Fakultät für Informatik
Lehrstuhl für Echtzeitsysteme und Robotik

Get Me Out Of Here: Determing Optimal Policies

Marcel Bruckner

Seminar *Cyber-Physical Systems* WS 2017/18

Advisor:	Christian Pek
Supervisor:	Prof. Dr.-Ing. Matthias Althoff
Submission:	07. February 2018

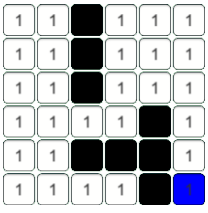
Get Me Out Of Here: Determining Optimal Policies

Marcel Bruckner
Technische Universität München
Email: mbruckner94@gmail.com

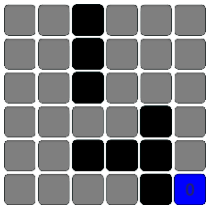
***Abstract*—This paper gives a brief overview on the field of dynamic programming and it's applicability in robot motion planning. Exemplary, some use cases in other papers will be shown and an algorithm to find optimal policies in a maze will be introduced.**

I. Introduction

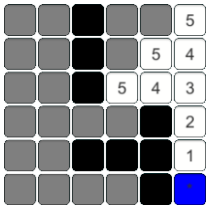
Remember the game Labyrinth, in which the game represents a maze where the field is build up by fixed and moving pieces showing walls and passages. The goal of the game is to rearrange the maze by moving rows of not fixed tiles and enable ones game character to find treasures and return them to goal tiles.
To reach the goal in this game, a way for calculating the shortest path to exit the maze has to be found. This can be achieved with classical motion planning algorithms, like Dijkstras Algorithm or Breadth-First Search, but also with **Dynamic Programming**, which will be used in this Paper.
Therefore, let a grid map $M = \mathbb{N}^{n \times m}$ containing static walls $M(i, j) = \infty$ represent the maze and let there be a set of exits X_{goal} in the labyrinth which the player wants to reach.
The goal of this paper is to point out ways to determine optimal policies $P = \mathbb{N}^{nxm} \rightarrow A$ to exit the labyrinth. To achieve this, a way to calculate the value $v(i, j)$ of the current position is introduced from what the actual best action can be derived.
Later on, the algorithm will look like in the following example:



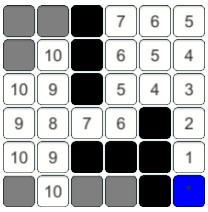
The original grid showing the cost of each tile, walls (black) and goals (blue).



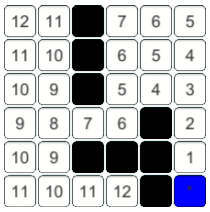
Initialization of the algorithm with ∞ cost. Only the goal (blue) is set and is the starting point. It has a value of 0.



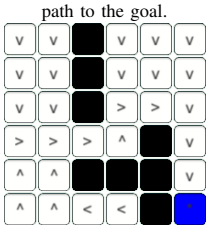
After a few iterations the values of some tiles are set. The value is equal to the length of the shortest path to the goal.



A few more iterations. The algorithm is reaching tiles further away from the goal.



All values are calculated. Every tile is holding its value.



The final optimal policy showing the best action for every position derived from the values of the tiles.

II. Fundamentals

A. What is Dynamic Programming

Dynamic Programming is a method used in mathematical optimization and computer programming to transform complex problems into sequences of simpler subproblems. Each of those subproblems will be solved just once and their solutions will be stored, so when reoccuring, the previously computed solution will be looked up to save computation time.
For solving a problem with Dynamic Programming, the problem gets broken down and by recursively finding the optimal solution for the subproblems a optimal solution for the whole problem can be found.
It provides a general framework for analyzing many problem types, like planning of industrial production lines, scheduling patients at a medical clinic, long term investment programs and, of course, robot motion planning.
Underlying of these problems is always a sequence of decisions that have to be made within a finite number of stages. E.g. a robot wants to move from one position to another by a finite number of steps and investment programs do have to calculate their wealth at every step in time.
A sequential decision making problem with finite stages can be defined with the following statements:

- $\mathbf{X}$ is the set of all possible states of the system;
- $\mathbf{U}$ is the set of controls or also called transformations;
- $\mathbf{f} : X \times U \mapsto X$ is a transition function, a mapping specifying the next state $f(x, u)$ if control u is executed when the system is in state x;
- $\mathbf{c} : X \times U \mapsto \mathbb{R}$ is a cost function that gives a real value $c(x, u)$ obtained if control u is executed when the system is in the state x;
- $\mathbf{v} : X \mapsto \mathbb{R}$ is a value function that gives a real value $v(x)$ associated with the state x;
- $\mathbf{x}_{\text{init}} \in X$ is the initial state of the system. [1]

