open_spiel Documentation

The open_spiel authors

Apr 22, 2024

GETTING STARTED

CHAPTER

ONE

WHAT IS OPENSPIEL?

OpenSpiel is a collection of environments and algorithms for research in general reinforcement learning and search/planning in games. OpenSpiel also includes tools to analyze learning dynamics and other common evaluation metrics. Games are represented as procedural extensive-form games, with some natural extensions.

Open Spiel supports

- Single and multi-player games
- Fully observable (via observations) and imperfect information games (via information states and observations)
- Stochasticity (via explicit chance nodes mostly, even though implicit stochasticity is partially supported)
- n-player normal-form "one-shot" games and (2-player) matrix games
- Sequential and simultaneous move games
- Zero-sum, general-sum, and cooperative (identical payoff) games

Multi-language support

- C++17
- Python 3

The games and utility functions (e.g. exploitability computation) are written in C++. These are also available using pybind11 Python bindings.

The methods names are in `CamelCase` in C++ and `snake_case` in Python (e.g. `state.ApplyAction` in C++ will be `state.apply_action` in Python). See the pybind11 definition in open_spiel/python/pybind11/pyspiel.cc for the full mapping between names.

For algorithms, many are written in both languages, even if some are only available from Python.

Platforms

OpenSpiel has been tested on Linux (Ubuntu and Debian), MacOS. There is limited support for on Windows 10.

Visualization of games

There is a basic visualizer based on graphviz, see open_spiel/python/examples/treeviz_example.py.

There is an interactive viewer for OpenSpiel games called SpielViz.

CHAPTER

TWO

INSTALLATION

2.1 Python-only installation via pip

If you plan to only use the Python API, then the easiest way to install OpenSpiel is to use pip. On MacOS or Linux, simply run:

```
python3 -m pip install open_spiel
```

The binary distribution is new as of OpenSpiel 1.0.0, and is only supported on x86_64 architectures. If you encounter any problems, you can still install OpenSpiel via pip from source (see below), but please open an issue to let us know about the problem.

2.1.1 Python-only installation via pip (from source).

If the binary distribution is not an option, you can also build OpenSpiel via pip from source. CMake, Clang and Python 3 development files are required to build the Python extension. Note that we recommend Clang but g++ >= 9.2 should also work.

E.g. on Ubuntu or Debian:

```
# Check to see if you have the necessary tools for building OpenSpiel:
cmake --version        # Must be >= 3.17
clang++ --version      # Must be >= 7.0.0
python3-config --help

# If not, run this line to install them.
# On older Linux distros, the package might be called clang-9 or clang-10
sudo apt-get install cmake clang python3-dev

# On older Linux distros, the versions may be too old.
# E.g. on Ubuntu 18.04, there are a few extra steps:
# sudo apt-get install clang-10
# pip3 install cmake  # You might need to relogin to get the new CMake version
# export CXX=clang++-10

# Recommended: Install pip dependencies and run under virtualenv.
sudo apt-get install virtualenv python3-virtualenv
virtualenv -p python3 venv
source venv/bin/activate
```

(continues on next page)

(continued from previous page)

```
# Finally, install OpenSpiel and its dependencies:
python3 -m pip install --upgrade setuptools pip
python3 -m pip install --no-binary=:open_spiel: open_spiel

# To exit the virtual env
deactivate

## **IMPORTANT NOTE**. If the build fails, please first make sure you have the
## required versions of the tools above and that you followed the recommended
## option. Then, open an issue: https://github.com/deepmind/open_spiel/issues
```

Note that the build could take several minutes.

On MacOS, you can install the dependencies via `brew install cmake python3`. For clang, you need to install or upgrade XCode and install the command-line developer tools.

2.2 Installation from Source

The instructions here are for Linux and MacOS. For installation on Windows, see these separate installation instructions. On Linux, we recommend Ubuntu 22.04, Debian 10, or later versions. On MacOS, we recommend XCode 11 or newer. For the Python API: our tests run using Python versions 3.7 - 3.10. If you encounter any problems on other setups, please let us know by opening an issue.

Currently there are three installation methods:

1. building from the source code and editing `PYTHONPATH`.
2. using `pip install`.
3. installing via Docker.

2.3 Summary

In a nutshell:

```
./install.sh  # Needed to run once and when major changes are released.
./open_spiel/scripts/build_and_run_tests.sh # Run this every-time you need to rebuild.
```

1. (Optional) Configure *Conditional Dependencies*.
2. Install system packages (e.g. cmake) and download some dependencies. Only needs to be run once or if you enable some new conditional dependencies.

   ```
   ./install.sh
   ```

3. Install your *Python dependencies*, e.g. in Python 3 using `virtualenv`:

   ```
   virtualenv -p python3 venv
   source venv/bin/activate
   ```

   Use `deactivate` to quit the virtual environment.

   `pip` should be installed once and upgraded:

```
curl https://bootstrap.pypa.io/get-pip.py -o get-pip.py
# Install pip deps as your user. Do not use the system's pip.
python3 get-pip.py
pip3 install --upgrade pip
pip3 install --upgrade setuptools testresources
```

Additionally, if you intend to use one of the optional Python dependencies (see open_spiel/scripts/install.sh), you must manually install and/or upgrade them, e.g.: `bash pip install --upgrade torch==x.xx.x jax==x.x.x` where `x.xx.x` should be the desired version numbers (which can be found at the link above).

4. This sections differs depending on the installation procedure:

   Building and testing from source

   ```
   python3 -m pip install -r requirements.txt
   ./open_spiel/scripts/build_and_run_tests.sh
   ```

   Building and testing using PIP

   ```
   python3 -m pip install .
   ```

   Optionally, use `pip install -e` to install in editable mode, which will allow you to skip this `pip install` step if you edit any Python source files. If you edit any C++ files, you will have to rerun the install command.

5. Only when building from source:

   ```
   # For the python modules in open_spiel.
   export PYTHONPATH=$PYTHONPATH:/<path_to_open_spiel>
   # For the Python bindings of Pyspiel
   export PYTHONPATH=$PYTHONPATH:/<path_to_open_spiel>/build/python
   ```

   to `./venv/bin/activate` or your `~/.bashrc` to be able to import OpenSpiel from anywhere.

To make sure OpenSpiel works on the default configurations, we do use the `python3` command and not `python` (which still defaults to Python 2 on modern Linux versions).

2.4 Installing via Docker

Please note that we don't regularly test the Docker installation. As such, it may not work at any given time. If you encounter a problem, please open an issue.

Option 1 (Basic, 3.13GB):

```
docker build --target base -t openspiel -f Dockerfile.base .
```

Option 2 (Slim, 2.26GB):

```
docker build --target python-slim -t openspiel -f Dockerfile.base .
```

If you are only interested in developing in Python, use the second image. You can navigate through the runtime of the container (after the build step) with:

```
docker run -it --entrypoint /bin/bash openspiel
```

Finally you can run examples using:

```
docker run openspiel python3 python/examples/matrix_game_example.py
docker run openspiel python3 python/examples/example.py
```

Option 3 (Jupyter Notebook):

Installs OpenSpiel with an additional Jupyter Notebook environment.

```
docker build -t openspiel-notebook -f Dockerfile.jupyter --rm .
docker run -it --rm -p 8888:8888 openspiel-notebook
```

More info: https://jupyter-docker-stacks.readthedocs.io/en/latest/

2.5 Running the first examples

In the `build` directory, running `examples/example` will prints out a list of registered games and the usage. Now, let's play game of Tic-Tac-Toe with uniform random players:

```
examples/example --game=tic_tac_toe
```

Once the proper Python paths are set, from the main directory (one above `build`), try these out:

```
# Similar to the C++ example:
python3 open_spiel/python/examples/example.py --game_string=breakthrough

# Play a game against a random or MCTS bot:
python3 open_spiel/python/examples/mcts.py --game=tic_tac_toe --player1=human --
↪player2=random
python3 open_spiel/python/examples/mcts.py --game=tic_tac_toe --player1=human --
↪player2=mcts
```

2.6 Detailed steps

2.6.1 Configuring conditional dependencies

Conditional dependencies are configured using environment variables, e.g.

```
export OPEN_SPIEL_BUILD_WITH_HANABI=ON
```

`install.sh` may need to be rerun after enabling new conditional dependencies.

See open_spiel/scripts/global_variables.sh for the full list of conditional dependencies.

See also the Developer Guide.

2.6.2 Installing system-wide dependencies

See open_spiel/scripts/install.sh for the required packages and cloned repositories.

2.6.3 Installing Python dependencies

Using a `virtualenv` to install python dependencies is highly recommended. For more information see: https://packaging.python.org/guides/installing-using-pip-and-virtual-environments/

Required dependencies

Install required dependencies (Python 3):

```
# Ubuntu 22.04 and newer:
python3 -m venv ./venv
source venv/bin/activate
python3 -m pip install -r requirements.txt
# Older than Ubuntu 22.04:
virtualenv -p python3 venv
source venv/bin/activate
python3 -m pip install -r requirements.txt
```

Alternatively, although not recommended, you can install the Python dependencies system-wide with:

```
python3 -m pip install --upgrade -r requirements.txt
```

Optional dependencies

Additionally, if you intend to use one of the optional Python dependencies (see open_spiel/scripts/install.sh), you must manually install and/or upgrade them. The installation scripts will not install or upgrade these dependencies. e.g.:

```
python3 -m pip install --upgrade torch==x.xx.x jax==x.x.x
```

where `x.xx.x` should be the desired version numbers (which can be found at the link above).

2.6.4 Building and running tests

Make sure that the virtual environment is still activated.

By default, Clang C++ compiler is used (and potentially installed by open_spiel/scripts/install.sh).

