Skoltech

Skolkovo Institute of Science and Technology

L05: Discrete Optimal Planning

Planning Algorithms in AI

Prof. Gonzalo Ferrer

Skoltech, 25 November 2022

Summary L05, LaValle 2.3, 8.1-8.2

- Discrete Optimal planning
- Value Iteration, fixed-length paths
- Dynamic Programming and the Principle of Optimality
- Value Iteration, infinite-length paths
- Policy and Feedback Plan
- Navigation functions
- Wavefront propagation

Discrete Optimal Planning

- Let's get back to discrete planning and formulate an optimal planning problem.
- We discussed in L02 how to find optimal paths with the Dijkstra and A* algorithms, from the initial state (configuration) to the goal state.

$x' = f(x, u)$

- Now, the idea is to solve the optimal problem for all possible sequences of state-action.
- Let's start by selecting a sequence of constant size *K*, and find the one that gives the best cost.

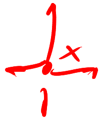
Optimal fixed-length plan: Problem Formulation

1. A non-empty and finite **state space** X
2. For each state $x \in X$, a finite **action space** $U(x)$
3. A **state transition function** f that produces a state:

$$x' = f(x, u)$$

4. An **initial state** $x_I \in X$
5. A **goal set** $X_G \subset X$
6. A number of stages K, the exact length of the plan $\pi_K = \{u_1, \ldots, u_K\}$
7. The **cost** of the plan, evaluated at each of the sates and actions in the plan.

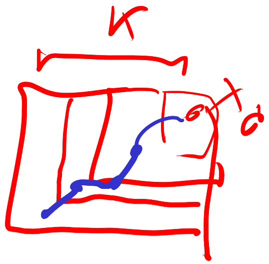
$$L(\pi_K) = \sum_{k=1}^{K} l(x_k, u_k) + l_F(x_F), \qquad \text{where } l_F(x_F) = \begin{cases} 0 & x_F \in X_G \\ \infty & \text{otherwise} \end{cases}$$



$U(x)$

traj = $\{x_1, \ldots, x_{K+1}\}$

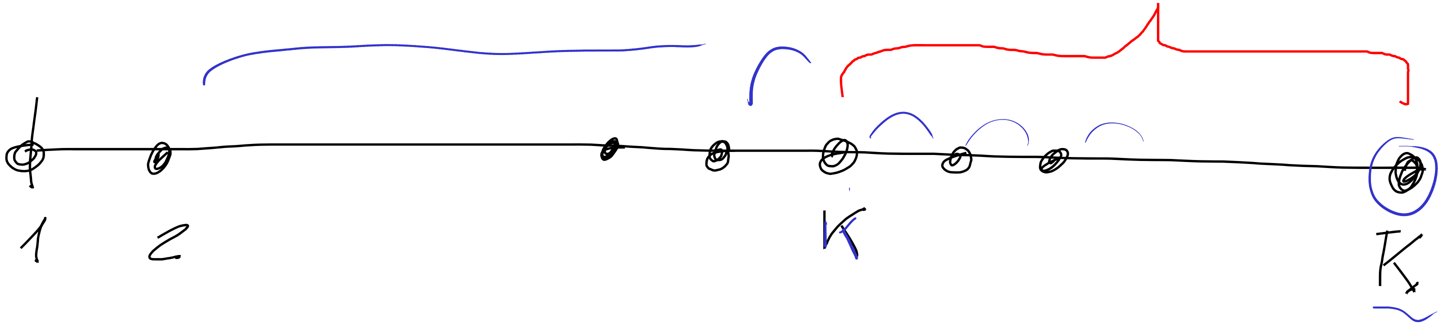
Backward Value Iteration



To solve this problem we can define a function, which is unknown for now, that is the optimal cost-to-go to the final state from the *k*th state:

$$G_k^*(x_k) = \min_{u_k,\dots,u_K} \left\{ \sum_{i=k}^{K} l(x_i, u_i) + l_F(x_F) \right\}$$

