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:	01. January 2017

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 discussed.**

I. Introduction

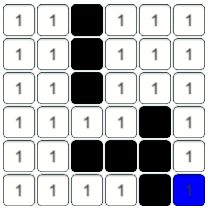
Remember the game Labyrinth, in which the game represents a maze. The field is build up by fixed and moving pieces showing walls and passages. The goal of the game is to rearrange the maze by moving rows of not fixed tiles and enable ones game character to find treasures and return them to goal tiles.

For reaching 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 pathfinding algorithms like Dijkstras Algorithm or Breadth-First Search, but also with **Dynamic Programming**, which I will be using in this Paper.

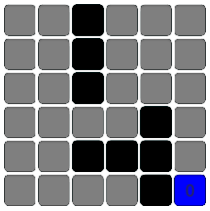
Therefore, I define a grid map $M = \mathbb{N}^{nxm}$ containing static walls $M(i, j) = \infty$ representing the maze, and exit(s) of the labyrinth which the player wants to reach.

The goal of this paper is to point out ways to determine an optimal policy $P = \mathbb{N}^{nxm} \rightarrow A$ to exit the labyrinth. To achieve this, a way to calculate the value $v(i, j)$ of the current position is searched from what the actual best action can be derived.

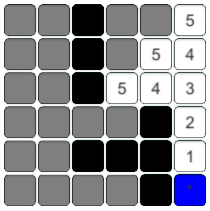
Later on, the algorithm will look like in the following example:



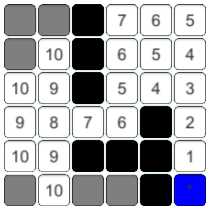
The original grid showing the costs of each tile, walls (black) and goals (blue).



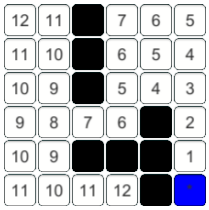
Initialization of the algorithm with ∞ cost. Only the goal (blue) is set and is the starting point. It has a value of 0.



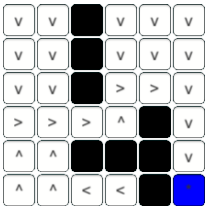
After a few iterations the values of some tiles are set. The value is equal to the length of the shortest path to the goal.



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 that transforms 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 of patients at a medical clinic or long term investment programs.

//TODO: Mehr ausbauen Review ARSLAN DISSERT P.13

Underlying of these problems is always a multistage decision problem. "The characteristic feature of these problems is that they concern a system whose state $x_k \in X_k$ evolves according to a recursive relation of the form

$$x_{k+1} = f_k(x_k, a_k, w_k); \quad k = 0, 1, ...$$

where $a_k \in C_k$ is the decision in period k and $w_k \in D_k$ is the value in period k"[11].

B. Richard Bellman

Dynamic Programming was first described by the american mathematician Richard Bellman who was working on multistage decision problems from 1949 until 1955. After 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 decides 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 the word programming.

So the name Dynamic Programming was born and no one

could imagine what he was doing which made it perfect for hiding his mathematical research.

C. The Principle of Optimality

But how does Dynamic Programming break down complex problems and ensures that by solving the subproblems a optimal policy is found?
For starting a more precise discussion, let me 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."[3]

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

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

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

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

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

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. "We shall call this function the N-stage return. A policy consists of a selection of N transformations

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

yielding successively the states

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

"[3] Now considering the maximum value of the return function $R(p_N)$, regulated by the optimal policy, will only depend of the initial vector p and the number of stages N.
Now we can 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

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

To get a functional equation for $f_N(p)$, 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 $f_{N-1}(T_k(p))$.
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

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

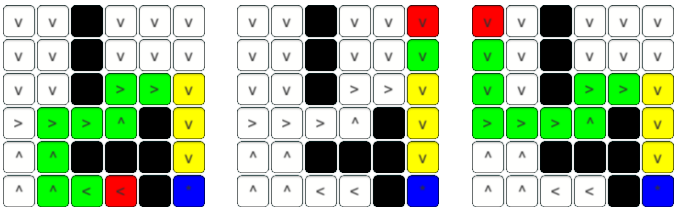
"It is clear that a knowledge of any particular optimal policy, not necessarily unique, will yield $f_N(p)$, which is unique. Conversely, given the sequence $\{f_N(p)\}$, all optimal policies may be determined."[3]

