

Guide: Using the RNA Folding Tool for Your EA Assignment

This guide will help you use the provided RNA folding software ([ipknote](#)) for your Evolutionary Algorithm (EA) assignment. The main goal of this assignment is to **design and implement an effective EA**, not to become an expert in computational biology.

Therefore, you should treat the folding software as a simple "black box" or a calculator. You give it an RNA sequence, and it gives you back a score. Your EA's job is to find the best possible sequence that achieves the best score.

For Linux Users (Recommended)

You can use a pre-compiled, executable version of the software.

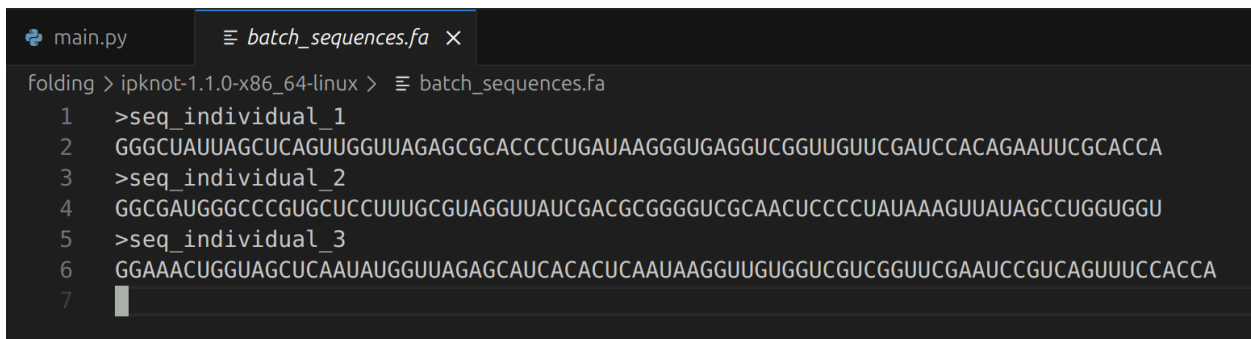
1. Download the [ipknot-1.1.0-x86_64-linux.zip](#) file from the [official releases page](#).
2. Unzip the file. You will find an executable file named [ipknot](#)

For Windows and macOS Users (Docker)

Compiling from source can be complex. The recommended method is to use Docker, which provides a ready-to-use environment. However, you are not restricted if you have experience with compiling from source code. You can find detailed instructions in IPKnot's [official Github page](#).

To Use IPKnot

Create a plain text file named [batch_sequences.fa](#). For each sequence in your population, add two lines in the FASTA format: a header line starting with `>` and the sequence itself on the next line.



```
main.py  batch_sequences.fa x
Folding > ipknot-1.1.0-x86_64-linux >  batch_sequences.fa
1  >seq_individual_1
2  GGGCUAUUAGCUCAGUUGGUUAGAGCGCACCCUGAUUAGGGUGAGGUCGGUUGUUCGAUCCACAGAAUUCGCACCA
3  >seq_individual_2
4  GGCGAUGGGCCCGUGCUCCUUUGCGUAGGUUAUCGACGCGGGGUCGCAACUCCCUAAUAAAGUUUAUAGCCUGGUGGU
5  >seq_individual_3
6  GGAAACUGGUAGCUCAAUAUGGUUAGAGCAUCACACUCAUAAGGUUGUGGUCGUCGGUUCGAUCCGUCAGUUUCCACCA
7  
```

To evaluate the sequences, directly call the sequence file name after IPKnot. Like the following image

```
(base) kaiyu@kaiyu-System-Product-Name: ~/Desktop/COEN433/Assignment_1/folding/ipknot-1.1.0-x86_64-linux
batch_sequences.fa ipknot README.md
(base) kaiyu@kaiyu-System-Product-Name: ~/Desktop/COEN433/Assignment_1/folding/ipknot-1.1.0-x86_64-linux$ ./ipknot batch_sequences.fa
>seq_individual_1
GGGCUAUUAGCUCAGUUGGUUAGAGCGCACCCCGUAUAGGGUGAGGCGGUUGUUCGAUCCACAGAAUUCGCCACCA
(((.....[.....]))((((.....))))((((.....)))).....
>seq_individual_2
GGCGAUGGGCGGCGUCUCUUUGCGUAGGUUAUUCGACGCGGGGUCGCAACUCCCCUAUAAAGUUAUAGCCUGUGGU
.....[.....][[[(.....)]][((.....)).....]].....
>seq_individual_3
GGAAACUGGUAGCUCAAUUGGUUAGGACUACACUCAUAAAGGUUGUGGUCGCGGUUCGAAUCCGUCAGUUUCCACCA
((((((((.....((([.....]))((((.....))))([.....]))[.....]))))))[.....]].....
(base) kaiyu@kaiyu-System-Product-Name: ~/Desktop/COEN433/Assignment_1/folding/ipknot-1.1.0-x86_64-linux$
```

