_findEmbedding

```
sapi_Code sapi_findEmbedding(const sapi_Problem *S, const sapi_Problem *A,
                             const sapi_FindEmbeddingParameters *find_embedding_
↪params,
                             sapi_Embeddings **embeddings, char *err_msg);
```

Attempts to find an embedding of a Ising/QUBO problem in a graph. This function is entirely heuristic: failure to return an embedding does not prove that no embedding exists.

Note: Use *sapi_freeEmbeddings* function to free the embeddings pointer.

6.2.1 Parameters

S: edge structures of a problem, can be Ising/QUBO. The embedder only cares about the edge structure (i.e. which variables have nontrivial interactions), not the coefficient values.

A: an adjacency matrix of the graph.

find_embedding_params: parameters for find embedding algorithm. Must be a pointer to type *sapi_FindEmbeddingParameters*.

embeddings: a pointer to a pointer to *sapi_Embeddings*.

err_msg: error message.

6.2.2 Return Value

SAPI error code.

6.3 sapi_embedProblem

```
sapi_Code sapi_embedProblem(const sapi_Problem *problem, const sapi_Embeddings␣
↪*embeddings,
                            const sapi_Problem *adj, int clean, int smear,
                            const sapi_IsingRangeProperties *ranges,
                            sapi_EmbedProblemResult **result, char *err_msg);
```

Embed problem.

6.3.1 Parameters

problem: must be an Ising problem.

embeddings: embeddings that user provides or returned by *sapi_findEmbedding* function. It defines the embedding of the problem structure into Chimera. Each array gives the qubits that the variables in the Ising problem are embedded to.

adj: adjacency matrix of target graph. Uses sapi_Problem data structure, but only the i and j fields of each sapi_ProblemEntry are used (value is ignored).

clean: boolean value. "Cleaning" an embedding means iteratively removing vertices that are: * adjacent to a single vertex representing the same variable * not adjacent to any other embedded variables

smear: boolean value. "Smearing" an embedding means attempts to lower the scale of h values compared to those of J values (relative to their respective ranges) by adding more vertices to variables with large h values. Smearing is done after cleaning, so it is potentially useful to do both.

ranges: h and J ranges. Only used when smearing is enabled. May be NULL, in which case both ranges are assumed to be [-1, 1]

result: embedded problem and possibly modified embedding. Returned embedding will only differ from passed embedding if cleaning or smearing is enabled.

err_msg: a buffer of size at least SAPI_ERROR_MESSAGE_MAX_SIZE. Error message will be copied here if the function fails. May be NULL.

6.3.2 Return Value

SAPI error code.

6.4 sapi_unembedAnswer

```
sapi_Code sapi_unembedAnswer(const int *solutions, size_t solution_len,
                             size_t num_solutions, const sapi_Embeddings *embeddings,
                             sapi_BrokenChains broken_chains, const sapi_Problem␣
↪*problem,
                             int *new_solutions, size_t *num_new_solutions,
                             char *err_msg);
```

6.4.1 Parameters

solutions, *solution_len*, *num_solutions*: embedded solutions, same as corresponding fields in sapi_IsingResult structure.

embeddings: same embeddings as used to embed the original problem. Note: if you used sapi_embedProblem with cleaning or smearing enabled, be sure to use the returned embeddings.

broken_chains: strategy for repairing broken chains. The solution bits representing a single variable (a "chain") may not agree. This parameter controls how the value is chosen.

problem: original problem before embedding. Required when broken_chains is SAPI_BROKEN_CHAINS_MINIMIZE_ENERGY, ignored otherwise.

new_solutions: array of size num_solutions * (# of original variables) that will be filled with unembedded solutions.

num_new_solutions: output value that will be set to the number of new solutions. This will equal num_solutions unless broken_chains is SAPI_BROKEN_CHAINS_DISCARD, in which case it could be less.

err_msg: a buffer of size at least SAPI_ERROR_MESSAGE_MAX_SIZE. Error message will be copied here if the function fails. May be NULL.

6.4.2 Return Value

SAPI error code.

6.5 sapi_getChimeraAdjacency

```
sapi_Code sapi_getChimeraAdjacency(int M, int N, int L, sapi_Problem **A);
```

Build the adjacency matrix for the Chimera architecture. The architecture is an M-by-N lattice of elements where each element is a K_{L,L} bipartite graph.

Note: Use *sapi_freeProblem* function to free the A pointer.

6.5.1 Parameters

M, N, L: Chimera dimensions.

A: adjacency structure.

6.5.2 Return Value

SAPI error code.

6.6 sapi_getHardwareAdjacency

```
sapi_Code sapi_getHardwareAdjacency(const sapi_Solver *solver, sapi_Problem **A);
```

Build the adjacency matrix for the solver.

Note: Use *sapi_freeProblem* function to free the A pointer.

6.6.1 Parameters

solver: a solver pointer.

A: adjacency structure.

6.6.2 Return Value

SAPI error code.

6.7 sapi_freeProblem

```
void sapi_freeProblem(sapi_Problem *problem);
```

Free *sapi_Problem* pointer.

6.7.1 Parameters

problem: a *sapi_Problem* pointer returned by *sapi_getChimeraAdjacency*, *sapi_getHardwareAdjacency* or *sapi_makeQuadratic*.

6.8 sapi_freeEmbeddings

```
void sapi_freeEmbeddings(sapi_Embeddings *embeddings)
```

Free *sapi_Embeddings* pointer.

6.8.1 Parameters

embeddings: a *sapi_Embeddings* pointer returned by the *sapi_findEmbedding* function.

6.9 sapi_freeEmbedProblemResult

```
void sapi_freeEmbedProblemResult(sapi_EmbedProblemResult *result);
```

Free *sapi_EmbedProblemResult* pointer.

6.9.1 Parameters

result: embed problem result.

CHAPTER

SEVEN

REDUCING ORDER INTERACTION

Many problems involve interactions between groups of 3 or more variables, and thus, cannot be directly modeled within the Ising/QBOS model due to limitations of those models to pairwise interactions. Functions having higher-order interactions are conveniently represented as a weighted sum of products of literals, for example, $f(x_0, x_1, x_2, x_3) = 5x_0x_1 - 3x_1x_2x_3$. Each product of variables is called a term. A function expressed in this manner can be stored in the computer as a list of terms where each term is itself a list of the variables in the product. The weighting of each term can be stored as a separate list, e.g. $[5, -3]$. For example, the terms in the above function are t_0 and t_1 where $t_0 = 0, 1$ and $t_1 = 1, 2, 3$ indicate that x_0 and x_1 appear in term t_0, and x_1, x_2, x_3 appear in term t_1. The function f cannot be represented in hardware because of the third-order interactions in term t_1. However, by introducing a new variable $y = x_1x_2$ we can write f as $5x_0x_1 - 3yx_3$. Fortunately, the constraint $y = x_1x_2$ can be represented using a penalty function $P(x_1, x_2; y)$ having only pairwise interactions so that $\tilde{f}(x_0, x_1, x_2, x_3, y) = 5x_0x_1 - 3yx_3 + P(x_1, x_2; y)$ when minimized over y represents f. The routines in this chapter facilitate these kinds of reductions to QUBOs.

