bincfg.cfg package

See the homepage for a quick example of how to use these cfg objects.

Submodules

bincfg.cfg.cfg module

class `bincfg.cfg.cfg.`**`CFG`**(*data=None*, *normalizer=None*, *metadata=None*)

Bases: `object`

A Control Flow Graph (CFG) representation of a binary

Can currently load in from:

- Rose binary analysis tool (both text and graphviz dotfile outputs)

NOTE: 'indeterminate' blocks/calls/etc. are completely ignored

Parameters:

- **data** (*Optional[Union[str, TextIO, Sequence[str], pd.DataFrame]]*) –

  the data to use to make this CFG. Data type will be inferred based on the data passed:

  - string: either string with newline characters that will be split on all newlines and treated as either a text or graphviz rose input, or a string with no newline characters that will be treated as a filename. Filenames will be opened as ghidra parquet files if they end with either '.pq' or '.parquet', and text/graphviz rose input otherwise
  - Sequence of string: will be treated as already-read-in text/graphviz rose input
  - open file object: will be read in using *.readlines*, then treated as text/graphviz rose input
  - pandas dataframe: will be parsed as ghidra parquet file
  - anything else: an error will be raised

- **normalizer** (*Optional[Union[str, Normalizer]]*) – the normalizer to use to force-renormalize the incoming CFG, or None to not normalize
- **metadata** (*Optional[dict]*) – a dictionary of metadata to add to this CFG NOTE: passed dictionary will be shallow copied

`add_function`(**functions*, *override=False*)

Adds the given function(s) to this cfg. This should only be done once the given function(s) have been fully initialized

This will do some housekeeping things such as:

- setting the parent_cfg and parent_function attributes of functions and blocks respectively
- adding missing edges to their associated edges_out and edges_in
- converting edges from (None/address, None/address, edge_type) tuples into CFGEdge() objects
- adding from_block and to_block in new edges if missing

Parameters:

- **function** (*CFGFunction*) – arbitrary number of CFGFunction's to add
- **override** (*bool*) – if False, an error will be raised if a function or basic block contains an address that already exists in this CFG. If True, then that error will not be raised and those functions/basic blocks will be overriden (which has unsupported behavior). Defaults to False.

property **`asm_counts`**

A collections.Counter() of all unique assembly lines and their counts in this cfg

property **`blocks`**

A list of basic blocks in this CFG (in order of memory address)

`blocks_dict` = *None*

Dictionary mapping integer basic block addresses to their `CFGBasicBlock` objects

property **`edges`**

A list of all outgoing `CFGEdge`'s in this `CFG`

classmethod **`from_cfg_dict`**(*cfg_dict*, *cfg=None*)

Converts a cfg dict object into a CFG

Expects the cfg_dict to have the exact same structure as that listed in CFG().to_cfg_dict()

You can optionally pass a cfg, in which case this data will be added to (and override) that cfg

classmethod **`from_networkx`**(*graph*, *cfg=None*)

Converts a networkx graph to a CFG

Expects the graph to have the exact same structure as is shown in CFG().to_networkx()

You can optionally pass a cfg, in which case this data will be added to (and override) that cfg

property **functions**
A list of functions in this CFG (in order of memory address)

functions_dict = *None*
Dictionary mapping integer function addresses to their `CFGFunction` objects

get_block(*address*, *raise_err=True*)
Returns the basic block in this CFG with the given address

Parameters:
- **address** (*Union[str, int, Addressable]*) – a string/integer memory address, or an addressable object (EG: CFGBasicBlock /CFGFunction)
- **raise_err** (*bool, optional*) – if True, will raise an error if the basic block with the given memory address was not found, otherwise will return None. Defaults to True.

Raises: **ValueError** – if the basic block with the given address could not be found

Returns: the basic block with the given address

Return type: Union[CFGBasicBlock, None]

get_block_containing_address(*address*, *raise_err=True*)
Returns the basic block in this CFG that contains the given address at the start of one of its instructions

This will lazily compute an instruction lookup dictionary mapping addresses to the blocks that contain them

Parameters:
- **address** (*Union[str, int, Addressable]*) – a string/integer memory address, or an addressable object (EG: CFGBasicBlock /CFGFunction)
- **raise_err** (*bool, optional*) – if True, will raise an error if the basic block with the given memory address was not found, otherwise will return None. Defaults to True.

Raises: **ValueError** – if the basic block containing the given address could not be found

Returns: the basic block that contains the given address

Return type: Union[CFGBasicBlock, None]

get_cfg_build_code()
Returns python code that will build the given cfg. Used for testing

Parameters: **cfg** (*CFG*) – the cfg

Returns: string of python code to build the cfg

Return type: str

get_compressed_stats(*tokens*)
Returns some stats about this CFG in a compressed version

These are meant to be very basic stats useful for simple comparisons (EG: dataset subsampling). These values are highly compressed /convoluted as they are used for generating statistics on 100+ million cfg's on HPC, and thus output space requirements outweigh single-graph compute time. Will return a single numpy array (1-d, dtype=np.uint8) with indices/values:

- [0:12]: graph-level stats (number of nodes, number of functions, number of assembly lines), each a 4-byte unsigned integer of the exact value in the above order. The bytes are always stored as little-endian.
- [12:20]: node degree histogram. Counts the number of nodes with degrees: 0 incomming, 1 incomming, 2 incomming, 3+ incomming, 0 outgoing, 1 outgoing, 2 outgoing, 3+ outgoing. See below in things that are not in these stats for reasoning. Values will be a list in the above order:

  [0-in, 1-in, 2-in, 3+in, 0-out, 1-out, 2-out, 3+out]

  Reasoning: the vast majority of all nodes will have 0, 1 or 2 incomming normal edges, and 0, 1, or 2 outgoing normal edges, so this should be a fine way of storing that data for my purposes. Function call edges will be handled by the function degrees.
- [20:46]: a histogram of node sizes (number of assembly lines per node). Histogram bins (left-inclusive, right-exclusive, 26 of them) will be:

  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 14, 16, 18, 20, 25, 30, 35, 40, 50, 60, 80, 100, 150, 200+]

Reasoning: different compiler optimizations (inlining, loop unrolling, AVX instructions, etc.) will likely drastically change the sizes of nodes. The histogram bin edges were chosen arbitrarily in a way that tickled my non-neurotypical, nice-number-loving brain.

- [46:72]: a histogram of (undirected) function degrees (in the function call graph). Histogram bins (left-inclusive, right-exclusive, 26 of them) will be:

  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 14, 16, 18, 20, 25, 30, 35, 40, 50, 60, 80, 100, 150, 200+]

  Reasoning: most functions will only be called a relatively small number of unique times across the binary (EG: <10), while those that are called much more are likely

- [72:93]: a histogram of function sizes (number of nodes in each function). Histogram bins (left-inclusive, right-exclusive, 21 of them) will be:

  [0, 1, 2, 3, 4, 6, 8, 10, 15, 20, 25, 30, 40, 60, 80, 100, 150, 200, 300, 400, 500+]

  Reasoning: different compiler optimizations (especially inlining) will drastically change the size of functions. The histogram bins can be more spread out (IE: not as focused on values near 0, and across a larger range) since the number of nodes in a function has a noticeably different distribution than, say, the histogram of node sizes

- [93:]: a histogram of assembly tokens. One value per token. You should make sure the normalization method you are using is good, and doesn't create too many unique tokens.

