Lab 3.

The following exercises contain alternative sub-tasks of varying effort and difficulty. You are free to choose any sub-task that you want to do. We distinguish between simple (∗), more involved (∗∗), and challenging / time-consuming (∗ ∗ ∗) tasks. We recommend to start with the simple or more involved exercises, and move on to the challenging exercises only if time allows.[1]
All instructions below assume that you are in directory `/vagrant/fast-downward`, i.e., all paths are relative to this base path, unless stated otherwise.
If you make changes to the planning system, you need to recompile it with

```
./build.py release64
```

from the base path.

Exercise 1. Modeling PDDL: Sokoban

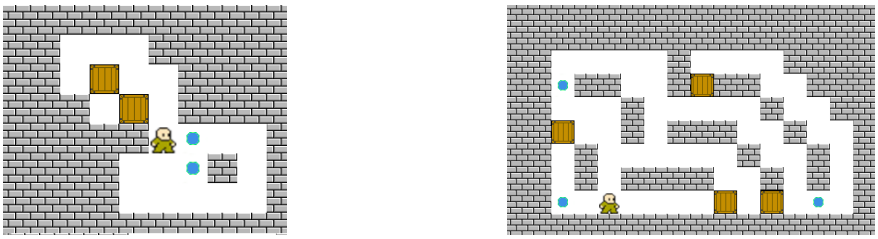

Figure 1: Two initial states for Sokoban

Sokoban (`https://en.wikipedia.org/wiki/Sokoban`) is a puzzle where the player needs to push crates from their initial position to designated goal locations (marked by blue dots).
The game is played on a grid, where some positions are blocked by walls. The player can move horizontally and vertically but cannot walk through a wall or enter a position that is blocked by a crate. If the player stands next to a crate and the position behind this crate

[1] Sample solutions to all exercises are provided with the repository. To check-out the solution of a specific exercise, run `hg revert -r summer_school_18_solution PATH` where `PATH` is replaced by the path to the file(s) of the respective exercise. Do not run this command without specifying `PATH`, unless you want to checkout the solutions to all exercises.

is empty, it can push the crate to this position (which leaves the player at the original position of the crate). It is not possible to push several crates at once or to pull a crate. At the end, each crate should rest on a goal position, but it does not matter which crate ends up on which position. The solution quality is measured by the number of pushes.
In this exercise, you will solve some instances of the puzzle with the Fast Downward planning system.

1.) Create the model (**) In a first step, we need to model the game in PDDL. Extend the domain file in `models/sokoban-pddl/domain.pddl` with suitable predicates and actions. Then encode the instance from the left example picture in `models/sokoban-pddl/small.pddl` by specifying suitable objects, the initial state and the goal condition (using your predicates).

For debugging, you can use the tool `parser` from the plan validation tool VAL or (much less verbose, not reporting all errors) the `validate` tool itself with the `-v` option:

```
VAL/parser domain.pddl small.pddl
VAL/validate -v domain.pddl small.pddl
```

(*) If you prefer to solve a simpler variant of this exercise, you can retrieve the solution[1] for `models/sokoban-pddl/small.pddl` and use the predicates from there in the domain file.

If you are done, you can model the right example task in file `models/sokoban-pddl/medium.pddl` (or retrieve it from the solution branch). Note that in the picture of this example task, one of the goal positions is obscured by a crate sitting on top of it. The fourth goal position is the one where the rightmost crate is located in the picture.

2.) Run the planner (*) Now you can run the planner on your instances, for example on the small instance with the A*-algorithm and the LM-Cut heuristic:

```
./fast-downward.py --build=release64 \
  models/sokoban-pddl/domain.pddl \
  models/sokoban-pddl/small.pddl \
  --search "astar(lmcut())"
```

Have a look at the output, which (besides some other information) contains the plan and some statistics on the search. Is the plan optimal or is there a cheaper solution?

You find the list of implemented search engines and heuristics at `http://www.fast-downward.org/`. What happens to plan length and search time if you use other configurations, e.g. greedy best first search with the FF heuristic (this is less interesting with the small instance, which is naturally very easy)?

3.) Connect your model to the GUI (∗∗∗) The summer school version of the planner already ships a graphical front-end for Sokoban, allowing you to see the different planner configurations in action. To use it with your model, you need to implement an interface to your encoding of the Sokoban board. The implementation is in `src/search/sokoban/interface.cc`. In a nutshell, you will have to parse predicate instantiations, *facts*, and action instantiations of your model, and translate them to provided concepts. **TODO** and comments in the source file give more information on what needs to be implemented. Note that the same GUI will be used for the probabilistic planning exercises 3 and 4. You can ignore the aspects of the code that relate to probabilistic extensions of Sokoban (see following exercises).