7.1 Types

7.1.1 sapi_TermsEntry

```
typedef struct sapi_TermsEntry
{
    int *terms;
    size_t len;
} sapi_TermsEntry;
```

SAPI terms entry struct.

Fields:

terms: an array of term. Must only contain non negative integers.

len: the length of the terms array.

7.1.2 sapi_Terms

```
typedef struct sapi_Terms
{
    sapi_TermsEntry *elements;
    size_t len;
} sapi_Terms;
```

SAPI terms struct. Returned by *sapi_reduceDegree* or *sapi_makeQuadratic* function.

Note: Use *sapi_freeTerms* function to free sapi_Terms pointer.

Fields:

elements: an array of *sapi_TermEntry* struct.

len: the length of the elements array.

7.1.3 sapi_VariablesRepEntry

```
typedef struct sapi_VariablesRepEntry
{
    int variable;
    int rep[2];
} sapi_VariablesRepEntry;
```

SAPI variables rep entry struct.

Fields:

variable: the newly introduced variable v.

rep: an array which contains two variables x, y so that v = x * y.

7.1.4 sapi_VariablesRep

```
typedef struct sapi_VariablesRep
{
    sapi_VariablesRepEntry *elements;
    size_t len;
} sapi_VariablesRep;
```

SAPI variables rep struct. Returned by *sapi_reduceDegree* or *sapi_makeQuadratic* function.

Note: Use *sapi_freeVariablesRep* function to free sapi_VariablesRep pointer.

Fields:

elements: an array of *sapi_VariablesRepEntry* struct.

len: the length of the elements array.

7.2 sapi_reduceDegree

```
sapi_Code sapi_reduceDegree(const sapi_Terms *terms, sapi_Terms **new_terms,
                            sapi_VariablesRep **variables_rep, char *err_msg);
```

Reduce the degree of a set of objectives specified by terms to have maximum two degrees via the introduction of ancillary variables.

Note: Use *sapi_freeTerms* function to free the *new_terms* pointer; use *sapi_freeVariablesRep* function to free the *variables_rep* pointer.

7.2.1 Parameters

terms: each term's variables in the expression, the index in terms must be a non-negative integer.

new_terms: terms after using ancillary variables.

variables_rep: ancillary variables.

err_msg: error message.

7.2.2 Return Value

SAPI error code.

The returned *variables_rep* indicates that the routine has introduced new variables numbered 13 to 21 and for example, variable 13 is the product of variable x_1 and x_6. *new_terms* are the terms in the new variables. No term in new terms will have more than pair-wise interactions. For example,

$new\ term_0 = Term_{17} \times Term_{19}$
$Term_{17} = Term_{14} \times Term_{15}$
$Term_{19} = x_4 x_8$
$Term_{14} = x_0 x_2$
$Term_{15} = x_3 x_5$; therefore
$Term_{17} = x_0 x_2 x_3 x_5$; therefore
$new\ term_0 = x_0 x_2 x_3 x_4 x_5 x_8$

Therefore, as can be seen from above, the non-pair-wise term in the original problem has been replaced with a new pair-wise term comprising the product of $Term_{17}$ and $Term_{19}$.

7.3 sapi_makeQuadratic

```
sapi_Code sapi_makeQuadratic(const double *f, int f_len, const double *penalty_weight,
                             sapi_Terms **new_terms, sapi_VariablesRep **variables_
↪rep,
                             sapi_Problem **Q, char *err_msg);
```

If an objective function f is represented explicitly as a vector of numbers (e.g. [f_{000}, f_{001}, f_{010}, f_{011}, f_{100}, ..., f_{111}]), we may not know the representation as sums of terms. *sapi_makeQuadratic* function performs similar optimization as in *sapi_reduceDegree* when the input is a vector of numbers instead of an array of terms. For example, if we define a problem with 3 variables x_0, x_1 and x_2, then f_{000} represents the value of f with $x_0 = 0$, $x_1 = 0$ and $x_2 = 0$. Similarly, the second term of f (f_{001}) will have $x_0 = 1$, $x_1 = 0$ and $x_2 = 0$. Then, for a problem defined as $f = 8x_0x_2 - x_0x_1x_2$:

f_{000}	f_{001}	f_{010}	f_{011}	f_{100}	f_{101}	f_{110}	f_{111}
0	0	0	0	0	8	0	7

The function looks at the length of the problem submitted and determines if the length is equal to a power of 2. If it is so, it then acts similar to *sapi_reduceDegree* by replacing the variables with equivalent pair-wise interactions.

sapi_makeQuadratic takes an explicit function indicated by the *f*, and generates an equivalent QUBO representation specified by the *Q*.

Note: Use *sapi_freeTerms* function to free the *new_terms* pointer; use *sapi_freeVariablesRep* function to free the *variables_rep* pointer; use *sapi_freeProblem* function to free the *Q* pointer.

7.3.1 Parameters

f: a function defined over binary variables represented as an array stored in decimal order.

f_len: *f* 's length, must be power of 2.

penalty_weight: the strength of the penalty used to define the product constraints on the new ancillary variables. Set it as NULL if want to use the default value, the default value is usually sufficiently large, but may be larger than necessary.

new_terms: the terms in the QUBO arising from quadraticization of the interactions present in f.

variables_rep: the definition of the new ancillary variables.

Q: quadratic coefficients.

err_msg: error message.

7.3.2 Return Value

SAPI error code.

Note: The length of f has to be a power of 2. i.e., the length of f is 2^m where m is the number of variables in the given problem.

7.4 sapi_freeTerms

```
void sapi_freeTerms(sapi_Terms *terms)
```

Free *sapi_Terms* pointer.

7.4.1 Parameters

terms: returned by *sapi_reduceDegree* or *sapi_makeQuadratic*

7.5 sapi_freeVariablesRep

```
void sapi_freeVariablesRep(sapi_VariablesRep *variables_rep)
```

Free *sapi_VariablesRep* pointer.

7.5.1 Parameters

variables_rep: returned by *sapi_reduceDegree* or *sapi_makeQuadratic*

CHAPTER

EIGHT

QSAGE

This chapter describes the QSage algorithm and how it can help achieve better results for problems submitted to the D-Wave QPU. At the end of the chapter is a description of the related C data types and functions.

