Shortest paths

Guillaume Dalle (ENPC, CERMICS)

REOP - Class 2 (29/09/2021)

Contents

Any feedback about the previous class?

1 Homework solutions

1.1 Exercise 3.3

In the adjacency matrix, $A_{uv} = 1$ iff there is an edge between u and v. Therefore,

$$A^2_{uv} = \sum_{w \in V} A_{uw} A_{wv} = \sum_{w \in V} \mathbf{1}\{ \text{ the path } u \to w \to v \text{ exists}\} = \text{ nb of paths } u \to w \to v$$

We can prove recursively that A^k_{uv} is the number of paths of length k from u to v.

1.2 Exercise 3.5

Suppose G is connected and undirected.

If G is a Eulerian graph, then it has a Eulerian cycle (containing every edge exactly once). Each vertex v appears in this cycle a certain number $k_v \geq 1$ of times (not 0). This means that $\deg(v) = 2k_v$.

If every vertex has even degree, we can construct the Eulerian cycle. Start from any vertex v_1 and iteratively pick uncrossed edges until you are stuck. You will necessarily be stuck in v_0 again because the even degree of the vertices implies that for every way in there is a way out. If there is a vertex u on the cycle that has uncrossed incident edges, start a new cycle from $v_2 = u$ and join it with the previous one. This allows us to cross all edges of the connected component.

1.3 Exercise 3.10

S be a stable iff

- no edge has both its endpoints in S
- every edge has at least one endpoint in $V \backslash S$
- $V \backslash S$ is a vertex cover

S is the largest stable set in G iff $V \backslash S$ is the smallest vertex cover. Hence $\alpha(G) = |S| = |V| - |V \backslash S| = |V| - \tau(G)$.

2 Shortest path problem

2.1 Statement

Input:

- a directed graph $D = (V, A)$
- a cost function $c : A \to \mathbb{Q}$
- two vertices o and d

Output: an $o \to d$ path P of minimum cost $c(P) = \sum_{a \in P} c(a)$ (or a proof that none exists because o and d are not connected)

We denote by $c(v)$ the cost of a shortest $o \to v$ path.

In cases where the problem is not well-defined (example: all weights are negative), we will ask for the shortest *simple* or *elementary* path instead.

2.2 Integer Programming formulation

Reminders

A Linear Program is an optimization problem with a linear objective and linear constraints:

$$\min_{x \in \mathbb{R}^n} c^\top x \quad \text{s.t.} \quad Ax \leq b \tag{LP}$$

A Mixed Integer Linear Program is a LP where some of the variables are constrained to be integers

$$\min_{x \in \mathbb{Z}^p \times \mathbb{R}^{n-p}} c^\top x \quad \text{s.t.} \quad Ax \leq b \tag{MILP}$$

In theory, solving a MILP is hard. However:

- they are very useful to model lots of real-life problems
- there are practically efficient solvers that can handle millions of variables if the problem has a certain structure

Formulation

Let x_{uv} be a binary variable equal to 1 if edge (u, v) is part of the path we choose, and 0 otherwise.

The shortest *elementary* path problem can be stated as:

$$\min_x \sum_{(u,v)\in A} c(u,v)x_{uv} \quad \text{s.t.} \quad x \in \{0,1\}^A \text{ defines an } o \to d \text{ path}$$

The constraint "x defines an $o \to d$ path" can be expressed linearly: if an vertex v different from o and d is visited, then it needs one incoming edge and one outgoing edge:

$$\forall v \in V, \quad \sum_{u\in\mathcal{N}^-(v)} x_{uv} - \sum_{w\in\mathcal{N}^+(v)} x_{vw} = \begin{cases} 0 & \text{if } v \neq o, d \\ -1 & \text{if } v = o \\ 1 & \text{if } v = d \end{cases}$$

2.3 Complexity

Theorem: The shortest path problem is NP-complete in the general case.