  Reasoning: obvious

The returned array will be of varrying length based on the number of unique tokens in the tokens dictionary.

Values above (unless otherwise stated) are stored as 1-byte percentages of the number of nodes in the graph that are in that bin. EG: 0 would mean there are 0 nodes with that value, 1 would mean between [0, 1/255) of the nodes/functions in the graph have that value, 2 would be between [1/255, 2/255), etc., until a 255 which would be [245/255, 1.0]

Things that are NOT in these stats and reasons:

- Other node degrees: these likely don't change too much between programs (specifically their normalized values) as even with different programs/compiler optimizations. Changes between cfg's will likely only change the relative proportions of nodes with 1 vs. 2 incoming edges, and those with 1 vs. 2 vs. 3 outgoing edges. Any other number of edges are pretty rare, hence why we only keep those edge measures and only those using normal edges (since function call edges will be gathered in the func_degree_hist and would mess with this premise)
- Edge statistics (EG: number of normal vs. function call edges): this information is partially hidden in the histograms already present, and exact values do not seem too important
- Other centrality measures: I belive (but have not proven) that node-based centrality measures would not contain enough information to display differences between CFG's to be worth it. Because of the linear nature of sequential programs, I believe their centrality measures would be largely similar and/or dependant on other graph features already present in the stats above (EG: number of nodes in a function). I think any differences between centrality measurements on these graphs will be mostly washed out by the linear nature, especially since we would only be looking at normal edges, not function call ones. The only differences that would be highlighted would be information about the number of branches /loops in each function (which is already partially covered by the assembly line info), and a small amount of information on where within functions these branches tend to occur. However, combining these features into graph-level statistics would likely dilute these differences even further. It may, however, be useful to include one or more of these measures on the function call graph, but I am on the fence about its usefulness vs extra computation time/space required. I think for my purposes, the stats above work just fine

Parameters: **tokens** (*Union[Dict[str, int], AtomicData]*) – the token dictionary to use and possibly add to. Can also be an AtomicData object for atomic token dictionary file.

Returns: the compressed stats, a numpy 1-d uint8 array of shape (97 + len(tokens),)

Return type: np.ndarray

get_function(*address*, *raise_err=True*)

Returns the function in this CFG with the given address

Parameters:
- **address** (*Union[str, int, Addressable]*) – a string/integer memory address, or an addressable object (EG: CFGBasicBlock /CFGFunction)
- **raise_err** (*bool, optional*) – if True, will raise an error if the function with the given memory address was not found, otherwise will return None. Defaults to True.

Raises: **ValueError** – if the function with the given address could not be found

Returns: the function with the given address, or None if that function does not exist

Return type: Union[CFGFunction, None]

get_function_by_name(*name*, *raise_err=True*)

Returns the function in this `CFG` with the given name

NOTE: if the name of the function is None, then the expected string name to this method would be: "__UNNAMED_FUNC_%d" % func.address

Parameters:
- **name** (*str*) – the name of the function to get
- **raise_err** (*bool, optional*) – if True, will raise an error if the function with the given memory address was not found, otherwise will return None. Defaults to True.

Raises: **ValueError** – if the function with the given address could not be found

Returns: the function with the given address, or None if that function does not exist

Return type: Union[CFGFunction, None]

`insert_library`(*cfg*, *function_mapping*, *offset=None*)
Inserts the cfg of a shared library into this cfg

This will modify the memory addresses of *cfg* (adding an appropriate offset), then add all of the functions and basic blocks from *cfg* into this cfg. Finally, external functions in this cfg that have implemented functions in the function_mapping will have normal edges added.

NOTE: this assumes that no other libraries will be added later that depend on this one that is currently being added (otherwise, the external function edges might not be added properly). Make sure you add them in the correct order!

Parameters:
- **cfg** (*CFG*) – the cfg of the library to insert. It will be copied
- **function_mappping** (*Dict[str, int]*) – dictionary mapping known exported function names to their addresses within *cfg*. While we can sometimes determine these mappings from function names in the new *cfg*, that is not always the case (EG: stripping function names from binaries, or compilers/linkers emitting aliases for the functions in *cfg*), hence why this parameter exists. If you don't wish to add in new normal edges, or if you wish to add them in manually, you can pass an empty dictionary
- **offset** (*Optional[int]*) – if None, then the library will be inserted in the first available memory location. Otherwise this can be an integer memory address to insert the cfg at (this will raise an error if it can't fit there)

classmethod **`load`**(*path*)
Loads this CFG from path

`metadata` = *None*
Dictionary of metadata associated with this `CFG`

`normalize`(*normalizer*, *inplace=True*, *force_renormalize=False*)
Normalizes this cfg in-place.

Parameters:
- **normalizer** (*Union[str, Normalizer], optional*) – the normalizer to use. Can be a `Normalizer` object, or a string of a built-in normalizer to use.
- **inplace** (*bool, optional*) – whether or not to normalize inplace. Defaults to True.
- **force_renormalize** (*bool, optional*) – by default, this method will only normalize this cfg if the passed *normalizer* is != *self.normalizer*. However if *force_renormalize=True*, then this will be renormalized even if it has been previously normalized with the same normalizer. Defaults to False.

Returns: this `CFG` normalized

Return type: CFG

`normalizer` = *None*
The normalizer used to normalize assembly lines in this `CFG`, or None if they have not been normalized

property **`num_asm_lines`**
The number of asm lines across all blocks in this cfg

property **`num_blocks`**
The number of basic blocks in this cfg

property **`num_edges`**
The number of edges in this cfg

property **`num_functions`**
The number of functions in this cfg

`save`(*path*)
Saves this CFG to path

`to_adjacency_matrix`(*type: str = 'np'*, *sparse: bool = False*)
Returns an adjacency matrix representation of this cfg's graph connections

Currently is slow because I just convert to a MemCFG, then call that object's to_adjacency_matrix(). I should probably speed this up at some point…

Connections will be directed and have values:

- 0: No edge
- 1: Normal edge
- 2: Function call edge

See `bincfg.memcfg.to_adjacency_matrix()` for more details

Parameters:

- **type** (*str, optional*) –

  the type of matrix to return. Defaults to 'np'. Can be:

  - 'np'/'numpy' for a numpy ndarray (dtype: np.int32)
  - 'torch'/'pytorch' for a pytorch tensor (type: LongTensor)

- **sparse** (*bool, optional*) –

  whether or not the return value should be a sparse matrix. Defaults to False. Has different behaviors based on type:

  - numpy array: returns a 2-tuple of sparse COO representation (indices, values).
    NOTE: if you want sparse CSR format, you already have it with self.graph_c and self.graph_r
  - pytorch tensor: returns a pytorch sparse COO tensor.
    NOTE: not using sparse CSR format for now since it seems to have less documentation/supportedness.