8.1 Motivation

The restricted connectivity between qubits limits the ability to directly solve arbitrarily structured problems. To solve a problem directly in the QPU if there is an interaction between problem variables s_1 and s_2, then a physical connection (edge) must exist between the qubits representing the values of these variables. For most problems, the interactions between variables do not match the qubit connectivity. This limitation can be circumvented using embedding. However, this solution requires you to find an embedding or mapping of problem variables to qubits. Finding such embeddings is itself a hard optimization problem.

Moreover, the native D-Wave QPU is limited to the minimization of Ising or QUBO objective functions:

$$\text{ising:} \qquad \boldsymbol{s}^\star = \underset{\boldsymbol{s}}{\operatorname{argmin}}\, E(\boldsymbol{s}|\boldsymbol{h},\boldsymbol{J}) = \underset{\boldsymbol{s}}{\operatorname{argmin}} \left\{ \sum_{i\in V} h_i s_i + \sum_{(i,j)\in E} s_i J_{i,j} s_j \right\} \qquad s_i \in \{-1,+1\}$$

$$\text{qubo:} \qquad \boldsymbol{x}^\star = \underset{\boldsymbol{x}}{\operatorname{argmin}}\, E(\boldsymbol{x}|\boldsymbol{Q}) = \underset{\boldsymbol{x}}{\operatorname{argmin}} \left\{ \sum_{i\in V} Q_{i,i} x_i + \sum_{(i,j)\in E} x_i Q_{i,j} x_j \right\} \qquad x_i \in \{0,1\}$$

A graph with edge set E defines the allowed interactions between variables. This functional form is restricting in two ways. First, your problem may involve interactions between more than pairs of variables. Though this problem can be addressed using the methods of reducing higher-order interactions, this costs qubits, and requires additional programming. Second, the function you want to minimize may not have a mathematical description. The objective function you want to minimize may be represented as a computer program, which when input with a bit string, returns a number representing the value of the objective function. The optimization of problems not expressible mathematically in terms of $\boldsymbol{h}$ and $\boldsymbol{J}$ is not possible directly on the D-Wave 2X QPU.

In this chapter, we show how all of these problems can be addressed using a method called *QSage*, which relies on quantum annealing to heuristically minimize (*i.e.* minimize without a guarantee of optimality) arbitrary objective functions. The method can be applied to any objective function. As a user, all you need to do is supply an objective function that returns the objective value of any configuration $\boldsymbol{s}$. We will give an overview of how the QSage method works and provide a description of the parameters needed by the function. In cases where the objective function that you wish to minimize is computationally expensive we also show how the QSage method allows for parallelization across function evaluations.

It is important to stress that even though the QSage method can be applied to any objective function, we cannot expect good results on all problems. Due to the generality of the method, an optimization expert who studies the details of a particular problem is likely to develop a better, specifically-tailored optimization approach. Our goal is to provide a method that will typically yield good results, and that only requires the user to code an objective function evaluator.

As you define your objective functions to be solved by the QSage method, take note that there are many ways to do this, and that some objectives may be easier to solve using this method than others. While there is as much art as science in crafting objective functions, we offer one important guiding principle:

Hint: As much as possible keep your objective function smooth so that small changes in input cause small changes in objective value.

Smooth objectives result in fewer local optima and make the problem more solvable by quantum annealing. If you can think of multiple ways to represent your optimization task it may be worthwhile coding many of them.

In the remainder of this chapter we provide an overview describing how the QSage optimizer works and its interface.

8.2 Algorithm Overview

The QSage method attempts to minimize an objective function $G(\boldsymbol{s})$ defined over array $\boldsymbol{s}$ of length n consisting of -1/1 or 0/1 values. We assume that the user has written code which evaluates $G(\boldsymbol{s})$, and returns a number indicating the objective value. The QSage method attempts to find the configuration $\boldsymbol{s}$ that has the lowest objective value. Note that number of variables n may be larger than, smaller than, or equal to the number of qubits available in the QPU. If n is larger than the number of qubits then the larger problem is solved by large neighbourhood local search *[Ahuja2000]* whereby random subsets of problem variables are optimized in the context of fixed values of the remaining variables. If n is smaller than the number of qubits then the extra qubits are exploited to create additional edge interactions between variables through embedding. Since QSage is unaware of any problem structure in the objective the additional edge interactions are assigned randomly, and fixed through the course of the algorithm.

At the highest level, QSage works by extending a very successful heuristic, tabu search, for discrete optimization problems. Tabu is a local search algorithm. In tabu search an initially random configuration $\boldsymbol{s}_0$ is perturbed to generate many different, but similar, variants. Amongst these variants we hope to discover a new configuration having an objective value lower than $G(\boldsymbol{s}_0)$. Most commonly, the variants of a configuration $\boldsymbol{s}_0$ are obtained by flipping the sign of one of the n spin values. In this way, we generate n variants with each variant differing from $\boldsymbol{s}_0$ in a single spin variable. Like all local search algorithms tabu search usually adopts one of the variants if it has a lower objective value. This new and improved configuration becomes the new configuration $\boldsymbol{s}_1$. We now iterate this procedure and look at all the variants of $\boldsymbol{s}_1$ in the hopes of finding yet another improvement.

As the algorithm runs we accumulate improvements, but eventually this iterative improvement becomes stuck after t steps in a configuration $\boldsymbol{s}_t$ that has lower objective than all of its variants. Such a configuration is called a *local minimum* as $\boldsymbol{s}_t$ is lower than all the locally perturbed variants. To make further progress, configurations that worsen the objective value must be adopted. However, there is no point in moving to a new configuration $\boldsymbol{s}_{t+1}$ only at the next time step $t + 2$ to move right back to the previously considered configuration $\boldsymbol{s}_t$.

Tabu search adopts a simple, but effective, short-term memory strategy to prevent such behaviour. Every time a bit is flipped when moving from configuration $\boldsymbol{s}_t$ to $\boldsymbol{s}_{t+1}$ the flipped bit is marked as tabu which indicates that it cannot be altered again until after a certain number of iterations have elapsed. A parameter called the tabu tenure determines this length of time. The QSage algorithm automatically sets this parameter to a reasonable value. This tabu mechanism prevents the algorithm from rapidly returning to configurations it has already visited. The tabu tenure affects the adoption of new configurations with a simple new rule. Amongst all the variants generated around a particular configuration $\boldsymbol{s}$ we adopt the move to the variant which has the lowest objective value *and* which is not prevented by a tabu restriction. We note that this move to the lowest non-tabued variant may increase the objective value. This mechanism allows for escape from local minima in order to explore new and potentially more fruitful regions. Further details regarding refinements of tabu search not discussed here can be found in *[Glover90]*.