This function can be seen as the cost of evaluating the tail of the problem, if we divide this way:


Tail sub-problem

Backward Value Iteration

In the last state (zero length path), the cost-to-go is perfectly defined, since there is no required action to do:

$$\boxed{G_F^*(x_F) = l_F(x_F)} \ *$$

$x \in$

On the next iteration (length 1), we can calculate $G*$ for the last action in the plan:

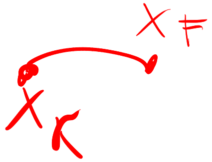
$$G_K^*(x_K) = \min_{u_K} \{l(x_K, u_K) + l_F(x_F)\}$$

One can simply manipulate the the previous expression, considering the transition function after executing the last action u_K:

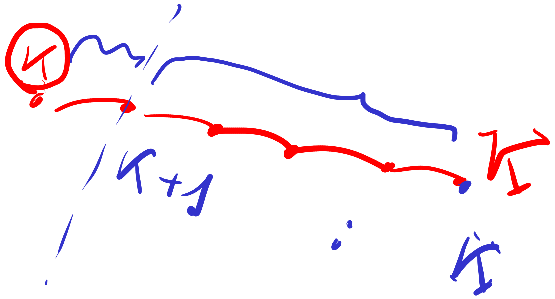
$$G_K^*(x_K) = \min_{u_K} \{l(x_K, u_K) + G_F^*(f(x_K, u_K))\}$$

$x_{k+1} = f(x_K, u_K)$

$*$



Backward Value Iteration



The optimal cost-to-go function $G*$ can be *exactly* divided in two parts:

$$G_k^*(x_k) = \min_{u_k,\dots,u_K}\left\{\sum_{i=k}^{K} l(x_i,u_i) + l_F(x_F)\right\}$$

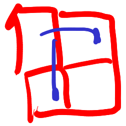
$u_{k+1}: K$

$$G_k^*(x_k) = \min_{u_k}\left\{\min_{u_{k+1},\dots,u_K}\left\{l(x_k,u_k) + \sum_{i=k+1}^{K} l(x_i,u_i) + l_F(x_F)\right\}\right\}$$

$$G_k^*(x_k) = \min_{u_k}\left\{l(x_k,u_k) + \min_{u_{k+1},\dots,u_K}\left\{\sum_{i=k+1}^{K} l(x_i,u_i) + l_F(x_F)\right\}\right\}$$

A recursive form emerges, what is known as **Value Iteration**:

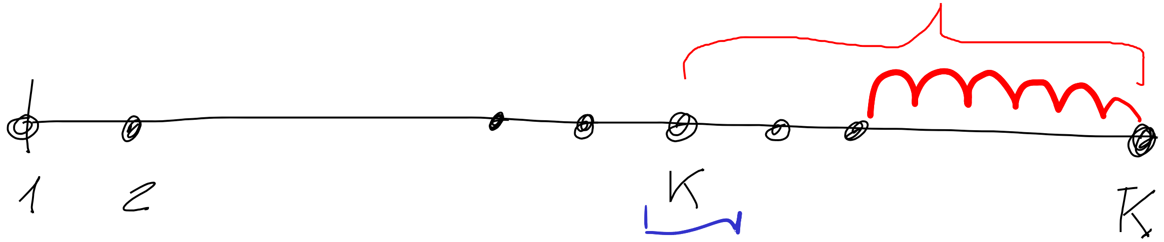
$$\boxed{G_k^*(x_k) = \min_{u_k}\left\{l(x_k,u_k) + \boxed{G_{k+1}^*(x_{k+1})}\right\}, \text{ where } x_{k+1} = f(x_k,u_k)}$$



Dynamic Programming and the Principle of Optimality

An interpretation of the recursive form we have just derived can be explained in terms of the **Principle of Optimality**:

> *The given an optimal sequence* $\{u_1^*, \ldots, u_K^*\}$ *which together with the initial state determines the state sequence* $\{x_1^*, \ldots, x_K^*, x_{K+1}^*\}$ *, then the principle of optimality states that the optimal solution to the corresponding tail sub-problem is the truncated sequence* $\{u_k^*, \ldots, u_K^*\}$


Tail sub-problem

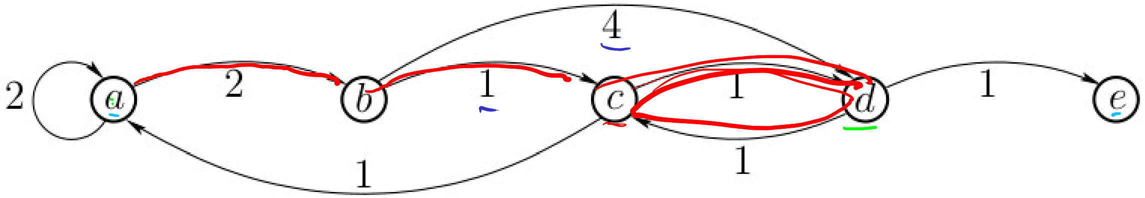
Dynamic Programming (DP) is based on the optimality principle and computes in a backwards fashion a solution to the optimal value incrementally, without need to recalculate plans.

Backward Value Iteration: Example

K=5

The goal state is $x_G = d$ and the length $K = \overset{3}{5}$

$$G_k^*(x_k) = \min_{u_k} \left\{ l(x_k, u_k) + G_{k+1}^*(f(x_k, u_k)) \right\}$$



G×G

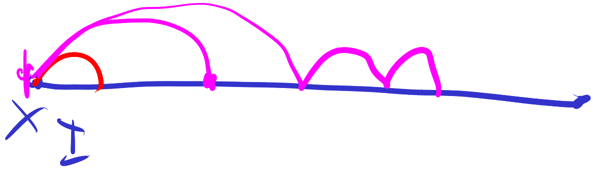
		a	b	c	d	e
(l=0)	G_F^*	∞	∞	∞	0	∞
(l=1)	G_5^*	∞	4	1	∞	∞
(l=2)	G_4^*	6	2	∞	2	∞
(l=3)	G_3^*	4	6	3	∞	∞
	G_2^*					
	G_1^*					

Forward Value Iteration (VI)

$x_{k+1} = f(x_k, u_k)$ (backward VI)

The same derivation of DP can be repeated, now in forward mode, calculating the **cost-to-come**:

$$C_k^*(x_k) = \min_{u_1,\dots,u_{k-1}} \left\{ l_I(x_1) + \sum_{i=1}^{k-1} l(x_i, u_i) \right\}$$



The cost at the initial first state (it could be 0)

$$C_1^*(x_1) = l_I(x_1)$$

After applying DP we obtain a recursively, from *1* to *K*, the cost-to-come cost:

$$C_k^*(x_k) = \min_{u^{-1} \in U^{-1}(x_k)} \left\{ C_{k-1}^*(x_{k-1}) + l(x_{k-1}, u_{k-1}) \right\}, \text{ where } x_{k-1} = f^{-1}(x_k, u_k^{-1})$$

VI of unspecified length

- Now, we will modify the problem formulation and remove the length of the trajectories we are optimizing and instead just try to reach the goal state.
- To do that, we need to introduce a special action, the **termination action** u_T.
- Each $U(x)$ contains the termination action. If u_T is applied at x_k, then the state remains unchanged and no more cost is accumulated.
- Recall the previous example, where we could not select the initial state, but only the length of the plan, and that results in different costs for the same state.
- For the initial state being already the goal, then the optimal length =5 sequence to execute would be $\{u_T, u_T, u_T, u_T, u_T\}$