Returns: an adjacency matrix representation of this `CFG`

Return type: Union[np.ndarray, torch.Tensor]

`to_cfg_dict`()

Converts this cfg to a dictionary of cfg information

The cfg dictionary will have the structure:

```
{
    'normalizer': the string name of the normalizer used,
    'metadata': a dictionary of metadata,
    'functions': {

        func_address_1: {
            'name' (str): string name of the function, or None if it has no name,
            'is_extern_func' (bool): True if this function is an extern function, False otherwise.
                An extern function is one that is located in an external library intended to be
found at
                runtime, and that doesn't have its code here in the CFG, only a small function
meant to jump to
                the external function when loaded at runtime

            'blocks': {
                block_address_1: {
                    'labels' (Set[str]): a set of string labels for nodes, empty meaning it is
unlabeled
                    'edges_out' (Tuple[Tuple[int, str], ...]): tuple of all outgoing edges. Each
'edge' is a tuple
                        of (other_basic_block_address: int, edge_type: str), where `edge_type` can
be 'normal'
                        for a normal edge and 'function_call' for a function call edge
                    'asm_lines' (Tuple[Tuple[int, str], ...]): tuple of all assembly lines in this
block. Each
                        assembly line is a (address, line) tuple where `address` is the integer
address of that
                        assembly line, and `line` is a cleaned, space-separated string of tokens
in that assembly
                        line
                },

                block_address_1: ...,
                ...
            }

        },

        func_address_2: ...,
        ...
    }
}
```

- func_address_X: integer address of that function
- block_address_X: integer address of that block

to_networkx()

Converts this CFG to a networkx DiGraph() object

Requires that networkx be installed.

Creates a new MultiDiGraph() and adds as attributes to that graph:

- 'normalizer': string name of normalizer, or None if it had none
- 'metadata': a dictionary of metadata
- functions: a dictionary mapping integer function addresses to named tuples containing its data with the structure ('name': Union[str, None], 'is_extern_func': bool, 'blocks': Tuple[int, ...]).
    - The 'name' element (first element) is a string name of the function, or None if it doesn't have a name
    - The 'is_extern_func' element (second element) is True if this function is an extern function, False otherwise. An extern function is one that is located in an external library intended to be found at runtime, and that doesn't have its code here in the CFG, only a small function meant to jump to the external function when loaded at runtime
    - The 'blocks' element (third element) is an arbitrary-length tuple of integers, each integer being the memory address (equivalently, the block_id) of a basic block that is a part of that function. Each basic block is only part of a single function, and each function should have at least one basic block

NOTE: the ADDRESS value will be and uppercase hex starting with a '0x'

NOTE: we use a multidigraph because edges are directed (in order of control flow), and it is theoretically possible (and occurs in some data) to have a node that calls another node, then has a normal edge back out to it. This has occured in some libc setup code

Then, each basic block will be added to the graph as nodes. Their id in the graph will be their integer address. Each block will have the following attributes:

- 'asm_lines' (Tuple[Tuple[int, str]]): tuple of assembly lines. Each assembly line is a (address, line) tuple where *address* is the integer address of that assembly line, and *line* is a cleaned, space-separated string of tokens in that assembly line
- 'labels' (Set[str]): a set of string labels for nodes, empty meaning it is unlabeled

Finally, all edges will be added (directed based on control flow direction), and with the attributes:

- 'edge_type' (str): the edge type, will be 'normal' for normal edges and 'function_call' for function call edges

classmethod **uncompress_stats**(*stats*, *dtype=<class 'numpy.uint32'>*)

Uncompressed the stats from cfg.get_compressed_stats()

Will return a numpy array with specified dtype (defaults to np.uint32) of stats in the same order they appreared in get_compressed_stats(). The size will decrease by around 12 indices as the initial 4-byte values are converted back into a one-index integer.

Parameters:
- **stats** (*np.ndarray*) – either a 1-d or 2-d numpy array of stats. If 2-d, then it is assumed that these are multiple stats for multiple cfgs, one cfg per row
- **dtype** (*np.dtype*) – the numpy dtype to return as. Defaults to np.uint32

Returns: either a 1-d or 2-d numpy array of uncompressed stats, depending on what was passed to *stats*

Return type: np.ndarray

exception bincfg.cfg.cfg.**InvalidInsertionMemoryAddressError**

Bases: Exception

bincfg.cfg.cfg_basic_block module

class bincfg.cfg.cfg_basic_block.**CFGBasicBlock**(*parent_function=None*, *address=None*, *edges_in=None*, *edges_out=None*, *asm_lines=None*, *labels=None*)

Bases: object

A single basic block in a CFG.

Can be initialized empty, or with attributes. Assumes its memory address is always unique within a cfg.

NOTE: these objects should not be pickled/copy.deepcopy()-ed by themselves, only as a part of a cfg

Parameters:
- **parent_function** (*CFGFunction*) – the CFGFunction this basic block belongs to
- **address** (*Union[int, str, Addressable]*) – the memory address of this CFGBasicBlock. Should be unique to the CFG that contains it
- **edges_in** (*Optional[Iterable[CFGEdge]]*) – an iterable of incoming CFGEdge objects
- **edges_out** (*Optional[Iterable[CFGEdge]]*) – an iterable of outgoing CFGEdge objects
- **asm_lines** (*Optional[Union[Iterable[Tuple[int, str]], Iterable[Tuple[int, Tuple[str]]]]]*) – an iterable of assembly lines present at this basic block. Each element should be a 2-tuple of (address: int, instruction: inst_strs), where *inst_strs* can either be a single, non-normalized string, or a tuple of normalized strings
- **labels** (*Optional[Iterable[int]]*) – an iterable of integer labels for this basic block. Labels are indices in the bincfg.labelling.NODE_LABELS list of node labels. Duplicate labels will be ignored

address = *None*

The unique integer memory address of this CFGBasicBlock

property **all_edges**

Returns a set of all edges in this basic block

property **asm_counts**

A collections.Counter of all unique assembly lines/tokens and their counts in this basic block

asm_lines = *None*

List of assembly instructions in this CFGBasicBlock

Each element is a tuple of (address, instruction). Each *instruction* is either a string (if this block has not yet been normalized) or a tuple of strings (if this block has been normalized)

property **asm_memory_addresses**
A set containing all memory addresses for assembly lines in this block

calls(*address*)
Checks if this block calls the given address

IE: checks if this block has an outgoing *function_call* edge to the given address

Parameters: **address** (*Union[str, int, Addressable]*) – a string/integer memory address, or an addressable object (EG: CFGBasicBlock /CFGFunction)

Returns: True if this block calls the given address, False otherwise

Return type: bool

edges_in = *None*
List of incoming `CFGEdge`'s

edges_out = *None*
List of outgoing `CFGEdge`'s

get_sorted_edges(*edge_types=None*, *direction=None*, *as_sets=False*)
Returns a tuple of sorted lists of edges (sorted by address of the "other" block) of each type/direction in this block

Will return edge lists ordered first by edge type (their order of appearance in the cfg_edge.EdgeType enum), then by direction ('in', then 'out'). Unless, if *edge_types* is passed, then only those edge types will be returned and the edge lists will be returned in the order of the edge types in *edge_types*, then by direction ('in', then 'out').

For example, with *edge_types=None* and *direction=None*, this would return the 4-tuple of: (normal_edges_in, normal_edges_out, function_call_edges_in, function_call_edges_out) Where each element is a list of CFGEdge objects.

Parameters:
- **edge_types** (*Union[EdgeType, str, Iterable[Union[EdgeType, str]], None], optional*) – either an edge type or an iterable of edge types. Only edges with one of these types will be returned. If not None, then the edge lists will be returned sorted based on the order of the edge types listed here, then by direction. Defaults to None.
- **direction** (*Union[str, None], optional*) – the direction to get (strings 'in'/'from' or 'to'/'out'), or None to get both (in order ['in', 'out']). Defaults to None.
- **as_sets** (*bool, optional*) – if True, then this will return unordered sets of edges instead of sorted lists. This may save a ~tiny~ bit of time in the long run, but will hinder deterministic behavior of this method. Defaults to False.

