Lecture Notes
"Discrete Optimization"

Bernd Gärtner
ETH Zürich

(Figures and Proofreading by Hiroyuki Miyazawa)

Contents

Chapter 1

Introduction

This chapter is intended to give you an idea what discrete optimization is, by showing you three problems from the area. You won't see any solutions here, only discussions of various aspects of the problems under consideration. This is quite natural, because a thorough understanding of a problem is a prerequisite for a solution. Also, I want to give you a chance to think about how *you* would solve the problems, before you learn how all the wise people in history have done it.

1.1 The Warehouse Problem – From an Ancient Competition

The following problem does not quite go back to the Greeks or the Romans, but in computer science categories, it is almost as old: it comes from the *5. Bundeswettbewerb Informatik - 2. Runde* (Germany, 1987). We are given a rectangular warehouse of dimensions $50m \times 80m$, partitioned into squares of $1m^2$ each (see Figure 1.1 for a small illustrating example of size $9m \times 9m$).

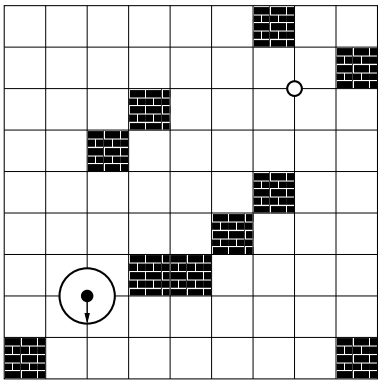


Figure 1.1: Warehouse with obstacles, initial robot position and goal position

Each square is either empty or completely occupied by an obstacle (walls, palettes, etc.). In the warehouse, we have a cylindrical robot, able to move along guide rails that separate the squares in a grid-like fashion. What we would like to have is a program that is able to compute a fastest robot trip from any given start position (intersection of guide rails) to any given goal position (small circle in Figure 1.1), along the guide rails. Because of the size of the robot, guide rails incident to an obstacle cannot be used. Also, there are no guide rails along the boundary of the warehouse.

In addition to its current position, the robot has a forward direction (N(orth), S(outh), E(ast), or W(est)) in which it can travel. At intersections, it may turn and continue in a different direction. Here is the exact list of control commands the robot understands, where it needs exactly one second to execute a single command (in brackets you find the distance the robot travels during execution of each command).

Command	START	MOVE(x)
Action	Start moving in forward direction (0.5m)	Continue to move in forward direction ($x = 1, 2,$ or $3m$)
Precondition	standstill	moving
Postcondition	moving	moving

Command	STOP	TURN(dir)
Action	Slow down until standstill (0.5m)	Turn by 90^0 in direction dir ($+$ = counterclockwise, $-$ = clockwise) (0m)
Precondition	moving	standstill
Postcondition	standstill	standstill

As an example for a robot trip, consider the one drawn in Figure 1.2. It takes 23 seconds, after processing the following (boring) list of commands. Note that we require the robot to be in standstill at the goal position.

1.	START	7.	STOP	13.	STOP	19.	START
2.	STOP	8.	TURN(+)	14.	TURN(+)	20.	STOP
3.	TURN(+)	9.	START	15.	START	21.	TURN(−)
4.	START	10.	STOP	16.	MOVE(3)	22.	START
5.	MOVE(2)	11.	TURN(−)	17.	STOP	23.	STOP
6.	MOVE(2)	12.	START	18.	TURN(+)		

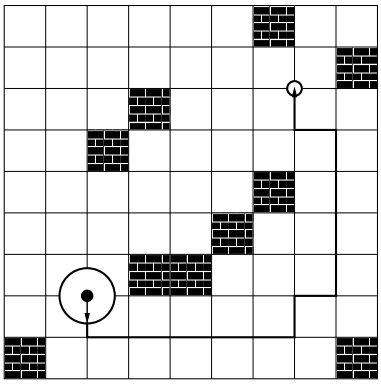


Figure 1.2: Robot trip of 23 seconds

1.1.1 Upper and Lower Bounds

What is the fastest possible robot trip from start to goal position in the warehouse of Figure 1.1? We don't know yet. What we know, however, is that the fastest trip will take at most 23 seconds. This is obvious, because we have already found a trip of that duration, so the fastest trip cannot take longer. We have found an *upper bound* for the duration of the fastest trip, simply by exhibiting a trip.

Of course, this was just some trip and we have no reason to believe that it is the optimal one. Staring at the warehouse somewhat longer, we indeed find a faster trip, taking only 19 seconds (see Figure 1.3; to find the corresponding command sequence is straightforward).

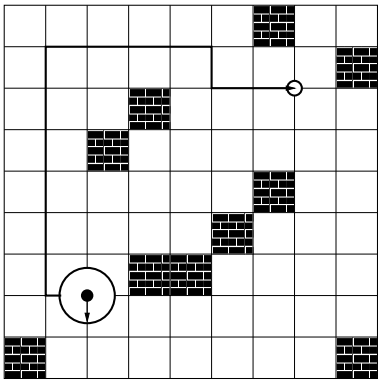


Figure 1.3: Robot trip of 19 seconds

Is this an optimal trip already? At least it seems difficult to improve it on the spot, so we get stuck with an upper bound of 19 for the moment. Already in the original problem statement, this trip was given as an example, together with the claim that this is indeed a fastest possible trip. In other words, the claim is that 19 is also a *lower bound* for the duration of the fastest trip – no trip can be faster. Are we going to believe this claim?

In trying to prove it, we are in a much more difficult situation than we were with the upper bound. For the latter, it was sufficient to exhibit some trip of duration 19. Now we need to argue that there exists no trip of duration smaller than 19. How can you do this, except by going through all possible trips? The situation is familiar in daily life: assume you have used somebody else's key but subsequently lost it in the building; now you are asked to find it. If you succeed, you simply give it back, but if you fail, you must argue that you have searched the whole building without any success.

Thus, proving existence (of the key in the building, or a trip that takes 19 seconds) is easy (once you have it), while proving non-existence (of the key in the building, or a trip that takes less than 19 seconds) can be quite difficult.

This of course, does not mean that *finding* the key (or a reasonably fast trip) is easy. For example, if the trip of Figure 1.3 indeed turns out to be optimal, we were simply lucky in finding it, but so far we have no systematic way of doing this in all cases.

Finding and proving upper and lower bounds is the key issue in discrete optimization, and there is a host of techniques to do this. Luckily, we don't need to go through all possible solutions in many cases. It turns out that most optimization algorithms *find* optimal solutions by going through an iterative process in which better and better upper and lower bounds are *proved*; as soon as the bounds coincide, an optimal solution has been discovered.

1.1.2 A Concrete Lower Bound

Unless we have an algorithm to solve the warehouse problem, we are forced to find a lower bound 'by hand', if we want to say something about how good our upper bound is. Recall that we are still not sure how far the 19-seconds-trip is from the truly fastest trip (unless we believe the unsubstantiated optimality claim by the designers of the problem). If for example, we could easily show that every trip takes at least 10 seconds, we would know that the trip we have found is at most twice as long as the optimal one. This might not sound like a great achievement in this case, but often reasonably good approximations of the optimal solution are sufficient in practice. However, the point is that we can only be sure to have such a reasonably good approximation, if we have a lower bound.

Here is such a hand-made lower bound for the warehouse problem.

Theorem 1.1 *If the start position has coordinates (i, j), with the robot pointing in direction South, the fastest trip to the goal position (k, ℓ) with $k > i, \ell > j$, takes at least*

$$6 + \left\lceil \frac{k-i-1}{3} \right\rceil + \left\lceil \frac{\ell-j-1}{3} \right\rceil$$

seconds.

Applied to Figure 1.1 with $(i, j) = (2, 2)$ and $(k, \ell) = (7, 7)$, we get a lower bound of 10 for the duration of the fastest trip. Thus we know that the fastest trip must take between

10 and 19 seconds. To verify the lower bound of 19, we will use the techniques of the next chapter.

