EXERCISE 3
Forward DP and other reformulations

Tue Herlau
tuhe@dtu.dk

18 February, 2022

Objective: The goal of this exercise is to continue working with the dynamical programming (DP) algorithm, with an emphasis of the forward-formulation suitable for deterministic problems. The forward-form is an important stepping stone towards deriving search methods you are familiar to from Algorithms and Data structures (not done in this course), but is also related to methods used in reinforcement learning and video-game AIs such as search trees. (18 lines of code)
Material: Obtain exercise material from our gitlab repository at
`https://gitlab.gbar.dtu.dk/02465material/02465students`

Contents

1 Implement forward-DP (`search_problem.py`, `dp_forward.py`)

For this problem, we will implement the shortest-path formulation of DP, specifically [Her21, algorithm 5]. A minor annoyance with the forward-DP algorithm is we still have to specify N, and it will then search for the optimal control sequence of exactly length N. For some problems (and graph-traversal in particular) this possibly not what we want, since there might exist a path of length $< N$ which is even better. We fix this in the second half of the problem by ensuring that the terminal state $t = 5$ is absorbing in the sense of [Her21, section 5.2.1].

Problem 1 *Search problem*

Inspect the code for `GraphSP` in the file `search_problem.py`. Make sure you understand what the `SearchProblem` class and how it applies to `GraphSP`. In what sense is this a more general way of stating a search problem?

Problem 2 *Forward-DP*

Implement the forward-DP shortest-path algorithm in `forward_dp.py` and check your implementation on the small graph problem. Note you get the wrong answer when using $N = 4$ (the maximal length of a path); understand why this happen, and how the problem is fixed by making the terminal state absorbing. When you are done, part A and B should run.

Info: The first part of the code should produce the following output

```
GraphSP> Optimal cost from 2 -> 5: 5.5
GraphSP> Optimal path from 2 -> 5: [2, 3, 2, 3, 4, 5]

GraphSP[Wrapped]> Optimal cost from 2 -> 5: 4.5
GraphSP[Wrapped]> Optimal path from 2 -> 5: [2, 3, 4, 5, 5, 5]
```

1.1 Regular DP problem to shortest path (`search_problem.py`)

An annoyance with shortest-path DP is that it right now only work with our new graph-based environments matching this signature:

```python
# search_problem.py
class SearchProblem:
    def __init__(self, initial_state=None):
        if initial_state is not None:
            self.set_initial_state(initial_state)

    def set_initial_state(self, state):
        """ Re-set the initial (start) state of the search problem. """
        self.initial_state = state

    def is_terminal(self, state):
        """ Return true if and only if state is the terminal state. """
        raise NotImplementedError("Implement a goal test")
```

```
14
15      def available_transitions(self, state):
16          """ return the available set of transitions in this state
17          as a dictionary {a: (s1, c), a2: (s2,c), ...}
18          where a is the action, s1 is the state we transition to when we take action
    ↪  'a' in state 'state', and
19          'c' is the cost we will obtain by that transition.
20          """
21          raise NotImplementedError("Transition function not impelmented")
```

This is a bit annoying, since most of the required functionality for computing the available transitions is already contained in the DP model class in the f_k/g_k format, which I think is easier to understand. For instance, the small graph problem was already implemented in `SmallGraphDP`.

This exercise will implement the *recipe* for converting a DP model specification into a shortest-path problem discussed [Her21, theorem 7.2.2]. The result is a class which takes any DP model (in this case, the `SmallGraphDP`), and return a shortest-path version matching the signature given above.

Problem 3 *DP to shortest-path formulation*

Implement the `DP2SP` class to complete sub problem C in the code. You should only implement `def available_transitions(self, state)` to return the required dictionary (see code for hints). Your implementation should match the description in [Her21, theorem 7.2.2] exactly.

Info: Once completed, you should get the (by now familiar) correct output:

```
1  DP2SP> Optimal cost from 2 -> 5: 4.5
2  DP2SP> Optimal path from 2 -> 5: [3, 4, 5, 5]
```

1.2 Benefits of forward-DP

What benefits does forward-DP really offer? After all, we started with a regular DP environment, then had to do all the search-problem formulation. So why not just use regular DP?

The answer is we avoid having to compute $\mathcal{S}_k$ prior to running the method. This might (on the surface!) seem like a really trivial thing (in the cases we have seen the $\mathcal{S}_k$ method have been really simple), but in other cases it turns out to be really good.

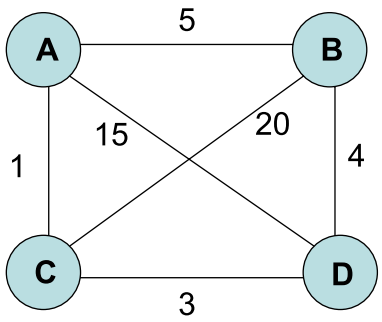


Figure 1: Traveling salesman problem

2 Traveling salesman (`travelman.py`)

Let's consider a case where $\mathcal{S}_k$ is difficult to define: The traveling salesman. In the traveling salesman problem we are given a graph (represented as we have previously seen) and the problem is, starting from a given city, to find a route which visits every city exactly once and ends up where it started. See [Her21, section 7.1.1] for more information.

