Foundations of Artificial Intelligence

M. Helmert
S. Eriksson
Spring Term 2022

University of Basel
Computer Science

Exercise Sheet 13

Due: May 29, 2022

All exercises on this sheet are bonus exercises.

For this exercise you should *formalize*, *analyze* and *solve* a concrete problem with one of the approaches presented in the lecture. You can solve the sheet with **one** of the following settings:

- solving Sliding Tiles optimally with *state space search*
- solving k-Vertex Cover with *combinatorial optimization*
- solving k-Vertex Cover with *propositional logic*
- solving Sliding Tiles optimally with *automated planning*

Please choose only one of the settings above and do not switch between settings for the different parts of the exercise. Solving several settings will not yield more marks.

Sliding Tiles: In the sliding tiles puzzle we are given an $\mathbf{n} \times \mathbf{m}$ grid (position $(1, 1)$ is the upper left corner) in which each cell contains either a tile containing a number from 1 to $\mathbf{nm} - 1$, or is blank, denoted by $\square$. The initial configuration is given as a surjective function $\mathbf{f} : \{1, \ldots, \mathbf{n}\} \times \{1, \ldots, \mathbf{m}\} \to \{1, \ldots, \mathbf{nm} - 1\} \cup \{\square\}$. The goal is to rearrange the tiles by sliding them into the blank position (which changes the location of the blank as well), such that from left to right, top to bottom, the tiles are ordered in ascending order, and the blank is in the bottom right position. Slide 17 of Chapter 19 (print version) shows the goal configuration for a 4×4 puzzle (on the right side). All actions have cost 1.

k-Vertex Cover: Given an undirected graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ and natural number $\mathbf{k} \in \mathbb{N}_1$, the vertex cover problem asks whether there is a set $V' \subseteq \mathbf{V}$ of at most size $\mathbf{k}$ such that $v_i \in V'$ or $v_j \in V'$ for all $\{v_i, v_j\} \in \mathbf{E}$.

Exercise 13.1 (3 marks)

Formalize the problem in the formalism of the setting you consider and explain your formalization on a high level (two to three sentences).

- *state space search*: Formalize a state space space $\mathcal{S} = \langle S, A, \mathit{cost}, T, s_0, S^* \rangle$ describing Sliding Tiles. This exercise is similar to Exercise 2.3, but instead of a concrete problem instance your formalization should be parameterized by $\mathbf{n}$, $\mathbf{m}$ and $\mathbf{f}$.
- *combinatorial optimization*: Formalize k-Vertex Cover as a combinatorial optimization problem $P = \langle C, S, \mathit{opt}, v \rangle$.
- *propositional logic*: Formalize k-Vertex Cover as a propositional formula φ in CNF. Also specify the set of variables used in φ.

  Hint: The tricky part is ensuring that the vertex cover has at most size $\mathbf{k}$*. We suggest using variables* $x_{i,v}$ *encoding that vertex* v *has been selected as the* i*-th vertex and variables* x_v *encoding that vertex* v *has been selected at all. For each vertex, encode* $(x_v \leftrightarrow \bigvee_{i=1}^{\mathbf{k}} x_{i,v})$ *as clauses and encode that at most one vertex can be selected as the* i*-th vertex.*

- *automated planning*: Formalize Sliding Tiles as a parameterized STRIPS planning task. For example, the STRIPS representation of blocks-world in Chapter 34 of the lecture could be parameterized by defining a set of blocks B and introducing variables $\text{on}_{b,b'}$ for all $b, b' \in B$ with $b \neq b'$ and on-table_b, clear_b for all $b \in B$.

Exercise 13.2 (2 marks)
Analyze your problem formulation from Exercise 13.1.

- *state space search:* How many states does your state space contain? Is the state space cyclic, acyclic, a tree?
- *combinatorial optimization*: How many candidates does your formalization have? Is it a pure optimization problem, a pure search problem, or a combination of the two?
- *propositional logic*: How many assignments does φ have? How many clauses?
- *automated planning*: How many states does your STRIPS task encode? Is the state space cyclic, acyclic, a tree?

