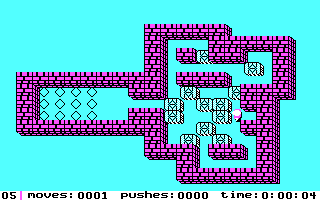
# REPORT FOR SEMINAR WORK IN ARTIFICAL INTELLIGENCE

Team:

* Jernej Jerin (63080038),
* Žiga Stopinšek (63090153)

## Environment

Implemented environment is **SOKOBAN** (<http://en.wikipedia.org/wiki/Sokoban>):



* 1. Description:
     + Is a type of transport puzzle, in which the player pushes boxes or crates around in a warehouse, trying to get them to storage locations.
     + The game is played on a board of squares, where each square is a floor or a wall. Some floor squares are marked as storage locations, and some of them have boxes.
     + The player is confined to the board, and may move horizontally or vertically onto empty squares (never through walls or boxes). The player can also move into a box, which pushes it into the square beyond. Boxes may not be pushed into other boxes or walls, and they cannot be pulled. The puzzle is solved when all boxes are at storage locations.
  2. Scientific research:
     + The problem of solving Sokoban puzzles was proven to be NP-hard.
     + This is also interesting for artificial intelligence researchers**, because solving Sokoban can be compared to designing a robot which moves boxes in a warehouse.**
  3. Variations:
     + In CyberBox, each level has a designated exit square, and the goal is to reach that exit.
     + Sokonex, Xsok, Cyberbox and Block-o-Mania all add new elements to the basic puzzle. Examples include holes, teleports, moving blocks and one-way passages.
  4. Placement within template:
     + Class Environment:
       1. \_\_init\_\_: The environment can be initialized either with graphical representation depicting stones, agent position, box positions and end state positions. The second option is to specify them with arrays of tuples. The tuples that represent 2D position are defined as x is column and y is a row.
       2. getStartingState: Returns starting position which is a tuple that contains:
          1. First tuple is position of the agent.
          2. Next tuples are positions of the boxes. There can be multiple tuples as we can have multiple boxes.
       3. do: In do after each move we check for new agent position as well as if the movement of the agent moves any of the boxes. If the box is only moved but does not reach any of the end positions the reward is 5. If the new position of the box is in one of the final positions the reward is 10. Otherwise if the movement of the agent does not result in movement of the box the reward is -5. We want the agent to finish the game as soon as possible. Also the environment check whether the move resulted in deadlock or finish position (all boxes in end position). The deadlock detection is simple and checks only for simple deadlocks such as corner. The reward for deadlock is -100, while the reward for successfully finished position is 100.
       4. getActions: We search for possible actions that an agent can execute given current position. Here we need to check if agent can hit a wall, if the box that the agent has possibility to move can really be moved.
  5. Literature:
     + [Reinforcement Learning Agents with Primary Knowledge](https://mailserver.di.unipi.it/ricerca/proceedings/AppliedComputing05/PDFs/papers/T01P04.pdf)
     + [Learning state features from policies to bias exploration in reinforcement learning](http://www.dtic.mil/cgi-bin/GetTRDoc?AD=ADA363533)
     + [Reinforcement learning based on local state feature learning and policy adjustment](http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.84.4434&rep=rep1&type=pdf)

## Code review

## Test of learning

Graph for scalability of the learning agent (different environments).