B. Richard Bellman

Dynamic Programming was introduced by the american mathematician Richard Bellman who was working on multistage decision problems from 1949 until 1955. After he received his Ph.D. from Princeton at the age of 25, he started his research at RAND Corporation (Research and Development), where his secretary really had a hatred of the term *mathematical research*. So he started his work with finding a name under which he could hide his projects.
He was interested in planning, decision making and thinking, but none of these were good terms, so he decided on the word

programming, and because of his love for physics and his ideas in multistage and time varying problems, he came to the conclusion that dynamic is the best adjective to accompany it. Following the name Dynamic Programming was born and no one could imagine what he was doing which made it perfect for hiding his mathematical research. [2]

C. The Principle of Optimality

For starting a more precise discussion, lets define the term optimal policy:

- A **policy** is a sequence of decisions.
- An **optimal policy** is a sequence of decisions that has the best outcome w.r.t. a predefined criterion.

When being asked the task for finding the optimal policy for a given problem, the classical approach would be to compute all possible policies and maximize the return to determine which of these is the optimal one. In practice, this is often not practical due to the fact, that even for a low number of stages and choices the dimension of the resulting maximization problem will explode.
But Bellman saw that the knowledge of the complete sequence of decisions isn't even required to solve these multistage decision problems. He stated that it would be much better to find a general prescription which gives the best decision at any time and depending only on the current state of the system.

"If at any particular time we know what to do, it is never necessary to know the decisions required at subsequent times."

[3]

1) The fundamental approach:

From the previous stated point of view, Bellman went over to define an optimal policy and described it as one "determining the decision required at each time in terms of the current state of the system. Following this line of thought, "[3] he was able to formulate his

Principle of Optimality

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

and thereof derive the fundamental mathematical equations on which Dynamic Programming is based on. [3]

2) Mathematical formulation:

But what does this mean for the functional equations needed for actual problem solving based on the principle of optimality?
Imagine the simplest case, where a process of a system is described at any time by an M-dimensional vector

$$p = (p_1, p_2, ..., p_M) \in X \tag{1}$$

Let $U = \{T_k\}$ be a set of transformations with the property

$$p \in X \Rightarrow T_k(p) \in X, \quad \forall k \tag{2}$$

Now thinking of an N-stage process, the goal is to maximize a scalar function $R(p)$ giving the return or the value of the final state. This function will be called the N-stage return and a policy consists of a selection of N transformations

$$P = (T_1, T_2, ..., T_N) \tag{3}$$

yielding successively the states

$$\begin{aligned} p_1 &= T_1(p), \\ p_2 &= T_2(p_1), \\ &\vdots \\ p_N &= T_N(p_{N-1}) \end{aligned} \tag{4}$$

As seen above, the maximum value of the return function $R(p_N)$, regulated by the optimal policy, will only depend on the initial vector p and the number of stages N.
One can now give a direct formula for the N-stage return that we get only using the optimal policy and by starting from the initial state p as

$$h_N(p) = \max_P R(p_N) \tag{5}$$

To get a functional equation for $h_N(p)$, the principle of of optimality is used. By making our first decision, some transformation T_k will be made and we come to a new state $T_k(p)$. From the new state $T_k(p)$, there are now left $(N-1)$ stages and the maximum return of these is, as defined above, $h_{N-1}(T_k(p))$.
As you might see, k and thereof T_k have to be chosen so as to maximize the return, which finally results in the basic functional equation

$$h_N(p) = \max_k h_{N-1}(T_k(p)), \quad N = 2, 3, ... \tag{6}$$

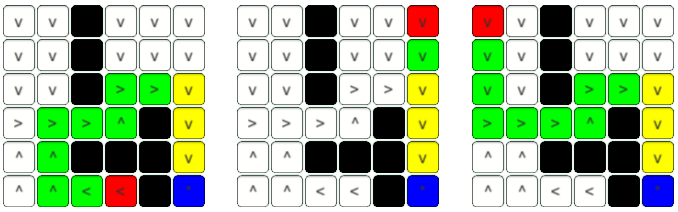
Resulting from this equation it is now clear, that any optimal policy does not have to be unique, but will yield $h_N(p)$. On the other hand, if looking at the sequence $\{h_N(p)\}$, one can reversely determine all optimal policies. [3]

3) Which characteristics does a problem need to be solvable by Dynamic Programming:

a) Overlapping subproblems:

A problem is said to have overlapping subproblems if the space of subproblems is small and thus, if broken down, solutions for these are reused many times or a recursive algorithm solves the same subproblems over and over, insted of generating new subproblems. [4]
Remember the game Labyrinth, where the goal was to find treasures and return to a goal tile.
If there is a treasure you want to discover and it lays down a narrow passage, any algorithm calculating the shortest path from any arbitrary position in the maze will guide you through the passage.
So if you would calculate the shortest path from your current position and move one tile, you'd have to recalculate the path

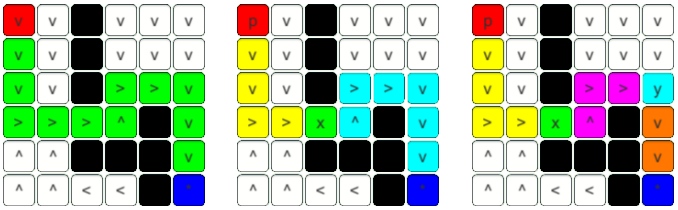
again, which leads you to also recalculate the way down the passage.
So for any position, the passage would be a subproblem that overlaps in every calculation.



An example of overlapping subproblems. In all three cases the yellow passage will be recalculated no matter from where you started (red).

b) Optimal substructure:

A problem has optimal substructure if the solution for the whole problem can be found by a combination of solutions for its subproblems. Normally, greedy algorithms are used to solve problems with this property because they lead to an optimal overall solution in this case. [4]
But also the principle of optimality is based in this idea.
To come back to the labyrinth, for any given position p in the labyrinth a shortest path s avoiding the walls and reaching a goal can be calculated. If s is really the shortest path, it can be split into subpaths s_1 from p to an point x on the shortest path and s_2 from x to the goal, such that s_1, s_2 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 is applied in Dynamic Programming.



The overall shortest path s (green) from start (red) to goal (blue).

A first split of s into s_1 (yellow) and s_2 (cyan).

A second split of s_2 into $s_{2,1}$ (magenta) and $s_{2,2}$ (orange).

An example of optimal substructure. The shortest path s (green) can be split down into subproblems recursively until only the shortest path between two neighbored tiles has to be calculated. Based on these optimal solutions for the subproblems, the whole problem can be solved.

4) How to apply Dynamic Programming in computer science:

a) Memoization:

An important concept when applying Dynamic Programming in computer science is memoization. As a result of the overlapping subproblems a problem can be broken down into simpler subproblems and for each of these subproblems, a solution can be calculated and the result can be stored in a table, so that a solution has to be found only once. When a subproblem reoccurs, the cached result can be returned to save computation time. [4]
On the other hand the storage consumption will expand due to the need for storing the results. But in times of rapidly increasing memory sizes and shrinking costs this draw off can often be considered as negligible.

b) Bellman equation:

The Bellman equation, also often referred to as the Dynamic Programming equation, is a necessary condition for optimality. It states a relationship between the value of a decision problem at a certain point in time depending on the outcome of the initial choices and the value of the remaining decisions that result from the previous choices. This breaks the problem down into subproblems that can be calculated seperately, as Bellman's principle of optimality suggests.
Before coming to the actual Bellman equation, some underlying concepts have to be defined. At first, optimizing a problem is always under a certain objective: maximizing utility, minimizing path length, maximizing profits. The function describing these objectives is called the **objective function**.
Second, Dynamic Programming needs to keep track of the **state** of the system when evolving over time, because when breaking down a multi-period planning problem into simpler steps at different points in time, in every step the information about the current situation that is needed has to be available. Furthermore, the next state is always affected by some factors in addition to the current **control**. That means, that choosing the control variables may be equivalent to choosing the action to reach the next state. This, in fact means making a decision. In dynamic programming, an optimal plan can now be described by finding a rule that makes an interconnection between what the controls should be when given any possible value of the state. To achieve the best possible value of the objective, the optimal decision rule has to be found.
Finally, by definition, the optimal decision rule is the one that achieves the best possible value of the objective. When written as a function of the state, the best possible value of the objective is called the **value function**.
Richard Bellman showed, that when calculating a relation between the value function of one step and the value function of the next step, a dynamic optimization problem can be solved in a recursive manner. The relationship connecting these two value functions is called the **Bellman equation** and uses an approach called **backwards induction**.
In this approach, the optimal policy of the last time period t_N is specified at first as a function depending on the value of this state and the optimal value of the objective function. Next, the previous step in time t_{N-1} is calculated by optimizing the current (t_{N-1}) objective function and the optimal value for the future (t_N) objective function. This can be applied recursively until the initial state (t = 0) is reached and the first period decision rule is derived, as a function of the initial state and the value, by optimizing the sum of the objective function at the initial time step t_0 and the value of the second timestep t_1, which is dependent on all future decisions. Resulting from that, each periods decision is made by explicitly acknowledging that all future decisions will be optimally made.