VI of unspecified length

The idea is to apply an infinite length until we find a **stationary** optimal cost-to-go function: Convergence of the algorithm.

$$G^*_{i-1}(x) = G^*_i(x), \forall x \in X$$

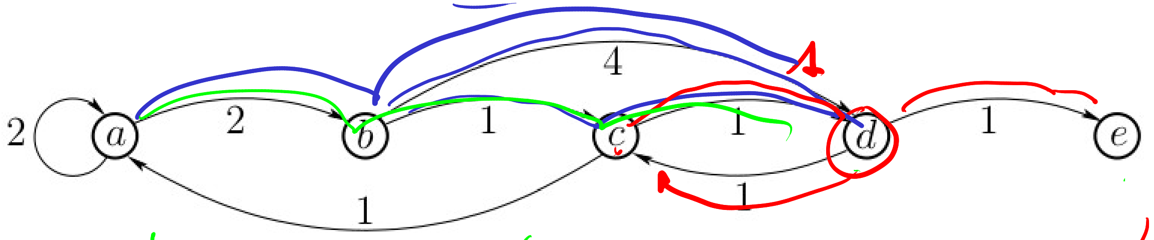
Then, the function *G* will indicate the optimal cost-to-go, from any initial state to the goal state

$$\boxed{G^*(x) = \min_u \{l(x,u) + G^*(f(x,u))\}}$$

VI: Example (again)

The goal state is $x_G = d$

$$G_k^*(x_k) = \min_{u_k} \{ \overset{0}{l(x_k, u_k)} + \overset{0}{G_{k+1}^*(f(x_k, u_k))} \}$$



	a	b	c	d	e
G_0^*	∞	∞	∞	0	∞
G_{-1}^*	∞	4	1	0	∞
G^*	6	2	1	0	∞
G^*	4	2	1	0	∞
G^*	4	2	1	0	∞

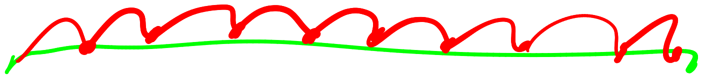
μ_t

=

$x_k = x_d$

$\{\mu_{c\to d}, \mu_t, \mu_t, u_t\}$

Policy definition



The optimal cost-to-go, in addition, can be used to recover the optimal plan in the following way:

$$u^* = \arg \min_{u \in U(x)} \{l(x,u) + G^*(f(x,u))\}$$

x	$\pi(x)$
x_1	u_{up}
$\vdots$	$\vdots$

This implies that we can define the term **policy,** which in this case is a function (table) from all states to the optimal action, thanks to DP and the $G*$ function.