- In this problem, each state k is a $k+1$-length path of vertices's. In other words, since we start in A the first state is $s = x_0 = (A,)$, the next could be $x_1 = (A, B)$ and so on
- Actions are nodes we can travel to
- Cost is the distance between two vertices
- Terminal cost is more tricky. You have to check if the path actually represents a traveling salesman path; see code for detail. If it does, return 0 (we are done!) otherwise ∞

If you consider this problem and how we would implement it in the non-forward setting, then for instance $\mathcal{S}_3$ would consist of all 4-vertex sequences, and S_N of all possible $N+1$-long paths (if we build a wasteful representation) or alternatively all $N+1$-long paths that satisfied the traveling salesman requirement. In other words, not only would these sets be incredibly large; building them would involve solving the problem. For instance, given S_N we could just apply the cost function g_N to all elements and compute which were a solution. See also the next question about 8-queens for an instance where this approach will literally take hours.

Problem 4 *Traveling salesman*

Complete the traveling salesman environment and verify the solution starting in $s = (A,)$

Info: Note that the optimal policy is a sequence of $k+1$-long partial paths. In other words, it is the last element which actually represents the full path. The code should produce the output:

```
Cost of optimal path (should be 13): 13
Optimal path: [(('A',), 0), (('A', 'B'), 1), (('A', 'B', 'D'), 2), (('A', 'B', 'D', 'C'), 3), (('A', 'B', 'D', 'C', 'A'), 4), 'terminal_state']
(Should agree with (Her21, Subsection 7.1.1))
```

Carefully inspect the output; the integers corresponds to k and arises from `DP2SP` implemented earlier.

3 Forward-DP and the N-queens problem (`queens.py`)★

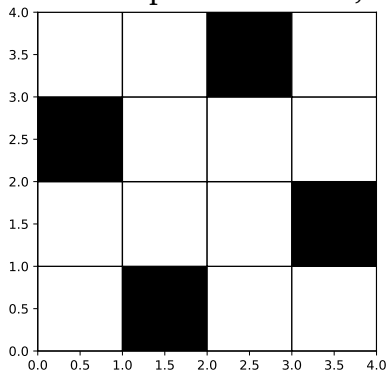
The four-queens problem is the problem of placing $N = 4$ queens on a 4×4 chessboard so no two queens attack each other. This might not seem like a DP problem, but we can consider the queens as being labeled $k = 0, 1, 2, 3$ so that we first place queen 0, then queen 1 and so on. In this way, it is a sequential decision problem, and we are interested in the final cost which is infinite if two queens can attack each other and 0 otherwise.

- Starting position is the empty board
- Each action u_k is a tuple (i, j) of where to put queen k
- Each x_k is a list of length k of tuples. For instance: `((1,3), (0,1))` is a valid position
- Cost of each transition is 0; terminal cost is 0 if position is ok and otherwise ∞

Problem 5 *4-queens*

Implement the four-queens problem and check that the final position is correct. I recommend using the supplied function which checks if a position is still valid after adding an extra queen when choosing actions.

Info: The output is included, and should be



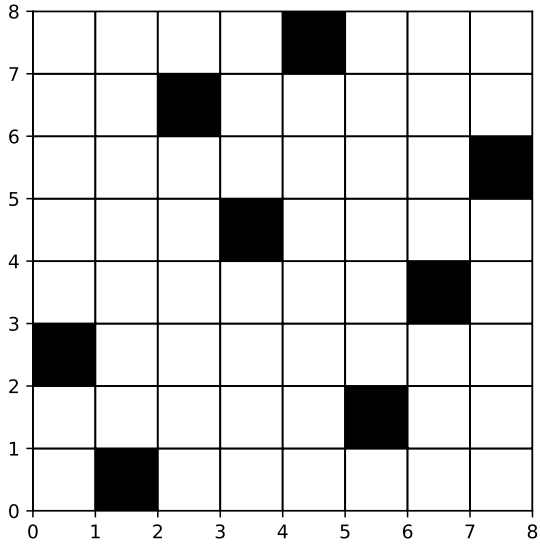
3.1 Getting ambitious ★★

If you are feeling ambitious, you can try increasing N. Note this problem, if poorly implemented, can take a long time, and speeding it up reasonably is mainly about choosing actions smartly.

Problem 6 *8-queens*

Do the eight-queens problem and visually check that the final position is valid.

Info: This is not about looking at the screen for 2 hours (although it is easy to make an implementation which does that or worse!), but rather about smartly implementing the environment. I have included a benchmark for my implementation.