Returns: a tuple of lists/sets of CFGEdge's

Return type: Union[Tuple[List[CFGEdge], …], Tuple[Set[CFGEdge], …]]

has_edge(*address*, *edge_types=None*, *direction=None*)
Checks if this block has an edge from/to the given address

Parameters:
- **address** (*AddressLike*) – a string/integer memory address, or an addressable object (EG: `CFGBasicBlock`/`CFGFunction`).
- **edge_types** (*Union[EdgeType, str, Iterable[Union[EdgeType, str]], None], optional*) – either an edge type or an iterable of edge types. Only edges with one of these types will be considered. If None, then all edge types will be considered. Defaults to None.
- **direction** (*Union[str, None], optional*) – the direction to check (strings 'in'/'from' or 'to'/'out'), or None to check both. Defaults to None.

Returns: True if this block has an edge from/to the given address, False otherwise

Return type: bool

has_edge_from(*address*, *edge_types=None*)
Checks if this block has an incoming edge from the given address

Parameters:
- **address** (*Union[str, int, Addressable]*) – a string/integer memory address, or an addressable object (EG: CFGBasicBlock /CFGFunction)
- **edge_types** (*Union[EdgeType, str, Iterable[Union[EdgeType, str]], None], optional*) – either an edge type or an iterable of edge types. Only edges with one of these types will be considered. Defaults to None.

Returns: True if this block has an incoming edge from the given address, False otherwise

Return type: bool

has_edge_to(*address*, *edge_types=None*)
Checks if this block has an outgoing edge to the given address

Parameters:
- **address** (*Union[str, int, Addressable]*) – a string/integer memory address, or an addressable object (EG: CFGBasicBlock /CFGFunction)
- **edge_types** (*Union[EdgeType, str, Iterable[Union[EdgeType, str]], None], optional*) – either an edge type or an iterable of edge types. Only edges with one of these types will be considered. Defaults to None.

Returns: True if this block has an outgoing edge to the given address, False otherwise

Return type: bool

property **instruction_addresses**
Returns a set of addresses for all instructions in this basic block

property **is_function_call**
True if this block is a function call, False otherwise

Checks if this block has one or more outgoing function call edges

property **is_function_entry**
True if this block is a function entry block, False otherwise

Specifically, returns True if this block's address matches its parent function's address. If this block has no parent, False is returned.

property **is_function_jump**
True if this block is a function jump, False otherwise

Checks if this block has a 'jump' instruction to a basic block in a different function. Specifically, checks if this block has an outgoing EdgeType.NORMAL edge to a basic block who's parent_function has an address different than this basic block's parent_function's address.

property **is_function_return**
True if this block is a function return, False otherwise

Specifically, returns True iff this block's final assembly instruction is a 'return' instruction (IE: if the final assembly instruction re.fullmatch()' s RE_RETURN_INSTRUCTION), or if this is a block in an external function

property **is_multi_function_call**
True if this block is a multi-function call, False otherwise

IE: this block has either two or more function call edges out, or one function call and two or more normal edges out

property **is_padding_node**
Returns True if this is a padding node (contains only NOP instructions for memory alignment)

labels = *None*
Set of all labels for this `CFGBasicBlock`

Labels should be indices in the `bincfg.labelling.NODE_LABELS` list of node labels

property **normal_children**
Returns a set of all basic blocks that are normal children to this one (have an incoming normal edge from this block, and are within the same function

property **normal_parents**
Returns a set of all basic blocks that are normal parents to this one (have an outgoing normal edge to this block, and are within the same function

property **num_asm_lines**
The number of assembly lines in this basic block

property **num_edges**
The number of edges out in this basic block

property **num_edges_in**
The number of incoming edges in this basic block

property **num_edges_out**
The number of outgoing edges in this basic block

parent_function = *None*
The parent function containing this `CFGBasicBlock`

remove_edge(*edge*)
Removes the given edge from this block's edges (both incoming and outgoing)

Parameters: **edge** (*CFGEdge*) – the CFGEdge to remove

Raises: **ValueError** – if the edge doesn't exist in the incomming/outgoing edges

bincfg.cfg_dataset module

class `bincfg.cfg_dataset.`**`CFGDataset`**(*cfg_data=None*, *normalizer=None*, *load_path=None*, *max_files=None*, *allow_multiple_norms=False*, *progress=False*, *metadata=None*, ***add_data_kwargs*)

Bases: `object`

A dataset of `CFG`'s.

Parameters:
- **cfg_data** (*Optional[Union[CFG, CFGDataset, Iterable]]*) – a `CFG`, `CFGDataset` or iterable of `CFG`'s or `CFGDataset`'s to add to this dataset, or None to initialize this `CFGDataset` empty
- **normalizer** (*Optional[Union[str, Normalizer]]*) – if not None, then a normalizer to use. Will normalize all incoming `CFG`'s if they do not already have the name normalization (will attempt to renormalize incoming `CFG`'s if they already have a normalization). Can be a `Normalizer` object or string.
- **load_path** (*str*) – if not None, loads all files in this directory that end with '.txt' or '.dot'. Will raise an error if there are no files. Will ignore any files that end with '.txt' or '.dot', but cannot be parsed.
- **max_files** (*Optional[int]*) – stops after loading this many files. If None, then there is no max
- **allow_multiple_norms** (*bool*) – by default, `CFGDataset` will only allow unnormalized cfg's when *normalizer=None* (if *normalizer* is not None, then any normalized cfg added will be renormalized). Setting *allow_multiple_norms* to True will allow this `CFGDataset` to store cfg data with any normalization method (assuming *normalizer=None*)
- **progress** (*bool*) – if True, will show a progressbar when loading cfg's from load_path
- **metadata** (*Optional[Dict]*) – a dictionary of metadata to attach to this CFGDataset NOTE: passed dictionary will be shallow copied
- **add_data_kwargs** (*Any*) – extra kwargs to pass to add_data while adding cfgs

`add_data`(**cfg_data*, *inplace=True*, *force_renormalize=False*, *progress=False*)

Adds data to this dataset

Parameters:
- **cfg_data** (*Union[CFG, CFGDataset, Iterable]*) – arbitrary amount of `CFG`/`CFGDataset`'s, or iterables of them, to add to this dataset
- **inplace** (*bool, optional*) – whether or not to normalize the incoming cfg_data inplace. Defaults to True.
- **force_renormalize** (*bool, optional*) – by default, this method will only normalize cfg's whose .normalizer != to this dataset's normalizer. However if *force_renormalize=True*, then all cfg's will be renormalized even if they have been previously normalized with the same normalizer. Defaults to False.
- **progress** (*bool, optional*) – if True, will show a progressbar when adding multiple cfgs. Defaults to False.

Raises:
- **TypeError** – when attempting to add something that is not a `CFG`, `CFGDataset`, or iterables of them
- **ValueError** – when attempting to use multiple different normalizers and *self.allow_multiple_norms=False*

property **`asm_counts`**

A collections.Counter() of all unique assembly lines and their counts accross all cfg's in this dataset

`cfgs` = *None*

The list of all cfgs in this dataset

classmethod **`load`**(*path*)

Loads this CFGDataset from path

`metadata` = *None*

A dictionary of metadata associated with this `CFGDataset`

`normalize`(*normalizer=None*, *inplace=True*, *force_renormalize=False*, *progress=False*)

Normalize this `CFGDataset`.

Parameters:
- **normalizer** (*Union[str, Normalizer]*) – the normalizer to use. Can be a `Normalizer` object, or a string, or None to use the default BaseNormalizer(). Defaults to None.
- **inplace** (*bool, optional*) – by default, normalizes this dataset inplace (IE: without copying objects). Can set to False to return a copy. Defaults to True.
- **force_renormalize** (*bool, optional*) – by default, this method will only normalize cfg's whose .normalizer != to the passed normalizer. However if *force_renormalize=True*, then all cfg's will be renormalized even if they have been previously normalized with the same normalizer.. Defaults to False.
- **progress** (*bool, optional*) – if True, will show a progressbar while normalizing. Defaults to False.

Returns: this dataset normalized

Return type: CFGDataset

`normalizer` = *None*

The normalizer used in this dataset, or None if there is no normalizer

property **`num_asm_lines`**

Return total number of assembly lines across all cfg's

property **num_blocks**
Return total number of blocks across all cfg's

property **num_cfgs**
Return the number of cfgs in this dataset

property **num_edges**
Return total number of edges across all cfg's

property **num_functions**
Return total number of functions across all cfg's

save(*path*)
Saves this CFGDataset to path

bincfg.cfg_edge module

Classes/Methods involving edges in a `CFG` object

class `bincfg.cfg_edge.`**CFGEdge**(*from_block*, *to_block*, *edge_type*)
Bases: `object`

A single immutable edge in a `CFG` object

Parameters:
- **from_block** (*CFGBasicBlock*) – 'from' `CFGBasicBlock` object
- **to_block** (*CFGBasicBlock*) – 'to' `CFGBasicBlock` object
- **edge_type** (*Union[EdgeType, str]*) –

  the edge type. can be either an `EdgeTypes` object, or a string. String values include:
  - 'normal': a `EdgeTypes.NORMAL` edge
  - 'function_call': a `EdgeTypes.FUNCTION_CALL` edge

edge_type = *None*
the edge type

from_block = *None*
the 'from' `CFGBasicBlock` object

property **is_branch**
True if this edge is one of a branching instruction, False otherwise

Specifically, returns True if this edge's *from_block* has exactly two outgoing edges, both of which are 'normal' edges. Sometimes, it is possible for blocks to have more than two 'normal' edges out (IE: jump tables), and those are NOT considered branches and this method would return False

property **is_function_call_edge**
True if this is a 'function_call' edge type, False otherwise

property **is_normal_edge**
True if this is a 'normal' edge type, False otherwise

to_block = *None*
the 'to' `CFGBasicBlock` object

class `bincfg.cfg_edge.`**EdgeType**(*value*)
Bases: `Enum`

Enum for different edge types for `CFGBasicBlock` objects.

FUNCTION_CALL = *re.compile('(?:call|fc|function|func)(?:[-_]?call)?(?:[-_]?edge)?')*
an edge going from a basic block to another basic block in another function (or the same function).

The outgoing edge should always connect to a function entry block (IE: that block's `.is_function_entry` would be True).

NORMAL = *re.compile('(?:normal|plain|jump|branch)(?:[-_]?edge)?')*
a normal edge as a result of some branching/jumping instruction, or plain continuation to a next block

(IE: an edge of control flow that does not involve calling a function)

`bincfg.cfg_edge.`**get_edge_type**(*edge_type*)
Returns the edge type (instance of EdgeTypes enum class)

Parameters: **edge_type** (*Union[EdgeType, str]*) – can be either an EdgeTypes object, or a string. String values include: - 'normal': a EdgeTypes.NORMAL edge - 'function_call': a EdgeTypes.FUNCTION_CALL edge

Raises:
- **ValueError** – for an unknown EdgeType string
- **TypeError** – for a bad *edge_type* type

Returns: the given *edge_type* as a class from the EdgeType enum

Return type: EdgeType

bincfg.cfg_function module

class `bincfg.cfg_function.`**`CFGFunction`**(*parent_cfg=None*, *address=None*, *name=None*, *blocks=None*, *is_extern_func=False*)

Bases: `object`

A single function in a `CFG`

Can be initialized empty, or by passing kwarg values.

NOTE: these objects should not be pickled/copy.deepcopy()-ed by themselves, only as a part of a cfg

Parameters:
- **parent_cfg** (*CFG*) – the parent `CFG` object to which this `CFGFunction` belongs
- **address** (*Union[str, int, Addressable]*) – the memory address of this function
- **name** (*Optional[str]*) – the string name of this function, or None if it doesn't have one
- **blocks** (*Optional[Iterable[CFGBasicBlock]]*) – if None, will be initialized to an empty list, otherwise an iterable of `CFGBasicBlock` objects that are within this function
- **is_extern_func** (*bool*) – if True, then this function is an external function (a dynamically loaded function)

`address` = *None*
the integer memory address of this function

property **`asm_counts`**
A `collections.Counter` of all unique assembly lines and their counts in this function

`blocks` = *None*
list of all basic blocks in this function

property **`called_by`**
A list of `CFGBasicBlock`'s that call this function

Specifically, the list of all `CFGBasicBlock` objects in this `CFGFunction.parent_cfg` cfg object that call this function. If this `CFGFunction` has no parent, then the empty list will be returned.

NOTE: this is computed dynamically each call (as `CFG` objects are mutable), so it may be useful to compute it once per function and save it if needed

property **`function_entry_block`**
The `CFGBasicBlock` that is the function entry block

Specifically, returns the first `CFGBasicBlock` found that has the same address as this function (there ~should~ only be one as each basic block ~should~ have a unique memory address)

property **`is_extern_function`**
True if this function is an external function, False otherwise

property **`is_intern_function`**
True if this function is an internal function, False otherwise

property **`is_recursive`**
True if this function calls itself at some point

Specifically, if at least one `CFGBasicBlock` in this `CFGFunction.blocks` list has an *edges_out* function call address that is equal to this `CFGFunction`'s address

property **`is_root_function`**
True if this function is not called by any other functions, False otherwise

`name` = *None*
the string name of this function, or `None` if it doesn't have a name

property **`nice_name`**
Returns the name of this function, returning ("__UNNAMED_FUNC_%d" % self.address) if the name is None

property **`num_asm_lines`**

The total number of assembly lines across all blocks in this function

property **num_blocks**
The number of basic blocks in this function

property **num_fc_edges**
Returns the number of function call edges in/out of this function

parent_cfg = *None*
the parent `CFG` object to which this `CFGFunction` belongs

property **symbol_name**
Returns the symbol name of this function if it is an external function, or None if it isnt or the name is None

bincfg.cfg.cfg_parsers module

Functions to parse cfg inputs into `CFG` objects.

exception `bincfg.cfg.cfg_parsers.`**CFGParseError**
Bases: `Exception`

`bincfg.cfg.cfg_parsers.`**get_asm_from_node_label**(*label*)
Converts a node's label into a list of assembly lines at that basic block.

Parameters: **label** (*str*) – the unparsed string label

Returns: a list of 2-tuples of (memory_address, asm_instruction)

Return type: List[Tuple[int, str]]

`bincfg.cfg.cfg_parsers.`**parse_cfg_data**(*cfg*, *data*)
Parses the incoming cfg data. Infers type of data

Parameters:
- **cfg** (*CFG*) – the cfg to parse into
- **data** (*Union[str, Sequence[str], TextIO, pd.DataFrame]*) –

  the data to parse, can be:

  - string: either string with newline characters that will be split on all newlines and treated as either a text or graphviz rose input, or a string with no newline characters that will be treated as a filename. Filenames will be opened as ghidra parquet files if they end with either '.pq' or '.parquet', and text/graphviz rose input otherwise
  - Sequence of string: will be treated as already-read-in text/graphviz rose input
  - open file object: will be read in using *.readlines*, then treated as text/graphviz rose input
  - pandas dataframe: will be parsed as ghidra parquet file

Raises:
- **ValueError** – bad `str` filename, or an unknown file start string
- **TypeError** – bad `data` input type
- **CFGParseError** – if there is an error during CFG parsing (but data type was inferred correctly)

`bincfg.cfg.cfg_parsers.`**parse_ghidra_parquet**(*cfg*, *data*)
Parses a CFG from a single ghidra output parquet file

Parameters:
- **cfg** (*CFG*) – prebuilt `CFG` object to modify
- **data** (*Union[str, pd.DataFrame]*) – a pandas dataframe loaded from a ghidra parquet file, or a string path to one to load

Raises: **TypeError** – bad *data* input type

Returns: parsed `CFG` object

Return type: CFG

`bincfg.cfg.cfg_parsers.`**parse_rose_gv**(*cfg*, *lines*)
Reads input as a graphviz file

Parameters:
- **cfg** (*CFG*) – an empty/loading CFG() object
- **lines** (*str, Iterable[str], TextIO*) – the data to parse. Can be a string (which will be split on newlines to get each individual line), a list of string (each element will be considered one line), or an open file to call *.readlines()* on

Raises: **CFGParseError** – when the file cannot be parsed correctly

`bincfg.cfg.cfg_parsers.`**parse_rose_txt**(*cfg*, *lines*)
Reads input as a .txt file

Parameters:
- **cfg** (*CFG*) – an empty/loading CFG() object
- **lines** (*str, Iterable[str], TextIO*) – the data to parse. Can be a string (which will be split on newlines to get each individual line), a list of string (each element will be considered one line), or an open file to call *.readlines()* on

Raises: **CFGParseError** – when file does not fit expected format

bincfg.cfg.mem_cfg module

class `bincfg.cfg.mem_cfg.`**`MemCFG`**(*cfg*, *normalizer=None*, *inplace=False*, *using_tokens=None*, *force_renormalize=False*)

Bases: `object`

A CFG that is more memory/speed efficient.

Keeps only the bare minimum information needed from a CFG. Stores edge connections in a CSR-like format.

Parameters:
- **cfg** (*CFG*) – a CFG object. Can be a normalized or un-normalized. If un-normalized, then it will be normalized using the *normalizer* parameter.
- **normalizer** (*Optional[Union[str, Normalizer]]*) – the normalizer to use to normalize the incoming CFG (or None if it is already normalized). If the incoming CFG object has already been normalized, and *normalizer* is not None, then this will attempt to normalize the CFG again with this normalizer
- **inplace** (*bool*) – if True and cfg needs to be normalized, it will be normalized inplace
- **using_tokens** (*Union[Dict[str, int], AtomicTokenDict]*) – if not None, then a dictionary mapping token strings to integer values. Any tokens in cfg but not in using_tokens will be added. Can also be an AtomicTokenDict for atomic updates to tokens
- **force_renormalize** (*bool*) – by default, this method will only normalize cfg's whose .normalizer != to the passed normalizer. However if *force_renormalize=True*, then all cfg's will be renormalized even if they have been previously normalized with the same normalizer.

class **`BlockInfoBitMask`**(*value*)

Bases: `Enum`

An Enum for block info bit masks

Each value is a tuple of the bit mask for that boolean, and a function to call with the block that returns a boolean True if that bit should be set, False otherwise. If True, then that bit will be '1' in that block's block_flags int.

`IS_FUNCTION_CALL` = *(1, <function MemCFG.BlockInfoBitMask.<lambda>>)*

Bit set if this block is a function call. See `is_function_call()`

`IS_FUNCTION_ENTRY` = *(2, <function MemCFG.BlockInfoBitMask.<lambda>>)*

Bit set if this block is a function entry. See `is_function_entry()`

`IS_FUNCTION_JUMP` = *(16, <function MemCFG.BlockInfoBitMask.<lambda>>)*

this block has a jump instruction that resolves to a basic block in a separate function. See `is_function_jump()`

Type: Bit set if this block is a function jump. IE

`IS_FUNCTION_RETURN` = *(4, <function MemCFG.BlockInfoBitMask.<lambda>>)*

Bit set if this block is a function return. See `is_function_return()`

`IS_IN_EXTERN_FUNCTION` = *(8, <function MemCFG.BlockInfoBitMask.<lambda>>)*

Bit set if this block is within an external function. See `is_extern_function()`

`IS_MULTI_FUNCTION_CALL` = *(32, <function MemCFG.BlockInfoBitMask.<lambda>>)*

this block has either two or more function call edges out, or one function call and two or more normal edges out. See `is_multi_function_call()`

Currently not setting the block here in _block_flags_int(), but instead in MemCFG initialization in order to save time (we don't have to compute get_sorted_edges() multiple times)

Type: Bit set if this block is a multi-function call. IE

`asm_lines` = *None*

Assembly line information

A contiguous 1-d numpy array of shape (num_asm_lines,) of integer assembly line tokens. Dtype is the smallest unsigned dtype needed to store the largest token value in this `MemCFG`

To get the assembly lines for some block index *block_idx*, you must get the assembly line indices from `block_asm_idx`, and use those to slice the assembly lines:

```
>>> block_idx = 7
>>> memcfg.asm_lines[memcfg.block_asm_idx[block_idx]:memcfg.block_asm_idx[block_idx + 1]]
```

Also see `get_block_asm_lines()`

block_asm_idx = *None*
Indices in `asm_lines` that correspond to the assembly lines for each basic block in this `MemCFG`

A 1-d numpy array of shape (num_blocks + 1,). Dtype is the smallest unsigned dtype needed to store the value *num_asm_lines*. Assembly tokens for a block at index *i* would have a start index of *block_asm_idx[i]* and an end index of *block_asm_idx[i + 1]* in `asm_lines`.

block_flags = *None*
Integer of bit flags for each basic block

A 1-d numpy array of shape (num_blocks,) where each element is an integer of bit flags. See `BlockInfoBitMask` for more info. Dtype is the smallest unsigned dtype with enough bits to store all flags in `BlockInfoBitMask`

Also see `get_block_flags()`

block_func_idx = *None*
Integer ids for the function that each basic block belongs to

A 1-d numpy array of shape (num_blocks,) where each element is a function id for the block at that index. The id can be found in `function_name_to_idx`. Dtype is the smallest unsigned dtype needed to store the value *num_functions*

Also see `get_block_function_idx()` and `get_block_function_name()`

block_labels = *None*
Dictionary mapping block indices to integer block label bit flags

Only blocks that have known labels will be in this dictionary. The bit flags integer will have the bit set for each index in the *bincfg.labelling.NODE_LABELS* list. EG: the 0-th element would correspond to the 1's bit, the 1-th element would correspond to the 2's bit, etc.

Also see `get_block_labels()`

function_name_to_idx = *None*
Dictionary mapping string function names to their integer ids used in this `MemCFG`

get_block(*block_idx*, *as_dict=False*)
Returns all the info associated with the given block index

Parameters:
- **block_idx** (*int*) – integer block index
- **as_dict** (*bool*) – if True, will return a dictionary of the values with the same names in the "Returns" section

Returns:
the block info -
(asm_lines, edges_out, edge_types, function_idx, is_function_call, is_function_entry, is_function_return, is_extern_function, is_function_jump, is_block_multi_function_call, labels_list)

Return type: Tuple[np.ndarray, np.ndarray, int, bool, bool, bool, bool, bool, List[int]]

get_block_asm_lines(*block_idx*)
Get the asm lines associated with this block index

Parameters: **block_idx** (*int*) –

Returns: a numpy array of assembly tokens

Return type: np.ndarray

get_block_edges_out(*block_idx*, *ret_edge_types=False*)
Get numpy array of block indices for all edges out associated with the given block index

Parameters:
- **block_idx** (*int*) – integer block index
- **ret_edge_types** (*bool*) –

  if True, will also return a numpy array (1-d, dtype np.uint8) containing the edge type values for each edge with values:

  - 1: normal edge
  - 2: function call edge

Returns: a numpy array of block indices

Return type: np.ndarray

get_block_flags(*block_idx*)
Get all block flags for the given block index

Parameters: **block_idx** (*int*) – integer block index

Returns:
(is_block_function_call, is_block_function_start, is_block_function_return,
is_block_extern_function, is_block_function_jump, is_block_multi_function_call)

Return type: Tuple[bool, bool, bool, bool, bool]

get_block_function_idx(*block_idx*)
Get the function index for the given block index

Parameters: **block_idx** (*int*) – integer block index

Returns: the integer function index for the given block index

Return type: int

get_block_function_name(*block_idx*)
Get the function name for the given block index

Functions without names will start with '__unnamed_func__'

Parameters: **block_idx** (*int*) – integer block index

Returns: the function name for the given block index

Return type: str

get_block_info(*block_idx*, *as_dict=False*)
Alias for `get_block()`. Returns all the info associated with the given block index

Parameters:
- **block_idx** (*int*) – integer block index
- **as_dict** (*bool*) – if True, will return a dictionary of the values with the same names in the "Returns" section

Returns:
the block info -
(asm_lines, edges_out, edge_types, function_idx, is_function_call, is_function_entry, is_function_return, is_extern_function, is_function_jump, is_block_multi_function_call, labels_list)

Return type: Tuple[np.ndarray, np.ndarray, int, bool, bool, bool, bool, bool, List[int]]

get_block_labels(*block_idx*)
Returns a list of integer block labels for the given block_idx.

Labels will be integers of the indices in NODE_LABELS. IE: if this CFG has labels ['ecryption', 'file_io', 'network_io', 'error_handler', 'string_parser'], and a block at block_idx has block_labels of [0, 3], then that block would be both an 'ecryption' block and a 'error_handler' block. If a block has no labels ([]), then it should be assumed that we don't know what labels it should have, as opposed to it having no labels.

Parameters: **block_idx** (*int*) – integer block index

Returns: list of integer labels for the given block index

Return type: List[int]

get_coo_indices()
Returns the COO indices for this MemCFG

Returns a 2-d numpy array of shape (num_edges, 2) of dtype np.int32. Each row is an edge, column 0 is the 'row' indexer, and column 1 is the 'column' indexer. EG:

```
original = np.array([
    [0, 1],
    [1, 1]
])

coo_indices = np.array([
    [0, 1],
    [1, 0],
    [1, 1]
])
```

NOTE: this returns as type np.int32 since pytorch can be finicky about what dtypes it wants NOTE: pytorch sparse_coo_tensor's indicies are the transpose of the array this method returns

Returns: the coo indices

Return type: np.ndarray

get_edge_values()
Returns the edge type values

Returns a 1-d numpy array of length self.num_edges and dtype np.int32 containing an integer type for each edge depending on if it is a normal/function call/call return edge:

Edges are directed and have values:

- 0: No edge
- 1: Normal edge
- 2: Function call edge

NOTE: this returns as type np.int32 since pytorch can be finicky about what dtypes it wants

Returns: the edge type values

Return type: np.ndarray

graph_c = *None*
Array containing all of the outgoing edges for each block in order

1-D numpy array of shape (num_edges,). Dtype will be the smallest unsigned dtype required to store the value *num_blocks + 1*. Each element is a block index to which that edge connects. Edges will be in the order they appear in each block's `edges_out` attribute, for each block in order of their block_idx.

Also see `get_edges_out()`

NOTE: this also contains information on which types of edges they are inherently. If the block is NOT a function call (stored as bit flag in the block_info array), then all edges for that block are normal edges. If it IS a function call, then there are 3 cases:

1. it has one outgoing edge: that edge is always a function call
2. it has two outgoing edges, one function call, one normal: the first edge is the function call edge, the second is a normal edge
3. it has >2 outgoing edges, or 2 function call edges: the edges will be listed first by function call edges, then by normal edges, with a separator inbetween. The separator will have the max unsigned int value for graph_c's dtype. This is why we use the dtype that can store *num_blocks + 1*, since we need this extra value just in case. Whatever exactly it means for a basic block to have >2 outgoing edges while being a function call is left up to the user. Possibly due to call operators with non-explicit operands (eg: register memory locations)?

graph_r = *None*
Array containing information on the number of outgoing edges for each block

1-D numpy array of shape (num_edges + 1,). Dtype will be the smallest unsigned dtype required to store the value *num_edges*. This array is a cumulative sum of the number of edges for each basic block. One could get all of the outgoing edges for a block using:

```
>>> start_idx = memcfg.graph_r[block_idx]
>>> end_idx = memcfg.graph_r[block_idx + 1]
>>> edges = memcfg.graph_c[start_idx:end_idx]
```

Also see `get_edges_out()`

is_block_extern_function(*block_idx*)
True if this block is in an external function, False otherwise

is_block_function_call(*block_idx*)
True if this block is a function call, False otherwise

is_block_function_entry(*block_idx*)
True if this block is a function entry, False otherwise

is_block_function_jump(*block_idx*)
True if this block is a function jump, False otherwise

is_block_function_return(*block_idx*)
True if this block is a function return, False otherwise

is_block_labeled(*block_idx*, *label=None*)
Checks if the given block_idx is labeled

If label is None, returns True if the block at the given index has a label, False if it has no labels (self.block_labels[block_idx] == 0)
Otherwise returns True if the block at the given index has the given label, False if not

Parameters:
- **block_idx** (*int*) – integer block index
- **label** (*Union[str, int, None], optional*) – if not None, then the label to check the block_idx for. Can be either a string (whos lowercase name must be in NODE_LABELS), or an integer for the index in NODE_LABELS to check for. Otherwise if label is None, then this will check to see if the given block_idx is labeled at all. Defaults to None.

Raises:
- **ValueError** – for a bad/unknown *label* value
- **TypeError** – for a bad *label* type

Returns: True if the block has the label, False otherwise

Return type: bool

`is_block_multi_function_call`(*block_idx*)
True if this block is a multi-function call, False otherwise

classmethod **`load`**(*path*)
Loads a MemCFG from the given path

`metadata` = *None*
Dictionary of metadata associated with this MemCFG

`normalize`(*normalizer=None*, *inplace=True*)
Normalizes this memcfg in-place.