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 an 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.
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

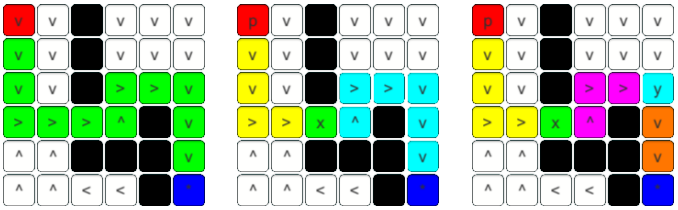
again, which leads you to also recalculate the way down the passage.
So for any position, the passage would be a subproblem that overlaps in every calculation.



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.
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 optimal substructure 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.
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, often also 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 **vorhergehenden** 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 Bellmn 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 now 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 at t_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.

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 an optimal policy $P = \mathbb{N}^{nxm} \rightarrow A$ 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

$$v(i, j) \in \mathbb{N} \quad 1 <= i <= n, 1 <= j <= m \tag{9}$$

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).

So now let us at first begin with the trivial cases.
If a prisoner is already standing on a goal field, the cost to reach the goal would, obviously, be

$$c(i, j) = 0, \quad \text{if } (i, j) \in \{Goals\} \tag{10}$$

. Furthermore would also the value of a goal be 0, because the length of the shortest path from a position to itself is 0. In addition, if the prisoner isn't even in the goal, he will never be able to reach a goal, resulting in

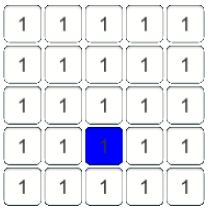
$$v(i, j) = \infty, \quad \text{if } i < 1, i > n, j < 1, j > m \tag{11}$$

. At last if the position we're looking for is a wall, the cost is, by definition

$$c(i, j) = \infty, \quad if(i, j) \in \{Walls\} \tag{12}$$

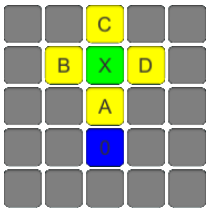
and the value of the tile is also ∞, because there will never be a payoff of stepping into a wall, because it isn't possible at all.

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.
$c(i, j) = 1, \quad \forall(i, j) \in n \times m$

Now 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, so the positions you could reach would be A, B, C and D.
Now, if you moved onto one of these positions, you would have to pay the cost for each position and have to add it onto the value of this position. So making one of the steps, you would have to pay 1. The resulting length of the shortest path now would be the value of the new position added to the cost it needed to step here.
Now you could repeat this step and look at all the positions reachable from this position. As an example, let us look at the position A.
One can now easily repeat these steps in an recursive way, summing up the cost until a goal is reached.
So, if finally a goal state is reached, we would know that it's value is 0, what would give us the possibility to calculate the values of the positions visited to reach it.



Standing at position X you could reach A, B, C, D.

Now standing at position A looking for all the reachable positions.

A goal state has been reached. We know, by definition, it's value is 0.

To step on the goal, we would have to pay a cost of 1 and the value of the goal is 0. So summed up, for a tile that is in direct neighborhood of a goal state, it's value would be

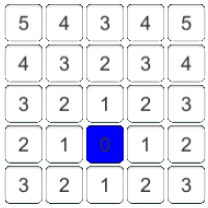
$$v(i, j) = v(x, y) + 1, \quad (x, y) \in \{Goals\} \tag{13}$$

This would propagate back until the initial position is reached.

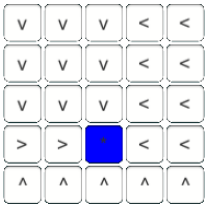
Position A has a value of 1.

Position X has a value of 2.

If you apply this approach to every position in the maze, you could calculate the value of every single position and from this grid of values, you could just always step on the position next to you which has the lowest cost which would result in a the optimal action for every tile in the maze.