Proof. The robot spends at least two seconds to START at (i, j) and to STOP at (k, ℓ). In addition, it needs to TURN at least once after it has left the start position and before it reaches the goal position (otherwise, it could only go along a straight line, never reaching the goal). This requires additional STOP and START commands, because turning requires a standstill. At least one more TURN is necessary, because the robot cannot reach direction North without turning at least twice. This already makes for 6 seconds in total. In order to travel the horizontal distance of $k-i$, the robot may use a START and STOP command already counted to go 1m. The remaining $(k-i-1)m$ cannot be traveled faster than in $\lceil (k-i-1)/3 \rceil$ seconds, because the robot's speed per second is limited to $3m$. The same applies to the vertical distance. ◻

In our example, the lower bound seems to be quite weak. However, there are scenarios in which this bound actually gives the exact duration of the fastest trip. Namely, in case there are no obstacles blocking the L-shaped path between start and goal position drawn in Figure 1.4), it is not hard to see that the commands shown to be necessary in the lower bound proof are also sufficient to carry out the trip. We say that the lower bound is *worst-case optimal*, because there are scenarios in which it cannot be improved.

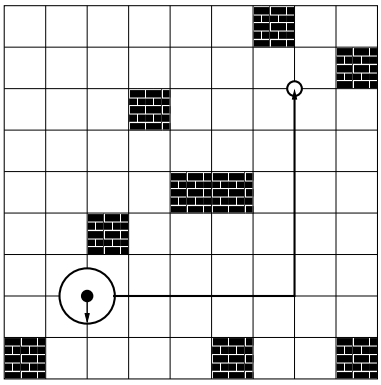


Figure 1.4: Robot trip whose duration matches the lower bound

1.1.3 A Mathematical Model

In order to solve a general instance of the warehouse problem (of size 50×80), we don't want to resort to guessing trips – we need an algorithm. The input to the algorithm must be a formal description of the warehouse, complete with obstacles, start and goal positions and the initial orientation of the robot. Moreover, the algorithm needs to know what the allowed moves of the robot are in any given position.

Such a formal description is also called a *mathematical model*. A suitable model in this case is a *directed graph*: the vertices are the possible *states* of the robot, given by

its coordinates and direction. Note that the information whether the robot is moving or in standstill can be deduced from its coordinates: we have standstill if and only if both coordinates are integer. The directed edges leaving a state correspond to the commands that are allowed in this state. What we need then is to find a shortest directed path in this graph between the start state s and some goal state t (there are four possible goal states, differing in the direction the robot has), see Figure 1.5.

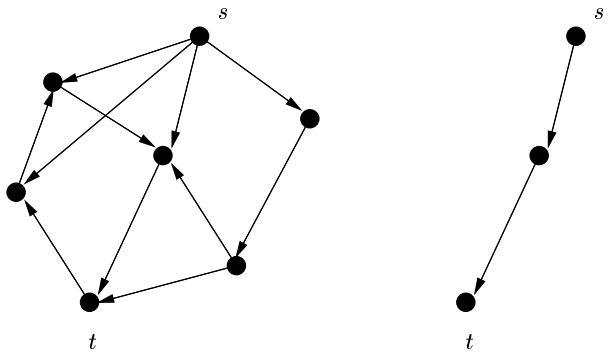


Figure 1.5: Directed graph and shortest directed path between s and t

Here is a more formal description of the graph $G = (V, E)$. The set of vertices (possible states of the robot) is given by

$$V = \{(i, j, dir) \mid 1 \leq i \leq 49, 1 \leq j \leq 79, dir \in \{N, S, E, W\}\}.$$

Here, the meaningful values of i and j are *half-integral*, i.e. of the form $t/2$ for some integer t. The edges are straightforward to find from the description of the commands. Let us just consider what happens in case of START and TURN(+). The START command gives rise to the edges

$$\begin{array}{lcll} (i,j,N) & \rightarrow & (i,j+1/2,N), & j \in \{1,\ldots,78\}, \\ (i,j,S) & \rightarrow & (i,j-1/2,S), & j \in \{2,\ldots,79\}, \\ (i,j,E) & \rightarrow & (i+1/2,j,E), & i \in \{1,\ldots,48\}, \\ (i,j,W) & \rightarrow & (i-1/2,j,W), & i \in \{2,\ldots,49\}, \end{array}$$

while TURN(+) induces

$$\begin{array}{lcll} (i,j,N) & \rightarrow & (i,j,W), & j \in \{1,\ldots,79\}, \\ (i,j,S) & \rightarrow & (i,j,O), & j \in \{1,\ldots,79\}, \\ (i,j,E) & \rightarrow & (i,j,N), & i \in \{1,\ldots,49\}, \\ (i,j,W) & \rightarrow & (i,j,S), & i \in \{1,\ldots,49\}. \end{array}$$

1.2 The Oracle of Bacon, or How Connected is the World?

Here, we consider another scenario which gives rise to the problem of finding shortest paths in a graph.

You may have heard claims of the form "Any two persons in the world are connected via at most 5 other people". What this means is that you know someone, who knows someone, who knows someone, who knows someone, who knows someone, who knows Ms. N. Obody from Grand Rapids, Michigan. Mathematically, we are talking about the structure of the graph G whose vertices are all the persons in the world, and there is an undirected edge between two persons if they know each other. (Of course, what it means to know each other has to be defined; it could mean, for example, that there has been a handshake, or eye contact, or some conversation,...) The above claim can then be rephrased as follows: in G, the shortest path between any two persons has length at most 6.

In practice, the claim cannot be verified, because the graph G is not known. But there is an interesting subgraph G_{movies} of G which is well understood: the vertices are movie actors and actresses, and there is an edge between two of them if they have appeared in a common movie. The *Oracle of Bacon* (`http://www.cs.virginia.edu/oracle/`) determines the shortest path in this graph between Kevin Bacon and any given actor or actress. The length of this path is the so-called *Bacon number* of the actor or actress. As it turns out, Bacon numbers greater than 4 are rare, meaning that G_{movies} is very well-connected.

It is important to note that a Bacon number output by the oracle is in general only an *upper bound* for the real Bacon number, because it is unlikely that the movie database the oracle is based on is complete. Lower bounds are harder (if not impossible) to prove, but for a different reason than in the warehouse problem. In the warehouse problem, we have complete information, and I have already indicated that there are efficient algorithms to compute true shortest paths (and therefore best possible lower bounds). Therefore, we have failed to come up with good lower bounds, just because we don't know the algorithms yet. The Oracle of Bacon, of course, knows and uses these algorithms, but still it subject to failure because of incomplete information.

For example, the oracle claims that Liselotte Pulver has Bacon number 2. The 'proof' is that she appeared with George Petrie in *Swiss Tour* (1949), while George Petrie appeared with Kevin Bacon in *Planes, Trains and Automobiles* (1987). Still, it is conceivable (although unlikely) that Liselotte Pulver has Bacon number 1, because the database does not contain that forgotten B-movie from 1960 in which 2-year old Kevin Bacon plays the bewitched son of Liselotte Pulver.

Still, the fact in common to the warehouse problem and the Bacon oracle is that upper bounds can simply be proved by example.

1.3 The Perfect Team – Planning for SOLA 2001

Suppose you have registered a team of 14 people for the *SOLA-Stafette*, the biggest annual running event in the Zurich area. What you still need to do is to assign each member of your team to one of the 14 courses. As you have plenty of time left, you decide to let everybody run all courses once, and note the respective runtimes. You end up with a table like 1.1.

Assuming that the runtimes are reproducible during the day the SOLA-Stafette takes place, you are looking for an assignment of people to courses that minimizes the total runtime.

As before, an upper bound is easily obtained from any assignment. For example, if you assign the members in the order of the table (Clara is running first, followed by Peter, then Heidi etc.), you get a total runtime of $9:57:10$, slightly below 10 hours. Again, a lower bound requires some arguing. Here is an argument: no matter how the assignment looks like, a course cannot be run faster than it would be run by the team member which is fastest on that particular course. This means, if we sum up the best runtimes for all 14 courses, we get a lower bound for the total runtime.