Exercise 13.3 (5 marks)
Implement your problem formalization, **solve** the problems contained in the archive on ADAM (`sliding-tiles-problems.zip` or `vertex-cover-problems.zip` respectively), and **report** your findings. You are free to solve the problems with any tool available to you (through the lecture, your exercise solutions, or the internet), but below we provide you with a suggestion on which tool you can use for each setting.
Hint: For most settings, it is advisable to set memory and time limits. On Ubuntu you can set limits with `ulimit`*. The following example sets a time limit of 300 seconds and a memory limit of roughly 2GiB:* `ulimit -t 300 -Sv 2000000`
Alongside your report, submit all information that is relevant for reproducing your results. Specifically, provide a short description of how you solved the problems and what limits you used, include any source code you wrote or altered, and include any files encoding the problem instances in specific formalisms like PDDL or CNF.

Suggestions

- *state space search*: Implement your state space in the framework you used for Exercises 2, 3 and 5. Your implementation should include the functions `buildFromCmdline`, which parses the given problem instances, and `h`, which returns a suitable admissible heuristic value. Then solve the provided problems with A* using suitable time and memory limits. Report runtime, number of expansions, solution cost, and for problems that were not solved whether they ran out of time or memory, or whether the search space was exhausted.

  Provide all `.java` files needed to run your implementation.
- *combinatorial optimization*: Implement your COP encoding in the framework you used for Exercise 6. You will need to alter the framework such that you can read in the given problem instances and build the COP from there. Also implement a suitable neighborhood and heuristic. Report for each problem the percentage of successful runs and the average number of steps, either for normal hill climbing or hill climbing with stagnation.

  Provide all `.java` files needed to run your implementation.
- *propositional logic*: Write a script that encodes the problem instances as CNF formulas in the DIMACS format (`http://www.satcompetition.org/2009/format-benchmarks2009.html`). Then solve these formulas with the SAT solver *MiniSat* (`http://minisat.se/`). Report runtime, memory usage, if the formula was satisfiable or unsatisfiable, and for problems that were not solved whether they ran out of time or memory.

  Provide your encoding script and the resulting DIMACS files.

- *automated planning*: Encode the given problem instances in PDDL. You can use the PDDL encoding of the 8-puzzle from the bonus material of Chapter 6 as a basis. Write a single domain file and a script that translates the given problem instances into PDDL task files. Then use Fast Downward (`https://www.fast-downward.org/`) with A* and an admissible heuristic. Report runtime, memory usage, number of expansions, solution cost, and for problems that were not solved whether they ran out of time or memory, or whether the search space was exhausted.

  For example, the following command line uses A* with the LM-cut heuristic, a time limit of 300 seconds and a memory limit of 2GiB:

  ```
  ./fast-downward.py --overall-time-limit 300 --overall-memory-limit 2G
  domain.pddl task.pddl --search "astar(lmcut())"
  ```

  Provide all PDDL files and the script used for translating the problem instances into PDDL task files.

Submission rules:

- Exercise sheets must be submitted in groups of two students. Please submit a single copy of the exercises per group (only one member of the group does the submission).
- Create a single PDF file (ending .pdf) for all non-programming exercises. Use a file name that does not contain any spaces or special characters other than the underscore "_". If you want to submit handwritten solutions, include their scans in the single PDF. Make sure it is in a reasonable resolution so that it is readable, but ensure at the same time that the PDF size is not astronomically large. Put the names of all group members on top of the first page. Either use page numbers on all pages or put your names on each page. Make sure your PDF has size A4 (fits the page size if printed on A4).
- For programming exercises, only create those code textfiles required by the exercise. Put your names in a comment on top of each file. Make sure your code compiles and test it. Code that does not compile or which we cannot successfully execute will not be graded.
- For the submission: if the exercise sheet does not include programming exercises, simply upload the single PDF. If the exercise sheet includes programming exercises, upload a ZIP file (ending .zip, .tar.gz or .tgz; *not* .rar or anything else) containing the single PDF and the code textfile(s) and nothing else. Do not use directories within the ZIP, i.e., zip the files directly.
- Do not upload several versions to ADAM, i.e., if you need to resubmit, use the same file name again so that the previous submission is overwritten.