EE365: Shortest Paths

Deterministic optimal control

The simplest shortest path algorithm

Dijkstra’s algorithm
Deterministic optimal control
Deterministic optimal control

minimize \[ \sum_{t=0}^{T-1} g_t(x_t, u_t) + g_T(x_T) \]
subject to \[ x_{t+1} = f_t(x_t, u_t), \quad t = 0, \ldots, T - 1 \]

▶ variables are \( x_1, \ldots, x_T, u_0, \ldots, u_{T-1} \). \( x_0 \) is given
▶ just an optimization problem, with a trivial information pattern
▶ can extend to case when costs are random, when dynamics are deterministic
▶ useful way to formulate many general optimization problems (e.g., knapsack)
Equivalent shortest path problems

create the *unrolled graph*

- **vertex set** is $\mathcal{V} = \mathcal{X}_0 \cup \cdots \cup \mathcal{X}_T$; if time-invariant, then $\mathcal{V} = \mathcal{X} \times \{0, \ldots, T\}$
- **directed edges** corresponding to $u_t$ from $x_t$ to $x_{t+1} = f_t(x_t, u_t)$
  if there are multiple edges, keep the lowest cost one
- **edge weights** are $g(x_t, u_t)$
- add additional *target vertex* $z$ with an edge from each $x \in \mathcal{X}_T$ with weight $g_T(x)$
- a sequence of actions is a path through the unrolled graph from $x_0$ to $z$
- associated objective is total, weighted path length
Unrolled graph

vertex set is $\mathcal{X}_0 \cup \cdots \cup \mathcal{X}_T$
Unrolled graph

directed edges, labeled by $u_t$, from $x_t$ to $x_{t+1} = f_t(x_t, u_t)$
Unrolled graph

A sequence of actions is a path through the unrolled graph.
Dynamic programming

- Dynamic programming is often too computationally expensive.
- \( T|X||U| \) operations.
- The state space can be so large that we cannot store the value function \( v \).
- Specify the system in code by a function that returns \( f(x, u) \) given \( x, u \); called an oracle or an implicit description.
- For MPC, we are only interested in finding the best action to take given the current state, not given any possible state (as given by DP).
The simplest shortest path algorithm
The simplest shortest path problem

- given weighted directed graph with vertices \( V \) and a source vertex \( s \in V \)
- find lowest cost path from source to every vertex
Problem description

- directed graph
- $\mathcal{V}$ is a finite set of vertices
- $\mathcal{E}$ is the set of directed edges $i \to j$
- for each $i$, $\mathcal{N}_i$ is the set of neighbors $j$ such that $i \to j$ is an edge
- $g_{ij}$ is the cost of edge $i \to j$
- $s$ is the source vertex
- $\mathcal{T} \subset \mathcal{V}$ is the target set
- $\text{dist}(s, i)$ is the cost of the minimum-cost path from $s$ to $i$
- $\text{dist}(s, \mathcal{T}) = \min_{i \in \mathcal{T}} \text{dist}(s, i)$
Problems with value iteration

- we have seen (one form of) the Bellman-Ford algorithm
- it finds the shortest path from a vertex $s$ to all vertices
- often we only want the shortest path from $s$ to some target set $T \subset V$
- e.g., in the unrolled graph, $V = X_0 \cup \cdots \cup X_T$, the source vertex is $x_0 \in X_0$ and the target set is $T = \{z\}$
The simplest algorithm

\[ v_s = 0 \]
\[ v_i = \infty \text{ for all } i \neq s \]

while there is an edge \( i \rightarrow j \) such that \( v_j > v_i + g_{ij} \)

let \( i \rightarrow j \) be any such edge

\[ v_j = v_i + g_{ij} \]

- negative edge weights \( g_{ij} \) allowed
- each step is called edge relaxation
- requires storing the array \( v \), which is the size of \( V \)
- finds the shortest path from source vertex \( s \) to all vertices
The simplest algorithm

\[ v_s = 0 \]
\[ v_i = \infty \text{ for all } i \neq s \]

\[ \textbf{while} \text{ there is an edge } i \rightarrow j \text{ such that } v_j > v_i + g_{ij} \]

let \( i \rightarrow j \) be any such edge

\[ v_j = v_i + g_{ij} \]

if the graph has no negative cycles, then the algorithm terminates, because

\[ \Rightarrow \text{ by induction, at every step } v_i \text{ is either } \infty \text{ or the cost of some path } s \rightsquigarrow i \]

\[ \Rightarrow \text{ these paths are always acyclic} \]

\[ \Rightarrow \text{ at every step, some } v_i \text{ decreases, and there are only finitely many paths} \]
The simplest algorithm

\[ v_s = 0 \]
\[ v_i = \infty \text{ for all } i \neq s \]

while there is an edge \( i \to j \) such that \( v_j > v_i + g_{ij} \)

let \( i \to j \) be any such edge

\[ v_j = v_i + g_{ij} \]

to show the algorithm terminates correctly, we will show that if there is no edge such that \( v_j > v_i + g_{ij} \), then \( v_i = \text{dist}(s, i) \) for all \( i \).