From the table we see that Heidi is the best runner on course 1, Barbara is the best on course 2, etc. Summing up all 14 best runtimes gives a lower bound of $8:26:11$. Why isn't this exactly the minimal total runtime that your team can achieve? It seems to make perfect sense to give every course to the team member that runs it fastest. The catch is that you would have to assign several courses to the *same* person: for example, Johanna is fastest on courses 8, 10 and 14, and also Ueli and Brunhilde are the fastest runners on more than one course. So your simple strategy does not work.

There are several mathematical models that would make sense in this situation, but as we have talked about graphs before, let us try to model this as a graph problem as well: the vertices are the team members and courses, and an edge connects any member with any course. This gives a complete bipartite graph. The additional feature is that each edge is labeled with a *cost* value, which is the runtime achieved by the member in running the course, see Figure 1.6

An assignment is any set of 14 edges which do not share a common endpoint. Such a set is also called a *matching*, and because all vertices participate in the matching, it is called a *perfect matching* in this case. The best assignment is then the *cost-minimal* perfect matching, where the cost of a matching is the sum of its edge costs.

1.4 Outlook

Already in the three problems we have discussed above, we have seen some crucial features of discrete optimization that we will discover again and again throughout the course.

- Discrete optimization deals with problems in which we have a *finite* (or at least countable) set of possible solutions, along with an *objective function* that assigns

Table 1.1: SOLA runtimes for all team members on all courses

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Clara	0:25:42	0:58:53	0:46:27	0:31:31	1:03:53	0:59:18	0:19:33	0:36:27	0:55:07	0:47:16	1:03:12	0:35:06	0:24:04	0:32:04
Peter	0:25:10	1:02:46	0:53:06	0:30:15	1:22:11	0:53:41	0:24:38	0:35:07	0:55:44	0:47:10	1:20:16	0:39:19	0:20:23	0:32:16
Heidi	0:19:20	1:05:01	0:35:00	0:25:45	1:14:44	0:55:26	0:22:35	0:30:24	0:51:28	0:42:43	1:00:27	0:36:37	0:22:34	0:28:09
Ueli	0:20:03	1:07:29	0:38:15	0:26:26	0:59:16	1:06:49	0:18:30	0:38:02	1:06:30	0:44:52	1:02:31	0:39:59	0:28:11	0:29:46
Barbara	0:23:46	0:52:43	0:37:22	0:27:39	1:16:57	0:54:00	0:28:32	0:33:07	1:00:37	0:50:37	1:14:31	0:36:06	0:22:36	0:31:18
Hans	0:19:32	1:05:12	0:32:31	0:24:29	1:11:12	0:53:59	0:25:46	0:32:04	0:59:20	0:41:53	1:18:14	0:35:27	0:32:31	0:28:13
Claudia	0:27:27	1:04:49	0:34:57	0:27:26	1:02:29	1:00:56	0:19:10	0:32:37	0:56:13	0:44:38	1:01:51	0:29:48	0:21:19	0:29:12
Ludovico	0:20:19	1:02:34	0:39:08	0:28:06	1:11:37	1:01:17	0:22:44	0:33:15	1:08:58	0:48:08	1:00:43	0:33:13	0:24:41	0:31:20
Brunhilde	0:27:17	1:04:55	0:36:05	0:26:15	1:15:12	1:05:03	0:24:59	0:32:15	0:46:00	0:52:31	0:52:42	0:38:26	0:24:53	0:28:30
Siegfried	0:23:11	1:06:32	0:32:22	0:30:46	1:14:10	0:53:31	0:27:06	0:32:32	0:58:15	0:49:11	1:15:41	0:33:34	0:27:19	0:32:12
Bernadette	0:21:22	1:11:32	0:50:48	0:34:34	1:14:18	0:57:40	0:21:40	0:32:32	0:58:15	0:49:11	1:15:41	0:33:34	0:27:19	0:32:12
Friedrich	0:26:38	1:02:01	0:40:57	0:31:20	1:07:53	0:52:44	0:30:29	0:32:32	0:58:15	0:49:11	1:15:41	0:33:34	0:27:19	0:32:12
Johanna	0:23:04	1:09:02	0:40:47	0:30:58	1:12:47	0:53:09	0:29:34	0:28:19	0:50:40	0:41:39	1:05:11	0:36:43	0:23:53	0:27:56
Wilhelm	0:24:34	1:03:35	0:36:54	0:33:17	1:15:15	0:53:22	0:22:32	0:32:22	0:54:34	0:41:42	1:25:38	0:39:55	0:29:27	0:35:36

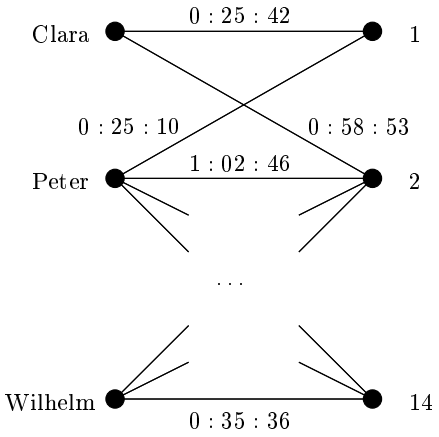


Figure 1.6: The SOLA graph

some value to every solution. The goal is to find the solution with smallest (or largest) objective function value.

In the warehouse problem, the possible solutions are all obstacle-avoiding trips of the robot from the start to the goal position, and the objective function is the duration of the trip. In case of the Oracle of Bacon, the solutions are all paths between Kevin bacon and a given actor/actress in the graph G_{movies}, and the objective function is the length of the path.

In the SOLA problem, the solutions are the perfect matchings, and the objective function is the sum of edge costs in the matching.

- For a theoretical understanding of a discrete optimization problem, but also for an algorithmic solution, bounds play an important role. For a minimization problem, upper bounds can easily be proved by exhibiting solutions, while lower bounds require more work in general. For maximization problems, it's just the other way around.

  In the warehouse problem (a minimization problem), we were able to prove an upper bound of 19 seconds for the duration of the fastest trip, by simply drawing a trip with this duration. We were not able to prove that 19 seconds is also a lower bound, and even the weaker lower bound of 10 seconds required some work. In the other two problems, upper bounds could be proved by example as well; lower bounds required more (and in each case different) insights into the problem. A major goal of the theory of discrete optimization is to develop techniques that are able to find or prove lower bounds for a large class of problems in the same unified way.

  Often, discrete optimization techniques can be interpreted as iterative methods that go through a sequence of solutions, and at the same time obtain better and better upper and lower bounds for the optimal solution. Once these bounds coincide, an optimal solution has been found.

- In order to attack any discrete optimization problem, either theoretically or algorithmically, an exact mathematical model is needed. Preferably, the problem should be reduced to a standard problem, for which solution methods are known.

  In the warehouse problem and for the Oracle of Bacon, the mathematical model is that of a directed (respectively undirected) graph, and the problem of finding the fastest trip (respectively, the shortest sequence of connecting movies) is reduced to the problem of finding a shortest path between two vertices in a graph. This is indeed a well-known (and well-solved) problem, as we will see in the next chapter.

  The problem of finding the cost-minimal perfect matching in a bipartite graph is a classic as well, and we will get to it in some later chapter.

Chapter 2

Shortest Paths in Graphs

In the introductory chapter, we have discussed the warehouse problem, and we have seen that it can be formulated as the problem of finding a shortest path from a start vertex to a set of goal vertices in a directed graph. Here, the length of a path is its number of edges.

This chapter discusses two algorithms, *breadth-first-search* and *Dijkstra's algorithm*, to find shortest paths in directed and undirected graphs. *Dijkstra's algorithm* actually solves the more

general problem of finding paths of minimum total weight in an edge-weighted graph.

