Get Me Out of Here: Determining Optimal Policies

# Introduction

Imagine being an employee at a funfair and your task is to help people that are stuck in a labyrinth find the fastest way to the nearest exit.

* Go in and then look at map and search for shortest path each time
* Or search for knots that often occur in the shortest paths and from them you already know the next steps because you have walked the way often, and walk to them and follow the best way
* And now imagine a map, in which you have already depicted the shortest path for any position in the labyrinth, so you always know what the best action to any given position is
* You cannot argue that such a map would be much better than always recalculating a new shortest path every time someone gets stuck.
* But how does one calculate such a map with the best actions for any position in the labyrinth.
* That is where dynamic programming comes into account.

# Was ist Dynamic Programming?

## A short example

* Take a sheet of paper and write down 1+1+1+1+1+1+1+1 =
* Now calculate the result. You will see you will count all the single 1s and come to a result of 8.
* Now write down another 1+ on the left and calculate the new result.
* This time you will directly tell that the result is 9. But why? Because just 1 was added and you didn’t need to recount because you remembered there were eight!
* Dynamic Programming is just a fancy way to say remembering stuff to save time later.

## Some mathematical stuff

* Mathematical optimization method & computer programming method
* Simplifying a complicated problem by breaking down into simpler subproblems recursively
* In computer science, problem that solved by breaking into subproblems & recursively finding optimal solution to subproblems has optimal substructure

## In computer programming

* Two key attributes a problem must have for DP
* Optimal substructure and overlapping subproblems

### Optimal substructure

* Solution to a given problem from
* Combination of optimal solutions to its subproblems
* Remember the labyrinth for example.
* For any given position p in the labyrinth a shortest path s around the walls to the nearest exit e can be calculated.
* If s is really the shortest path, can be split into subpaths s\_1 from p to x and s\_2 from x to e such that these are indeed the shortest paths between the three positions
* From that, a recursive way for finding shortest paths can easily be formulated, which is what the Bellman-Ford algorithm does

### Overlapping subproblems

* Means, that space of sub-problems is small
* An algorithm solves the same sub-problem over and over again.
* Again, remember the labyrinth.
* If there is only one exit and it lays at the end of a long and narrow passage.
* If you calculate the shortest paths, you will always end up walking down that narrow passage
* So, if you calculate the paths you will end up calculating the way down this passage over and over again, rather than generating new subproblems

# Wie funktioniert Dynamic Programming?

* Solve problem by breaking down into smaller subproblems
* Results of previous calculated subproblems is stored -> Memoization
* When a subproblem is reoccurring, the memorized solution is looked up
* This saves computation time, but expands memory usage (hopefully negligible)
* DP algorithms often used for optimization
* Will examine previously solved subproblems & combines solution for optimal results

# Welche Algorithmen verwenden Dynamic Programming?

* Instead of solving the subproblems over and over again, dynamic programming takes into account, that there are these overlapping subproblems and the optimal substructure
* Two ways:

### Top down approach

### Top-down approach: This is the direct fall-out of the recursive formulation of any problem. If the solution to any problem can be formulated recursively using the solution to its sub-problems, and if its sub-problems are overlapping, then one can easily memoize or store the solutions to the sub-problems in a table. Whenever we attempt to solve a new sub-problem, we first check the table to see if it is already solved. If a solution has been recorded, we can use it directly, otherwise we solve the sub-problem and add its solution to the table.

### Bottom up approach

* Bottom-up approach: Once we formulate the solution to a problem recursively as in terms of its sub-problems, we can try reformulating the problem in a bottom-up fashion: try solving the sub-problems first and use their solutions to build-on and arrive at solutions to bigger sub-problems. This is also usually done in a tabular form by iteratively generating solutions to bigger and bigger sub-problems by using the solutions to small sub-problems. For example, if we already know the values of F41 and F40, we can directly calculate the value of F42.

# Lösungsstrategien für DP

## Memoization

* Optimization technique to speed up programs
* Stores results of expensive calculations and returning cached results when same calculation should be done again
* A memoized function rememberes the result to specific input and returns the remembered value rather then recalulating if the same inputs are given again
* With that, a functions time cost can be lowered dramatically
* Space cost will increase
* Time-space trade-off for optimizing an programs speed

## Bellman’s principle of optimality

* *An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.* Bellman 1954

### Bellman equation

* Named after Richard Bellmann
* Also known as dynamic programming equation
* Necessary condition for optimallity assoc with mathematical optimization in dp

# Finden von Literatur welche Dynamic Programming für Motion Planning von Robotern verwendet

# Vergleich von DP zu klassischen Planungsalgorithmen

# Conclusion