- Suppose for a contradiction that \( v_i \neq \text{dist}(s, i) \) but \( v_j = v_i + g_{ij} \) for all edges

- \( v_i \) is the cost of some path \( s \leadsto j \rightarrow k \leadsto i \)

- Let \( j \rightarrow k \) be the first edge along the path such that \( v_k > \text{dist}(s, k) \)

- Then \( v_j = \text{dist}(s, j) \) and \( \text{dist}(s, k) \geq v_j + g_{jk} \), hence \( v_k > v_j + g_{jk} \)
Properties of the simplest algorithm

- many well-known shortest path algorithms correspond to a particular choice of which order to relax edges
- often very fast
- one can construct (pathological) examples where it is very slow
Dijkstra’s algorithm
Dijkstra’s algorithm

\[ v_s = 0 \]
\[ v_i = \infty \text{ for all } i \neq s \]
\[ F = \{s\} \]

\[ \textbf{while } F \neq \emptyset \]
\[ i = \arg\min_{i \in F} v_i \]  \hspace{1cm} // extract vertex \( i \)
\[ F = F \setminus \{i\} \]
\[ \textbf{for } j \in \mathcal{N}_i \]
\[ \textbf{if } v_j > v_i + g_{ij} \]
\[ v_j = v_i + g_{ij} \]
\[ F = F \cup \{j\} \]

- maintains a set \( F \) called the \textit{frontier} or \textit{open} set
- terminates with \( v_i = \text{dist}(s, i) \) if graph has no negative cycles
- extracts the vertex with smallest \( v_i \), relaxes its outgoing edges
Dijkstra’s algorithm

\[ v_s = 0 \]
\[ v_i = \infty \text{ for all } i \neq s \]
\[ F = \{s\} \]

**while** \( F \neq \emptyset \)

\[ i = \arg\min_{i \in F} v_i \quad // \text{extract vertex } i \]
\[ F = F \setminus \{i\} \]

**for** \( j \in \mathcal{N}_i \)

**if** \( v_j > v_i + g_{ij} \)

\[ v_j = v_i + g_{ij} \]
\[ F = F \cup \{j\} \]

when all \( g_{ij} \geq 0 \)

- algorithm extracts vertices in order of distance from \( s \)
- each vertex is extracted at most once
- \( v_i \geq \text{dist}(s, i) \) always; equality when \( i \) is extracted
Interpretation of Dijkstra’s algorithm

- the algorithm may be thought of as a *simulation of fluid flow*
- imagine fluid traveling from the source vertex $s$, moving at speed 1
- $g_{ij}$ is time for fluid to traverse edge $i \to j$
- set $v_i$ at neighbors of $i$ to be the estimated time of arrival
- when fluid arrives at the next vertex, update the ETA of its neighbors
- some of these estimates may be too large, since the fluid might find shortcuts
Keeping track of visited vertices

\[ v_s = 0 \]
\[ v_i = \infty \text{ for all } i \neq s \]
\[ F = \{ s \} \]
\[ E = \emptyset \]

while \( F \neq \emptyset \)

\[ i = \arg \min_{i \in F} v_i \]  // extract vertex \( i \)

\[ F = F \setminus \{ i \} \]
\[ E = E \cup \{ i \} \]

for \( j \in \mathcal{N}_i \)

\[ \text{if } v_j > v_i + g_{ij} \]

\[ v_j = v_i + g_{ij} \]
\[ F = F \cup \{ j \} \]

\( \triangleright \) keeps track of \( E \), the set of visited vertices, called the \textit{closed} set
Inductive proof

When all weights $g_{ij} \geq 0$, one can show by induction that, after each iteration

- there is some $d$ such that

\[
\text{dist}(s, i) \leq d \quad \text{for all } i \in E
\]
\[
\text{dist}(s, i) \geq d \quad \text{for all } i \notin E
\]

- for all $i$, $v_i$ is the length of the shortest path $s \rightsquigarrow i$ fully contained in $E$
Termination

\[ F = \{s\}; \ E = \emptyset \]
\[ v_s = 0 \]
\[ \textbf{while } F \neq \emptyset \]
\[ i = \arg\min_{i \in F} v_i \quad \text{// extract vertex } i \]
\[ F = F \setminus \{i\}; \ E = E \cup \{i\} \]
\[ \textbf{if } i \in T \text{ terminate} \quad \text{// found target} \]
\[ \textbf{for } j \in N_i \]
\[ \textbf{if } j \notin F \cup E \]
\[ v_j = v_i + g_{ij}; \ F = F \cup \{j\} \]
\[ \textbf{else if } j \in F \]
\[ v_j = \min\{v_j, v_i + g_{ij}\} \]
\[ \textbf{else if } v_j > v_i + g_{ij} \]
\[ v_j = v_i + g_{ij} \]
\[ E = E \setminus \{j\}; \ F = F \cup \{j\} \quad \text{// removing from } E \text{ is optional} \]
Theorem

for any weights $g$ such that $\text{dist}(i, T) \geq 0$ for all $i \in V$

- the algorithm terminates
- on termination, $v_i = \text{dist}(s, i)$

- condition allows negative edges, but no negative cycles
- since $v_i$ is the optimal cost, assigning parents as the algorithm progresses gives a shortest path from $s$ to $i$
- note that the only reason we need additional assumptions (compared with the simplest algorithm) is that we are terminating the search early
The closed set

- maintaining the set $E$ is optional
- the algorithm reduces to the previous one if we do not maintain $E$
- often $E$ is stored as a hash table, along with the values of $v$ in $E$
- if we remove from $E$ (in line 2), then $E$ and $F$ are always disjoint
- then (depending on the implementation) it may be easier to implement addition of elements to $E$ (in line 1)
Efficient implementation

- store $F$ as a *heap* providing insert, delete, and extract-min operations
- since we terminate early, we do not need to store $v_i$ for every vertex $i$
- store $v$ using a *hash table*, or keep values of $v$ with vertices
- implement set $E$ as a hash table
- neither hash tables nor heaps are available in Matlab
- in Matlab arrays are a workaround, but scale poorly
- if $\mathcal{V}$ is small, then we can mark vertices as open/closed in an array instead of maintaining sets/lists
Example: Two dimensional grid

- frontier $F$ shown in yellow
- closed set $E$ shown in blue