2.1 Breadth-First Search

We are given a graph $G = (V, E)$. The breadth-first search algorithm (BFS) needs one data structure: a *queue* Q which maintains at each time a sequence

$$(v_1, \ldots, v_r)$$

of vertices. The queue can be initialized by entering a single vertex (we write $Q := \{s\}$ for this), and we can test it for emptyness. Moreover, it supports the following three operations.

head	(Q)	: returns the first element of Q
enqueue	(Q, v)	: makes v the new last element of Q
dequeue	(Q)	: removes the first element from Q

The algorithm BFS computes shortest paths from a source vertex s to all vertices that are reachable from s by some path. (In the directed case, reachability is defined via directed paths, of course.) For this, it performs a systematic graph search with the property that for any k, all vertices of distance k from s are found before any vertex of distance $k + 1$ is seen. This explains the term breadth-first search.

Upon termination, the entry $d[v]$ contains the length of a shortest path from s to v, and for all $v \neq s$, $\pi[v]$ contains the predecessor of v on some shortest path from s to v (or `NIL`, if no path exists).

The procedure BFS also marks a vertex when it is discovered first.

```
BFS(G, s):
    FOR u ∈ V \ {s} DO
        d[u] := ∞
        π(u) := NIL
    END
    mark s
    d[s] := 0
    Q := {s}
    WHILE Q ≠ ∅ DO
        u := head(Q)
        FOR all v adjacent to u DO
            IF v not marked THEN
                mark v
                d[v] := d[u] + 1
                π(v) := u
                enqueue(Q, v)
            END
        END
        dequeue(Q)
    END
```

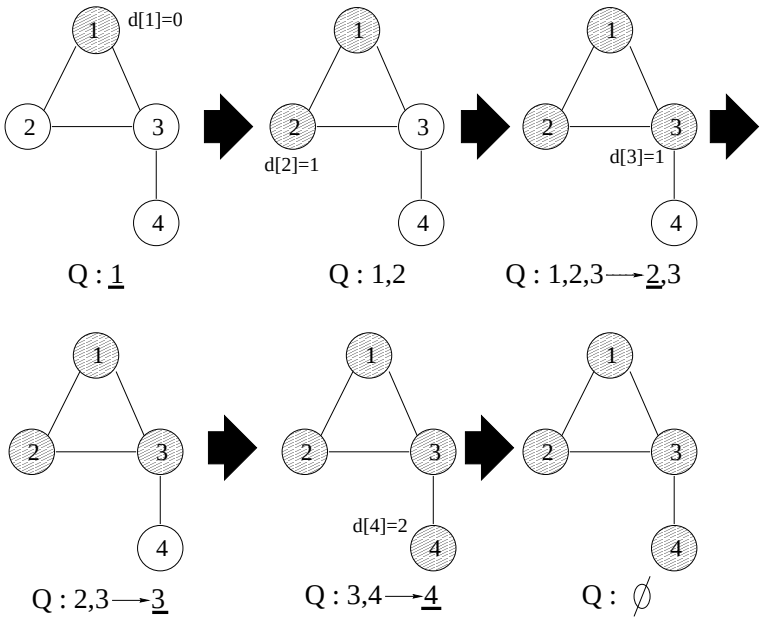


Figure 2.1: An easy example of BFS

Before we dive into the correctness proof, let us analyze the runtime of BFS. Every

vertex is enqueued at most once (directly after it has been marked), so there are at most $|V|$ iterations through the `WHILE` loop. Because the queue operations can be implemented in constant time per operation (we may maintain Q as a linked list, for example), we get $O(|V|)$ runtime in total for all operations outside the `FOR` loop. In th `FOR` loop, we process each pair (u, v) with v adjacent to u at most once. This means, we consider at most $|E|$ such pairs in the directed and $2|E|$ such pairs in the undirected case. The time spent per pair is constant, so we get $O(|E|)$ time for all operations in the `FOR` loop.

Lemma 2.1 *BFS takes $O(|V| + |E|)$ time.*

Now we want to prove that $d[v] = \delta(s, v)$ upon termination of the algorithm, where $\delta(s, v)$ denotes the length of the shortest path from s to v.

Fact 2.2 *For all v, $d[v] \geq \delta(s, v)$.*

The procedure BFS computes some path of length $d[v]$ from s to v, in reverse order given by the vertices

$$v, \pi(v), \pi(\pi(v)), \ldots, s.$$

This means, $d[v]$ is an upper bound for the length of the shortest possible path. You are of course right when you suspect that the path the algorithm computes is actually a shortest path. To prove this, we need to understand how the queue Q can look like at any point during the algorithm.

Lemma 2.3 *Assume that at some point, Q holds vertices $v_1, \ldots, v_r$ in this order. Then the following holds.*

(i) *there is a value $\ell \geq 0 (\ell \in)$ such that $d[v_i] \in \{\ell, \ell + 1\}$, for $i = 1, \ldots, r$.*

(ii) $d[v_i] \leq d[v_{i+1}], i = 1 \ldots, r - 1$.

This means, the queue looks like I have drawn it in Figure 2.2.

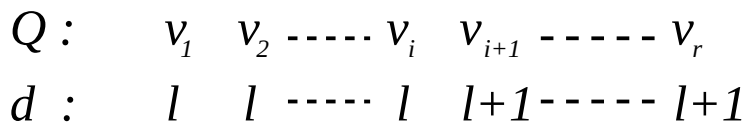


Figure 2.2: The queue

Proof. By induction over the number of enqueue and dequeue operations. Initially, we have $Q = \{s\}$, and the claims obviously holds. Now assume we are at some later stage in the algorithm, and Q satisfies the claims (i) and (ii), prior to an enqueue or dequeue operation. We distinguish the case where Q contains vertices with two distinct d-values from the case where only one value occurs. The effects of the queue operations in both cases are summarized in Figure 2.3, and this concludes the proof. □

Now we can prove the correctness of BFS.

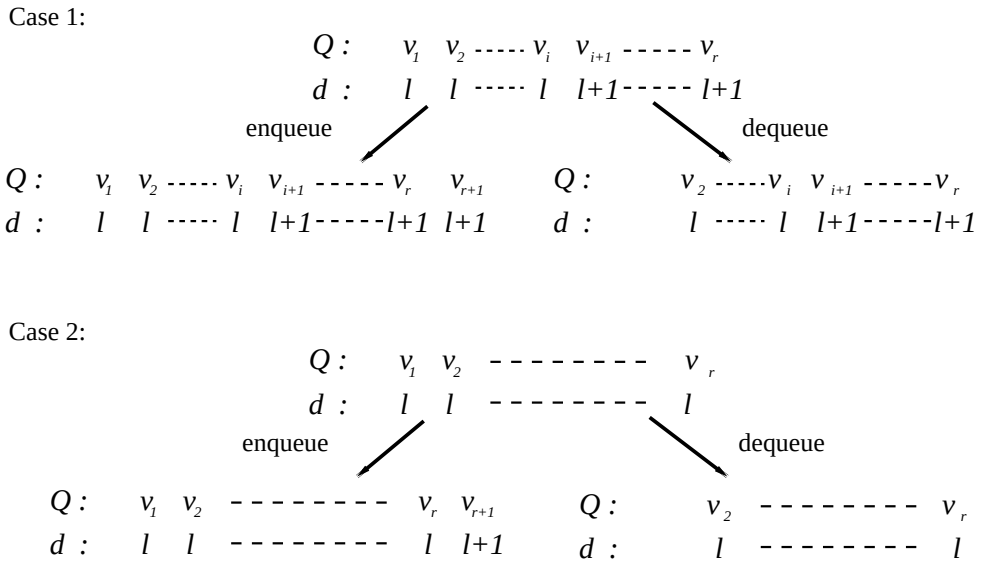


Figure 2.3: The effects of the queue operations

Theorem 2.4 *Upon termination of BFS, $d[v] = \delta(s, v)$ for all $v \in V$.*

Proof. We proceed by induction over $\delta(s, v)$. If $\delta(s, v) = 0$, then $v = s$ and $d[s] := 0$. Because the d-value of a vertex remains fixed once it is set, we also have $d[s] = 0$ at the end.