With

```
hg revert -r summer_school_18_solution \
    src/search/sokoban/interface.cc
```

you can get the implementation for the reference Sokoban encoding (which might not work though with your model). You can start the GUI as follows:

```
# Starting the graphical Sokoban simulator,
# using A* with the LM-Cut heuristic.
  ./fast-downward.py --build=release64 \
    models/sokoban-pddl/domain.pddl \
    models/sokoban-pddl/small.pddl \
    --search "sokoban(astar(lmcut()))"
```

The speed of the simulation can be controlled via number keys `0` – `6`. `Space` starts and pauses the simulation. `ESC` closes the window.

Exercise 2. Classical Planning: Network Flow Heuristic (∗∗∗)

To solve this exercise, you must have installed the SOPLEX LP solver (following the instructions from the email). If you have not done this already, we recommend to continue with some other exercise because compiling everything takes quite some time.
In this exercise you will implement the network flow heuristic in Fast Downward. The planner already comes with a framework for so-called operator counting heuristics. To add a new heuristic, you only need to implement the setup of the constraints. You find a stub for your implementation in file `search/operator_counting/flow_constraints.cc`.

1.) Preparation It makes sense to precompute some information before setting up the linear program. The `AtomInfo` stores for each atom the actions that produce and

consume it (and will also remember the index of the corresponding constraint). Complete `FlowConstraints::build_atom_information` so that it precomputes the relevant action sets.

2.) Set up the constraints In the next step in the heuristic initialization we actually add the constraints. Note that in the definition of the constraints (in the slides) the current state is only relevant for the (lower) bound. For this reason, we add the constraints once and only update the bounds for each heuristic computation. This makes the implementation more efficient and is also very beneficial for the performance of the LP solver.

Finish the implementation of `FlowConstraints::add_constraints`.

3.) Update constraints If an operator counting heuristic gets evaluated on a state it first calls `update_constraints` and then runs the LP solver. In the case of the network flow constraints, we only need to set the lower bound for each constraint.

Complete the implementation in `FlowConstraints::update_constraints`.

4.) Run it! You can run your implementation with:

```
# Running A* with the Network Flow Heuristic
./fast-downward.py --build=release64 \
  models/sokoban-pddl/domain.pddl \
  models/sokoban-pddl/small.pddl --search \
  "astar(operatorcounting([flow_constraints()], lpsolver=soplex))"
```

For comparison, you can run the original implementation using `state_equation_constraints` instead of `flow_constraints`. If your implementation is correct, you will see the same number of expanded states on each f-layer (`f = ... [... expanded ...]`).

The operator counting framework allows to combine constraints from different sources. For example, using `[flow_constraints(),lmcut_constraints()]` includes also some landmark constraints, strengthening the heuristic.

Exercise 3. Modeling PPDDL: Probabilistic Sokoban

Create a PPDDL[2] Sokoban model by extending the Sokoban PDDL model that you have designed in an earlier exercise with the probabilistic action effects listed below. In all

[2] `http://reports-archive.adm.cs.cmu.edu/anon/2004/CMU-CS-04-167.pdf`

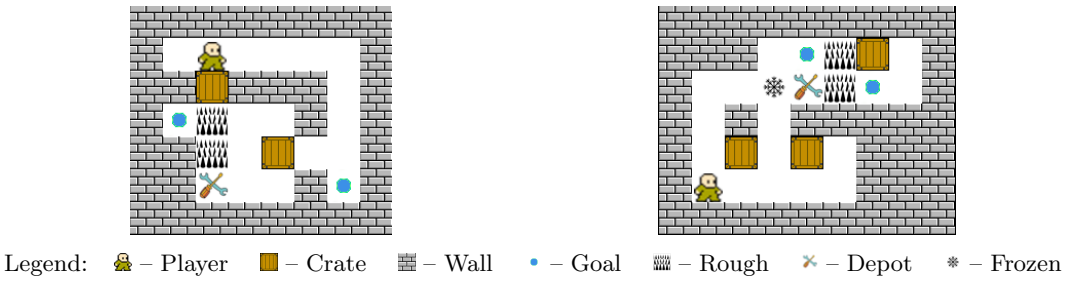

Figure 2: Two initial states for probabilistic Sokoban: (left) 8×7 board, requires rough ground; (right) 9×7 board, requires rough and frozen ground. Depending on the type of the modeled repair action, the depot cell can be left out.

succeeding exercises involving any probabilistic Sokoban version, we are interested in minimizing the expected cost to reach the goal (ExpCost). Note that in probabilistic Fast-Downward (`P-FD`), ExpCost is internally represented as reward-maximization, where action rewards are given by their negative cost. To ensure that the SSP requirements are met, `P-FD` automatically introduces a *give-up* action with cost 1000 that is applicable in every state, and leads to a goal state. Consequently, the minimal expected cost to reach the goal is bounded from above by 1000 for every state.
Store your PPDDL domain file in `models/sokoban-ppddl/domain.ppddl`. We will refer to this path by domain.

