ChatGPT

Biq Mac Max-Cut Benchmark vs Project Graph Data

What Is the Biq Mac Max-Cut Benchmark?

The **Biq Mac Library** is a public collection of Max-Cut problem instances (as well as related quadratic binary optimization instances) intended as a standard benchmark for algorithms [1]. These instances are of *medium size* (number of vertices *n* ranging roughly from 20 up to 500, with many around *n* = 100) and were assembled during the development of the Biq Mac solver [1] [2]. Each instance class in the library comes with known optimal cut values or best-known bounds, along with its number of vertices (*n*) and edge density (*d*) listed [3]. In other words, the library is organized by instance *types*, where each type has a name and fixed parameters, and usually **10 instances** per type (each with a specific name and known optimal Max-Cut value) [4].

Types of instances included: Chapter 3 of *biqmaclib.pdf* describes several categories of Max-Cut instances and how they were generated:

- **Random unweighted graphs:** e.g. *G0.5* class (denoted `g05` in instance names) – these are simple unweighted graphs where each possible edge is included with probability 1/2 (i.e. ~50% density). Instances were generated for sizes *n* = 60, 80, 100 [5]. Edges have no weights (effectively weight = 1 for each present edge). This provides baseline random Max-Cut problems of moderate density.

- **Random ±1-weighted graphs (sparse and dense):** e.g. *G^{−1/0/1}* classes – these are graphs with **integer weights in {-1, 0, 1}** on each edge. Two subtypes are given: a **sparse** version (~10% of edges present, noted as `pm1s`) and a **dense** version (~99% edges, nearly complete, noted as `pm1d`). Instances of these were generated for *n* = 80 and 100 vertices. In the sparse case most pairs of nodes have no edge (weight 0), whereas in the dense case almost every pair is connected by either a +1 or –1 weight edge.

- **Random weighted graphs with larger weight range:** e.g. *G^[-10,10]* – graphs with integer edge weights uniformly chosen from –10 up to 10 (including negatives and positives). These were generated at multiple densities (d = 0.1, 0.5, 0.9) for *n* = 100. There is also a variant *G^[0,10]* (sometimes called **"pw"** for positive weights) where edge weights are non-negative (0 to 10) with similar density settings. These instances introduce heavier weight magnitudes and both sparse and dense connectivity.

- **Ising model instances (graphs from physics applications):** Section 3.2 covers Max-Cut instances derived from Ising spin glass models. For example, the library includes **one-dimensional chain** instances (linear graphs) and **two-dimensional or three-dimensional grid (torus) graphs** with random weights [6]. Specifically, the library lists 2D toroidal grid graphs of sizes 10×10, 15×15, 20×20 and 3D toroidal grids 5×5×5 up to 7×7×7, with weights drawn from a Gaussian distribution (these were contributed by Frauke Liers) [7] [8]. Such instances have structured topologies (regular lattice connections) as opposed to the random Erdős-Rényi style

graphs above. They are relevant to physical applications of Max-Cut (finding ground states of spin glass models).

Each instance in the Biq Mac dataset is typically provided as a **graph with weighted undirected edges**. The library's documentation tables list, for each problem instance, its name, number of nodes (*n*), edge density (*d*), and the optimal cut value (or a bound if optimum is unknown) [3]. For Max-Cut, *density* is defined as the fraction of all $\binom{n}{2}$ possible edges that are present [2]. In summary, the Biq Mac benchmark spans a variety of graph types – from purely random graphs (unweighted or with small random weights) to more structured or heavy-weighted graphs – all in order to test algorithms under diverse conditions.

Comparison to the Project's Generated Data

Your project's data (from `generate_data.py`) differs from the Biq Mac instances in several key aspects:

- **Graph Size:** The Biq Mac instances are medium-sized graphs, often around 100 nodes (and up to 500 in some cases) [1]. In contrast, the project's generator is currently set to a very small graph size (e.g. `n = 6` in the provided script) and could be adjusted to other sizes for experiments. Even if scaled up, the project is likely focusing on relatively small-n cases for training (tens of nodes) due to the pointer network's complexity. By comparison, Biq Mac provides a challenging test bed at 60, 80, 100 nodes (and beyond) [5]. This means Biq Mac graphs can be much larger than the training graphs, which tests the model's ability to generalize to bigger problem sizes.