Build and run tests (Python 3):

```
mkdir build
cd build
CXX=clang++ cmake -DPython3_EXECUTABLE=$(which python3) -DCMAKE_CXX_COMPILER=${CXX} ../
↪open_spiel
make -j$(nproc)
ctest -j$(nproc)
```

The CMake variable `Python3_EXECUTABLE` is used to specify the Python interpreter. If the variable is not set, CMake's FindPython3 module will prefer the latest version installed. Note, Python >= 3.7 is required.

One can run an example of a game running (in the `build/` folder):

```
./examples/example --game=tic_tac_toe
```

2.6.5 Setting Your PYTHONPATH environment variable

To be able to import the Python code (both the C++ binding `pyspiel` and the rest) from any location, you will need to add to your PYTHONPATH the root directory and the `open_spiel` directory.

When using a virtualenv, the following should be added to `<virtualenv>/bin/activate`. For a system-wide install, ddd it in your `.bashrc` or `.profile`.

```
# For the python modules in open_spiel.
export PYTHONPATH=$PYTHONPATH:/<path_to_open_spiel>
# For the Python bindings of Pyspiel
export PYTHONPATH=$PYTHONPATH:/<path_to_open_spiel>/build/python
```

CHAPTER

THREE

FIRST EXAMPLES

One can run an example of a game running (in the `build/` folder):

```
./examples/example --game=tic_tac_toe
```

Similar examples using the Python API (run from one above `build`):

```
# Similar to the C++ example:
python3 open_spiel/python/examples/example.py --game_string=breakthrough

# Play a game against a random or MCTS bot:
python3 open_spiel/python/examples/mcts.py --game=tic_tac_toe --player1=human --
↪player2=random
python3 open_spiel/python/examples/mcts.py --game=tic_tac_toe --player1=human --
↪player2=mcts
```

CHAPTER

FOUR

CONCEPTS

The following documentation describes the high-level concepts. Refer to the code comments for specific API descriptions.

Note that, in English, the word "game" is used for both the description of the rules (e.g. the game of chess) and for a specific instance of a playthrough (e.g. "we played a game of chess yesterday"). We will be using "playthrough" or "trajectory" to refer to the second concept.

The methods names are in `CamelCase` in C++ and `snake_case` in Python without any other difference (e.g. `state.ApplyAction` in C++ will be `state.apply_action` in Python).

4.1 The tree representation

There are mainly 2 concepts to know about (defined in open_spiel/spiel.h):

- A `Game` object contains the high level description for a game (e.g. whether it is simultaneous or sequential, the number of players, the maximum and minimum scores).
- A `State`, which describe a specifics point (e.g. a specific board position in chess, a specific set of player cards, public cards and past bets in Poker) within a trajectory.

All possible trajectories in a game are represented as a tree. In this tree, a node is a `State` and is associated to a specific history of moves for all players. Transitions are actions taken by players (in case of a simultaneous node, the transition is composed of the actions for all players).

Note that in most games, we deal with chance (i.e. any source of randomness) using a an explicit player (the "chance" player, which has id `kChancePlayerId`). For example, in Poker, the root state would just be the players without any cards, and the first transitions will be chance nodes to deal the cards to the players (in practice once card is dealt per transition).

See `spiel.h` for the full API description. For example, `game.NewInitialState()` will return the root `State`. Then, `state.LegalActions()` can be used to get the possible legal actions and `state.ApplyAction(action)` can be used to update `state` in place to play the given `action` (use `state.Child(action)` to create a new state and apply the action to it).

CHAPTER

FIVE

LOADING A GAME

The games are all implemented in C++ in open_spiel/games. Available games names can be listed using `RegisteredNames()`.

A game can be created from its name and its arguments (which usually have defaults). There are 2 ways to create a game:

- Using the game name and a structured `GameParameters` object (which, in Python, is a dictionary from argument name to compatible types (int, bool, str or a further dict). e.g. `{"players": 3}` with `LoadGame`.
- Using a string representation such as `kuhn_poker(players=3)`, giving `LoadGame(kuhn_poker(players=3))`. See `open_spiel/game_parameters.cc` for the exact syntax.

5.1 Creating sequential games from simultaneous games

It is possible to apply generic game transformations (see open_spiel/game_transforms/) such as loading an `n`-players simultaneous games into an equivalent turn-based game where simultaneous moves are encoded as `n` turns.

One can use `LoadGameAsTurnBased(game)`, or use the string representation, such as `turn_based_simultaneous_game(game=goofspiel(imp_info=True,num_cards=4,points_order=descending))`.

CHAPTER

SIX

PLAYING A TRAJECTORY

Here are for example the Python code to play one trajectory:

```
import random
import pyspiel
import numpy as np

game = pyspiel.load_game("kuhn_poker")
state = game.new_initial_state()
while not state.is_terminal():
  legal_actions = state.legal_actions()
  if state.is_chance_node():
    # Sample a chance event outcome.
    outcomes_with_probs = state.chance_outcomes()
    action_list, prob_list = zip(*outcomes_with_probs)
    action = np.random.choice(action_list, p=prob_list)
    state.apply_action(action)
  else:
    # The algorithm can pick an action based on an observation (fully observable
    # games) or an information state (information available for that player)
    # We arbitrarily select the first available action as an example.
    action = legal_actions[0]
    state.apply_action(action)
```

See open_spiel/python/examples/example.py for a more thorough example that covers more use of the core API.

See open_spiel/python/examples/playthrough.py (and open_spiel/python/algorithms/generate_playthrough.py) for an richer example generating a playthrough and printing all available information.

In C++, see open_spiel/examples/example.cc which generates random trajectories.

CHAPTER

SEVEN

OPENSPIEL CORE API REFERENCE

OpenSpiel consists of several core functions and classes. This page acts as a helpful reminder of how to use the main functionality of OpenSpiel.

Most of the functions are described and illustrated via Python syntax and examples, and there are pointers to the corresponding C++ functions.

Disclaimer: This is meant as a guide to facilitate OpenSpiel development in Python. However, spiel.h remains the single source of truth for documentation on the core API.

7.1 Core Functions

7.2 State methods

7.3 Game methods

CHAPTER

EIGHT

AVAILABLE ALGORITHMS

•: thoroughly-tested. In many cases, we verified against known values and/or reproduced results from papers.

~: implemented but lightly tested.

X: known problems; please see github issues.

CHAPTER

NINE

AVAILABLE GAMES

: thoroughly-tested. In many cases, we verified against known values and/or reproduced results from papers.

: implemented but lightly tested.

: known issues (see notes below and code for details).

9.1 Details

9.1.1 2048

- A single player game where player aims to create a 2048 tile by merging other tiles.
- Numbers on a grid.
- Modern game.
- Non-deterministic.
- Perfect information.
- 1 player.
- Github

9.1.2 Amazons

- Move pieces on a board trying to block opponents from moving.
- Pieces on a grid.
- Modern game.
- Deterministic.
- Perfect information.
- 2 players.
- Wikipedia

9.1.3 Atari

- Agent plays classic games from Gym's Atari Environments, such as Breakout.
- Single player.
- Most games are non-deterministic.
- Perfect information.

9.1.4 Backgammon

- Players move their pieces through the board based on the rolls of dice.
- Idiosyncratic format.
- Traditional game.
- Non-deterministic.
- Perfect information.
- 2 players.
- Wikipedia

9.1.5 Bargaining

- Agents negotiate for items in a pool with different (hidden) valuations.
- Research game.
- Non-deterministic (randomized pool and valuations).
- Imperfect information.
- 2 players.
- Lewis et al. '17, DeVault et al. '15

9.1.6 Battleship

- Players place ships and shoot at each other in turns.
- Pieces on a board.
- Traditional game.
- Deterministic.
- Imperfect information.
- 2 players.
- Good for correlated equilibria.
- Farina et al. '19, Correlation in Extensive-Form Games: Saddle-Point Formulation and Benchmarks. Based on the original game (wikipedia)

9.1.7 Blackjack

- Simplified version of blackjack, with only HIT/STAND moves.
- Traditional game.
- Non-deterministic.
- Imperfect information.
- 1 player.
- Wikipedia

9.1.8 Block Dominoes

- Most simple version of dominoes.
- Consists of 28 tiles, featuring all combinations of spot counts (also called pips or dots) between zero and six.
- Traditional game.
- Non-deterministic.
- Imperfect information.
- 2 players.
- *Wikipedia*

9.1.9 Breakthrough

- Simplified chess using only pawns.
- Pieces on a grid.
- Modern game.
- Deterministic.
- Perfect information.
- 2 players.
- Wikipedia

9.1.10 Bridge

- A card game where players compete in pairs.
- Card game.
- Traditional game.
- Non-deterministic.
- Imperfect information.
- 4 players.
- Wikipedia

9.1.11 (Uncontested) Bridge bidding

- Players score points by forming specific sets with the cards in their hands.
- Card game.
- Research game.
- Non-deterministic.
- Imperfect information.
- 2 players.
- Wikipedia

9.1.12 Catch

- Agent must move horizontally to 'catch' a descending ball. Designed to test basic learning.
- Agent on a grid.
- Research game.
- Non-deterministic.
- Perfect information.
- 1 players.
- Mnih et al. 2014, Recurrent Models of Visual Attention,Osband et al '19, Behaviour Suite for Reinforcement Learning, Appendix A

9.1.13 Checkers

- Players move pieces around the board with the goal of eliminating the opposing pieces.
- Pieces on a grid.
- Traditional game.
- Deterministic.
- Perfect information.
- 2 players.
- Wikipedia

9.1.14 Cliff Walking

- Agent must find goal without falling off a cliff. Designed to demonstrate exploration-with-danger.
- Agent on a grid.
- Research game.
- Deterministic.
- Perfect information.
- 1 players.

