Chapter 5

Combinatorial Optimization

Optimization is one area where discrete techniques are almost entirely different from continuous ones. We cannot simply set the first derivative of an objective function to zero. If the function is only defined over discrete values of variables, it is in general not differentiable. And even if we could perform some kind of real-valued relaxation, or perhaps an analytic continuation of the objective function, its minimum would not necessarily map to any valid solution of the original optimization problem. Having discrete or integer constraints on the variables can change the whole nature of the problem.

We will consider a number of combinatorial, or discrete, optimization problems on graphs that are central in discrete modeling. All of these problems involve *weighted* graphs. Weighted graphs are generalizations of the graph structures we saw previously, and are defined as follows.

Definition 5.1. *A* weighted *graph is a graph with a real-valued* weight $w(e)$ *assigned to each edge* $e \in E$*:*

$$w : E \to \mathbb{R}.$$

We often refer to weights $w(u,v)$, meaning the weight $w(e)$ where $e = \{u,v\}$ for an undirected graph or $e = (u,v)$ for a directed graph. In general, these weights can be positive, negative or zero. An example where they are restricted to nonnegative values is the following.

Example 5.2. Let $\vec{x}_1$ through $\vec{x}_n$ be the positions of n points in $\mathbb{R}^2$. Represent each point i by a vertex $v_i \in V$, and place an undirected edge $\{v_i, v_j\}$ between every pair of vertices $v_i \neq v_j \in V$. Let the weight $w(v_i, v_j)$ be the distance between $\vec{x}_i$ and $\vec{x}_j$.

Given a weighted graph, we correspondingly define a weighted path length.

Definition 5.3. *The weighted length of a path P is the sum of the weights of all edges in P.*

Let us turn to some specific problems. In certain cases, efficient algorithms exist to solve the problem exactly. In other cases, we resort to *heuristic methods* that give us approximate solutions.

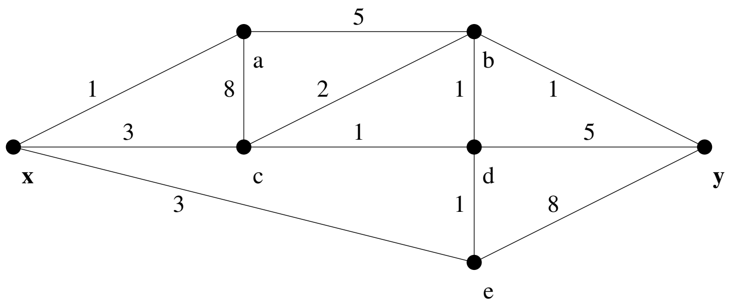
5.1 Exact algorithms

We will now study three classic combinatorial optimization problems. Although the problems seem difficult at first glance, they are computationally tractable using the right techniques. The problems, as well as the algorithmic techniques involved, turn out to be widely applicable in discrete mathematical modeling.

5.1.1 Shortest path

Given a weighted, undirected graph, how do we find a path of minimum weight from one vertex to another? Note that we often refer to this minimum-weight path as the "shortest path," even if the weights do not represent distances.

Example 5.4. Consider the following graph, with weights given next to the edges:



What is the path of minimum weight ("shortest path") between vertices x and y?

This kind of problem is clearly of interest in transportation and communication networks, not to mention software such as waze or google maps. (The homework problems give a few less obvious applications.) How can we solve it? It is clear by looking at Example 5.4 that if we were to search exhaustively among all paths from x to y, the complexity could be exponential in the number of edges.

In 1959, Edsger Dijkstra proposed the following approach. From the starting vertex x, consider a sequence of growing neighborhoods, and at each step keep track of the shortest possible distances to vertices seen so far, until vertex y is reached. (Again, by "shortest" and "nearest" we really mean weight, not distance.) This is efficient because, at step k, the shortest path to the kth-nearest neighbor can be at most one edge longer than the shortest paths to one of the vertices already considered in previous steps.

Specifically, Dijkstra's algorithm is as follows:

```
Given vertex x and y:
Initialize shortest weighted path lengths from x to all vertices:
  ∀v≠x ∈ V, l(v):=∞
  l(x)=0
Let v:=x

While v≠y, repeat:
  Mark vertex v as "visited"
  For each neighbor u of v that is not yet visited:
    Let l(u):=min{l(u),l(v)+w(v,u)}
  Let z be an unvisited vertex that minimizes l(z)
  Let v:=z
End
```

Note that z does *not* need to be a neighbor of v: it is simply the unvisited vertex that, according to what has been calculated so far, appears to be the closest to x. (If more than one unvisited vertex z minimizes $l(z)$, any one of them can be picked.)

Example 5.4 illustrates how the algorithm works.

- Visit x:
  $l(a) := \min\{l(a), l(x) + w(x,a)\} = \min\{\infty, 0 + 1\} = 1$
  $l(c) := \min\{l(c), l(x) + w(x,c)\} = \min\{\infty, 0 + 3\} = 3$
  $l(e) := \min\{l(e), l(x) + w(x,e)\} = \min\{\infty, 0 + 3\} = 3$

- Visit a:
  $l(b) := \min\{l(b), l(a) + w(a,b)\} = \min\{\infty, 1 + 5\} = 6$
  $l(c) := \min\{l(c), l(a) + w(a,c)\} = \min\{3, 1 + 8\} = 3$

- Visit c:
  $l(b) := \min\{l(b), l(c) + w(c,b)\} = \min\{6, 3 + 2\} = 5$
  $l(d) := \min\{l(d), l(c) + w(c,d)\} = \min\{\infty, 3 + 1\} = 4$

- Visit e:
  $l(d) := \min\{l(d), l(e) + w(e,d)\} = \min\{4, 3 + 1\} = 4$
  $l(y) := \min\{l(y), l(e) + w(e,y)\} = \min\{\infty, 3 + 8\} = 11$

- Visit d:
  $l(b) := \min\{l(b), l(d) + w(d,b)\} = \min\{5, 4 + 1\} = 5$
  $l(y) := \min\{l(y), l(d) + w(d,y)\} = \min\{11, 4 + 5\} = 9$

- Visit b:
  $l(y) := \min\{l(y), l(b) + w(b,y)\} = \min\{9, 5 + 1\} = 6$

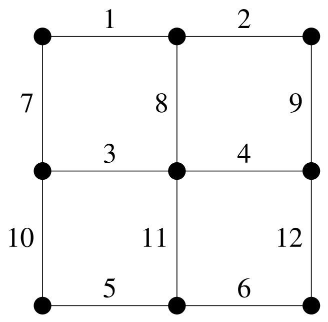
- Visit y:
  END

What is the complexity of Dijkstra's algorithm? Each time we visit a new vertex, the number of operations performed scales linearly with the number of new edges starting from that vertex. So the overall complexity is $O(|E|)$. If $n = |V|$, this means that $f(n) = O(n^2)$. As as we saw with the spanning tree algorithm in the previous chapter, in graphs of practical interest, vertices often have a constant number of neighbors, making the complexity of the algorithm linear. In some cases, this is not true: imagine a google maps graph where geographical locations have edges connecting to *all* other geographical locations, with edge weights given by distance. In practice, however, limiting a vertex's neighborhood to (say) its 10 or 20 closest neighbors will often give an excellent approximation.