All values are found.



Stepping onto the tile with the lowest value next to you, you can derive the best action.

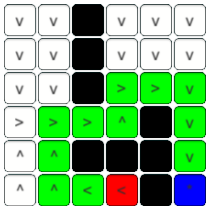
When coming to a mathematical representation of this problem, we saw that for every tile it's value was the value of the tile with the lowest cost next to it summed up with the cost it would take to step on this tile.
From this the following function can be derived, which solves the Bellman equation.

$$v(i,j) = \begin{cases} \infty & \text{if } (i,j) \in \{\text{Walls}\} \cup \{\mathbb{R}^2 \setminus \{n \times m\}\} \\ 0 & \text{if } (i,j) \in \{\text{Goals}\} \\ \min\{v(i,j+1)+1, & \\ v(i+1,j)+1, & \\ v(i,j-1)+1, & \\ v(i-1,j)+1\} & \text{if in the maze and not a wall or a goal} \end{cases} \tag{14}$$

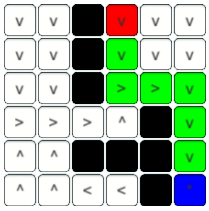
The first to lines are the base cases to stop the recursion if a goal is found, the maze is left or a wall would be visited. If n and m are both finite numbers we can see that the recursion will always terminate because at some point in iterations the maze will be left in all directions which causes the calculation to stop.
The third one is the recursive step searching for the minimum of the sum of the value of the four adjacent positions and their corresponding costs.
If we now don't only allow 1 as cost per tile, but also arbitrary positive values, the equation can be generalized to be used in any grid or graph with non negative costs.

$$v(i,j) = \begin{cases} \infty & \text{if } (i,j) \in \{\text{Walls}\} \cup \{\mathbb{R}^2 \setminus \{n \times m\}\} \\ 0 & \text{if } (i,j) \in \{\text{Goals}\} \\ \min\{v(i,j+1)+c(i,j+1), & \\ v(i+1,j)+c(i+1,j), & \\ v(i,j-1)+c(i,j-1), & \\ v(i-1,j)+c(i-1,j)\} & \text{if in the maze and not a wall or a goal} \end{cases} \tag{15}$$

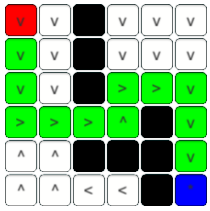
When the precomputation of the best action for every tile is finished, one can easily derive the optimal policy with ease.



The optimal policy is $(\leftarrow, \leftarrow, \uparrow, \rightarrow, \rightarrow, \uparrow, \rightarrow, \rightarrow, \downarrow, \downarrow, \downarrow)$



The optimal policy is $(\downarrow, \downarrow, \rightarrow, \rightarrow, \downarrow, \downarrow, \downarrow)$



The optimal policy is $(\downarrow, \downarrow, \downarrow, \rightarrow, \rightarrow, \rightarrow, \uparrow, \rightarrow, \rightarrow, \downarrow, \downarrow, \downarrow)$