Now assume $\delta(s, v) = k < \infty$, and suppose the claim of the theorem holds for all vertices of smaller distance from s. There must be a vertex u such that (u, v) is an edge and $\delta(s, u) = k - 1$. (Any predecessor of v on a shortest path from s to v would qualify as the vertex u.) Let u_0 be the first such vertex which appears in Q. This must happen for any u under consideration, because we know by the inductive hypothesis that $d[u] = k - 1$, so its value has been set at some point, after which u was enqueued.

If v was not marked at the time where u_0 became head of Q, we are done, because then $d[v] = d[u_0] + 1 = k$. But is it possible that v is already marked? If so, v has been marked in processing some vertex u_1 such that (u_1, v) is an edge, and u_1 was head of Q earlier than u_0. From the lemma above about the structure of the queue it follows that $d[u_1] \leq d[u_0]$; in fact, we must have $d[u_1] = \delta(s, u_1) < \delta(s, u_0) = k - 1$, because u_0 was chosen to be the first neighbor of v with distance $k - 1$ to s that appeared in Q. Then, however, we get $d[v] = d[u_1] + 1 < k$, a contradiction to the fact that $d[v]$ is an upper bound for $\delta(s, k)$.

What if $\delta(s, v) = \infty$? Then there is no path from s to v, and the initial value $d[v] = \infty$ never changes. □

When we apply the BFS algorithm to find fastest robot tours in warehouse problem of the introductory chapter, we indeed obtain an efficient algorithm. For the sake of generality, let us assume the warehouse dimensions are $n \times k$ (we actually had $n = 50, k = 80$). The resulting graph model has $O(nk)$ vertices. Namely, we have assigned four states to every possible position of the robot, which is a pair (x, y) of coordinates such that

$2x \in \{2, \ldots, 2(n-1)\}$ and $2y \in \{2, \ldots, 2(k-1)\}$. Furthermore, there are also $O(nk)$ edges, because there is only a constant number of legal commands in a given state. This means, the resulting graph is *sparse*: asymptotically (in the O-notation), it has no more edges than vertices.

From our runtime analysis above, we conclude that BFS solves the warehouse problem in $O(nk)$ time. This is in a certain sense best possible, because the input description already has size $\Theta(nk)$ in the worst case. This means, we may need $\Theta(nk)$ time simply to read the input. As we have shown, we can solve the whole problem with the same asymptotic time bound.

2.2 Dijkstra's Algorithm

In general, shortest paths problems are defined over edge-weighted graphs, and we measure path lengths not in terms of the number of edges, but in terms of the sum of edge weights along the path (which we also call the weight of the path). In this case, BFS will obviously not work, see Figure **??**. We would have obtained such an edge-weighted graph from the warehouse problem if the robot commands would have had different execution times.

So let $G = (V, E)$ be a graph and consider a weight function $w : E \mapsto \mathbb{R}_0^+$ that assigns nonnegative weights to the edges. We could also allow negative weights, but then the simple algorithm we are going to present below does not work.

Just like BFS, the algorithm computes values $d[v]$ and vertices $\pi[v]$. Upon termination, $d[v]$ is the weight of the shortest path from the source vertex s to v, and $\pi(v)$ is a predecessor of v on some shortest path. Unlike in BFS, however, these values can change several times throughout the algorithm for a single vertex v. However, $d[v]$ will always be an upper bound for the weight of the shortest path.

Again, the algorithm requires an extra data structure, this time a *priority queue* Q, which maintains a set of vertices, sorted by their current d-values. It can be initialized by a set with their d-values ($Q := (V, d)$), and it can be tested for emptiness. In addition, it supports the following operations.

extract_min	(Q)	: removes the element with smallest d-value from Q and returns it
decrease_key	(Q, v, ρ)	: sets the d-value of v to ρ. *Precondition:* $\rho < d[v]$

Dijkstra's algorithm builds up a set S of vertices (initially empty), for which the shortest paths have already been found. This set grows by one vertex in each iteration.

Dijkstra(G, s)
  `FOR` all $u \in V \setminus S$ `DO`
    $d[u] := \infty$
    $\pi(u) :=$ `NIL`
  `END`
  $d[s] := 0$
  $S := 0$
  $Q := V$

```
WHILE Q ≠ ∅ DO
    u :=extract_min(Q)
    S := S ∪ {u}
    FOR all v adjacent to u DO
        IF d[v] > d[u] + w(u, v) THEN
            d[v] := d[u] + w(u, v)
            π[v] := u
            decrease_key(Q, v, d[v])
        END
    END
END
```

As before, let us do some runtime considerations. If we implement the priority queue simply as an array which we search and rearrange each time we extract the minimum, we have runtime $O(|V|)$ in each iteration, for all operations outside the `FOR` loop. This gives $O(|V|^2)$ in total. In the `FOR` loop, we process each pair (u, v) with v adjacent to u at most once. As in BFS, we therefore get $O(|E|)$ time for all operations in the `FOR` loop. For this, note that the decrease_key operation can be done in constant time with our array representation of Q.

We obtain an overall runtime of $O(|V|^2 + |E|)$. Using a more fancy priority queue, this can be reduced to $O(|V| \log |V| + |E|)$, still slower than BFS. This means, if we have an unweighted graph, BFS is the smarter algorithm, although we could of course use Dijkstra's algorithm after introducing artificial edge weights equal to one for all edges.

In particular, if the graph is sparse (like in the warehouse problem), BFS is a real saving; it runs in $O(|V|)$ times in this case, while Dijkstra's algorithm needs $\Theta(|V| \log |V|)$, even with the best priority queue.

Now we can prove the correctness of Dijkstra's algorithm.

Theorem 2.5 *At the time a vertex u is included in S, $d[u] = \delta(s, u)$, where $\delta(s, u)$ now denotes the weight of the shortest path from s to u.*

It follows that upon termination, all vertices have the correct d-values, because all vertices end up in S at some point, and d never changes for vertices in S.

Proof. By contradiction. Assume there exists some u with $d[u] \neq \delta(s, u)$ when u is added to S. Choose u_0 as the first vertex of this type to be included in S.

We have two easy facts.

(i) $u_0 \neq s$

(ii) there exists a path from s to u_0

(i) follows, because by construction, s is the first vertex to appear in S, at which point $d[s] = \delta(s, s) = 0$. The path claimed in (ii) must exist, because otherwise $\delta(s, u_0) = \infty = d[u_0]$ at the time u_0v is added.

Now consider the shortest path from s to u_0. Because $s \in S$ and $u_0 \notin S$ yet, this path must have a first vertex y with $y \notin S$. Consider its predecessor $x \in S$. We must have

$$d[x] = \delta(s, x),$$

because u_0 was the first vertex for which we have a contradiction when it gets added to S. Moreover,

$$d[y] = \delta(s, y).$$

To see this observe first that some shortest path from s to y goes through x (otherwise the path to u_0 through x could be shortened). This means that

$$\delta(s, y) = \delta(s, x) + w(x, y) = d[x] + w(x, y).$$

Moreover, in the course of adding x to s, $d[y]$ has been updated to $d[x] + w(x, y) = \delta(s, y)$ if it did not already have this value.

This implies that

$$d[y] = \delta(s, y) \leq \delta(s, u_0) \leq d[u_0], \tag{2.1}$$

because we have nonnegative weights, and because $d[u_0]$ is always an upper bound for $\delta(s, u_0)$ (using the same argument as in case of BFS). On the other hand, we have

$$d[u_0] \leq d[y], \tag{2.2}$$

because both y and u_0 are currently not in S, but u_0 was chosen from Q as the element with smallest d-value.

(2.1) and (2.2) together imply that

$$d[u_0] = \delta(s, u_0)$$

must hold, a contradiction to our assumption. □

Chapter 3

Linear Programming

3.1 The SOLA problem revisited

After we have seen in the last chapter how to solve the warehouse problem (and how to implement the Oracle of Bacon, if we have to), there is still one unsolved problem from the introduction: the SOLA problem.