- Sutton et al. '18, page 132

9.1.15 Clobber

- Simplified checkers, where tokens can capture neighbouring tokens. Designed to be amenable to combinatorial analysis.
- Pieces on a grid.
- Research game.
- Deterministic.
- Perfect information.
- 2 players.
- Wikipedia

9.1.16 Coin Game

- Agents must collect their and their collaborator's tokens while avoiding a third kind of token. Designed to test divining of collaborator's intentions
- Agents on a grid.
- Research game.
- Non-deterministic.
- Imperfect information (all players see the grid and their own preferences, but not the preferences of other players).
- 2 players.
- Raileanu et al. '18, Modeling Others using Oneself in Multi-Agent Reinforcement Learning

9.1.17 Colored Trails

- Agents negotiations for chips that they they play on a colored grid to move closer to the goal.
- Agents on a grid.
- Research game.
- Non-deterministic (randomized board & chip configuration).
- Imperfect information.
- 3 players.
- Ya'akov et al. '10, Fecici & Pfeffer '08, de Jong et al. '11

9.1.18 Connect Four

- Players drop tokens into columns to try and form a pattern.
- Tokens on a grid.
- Traditional game.
- Deterministic.
- Perfect information.
- 2 players.
- Wikipedia

9.1.19 Cooperative Box-Pushing

- Agents must collaborate to push a box into the goal. Designed to test collaboration.
- Agents on a grid.
- Research game.
- Deterministic.
- Perfect information.
- 2 players.
- Seuken & Zilberstein '12, Improved Memory-Bounded Dynamic Programming for Decentralized POMDPs

9.1.20 Chess

- Players move pieces around the board with the goal of eliminating the opposing pieces.
- Pieces on a grid.
- Traditional game.
- Deterministic.
- Perfect information.
- 2 players.
- Wikipedia

9.1.21 Dots and Boxes

- Players put lines between dots to form boxes to get points.
- Traditional game.
- Deterministic.
- Perfect information.
- 2 players.
- Wikipedia

9.1.22 Crazy Eights

- A precursor of UNO (see here).
- Players try to match the rank or suit of the previous played card.
- Eights are viewed as wild cards.
- In an alternative version, special cards such as skip, reverse, draw-two are permitted.
- Nondeterministic.
- Imperfect information.
- =2 players.
- Wikipedia

9.1.23 Dark Hex

- Hex, except the opponent's tokens are hidden. (Imperfect-information version)
- Uses tokens on a hex grid.
- Research game.
- Deterministic.
- Imperfect information.
- 2 players.

9.1.24 Deep Sea

- Agent must explore to find reward (first version) or penalty (second version). Designed to test exploration.
- Agent on a grid.
- Research game.
- Deterministic.
- Perfect information.
- 1 players.
- Osband et al. '17, Deep Exploration via Randomized Value Functions

9.1.25 Dou Dizhu

- A three-player games where one player (dizhu) plays against a team of two (peasants).
- Uses a 54-card deck.
- Non-deterministic.
- Imperfect information.
- Three players.
- Wikipedia

9.1.26 Euchre

- Trick-taking card game where players compete in pairs.
- Card game.
- Traditional game.
- Non-deterministic.
- Imperfect information.
- 4 players.
- Wikipedia

9.1.27 First-price Sealed-Bid Auction

- Agents submit bids simultaneously; highest bid wins, and that's the price paid.
- Idiosyncratic format.
- Research game.
- Non-deterministic.
- Imperfect, incomplete information.
- 2-10 players.
- Wikipedia

9.1.28 Gin Rummy

- Players score points by forming specific sets with the cards in their hands.
- Card game.
- Traditional game.
- Non-deterministic.
- Imperfect information.
- 2 players.
- Wikipedia

9.1.29 Go

- Players place tokens on the board with the goal of encircling territory.
- Tokens on a grid.
- Traditional game.
- Deterministic.
- Perfect information.
- 2 players.
- Wikipedia

9.1.30 Goofspiel

- Players bid with their cards to win other cards.
- Card game.
- Traditional game.
- Non-deterministic.
- Imperfect information.
- 2-10 players.
- Wikipedia

9.1.31 Hanabi

- Players can see only other player's pieces, and everyone must cooperate to win.
- Idiosyncratic format.
- Modern game.
- Non-deterministic.
- Imperfect information.
- 2-5 players.
- Wikipedia and Bard et al. '19, The Hanabi Challenge: A New Frontier for AI Research
- Implemented via Hanabi Learning Environment

9.1.32 Havannah

- Players add tokens to a hex grid to try and form a winning structure.
- Tokens on a hex grid.
- Modern game.
- Deterministic.
- Perfect information.
- 2 players.
- Wikipedia

9.1.33 Hearts

- A card game where players try to avoid playing the highest card in each round.
- Card game.
- Traditional game.
- Non-deterministic.
- Imperfect information.
- 3-6 players.

- Wikipedia

9.1.34 Hex

- Players add tokens to a hex grid to try and link opposite sides of the board.
- Uses tokens on a hex grid.
- Modern game.
- Deterministic.
- Perfect information.
- 2 players.
- Wikipedia
- Hex, the full story by Ryan Hayward and Bjarne Toft

9.1.35 Kriegspiel

- Chess with opponent's pieces unknown. Illegal moves have no effect - it remains the same player's turn until they make a legal move.
- Traditional chess variant, invented by Henry Michael Temple in 1899.
- Deterministic.
- Imperfect information.
- 2 players.
- Wikipedia
- Monte Carlo tree search in Kriegspiel
- Game-Tree Search with Combinatorially Large Belief States, Parker 2005

9.1.36 Kuhn poker

- Simplified poker amenable to game-theoretic analysis.
- Cards with bidding.
- Research game.
- Non-deterministic.
- Imperfect information.
- 2 players.
- Wikipedia

9.1.37 Laser Tag

- Agents see a local part of the grid, and attempt to tag each other with beams.
- Agents on a grid.
- Research game.
- Non-deterministic.
- Imperfect information.
- 2 players.
- Leibo et al. '17, Lanctot et al. '17

9.1.38 Leduc poker

- Simplified poker amenable to game-theoretic analysis.
- Cards with bidding.
- Research game.
- Non-deterministic.
- Imperfect information.
- 2 players.
- Southey et al. '05, Bayes' bluff: Opponent modelling in poker

9.1.39 Lewis Signaling

- Receiver must choose an action dependent on the sender's hidden state. Designed to demonstrate the use of conventions.
- Idiosyncratic format.
- Research game.
- Non-deterministic.
- Imperfect information.
- 2 players.
- Wikipedia

9.1.40 Liar's Dice

- Players bid and bluff on the state of all the dice together, given only the state of their dice.
- Dice with bidding.
- Traditional game.
- Non-deterministic.
- Imperfect information.
- 2 players.

- Wikipedia

9.1.41 Liar's Poker

- Players bid and bluff on the state of all hands, given only the state of their hand.
- Cards with bidding.
- Traditional game.
- Non-deterministic.
- Imperfect information
- 2 or more players.
- Wikipedia

9.1.42 Mensch Aergere Dich Nicht

- Players roll dice to move their pegs toward their home row while throwing other players' pegs to the out area.
- Traditional game.
- Non-deterministic.
- Perfect information.
- 2-4 players.
- Wikipedia

9.1.43 Mancala

- Players take turns sowing beans on the board and try to capture more beans than the opponent.
- Idiosyncratic format.
- Traditional game.
- Deterministic.
- Perfect information.
- 2 players.
- Wikipedia

9.1.44 Markov Soccer

- Agents must take the ball to their goal, and can 'tackle' the opponent by predicting their next move.
- Agents on a grid.
- Research game.
- Non-deterministic.
- Imperfect information.
- 2 players.

- Littman '94, Markov games as a framework for multi-agent reinforcement learning,He et al. '16, Opponent Modeling in Deep Reinforcement Learning

9.1.45 Matching Pennies (Three-player)

- Players must predict and match/oppose another player. Designed to have an unstable Nash equilibrium.
- Idiosyncratic format.
- Research game.
- Deterministic.
- Imperfect information.
- 3 players.
- "Three problems in learning mixed-strategy Nash equilibria"

9.1.46 Mean Field Game : routing

- Representative player chooses at each node where they go. They has an origin, a destination and a departure time and chooses their route to minimize their travel time. Time spent on each link is a function of the distribution of players on the link when the player reaches the link.
- Network with choice of route.
- Research game.
- Mean-field (with a unique player).
- Explicit stochastic game (only for initial node).
- Perfect information.
- Cabannes et. al. '21, Solving N-player dynamic routing games with congestion: a mean field approach.