To speed up the process, multiprocessing is necessary with the evaluation function. Since not all the students took a multiprocessing/parallel programming course, an example of multiprocessing code will be given below. However, you are also free to develop your own method, you do NOT have to follow the given code.

1. The Worker Function (`evaluate_chunk`)

This is a standalone function that performs the core logic for a *single small batch* of sequences. It's crucial that this is a top-level function, not a class method, so that the `multiprocessing` library can handle it correctly

What it does:

- Accepts a small list (a "chunk") of RNA sequences.
- Writes these sequences to a temporary FASTA file.
- Calls the external `ipknot` program on that file using `subprocess.run`.
- Parses the output from `ipknot`.
- Calculates the fitness for each sequence in its chunk.
- Returns a list of evaluated `Individual` objects.

```
# This is a TOP-LEVEL function, not part of a class.
def evaluate_chunk(args: Tuple) -> List[Individual]:
    # Unpack arguments: the sequences, ipknot path, and target structure
    sequences_chunk, ipknot_path, target_structure = args

    # This list will hold the results for just this chunk
    evaluated_individuals = []

    # --- Core Logic ---
```

```

# 1. Create a temporary file for the sequences in this chunk.
# 2. Run the external `ipknot` command on that file.
# 3. Parse the output to get the predicted structures.
# 4. For each sequence, calculate fitness and create an Individual
object.
#     evaluated_individuals.append(Individual(...))
# -----

# Return the list of evaluated individuals for this specific chunk
return evaluated_individuals

```

2. The Manager (**FitnessEvaluator**)

This class manages the whole parallel operation. Its main job is to split the data, create the pool of workers, and collect the results.

What it does:

- Determines how many CPU cores to use. A good rule of thumb is `os.cpu_count() - 1`.
- **Splits the population** into smaller chunks.
- **Prepares arguments** for each worker. Notice how we bundle the chunk of sequences, the `ipknot` path, and the target structure into a single tuple for each worker.
- Creates a `multiprocessing.Pool` with the desired number of worker processes.
- Uses `pool.map()` to apply the `evaluate_chunk` function to each of the chunks in parallel. This one line is where the magic happens! It automatically handles distributing the work and waiting for all workers to finish.
- **Combines the results** from all workers (which is a list of lists) into a single, flat list of evaluated individuals.

```

import multiprocessing
import os

class FitnessEvaluator:
    def __init__(self, ipknot_path: str, target_structure: str):
        self.ipknot_path = ipknot_path
        self.target_structure = target_structure

    def evaluate_population(self, population_sequences: List[str]) ->
List[Individual]:

```

```

    # Determine how many parallel worker processes to create
    num_workers = max(1, os.cpu_count() - 1)

    # 1. Split the big list of sequences into smaller chunks
    chunk_size = (len(population_sequences) + num_workers - 1) //
num_workers
    chunks = [population_sequences[i:i + chunk_size] for i in range(0,
len(population_sequences), chunk_size)]

    # 2. Prepare arguments for each worker's task
    args_for_workers = [(chunk, self.ipknot_path,
self.target_structure) for chunk in chunks]

    # 3. Create a pool of workers and distribute the tasks
    with multiprocessing.Pool(processes=num_workers) as pool:
        # pool.map applies `evaluate_chunk` to each item in
`args_for_workers`
        results_from_workers = pool.map(evaluate_chunk,
args_for_workers)

    # 4. Combine the lists returned by each worker into a single list
    evaluated_individuals = [individual for sublist in
results_from_workers for individual in sublist]

    return evaluated_individuals

```

Enjoy your EA.