Recall that in this problem, we have 14 runners and 14 courses, and we know the runtime of each runner on each course. The goal is to find an assignment of runners to courses which minimizes the total runtime.

We have modeled this problem as a minimum-weight perfect matching problem on a weighted complete bipartite graph $G = (V, E)$. In the general version of this problem, we have $2n$ vertices $V = L \cup R$, where $|L| = |R| = n$ and n^2 edges $E = \{\{v, w\} \mid v \in L, w \in R\}$. In addition, there is a weight function

$$\omega : E \mapsto \mathbb{R}.$$

where we abbreviate $\omega(e)$ by ω_e.

An optimal *assignment*, or a weight-minimal *perfect matching*, is a set of n edges M such that

- no two edges in M have a common vertex, and
- the sum of weights $\sum_{e \in M} \omega_e$ is minimized.

3.1.1 Integer linear programming formulation

In order to attack the problem, we reformulate in a seemingly more complicated way. For each edge e, we introduce a numerical variable x_e which can take on values 0 or 1.

Then we can encode the problem using these variables.

- A set of edges F corresponds to a 0/1-vector $\tilde{x}$ of length n^2, indexed with the edges, with the intepretation that

$$e \in F \quad \Leftrightarrow \quad \tilde{x}_e = 1.$$

- F corresponds to an assignment, if the following constraints are satisfied:

$$\sum_{e \ni v} \tilde{x}_e = 1, \quad \forall v \in V.$$

  This means, that for every vertex, exactly one incident edge is chosen.

- The weight of the assignment is given by

$$\sum_{e \in E} \omega_e \tilde{x}_e.$$

This means, the problem of finding the optimal assignment can be written as a *mathematical program* in n^2 variables, $2n$ equality constraints, and n^2 integrality constraints:

$$\begin{array}{lll} (\mathrm{ILP}_{Matching}(G)) & \text{minimize} & \sum_{e \in E} \omega_e x_e \\ & \text{subject to} & \sum_{e \ni v} x_e = 1, \qquad \forall v \in V, \\ & & x_e \in \{0,1\}, \qquad \forall e \in E. \end{array}$$

This mathematical program is actually an *integer linear program* (ILP). Linear, because both the *objective function* $\sum_{e \in E} \omega_e x_e$ as well as the left-hand sides of all constraints are linear functions in the variables. Integer, because there is an additional requirement that the variables only take on integer values. Let $\mathrm{opt}(\mathrm{ILP}_{Matching}(G))$ denote the minimum value of the objective function, hence the weight of the best assignment.

3.1.2 LP relaxation

What does this reformulation as in ILP buy us? A lower bound! Recall that for a minimization problem, an upper bound is easily proved by just presenting some solution, while lower bounds always require some work. For an ILP in minimization form, a lower bound is obtained from the solution of the so-called *LP relaxation*, given as

$$\begin{array}{lll} (\mathrm{LP}_{Matching}(G)) & \text{minimize} & \sum_{e \in E} \omega_e x_e \\ & \text{subject to} & \sum_{e \ni v} x_e = 1, \qquad \forall v \in V, \\ & & 0 \le x_e \le 1, \qquad \forall e \in E. \end{array}$$

This is a *linear program* (LP), because the integrality constraints have been relaxed to plain linear inequality constraints. Because all the vectors that satisfy the constraints of the ILP also satisfy the constraints of the LP, we get

$$\mathrm{opt}(\mathrm{LP}_{Matching}(G)) \le \mathrm{opt}(\mathrm{ILP}_{Matching}(G)).$$

The nice thing about an LP is that it is easy to solve, as we will see in the next section. This is not true for a general ILP – the integrality constraints usually make the problem difficult. Still, the lower bound given by the LP can easily be computed, and in quite a number of cases, this lower bound is a good approximation of the true optimal ILP value.

3.1.3 ILP versus LP

In the bipartite matching case, the situation is even nicer, and that's what makes the LP technique really useful here.

Theorem 3.1

$$\text{opt}(\text{LP}_{Matching}(G)) = \text{opt}(\text{ILP}_{Matching}(G)).$$

In other words, the LP does not only give us a lower bound, it gives us the optimum value of the assignment (and the proof below will show that we can also get the optimal assignment itself).

Proof. We will show that there is an optimal solution $\tilde{x}$ to the LP which has integer entries. This does it, because this solution must then also be an optimal solution to the ILP, and both optima coincide. To formulate it differently: the LP can have a better optimum value than the ILP only if all its optimal solutions $\tilde{x}$ have at least one fractional entry; and we are going to exclude this situation.

Let $\tilde{x}$ be some optimal solution to the LP with associated objective function value $c_{opt} = \sum_{e \in E} \omega_e \tilde{x}_e$. If all entries of $\tilde{x}$ are integer, we are done. Otherwise, there must be some $\tilde{x}_{e_1}$ strictly between 0 and 1. This entry corresponds to some edge $e_1 = \{v_1, v_2\}$. Because we have the property that

$$\sum_{e \ni v_2} \tilde{x}_e = 1,$$

there must be another edge $e_2 = \{v_2, v_3\}, v_3 \neq v_1$ such that $0 < \tilde{x}_{e_2} < 1$. Because of the same reason, we find a third edge $e_3 = \{v_3, v_4\}, v_4 \neq v_2$ such that $0 < \tilde{x}_{e_3} < 1$.

This continues, as we find fractional values of $\tilde{x}$ corresponding to edges on a longer and longer path $v_1, v_2, v_3, v_4, \ldots$. Because there are only finitely many vertices, we must finally reach a vertex we have already seen before. Without loss of generality, assume that this vertex is v_1. This means, we have found a *cycle* of edges

$$e_1 = \{v_1, v_2\}, e_2 = \{v_2, v_3\}, \ldots, e_t = \{v_t, v_1\},$$

such that $0 < \tilde{x}_e < 1$ for all $e \in \{e_1, \ldots, e_t\}$, see Figure 3.1.

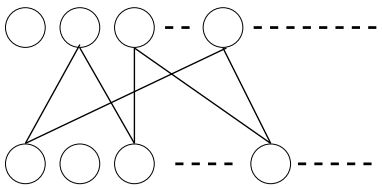


Figure 3.1: Cycle of edges with noninteger entries in $\tilde{x}$

Because we have a bipartite graph, the cycle has even length t. Let us for some small $\varepsilon > 0$ define

$$\tilde{x}'_e = \begin{cases} \tilde{x}_e - \varepsilon, & \text{if } e \in \{e_1, e_3, \ldots, e_{t-1}\} \\ \tilde{x}_e + \varepsilon, & \text{if } e \in \{e_2, e_4, \ldots, e_t\} \\ \tilde{x}_e, & \text{otherwise.} \end{cases}$$

For vertices v_i in this cycle, we have

$$\sum_{e \ni v_i} \tilde{x}'_e = \sum_{e \ni v_i} \tilde{x}_e + \varepsilon - \varepsilon = 1, \quad \forall i \in \{1, \ldots, t\}.$$

For vertices v not in the cycle, $\tilde{x}'_e = \tilde{x}_e$ for all incident edges, so

$$\sum_{e \ni v} \tilde{x}'_e = 1$$

holds for all vertices $v \in V$. Moreover, if ε is small enough, $\tilde{x}'$ still respects the constraints $0 \leq \tilde{x}'_e \leq 1$, because we have only changed values strictly between 0 and 1. This shows that $\tilde{x}'$ is a solution to the LP (a vector which satisfies all the constraints) again.

Moreover,

$$\sum_{e \in E} \omega_e \tilde{x}'_e = c_{opt} + \varepsilon \sum_{i=1}^{t} (-1)^t \omega_{e_i}.$$

Because c_{opt} was the smallest obtainable objective function value, we must have $\Delta := \sum_{i=1}^{t} (-1)^t \omega_{e_i} = 0$, otherwise we would get a smaller value by chosing $\varepsilon < 0$ if $\Delta > 0$, or $\varepsilon > 0$ if $\Delta < 0$. This means, $\tilde{x}'$ is still an optimal solution to the LP, for all ε that lead to a solution at all.

If we now choose ε to be the largest value such that $\tilde{x}'$ is still a solution, we get $\tilde{x}'_e \in \{0, 1\}$ for some $e \in \{e_1, \ldots, e_t\}$. This means, $\tilde{x}'$ has less fractional entries than $\tilde{x}$. Continuing in this way with $\tilde{x}'$, we finally obtain an optimal solution where all fractional entries have been eliminated. This solution is then also a solution to the ILP. ◻

3.1.4 ILP versus LP in other cases

We note that also the problem of finding shortest paths in weighted graphs can be formulated as an ILP whose LP-relaxation has an optimal solution with integer entries (exercise). However, this nice behavior is the *exception*, not the rule. Just to demonstrate that the ILP and the LP can have radically different optimal solutions, we consider the problem of finding a largest independent set in an undirected graph $G = (V, E)$. An independent set is a subset I of V with the property that I contains no edge; in other words, no edge of E has *both* endpoints in I.

Introducing 0/1-variables x_v for all $v \in V$, every subset W of vertices can be encoded by a vector $\tilde{x}$, with the interpretation that

$$v \in W \quad \Leftrightarrow \quad \tilde{x}_v = 1.$$

The condition that W is an independent set can be enforced by the constraints $x_v + x_w \leq 1$ for all edges $\{v, w\} \in E$. Thus, the problem of finding a largest independent set can be formulated as an ILP as follows.

$$\begin{array}{lll} (\mathrm{ILP}_{IndepSet}(G)) & \text{maximize} & \sum_{v \in V} x_v \\ & \text{subject to} & x_v + x_w \leq 1, \qquad \forall \{v, w\} \in E, \\ & & x_v \in \{0, 1\}, \qquad \forall v \in V. \end{array}$$

Here is its LP relaxation.

$$\begin{array}{lll} (\mathrm{LP}_{IndepSet}(G)) & \text{maximize} & \sum_{v \in V} x_v \\ & \text{subject to} & x_v + x_w \leq 1, \qquad \forall \{v, w\} \in E, \\ & & 0 \leq x_v \leq 1, \qquad \forall v \in V. \end{array}$$

Let G be the complete graph on n vertices. We have

$$\mathrm{opt}(\mathrm{ILP}_{IndepSet}(G)) = 1,$$

because the complete graph has only independent sets of size at most 1. On the other hand, we get

$$\mathrm{opt}(\mathrm{LP}_{IndepSet}(G)) \geq \tfrac{n}{2},$$

because the vector

$$\tilde{x} = \left(\frac{1}{2}, \frac{1}{2}, \ldots, \frac{1}{2}\right)$$

is obviously a solution of the LP. We see that the optimal values of LP and ILP are far apart; moreover, the solution $\tilde{x}$ is always a solution, no matter how G looks like. This means that the LP captures too little of the structure of G to be of any interest in solving the independent set problem over G.

3.2 The Simplex Method

Linear Programming (LP) in a quite general form is the problem of maximizing a linear function in n variables subject to m linear inequalities. If, in addition, we require all variables to be nonnegative, we have an LP in *standard form.* which can be written as follows.

$$\begin{array}{lll} (\mathrm{LP}) & \text{maximize} & \sum_{j=1}^{n} c_j x_j \\ & \text{subject to} & \sum_{j=1}^{n} a_{ij} x_j \leq b_i \quad (i = 1, \ldots, m), \\ & & x_j \geq 0 \qquad\qquad (j = 1, \ldots, n), \end{array} \tag{3.1}$$

where the c_j, b_i and a_{ij} are real numbers. By defining

$$\begin{aligned} x &:= (x_1, \ldots, x_n)^T, \\ c &:= (c_1, \ldots, c_n)^T, \\ b &:= (b_1, \ldots, b_m)^T, \\ A &:= \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix} \end{aligned}$$

this can be written in more compact form as

$$\begin{array}{lll} \text{(LP)} & \text{maximize} & c^T x \\ & \text{subject to} & Ax \leq b, \\ & & x \geq 0, \end{array} \tag{3.2}$$

where the relations $\leq$ and $\geq$ hold for vectors of the same length if and only if they hold componentwise.

The vector c is called the *cost vector* of the LP, and the linear function $z : x \mapsto c^T x$ is called the *objective function*. The vector b is referred to as the *right-hand side* of the LP. The inequalities $\sum_{j=1}^{n} a_{ij} x_j \leq b_i$, for $i = 1, \ldots, m$ and $x_j \geq 0$, for $j = 1, \ldots, n$ are the *constraints* of the linear program. (Due to their special nature, the constraints $x_j \geq 0$ are sometimes called *nonnegativity constraints* or *restrictions*).

The LP is called *feasible* if there exists a nonnegative vector $\tilde{x}$ satisfying $A\tilde{x} \leq b$ (such an $\tilde{x}$ is called a *feasible solution*); otherwise the program is called *infeasible*. If there are feasible solutions with arbitrarily large objective function value, the LP is called *unbounded*; otherwise it is *bounded*. A linear program which is both feasible and bounded has a unique maximum value $c^T \tilde{x}$ attained at a (not necessarily unique) optimal feasible solution $\tilde{x}$. Solving the LP means finding such an optimal solution $\tilde{x}$ (if it exists).

3.2.1 Tableaus

When confronted with an LP in standard form, the simplex algorithm starts off by introducing *slack variables* $x_{n+1}, \ldots, x_{n+m}$ to transform the inequality system $Ax \leq b$ into an equivalent system of equalities and additional nonnegativity constraints on the slack variables. The slack variable x_{n+i} closes the gap between the left-hand side and right-hand side of the i-th constraint, i.e.

$$x_{n+i} := b_i - \sum_{j=1}^{n} a_{ij} x_j,$$

for all $i = 1, \ldots, m$. The i-th constraint is then equivalent to

$$x_{n+i} \geq 0,$$

and the linear program can be written as

$$\begin{array}{lll} \text{(LP)} \quad \text{maximize} & \sum_{j=1}^{n} c_j x_j & \\ \text{subject to} & x_{n+i} = b_i - \sum_{j=1}^{n} a_{ij} x_j & (i = 1, \ldots, m), \\ & x_j \geq 0 & (j = 1, \ldots, n+m), \end{array} \tag{3.3}$$

or in a more compact form as

$$\begin{array}{lll} \text{(LP)} \quad \text{maximize} & \underline{c}^T x & \\ \text{subject to} & \underline{A} x = b, & \\ & x \geq 0, & \end{array} \tag{3.4}$$

where $\underline{A}$ is the $m \times (n+m)$-matrix

$$\underline{A} := (A|E), \tag{3.5}$$

$\underline{c}$ is the $(n+m)$-vector

$$\underline{c} := \begin{pmatrix} c \\ 0 \\ \vdots \\ 0 \end{pmatrix} \tag{3.6}$$

and x is the $(n+m)$-vector

$$x = \begin{pmatrix} x_O \\ x_S \end{pmatrix},$$

where x_O is the vector of **o**riginal variables, x_S the vector of **s**lack variables.

Together with the objective function, the m equations for the x_{n+i} in (3.3) contain all the information about the LP. Following tradition, we will represent this information in *tableau* form where the objective function – denoted by z – is written last and separated from the other equations by a solid line. (The restrictions $x_j \geq 0$ do not show up in the tableau but represent implicit knowledge.) In this way we obtain the *initial tableau* for the LP.

$$\begin{array}{rcrcccr} x_{n+1} & = & b_1 & -a_{11}x_1 & - & \cdots & -a_{1n}x_n \\ & & & \vdots & & & \\ x_{n+m} & = & b_m & -a_{m1}x_1 & - & \cdots & -a_{mn}x_n \\ \hline z & = & & c_1 x_1 & + & \cdots & +c_n x_n \end{array} \tag{3.7}$$

The compact form here is

$$\begin{array}{rcrcr} x_S & = & b & - & Ax_O \\ \hline z & = & & & c^T x_O \end{array} \tag{3.8}$$

An example illustrates the process of getting the initial tableau from an LP in standard form.