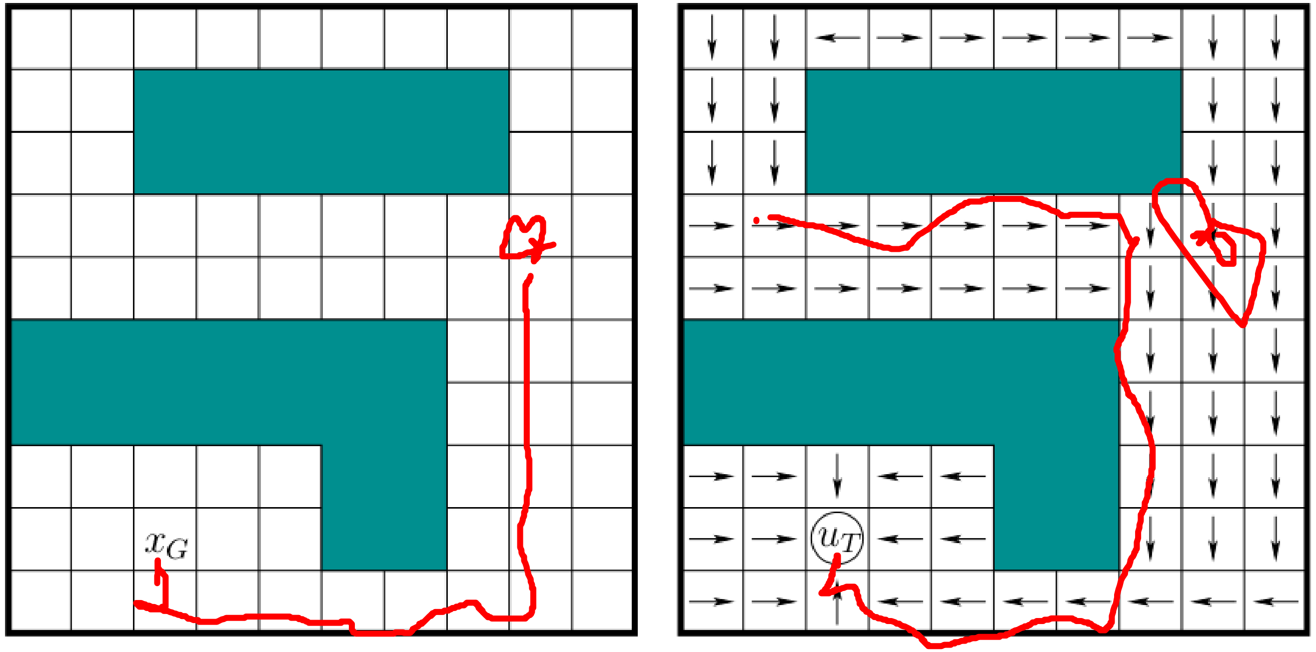
$$\pi^*(x) = u^*, \forall x \in X$$

In terms of planning, after calculating VI and converge, we have guarantees that the optimal action to execute is in the policy, and thus, the optimal plan becomes a sequence of transition functions until the goal is reached.

$$x' = f(x, \pi^*(x))$$

Example: 2D Policy or Feedback Plan

The optimal cost-to-go allows us to calculate the optimal policy and generate from every state the optimal action to reach the goal or when the terminal action u_T is executed. This can also be seen as a **Feedback Plan**: $\pi(x) : X \to U$



Dijkstra revisited

- Recall the **Dijsktra** algorithm from L02, it is a form of DP, with some improvements in efficiency.
- Infinite values of costs can be considered as "not visited states", and Dijkstra makes a clever use of not checking them, saving a lot of comp. time.
- For stationary states, where the optimal cost does not change, Dijkstra considers them as "dead" and does not check neither.
- The queue allows to **evaluate only** those states where the cost-to-go can *change*, making it a very efficient algorithm.

Dijkstra revisited

- Why do we explain VI then?? Unfortunately, keeping the priority queue can be too expensive and VI could be more efficient sometimes.
- VI is a more general algorithm that generalizes well to continuous state spaces, continuous-time, stochastic optimal planning, etc.
- More generally, Dijkstra belongs to **label-correcting algorithms**, which produces optimal plans by small modifications to the current plan. These kind of algorithms are better suited for logic-based planning and potential issues with combinatorial problems.

Policies and VI: some remarks

- So far we have not discussed about **uncertainty**, however policies add some robustness implicitly.
- Imagine there are some perturbations in the sequence of transitions, resulting in a different state transition than the optimal one.
- Policies allow to correct for these **disturbances** since we still have an optimal plan from any initial state!!
- In the limit, DP and VI could be taken to infinitesimally small discrete states leading to **continuous state** spaces and **continuous-time** transition functions. Today was just an introduction
- DP also admits for random variables and stochastic processes.

Navigation functions and feedback plan

Consider a discrete **potential function**:

$$\phi(x) : X \to [0, \infty)$$

We can define a **feedback plan** through the use of the *local operator*, which selects the action that reduces the potential the most (greedy):

$$u^* = \arg \min_{u \in U(x)} \{\phi(f(x, u))\}$$

Now, the local operator can be slightly modified to yield the expression:

$$u^* = \arg \min_{u \in U(x)} \{l(x, u) + \phi(f(x, u))\}$$

which resembles DP and is exactly equal when the potential function is the optimal cost-to-go.