Proof:

- Reduction from Hamiltonian path to longest simple path
- Reduction from longest simple path to shortest simple path

Polynomial cases:

- Unweighted graph: Breadth-First Search
- Acyclic graphs:
  - Undirected: forest (at most 1 path between each pair)
  - Directed: topological sorting
- Nonnegative costs
  - Directed & undirected: Dijkstra's algorithm
- No negative / absorbing cycles
  - Directed: Bellman-Ford
  - Undirected: T-joints

Algorithm	Naive complexity	Best complexity
Topological sorting	$O(m+n)$	$O(m+n)$
Dijkstra	$O(n^2)$	$O(m+n\log n)$
Bellman-Ford	$O(mn)$	$O(mn)$

2.4 Dynamic programming

Underlying idea: Bellman's optimality principle

A subtrajectory of an optimal trajectory is itself optimal.

Generalize the problem to derive a recursion called the Bellman equation. Usually done by changing the bounds or adding parameters.

Knapsack example $\implies$ IP formulation, DP algorithm.

First compute the value of a solution. Is is usually easy to go back to the minimizer by working your way backwards.

2.5 Shortest path variants

Various flavors:

- Single source o, multiple destinations: the one studied here
- Single source o, single destination d: almost as hard
- Multiple sources, multiple destinations: harder

Extensions:

- Shortest paths with resource constraints (A. Parmentier's thesis)
- Multi-criteria shortest paths
- Shortest paths on transportation networks: timed trips, multiple modes (Bast et al. 2016)

2.6 Modeling examples

Exercise 5.12: disjoint intervals and profitable rentals

3 Solution algorithms

3.1 Directed graphs with no absorbing cycles

3.1.1 Bellman principle

Proposition 5.2:

Let P be an $o \to v$ path with k arcs and Q be an $o \to u$ path, where u is the vertex before v on P. If P is a shortest $o \to v$ path among those with k arcs, then Q is a shortest $o \to u$ path among those with $k-1$ arcs.

Proof:

- Suppose there is another $o \to u$ path Q' with $k-1$ arcs such that $c(Q') < c(Q)$.
- $P' = Q' \cup (u, v)$ is an $o \to v$ path with k arcs.
- P' has a cost $c(P') = c(Q') + c(u, v) < c(Q) + c(u, v) = c(P)$.

3.1.2 Bellman-Ford algorithm

The length of a shortest $o \to v$ path satisfies the Bellman equation

$$c(v, k) = \min_{u \in N^-(v)} c(u, k-1) + c(u, v)$$

with initial conditions

$$c(v, 0) = \begin{cases} 0 & \text{if } v = o \\ -\infty & \text{otherwise} \end{cases}$$

We can compute its values starting from $k = 0$. But when do we stop? Since D has no negative cycles, there is a simple shortest path of length at most $n - 1$. We compute $c(v, k)$ for all $v \in V$ and $k \in [0, n]$, and then pick k minimizing $c(d, k)$.

By remembering, for each v, the in-neighbor u that achieved the minimum, we can build a shortest-path tree.

3.2 Directed Acyclic Graphs (DAGs)

3.2.1 Bellman principle

Proposition 5.4:

Let D be a DAG, P be an $o \to v$ path ending with edge (u, v), and Q be an $o \to u$ path. If P is a shortest $o \to v$ path, then Q is a shortest $o \to u$ path.

Proof:

- Suppose there is another $o \to u$ path Q' such that $c(Q') < c(Q)$.
- $P' = Q' \cup (u, v)$ is an $o \to v$ path.
- P' has a cost $c(P') = c(Q') + c(u, v) < c(Q) + c(u, v) = c(P)$.

Recursive equation

The length of a shortest $o \to v$ path satisfies the Bellman equation

$$c(v) = \min_{u \in N^-(v)} c(u) + c(u, v) \quad \text{and} \quad c(o) = 0$$

Problem: in which order do we enumerate the vertices? The constraint is that we must compute $c(u)$ before $c(v)$ if there is an edge (u, v) in A.