5.1.2 Minimum spanning tree

We have seen how an algorithm can find a spanning tree of an unweighted graph, or show that none exists. But given a *weighted* (undirected) graph, how can we find a *minimum spanning tree* (MST): one where the sum of all edge weights is minimized?

Example 5.5. Imagine that we need to create a cable network connecting the 9 nodes shown below. The weight of an edge (shown next to each edge) is the cost of connecting the cable between the edge's two endpoints.

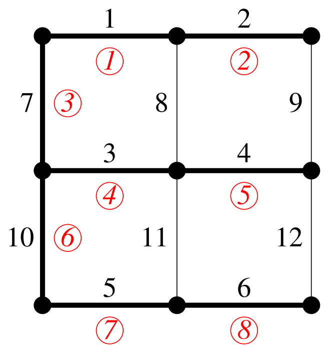


What is the cheapest way of putting together the network?

The following algorithm for finding a minimum spaning tree T was developed in 1930 by Vojtech Jarnik. It was independently developed by Robert Prim in 1957, and has come to be known as Prim's algorithm.

```
Initialize T to contain only the smallest-weight edge
Repeat:
  Find remaining smallest-weight edge e with one endpoint in T and one not in T
  If no such edge exists, STOP
  Let T:=T ∪ e
End
```

In our example, this algorithm would build T in the order shown by the circled red numbers below:



The optimality of this construction is not completely obvious, but is shown in the following lemmas and theorem.

Lemma 5.6. *Given a weighted graph, every step of Prim's algorithm generates a tree.*

Proof. Consider a step of Prim's algorithm. If adding edge e to T ever resulted in a cycle, both endpoints of e would have had to be in T. Therefore, by construction, T must always be a tree. □

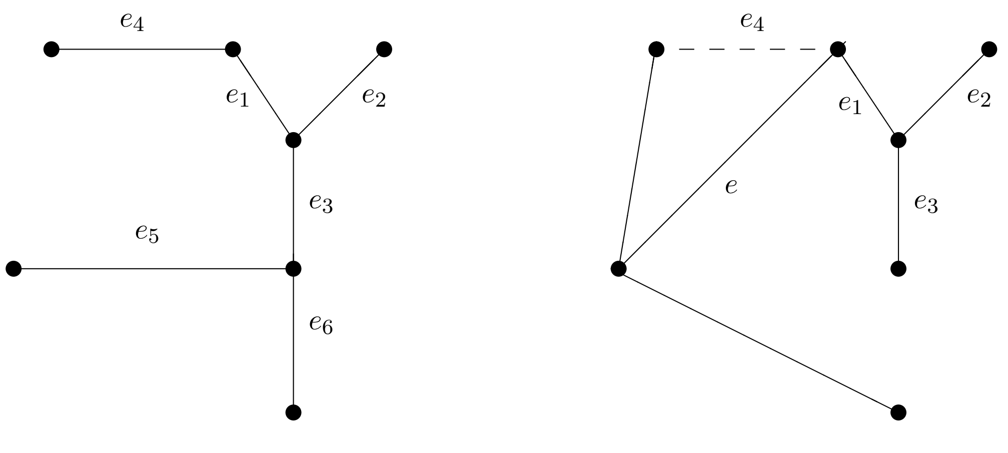
Lemma 5.7. *Given a connected, weighted graph, Prim's algorithm returns a spanning tree.*

Proof. From the previous lemma, Prim's algorithm generates a tree T. Now proceed by contradiction. If tree T were not a spanning tree, it would contain certain vertices $V_1 \subset V$ such that there exists a nonempty set $V_0 = V \setminus V_1$ outside of T. But once the algorithm terminates, there can be no more edges with one endpoint in V_0 and one endpoint in V_1. Therefore, there can be no path from V_0 to V_1, and the graph would have to be disconnected, violating the conditions of the lemma. Hence, T must be a spanning tree. □

Theorem 5.8. *Given a connected, weighted graph, Prim's algorithm returns a MST.*

Proof. Let $n = |V|$. From the lemma above, Prim's algorithm must return a spanning tree T, hence a tree with $n-1$ edges. Let e_k be the edge placed by the kth step of Prim's algorithm, for $k \leq n-1$. Let $T_k \subseteq T$ be the tree formed at the kth step, containing edges $e_1, \ldots, e_k$.

Now proceed by contradiction, assuming that spanning tree T is *not* minimal. Consider the largest possible k such that T_k is a subgraph of a MST (clearly $k < n-1$, since T itself is not a MST), and let S be that MST. Then, since $e_{k+1} \notin S$, adding e_{k+1} to S forms a cycle. As long as $k > 0$, S must therefore contain a path from T_k to the other endpoint of e_{k+1}. In that case, there must exist another edge $e \in S$ that is adjacent to T_k, such that $e \notin T_{k+1}$. The situation is illustrated in the following example with $k = 3$.



T **S**

Let S' be a new tree, where edge e is replaced by e_{k+1}:

$$S' = S \cup e_{k+1} \setminus e.$$

Now $T_{k+1} \subseteq S'$. But since Prim's algorithm chose e_{k+1} and not e,

$$w(e_{k+1}) \leq w(e),$$

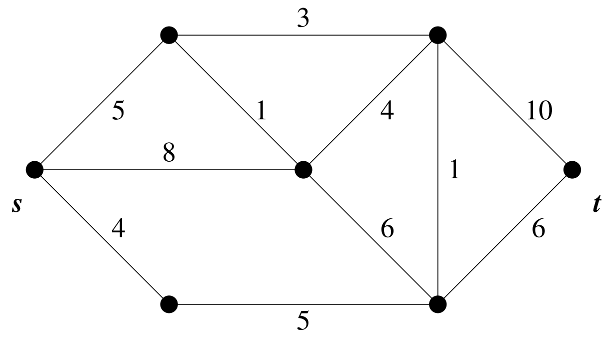
and so S' must also be a MST. However, we said that we picked the largest $k < n-1$ such that T_k is a subgraph of a MST, and now we find that T_{k+1} is a subgraph of a MST. (This is under the assumption that $k > 0$, but an almost identical argument applies if one assumes that $k = 0$.) So our initial assumption that T is not a MST leads to a contradiction. □

What is the complexity of Prim's algorithm? In the homework, you will show that $f(n) = O(n^3)$, but with a little more care, it is possible to bound the complexity by $O(|E| \log |V|)$. So once again, if the graph is not too dense, the algorithm runs in *almost* linear time.

5.1.3 Maximum flow

Finally, imagine that we have a network through which resources can be transported. We represent this by a weighted graph, where the weight of an edge denotes its capacity, meaning the maximum amount of the resource that can flow through it. What is the maximum flow through the network from a given source node to a given sink node, taking into account any possible bottlenecks along the way?

Example 5.9. Consider the following network and associated edge capacities:



What is the maximum possible flow through the network from source s to sink t? Clearly, the capacity at the source is limited by the sum of the capacities of edges incident on s: $5+8+4=17$. The capacity at the sink is limited by the sum of the capacities of edges incident on t: $10+6=16$. But is it really possible to have a flow of 16 all the way from source to sink?

In order to answer questions of this sort, we start with a few formal definitions.