Navigation functions and feedback plans

How to define a potential function (PF)? Clearly there are many useless PF...

The most desirable PF is that for any state variable causes the arrival in X_G, if it is reachable. If the PF satisfies this, then it is called a **navigation function**.

In order to achieve this, some properties are necessary:

$$\phi(x) = 0, \forall x \in X_G$$

$$\phi(x) = \infty \iff \text{no point in } X_G \text{ is not reachable } \forall x \in X$$

$$\forall x \in X \setminus X_G, \text{the local operator produces a state } x' : \phi(x') < \phi(x)$$

If in addition the plan is optimal, then we have an *optimal navigation function*.

How to construct a PF? Dijkstra for instance construct an optimal NF, and any other forward search method can calculate a NF as well.

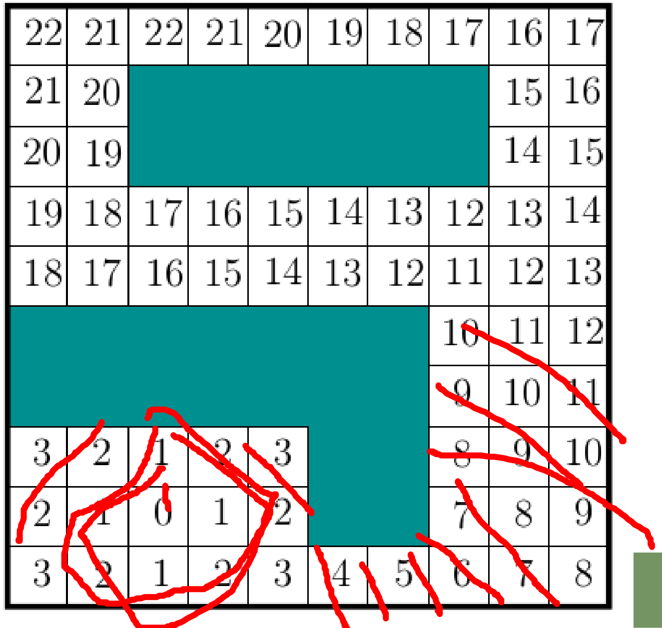
One can also imagine a PF by proposing some functions.

Wavefront propagation

Wavefront is a particular case of a discrete optimal navigation function and a simplified case of the Dijkstra algorithm.

Each action has a unit cost, and as such, there will be many states in the queue with identical same cost.

Wavefront exploits this fact and process them in batch-mode. Complexity O(n)



22	21	22	21	20	19	18	17	16	17
21	20	▨	▨	▨	▨	▨	▨	15	16
20	19	▨	▨	▨	▨	▨	▨	14	15
19	18	17	16	15	14	13	12	13	14
18	17	16	15	14	13	12	11	12	13
▨	▨	▨	▨	▨	▨	▨	10	11	12
▨	▨	▨	▨	▨	▨	▨	9	10	11
3	2	1	2	3	▨	▨	8	9	10
2	1	0	1	2	▨	▨	7	8	9
3	2	1	2	3	4	5	6	7	8

Maximum clearance

Optimal motion plans tend to be close to obstacles, for this reason a maximum clearance algorithm was proposed, also known as NF2. The idea is as follows:

1) Perform wavefront propagation from any state the is in the boundary with the obstacle region.
2) As wavefronts propagate, they meet (approximately) at the maximum clearance point. Then we add such states into the *skeleton state* set S.
3) After wavefront ends, add the goal state into S and perform graph search to find the optimal path.
4) A navigation function can be calculated only from S.
5) Any initial state can be connect to its nearest state in S, in a two step plan.