While often effective, tabu search can be slow to explore the search space. Moreover, the optimal setting of the tabu tenure depends on the representation of the problem being solved and optimization performance can be quite sensitive to the tenure setting.

Here we improve upon tabu search by generating additional, and hopefully promising, targeted variants beyond the single-bit-flip variants of standard tabu. These additional variants are added to the pool of single-bit-flip variants and the best (lowest objective value) non-tabued variant within this larger pool is selected as the next candidate configuration.

Expressed as pseudocode, the important high-level steps of the algorithm are as follows:

1. Create a random initial configuration and determine its objective value.
2. Initialize the tabu tenure of all bits to 0.
3. While an outer loop termination condition is not met.
   (a) Generate all single bit flips of the current configuration.
   (b) Generate additional targeted variants of the current configuration by building a hardware-compatible surrogate model and sampling low energy configurations of the model.
   (c) Evaluate the objective value of all variants.
   (d) Update the current configuration to the variant with lowest objective value and whose bit-flips are not tabued.
   (e) Update the tabu list by setting the tabu tenure of the just flipped bits to tabuTenure, and by decrementing the tenure of all other bits.
4. Return the best configuration seen and its objective value.

Next, we drill down into the mechanism by which targeted variants are generated, and how these variants are tabued to prevent trapping in poor local minima.

8.3 Models for Targeted Variation

Tabu search relies on small changes to the current configuration to generate new variants. Small, rather than large, changes are critical to the success of this incremental approach. Once the algorithm has accumulated a number of improvements, the resulting configuration is significantly better than a randomly chosen configuration. To find another configuration which improves even further, a large change is unlikely to be successful unless the alteration is very carefully designed. Thus, making small changes is key to the success of the local search approach. Unfortunately, the small scale incremental approach means that when larger scale changes are necessary, the algorithm may fail to identify these larger changes through iterative local improvement. Thus, we develop a mechanism by which larger, but targeted, alterations can be proposed.

We first observe that if the problem you are trying to solve was in fact a hardware-structured Ising model we could use the QPU to propose variants that were very good solutions by running quantum annealing. Of course, most problems do not have the structure of the hardware. We can, however, build a model compatible with a hardware-structured Ising model, and which at least locally around the current configuration, approximates the true objective function $G(\boldsymbol{s})$. This approximation, when minimized in hardware, yields variants that do a good job at minimizing the model. Thus, if the model is moderately accurate (at least locally), then these model minimizers may be very useful variants for tabu to consider.

Model building can be made very flexible, and adaptable to any hardware qubit connectivity. All that is required to build a model is training data consisting of some spin configurations $\{\boldsymbol{s}_i\}$ and corresponding objective function values $\{G(\boldsymbol{s}_i)\}$. Within the machine learning literature, a great deal is known about building models from training data like this (the construction of such models is called supervised learning), and all of this insight can be brought to bear. As one simple example, linear regression, can be used to build a model. The parameters available in the linear model are the $\boldsymbol{h}$ values of all qubits, and the $\boldsymbol{J}$ values of all edges in the hardware connectivity graph. Values for $\boldsymbol{h}$ and $\boldsymbol{J}$ are set by minimizing the squared error between the hardware Chimera energies at $\{\boldsymbol{s}_i\}$ and the true values $\{G(\boldsymbol{s}_i)\}$.

However, in spite of the similarity to other supervised learning applications, there is one significant difference in the present case. In this application it is not the model itself that matters, but rather the minimizers of the model as these are proposed as variants for tabu search. Once a model is built, and some of its minimizers identified, it is important that these minimizers be local to the current configuration around which the model was constructed. If the minimizers are far from the current configuration then they will be large extrapolations from the region in which the model is likely to be valid. Consequently, distant minimizers are likely to be artifacts without statistical validity, and thus poor variants to suggest to tabu search. In building models then we want to bias towards those models having minimizers which are nearby to the current configuration around which the model has been trained.

In addition, models are built at every step in the tabu search, and because this happens so frequently the model building process must be fast. To minimize the time spent model building we use the training data we have available at hand, and very fast learning algorithms to construct the model. Fortunately, there is a nice supply of training data available in the configuration itself and all of its single bit flip variants which have been generated for tabu search. These configurations and the objective function values are used as training data. With more training data better models might be constructed, and the proposed variants might be better targeted. However, more training data requires evaluation of the objective at these new data points, and if the objective function is computationally expensive this may be costly. Consequently, we have adopted a conservative approach, and based the model only on the configuration and all its single bit flip variants. We rely on fast linear programming solvers to construct models. The linear programming approach strikes a balance between speed and accurate models.

We rely on the quantum annealing to perform approximate minimization of the local model. The fact that the QPU returns a diversity of answers is useful in this setting because the model is only an approximation to the true local objective. Generating multiple samples from the QPU is useful in supplying additional variants, but too many samples can become costly if evaluating the objective function on these samples is expensive. We address ways to work with costly objective functions in *Parallelization*.

8.4 Tabuing Variants

The tabu mechanism applied when moving to a single flip variant is simple—the single bit that was flipped is made tabu and prevented from flipping again for tabuTenure iterations. However, a variant proposed from the modeling process will differ from the current configuration in two or more bits. If such a configuration is adopted because it has lower objective value, then a new tabu mechanism must be specified. We might, for example, tabu all the flipped bits to prevent them moving again for tabuTenure iterations. However, such a move is too drastic and can rapidly lead to all bits being made tabu so that no moves are permitted. Moreover, the goal of the tabu mechanism is to prevent revisiting previously examined configurations, and some of these bits could be flipped and still not return to a previous state. Consequently, when making a multi-spin-flip move we randomly select one of the flipped bits and make that bit tabu. None of the other flipped bits are tabued.

When determining whether a multispin flip move can be adopted a similar issue arises. In the current QSage algorithm we allow the new state to be adopted as long as at least one of the flipped spins is not tabued. So even though some of the flipped spins may be tabued it is unlikely the move will be returning to a recently visited configuration because a non-tabued spin has also flipped.

While the choice adopted here in QSage works well for many problems, but alternative tabu mechanisms for variants at multiple bit-flip distances is an open research question, and future versions of QSage may change the current tabu mechanism.