9.1.47 Mean Field Game : Linear-Quadratic

- Players are uniformly distributed and are then incentivized to gather at the same point (The lower the distanbce wrt. the distribution mean position, the higher the reward). A mean-reverting term pushes the players towards the distribution, a gaussian noise term perturbs them. The players' actions alter their states linearly (alpha * a * dt) and the cost thereof is quadratic (K * a^2 * dt), hence the name. There exists an exact, closed form solution for the fully continuous version of this game.
- Research game.
- Mean-field (with a unique player).
- Explicit stochastic game (only for initial node).
- Perfect information.
- [Perrin & al. 2019 (https://arxiv.org/abs/2007.03458)]

9.1.48 Morpion Solitaire (4D)

- A single player game where player aims to maximize lines drawn on a grid, under certain limitations.
- Uses tokens on a grid.
- Traditional game.
- Deterministic
- Perfect information.
- 1 player.
- Wikipedia

9.1.49 Negotiation

- Agents with different utilities must negotiate an allocation of resources.
- Idiosyncratic format.
- Research game.
- Non-deterministic.
- Imperfect information.
- 2 players.
- Lewis et al. '17, Cao et al. '18

9.1.50 Nim

- Two agents take objects from distinct piles trying to either avoid taking the last one or take it. Any positive number of objects can be taken on each turn given they all come from the same pile.
- Traditional mathematical game.
- Deterministic.
- Perfect information.
- 2 players.
- Wikipedia

9.1.51 Nine men's morris

- Two players put and move stones on the board to try to form mills (three adjacent stones in a line) to capture the other player's stones.
- Traditional game.
- Deterministic.
- Perfect information.
- 2 players.
- Wikipedia

9.1.52 Oh Hell

- A card game where players try to win exactly a declared number of tricks.
- Card game.
- Traditional game.
- Non-deterministic.
- Imperfect information.
- 3-7 players.
- Wikipedia

9.1.53 Oshi-Zumo

- Players must repeatedly bid to push a token off the other side of the board.
- Idiosyncratic format.
- Traditional game.
- Deterministic.
- Imperfect information.
- 2 players.
- Buro, 2004. Solving the oshi-zumo game Bosansky et al. '16, Algorithms for Computing Strategies in Two-Player Simultaneous Move Games

9.1.54 Oware

- Players redistribute tokens from their half of the board to capture tokens in the opponent's part of the board.
- Idiosyncratic format.
- Traditional game.
- Deterministic.
- Perfect information.
- 2 players.
- Wikipedia

9.1.55 Pathfinding

- Agents must move to their desitnation.
- Agents on a grid. Single-agent game is the classic examples from Sutton & Barto.
- Research game.
- Non-deterministic (in multiagent, collisions resolved by chance nodes).
- Perfect information.
- 1-10 players.

- Similar games appeared in Austerweil et al. '15, Greenwald & Hall '03, and Littman '01.

9.1.56 Pentago

- Players place tokens on the board, then rotate part of the board to a new orientation.
- Uses tokens on a grid.
- Modern game.
- Deterministic.
- Perfect information.
- 2 players.
- Wikipedia

9.1.57 Phantom Go

- Go, except the opponent's stones are hidden. The analogue of Kriegspiel for Go.
- Research game.
- Deterministic.
- Imperfect information.
- 2 players.
- Cazenave '05, A Phantom Go Program

9.1.58 Phantom Tic-Tac-Toe

- Tic-tac-toe, except the opponent's tokens are hidden. Designed as a simple, imperfect-information game.
- Uses tokens on a grid.
- Research game.
- Deterministic.
- Imperfect information.
- 2 players.
- Auger '11, Multiple Tree for Partially Observable Monte-Carlo Tree Search,Lisy '14, Alternative Selection Functions for Information Set Monte Carlo Tree Search, Lanctot '13

9.1.59 Pig

- Each player rolls a dice until they get a 1 or they 'hold'; the rolled total is added to their score.
- Dice game.
- Traditional game.
- Non-deterministic.
- Perfect information.

- 2-10 players.
- Wikipedia

9.1.60 Prisoner's Dilemma

- Players decide on whether to cooperate or defect given a situation with different payoffs.
- Simultaneous.
- Traditional game.
- Deterministic.
- Perfect Information.
- 2 players.
- Wikipedia

9.1.61 Poker (Hold 'em)

- Players bet on whether their hand of cards plus some communal cards will form a special set.
- Cards with bidding.
- Traditional game.
- Non-deterministic.
- Imperfect information.
- 2-10 players.
- Wikipedia
- Implemented via ACPC.
- Known issues: see issue #1033.

9.1.62 Quoridor

- Each turn, players can either move their agent or add a small wall to the board.
- Idiosyncratic format.
- Modern game.
- Deterministic.
- Perfect information.
- 2-4 players. (Note, from Wikipedia: "Though it can be played with 3 players, it's advised against. Since the 3rd player doesn't have player on the opposite side, they have an advantage.")
- Wikipedia
- Known issues: see #1158.

9.1.63 Reconnaissance Blind Chess

- Chess with opponent's pieces unknown, with sensing moves.
- Chess variant, invented by John Hopkins University Applied Physics Lab. Used in NeurIPS competition and Hidden Information Game Competition.
- Deterministic.
- Imperfect information.
- 2 players.
- JHU APL Main site
- Markowitz et al. '18, On the Complexity of Reconnaissance Blind Chess
- Newman et al. '16, Reconnaissance blind multi-chess: an experimentation platform for ISR sensor fusion and resource management
- Known issues: see #811.

9.1.64 Routing game

- Players choose at each node where they go. They have an origin, a destination and a departure time and choose their route to minimize their travel time. Time spent on each link is a function of the number of players on the link when the player reaches the link.
- Network with choice of route.
- Research game.
- Simultaneous.
- Deterministic.
- Perfect information.
- Any number of players.
- Cabannes et. al. '21, Solving N-player dynamic routing games with congestion: a mean field approach.

9.1.65 Sheriff

- Bargaining game.
- Deterministic.
- Imperfect information.
- 2 players.
- Good for correlated equilibria.
- Farina et al. '19, Correlation in Extensive-Form Games: Saddle-Point Formulation and Benchmarks.
- Based on the board game "Sheriff of Nottingham" (bbg)

9.1.66 Slovenian Tarok

- Trick-based card game with bidding.
- Traditional game.
- Non-deterministic.
- Imperfect information.
- 3-4 players.
- Wikipedia
- Luštrek et al. 2003, A program for playing Tarok

9.1.67 Skat (simplified bidding)

- Each turn, players bid to compete against the other two players.
- Cards with bidding.
- Traditional game.
- Non-deterministic.
- Imperfect information.
- 3 players.
- Wikipedia

9.1.68 Solitaire (K+)

- A single-player card game.
- Card game.
- Traditional game.
- Non-deterministic.
- Imperfect information.
- 1 players.
- Wikipedia and Bjarnason et al. '07, Searching solitaire in real time

9.1.69 Tic-Tac-Toe

- Players place tokens to try and form a pattern.
- Uses tokens on a grid.
- Traditional game.
- Deterministic.
- Perfect information.
- 2 players.
- Wikipedia

9.1.70 Tiny Bridge

- Simplified Bridge with fewer cards and tricks.
- Cards with bidding.
- Research game.
- Non-deterministic.
- Imperfect information.
- 2, 4 players.
- See implementation for details.

9.1.71 Tiny Hanabi

- Simplified Hanabi with just two turns.
- Idiosyncratic format.
- Research game.
- Non-deterministic.
- Imperfect information.
- 2-10 players.
- Foerster et al 2018, Bayesian Action Decoder for Deep Multi-Agent Reinforcement Learning

9.1.72 Trade Comm

- Players with different utilities and items communicate and then trade.
- Idiosyncratic format.
- Research game.
- Non-deterministic.
- Imperfect information.
- 2 players.
- A simple emergent communication game based on trading.

9.1.73 TwixT

- Players place pegs and links on a 24x24 square to connect a line between opposite sides.
- pegs and links on a grid.
- Modern game.
- Deterministic.
- Perfect information.
- 2 players.
- Wikipedia

9.1.74 Ultimate Tic-Tac-Toe

- Players try and form a pattern in local boards and a meta-board.
- Uses tokens on a grid.
- Deterministic.
- Perfect information.
- 2 players.
- Wikipedia

9.1.75 Weighted Voting Games

- Classic coalitional game.
- Players each have a weight w_i, and there is a quota q.
- Denote p the binary vector representing a coalition over n players. The utility is 1 is p dot w >= q, 0 otherwise.
- n players.
- Chalkiadakis, Elkind, & Wooldridge '12

9.1.76 Y

- Players place tokens to try and connect sides of a triangular board.
- Tokens on hex grid.
- Modern game.
- Deterministic.
- Perfect information.
- 2 players.
- Wikipedia

CHAPTER

TEN

-RANK

OpenSpiel now supports using Alpha-Rank ("-Rank: Multi-Agent Evaluation by Evolution", 2019) for both single-population (symmetric) and multi-population games. Specifically, games can be specified via payoff tables (or tensors for the >2 players case) as well as Heuristic Payoff Tables (HPTs).

The following presents several typical use cases for Alpha-Rank. For an example complete python script, refer to open_spiel/python/egt/examples/alpharank_example.py.

10.1 Importing the Alpha-Rank module

```
from open_spiel.python.egt import alpharank
from open_spiel.python.egt import alpharank_visualizer
```

10.2 Running Alpha-Rank on various games

10.2.1 Example: symmetric 2-player game rankings

In this example, we run Alpha-Rank on a symmetric 2-player game (Rock-Paper-Scissors), computing and outputting the rankings in a tabular format. We demonstrate also the conversion of standard payoff tables to Heuristic Payoff Tables (HPTs), as both are supported by the ranking code.

```
# Load the game
game = pyspiel.load_matrix_game("matrix_rps")
payoff_tables = utils.game_payoffs_array(game)

# Convert to heuristic payoff tables
payoff_tables= [heuristic_payoff_table.from_matrix_game(payoff_tables[0]),
                heuristic_payoff_table.from_matrix_game(payoff_tables[1].T)]

# Check if the game is symmetric (i.e., players have identical strategy sets
# and payoff tables) and return only a single-player's payoff table if so.
# This ensures Alpha-Rank automatically computes rankings based on the
# single-population dynamics.
_, payoff_tables = utils.is_symmetric_matrix_game(payoff_tables)

# Compute Alpha-Rank
(rhos, rho_m, pi, num_profiles, num_strats_per_population) = alpharank.compute(
    payoff_tables, alpha=1e2)
```

(continues on next page)

(continued from previous page)

```
# Report results
alpharank.print_results(payoff_tables, payoffs_are_hpt_format, pi=pi)
```