4 Pacman and search (`pacman_search.py`)★

Note: One of the problems in the report might be solvable using the methods in this exercise; if that is possible it is fine, but it is not actually the recommended

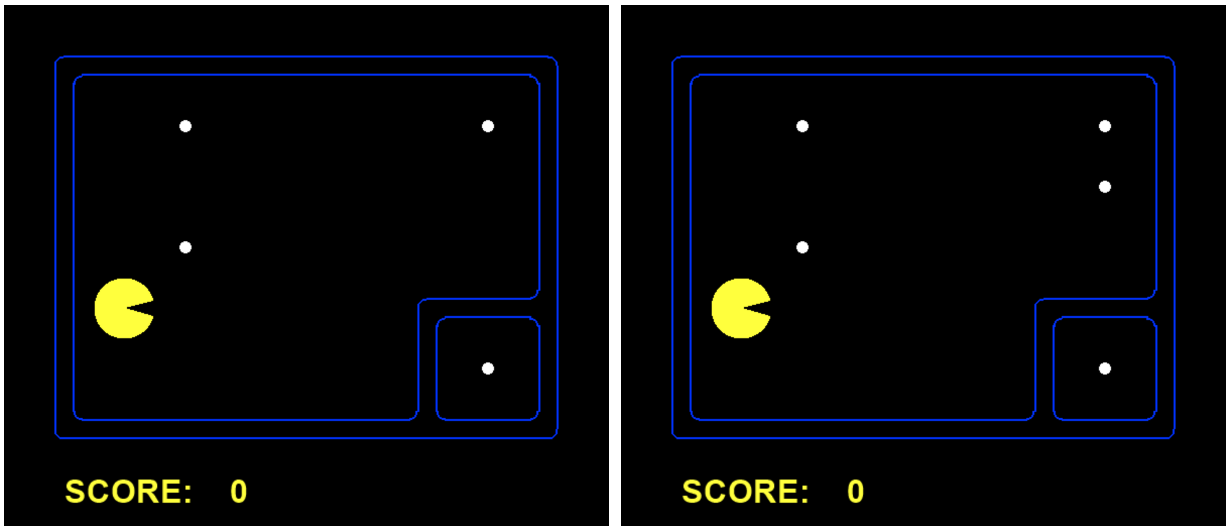


Figure 2: Layout 1 and Layout 2 used in the pacman search agent problem

approach considering the problem is a warm-up to the slightly more tricky problems involving multiple ghosts where the deterministic search-formulation will not work. The problem can still be of help to understand the pacman-environment. We will finish this part of the course by returning to the Pacman-environment. We will consider the following two layouts shown in fig. 2. We will turn this into a deterministic DP problem by applying the following rules:

- Each move has a cost of 1
- Each captured food pellet has a cost of -10
- The goal for pacman is to find his way to the lower-left corner (square $(1, 1)$) while incurring as little cost as possible

Problem 7 *Discussion*

Explain why this problem is not suitable for search algorithms such as Djikstras algorithm you may be familiar with from Algorithms and data structures. Next, Pacmans behavior differ in the two problems. Can you explain why?

We are going to solve the problem using forward-DP. This will be the first example of model-based planning similar to control:

- Agent plan using a model
- The model is not an exact model of the environment (in this case, it does not simply correspond to eating all the dots)

The model we will use will be the `SearchProblem`. Since the DP search method requires us to search for exactly N steps we will make the terminal state absorbing as shown here (see section 5.2.1):

```
# pacman_search.py
    env1 = GymPacmanEnvironment(layout_str=layout1) # Create environment
    problem1 = PacmanSearchProblem(env1)            # Transform into a search problem
    problem1 = EnsureTerminalSelfTransitionsWrapper(problem1) # This will make the
    ↪  terminal states absorbing as described in (Her21, Subsection 5.2.1)
```

Problem 8 *Pacman as a search problem*

Complete the search problem class to compute the transitions. You should use the functions on the state (insert a breakpoint and read the comments). Once done, the forward DP algorithm should take care of the rest.

Info: The cost of the two paths is as follows:

```
Optimal cost in layout 1 -11.0
Optimal cost in layout 2 -21.0
```

Next, all we need to do is build a pacman model class which plan according to the actions the forward DP algorithm find. When done, we should be able to let it interact with the environment like any other agent, and thereby make a small animation of it's behavior. Discuss if it agrees with your conclusion from the first question.

```
# pacman_search.py
    agent1 = ForwardDPSearchAgent(env1, N=N)
    train(env1, agent1, max_steps=N)
```

Problem 9 *A search agent*

Complete the code for the Pacman search agent.

Info: This should be quite simple. The if/else statement and the special stop action is necessary but tedious; can you explain why this is needed?

References

[Her21] Tue Herlau. Sequential decision making. (See **02465_Notes.pdf**), 2021.