3.2.2 Topological ordering

A digraph $D = (V, A)$ is acyclic iff there exists a total order $\preceq$ (i.e. a numbering of the vertices) such that $(u, v) \in A \implies u \preceq v$.

We define the operation `DFS`(v) (Depth-First Search) as follows:

1. open v (put in S)
2. scan its children
3. close v (put in L)

This recursive definition is consistent since the graph has no cycles.

If we add a dummy vertex which has all the "orphan" nodes as children, we can consider the case with only one orphan node o. Then, Algorithm 2 is equivalent to applying `DFS`(o).

Reversing the order in which vertices are closed yields a topological sort, since children are always closed before their parents.

3.3 Directed graphs with nonnegative costs

3.3.1 Dijkstra's algorithm

Pseudocode:

Input: a digraph $D = (V, A)$ and costs $c \in \mathbb{Q}_+^A$.

1. Set $U = \emptyset$ (set of visited vertices)
2. Set $\lambda(v) = 0$ if $v = o$ and $\lambda(v) = +\infty$ otherwise (initialize labels)
3. While $V \backslash U \neq \emptyset$:
   1. Choose $v \in V \backslash U$ such that $\lambda(v) = \min_{v' \in V \backslash U} \lambda(v')$ (choose the closest unvisited vertex according to the label)
   2. Add v to U (visit it)
   3. Set $\lambda(w) = \min\{\lambda(w), \lambda(v) + c(v, w)\}$ for all $w \in N^+(w)$ (update neighbor labels)

Output: the vector λ which contains all distances $o \to v$

Proposition: (Values of the tentative distance)

- For all $u \in U$, $\lambda(u) = c(u)$
- For all $w \in V \backslash U$, $\lambda(w) = \min_{u \in U} c(u) + c(u, w) \geq c(w)$

Proof: Make a drawing!

Since they hold after initialization, we must only check that these properties are preserved by the loop.

Let v be the closest vertex according to the tentative distance, i.e. the one achieving $\min_{v' \in V \setminus U} \lambda(v')$. By property 2, $\lambda(v) = \min_{u \in U} c(u) + c(u, v')$, so let u be the minimizer there.

Consider any other path from o to v. Let v' be its first vertex outside of U, and u' the one before that.

The path $o \rightsquigarrow u \rightsquigarrow v$ has cost $c(u) + c(u, v) = \lambda(v)$. The path $o \rightsquigarrow u' \rightsquigarrow v'$ has cost $c(u') + c(u', v') = \lambda(v') \geq \lambda(v)$. The remaining path $v' \rightsquigarrow v$ has strictly positive cost. Hence $o \rightsquigarrow v' \rightsquigarrow v$ is not better.

Therefore, $\lambda(v) = c(v)$ and the first property is preserved. The second property is easier to verify.

3.3.2 A* algorithm

Idea: speed up Dijkstra using a heuristic $h(v)$ to lower-bound the remaining distance $v \to d$.

Procedure: grow a set of paths and trim the ones that are hopeless.

Special case of Branch & Bound and LP duality.

Essential in transportation networks because the graphs are large but we have an idea of where to go.

3.4 Dynamic Programming for MDPs

See course notes

Talk about my internship at EDF on the optimization of cleaning schedules for photovoltaic solar panels

3.5 Exercises

Exercise 5.19: longest common subword

Exercise 5.20: Held-Karp algorithm for the TSP

References

Bast, Hannah, Daniel Delling, Andrew Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. 2016. "Route Planning in Transportation Networks." In *Algorithm Engineering: Selected Results and Surveys*, edited by Lasse Kliemann and Peter Sanders, 19–80. Lecture Notes in Computer Science. Cham: Springer International Publishing. https://doi.org/10.1007/978-3-319-49487-6_2.