8.5 Parallelization

The QSage algorithm itself is very lightweight, and typically the vast majority of run time is spent in evaluating the objective function $G(\boldsymbol{s})$ at configurations proposed as candidate variants. Thus the QSage routine offers the user the ability to parallelize these objective function evaluations in cases where the objective is computationally expensive. QSage generates batches of proposed variants consisting of single bit flips and hardware-proposed variants

consisting of multiple bit flip variants. The number of variants within a batch is controlled by the solver parameter *num_reads*. QSage calls the objective function by passing in the entire batch of configurations, and not configuration by configuration. This allows the user to define a function returning objective values at all configurations within the batch. The user may then parallelize across these evaluations by spawning processes or threads to evaluate the objective at each constituent configuration, or at subsets of configurations.

Even if the objective is not parallelized, evaluating the objective at batches of configurations can prove beneficial. For some objectives required state can be calculated once and then shared across all configuration evaluations rather than being recalculated for each configuration. Depending on the form of the objective this savings can prove to be beneficial. The user should also keep in mind that the objective function may contain static variables which maintain their state between calls. This may also allow for faster evaluation if the objective function incrementally updates certain data structures which allow for faster evaluation.

8.6 Types and Enums

8.6.1 sapi_ProblemType

```
typedef enum sapi_ProblemType
{
    SAPI_ISING,
    SAPI_QUBO
} sapi_ProblemType;
```

SAPI problem type enum.

SAPI_ISING: Ising problem.

SAPI_QUBO: QUBO problem.

8.6.2 sapi_QSageObjectiveFunction

```
typedef sapi_Code (*sapi_QSageObjectiveFunction)(const int *states, int len,
                                                 int num_states,
                                                 void *extra_arg,
                                                 double *result);
```

SAPI QSage objective function signature.

Parameters:

states: can contain -1/1 (Ising) or 0/1 (QUBO).

len: the length of the states array.

num_states: number of states. Note that each state's length is len / num_states.

extra_arg: user can pass extra argument.

result: its length should be num_states.

Return Value:

SAPI error code.

8.6.3 sapi_QSageLPSolver

```
typedef sapi_Code (*sapi_QSageLPSolver)(const double *f, const double *Aineq,
                                        const double *bineq, const double *Aeq,
                                        const double *beq, const double *lb,
                                        const double *ub, int num_vars,
                                        int Aineq_size, int Aeq_size,
                                        void *extra_arg, double *result);
```

SAPI QSage lp solver function signature, a solver that can solve linear programming problems.

Finds the minimum of a problem specified by

```
min      f * x
st.      Aineq * x <= bineq
           Aeq * x = beq
           lb <= x <= ub
```

Parameters:

f: linear objective function, its length is *num_vars*.

Aineq: linear inequality constraints, its length is *Aineq_size* * *num_vars*.

bineq: righthand side for linear inequality constraints, its length is *Aineq_size*.

Aeq: linear equality constraints, its length is *Aeq_size* * *num_vars*.

beq: righthand side for linear equality constraints, its length is *Aeq_size*.

lb: lower bounds, its length is *num_vars*.

ub: upper bounds, its length is *num_vars*.

num_vars: number of variables.

Aineq_size: the length of *Aineq*.

Aeq_size: the length of *Aeq*.

extra_arg: user can pass extra argument.

result: its length should be *num_vars*.

Return Value:

SAPI error code.

8.6.4 sapi_QSageParameters

```
typedef struct sapi_QSageParameters
{
    int draw_sample;
    double exit_threshold_value;
    const int *initial_solution;
    sapi_ProblemType ising_qubo;
    sapi_QSageLPSolver lp_solver;
    void* lp_solver_extra_arg;
    long long int max_num_state_evaluations;
    int use_random_seed;
    unsigned int random_seed;
    double timeout;
```

```
    int verbose;
} sapi_QSageParameters;
```

SAPI QSage parameters struct.

Fields:

draw_sample: if 0, *sapi_solveQSage* will not draw samples, will only do tabu search. (must be an integer [0 1], default = 1)

exit_threshold_value: if best value found by *sapi_solveQSage* <= *exit_threshold_value* then exit. (can be any number, default = -infinity)

initial_solution: if provided, must only contain -1/1 if *ising_qubo* parameter is not set or set as *SAPI_ISING*; or 0/1 if *ising_qubo* parameter is set as *SAPI_QUBO*. Its length must be *num_vars*. (default is randomly set)

ising_qubo: if set as *SAPI_ISING*, the return best solution will be -1/1; if set as *SAPI_QUBO*, the return best solution will be 0/1. (must be *SAPI_ISING* or *SAPI_QUBO*, default = *SAPI_ISING*)

lp_solver: a solver that can solve linear programming problems, refer to the documentation of *sapi_QSageLPSolver*. (default uses Coin-or Linear Programming solver)

lp_solver_extra_arg: user can pass extra argument for lp_solver. (default = NULL)

max_num_state_evaluations: the maximum number of state evaluations, if the total number of state evaluations >= *max_num_state_evaluations* then exit. (must be an integer >= 0, default = 50,000,000)

use_random_seed: indicates whether to use the random_seed field or not. (must be an integer [0 1], default = 0)

random_seed: seed for random number generator that *sapi_solveQSage* uses. (must be an integer >= 0, default is a time-based seed)

timeout: timeout for *sapi_solveQSage* (seconds). (must be a number >= 0.0, default is approximately 10.0 seconds)

verbose: control the output information. (must be an integer [0 2], default = 0)

0: quiet

1, 2: different levels of verbosity

when *verbose* is 1, the output information will be like:

[num_state_evaluations = ..., num_obj_func_calls = ...,

num_solver_calls = ..., num_lp_solver_calls = ...],

best_energy = ..., distance to best_energy = ...

detailed explanation of the output information:

- "num_state_evaluations": the current total number of state evaluations.
- "num_obj_func_calls": the current total number of objective function calls.
- "num_solver_calls": the current total number of solver (loca/remote) calls.
- "num_lp_solver_calls": the current total number of lp solver calls.
- "best_energy": the global best energy found so far.
- "distance to best_energy": the hamming distance between the global best state found so far and the current state found by tabu search.

when *verbose* is 2, in addition to the output information when verbose is 1, the following output information will also be shown:

sample_num = ...

min_energy = ...

move_length = ...

detailed explanation of the output information:

- "sample_num": the number of unique samples returned by sampler.
- "min_energy": minimum energy found during the current phase of tabu search.
- "move_length": the length of the move (the hamming distance between the current state and the new state).