1.) Rough ground Certain cells of the Sokoban board may have rough ground. Pushing a crate onto cells with rough ground has a chance of 30% to damage the crate. Damaged crates cannot be pushed any further. To deal with damaged crates, you can choose to implement one of the two following actions:

(∗) A player can repair a damaged crate if the crate is located next to the current player position (in one of the four directions: up, down, left, right). Repairing a crate has cost 10.

(∗∗) Repairing a crate additionally requires the player to have a *repair-kit*. Repair-kits can be picked up by a player at certain *depot* cells (the player must be located at such a cell in order to pick-up a repair-kit). The repair-kit is consumed when the player repairs a damaged crate. Every player can carry at most one repair-kit at any time. Repairing a crate has cost 5, picking up a repair kit has cost 2.

Model the instance shown in Figure 2 (left), and store the PPDDL problem file in `models/sokoban-ppddl/small.ppddl`. We refer to this path by small. Solve this instance optimally using LRTDP:

```
./fast-downward.py --build=release64 domain small --search "lrtdp"
```

The minimal expected cost value for the initial state should be 39 for the simple repair action and 37.14 otherwise.

2.) Frozen ground Certain cells of the Sokoban board might be frozen. Pushing crates onto frozen cells has a chance of 50% to push the crate one cell further, i.e., to the next neighboring cell of the frozen cell in push direction, if that cell is not blocked. (If the next neighboring cell is blocked, the crate cannot slip on, and is guaranteed to move onto the frozen cell with probability 1.)

(**) You can assume that every cell that is adjacent to some frozen cell is not frozen.

(***) If the crate slips on to the neighboring cell of the frozen cell and this cell is frozen too, there is another chance of 50% that the crate moves even further. This scheme continues until the neighboring cell is either not frozen, or is blocked by a wall or other object. (Hint: it might make sense to model this not as a single action, but multiple actions, some of which have 0-cost.) In the VM, you can get the solution as well as an example instance by running the following command in the fast-downward folder:

```
hg revert -r summer_school_18_solution \
     models/sokoban-ppddl/domain-skate.ppddl \
     models/sokoban-ppddl/skate.ppddl
```

Model the instance shown in Figure 2 (right), and store the PPDDL problem file in `models/sokoban-ppddl/medium.ppddl`. We refer to this path by medium. Solve this instance optimally using LRTDP:

```
./fast-downward.py --build=release64 domain medium --search "lrtdp"
```

The minimal expected cost value for the initial state should be 53 for the simple repair action and 51.59 otherwise.

Connect your model to the GUI, continuation of Exercise 1.3 (***) Extend your implementation of the interface to the GUI from Exercise 1.3 to the probabilistic Sokoban extensions.

The reference solution of this exercise works for the reference solution for the PPDDL model from Exercises 3.1 and 3.2. You can obtain it with

```
hg revert -r summer_school_18_solution \
    src/search/sokoban/interface.cc
```

Don't forget to rebuild with `./build.py release64`. You can start the GUI as follows:

```
# Starting the graphical Sokoban simulator,
# using LRTDP to decide which action to take when
# num=20 lets the simulator run a total of 20 simulations of the
# policy and average the results
./fast-downward.py --build=release64 domain small \
    --search "sokoban_simulator(offline(lrtdp()), num=20)"
```

The speed of the simulation can be controlled via number keys `0` – `6`. `Space` pauses the simulation. `ESC` closes the window.

Exercise 4. Implementing an MDP Solver

In the following, you will be implementing your own MDP solver with the help of classical planners. You may want to use the reference Sokoban model as well as the interface from the solution branch if you want to have a graphical representation of what your implementation is doing. Remember though to backup your own model before running `hg revert`.