- **Edge Density and Structure:** The project's random graphs are generated with a **planted partition structure** – roughly half the nodes in each of two communities, with **dense intra-group connections** and **sparse inter-group connections** (using probabilities P1, P0 ~ 0.9–1.0 for within-group edges and P ~ 0.0–0.1 for cross-group edges) [9] [10]. This yields graphs that overall have a moderate density (~40–50% of possible edges, since each half is almost a clique but few edges between halves), and importantly, a known optimal cut (the planted split separates the two dense communities). In contrast, many Biq Mac benchmark instances are **uniform random graphs** without planted solutions – for example, G0.5 graphs have edges placed uniformly at random (no particular community structure) at 50% density [5]. Biq Mac also includes very **sparse** graphs (10% density) and **nearly complete** graphs (99% density), as well as intermediate densities like 50%. Moreover, the Ising grid instances have regular degree (each node connected to its lattice neighbors), which is a different topological structure from the project's random graphs. In summary, the project's training data is a special case of random graph (with a strong community structure and binary edges), whereas the Biq Mac set covers a broader range of random connectivity patterns and includes structured lattice graphs. The pointer network will face a more diverse set of edge patterns and densities in Biq Mac instances than it sees in the training set.

- **Graph Type (Weights and Direction):** The project graphs are **unweighted and undirected** – edges are either present (weight 1) or absent (0), and the adjacency matrix is symmetric [11]. All Biq Mac Max-Cut instances are also undirected graphs (Max-Cut is defined on undirected graphs), but **many are weighted**. Only the G0.5 class is strictly unweighted (edges present/absent) [5]. Other classes assign weights to edges, including negative weights (in the ±1 and ±10 ranges). This means a model trained only on unweighted (0/1) inputs will encounter new input distributions with Biq Mac data – edges now have integer values that can be greater than 1 or even negative. In Max-Cut, negative-weight edges effectively *want* to stay on the same side

(not be cut) to maximize the objective, which is a different dynamic than purely nonnegative edges. The pointer network architecture itself does not inherently assume binary weights – it can in principle take a weighted adjacency matrix as input – but the network would need to interpret these weight magnitudes appropriately. (If needed, one could normalize or scale the weights for the model, e.g. divide by a constant, but the raw values can be input directly as they are features of the graph.) All graphs remain undirected (symmetric matrices) in both cases, so there is no discrepancy in directed vs undirected – the main difference is the presence and distribution of weights.

- **Data Format:** There is a significant difference in how the data is stored and fed into the model. The project's generator outputs each graph instance as a **dense adjacency matrix** (flattened to a CSV row) followed by the solution bitmask [12]. For example, a 6-node graph becomes a 6×6 matrix (flattened to 36 entries) plus 6 solution entries, all on one line. In contrast, the Biq Mac instances are distributed in either **sparse text formats**. Typically, they use the *"Rudy" output format*, which lists only the edges with their weights. In Rudy format, a file begins with two numbers: *n* (number of nodes) and *m* (number of edges), followed by *m* lines each containing an edge: `node_i node_j weight` [13]. (Nodes are numbered 1 to *n* in these files [14]. Lines starting with `#` are comments.) There is also mention of a "sparse matrix format" in the documentation, which for Max-Cut is essentially the same idea – listing weighted edges – just phrased as a QUBO matrix input [15] [13]. In either case, the Biq Mac data is **not** a ready-made adjacency matrix, but rather a list of weighted edges. To use it in your workflow, you'll need to convert these lists into the full adjacency matrix form that your neural network expects.

Integrating Biq Mac Instances into the Workflow

Given the differences above, here's how you can incorporate the Biq Mac benchmark graphs into your deep learning process:

Parsing and Loading Biq Mac Graph Files

1. **Obtain the files:** First, download the desired Biq Mac instances (e.g. from the Biq Mac website). They may come as individual `.txt` files or a compressed archive. Make sure you know whether they are in Rudy format (edge list) or another format – most likely it will be the Rudy format for Max-Cut instances [13].

2. **Read the header:** Open a file and read the first line. It should contain two integers: the number of nodes *n* and the number of edges *m* [16]. For example, a line `100 2475` would mean 100 nodes and 2475 edges listed after.