Output

```
Agent        Rank         Score
-----        ----         -----
0            1            0.33
1            1            0.33
2            1            0.33
```

10.2.2 Example: multi-population game rankings

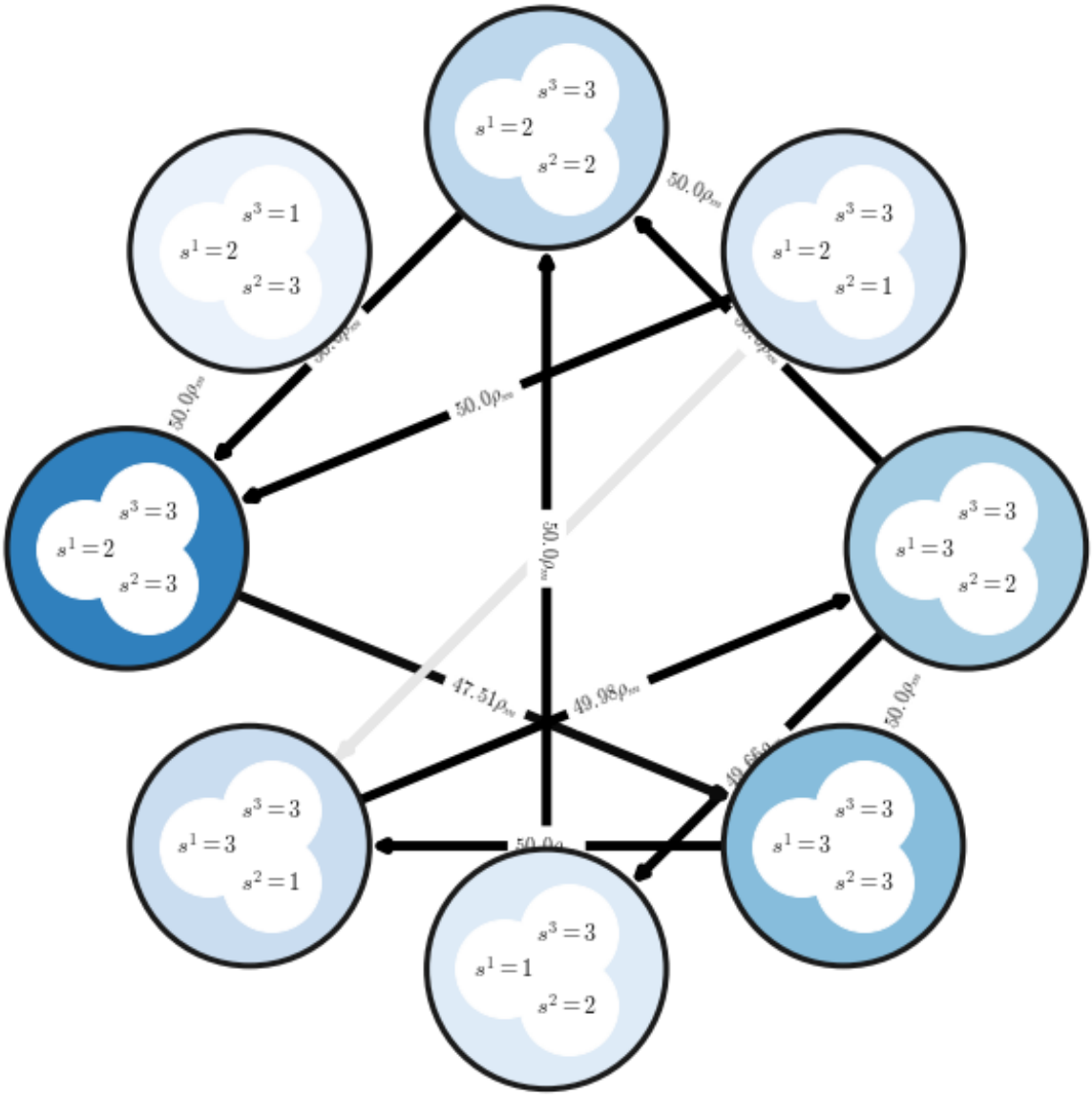
The next example demonstrates computing Alpha-Rank on an asymmetric 3-player meta-game, constructed by computing payoffs for Kuhn poker agents trained via extensive-form fictitious play (XFP). Here we use a helper function, `compute_and_report_alpharank`, which internally conducts the pre-processing and visualization shown in the previous example.

```
# Load the game
payoff_tables = alpharank_example.get_kuhn_poker_data(num_players=3)

# Helper function for computing & reporting Alpha-Rank outputs
alpharank.compute_and_report_alpharank(payoff_tables, alpha=1e2)
```

Output

```
Agent        Rank         Score
-----        ----         -----
(2,3,3)      1            0.22
(3,3,3)      2            0.14
(3,2,3)      3            0.12
(2,2,3)      4            0.09
(3,1,3)      5            0.08
(2,1,3)      6            0.05
(1,2,3)      7            0.04
(2,3,1)      8            0.02
...          ...          ...
```



10.3 Visualizing and reporting results

This section provides details on various methods used for reporting the final Alpha-Rank results.

10.3.1 Basic Ranking Outputs

The final rankings computed can be printed in a tabular manner using the following interface:

```
alpharank.print_results(payoff_tables, payoffs_are_hpt_format, pi=pi)
```

Output

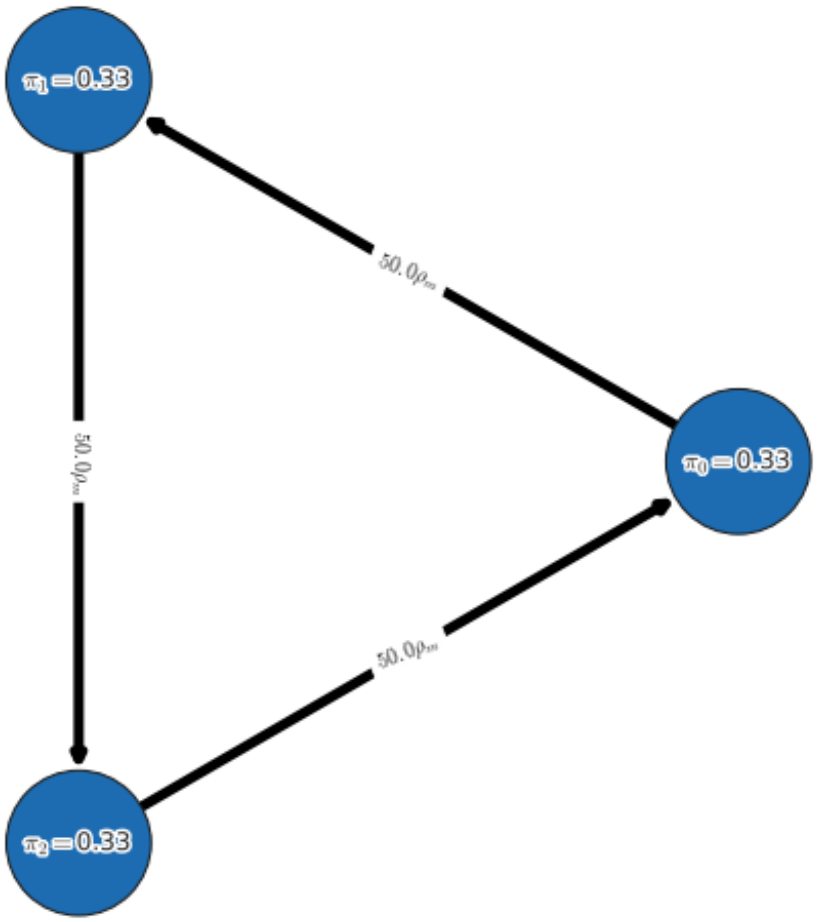
```
Agent        Rank         Score
-----        ----         -----
0            1            0.33
1            1            0.33
2            1            0.33
```

10.3.2 Markov Chain Visualization

One may visualize the Alpha-Rank Markov transition matrix as follows:

```
m_network_plotter = alpharank_visualizer.NetworkPlot(payoff_tables, rhos,
                                                     rho_m, pi,strat_labels,
                                                     num_top_profiles=8)
m_network_plotter.compute_and_draw_network()
```