B. Pseudo code for solving this task

```
public class Algorithm {

private static int[,] grid;
private static List<Vector2> goals;
private static List<Vector2> openList = new List<Vector2>();

public static void Recursive(int[,] grid, List<Vector2>
    ↪ goals){

 Algorithm.grid = grid;
 Algorithm.goals = goals;

 value = new int[Width, Height];
 policy = new char[Width, Height];

 for (int i = 0; i < Width; i++)
 {
  for (int j = 0; j < Height; j++)
   {
    value[i, j] = MaxCost;
   }
 }

 foreach (Vector2 goal in goals)
 {
  value[(int)goal.x, (int)goal.y] = 0;
  policy[(int)goal.x, (int)goal.y] = '*';
  ValueFunction(goal, new List<Vector2>() { goal });
 }
}



private static void ValueFunction(Vector2 pos, List<Vector2>
    ↪ open/*, List<Vector2> closed*/){
 open.Remove(pos);

 for (int i = 0; i < deltas.Count; i++){
  Vector2 prev = pos - deltas[i];
  if (prev.x >= 0 && prev.x < Width && prev.y >= 0 && prev.
      ↪ y < Height)
  {
   if (grid[(int)prev.x, (int)prev.y] == MaxCost)
   {
    value[(int)prev.x, (int)prev.y] = MaxCost;
   }
   else if (goals.Contains(prev))
   {
```

```
      value[(int)prev.x, (int)prev.y] = 0;
    }
    else if (value[(int)prev.x, (int)prev.y] > value[(int)
        ↪ pos.x, (int)pos.y] + grid[(int)pos.x, (int)pos.
        ↪ y])
    {
      if (!open.Contains(prev))
        open.Add(prev);

      if (goals.Contains(prev))
      {
        value[(int)prev.x, (int)prev.y] = 0;
        policy[(int)prev.x, (int)prev.y] = '*';
      }
      else
      {
        value[(int)prev.x, (int)prev.y] = value[(int)pos.x,
            ↪ (int)pos.y] + grid[(int)pos.x, (int)pos.y];
        policy[(int)prev.x, (int)prev.y] = deltaNames[i];
      }
    }
  }
 }

 //Sort list
 if (open.Count > 0)
 {
  int index = -1;
  int lowest = int.MaxValue;
  for (int i = 0; i < open.Count; i++)
  {
   if (value[(int)open[i].x, (int)open[i].y] + grid[(int)
       ↪ pos.x, (int)pos.y] < lowest)
   {
    lowest = value[(int)open[i].x, (int)open[i].y];
    index = i;
   }
  }
  ValueFunction(open[index], open/*, closed*/);
 }
}

}
```

IV. Related Work

A. Algorithmic Design of Feasible Trajectories

1) abstract:

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.

2) problem formulation:

1) State Space: $X \subset \mathbb{R}^n$
2) Boundary Values: $x_{\text{init}} \in X, X_{\text{goal}} \subset X$
3) Contraint Satisfaction: A function, $D : X \to \{\text{true, false}\}$, that determines wheater global constraints are satisfied from state x.
4) Inputs: $U(x) \subset \mathbb{R}^m, \forall x \in X$, which specifies the set of controls.
5) State Transition Equation: $\dot{x} = f(x, u)$

3) 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

$$L(x_1, \cdots, x_{K+1}, u_1, \cdots, u_K) = \sum_{k=1}^{K} l(x_k, u_k) + l_{K+1}(x_{K+1}) \tag{16}$$

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

$$L_k^*(x_k) = \min_{u_k, \cdots, u_K} \left\{ \sum_{i=k}^{K} l(x_i, u_i) + l_{K+1}(x_{K+1}) \right\}. \tag{17}$$

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:

$$L_k^*(x_k) = \min_{u_k} \left\{ l(x_k, u_k) + L_{K+1}^*(x_{K+1}) \right\}. \tag{18}$$

For further reading see [8]

B. 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 [**?**].

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 $X_{\text{goal}} \subset X$

WHAT IS A METRIC

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}.

WHAT IS EXPLICIT REPRESENTATION

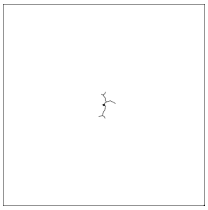
The nice thing about RRTs is, that X_{obs} not only represents solid obstacles like walls, trees or blocks as known from computer games, but can represent any type of constraints. These include velocity bounds of a self-driving car, configurations at which a robot is in collision with an obstacle in the world, Planning Detailed Animation at runtime and any other constraints occurring in motion planning problems.

When constructing the RRT, the vertices represent the states

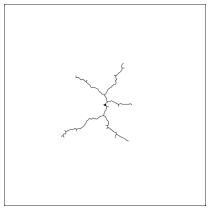
of the system in X_{free}, and each edge will correspond to a path that lies in completely in X_{free}. [**?**]

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

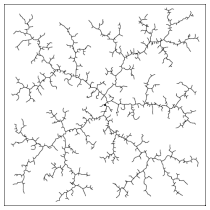
1) Pick a random sample in the search space.
2) Find the nearest neighbor of that sample.
3) Select an action from the neighbor that heads towards the random sample.
4) Create a new sample based on the outcome of the action applied to the neighbor.
5) Add the new sample to the tree, and connect it to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



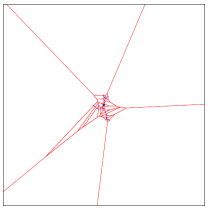
At the beginning the search space is completely unexplored. The tree begins spanning into it.