3. **Initialize data structures:** Create an empty adjacency matrix of size *n* × *n*, initialized with 0s. You can use a NumPy array (`np.zeros((n,n))`) or a Python list of lists. If the graph is unweighted, you'll be filling in 0/1; if weighted, use a numeric type (float or int) for the matrix entries.

4. **Parse edges:** Loop over the next *m* lines of the file. Each line has the format `i j w` (where *i* and *j* are node indices and *w* is the edge weight) [13]. Convert *i* and *j* from 1-based to 0-based indexing if you use Python (since Biq Mac uses 1…*n*). Then set `adj[i,j] = w` and also `adj[j,i] = w` (because the graph is undirected). Ignore any self-loop entries (where *i* = *j*, though those should be none or will be ignored by solver) [14]. Also, if an instance had multiple

edges between the same nodes (unlikely in these datasets), you would sum their weights as per the Biq Mac convention [14].

5. **Verify structure:** After reading, you should have a symmetric adjacency matrix. The diagonal will remain 0 (Max-Cut doesn't use self-loops). You might want to verify that the number of nonzero entries in your matrix is twice *m* (since each edge contributes to two symmetric entries) for consistency.

At this point, you have loaded the graph into an adjacency matrix `W` (of shape *n*×*n*). If you need the *solution* (optimal cut) for training or evaluation, note that the Biq Mac files **do not include the optimal partition** – they only give the graph. The library documentation provides the optimal cut value, but not the assignment of nodes to partitions. To get a solution vector (the 0/1 labels per node for the optimal cut), you would need to run a Max-Cut solver on the graph or use the solution if it's known from literature. For integration into a supervised learning pipeline, you'd have to obtain these optimal partitions separately. (If your workflow is reinforcement-learning-based or you only need to evaluate objective values, you might not require the explicit solution bitmask – more on this below.)

Converting to the Project's Format

Once you have the adjacency matrix `W` for a Biq Mac instance, you can integrate it similar to your generated data: - **As model input:** Your pointer network model expects an *n*×*n* adjacency matrix as input (which in code is flattened or treated as a set of node features) [17] [18]. You can feed the `W` matrix directly. If your model was trained on 0/1 matrices, `W` will now contain integers (and possibly negatives). This is fine, but be aware of scaling – a large positive weight (say 10) will have a different effect on the model's node embeddings than a weight of 1. The network's learned weights might not immediately generalize to these new ranges. In practice, you might normalize the adjacency matrix (for example, divide all weights by 10 or by the max weight magnitude) so that the inputs are in a range the model is used to. This is an optional preprocessing step to consider if you find the model struggling with weighted inputs.

- **As training/evaluation data:** If you want to create a dataset file (CSV) like your generated ones, you would flatten the adjacency matrix and append the solution vector. For instance, for an *n*-node Biq Mac graph, flatten `W` into length *n*², and then concatenate the length-*n* binary partition vector (0/1 indicating the side of each node). This would yield one line with *n*² + *n* entries. However, as mentioned, obtaining the true solution vector for each Biq Mac instance may require solving the Max-Cut. If you don't have the partition, you cannot directly use supervised training on these instances. One workaround for evaluation is to use the model in a predictive mode and then compare the *cut value* it achieves to the known optimal value.

Training vs. Evaluation Use of Biq Mac Data

In deciding whether to use the Biq Mac instances for training, evaluation, or both, consider the following:

- **Limited quantity for training:** Each instance class in Biq Mac has only 10 instances (or in some cases a few more) available [4]. This is a very small sample size by deep learning standards. The article you provided (*maxcut_article.pdf*) explicitly notes that 10 samples per size were "not enough to train the pointer network (training the pointer network model requires at least 100 groups of data)" [19]. Using the Biq Mac library solely as a training set would likely lead to severe overfitting and poor generalization, because the model could memorize those few graphs.

Therefore, **Biq Mac is not well-suited as a primary training dataset** for a deep learning approach.