1.) All-outcomes determinization (*) Implement the all-outcomes determinization in `P-FD`. The all-outcomes determinization allows to connect heuristics from classical planning with MDP heuristic search algorithms (such as LRTDP), and builds the basis for the next exercise. The implementation is in `src/search/probabilistic/determinization/all_outcomes_determinization.cc`. Follow the **TODO** and comments in the source file.

```
# Run LRTDP using the FF heuristic on domain and medium
./fast-downward.py --build=release64 domain medium \
    --determinization all_outcomes \
    --search "lrtdp(eval=classic(ff))"
```

2.) FF-Replan (**) Implement an online MDP solver that is based on replanning. More precisely, implement an engine that given a state, decides the action to be executed in this state based on a classical plan for that state. Classical plans are obtained from running a classical planner on the all-outcomes determinization. The implementation is in `src/search/probabilistic/engines/ff_replan/ff_replan.cc`. Follow the **TODO** and comments in the source file.

a) Caching (**) To improve efficiency, you may want to cache the state-action assignments that are induced by the computed plans. For that you need to re-execute the plan step-by-step, and for each touched state store the respective

action given by the plan. Then, whenever a state is considered for which an action has already been cached, that action can be returned immediately, sparing calls to the classical planner.

b) Run it! (∗) Run your implementation on the instances you have modeled before, and compare it to LRTDP:

```
# Get the average expected cost after 1000 simulation runs
# for the policy computed by LRTDP with the FF heuristic
./fast-downward.py --build=release64 domain medium \
    --determinization all_outcomes \
    --search "simulate(engine=offline(lrtdp(eval=classic(ff))))"

# Run 1000 simulation runs, using your replanning engine
# to decide which actions to take
./fast-downward.py --build=release64 domain medium \
    --determinization all_outcomes \
    --search "simulate(engine=ff_replan())"

# Run your replanning engine inside the GUI (if you have
# already implemented the interface) for 20 simulation runs
./fast-downward.py --build=release64 domain medium \
    --determinization all_outcomes \
    --search "sokoban_simulator(engine=ff_replan(), num=20)"
```

The default configuration uses Dijkstra search to compute plans. You can choose different configurations via the `classical_planner` option, e.g.,

```
    ff_replan(classical_planner=\"eager_greedy(evals=[ff])\")
```

using greedy best first search with the FF heuristic (note that you have to encapsulate the configuration with quotation marks).

c) New Sokoban instance (∗∗) Design a new probabilistic Sokoban instance (using the modifications from before) where completely ignoring probabilistic effects and greedily following classical plans has a high chance of resulting in dead-end states (states without path to the goal), while in an optimal policy one can avoid visiting such states. Ideally, your model should be small enough so that LRTDP can still solve it. To get the solution, run

```
hg revert -r summer_school_18_solution \
    models/sokoban-ppddl/replan.ppddl
```

3.) FF-Hindsight (∗ ∗ ∗) Implement a simplified version of FF-Hindsight, where the next action to take is not computed from a single plan as in FF-Replan but from multiple plans. Instead of using the all-outcomes determinization for the plan computation,

implement and use the *sampled-outcome* determinization. The sampled-outcome determinization has one deterministic action for each probabilistic action, corresponding to a randomly sampled outcome of the probabilistic action (simplifying here the original FF-Hindsight approach by ignoring the timestep, i.e., assuming that the same outcome happens at all future action applications).

To decide the action a_{best} to take in the given state s, proceed as follows. Let $\hat{\Pi}$ denote any sampled-outcome determinization. Let a be any probabilistic action that is applicable in s, let $\hat{a}$ denote the respective determinized action in $\hat{\Pi}$, and let s' be the state resulting from applying $\hat{a}$ to s. The value of s and a in $\hat{\Pi}$ is then given by $\hat{V}(s, a, \hat{\Pi}) = \text{cost}_\pi$ for some deterministic plan π for s' in $\hat{\Pi}$; and $\hat{V}(s, a, \hat{\Pi}) = \text{cost}_{\text{give-up}} = 1000$ if no such plan was found. a_{best} is given by

$$a_{\text{best}} := \underset{a \text{ applicable in } s}{\arg\min} \sum_{\text{considered } \hat{\Pi}} \hat{V}(s, a, \hat{\Pi})$$

where the sum is over a set of w randomly chosen sampled-outcome determinizations.

Your implementation can be called as follows:

```
# Run 1000 simulation runs, using your FF-Hindsight engine
# to decide which actions to take
./fast-downward.py --build=release64 domain medium \
    --search "simulate(engine=ff_hindsight())"
```

This call defaults to $w = 5$ and uses again Dijkstra search to compute plans. You can control those parameters via `width` and `classical_planner`, e.g.,

```
    ff_hindsight(classical_planner=\"eager_greedy(evals=[ff])\"))
```

The implementation of the sampled-outcome determinization is in `src/search/probabilistic/engines/ff_hindsight/hindsight_determinization.cc`, that for FF-Hindsight is in `src/search/probabilistic/engines/ff_hindsight/ff_hindsight.cc`. Follow the **TODO** and comments in the source file.