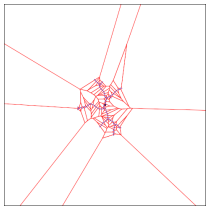
The tree reaches further into the unexplored space.



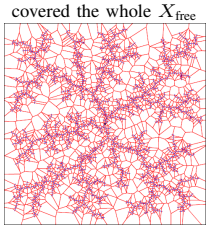
The tree is reaching to the limits of the plane and is getting denser. At the limit it will have covered the whole X_{free}



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.

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 expand it to a relaxation step, used in Dynamic Programming.

Relaxation is a term used iterative methods for solving systems of equations, in this case of optimization problems in numerical mathematics.

They can be written down in the form of

$$x(k+1) = f(x(k)), \quad k = 0, 1, \cdots \tag{19}$$

where $x(k) \in \mathbb{R}^n$ is an n-dimensional vector, and $f : \mathbb{R}^n \mapsto \mathbb{R}^n$ is a vector-valued function.

The Bellman function is of this form, what spans the bow between Dynamic Programming and RRT$^{\#}$.

For understanding the algorithm, some concepts have to be defined:

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

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

- 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:

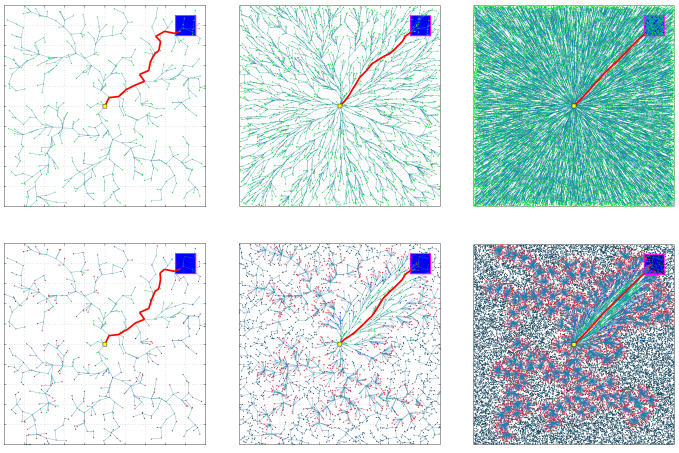
$$g*(v_i) = \begin{cases} 0 & v_i = x_{\text{init}} \\ \min_{v_j \in pred(G, v_i)}(g^*(v_j) + c(v_j, v_i)) & v_i \in V \setminus \{x_{\text{init}}\} \end{cases} \tag{20}$$

P.42 Based on their formulation of the Bellman equation, they split the Algorithm into two parts:

a) Exploration: This samples randomly a point in X_{free} 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. [10]

b) Exploitation: This step implements the task of improving the cost-to-come values of the current vertices as new information becomes available. It also encodes the lowest-cost path information for the promising vertices (see equation (20) below) and v^*_{goal} (the goal vertex that has the lowest cost-to-come value in the goal set) as a spanning tree rooted at the initial vertex. Cost-to-come values of nonstationary vertices are updated in an order based on their f-values, i.e., an underestimate of the cost of the optimal path from the initial vertex to the goal set passing through the vertex of interest, and ties are broken in favor of vertices which have smaller g-values at each iteration of the Gauss-Seidel version of the Bellman-Ford algorithm.

The following images show the two algorithms.



As you can see, dynamic programming is a great way to improve the runtime and the speed in which an algorithm converges to an optimal solution. The interested reader can have a look at [10] for further information and mathematical proofs.

C. Bellman Ford, Gauss Seidel and Jakobi

V. Conclusion

References

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

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

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

[4] 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).

[5] 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

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

[7] 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

[8] 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

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

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

[11] 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: