Nash

Developer Documentation

Contents

1 About Nash

Nash is a simple command line tool which searches for stable strategy choices, so called *Nash equilibria*, in games of two players. These equilibria determine strategies for each player such that nobody has the incentive to use a different strategy instead. They may not, however, capture the best strategies overall.

The program is designed with efficiency in mind, such that the running time is plausible even in situations where players have many different actions to choose from.

2 Building

To build the tool, run `make` in the root of the repository, which creates the `nash` executable in the same place.

If you wish to build the program for debuging, use the `make debug` rule.

3 Structure

The project uses an object-oriented approach to divide the code into self-contained parts. The intent is to separate command interface, input parsing and computations. The `main` function uses their interfaces to connect them together.

3.1 Workflow

First of all, the `Params` class parses command line arguments. Based on them, either an input file or standard input is read. This is handled by `Reader`, which opens the stream and splits lines into individual payoff entries. The entries can then be loaded into matrix objects. The `Matrix` class also provides basic matrix operations and transformations.

The matrices are then be passed to the Lemke-Howson algorithm, which returns an instance of the `Equilibrium` struct. According to options from `Params`, the algorithm can be repeated multiple times. The result is then printed to standard output and the program terminates.

3.2 Arguments

Arguments are parsed by an instance of the `Params` class. It sets the options into its own member variables. For all unspecified options, the default values are returned.

Whenever an argument is handled, the following argument is passed along with it, in case it happens to be its value. This happens even for long flags, although the only long flag is `--help` without any values, this is done for consistency purpose.

An error can be caused by an unknown option, missing value or invalid value. All errors print the usage schema and suggest the help option.

Help can be requested with both the short (`-h`) or long (`--help`) flags, although all other options use short flags. Someone who is using the tool for the first time does not know which one to use, therefore both are supported.

3.3 Input

Input parsing is handled by an instance of the `Reader` class. If reading from a file, it opens this file and redirects the stream to standard input so that subsequent reading can be the same for both input alternatives.

`Reader` always reads an entire line from input, splits it with the separator given in command line arguments and stores the splits into a buffer.

It provides two interface functions to access the entries, `nextLineLength()` and then `next()` to get the next entry as a number.

3.4 Matrix

The `Matrix` class works as a container. The matrix is constructed stored by rows, since row operations are more common.

Matrices are used not only to store payoffs, but also for the search algorithm itself. The class therefore supports necessary operations, such as multiplying a row by a constant, adding two rows, transposing the matrix or normalizing it (i.e. ensuring all entries are positive).

3.5 Equilibrium

`Equilibrium` is a struct which contains the resulting strategy vectors.

Many functions are related to it, such as normalizing a vector (rescaling such that probabilities sum up to 1), rounding a vector, comparing equilibria and printing them.

When output is created, equilibria returned by repeated iterations of the search algorithm must be compared so that they do not appear multiple times. Unfortunately, identical equilibria can differ slightly due to the imprecision of floating point calculations. It is therefore necessary to round the strategy vectors.

3.6 Errors

Most errors cause an exception to be thrown. The error message is stated in the exception's `what` argument. When the exception is caught, the message is printed to error output and the program terminates.

If a non-fatal error occurs, it is printed to error output directly and the program continues uninterrupted.

4 Algorithms

As of now, only one algorithm has been implemented.

4.1 Lemke-Howson

The Lemke-Howson algorithm is the best known algorithm for finding a Nash equilibrium in a bimatrix game. Its structure similar to the simplex method known from linear programming.

While the runtime of the algorithm is generally fast, it can be exponential in the number of actions in the worst case. It has been shown that the problem of finding a Nash equilibrium is PPAD-complete. Consequently, no polynomial-time algorithm exists (assuming FP $\neq$ PPAD).

The algorithm tries to find the probabilities $x_1, \ldots, x_m, y_{m+1}, \ldots, y_{m+n}$. It introduces slack variables $r_1, \ldots, r_m, s_{m+1}, \ldots, s_{m+n}$ to form $m+n$ equations with entries from payoff matrices that have to be satisfied.

The result of this are two sets of equations. Each equation has an isolated variable on one of its sides, these variables form the *basis*. The algorithm uses complementary pivoting, always isolating one of the variables, moving it to the basis, thus taking one variable out of the basis.

The variable which enters the basis is determined by a *label*. At first, the basis contains every label exactly once. One of the labels is chosen to enter the basis, causing one duplicate and one missing label. Complementary pivoting then exchanges the labels until all labels appear again, which marks the end of the algorithm.

The solution is determined by the variables in the basis and contant values in their corresponding equations. The strategy vectors are constructed, normalized and returned as an equilibrium.

The implementation makes heavy use of the `Matrix` class to represent the equations, since multiplying and adding rows are common operations. Each row represents one equation. Variables are sorted into columns based on their label.

There are $m + n$ total labels. Each of them can be picked as the first label to enter the basis, giving $m + n$ possible starting conditions, out of which each may lead to a new equilibrium. The algorithm is repeated until the desired number of equilibria is reached, or until all labels have been tried.

A more detailed description of the algorithm can be found in the resource linked below.

5 Links

- Nash: `https://github.com/kulisak12/Nash`
- Game theory, Lemke-Howson:
  `https://kam.mff.cuni.cz/~balko/ath2021/main.pdf`