Output

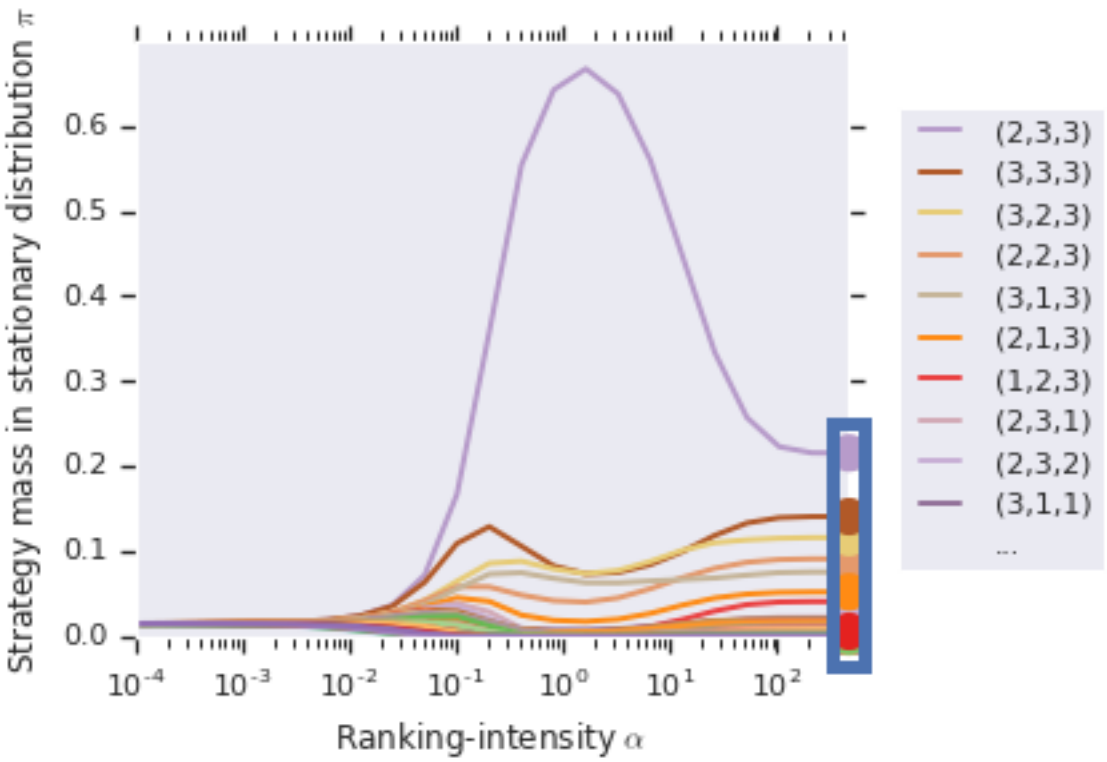


10.3.3 Alpha-sweep plots

One may choose to conduct a sweep over the ranking-intensity parameter, alpha (as opposed to choosing a fixed alpha). This is, in general, useful for general games where bounds on payoffs may be unknown, and where the ranking computed by Alpha-Rank should use a sufficiently high value of alpha (to ensure correspondence to the underlying Markov-Conley chain solution concept). In such cases, the following interface can be used to both visualize the sweep and obtain the final rankings computed:

```
alpharank.sweep_pi_vs_alpha(payoff_tables, visualize=True)
```

Output


Strategy mass in stationary distribution π
0.0
0.1
0.2
0.3
0.4
0.5
0.6
10^{-4}
10^{-3}
10^{-2}
10^{-1}
10^{0}
10^{1}
10^{2}
Ranking-intensity α
(2,3,3)
(3,3,3)
(3,2,3)
(2,2,3)
(3,1,3)
(2,1,3)
(1,2,3)
(2,3,1)
(2,3,2)
(3,1,1)
...

CHAPTER

ELEVEN

JULIA OPENSPIEL

We also provide a Julia wrapper for the OpenSpiel project. Most APIs are aligned with those in Python (some are extended to accept `AbstractArray` and/or keyword arguments for convenience). See `spiel.h` for the full API description.

11.1 Install

For general usage, you can install this package in the Julia REPL with `] add OpenSpiel`. Note that this method only supports the Linux platform and ACPC is not included. For developers, you need to follow the instructions bellow to install this package:

1. Install Julia and dependencies. Edit `open_spiel/scripts/global_variables.sh` and set `OPEN_SPIELOPEN_SPIEL_BUILD_WITH_JULIA=ON` (you may also turn on other options as you wish). Then run `./install.sh`. If you already have Julia installed on your system, make sure that it is visible in your terminal and its version is v1.3 or later. Otherwise, Julia v1.3.1 will be automatically installed in your home dir and a soft link will be created at `/usr/local/bin/julia`.
2. Build and run tests

```
./open_spiel/scripts/build_and_run_tests.sh
```

3. Install `] dev ./open_spiel/julia` (run in Julia REPL).

11.2 Known Problems

1. There's a problem when building this package on Mac with XCode v11.4 or above (see discussions here). To fix it, you need to install the latest `libcxxwrap` by following the instructions here after running `./install.sh`. Then make sure that the result of `julia --project=./open_spiel/julia -e 'using CxxWrap; print(CxxWrap.prefix_path())'` points to the newly built `libcxxwrap`. After that, build and install this package as stated above.

11.3 Example

Here we demonstrate how to use the Julia API to play one game:

```
using OpenSpiel

# Here we need the StatsBase package for weighted sampling
using Pkg
Pkg.add("StatsBase")
using StatsBase

function run_once(name)
    game = load_game(name)
    state = new_initial_state(game)
    println("Initial state of game[$(name)] is:\n$(state)")

    while !is_terminal(state)
        if is_chance_node(state)
            outcomes_with_probs = chance_outcomes(state)
            println("Chance node, got $(length(outcomes_with_probs)) outcomes")
            actions, probs = zip(outcomes_with_probs...)
            action = actions[sample(weights(collect(probs)))]
            println("Sampled outcome: $(action_to_string(state, action))")
            apply_action(state, action)
        elseif is_simultaneous_node(state)
            chosen_actions = [rand(legal_actions(state, pid-1)) for pid in 1:num_
↪players(game)]  # in Julia, indices start at 1
            println("Chosen actions: $([action_to_string(state, pid-1, action) for (pid,␣
↪action) in enumerate(chosen_actions)])")
            apply_action(state, chosen_actions)
        else
            action = rand(legal_actions(state))
            println("Player $(current_player(state)) randomly sampled action: $(action_
↪to_string(state, action))")
            apply_action(state, action)
        end
        println(state)
    end
    rts = returns(state)
    for pid in 1:num_players(game)
        println("Utility for player $(pid-1) is $(rts[pid])")
    end
end

run_once("tic_tac_toe")
run_once("kuhn_poker")
run_once("goofspiel(imp_info=True,num_cards=4,points_order=descending)")
```

11.4 Q&A

1. What is `StdVector`?

   `StdVector` is introduced in CxxWrap.jl recently. It is a wrapper of `std::vector` in the C++ side. Since that it is a subtype of `AbstractVector`, most functions should just work out of the box.

2. `0-based` or `1-based`?

   As this package is a low-level wrapper of OpenSpiel C++, most APIs are zero-based: for instance, the `Player` id starts from zero. But note that some bridge types, like `StdVector`, implicitly convert between indexing conventions, so APIs that use `StdVector` are one-based.

3. I can't find the `xxx` function/type in the Julia wrapper/The program exits unexpectedly.

   Although most of the functions and types should be exported, there is still a chance that some APIs are not well tested. So if you encounter any error, please do not hesitate to create an issue.

CHAPTER

TWELVE

ALPHAZERO

OpenSpiel includes two implementations of AlphaZero, one based on Tensorflow (in Python). The other based on C++ LibTorch. This document covers mostly the TF-based implementation and common components. For the Libtorch-based implementation, see here.

Disclaimer: this is not the code that was used for the Go challenge matches or the AlphaZero paper results. It is a re-implementation for illustrative purposes, and although it can handle games like Connect Four, it is not designed to scale to superhuman performance in Go or Chess.

12.1 Background

AlphaZero is an algorithm for training an agent to play perfect information games from pure self-play. It uses Monte Carlo Tree Search (MCTS) with the prior and value given by a neural network to generate training data for that neural network.

Links to relevant articles/papers:

- AlphaGo Zero: Starting from scratch has an open access link to the AlphaGo Zero nature paper that describes the model in detail.
- AlphaZero: Shedding new light on chess, shogi, and Go has an open access link to the AlphaZero science paper that describes the training regime and generalizes to more games.

12.2 Overview:

The Python and C++ implementations are conceptually fairly similar, and have roughly the same components: *actors* that generate data through self-play using *MCTS* with an *evaluator* that uses a *neural network*, a *learner* that updates the network based on those games, and *evaluators* playing vs standard MCTS to gauge progress. Both *write checkpoints* that can be *played* independently of the training setup, and logs that can be *analyzed* programmatically.

The Python implementation uses one process per actor/evaluator, doesn't support batching for inference and does all inference and training on the cpu. The C++ implementation, by contrast, uses threads, a shared cache, supports batched inference, and can do both inference and training on GPUs. As such the C++ implementation can take advantage of additional hardware and can train significantly faster.

12.2.1 Model

The model defined in open_spiel/python/algorithms/alpha_zero/model.py is used by both the python and C++ implementations.

The model defines three architectures in decreasing complexity:

- resnet: same as the AlphaGo/AlphaZero paper when set with width 256 and depth 20.
- conv2d: same as the resnet except uses a conv+batchnorm+relu instead of the residual blocks.
- mlp: same as conv2d except uses dense layers instead of conv, and drops batch norm.

The model is parameterized by the size of the observations and number of actions for the game you specify, so can play any 2-player game. The conv2d and resnet models are restricted to games with a 2d representation (ie a 3d observation tensor).

The models are all parameterized with a width and depth:

- The depth is the number of blocks in the torso, where the definition of a block varies by model. For a resnet it's a resblock which is two conv2ds, batch norms and relus, and an addition. For conv2d it's a conv2d, a batch norm and a relu. For mlp it's a dense plus relu.
- The width is the number of filters for any conv2d and the number of hidden units for any dense layer.

The networks all give two outputs: a value and a policy, which are used by the MCTS evaluator.

12.2.2 MCTS

Monte Carlo Tree Search (MCTS) is a general search algorithm used to play many games, but first found success playing Go back in ~2005. It builds a tree directed by random rollouts, and does usually uses UCT to direct the exploration/exploitation tradeoff. For our use case we replace random rollouts with a value network. Instead of a uniform prior we use a policy network. Instead of UCT we use PUCT.

We have implementations of MCTS in C++ and python.

12.2.3 MCTS Evaluator

Both MCTS implementations above have a configurable evaluator that returns the value and prior policy of a given node. For standard MCTS the value is given by random rollouts, and the prior policy is uniform. For AlphaZero the value and prior are given by a neural network evaluation. The AlphaZero evaluator takes a model, so can be used during training or with a trained checkpoint for play with open_spiel/python/examples/mcts.py.