The acceptable range and the default value of each field are given in the table below:

Field	Range	Default value
draw_sample	[0 1]	1
exit_threshold_value	any number	-infinity
initial_solution	N/A	randomly set
ising_qubo	*SAPI_ISING/SAPI_QUBO*	*SAPI_ISING*
lp_solver	N/A	uses Coin-or Linear Programming solver
lp_solver_extra_arg	N/A	NULL
max_num_state_evaluations	>= 0	50,000,000
use_random_seed	[0 1]	0
random_seed	>= 0	randomly set
timeout	>= 0.0	10.0
verbose	[0 2]	0

8.6.5 sapi_QSageObjFunc

```
typedef struct sapi_QSageObjFunc
{
    sapi_QSageObjectiveFunction objective_function;
    void* objective_function_extra_arg;
    int num_vars;
} sapi_QSageObjFunc;
```

QSage objective function related parameter struct.

Fields:

objective_function: function pointer of *sapi_QSageObjectiveFunction*.

objective_function_extra_arg: *objective function*'s extra argument.

num_vars: number of variables.

8.6.6 sapi_QSageStat

```
typedef struct sapi_QSageStat
{
    long long int num_state_evaluations;
    long long int num_obj_func_calls;
    long long int num_solver_calls;
```

```
    long long int num_lp_solver_calls;
} sapi_QSageStat;
```

SAPI QSage stat struct.

Fields:

num_state_evaluations: number of state evaluations.

num_obj_func_calls: number of user-provided objective function calls.

num_solver_calls: number of solver (local/remote) calls.

num_lp_solver_calls: number of lp solver calls.

8.6.7 sapi_QSageProgressEntry

```
typedef struct sapi_QSageProgressEntry
{
    sapi_QSageStat stat;
    double time;
    double energy;
} sapi_QSageProgressEntry;
```

SAPI QSage progress struct. This struct stores QSage progress history.

Fields:

stat: a *sapi_QSageStat* struct representing the QSage stat when energy was found.

time: the time (seconds) when energy was found.

energy: the objective value of a certain state of the objective function.

8.6.8 sapi_QSageProgress

```
typedef struct sapi_QSageProgress
{
    sapi_QSageProgressEntry* elements;
    size_t len;
} sapi_QSageProgress;
```

SAPI QSage progress struct.

Fields:

elements: an array of *sapi_QSageProgressEntry* struct.

len: the length of the elements array.

8.6.9 sapi_QSageInfo

```
typedef struct sapi_QSageInfo
{
    sapi_QSageStat stat;
    double state_evaluations_time;
    double solver_calls_time;
```

```
    double lp_solver_calls_time;
    double total_time;
    sapi_QSageProgress progress;
} sapi_QSageInfo;
```

SAPI QSage info struct.

Fields:

stat: a *sapi_QSageStat* struct which stores the total stat when *sapi_solveQSage* completes computation.

state_evaluations_time: state evaluations time (seconds).

solver_calls_time: solver (local/remote) calls time (seconds).

lp_solver_calls_time: lp solver calls time (seconds).

total_time: total running time of *sapi_solveQSage* (seconds).

progress: a *sapi_QSageProgress* struct which stores the *sapi_solveQSage* progress history during the computation.

8.6.10 sapi_QSageResult

```
typedef struct sapi_QSageResult
{
    int *best_solution;
    size_t len;
    double best_energy;
    sapi_QSageInfo info;
} sapi_QSageResult;
```

SAPI QSage result struct.

Fields:

best_solution: the best state founnd.

len: the length of the *best_solution* array.

best_energy: the best energy found.

info: the *sapi_QSageInfo* struct which stores the information during the *sapi_solveQSage* computation.

8.7 sapi_solveQSage

```
sapi_Code sapi_solveQSage(const sapi_QSageObjFunc *obj_func,
                          const sapi_Solver *solver,
                          const sapi_SolverParameters *solver_params,
                          const sapi_QSageParameters *qsage_params,
                          sapi_QSageResult **qsage_result, char *err_msg);
```

Solve a QSage problem. (If number of variables <= 10, the *sapi_solveQSage* will do a brute-force search)

Note: Use *sapi_freeQSageResult* function to free the qsage_result pointer.

8.7.1 Parameters

obj_func: a pointer to *sapi_QSageObjFunc*.

solver: a *sapi_Solver* pointer.

solver_params: parameters for *solver*.

qsage_params: parameters for QSage algorithm.

qsage_result: a pointer to a pointer to *sapi_QSageResult*.

err_msg: error message.

8.7.2 Return Value

SAPI error code.

8.8 sapi_freeQSageResult

```
void sapi_freeQSageResult(sapi_QSageResult* qsage_result);
```

Free *sapi_QSageResult* pointer.

8.8.1 Parameters

qsage_result: a *sapi_QSageResult* pointer.

CHAPTER

NINE

SAPI SOLVERS

9.1 Quantum Processor–Like Solvers

This section describes the SAPI interface to the quantum processing unit (QPU) along with software solvers designed to mimic the problem-solving behaviors of the QPU. These software solvers are useful for prototyping algorithms that make multiple calls to the hardware.

9.1.1 Common Parameters and Properties

The hardware and software solvers in this section behave nearly identically. All common attributes are listed below and solver-specific information appears in later subsections.

Properties

Every solver has the common property *supported_problem_types*:

supported_problem_types: “qubo” and “ising”.

In addition, quantum processor-like solvers have the following properties:

num_qubits: total number of qubits, both working and non-working, in the processor.

qubits: working qubit indices.

couplers: working couplers in the processor. A coupler contains two elements [*q1*, *q2*], where both *q1* and *q2* appear in the working qubits, in the range [0, *num_qubits* - 1] and in ascending order (i.e., $q1 < q2$). It is these couplers that may be programmed with non-zero J values.

Solving Parameters

num_reads: A positive integer that indicates the number of states (output solutions) to read from the solver.

answer_mode: A logical value indicating whether to return a histogram of answers, sorted in order of energy (‘histogram’); or to return all answers individually in the order they were read (‘raw’).

max_answers: Maximum number of answers returned from the solver in histogram mode (which sorts the returned states in order of increasing energy); this is the total number of distinct answers. In raw mode (i.e., when *answer_mode* = ‘raw’), this limits the returned values to the first *max_answers* of *num_reads* samples.

Answer Format

If *answer_mode* is 'raw', then the answer contains two fields: *solutions* and *energies*. The *solutions* field is a list of lists; the inner lists all have length *num_qubits* and entries from {-1, +1} (for Ising problems) or {0, 1} (for QUBOs). The *energies* field contains the energy of each corresponding solution.

If *answer_mode* is 'histogram', then the answer still contains *solutions* and *energies* fields, but in this case the solutions are unique and sorted in increasing-energy order. There is also a *num_occurrences* field indicating how many times each solution appeared.

9.1.2 Quantum Processor (Hardware) Solvers

This section describes additional parameters relevant to problems submitted to hardware solvers or via the virtual full-yield chimera (VFYC) solver.