Definition 5.10 (Flow). *Let $G=(V,E)$ be a simple graph, with* capacities $w\big(\{u,v\}\big) \geq 0\ \forall u,v \in V$*, where $w\big(\{u,v\}\big)=0$ if $\{u,v\} \notin E$. Then a* flow *from a given* source *vertex s to a given sink or* target *vertex t is a function $f(u,v)$, where*

$$\begin{aligned} f(u,v) &= -f(v,u) \\ |f(u,v)| &\leq w\big(\{u,v\}\big) \\ \sum_{w\in V} f(u,w) &= 0 \quad \text{unless } u=s \text{ or } u=t. \end{aligned}$$

Note that the capacities are weights, but they are defined for *any* unordered pair of vertices, with the capacity being zero if no edge is present there. Flows, on the other hand, are directed: $f(u,v)$ is positive if the flow is from u to v, and negative if the flow is from v to u.

The algorithm proposed by Ford and Fulkerson in 1956 is based on the idea of finding a *flow-augmenting path* from s to t.

Definition 5.11 (Remaining capacity). *The* remaining capacity *of a directed path P is the smallest difference between capacity and flow, over all (directed) edges in P:*

$$c = \min\big\{w\big(\{u,v\}\big) - f(u,v) : (u,v) \in P\big\}.$$

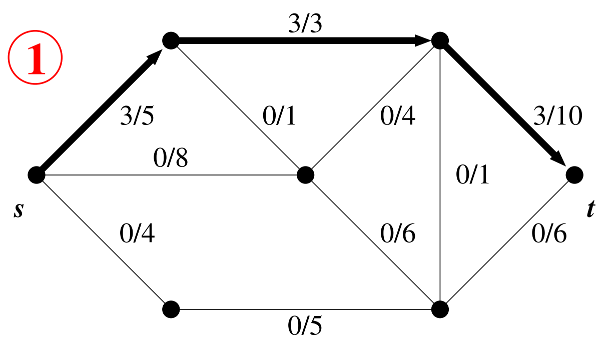
Note that since $|f(u,v)| \leq w\big(\{u,v\}\big)$, and all capacities are nonnegative, c must be nonnegative as well.

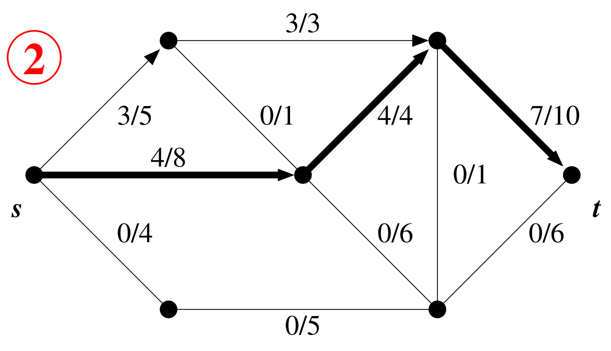
Definition 5.12 (Flow-augmenting path). *A directed path P is a* flow-augmenting path *if its remaining capacity is nonzero: $c>0$.*

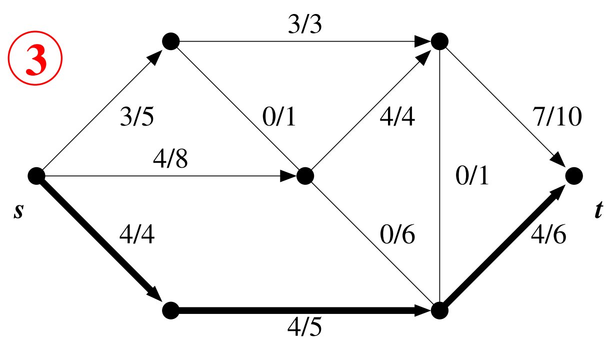
The Ford-Fulkerson algorithm then proceeds as follows.

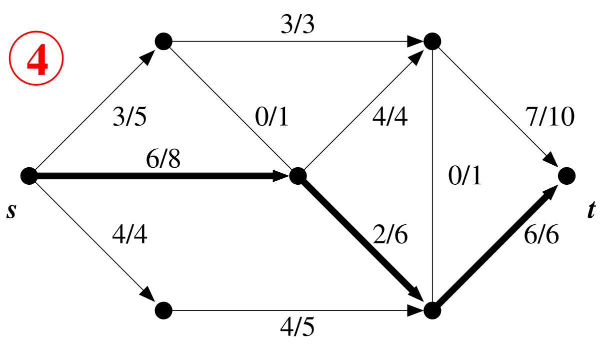
```
Initialize f(u,v)=0 for all edges {u,v}
Repeat:
  Find a flow-augmenting path P from s to t
  If so such path exists, STOP
  Augment flows f(u,v) on all directed edges in P by its remaining capacity c
End
```

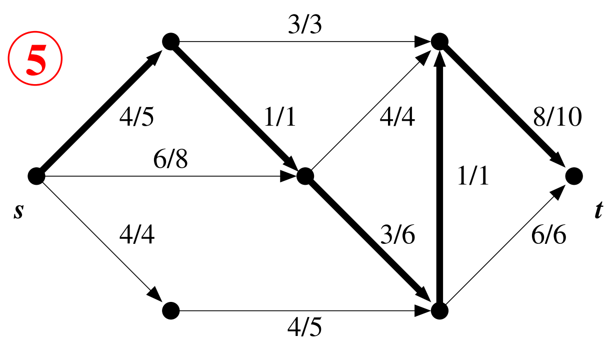
In our example, one possible sequence of flow-augmenting paths would be those in bold below, with associated flows and capacities shown next to each edge:











After the final step, no new flow-augmenting path exists, and the algorithm terminates. The resulting total flow F through the network, leaving s and arriving at t, is equal to 14.

How can we show in general that this algorithm is optimal, and how can we show that it is efficent? The following theorem addresses the first of these, under an important condition.

Theorem 5.13. *Given a graph $G = (V, E)$ with capacities $w\big(\{u, v\}\big)\ \forall u, v \in V$, the Ford-Fulkerson algorithm converges to the maximum flow $F_{\max}$ in finite time* if *all capacities are rational numbers.*

Proof sketch. First of all, we argue that if f is not a maximum flow, then there must still exist some flow-augmenting path. This holds by construction. If f is not a maximum flow, that means there is a way of modifying it so as to allow more resources to flow from s to t. But then there must be at least one path from s to t with remaining capacity (note that if for some u and v there is a positive flow $f(u, v)$, then there is always remaining capacity in the opposite direction $f(v, u)$).

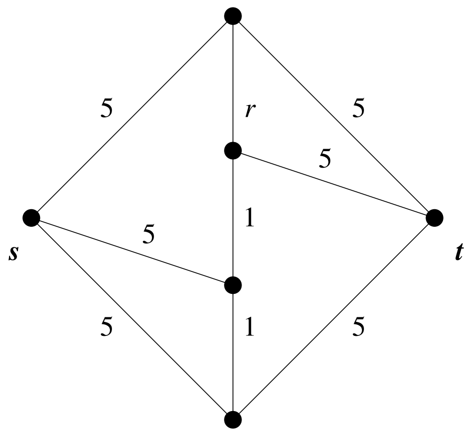
Of course, if flow-augmenting paths can cause flows to reverse direction along edges, it is conceivable that this algorithm will never converge. So now we argue that for rational-valued capacities, it will in fact converge in finite time. We do so by rescaling the problem, mutliplying all capacities by the product of their denominators. In the rescaled problem, the capacities are all integers, and each step of the algorithm must increase the rescaled flow (call it f') by at least 1. Thus, the algorithm must terminate in at most $F'_{\max}$ steps. By the previous argument, the maximum flow has now been found. □

Here is an example, from a paper by Uri Zwick in 1993, that shows what can happen when capacities are *irrational*. It also illustrates how a flow-augmenting paths can decrease a flow $f(u, v)$ that the Ford-Fulkerson assigned to edge $\{u, v\}$ in a previous step.