So, when used on a dynamic decision problem the function describing the previously stated thoughts is

$$V(x_0) = \max_{a_0}\{F(x_0, a_0) + V(x_1)\} \tag{7}$$

whereas x_t is the state of the system at time t, beginning at $t = 0, x_0$ with $a_0 \in \Gamma(x_0)$ being the action depending on the set of possible actions Γ in state x_0. The transition from one state x to another, is described by $T(x, a)$, when action a is chosen at state x, and the payoff of taking a in state x is $F(x, a)$. The Bellman equation can be simplified even more when dropping the time dependency resulting in a general functional equation

$$V(x) = \max_{a\in\Gamma(x)} \{F(x, a) + V(T(x, a))\} \tag{8}$$

Solving this functional equation means now finding the unknown function V, which is the value function of the Problem. [5]

III. Finding the value function and implementing the maze

A. Finding the value function

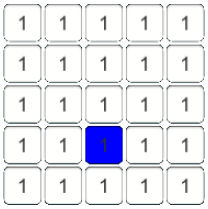
Remember the initial task where you are given a grid map $M = \mathbb{N}^{nxm}$ containing static walls $M(i, j) = \infty$ representing the maze, and exit(s) of the labyrinth which the player wants to reach.
The goal of this paper is to point out ways to determine optimal policies $P = \mathbb{N}^{nxm} \to A$ to exit the labyrinth by using Dynamic Programming. To achieve this, a way to calculate the value $v(i, j)$ of the current position is searched from what the actual best action can be derived.
To do so, we have to solve the Bellman equation for the task of finding all pair shortest paths in the maze.
This means, that for any arbitrary point in the maze a value function $v(i, j) \in \mathbb{N}$ representing the length of the shortest path of the position (i, j) to the nearest goal in the maze has to be found.
Therefore let $c(i, j) \in \mathbb{R}$ be the cost it would take a prisoner in the maze to step on the field (i, j).
The cost function is defined as

$$c(i, j) = \begin{cases} \infty & \text{if} (i, j) \notin M \\ M(i, j) & \text{else} \end{cases} \tag{9}$$

This describes that if we are looking at a position outside of the maze, it would need an infinite cost to reach it and if we are in the maze we just retrieve the value of the grip map.
Let us now begin with the trivial cases. If the player is already standing on a goal, its value would be 0, because the length of the shortest path from a position to itself is 0. In addition, if the player isn't even in the maze, he will never be able to reach a goal, so the shortest path would be infinite long, resulting in

$$v(i, j) = \begin{cases} \infty & \text{if } c(i, j) = \infty \\ 0 & \text{if } (i, j) \in X_{\text{goal}} \end{cases} \tag{10}$$

Now to the non trivial cases.
Therefore let us have a look at a maze looking like



All tiles are having a cost of 1.

Using the approach of backwards induction introduced by Richard Bellman we can start from the goal state in the grid and calculate the value at the step $N : v(3, 4) = 0$. Now having the result for N, we can induce backwards the results for the step $(N - 1)$, whereas their results are

$$v(3, 3) = v(3, 4) + c(3, 4) = 0 + 1 = 1 \tag{11}$$
$$v(4, 4) = v(3, 4) + c(3, 4) = 0 + 1 = 1 \tag{12}$$
$$v(3, 5) = v(3, 4) + c(3, 4) = 0 + 1 = 1 \tag{13}$$
$$v(2, 4) = v(3, 4) + c(3, 4) = 0 + 1 = 1 \tag{14}$$

When exemplary applying this procedure at the position $v(2, 4)$, we can again give the equations for the positions lying around it

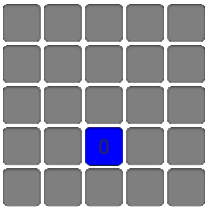
$$v(2, 3) = v(2, 4) + c(2, 4) = 1 + 1 = 2 \tag{15}$$
$$v(3, 4) = v(2, 4) + c(2, 4) = 1 + 1 = 2 \tag{16}$$
$$v(2, 5) = v(2, 4) + c(2, 4) = 1 + 1 = 2 \tag{17}$$
$$v(1, 4) = v(2, 4) + c(2, 4) = 1 + 1 = 2 \tag{18}$$

Now we have calculated the values of the step $(N - 2)$ depending on the decisions that will be made optimally in $(N-1)$ and N. The results are getting memoized in a second matrix $V = \mathbb{N}^{n\times m}$ which is initialized with the value ∞ at all positions. So $V(3, 4) = 0, V(2, 4) = 1 \ldots$
One thing that has to be considered at this step is, that the value for the tile $v(3, 4)$ is getting calculated again, which might cause troubles due to the fact that now the value of this position would be