Virtual Full-Yield Chimera Solver

The VFYC solver emulates a fully connected Chimera graph based on an idealized abstraction of the system. Through this solver, variables corresponding to a Chimera structured graph that are not representable on a specific working graph are determined via hybrid use of the QPU and the integrated postprocessing system, which fills in any missing qubits and couplers that may affect the QPU.

For problems submitted to this solver, postprocessing always runs. As with other solvers, users can choose to run sampling or optimization postprocessing on the result. If, however, neither option is specified, optimization postprocessing will run.

For more information on the VFYC solver and how it is integrated with the postprocessing system, see *Postprocessing Methods on D-Wave Systems* on the Qubist web user interface.

Additional Parameters for Hardware Solvers

auto_scale: Indicates whether h and J values will be rescaled to use as much of the range of h (*h_range*, see the solver properties on the Qubist web user interface) and the range of J (*j_range*, see the solver properties of the User Interface) as possible (true), or be used as is (false). When enabled, h and J values need not lie within the range of h and the range of J (but must still be finite). This parameter is enabled by default.

annealing_time: A positive integer that sets the duration (in microseconds) of quantum annealing time.

beta: Boltzmann distribution parameter. Only used when *postprocess* is set to "sampling".

chains: Defines which qubits represent the same logical variable (or "chain") when postprocessing is enabled.

num_spin_reversal_transforms: Number of spin-reversal transforms to perform.

> Use this parameter to specify how many spin-reversal transforms to perform on the problem. Valid values range from 0 (do not transform the problem; the default value) to a value equal to but no larger than the *num-reads* specified. If you specify a nonzero value, the system divides the number of reads by the number of spin-reversal transforms to determine how many reads to take for each transform. For example, if the number of reads is 10 and the number of transforms is 2, then 5 reads use the first transform and 5 use the second.
>
> Applying a spin-reversal transform can improve results by reducing the impact of analog errors that may exist on the QPU. This technique works as follows: Given an n-variable Ising problem, we can select a random $g \in \{\pm 1\}^n$ and transform the problem via $h_i \mapsto h_i g_i$ and $J_{ij} \mapsto J_{ij} g_i g_j$. A spin-reversal transform does not alter the mathematical nature of the Ising problem. Solutions s of the original problem and s' of the transformed problem are related by $s'_i = s_i g_i$ and have identical energies. However, the

sample statistics can be affected by the spin-reversal transform because the QPU is a physical object with asymmetries.

Spin-reversal transforms work correctly with postprocessing and chains. Majority voting happens on the original problem state, not on the transformed state.

Be aware that each transform reprograms the QPU; therefore, using more than 1 transform will increase the amount of time required to solve the problem. For more information about timing, see *Measuring Computation Time on D-Wave Systems* available on the Qubist web user interface.

postprocess: postprocessing options:

- "" (empty string): no postprocessing (default). If this option is selected for the VFYC solver, optimization postprocessing runs.
- "sampling": runs a short Markov chain Monte Carlo with single bit flips starting from each hardware sample. The target probability distribution is a Boltzmann distribution at inverse temperature β.
- "optimization": perform a local search on each sample, stopping at a local minimum.

When postprocessing is enabled, qubits in the same chain, defined by the *chains* parameter, are first set to the same value by majority vote. Postprocessing is performed on the logical problem but qubit-level answers are returned. For more information about postprocessing, see *Postprocessing Methods on D-Wave Systems* on the Qubist web user interface.

programming_thermalization: An integer that gives the time (in microseconds) to wait after programming the processor in order for it to cool back to base temperature (i.e., post-programming thermalization time). Lower values will speed up solving at the expense of solution quality.

readout_thermalization: An integer that gives the time (in microseconds) to wait after each state is read from the processor in order for it to cool back to base temperature (i.e., post-readout thermalization time). This value contributes to the *qpu_delay_time_per_sample* field.

Note: While still supported in SAPI Release 2.10, the `readout_thermalization` parameter is deprecated and will eventually be removed from the API. Plan code updates accordingly.

anneal_offsets: Amount to offset annealing paths, per qubit.

Provide an array of annealing offset values, in normalized offset units, for all qubits, working or not. Use 0 for no offset. Negative values produce a negative offset (qubits are annealed *after* the standard annealing trajectory); positive values produce a positive offset (qubits are annealed *before* the standard trajectory). Before using this parameter, query the solver properties to determine whether `anneal_offsets` exists in the `parameters` property. If so, retrieve `anneal_offset_ranges` to obtain the permitted offset values per qubit.

Note: Annealing offsets are not supported on D-Wave 2X and earlier systems.

9.1.3 Optimizing Emulators

This type of solver will solve the same type of optimization problems as the quantum hardware, but using a classical software algorithm.

Solver Name

These solvers have names ending with "-sw_optimize".

Answer Format

This class of solvers is entirely deterministic, so the semantics of some parameters is different. The number of solutions returned is always the lesser of *max_answers* and *num_reads* × *num_programming_cycles*. The solutions returned are the lowest-energy states, sorted in increasing-energy order.

When *answer_mode* is 'histogram', the *num_occurrences* field contains all ones, except possibly for the lowest energy solution. That first entry is set so that the sum of all entries is *num_reads* × *num_programming_cycles*.

Warning: The parameter *num_programming_cycles* is deprecated and will be removed in a future release.

9.1.4 Sampling Emulators

This type of solver mimics the probabilistic nature of the quantum processor. It draws samples from a Boltzmann distribution, that is, state *s* is sampled with probability proportional to:

$$\exp(-\beta E(s))$$

where β is some parameter and $E(s)$ is the energy of state s.

Solver Name

These solvers have names ending with "-sw_sample".

Additional Parameters

random_seed: Random number generator seed. When a value is provided, solving the same problem with the same parameters will produce the same results every time. If no value is provided, a time-based seed is selected.

beta: Boltzmann distribution parameter.

All parameters[1] of the quantum processor-like solvers and their default values are summarized below:

Parameter	Default value
num_reads	1
answer_mode	'histogram'
max_answers	*num_reads*
annealing_time	1000
programming_thermalization	1000
readout_thermalization	Hardware specific
auto_scale	Automatic scaling is enabled
random_seed	Time-based value
beta	Processor default
chains	No chains
postprocess	No postprocess Note: The VFYC solver always runs postprocessing. If blank, optimization postprocessing runs.
num_spin_reversal_transforms	0
anneal_offsets	No offsets

[1] Annealing offsets are not supported on D-Wave 2X and earlier systems.