Parameters:
- **normalizer** (*Union[str, Normalizer], optional*) – the normalizer to use. Can be a `Normalizer` object, or a string, or None to use the default BaseNormalizer(). Defaults to None.
- **inplace** (*bool, optional*) – whether or not to normalize inplace. Defaults to True.
- **force_renormalize** (*bool, optional*) – by default, this method will only normalize this cfg if the passed *normalizer* is != *self.normalizer*. However if *force_renormalize=True*, then this will be renormalized even if it has been previously normalized with the same normalizer. Defaults to False.

Returns: this `MemCFG` normalized

Return type: MemCFG

`normalizer` = *None*
The normalizer used to normalize input before converting to `MemCFG`

Can be shared with a `MemCFGDataset` object if this `MemCFG` is a part of one

property **`num_asm_lines`**
The number of assembly lines in this `MemCFG`

property **`num_blocks`**
The number of blocks in this `MemCFG`

property **`num_edges`**
The number of edges in this `MemCFG`

property **`num_functions`**
The number of functions in this `MemCFG`

`save`(*path*)
Saves this MemCFG to the given path

`to_adjacency_matrix`(*type='np'*, *sparse=False*)
Returns an adjacency matrix representation of this memcfg's graph connections

Connections will be directed and have values:

- 0: No edge
- 1: Normal edge

- 2: Function call edge

See `bincfg.memcfg.to_adjacency_matrix()` for more details

Parameters:

- **type** (*str, optional*) –

  the type of matrix to return. Defaults to 'np'. Can be:

  - 'np'/'numpy' for a numpy ndarray (dtype: np.int32)
  - 'torch'/'pytorch' for a pytorch tensor (type: LongTensor)

- **sparse** (*bool, optional*) –

  whether or not the return value should be a sparse matrix. Defaults to False. Has different behaviors based on type:

  - numpy array: returns a 2-tuple of sparse COO representation (indices, values).
    NOTE: if you want sparse CSR format, you already have it with self.graph_c and self.graph_r
  - pytorch tensor: returns a pytorch sparse COO tensor.
    NOTE: not using sparse CSR format for now since it seems to have less documentation/supportedness.

Returns: an adjacency matrix representation of this `MemCFG`

Return type: Union[np.ndarray, torch.Tensor]

`tokens` = *None*
Dictionary mapping token strings to integer values used in this `MemCFG`

Can be shared with a `MemCFGDataset` object if this `MemCFG` is a part of one.

Can also be an AtomicTokenDict object for atomic token updates

bincfg.cfg.mem_cfg_dataset module

class `bincfg.cfg.mem_cfg_dataset.`**`MemCFGDataset`**(*cfg_data=None*, *using_tokens=None*, *normalizer=None*, *metadata=None*, ***add_data_kwargs*)
Bases: `object`

A CFGDataset that is more memory efficient

Parameters:

- **cfg_data** (*Optional[Union[CFG, CFGDataset, MemCFG, MemCFGDataset, Iterable]]*) – the data to use. Can be None for an empty dataset, or a *CFG*, *CFGDataset*, *MemCFG*, *MemCFGDataset*, or iterable of those values to add that data to this dataset
- **tokens** (*Optional[Union[Dict[str, int], AtomicTokenDict]]*) – if passed, will initialize the token dictionary to this dictionary of tokens (will be copied). Can be an AtomicTokenDict to use an atomic file token dictionary
- **normalizer** (*Optional[Union[str, Normalizer]]*) – the normalizer to use, or None to default to the normalizer of the first added CFG/MemCFG
- **metadata** (*Optional[Dict]*) – a dictionary of metadata to attach to this MemCFGDataset NOTE: passed dictionary will be shallow copied
- **add_data_kwargs** (*Any*) – kwargs to pass to self.add_data() when adding the passed cfg_data

`add_data`(**cfg_data*, *inplace=True*, *force_renormalize=False*, *progress=False*)
Adds data to this dataset

Parameters:

- **cfg_data** (*Union[CFG, MemCFG, CFGDataset, MemCFGDataset, Iterable]*) – arbitrary amount of CFG/MemCFG/CFGDataset /MemCFGDataset's, or iterables of them, to add to this dataset
- **inplace** (*bool, optional*) – whether or not to normalize the incoming cfg_data inplace. Defaults to True.
- **force_renormalize** (*bool, optional*) – by default, this method will only normalize cfg's whose .normalizer != to this dataset's normalizer. However if *force_renormalize=True*, then all cfg's will be renormalized even if they have been previously normalized with the same normalizer. Defaults to False.
- **mp** (*bool, optional*) – if True, will use multiprocessing to normalize cfgs. Defaults to False.
- **progress** (*bool, optional*) – if True, will show a progressbar when adding multiple cfgs. Defaults to False.

Raises: **TypeError** – if something other than a cfg/dataset is passed in *cfg_data*

`cfgs` = *None*
The list of all memcfgs in this dataset

classmethod **`load`**(*path*)
Loads this MemCFGDataset from path

`metadata` = *None*
A dictionary of metadata associated with this `MemCFGDataset`

`normalize`(*normalizer=None*, *inplace=True*, *force_renormalize=False*, *progress=False*)
Normalize this `MemCFGDataset`.

Parameters:

- **normalizer** (*Union[str, Normalizer]*) – the normalizer to use. Can be a `Normalizer` object, or a string, or None to use the default BaseNormalizer(). Defaults to None.
- **inplace** (*bool, optional*) – by default, normalizes this dataset inplace (IE: without copying objects). Can set to False to return a copy. Defaults to True.
- **force_renormalize** (*bool, optional*) – by default, this method will only normalize cfg's whose .normalizer != to the passed normalizer. However if *force_renormalize=True*, then all cfg's will be renormalized even if they have been previously normalized with the same normalizer.. Defaults to False.
- **progress** (*bool, optional*) – if True, will show a progressbar while normalizing. Defaults to False.

Returns: this dataset normalized

Return type: MemCFGDataset

`normalizer` = *None*

The normalizer used in this dataset, or None if there is no normalizer

property **`num_blocks`**

`remove_cfg`(*cfg_or_idx*)

Removes the given MemCFG (or index of MemCFG if cfg_or_idx is an integer) from this MemCFGDataset

Parameters: **cfg_or_idx** (*Union[MemCFG, int]*) – cfg or index to remove

`save`(*path*)

Saves this MemCFGDataset to path

`train_test_split`(*test_size=None*, *test_uids=None*)

Splits this MemCFGDataset into two: one for testing, and one for training.

Either test_size or test_uids, but not both, should be passed. If test_size, then this data will be randomly split with test_size% being allocated to test. Otherwise if test_uids is passed, then those uids will be considered the test set and all other CFG's will be the training set.

Parameters:

- **test_size** (*Union[float, None], optional*) – a float in range [0, 1] for test set percent, or None to not use
- **test_uids** (*Union[float, None], optional*) – a list of test uids, or None to not use

Raises:

- **NotImplementedError** – _description_
- **TypeError** – if none or both of *test_size* and *test_uids* is passed

Returns: tuple of (train_dataset, test_dataset)

Return type: Tuple[MemCFGDataset, MemCFGDataset]

`using_tokens` = *None*

A dictionary mapping string tokens to their integer values

Can be an AtomicTokenDict for atomic updates to tokens

Module contents