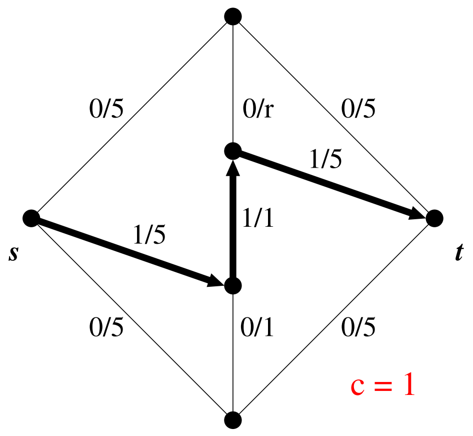
Example 5.14. Let r represent the golden ratio,

$$r = \frac{\sqrt{5} - 1}{2} \approx 0.618,$$

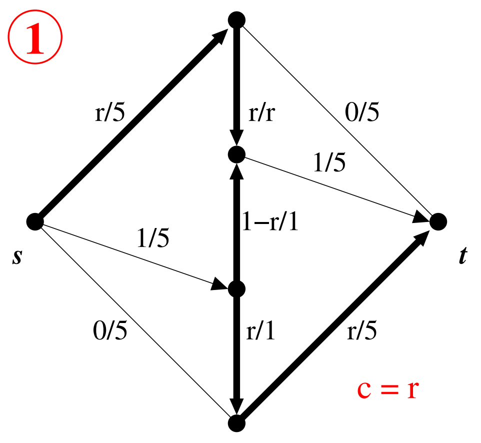
and consider a network with the following edge capacities:

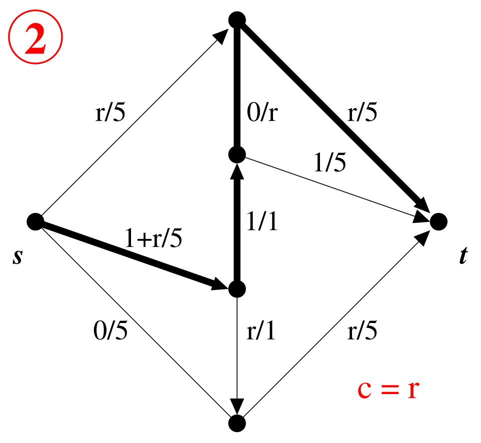


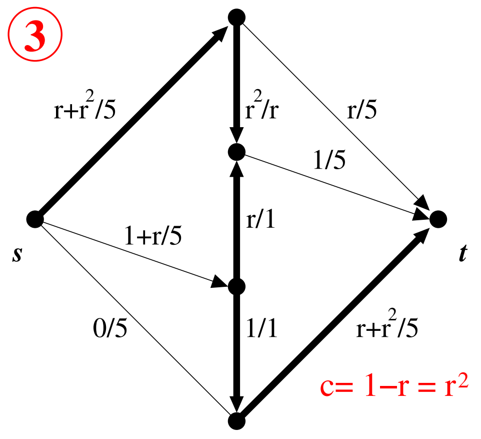
We could start with the following flow-augmenting path, with remaining capacity $c = 1$:

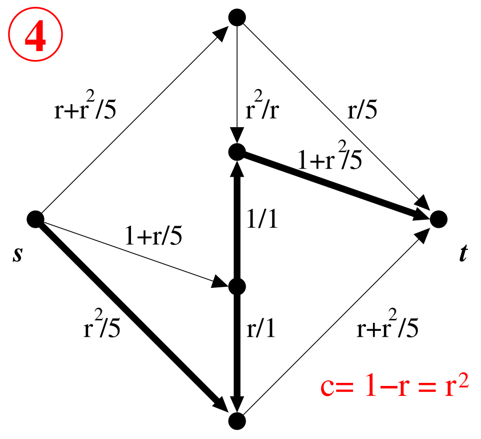


Then repeat the following sequence of flow-augmenting paths, with remaining capacities as shown.









Note that in the first step, by adding a flow of r in the downward direction of the middle edge, we are simply subtracting a flow of r from the upward direction.

It is straightforward to show that this sequence of four steps can be repeated forever, with the remaining capacity simply gaining a power of r at every two steps. Looking at the flow added at each step,

$$\begin{aligned} F &= 1 + r + r + r^2 + r^2 + r^3 + r^3 + \cdots \\ &= 2(1 + r + r^2 + r^3 + \cdots) - 1 \\ &= \frac{2}{1-r} - 1 \\ &= 2 + \sqrt{5} \approx 4.236. \end{aligned}$$

However, it is also clear by simple inspection that $F_{\max} = 11$. So in this case, not only does Ford-Fulkerson not converge in finite time, but it does not even approach the right value.

What about the complexity of Ford-Fulkerson? We have not mentioned at all *how* to go about finding a flow-augmenting path in a network. As it turns out, the method for doing that is quite similar to finding a shortest path, and has complexity of $O(|E|)$. Thus, if all edge capacities are rational, Ford-Fulkerson runs in $O(F'_{\max}|E|)$ steps.

A clever application of a path-finding algorithm, due to Edmunds and Karp (1972), improves on this further. The approach bounds the number of iterations by $O(|V| \cdot |E|)$, resulting in an overall complexity of $O(|V| \cdot |E|^2)$. So if $|V| = n$, while in the worst case the algorithm could take $O(n^5)$ time, in practice for sparse graphs the complexity is closer to $O(n^3)$.

There is another problem that turns out to be very closely related to maximum flow. Consider a weighted graph with source s and sink t. How can we cut the graph into two pieces, disconnecting source from sink, such that the sum of the weights of the cut edges is minimized?

Formally, we define a *cut* between two vertex subsets as follows.

Definition 5.15. *Let $G = (V, E)$ be a weighted simple graph, with a source and sink $s \neq t \in V$. A* cut *(S, T) is a partition of V into two subsets $S \subset V$ and $T \subset V$ such that*

$$V = S \cup T, \qquad S \cap T = \emptyset, \qquad s \in S, t \in T.$$

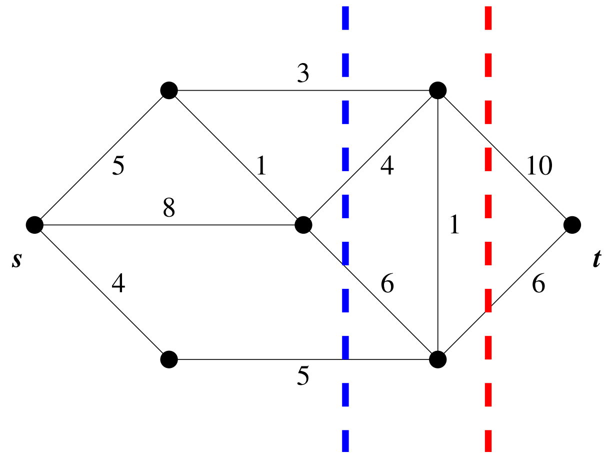
If the weights on the graph represent, say, the costs of cutting edges, then the *capacity* of a cut is its total cost.