9.2 Ising Heuristic Solver

The Ising heuristic solver is intended to solve complex problem structures. It has some important differences from other solvers:

- There is no fixed problem structure. In particular, this solver does not have the properties *num_qubits*, *qubits*, and *couplers*
- Only one solution is returned and it is not guaranteed to be optimal
- Solver properties and parameters are entirely disjoint from those of other solvers
- It cannot be used with the BlackBox solver.

Note that this heuristic solver can handle problems of arbitrary structure.

9.2.1 Algorithm

The core of the heuristic solver is a local search based on optimizing low-treewidth subgraphs. In pseudocode:

```
function local_search(problem, x)
  stuck = 0
  e = evaluate(problem, x)
  while (time limit not exceeded) and (stuck <= local_stuck_limit)
    select random low-treewidth subproblem
    (new_e, x) = solve(subproblem, x)
    if subproblem is the entire problem
      return (new_e, x, exact=true)
    if new_e < e
      stuck = 0
    else
      stuck = stuck + 1
    e = new_e
  return (e, x, exact=false)
```

The value *local_stuck_limit* is a user-supplied parameter. What constitutes a "low-treewidth subproblem" is determined by the parameter *max_local_complexity*. The time limit is provided by the parameter *time_limit_seconds*.

In order to escape local minima, multiple copies of the solution are made and bits are randomly flipped.

```
function ising_heuristic(problem)
  x = random solution vector
  (e, x, exact) = local_search(problem, x)
  if exact
    return (e, x)
  best_e = current_e = e; best_x = current_x = x
  iter = 0
  while (time limit not exceeded) and (iter < iteration_limit)
    iter = iter + 1
    new_e = current_e; new_x = current_x
    for i = 1 to num_perturbed_copies
      x = current_x
      flip each bit of x with probability p(i)
      (e, x, exact) = local_search(problem, x)
      if exact
        return (e, x)
      if e < new_e
        new_e = e; new_x = x
```

```
    current_e = new_e; current_x = new_x
    if current_e < best_e
      best_e = current_e; best_x = current_x
  return (best_e, best_x)
```

In the `ising_heuristic` function, *iteration_limit* and *num_perturbed_copies* are user-provided parameters. The (`i`-dependent) bit flip probability `p(i)` is determined by the parameters *min_bit_flip_prob* and *max_bit_flip_prob*.

9.2.2 Solver Details

Solver Name

"ising-heuristic".

Properties

None, except for the common property *supported_problem_types*.

Parameters

Many of these parameters require a high-level understanding of the heuristic algorithm.

Parameter settings can wildly affect solver performance and solution quality. It is difficult in general to know what good values are a priori; defaults are selected to favour quicker run times over aggressive searches. Therefore, experimentation with these values is recommended.

iteration_limit: Maximum number of solver iterations. This does not include the initial local search.

time_limit_seconds: Maximum wall clock time in seconds. Actual run times will exceed this value slightly.

random_seed: Random number generator seed. When a value is provided, solving the same problem with the same parameters will produce the same results every time. If no value is provided, a time-based seed is selected.

> The use of a wall clock-based timeout may in fact cause different results with the same *random_seed* value. If the same problem is run under different CPU load conditions (or on computers with different performance), the amount of work completed may vary despite the fact that the algorithm is deterministic. If repeatability of results is important, rely on the *iteration_limit* parameter rather than the *time_limit_seconds* parameter to set the stopping criterion.

num_variables: Lower bound on the number of variables. This solver can accept problems of arbitrary structure and the size of the solution returned is determined by the maximum variable index in the problem. The size of the solution can be increased by setting this parameter.

max_local_complexity: Maximum complexity of subgraphs used during local search. The run time and memory requirements of each step in the local search are exponential in this parameter. Larger values allow larger subgraphs (which can improve solution quality) but require much more time and space.

> Subgraph "complexity" here means treewidth+1.

local_stuck_limit: Number of consecutive local search steps that do not improve solution quality to allow before determining a solution to be a local optimum. Larger values produce more thorough local searches but increase run time.

num_perturbed_copies: Number of perturbed solution copies created at each iteration. Run time is linear in this value.

min_bit_flip_prob, *max_bit_flip_prob*: Bit flip probability range. The probability of flipping each bit is constant for each perturbed solution copy but varies across copies. The probabilities used are linearly interpolated between

min_bit_flip_prob and *max_bit_flip_prob*. Larger values allow more exploration of the solution space and easier escapes from local minima but may also discard nearly-optimal solutions.

All parameters of the Ising heuristic solvers and their default values are summarized below:

Parameter	Default value
iteration_limit	10
time_limit_seconds	5
random_seed	a time-based seed
num_variables	0
max_local_complexity	9
local_stuck_limit	8
num_perturbed_copies	4
min_bit_flip_prob	1/32
max_bit_flip_prob	1/8

9.3 Summary

The following table summarizes the properties and parameters of each solver[2] described above:

Solver	Properties	Parameters
Quantum processor, including the VFYC solver	• *supported_problem_types* • *num_qubits* • *qubits* • *couplers*	• *auto_scale* • *annealing_time* • *beta* • *chains* • *num_spin_reversal_transforms* • *postprocess* • *programming_thermalization* • *readout_thermalization* • *num_reads* • *max_answers* • *answer_mode* • *anneal_offsets*
Optimizing emulator	• *supported_problem_types* • *num_qubits* • *qubits* • *couplers*	• *num_reads* • *max_answers* • *answer_mode*
Sampling emulator	• *supported_problem_types* • *num_qubits* • *qubits* • *couplers*	• *random_seed* • *beta* • *num_reads* • *max_answers* • *answer_mode*
Ising heuristic solver	• *supported_problem_types*	• *iteration_limit* • *time_limit_seconds* • *random_seed* • *num_variables* • *max_local_complexity* • *local_stuck_limit* • *num_perturbed_copies* • *min_bit_flip_prob* • *max_bit_flip_prob*

[2] Annealing offsets are not supported on D-Wave 2X and earlier systems.

CHAPTER

TEN

API TOKENS

API tokens are used to authenticate the client in order to request data via web services as well as to connect to a remotely-located hardware solver. Using tokens eliminates the user from having to embed a username and password in their software when interacting with the solver. API tokens can be generated on the Qubist web user interface from the drop-down menu on the right-hand side of the page. Users can create and activate as many tokens as they need.

API tokens can be used by anyone who has access to the system and who knows the token ID.

BIBLIOGRAPHY

[Ahuja2000] Ahuja, R. K.; Orlin, J. B.; Sharma, D. *Very large-scale neighborhood search*, International Transactions in Operational Research 7 (4-5): 301-317, (2000).

[Glover90] Glover, F., *Tabu Search: A Tutorial*, Interfaces July/August 1990 20:74-94;