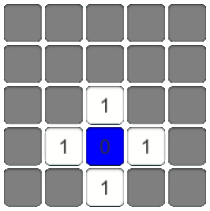
$$v(3, 4) = v(2, 4) + c(2, 4) = (v(3, 4) + c(3, 4)) + c(2, 4) = 2 \tag{19}$$

which is in fact bigger than the initially calculated value of 0.
At this point we have to take into account if there is already a result of the value function memoized in the matrix V. If there is a result, and the value is less than the one currently calculated, the result can be discarded.
Following this procedure, a general relation between the steps can be stated:

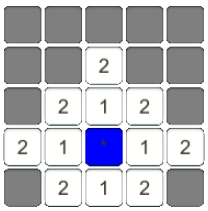
$$v(i-1, j) = v(i, j) + c(i, j) \quad \text{if } v(i-1, j) < V(i, j) \tag{20}$$
$$v(i+1, j) = v(i, j) + c(i, j) \quad \text{if } v(i+1, j) < V(i, j) \tag{21}$$
$$v(i, j-1) = v(i, j) + c(i, j) \quad \text{if } v(i, j-1) < V(i, j) \tag{22}$$
$$v(i, j+1) = v(i, j) + c(i, j) \quad \text{if } v(i, j+1) < V(i, j) \tag{23}$$



Only the value of the goal position is known at step N.



At step $(N-1)$ the values of the four adjacent positions are calculated.



At step $(N-2)$ the values of the positions adjacent to the previous positions are calculated.

Step $(N-3)$

Step $(N-4)$

At step $(N-5)$ the values of all the positions are calculated.

With these equations, the value of every position in the maze can be calculated.

When now solving the Bellman equation, a way to formulate the previous stated algorithm recursively has to be found.

In the backwards iterated method we started at the goal tile (N) and derived the value of the positions adjacent to it $(N-1)$.

On the opposite, we could have also started at the adjacent positions and look at the positions lying around them. For every position we want to find the position next to it, which holds the lowest sum of its value and its cost.

So for example when looking at the tile right of the goal state the four adjacent tiles would be observed. They would all but one have a value that is not yet calculated so we can't use these tiles to get a proper result. But the one tile that already has a value calculated is the goal tile lying left of this position. It's value would be 0 as defined and it's cost would be 1. So the value of the right adjacent tile would be 1.

For the position on the top of the right tile, we could use the same method. Look at the four adjacent positions, find that some of the values aren't calculated yet and thereof can not be used but that there is one tile that is already calculated. So we could again add up the value and the cost to find the current value.

But what to do when coming to a tile that has more than one value next to it already calculated? Due to the fact that we want to find optimal policies and shortest paths, we want to minimize the value function. This can be achieved by deciding for the adjacent tile with the lowest value for itself. When now adding up it's value and it's cost, one can be sure to have calculated the lowest value for the tile.

Coming from this thought, it leads to the function solving the functional Bellman equation:

$$v(i,j) = \begin{cases} \infty & \text{if } c(i,j) = \infty \\ 0 & \text{if } (i,j) \in X_{\text{goals}} \\ \min_{(x,y)}\{v(x,y) + c(x,y)\} & \text{else} \end{cases} \tag{24}$$

With $(x,y) \in \{(i+1,j),(i-1,j),(i,j+1),(i,j-1)\}$

B. *Finding optimal policies*

With the value of every tile calculated and stored in the matrix V, it is now trivial to find the optimal policy for any position.

Consider being at any position x in the maze. All you can do is move into one of the four directions right, down, left and up.

Due to the fact that we are searching for the shortest path to the nearest goal, and the value of every tile is known, we can now make our decision where to step depending on the values of the tiles adjacent to the current position. By always choosing the tile with the lowest value, the best action $a \in A$ can be chosen accordingly.

Defined in a mathematical sense, the equation describing the value of every tile is the equation

$$a(i,j) = \begin{cases} \text{up} & \text{if } v(i,j-1) = \min_{(x,y)}\{v(x,y)\} \\ \text{right} & \text{if } v(i+1,j) = \min_{(x,y)}\{v(x,y)\} \\ \text{down} & \text{if } v(i,j+1) = \min_{(x,y)}\{v(x,y)\} \\ \text{left} & \text{if } v(i-1,j) = \min_{(x,y)}\{v(x,y)\} \end{cases} \tag{25}$$

With $(x,y) \in \{(i+1,j),(i-1,j),(i,j+1),(i,j-1)\}$

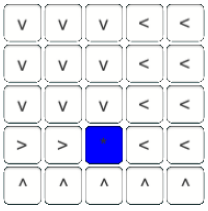
Starting from the red tile an optimal policy will be found.

So looking at the adjacent tiles the green one is one of the tiles with the lowest value.

Following equation 25 a path is getting found.

The path is reaching closer to the goal.

The path has reached the goal and the algorithm terminates returning one shortest path from red to blue.



When 25 is applied to every tile, the best action at every position can be given.

IV. Related Work

A. Autonomous vehicles

As cars being just one special type of robots, finding the navigation paths has been a field with many research dedicated to. They have to go straight, take turns, switch lanes and always avoid collisions with pedestrians or other cars.
Classic algorithms like Dijkstras algorithm or based on it A* are a very prominent example of algorithms solving these tasks. But their disadvantage is often the fact that they only solve the shortest path problem for the current location to the nearest goal.
The huge advantage Dynamic Programming brings to the field of autonomous driving is that with the calculation of optimal policies a best action for every position on the map can be calculated previously, so that even if there is an unexpected obstacle blocking a lane shift or a turn and resulting in being in a position not in the optimal shortest path, a alternative solution is already calculated.
But also for smaller autonomous robots like vacuum robots dynamic programming is a nice way to navigate. A map of the room has to be given or found by the little helper based on which it can then define solid obstacles like tables or cupboards, whereas goals can be defined as dirt hotspots or the loading station for the robot.

B. Extending the 2-dimensional algorithms onto N-dimensional problems

But how can the Dynamic Programming approach be used for robot motion planning with more than 2 dimensions? Imagine a two-armed robot having six degrees of freedom for every arm resulting in a 12-dimensional motion planning problem.
For every possible configuration of the arms a node in a graph can be defined holding the specific attributes of that configuration. For every transition between two configurations an edge between these two can be drawn describing uniquely the transformation between the two states.
With this approach every N-dimensional problem can be projected down onto a two-dimensional plane where the algorithms can easily be applied.
Based on this method many robot motion planning algorithms have been introduced.

C. Algorithmic Design of Feasible Trajectories

Written by Steven M. LaValle, this paper summarizes the recent development of algorithms that construct feasible trajectories made at the University of Illinois. They use et al dynamic programming approaches that produce approximately-optimal solutions for low-dimensional problems.
The difficulties they had to come over are on the one side differential constraints, which means that in a systems the number of actuators is strictly less than the dimension of the configuration space. For example imagine a spacecraft floating in $\mathbb{R}^3$ that has three thrusters. These are called the actuators of the spacecraft system. Opposite, the spacecraft has six degrees of freedom (X-Pos., Y-Pos., Z-Pos., Yaw, Pitch, Roll) which makes it an underactuated system having differential constraints.
On the other hand they had to deal with global constraints arising from robot collisions.

1) Dynamic Programming:

Using Bellman's principle of optimality, let k refer to a *stage* or *time step*, and let x_k and u_k refer to the state and input at stage k. For their purpose $K+1$ represented the final stage.
The cost function used to describe this problem is of the form

$$L(x_1, \cdots, x_{K+1}, u_1, \cdots, u_K) = \sum_{k=1}^{K} l(x_k, u_k) + l_{K+1}(x_{K+1}) \tag{26}$$

whereas $l_{K+1}(x_{K+1}) = 0$ if $x_{K+1} \in X_{\text{goal}}$, and $l_{K+1}(x_{K+1}) = \infty$ else.
They used classical Cost-to-go Iterations that can solve this problem numerically, whereas the cost-to-go function they used is

$$L_k^*(x_k) = \min_{u_k, \cdots, u_K} \left\{ \sum_{i=k}^{K} l(x_i, u_i) + l_{K+1}(x_{K+1}) \right\}. \tag{27}$$

This now induces the calculation of the cost-to-go functions iteratively from stage $K+1$ to 1. In each iteration, L_k^* is computed using the result of L_{K+1}^* by using the following dynamic programming equation, which involves a local optimization over the inputs:

$$L_k^*(x_k) = \min_{u_k} \left\{ l(x_k, u_k) + L_{K+1}^*(x_{K+1}) \right\}. \tag{28}$$

For further reading see [6]

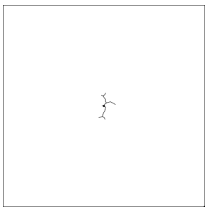
D. Rapidly exploring Random Trees (RRT)

1) What is a RRT:

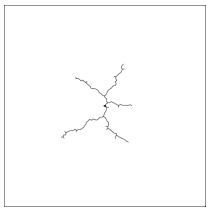
Another very interesting field of research are Rapidly-exploring Random Trees, which were first introduced by LaValle in October 1998 [7].
The main idea behind RRTs is a randomized data structure that is designed for a broad class of path planning problems that are viewed as search in a metric space X for a continuous path from an initial state x_{init} to a goal region $X_{\text{goal}} \subset X$
In addition, a fixed obstacle region $X_{\text{obs}} \subset X$ must be avoided, but there is no explicit representation of it available, thus one can only check if a specific state lies in X_{obs}.
The positive aspect about RRTs is, that X_{obs} not only represents solid obstacles like walls, trees or blocks, but can represent any type of constraints. These include velocity bounds of a self-driving car, configurations at which a robot is in collision with an obstacle in the world and any other constraints occurring in motion planning problems.
When constructing the RRT, the vertices represent the states of the system in X_{free}, and each edge will correspond to a path that lies in completely in X_{free}. [7]

2) How does a RRT look like and how to construct:

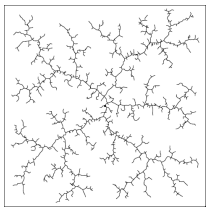
1) Pick a random sample in the search space.
2) Find the nearest neighbor of that sample.
3) Select an action from the neighbor that heads towards the random sample.
4) Create a new sample based on the outcome of the action applied to the neighbor.
5) Add the new sample to the tree, and connect it to the neighbor.



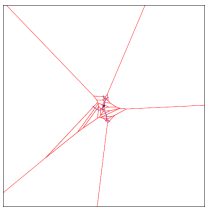
At the beginning the search space is completely unexplored. The tree begins spanning into it.



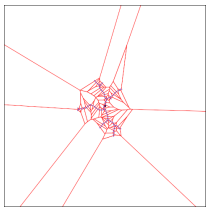
The tree reaches further into the unexplored space.



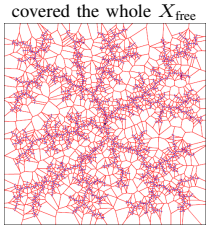
The tree is reaching to the limits of the plane and is getting denser. At the limit it will have covered the whole X_{free}



Looking at the voronoi bias of the vertices, there are huge unexplored parts.



The tree is always taking a sample out of the biggest unexplored part. This smallers the part.



When proceeding, the voronoi bias are simultaneously getting smaller with every iteration. Their area tends to be 0 at the limit.

[7], [1]

3) RRTs and dynamic programming:

In the dissertation of Oktay Arslan at the Georgia Institute of Technology in December 2015, he gives a extension of the solely sampling based approach of the original RRT.
He introduces an algorithm called RRT$^{\#}$ which utilizes the main ideas from RRT, but expanded it with a relaxation step which uses Dynamic Programming.
For understanding the algorithm, some concepts have to be defined:

- Predecessor vertices: Given a vertex $v \in V$ in a directed Graph $G = (V, E)$, the function pred : $(G, v) \mapsto V' \subseteq V$ returns the vertices in V from which vertex v can be reached.

$$\text{pred}(G, v) := \{u \in V : (v, u) \in E\}$$

Following thereof, they rewrote the bellman equation for their purpose in terms of the cost-to-come value as follows:

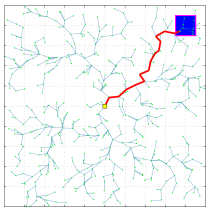
$$g^*(v_i) = \begin{cases} 0 & v_i = x_{\text{init}} \\ \min_{v_j \in pred(G,v_i)}(g^*(v_j) + c(v_j, v_i)) & v_i \in V \setminus \{x_{\text{init}}\} \end{cases} \tag{29}$$

Based on their formulation of the Bellman equation, they split the Algorithm into two parts:

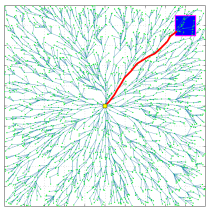
a) Exploration: This samples a point in X_{free} randomly and then extends the underlying graph towards the sampled point by including it as a new vertex in the current graph by connecting the missing edges. [1]

b) Exploitation: In this step the explored graph is getting refined by finding promising vertices. The values of these vertices are updated and unnecessary ties are broken so that the algorithm only expands vertices that are very likely to lie on the shortest path.

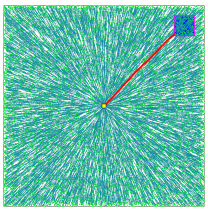
The following images show the two algorithms.



The tree after 250 iterations

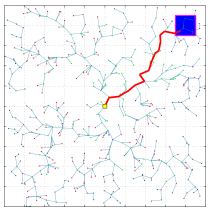


The tree after 2500 iterations

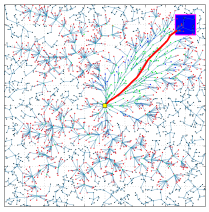


The tree after 25000 iterations

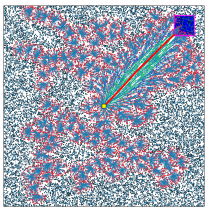
The evolution of the tree computed by RRT, the original algorithm. The tree is spanning equally in all directions resulting in a uniform density.



The tree after 250 iterations



The tree after 2500 iterations



The tree after 25000 iterations

The evolution of the tree computed by RRT$^{\#}$. The tree is spanning mostly into the direction of the promising vertices. The density around the shortest path is much higher than in the regions not associated with it.

In the uniform distribution of the original RRT the computation time is mostly wasted onto vertices that have no chance of being part of the shortest path. Due to the fact that in RRT$^{\#}$ only the promising vertices are expanded and not promising ones are left, the density around the shortest path is really high. Thereof RRT$^{\#}$ safes the time and effort wasted by the original RRT and uses it only on the promising vertices resulting in a huge improvement in runtime and memory usage. The interested reader can have a look at [1] for further information and mathematical proofs.

V. Conclusion

As seen in this paper, Dynamic Programming is a very powerful approach for solving multistage decision problems. By breaking down difficult problems into sets of overlapping subproblems with optimal substructure an overall optimal solution can be found by only solving the small subproblems. In robot motion, dynamic programming can be applied in a wide field of tasks and can be a really useful approach that takes the place of the classic algorithms. Many algorithms are already getting improved or replaced by Dynamic Programming algorithms and the set of fields where it can be applied is nearly unlimited.

VI. Appendix

The source code of the implementation of the maze example can be obtained under:

https://github.com/Brucknem/Get-me-out-of-here.

References

[1] O. Arslan, "Machine learning and dynamic programming algorithms for motion planning and control," 2015-12-01. [Online]. Available: http://hdl.handle.net/1853/54317

[2] S. Dreyfus, "Richard bellman on the birth of dynamic programming," vol. 50, pp. 48–51, 2002. [Online]. Available: https://doi.org/10.1287/opre.50.1.48.17791

[3] R. Bellman, "The theory of dynamic programming," The RAND Corporation, 1954. [Online]. Available: https://www.rand.org/content/dam/rand/pubs/papers/2008/P550.pdf

[4] T. H. Cormen, *Introduction to algorithms*, 2nd ed. MIT Press, 2007.

[5] R. Bellman, *Dynamic Programming*, ser. Dover Books on Computer Science. Dover Publications, 2013. [Online]. Available: http://gbv.eblib.com/patron/FullRecord.aspx?p=1897424

[6] S. LaValle, "From dynamic programming to rrts: Algorithmic design of feasible trajectories," University of Illinois. [Online]. Available: http://msl.cs.illinois.edu/~lavalle/papers/Lav02.pdf

[7] ——, "Rapidly-exploring random trees: A new tool for path planning," Iowa State University, October 1998. [Online]. Available: http://msl.cs.illinois.edu/~lavalle/papers/Lav98c.pdf

[8] R. Bellman, "On the theory of dynamic programming," The RAND Corporation, 1952.

[9] ——, "On a routing problem," vol. 16, pp. 87–90, 1958. [Online]. Available: http://www.jstor.org/stable/43634538

[10] S. P. Bradley, A. C. Hax, and T. L. Magnanti, *Applied mathematical programming*, [19. dr.] ed. Addison-Wesley, 1992, hax, Arnoldo C. (VerfasserIn) Magnanti, Thomas L. (VerfasserIn).

[11] K. L. Cooke and E. Halsey, "The shortest route through a network with time-dependent internodal transit times," vol. 14, pp. 493–498, 1966. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0022247X66900096

[12] "Operations research proceedings 1994: Selected papers of the international conference on operations research, berlin, august 30 – september 2, 1994," 1995.

[13] S. M. LaValle, *Planning Algorithms*. Cambridge University Press, 2006 % References from the book, available at http://planning.cs.uiuc.edu/.

[14] P. D. Pra, C. Rudari, and W. J. Runggaldier, "On dynamic programming for multistage decision problems under uncertainty," in *Operations Research Proceedings 1994: Selected Papers of the International Conference on Operations Research, Berlin, August 30 – September 2, 1994*, U. Derigs, A. Bachem, and A. Drexl, Eds. Springer Berlin Heidelberg, 1995, pp. 70–75. [Online]. Available: https://doi.org/10.1007/978-3-642-79459-9_14