Definition 5.16. *The* capacity *of a cut (S, T) is*

$$c(S, T) = \sum_{u \in S} \sum_{v \in T} w(\{u, v\}).$$

How can we find a cut (S, T) with minimal capacity?

If there is a total flow F through the network, there must exist some cut (S, T) with capacity $c(S, T) \geq F$, since flow can never exceed capacity. This can be seen, for instance, in the network in Example 5.9:



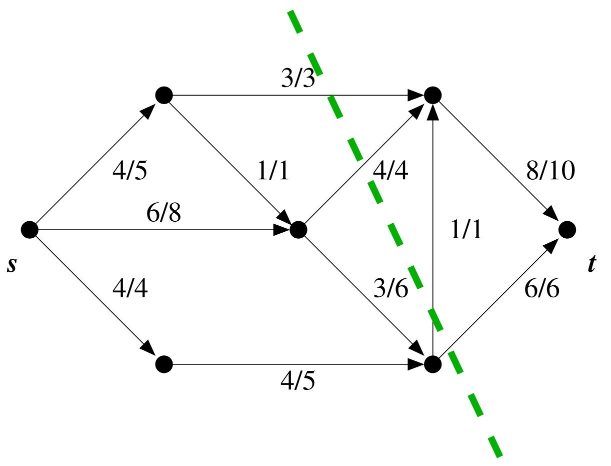
The blue dashed line shows a cut with capacity $c(S, T) = 18$, so there is no means of having a flow greater than $F = 18$. But the red dashed line shows a cut with capacity $c(S, T) = 16$, further bounding the flow to be no greater than $F = 16$. So clearly, the maximum flow must be a lower bound on the minimum cut.

The following theorem states that maximum flow and minimum cut are in fact exactly equal.

Theorem 5.17 (Max-flow equals min-cut)**.** *Let $G = (V, E)$ be a weighted simple graph, with a source and sink $s \neq t \in V$. If $F_{\max}$ is the maximum flow from s to t, and $c_{\min}$ is the capacity minimized over all cuts (S, T), then*

$$F_{\max} = c_{\min}.$$

While we will not prove the theorem, the intuition is clear: the maximum flow is determined by the smallest bottleneck in the graph, and that same bottleneck provides the minimum cut. In the network of Example 5.9, that bottleneck is the green dashed line shown below, through which all edges have flows equal to their capacities:



5.2 Heuristics

The combinatorial optimization problems we have so far considered are computationally tractable, in that algorithms exist to solve them in polynomial time. Furthermore, these algorithms are broadly applicable in modeling problems. But often, we are not so lucky.

Recall that certain decision problems are NP-complete, such as determining whether a graph has chromatic number k or less, or determining whether a graph has a Hamiltonian cycle. If an optimization problem is at least as hard as the hardest NP problem, we refer to it as *NP-hard*. For such problems, we may well have to settle for less than a perfect solution. We may prefer a method that will find an approximate solution and find it rapidly, rather than one that will find a genuinely optimal solution but exponentially slowly.

In order to demonstrate the use of heuristics in combinatorial optimization, we will consider one of the most famous NP-hard problems: the *traveling salesman problem* (TSP). Informally: given a certain number of "cities" and "distances" separating them, find the shortest "tour" that passes through each city exactly once and returns to its starting point. The more formal definition is as follows.

Definition 5.18 (Traveling salesman problem)**.** *Given a weighted,* complete *simple graph, find a Hamiltonian circuit minimizing the sum of the edge weights.*

The TSP is notable because, even though it is NP-hard, a wealth of efficient approximate methods have been developed for it. Using the TSP as an example, we will explore three different categories of heuristics: geometric approximations, greedy methods, and simulated annealing. Each one of these is very generally applicable to NP-hard combinatorial optimization problems.

5.2.1 Geometric approximations

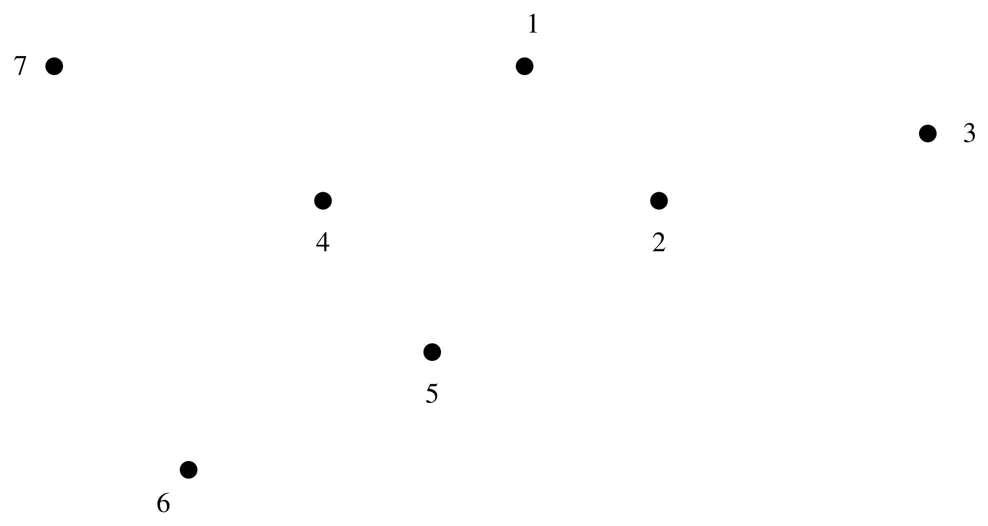
For the TSP, many cases of practical interest involve *metric* instances: the weights satisfy the triangle inequality. Clearly, this is always the case if the weights represent Euclidean distances. One of the earliest geometric approximation methods for the metric TSP was developed by Nicos

Christofides in 1976. The central idea of the Christofides algorithm is to approximate a TSP tour using a minimum spanning tree.

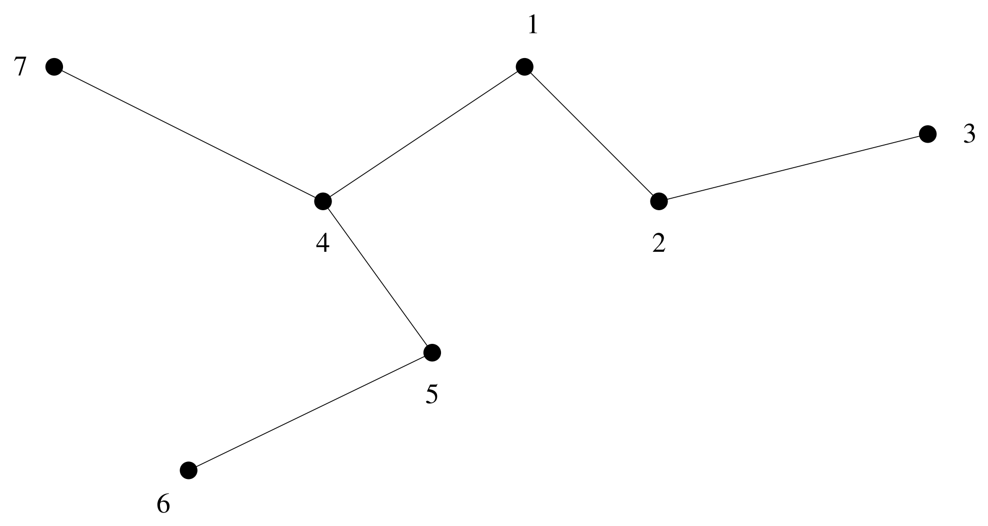
Consider a weighted, complete simple graph $G = (V, E)$, with weights satisfying the triangle inequality

$$\forall i, j, k \in V, \quad w(i, j) \leq w(i, k) + w(j, k).$$

For instance, in the following graph with 7 vertices, let $w(i, j)$ be the distance between vertex i and vertex j:

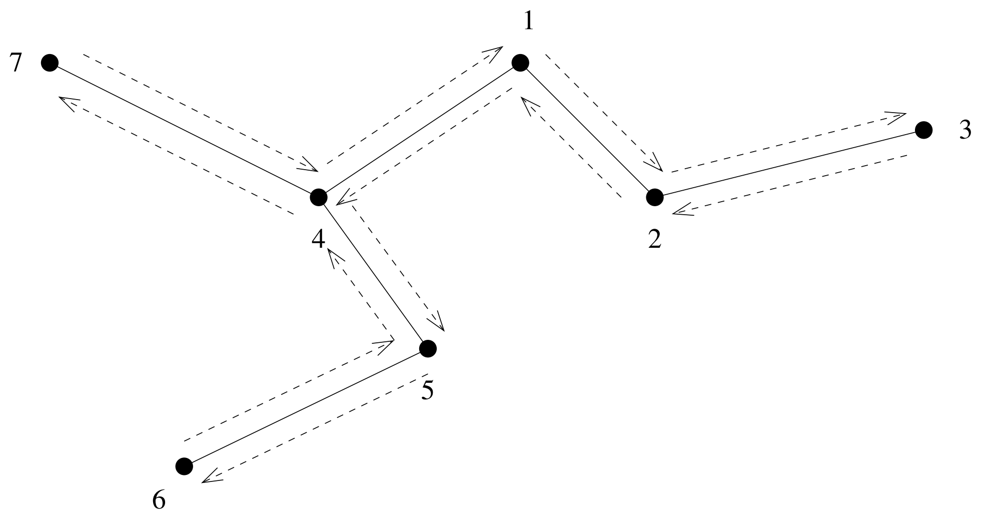


The first step in Christofides' algorithm is to find a MST T in the graph.



Call any vertex the "root" of the tree: in our example, let it be vertex 1. Then, starting from the root, construct a path P_0 that traces around the MST, using each edge exactly twice:

$P_0 :\quad 1\ 2\ 3\ 2\ 1\ 4\ 5\ 6\ 5\ 4\ 7\ 4\ 1$

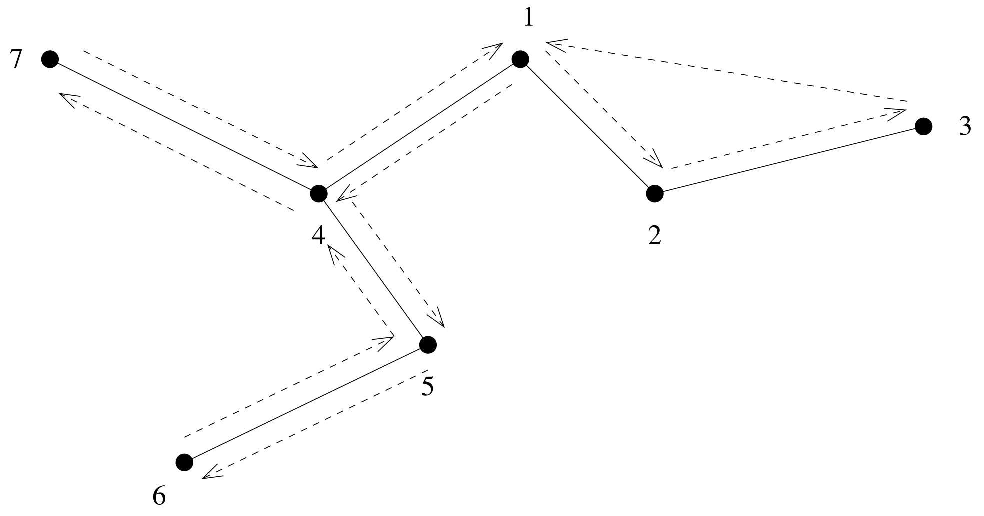


If $w(\mathrm{MST})$ is the total weight of the MST, then the total weight $w(P_0)$ of this path is simply

$$w(P_0) = 2w(\mathrm{MST}).$$

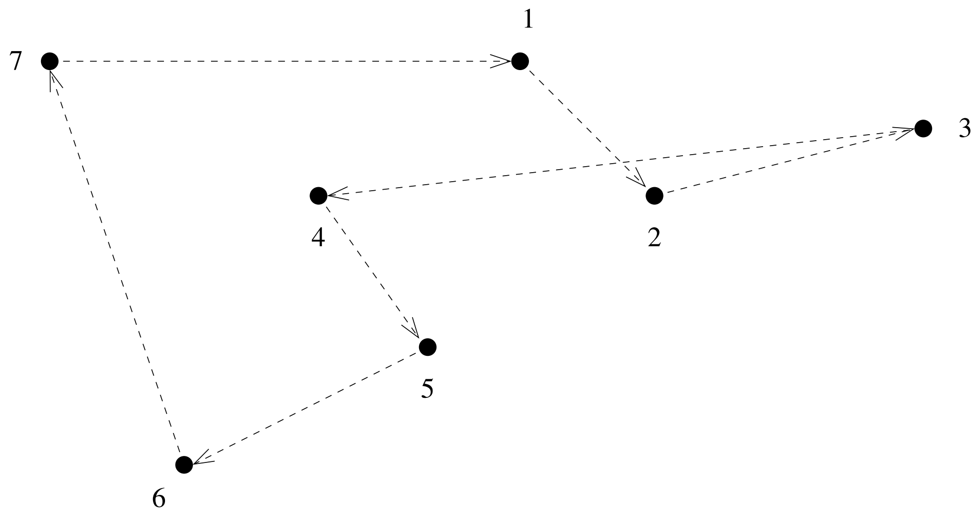
We now convert P_0 into a Hamiltonian circuit, by successively taking "shortcuts" that skip vertices already visited:

$$\begin{aligned} P_0 &: \quad 1\ 2\ 3\ 2\ 1\ 4\ 5\ 6\ 5\ 4\ 7\ 4\ 1 \\ \longrightarrow P_1 &: \quad 1\ 2\ 3\ \ 1\ 4\ 5\ 6\ 5\ 4\ 7\ 4\ 1 \end{aligned}$$



Continue in this way, until all possible shortcuts have been used.