- **Value for evaluation and benchmarking:** The Biq Mac instances are best used as a **benchmark test set**. After training your pointer network on a larger synthetically generated dataset (for example, the "Zhou dataset" or your own generator which can produce hundreds or thousands of random graphs), you can evaluate the trained model on Biq Mac instances to measure performance on known challenging problems [4]. This is exactly what the authors of the article did: they trained on a different dataset and **used Biq Mac only for testing** the generalization ability of their pointer network [20]. You can do the same by feeding each Biq Mac instance into your model and then comparing the model's cut result to the known optimal cut value (the library provides the optimum value for each instance). This will tell you how close the network's solution is to optimal, often reported as a percentage of optimal (for example, 95% of optimum).

- **Possible use in validation or fine-tuning:** You could use a few Biq Mac instances as a **validation set** during training – e.g. to periodically check how the model is doing on a small set of realistic problems. They could also be used for **fine-tuning** a model that was pre-trained on synthetic data: for instance, you might do a few epochs of training on the Biq Mac instances to adapt the model to the weight distributions (especially if you plan for the model to handle weighted graphs). Caution is needed, though – with only 10 samples of a given type, fine-tuning could easily overfit. If you do this, consider techniques like data augmentation (perhaps adding noise or mirror-flipping the partition labels) or just keep the fine-tuning learning rate very low.

- **Pointer network compatibility:** The pointer network architecture in your project is designed to output a sequence of node indices representing one partition [21] [22]. This architecture does not inherently limit the input graph type, so it *can* be applied to weighted Max-Cut instances. However, note that if your model was only trained on unweighted graphs with a very specific structure, its learned policy might not immediately handle, say, a graph with many negative edges or a 2D grid structure. The performance goals in the article suggest that a pointer network trained via reinforcement learning on random graphs was able to achieve high accuracy on Biq Mac test instances (often above 80–90% of optimum for 60–100 node graphs) [23] [24]. To replicate such performance, **training on a diverse set of random graphs is key**, rather than training on only one narrow graph distribution. You might consider expanding your training data generator to include weighted edges or different random models so that the network sees a variety of scenarios.

Recommendation: Use the Biq Mac dataset primarily for **evaluation**. After training your pointer network (on data generated by `generate_data.py` or an expanded generator), test it on the Biq Mac instances to measure how well it generalizes. To do this, you don't necessarily need the solution vectors from Biq Mac; you can have the model produce a cut and then compute that cut's weight and compare to the known optimal weight [25]. If you find the model's performance lacking on these, it indicates a generalization gap – you might then consider augmenting your training process (through more varied data or fine-tuning as discussed). But as a rule, **the Biq Mac instances serve as a gold-standard benchmark** to check your model's solution quality against known optima, rather than as bulk training data. This will ensure your model is ultimately evaluated on the same terms as other approaches in the literature, and you can directly cite percentages of optimal cut value achieved on these benchmark graphs to gauge your performance.

Sources:

1. Wiegele, A. *Biq Mac Library – A collection of Max-Cut and quadratic 0-1 programming instances of medium size* [1] [3]. (Details the contents and format of the Biq Mac benchmark instances.)
2. Biq Mac Solver – *Input format description* [13] [14]. (Defines the Rudy format for graph files – how nodes and edges are listed.)
3. Project source code (`generate_data.py`) [26] [11] and PointerNet description [27]. (Shows how the project generates graphs and the expected input/output format for the pointer network.)
4. Zhou *et al.*, "Solving Max-Cut with Pointer Networks" (excerpt from *maxcut_article.pdf*) [4] [25]. (Explains the use of Biq Mac as a test set and the performance of a pointer network on these instances.)

---

[1] [2] [3] biqmaclib.pdf
file://file-SnVVzx99dVkKiqtiFizKEH

[4] [19] [20] [23] [24] [25] maxcut_article.pdf
file://file-8hhh6u8tFufwSR738yz9pt

[5] [6] [7] [8] biqmac.aau.at
https://biqmac.aau.at/biqmaclib.pdf

[9] [10] [11] [12] [26] generate_data.py
file://file-A2djU5fGUqD25TknKQXct4

[13] [14] [15] [16] Biq Mac Solver - BInary Quadratic and MAx Cut Solver
https://biqmac.aau.at/

[17] train_network2.py
file://file-54TiQ7nui688B5fqu9fMsD

[18] [21] [22] [27] PointerNet.py
file://file-72F41pfUQkG4YCEXQAiU4s