12.2.4 Actors

The main script launches a set of actor processes (Python) or threads (C++). The actors create two MCTS instances with a shared evaluator and model, and play self-play games, passing the trajectories to the learner via a queue. The more actors the faster it can generate training data, assuming you have sufficient compute to actually run them. Too many actors for your hardware will mean longer for individual games to finish and therefore your data could be more out of date with respect to the up to date checkpoint/weights.

12.2.5 Learner

The learner pulls trajectories from the actors and stores them in a fixed size FIFO replay buffer. Once the replay buffer has enough new data, it does an update step sampling from the replay buffer. It then saves a checkpoint and updates all the actor's models. It also updates a `learner.jsonl` file with some stats.

12.2.6 Evaluators

The main script also launches a set of evaluator processes/threads. They continually play games against a standard MCTS+Solver to give an idea of how training is progressing. The MCTS opponents can be scaled in strength based on the number of simulations they are given per move, so more levels means stronger but slower opponents.

12.2.7 Output

When running the algorithm a directory must be specified and all output goes there.

Due to the parallel nature of the algorithm writing logs to stdout/stderr isn't very useful, so each actor/learner/evaluator writes its own log file to the configured directory.

Checkpoints are written after every update step, mostly overwriting the latest one at `checkpoint--1` but every `checkpoint_freq` is saved at `checkpoint-<step>`.

The config file is written to `config.json`, to make the experiment more repeatable.

The learner also writes machine readable logs in the jsonlines format to `learner.jsonl`, which can be read with the analysis library.

12.3 Usage:

12.3.1 Python

The code lives at open_spiel/python/algorithms/alpha_zero/.

The simplest example trains a tic_tac_toe agent for a set number of training steps:

```
python3 open_spiel/python/examples/tic_tac_toe_alpha_zero.py
```

Alternatively you can train on an arbitrary game with many more options:

```
python3 open_spiel/python/examples/alpha_zero.py --game connect_four --nn_model mlp --
↪actors 10
```

12.3.2 Analysis

There's an analysis library at open_spiel/python/algorithms/alpha_zero/analysis.py which reads the `config.json` and `learner.jsonl` from an experiment (either python or C++), and graphs losses, value accuracy, evaluation results, actor speed, game lengths, etc. It should be reasonable to turn this into a colab.

12.3.3 Playing vs checkpoints

The checkpoints are compatible between python and C++, and can be loaded by the model. You can try playing against one directly with open_spiel/python/examples/mcts.py:

```
python3 open_spiel/python/examples/mcts.py --game=tic_tac_toe --player1=human --
↪player2=az --az_path <path to your checkpoint directory>
```

CHAPTER

THIRTEEN

THE CODE STRUCTURE

Generally speaking, the directories directly under `open_spiel` are C++ (except for `integration_tests` and `python`). A similar structure is available in `open_spiel/python`, containing the Python equivalent code.

Some top level directories are special:

- `open_spiel/integration_tests`: Generic (python) tests for all the games.
- `open_spiel/tests`: The C++ common test utilities.
- `open_spiel/scripts`: The scripts useful for development (building, running tests, etc).

For example, we have for C++:

- `open_spiel/`: Contains the game abstract C++ API.
- `open_spiel/games`: Contains the games C++ implementations.
- `open_spiel/algorithms`: The C++ algorithms implemented in OpenSpiel.
- `open_spiel/examples`: The C++ examples.
- `open_spiel/tests`: The C++ common test utilities.

For Python you have:

- `open_spiel/python/examples`: The Python examples.
- `open_spiel/python/algorithms/`: The Python algorithms.

CHAPTER

FOURTEEN

C++ AND PYTHON IMPLEMENTATIONS.

Some objects (e.g. `Policy`, `CFRSolver`, `BestResponse`) are available both in C++ and Python. The goal is to be able to use C++ objects in place of Python objects for most of the cases. In particular, for the objects that are well supported, expect to have in the test for the Python object, a test checking that both the C++ and the Python implementation behave the same.

CHAPTER

FIFTEEN

ADDING A GAME

We describe here only the simplest and fastest way to add a new game. It is ideal to first be aware of the general API (see `open_spiel/spiel.h`). These guidelines primarily assume C++ games; the process is analogous for Python games and any special considerations are noted in the steps.

1. Choose a game to copy from in `open_spiel/games/` (or `open_spiel/python/games/`). Suggested games: Tic-Tac-Toe and Breakthrough for perfect information without chance events, Backgammon or Pig for perfect information games with chance events, Goofspiel and Oshi-Zumo for simultaneous move games, and Leduc poker and Liar's dice for imperfect information games. For the rest of these steps, we assume Tic-Tac-Toe.
2. Copy the header and source: `tic_tac_toe.h`, `tic_tac_toe.cc`, and `tic_tac_toe_test.cc` to `new_game.h`, `new_game.cc`, and `new_game_test.cc` (or `tic_tac_toe.py` and `tic_tac_toe_test.py`).
3. Configure CMake:
   - If you are working with C++: add the new game's source files to `open_spiel/games/CMakeLists.txt`.
   - If you are working with C++: add the new game's test target to `open_spiel/games/CMakeLists.txt`.
   - If you are working with Python: add the test to `open_spiel/python/CMakeLists.txt` and import it in `open_spiel/python/games/__init__.py`
4. Update boilerplate C++/Python code:
   - In `new_game.h`, rename the header guard at the the top and bottom of the file.
   - In the new files, rename the inner-most namespace from `tic_tac_toe` to `new_game`.
   - In the new files, rename `TicTacToeGame` and `TicTacToeState` to `NewGameGame` and `NewGameState`.
   - At the top of `new_game.cc`, change the short name to `new_game` and include the new game's header.
5. Update Python integration tests:
   - Add the short name to the list of expected games in `open_spiel/python/tests/pyspiel_test.py`.
6. You should now have a duplicate game of Tic-Tac-Toe under a different name. It should build and the test should run, and can be verified by rebuilding and running the example `build/examples/example --game=new_game`. Note: Python games cannot be run using this example; use `open_spiel/python/examples/example.py` instead.
7. Now, change the implementations of the functions in `NewGameGame` and `NewGameState` to reflect your new game's logic. Most API functions should be clear from the game you copied from. If not, each API function that is overridden will be fully documented in superclasses in `open_spiel/spiel.h`.
8. To test the game as it is being built, you can play test the functionality interactively using `ConsolePlayTest` in `open_spiel/tests/console_play_test.h`. At the very least, the test should include some random simulation tests (see other game's tests for an example). Note: Python games cannot be tested using `ConsolePlayTest`, however both C++ and Python games can also be tested on the console using `open_spiel/python/examples/mcts_example` with human players.

9. Run your code through a linter so it conforms to Google's style guides. For C++ use cpplint. For Python, use pylint with the pylintrc from the Google style guide. There is also YAPF for Python as well.
10. Once done, rebuild and rerun the tests to ensure everything passes (including your new game's test!).
11. Add a playthrough file to catch regressions:
    - Run `./open_spiel/scripts/generate_new_playthrough.sh new_game` to generate a random game, to be used by integration tests to prevent any regression. `open_spiel/integration_tests/playthrough_test.py` will automatically load the playthroughs and compare them to newly generated playthroughs.
    - If you have made a change that affects playthroughs, run `./scripts/regenerate_playthroughs.sh` to update them.

CHAPTER

SIXTEEN

CONDITIONAL DEPENDENCIES

The goal is to make it possible to optionally include external dependencies and build against them. The setup was designed to met the following needs:

- **Single source of truth**: We want a single action to be sufficient to manage the conditional install and build. Thus, we use bash environment variables, that are read both by the install script (`install.sh`) to know whether we should clone the dependency, and by CMake to know whether we should include the files in the target. Tests can also access the bash environment variable.
- **Light and safe defaults**: By default, we exclude the dependencies to diminish install time and compilation time. If the bash variable is unset, we download the dependency and we do not build against it.
- **Respect the user-defined values**: The `global_variables.sh` script, which is included in all the scripts that needs to access the constant values, do not override the constants but set them if and only if they are undefined. This respects the user-defined values, e.g. on their `.bashrc` or on the command line.

When you add a new conditional dependency, you need to touch:

- the root CMakeLists.txt to add the option, with an OFF default
- add the option to `scripts/global_variables.sh`
- change `install.sh` to make sure the dependency is installed
- use constructs like `if (${OPEN_SPIEL_BUILD_WITH_HANABI})` in CMake to optionally add the targets to build.

CHAPTER

SEVENTEEN

DEBUGGING TOOLS

For complex games it may be tricky to get all the details right. Reading through the playthrough (or visually inspecting random games via the example) is the first step in verifying the game mechanics. You can visualize small game trees using open_spiel/python/examples/treeviz_example.py or for large games there is an interactive viewer for OpenSpiel games called SpielViz.

CHAPTER

EIGHTEEN

ADDING GAME-SPECIFIC FUNCTIONALITY

OpenSpiel focuses on maintaining a general API to an underlying suite of games, but sometimes it is convenient to work on specific games. In this section, we describe how to get (or set) game-specific information from/to the generic state objects, and how to expose these functions to python.

Suppose, for example, we want to look at (or set) the private cards in a game of Leduc poker. We will use an example based on this this commit.