$$\begin{array}{rl}
P_0: & 1\,2\,3\,2\,1\,4\,5\,6\,5\,4\,7\,4\,1 \\
\longrightarrow P_1: & 1\,2\,3\quad\,1\,4\,5\,6\,5\,4\,7\,4\,1 \\
\longrightarrow P_2: & 1\,2\,3\qquad\,4\,5\,6\,5\,4\,7\,4\,1 \\
\longrightarrow P_3: & 1\,2\,3\qquad\,4\,5\,6\quad\,4\,7\,4\,1 \\
\longrightarrow P_4: & 1\,2\,3\qquad\,4\,5\,6\qquad\,7\,4\,1 \\
\longrightarrow P_5: & 1\,2\,3\qquad\,4\,5\,6\qquad\,7\quad 1
\end{array}$$



Due to the triangle inequality,

$$w(P_5) \le w(P_4) \le w(P_3) \le w(P_2) \le w(P_1) \le w(P_0) = 2w(\text{MST}).$$

Furthermore, P_5 is now a Hamiltonian circuit, whose weight cannot be any less than the true optimal TSP solution $w(\text{TSP})$, so we have bounds

$$w(\text{TSP}) \le w(P_5) \le 2w(\text{MST}).$$

Finally, notice that if we delete any one edge from the optimal TSP solution, we obtain a spanning tree, so necessarily

$$w(\text{MST}) < w(\text{TSP}).$$

It therefore follows that

$$w(\text{TSP}) \le w(P_5) < 2w(\text{TSP}),$$

meaning that P_5 is *guaranteed* to be within an approximation factor of 2 of the optimal TSP solution.

While a factor of 2 may seem large, the fact that this holds for *any* graph, of *any* size, makes it quite a strong result. There are many cases of combinatorial optimization problems where no heuristic can produce a constant-factor guarantee (the *nonmetric* TSP is in fact an example). Furthermore, a more careful form of the Christofides heuristic actually leads to an approximation factor of 3/2: TSP tours that are at worst 50% longer than optimal.

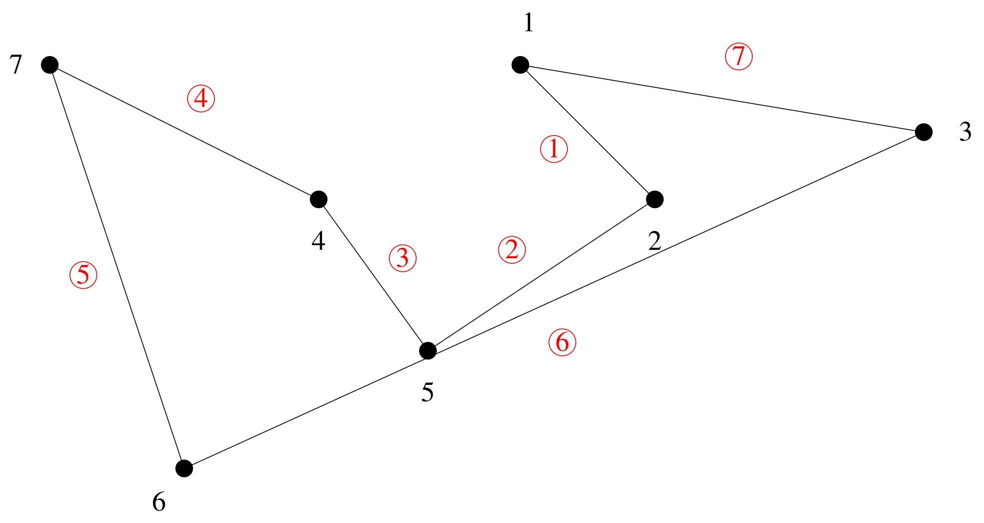
The main work in this algorithm is calculating the MST. Thus, the complexity of the Christofides heuristic is $O(|E| \log |V|)$.

5.2.2 Greedy methods

We have seen a few examples, so far, of algorithms that make an extreme choice at each step. For instance, Prim's algorithm for finding a MST always picks the edge of *smallest* weight that is adjacent to the current tree. Such methods are called "greedy".

For the TSP, one of the simplest greedy algorithms is the *nearest-neighbor* heuristic, which is actually quite similar to Prim's algorithm. Starting from an arbitrary vertex, add an edge to its nearest neighbor. Then move to that new neighbor, and add an edge to its nearest neighbor not already used in the tour. Repeat in this way, until we arrive at the nth edge and are forced to reconnect to the starting vertex.

Let us apply this method to our 7-vertex example, starting at vertex 1. The result is shown below, with circled red numbers indicating the order in which edges are added.



On metric TSP instances, this method also leads to an approximation guarantee, but a non-constant one: the approximation factor is $(1+\log_2 n)/2$, and so it grows (slowly) in n. On nonmetric TSP instances, as with Christofides' method, the approximation can be arbitrarily poor.

Recall that, when we first discussed the analysis of algorithms, we considered the *average-case* performance of an algorithm. What happens if we look at the approximation factor resulting from the greedy method for average-case instances of the TSP? In the metric case, we might consider placing n vertices uniformly at random within a unit square, and look at how the greedy tour length compares with the optimal tour length when averaged over this ensemble. Experimentally, averaging over those cases, one finds that greedy does in fact give a constant-factor approximation: regardless of n, it is about 22% larger than optimal.

The average-case analysis of the nonmetric TSP is even more intriguing, because we can bound the average approximation factor analytically. Since there is no triangle inequality in the nonmetric case, we can assume that there are no correlations betweeen weights, and so they are simply i.i.d. random variables chosen from the same distribution as that of Euclidean distances between vertices in the unit square. From a simple consideration of order statistics, the expected distance from a vertex to its nearest neighbor out of m possible neighbors is given asymptotically (for large m) by

$$\mathbb{E}[w_{\mathrm{NN}}^{(m)}] \approx \frac{1}{2\sqrt{m}}.$$

But since weights are uncorrelated, it is not hard to see that the expected weight of the greedy tour for large n is simply

$$\begin{aligned}\mathbb{E}[w(\text{Greedy})] &= \sum_{m=1}^{n-1} \mathbb{E}[w_{\text{NN}}^{(m)}] \\ &\approx \sqrt{n},\end{aligned}$$

where the last step comes about by approximating a sum with an integral. (Note that we have omitted the final nth step of closing up the tour, but that would add at most constant weight to $\mathbb{E}[w(\text{Greedy})]$, and is negligible compared to $\sqrt{n}$ when n is large.) The heuristic solution is clearly an upper bound on the true optimal solution, so

$$\mathbb{E}[w(\text{TSP})] \leq \mathbb{E}[w(\text{Greedy})] \approx \sqrt{n}.$$

However, a very simple lower bound on the optimal solution comes from observing that any vertex is *at best* connected to its first and second neighbors. Again using order statistics, that calculation leads to

$$\mathbb{E}[w(\text{TSP})] \geq \frac{5}{8}\sqrt{n}.$$

It therefore follows that

$$\mathbb{E}[w(\text{TSP})] \leq \mathbb{E}[w(\text{Greedy})] \approx \sqrt{n} \leq \frac{8}{5}\,\mathbb{E}[w(\text{TSP})],$$

and so asymptotically, greedy gives solutions that are on average at most 60% above optimal, even for this nonmetric case.

5.2.3 Simulated annealing

So far we have been looking at *construction heuristics* for TSP tours. Simulated annealing, on the other hand, is a *modification heuristic*: it takes a feasible solution (generally not an optimal one), and iteratively modifies it to try to improve upon it.