1. First, locate the game you want to access. The game implementations are in the `games/` subdirectory and have two main files: e.g. `leduc_poker.h` (header) and `leduc_poker.cc` (implementation).
2. For simple accessor methods that just return the information and feel free have the full implementation to the game's header file (e.g. `LeducState::GetPrivateCards`). You can also declare the function in the header and provide the implementation in source file (e.g. `LeducPoker::SetPrivateCards`).
3. That's it for the core game logic. To expose these methods to Python, add them to the Python module (via pybind11). Some games already have game-specific functionality, so if a files named `games_leduc_poker.h` and `games_leduc_poker.cc` exist within `python/pybind11`, add to them (skip to Step 5).
4. If the games-specific files do not exist for your game of interest, then:
   - Add the files. Copy one of the other ones, adapt the names, and remove most of the bindings code.
   - Add the new files to the `PYTHON_BINDINGS` list in `python/CMakeFiles.txt`.
   - Modify `pyspiel.cc`: include the header at the top, and call the init function at the bottom.
5. Add the custom methods to the game-specific python bindings (`games_leduc_poker.cc`, i.e. `LeducPoker::GetPrivateCards` and `LeducPoker::SetPrivateCards`). For simple types, this should be relatively straight-forward; you can see how by looking at the other game-specific functions. For complex types, you may have to bind additional code (see e.g. `games_backgammon.cc`). If it is unclear, do not hesitate to ask, but also please check the pybind11 documentation.
6. Add a simple test to `python/games_sim_test.py` to check that it worked. For inspiration, see e.g. `test_leduc_get_and_set_private_cards`.

CHAPTER

NINETEEN

LANGUAGE APIS

There are currently four other language APIs that expose functionality from the C++ core.

- Python.
- Julia
- Go (experimental)
- Rust (experimental)

CHAPTER

TWENTY

GUIDELINES

Above all, OpenSpiel is designed to be easy to install and use, easy to understand, easy to extend ("hackable"), and general/broad. OpenSpiel is built around two major important design criteria:

- **Keep it simple.** Simple choices are preferred to more complex ones. The code should be readable, usable, extendable by non-experts in the programming language(s), and especially to researchers from potentially different fields. OpenSpiel provides reference implementations that are used to learn from and prototype with, rather than fully-optimized / high-performance code that would require additional assumptions (narrowing the scope / breadth) or advanced (or lower-level) language features.
- **Keep it light.** Dependencies can be problematic for long-term compatibility, maintenance, and ease-of- use. Unless there is strong justification, we tend to avoid introducing dependencies to keep things easy to install and more portable.

CHAPTER

TWENTYONE

SUPPORT EXPECTATIONS

We, the OpenSpiel authors, definitely engage in supporting the community. As it can be time-consuming, we try to find a good balance between ensuring we are responsive and being able to continue to do our day-to-day work and research.

Generally speaking, if you are willing to get a specific feature implemented, the most effective way is to implement it and send a Pull Request. For large changes, or ones involving design decisions, open a bug to check the idea is ok first.

The higher the quality, the easier it will be to be accepted. For instance, following the C++ Google style guide and Python Google style guide will help with the integration.

As examples, MacOS support, Window support, example improvements, various bug-fixes or new games has been straightforward to be included and we are very thankful to everyone who helped.

21.1 Bugs

We aim to answer bugs at a reasonable pace, several times a week. However, for bugs involving large changes (e.g. adding new games, adding public state supports) we cannot commit to implementing it and encourage everyone to contribute directly.

21.2 Pull requests

You can expect us to answer/comment back and you will know from the comment if it will be merged as is or if it will need additional work.

For pull requests, they are merged as batches to be more efficient, at least every two weeks (for bug fixes, it will likely be faster to be integrated). So you may need to wait a little after it has been approved to actually see it merged.

CHAPTER

TWENTYTWO

ROADMAP AND CALL FOR CONTRIBUTIONS

Contributions to this project must be accompanied by a Contributor License Agreement (CLA). See CONTRIBUTING.md for the details.

Here, we outline our current highest priorities: this is where we need the most help. There are also suggestion for larger features and research projects. Of course, all contributions are welcome.

Before making a contribution to OpenSpiel, please read the guidelines. We also kindly request that you contact us before writing any large piece of code, in case (a) we are already working on it and/or (b) it's something we have already considered and may have some design advice on its implementation. Please also note that some games may have copyrights which might require legal approval. Otherwise, happy hacking!

- **Long-term and Ongoing Maintenance**. This is the most important way to help. Having OpenSpiel bug-free and working smoothly is the highest priority. Things can stop working for a variety of reasons due to version changes and backward incompatibility, but also due to discovering new problems that require some time to fix. To see these items, look for issues with the "help wanted" tag on the Issues page.
- **New Features and Algorithmsx**. There are regular requests for new features and algorithms that we just don't have time to provide. Look for issues with the "contribution welcome" tag on the Issues page.
- **Windows support**. Native Windows support was added in early 2022, but remains experimental and only via building from source. It would be nice to have Github Actions CI support on Windows to ensure that Windows support is actively maintained, and eventually support installing OpenSpiel via pip on Windows as well. The tool that builds the binary wheels (cibuildwheel) already supports Windows as a target platform.
- **Visualizations of games**. There exists an interactive viewer for OpenSpiel games called SpielViz. Contributions to this project, and more visualization tools with OpenSpiel, are very welcome as they could help immensely with debugging and testing the AI beyond the console.
- **Structured Action Spaces**. Currently, actions are integers between 0 and some value. There is no easy way to interpret what each action means in a game-specific way. Nor is there any way to easily represent a composite action in terms of its parts. A structured action space could represent actions as a sequence of values (like information states and observations– and can also include shapes) which can be learned instead of mappings to flat numbers. Then, each game could have a mapping from the structured action to the action taken.
- **APIs for other languages** (Go, Rust, Julia). We currently have these supported but little beyond the core API and random simulation tests. Several are very basic (or experimental). It would be nice to properly support these by having a few simple algorithms run via the bindings on OpenSpiel games.
- **New Games**. New games are always welcome. If you do not have one in mind, check out the Call for New Games issue.

CHAPTER

TWENTYTHREE

USING OPENSPIEL AS A C++ LIBRARY

OpenSpiel has been designed as a framework: a suite of games, algorithms, and tools for research in reinforcement learning and search in games. However, there are situations where one may only want or need a single game/algorithm or small subset from this collection, or a research experiment does not require modifying or otherwise interacting very closely with OpenSpiel other than strictly calling/using it.

In cases like this, it might be nice to use OpenSpiel as a library rather than a framework. This has the benefit of not forcing the use of certain tools like CMake or having to continually recompile OpenSpiel when doing your research.

Luckily, this is easy to achieve with OpenSpiel: you simply need to build it as a shared library once, and then load it dynamically at runtime. This page walks through how to do this assuming a bash shell on Linux, but is very similar on MacOS or for other shells.

23.1 Install Dependencies

The dependencies of OpenSpiel need to be installed before it can be used as a library. On MacOS and Debian/Ubuntu Linux, this is often simply just running `./install.sh`. Please see the installation from source instructions for more details.

23.2 Compiling OpenSpiel as a Shared Library

To build OpenSpiel as a shared library, simply run:

```
mkdir build
cd build
BUILD_SHARED_LIB=ON CXX=clang++ cmake -DPython3_EXECUTABLE=$(which python3) -DCMAKE_CXX_
↪COMPILER=${CXX} ../open_spiel
make -j$(nproc) open_spiel
```

This produces a dynamically-linked library `libopen_spiel.so` (or `lib_openspiel.dylib` on MacOS) in `build/` that can be linked against and loaded dynamically at run-time.

Suppose OpenSpiel was installed in `$HOME/open_spiel`. The following line adds the necessary environment variable to let the shell know where to find `libopen_spiel.so` at run-time:

```
export LD_LIBRARY_PATH="${HOME}/open_spiel/build"
```

You might want to add this line to your `$HOME/.bash_profile` to avoid having to do it every time you load the library. Of course, if you are already using `LD_LIBRARY_PATH` for something else, then you need to add `${HOME}/open_spiel/build` to it (space-separated paths).

23.3 Compiling and Running the Example

```
cd ../open_spiel/examples
clang++ -I${HOME}/open_spiel -I${HOME}/open_spiel/open_spiel/abseil-cpp \
        -std=c++17 -o shared_library_example shared_library_example.cc \
        -L${HOME}/open_spiel/build  -lopen_spiel
```

The first two flags are the include directory paths and the third is the link directory path. The `-lopen_spiel` instructs the linker to link against the OpenSpiel shared library.

That's it! Now you can run the example using:

```
./shared_library_example breakthrough
```

You should also be able to register new games externally without the implementation being within OpenSpiel nor built into the shared library, though we are always interested in growing the library and recommend you contact us about contributing any new games to the suite.

CHAPTER

TWENTYFOUR

AUTHORS

Names are ordered lexicographically. Typo or similar contributors are omitted.

24.1 OpenSpiel contributors

- Bart De Vylder
- Edward Hughes
- Edward Lockhart locked@google.com
- Daniel Hennes
- David Ding
- Dustin Morrill
- Elnaz Davoodi
- Finbarr Timbers
- Ivo Danihelka
- Jean-Baptiste Lespiau jblespiau@google.com
- Janos Kramar
- Jonah Ryan-Davis
- Julian Schrittwieser
- Julien Perolat
- Karl Tuyls
- Manuel Kroiss
- Marc Lanctot lanctot@google.com
- Matthew Lai
- Michal Sustr michal.sustr@aic.fel.cvut.cz
- Raphael Marinier
- Paul Muller
- Ryan Faulkner
- Satyaki Upadhyay
- Sebastian Borgeaud

- Sertan Girgin
- Shayegan Omidshafiei
- Srinivasan Sriram
- Thomas Anthony
- Thomas Köppe
- Timo Ewalds tewalds@google.com
- Vinicius Zambaldi vzambaldi@google.com

24.2 OpenSpiel with Swift for Tensorflow (now removed)

- James Bradbury jekbradbury@google.com
- Brennan Saeta saeta@google.com
- Dan Zheng danielzheng@google.com

24.3 External contributors

See https://github.com/deepmind/open_spiel/graphs/contributors.