Simulated annealing was developed by Scott Kirkpatrick, Daniel Gelatt and Mario Vecchi in 1983, and independently by Vlado Černy in 1985. It is a very general method in combinatorial optimization, broadly applicable in problems where more finely adapted heuristics might not be available. Even though this is hardly the case for the TSP, where some excellent heuristics have been developed over the years, we will still use the TSP to demonstrate how it works.

The idea of simulated annealing is as follows. Rather than trying to find an optimal solution directly, we sample among feasible solutions. Of course, we know of a very convenient way to do that: a Markov chain! But we do not want to sample uniformly. We want to sample preferentially, with higher weight assigned to better solutions. Recall the *Metropolis-Hastings* algorithm from Section 3.4.3: if for any state x we are given a set of allowable transitions to neighboring states $y \in N(x)$, with weights w_y, we construct a Markov chain with transition probabiliies

$$P_{xy} = \frac{1}{N_{\max}} \min\left(1, \frac{w_y}{w_x}\right)$$

for $y \in N(x)$, where

$$N_{\max} = \max_x |N(x)|.$$

In order to use this Markov chain to minimize some objection function $f(x)$, let the weight for state x be

$$w_x = \exp[-f(x)/T].$$

What is the parameter T, and why does w_x have this form? It comes from the physical analogy of a thermodynamic system, in thermal equilibrium with a large "heat bath" at fixed temperature T. Such a system will adopt a given "microstate" x (collective quantum state of the molecules in the system) with probability proportional to the *Boltzmann factor*

$$\mathbb{P}\mathrm{r}[\text{microstate } x] \propto e^{-E(x)/T},$$

where $E(x)$ is the potential energy of the system in microstate x.

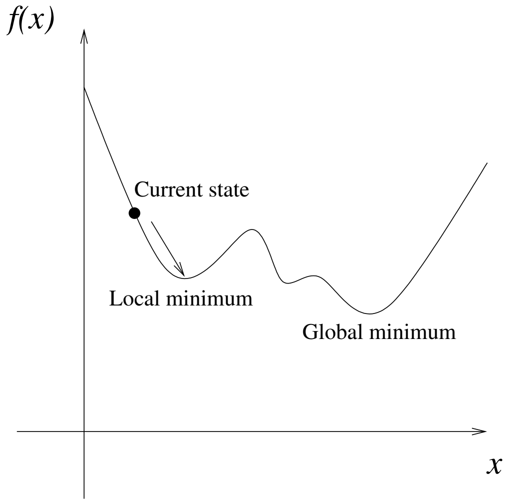
In the case of optimization, the "potential energy" is simply the objective function, and the "temperature" T controls how preferentially we sample:

- If T is high, all states x will have close to equal weight.
- If T is low, states x with low objective function values $f(x)$ will be favored.

Why can we not simply set $T = 0$ and force the Markov chain to sample only the optimal state? The difficulty is the mixing time. Since

$$P_{xy} = \frac{1}{N_{\max}} \min\left(1, \frac{w_y}{w_x}\right) = \frac{1}{N_{\max}} \min\left(1, e^{-[f(y)-f(x)]/T}\right)$$

for $y \in N(x)$, if T is close to zero, P_{xy} will be very small when $f(y) > f(x)$. The Markov chain will converge to its stationary distribution very slowly. It will almost only travel "downhill" to neighboring states with lower objective function values, thus performing what is essentially a greedy steepest-descent method. In the process, it will get trapped in a *local* minimum from which it will be unable to escape in any reasonable amount of time:



Simulated annealing solves this problem by sampling initially at high temperature, and then very gradually lowering the temperature. This is inspired by the process used in metallurgy, to bring a material into a crystalline state. The material is *annealed*, or cooled down very slowly, to prevent cracks or defects from forming. The *annealing schedule* determines how slowly the temperature is lowered.

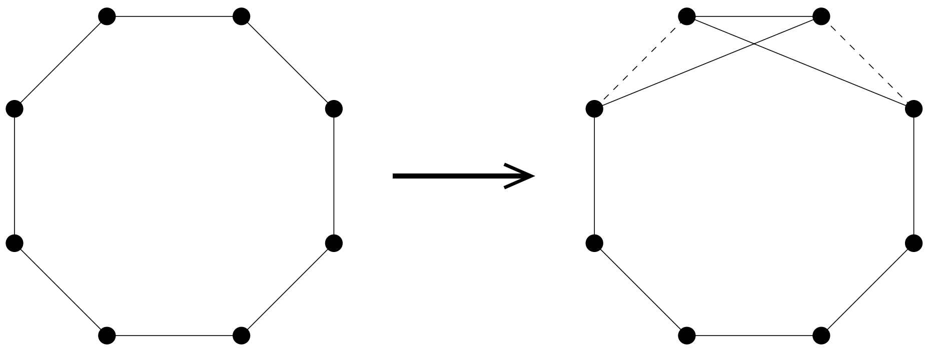
Notice that unlike the Markov chains we studied earlier, this one is *inhomogeneous*: its transition probabilities change over time, since T itself changes over time. Does this method find the global optimum? Amazingly enough, Bruce Hajek showed in 1988 that the answer is yes. . . but only if the temperature is lowered so slowly as to make the method hopelessly impractical. Specifically, if at time step t,

$$T = \frac{T_0}{\log(t+1)},$$

then as long as T_0 is sufficiently large and the Markov chain is irreducible (state space is connected), the optimal solution is guaranteed with probability 1. But this logarithmic annealing schedule takes so long that it is hardly better than using exhaustive search to solve the problem!

The real practical value of simulated annealing is as a heuristic. A common choice is to reduce T geometrically rather than logarithmically, leading only to an approximate solution. To illustrate one possible implementation of the algorithm, we return to the TSP:

- Let x be a feasible (though not necessarily optimal) solution: a tour, or Hamiltonian circuit.
- Let $f(x)$ be the length of the tour x.
- Let the neighborhood set $N(x)$ be all possible tours obtained by performing a *2-change* in x: removing two edges from the tour and then crossing those edges, as shown schematically below.



(Note that for the planar TSP, it is never optimal to have two edges crossing, but at least this neighborhood definition makes the resulting Markov chain irreducible.)

The simulated annealing procedure is then as follows:

1. Choose an arbitrary feasible solution x. Set the initial temperature T_0 such that the Metropolis-Hastings algorithm accepts approximately half of all *proposed* moves.
2. Perform m Markov chain steps at the current temperature, where m is large enough that the Markov chain “mixes”, i.e., its stationary distribution comes close to the Boltzmann factor. Generally, m should scale as $O(N_{\max})$, or $O(n^2)$ given the 2-change neighborhood. But in practice, one often takes $m = O(n)$ with a constant of a reasonable size, such as $m = 20n$.
3. Reduce temperature geometrically, multiplying it by a fixed parameter $\alpha < 1$ that is kept constant throughout the run (in practice, $\alpha = 0.95$ is often used). Thus the ith temperature in the schedule, set after im simulation steps, is $T_i = \alpha T_{i-1} = \alpha^i T_0$.
4. Repeat at step 2 as long as desired. In practice, one often runs until T is so low that any further improvement in unlikely.