Example 3.2 Consider the problem

$$\begin{array}{lrcrcl} \text{maximize} & x_1 + x_2 & & & & \\ \text{subject to} & -x_1 & + & x_2 & \leq & 1, \\ & x_1 & & & \leq & 3, \\ & & & x_2 & \leq & 2, \\ & x_1, x_2 \geq 0. & & & & \end{array} \tag{3.9}$$

After introducing slack variables x_3, x_4, x_5, the LP in equality form is

$$\begin{array}{lrcrcrcl} \text{maximize} & x_1 + x_2 & & & & & & \\ \text{subject to} & x_3 & = & 1 & + & x_1 & - & x_2, \\ & x_4 & = & 3 & - & x_1, & & \\ & x_5 & = & 2 & & & - & x_2, \\ x_1, \ldots, x_5 \geq 0. & & & & & & & \end{array} \tag{3.10}$$

From this we obtain the initial tableau

$$\begin{array}{rcrcrcr} x_3 & = & 1 & + & x_1 & - & x_2 \\ x_4 & = & 3 & - & x_1 & & \\ x_5 & = & 2 & & & - & x_2 \\ \hline z & = & & & x_1 & + & x_2 \end{array} \tag{3.11}$$

Abstracting from the initial tableau (3.7), a general tableau for the LP is any system $\mathcal{T}$ of $m+1$ linear equations in the variables $x_1, \ldots, x_{n+m}$ and z, with the properties that

(i) $\mathcal{T}$ expresses m left-hand side variables x_B and z in terms of the remaining d right-hand side variables x_N, i.e. there is an m-vector β, a n-vector γ, an $m \times n$-matrix Λ and a real number z_0 such that $\mathcal{T}$ (written in compact form) is the system

$$\begin{array}{rcrcr} x_B & = & \beta & - & \Lambda x_N \\ \hline z & = & z_0 & + & \gamma^T x_N \end{array} \tag{3.12}$$

(ii) Any solution of (3.12) is a solution of (3.8) and vice versa.

By property (ii), any tableau contains the *same* information about the LP but represented in a *different* way. All that the simplex algorithm is about is constructing a sequence of tableaus by gradually rewriting them, finally leading to a tableau in which the information is represented in such a way that the desired optimal solution can be read off directly. We will immediately show how this works in our example.

3.2.2 Pivoting

Here is the initial tableau (3.11) to Example 3.2 again.

$$\begin{array}{rcrcrcr} x_3 & = & 1 & + & x_1 & - & x_2 \\ x_4 & = & 3 & - & x_1 & & \\ x_5 & = & 2 & & & - & x_2 \\ \hline z & = & & & x_1 & + & x_2 \end{array}$$

By setting the right-hand side variables x_1, x_2 to zero, we find that the left-hand side variables x_3, x_4, x_5 assume nonnegative values $x_3 = 1, x_4 = 3, x_5 = 2$. This means, the vector $x = (0, 0, 1, 3, 2)$ is a feasible solution of (3.10) (and the vector $x' = (0, 0)$ is a feasible solution of (3.9)). The objective function value $z = 0$ associated with this feasible solution is computed from the last row of the tableau. In general, any feasible solution that can be obtained by setting the right-hand side variables of a tableau to zero is called a *basic feasible solution* (BFS). In this case we also refer to the tableau as a feasible tableau. The left-hand side variables of a feasible tableau are called *basic* and are said to constitute a *basis*, the right-hand side ones are *nonbasic*. The goal of the simplex algorithm is now either to construct a new feasible tableau with a corresponding BFS of higher z-value, or to prove that there exists no feasible solution at all with higher z-value. In the latter case the BFS obtained from the tableau is reported as an optimal solution to the LP; in the former case, the process is repeated, starting from the new tableau.

In the above tableau we observe that increasing the value of x_1 (i.e. making x_1 positive) will increase the z-value. The same is true for x_2, and this is due to the fact that both variables have positive coefficients in the z-row of the tableau. Let us arbitrarily choose x_2. By how much can we increase x_2? If we want to maintain feasibility, we have to be careful not to let any of the basic variables go below zero. This means, the equations determining the values of the basic variables may limit x_2's increment. Consider the first equation

$$x_3 = 1 + x_1 - x_2. \tag{3.13}$$

Together with the implicit constraint $x_3 \geq 0$, this equation lets us increase x_2 up to the value $x_2 = 1$ (the other nonbasic variable x_1 keeps its zero value). The second equation

$$x_4 = 3 - x_1$$

does not limit the increment of x_2 at all, and the third equation

$$x_5 = 2 - x_2$$

allows for an increase up to the value $x_2 = 2$ before x_5 gets negative. The most stringent restriction therefore is $x_3 \geq 0$, imposed by (3.13), and we will increase x_2 just as much as we can, so we get $x_2 = 1$ and x_3=0. From the remaining tableau equations, the values of the other variables are obtained as

$$\begin{aligned} x_4 &= 3 - x_1 = 3, \\ x_5 &= 2 - x_2 = 1. \end{aligned}$$

To establish this as a BFS, we would like to have a tableau with the new zero variable x_3 replacing x_2 as a nonbasic variable. This is easy – the equation (3.13) which determined the new value of x_2 relates both variables. This equation can be rewritten as

$$x_2 = 1 + x_1 - x_3,$$

and substituting the right-hand side for x_2 into the remaining equations gives the new tableau

$$\begin{array}{rcrcrcr} x_2 & = & 1 & + & x_1 & - & x_3 \\ x_4 & = & 3 & - & x_1 & & \\ x_5 & = & 1 & - & x_1 & + & x_3 \\ \hline z & = & 1 & + & 2x_1 & - & x_3 \end{array}$$

with corresponding BFS $x = (0, 1, 0, 3, 1)$ and objective function value $z = 1$. This process of rewriting a tableau into another one is called a *pivot step*, and it is clear by construction that both systems have the same set of solutions. The effect of a pivot step is that a nonbasic variable (in this case x_2) *enters* the basis, while a basic one (in this case x_3) *leaves* it. Let us call x_2 the *entering variable* and x_3 the *leaving variable*.

In the new tableau, we can still increase x_1 and obtain a larger z-value. x_3 cannot be increased since this would lead to smaller z-value. The first equation puts no restriction on the increment, from the second one we get $x_1 \leq 3$ and from the third one $x_1 \leq 1$. So the third one is the most stringent, will be rewritten and substituted into the remaining equations as above. This means, x_1 enters the basis, x_5 leaves it, and the tableau we obtain is

$$\begin{array}{rcrcrcr} x_2 & = & 2 & & & - & x_5 \\ x_4 & = & 2 & - & x_3 & + & x_5 \\ x_1 & = & 1 & + & x_3 & - & x_5 \\ \hline z & = & 3 & + & x_3 & - & 2x_5 \end{array}$$

with BFS $x = (1, 2, 0, 2, 0)$ and $z = 3$. Performing one more pivot step (this time with x_3 the entering and x_4 the leaving variable), we arrive at the tableau

$$\begin{array}{rcrcrcr} x_2 & = & 2 & & & - & x_5 \\ x_3 & = & 2 & - & x_4 & + & x_5 \\ x_1 & = & 3 & - & x_4 & & \\ \hline z & = & 5 & - & x_4 & - & x_5 \end{array} \tag{3.14}$$

with BFS $x = (3, 2, 2, 0, 0)$ and $z = 5$. In this tableau, no nonbasic variable can increase without making the objective function value smaller, so we are stuck. Luckily, this means that we have already found an optimal solution. Why? Consider any *feasible* solution $\tilde{x} = (\tilde{x}_1, \ldots, \tilde{x}_5)$ for (3.10), with objective function value z_0. This is a solution to (3.11) and therefore a solution to (3.14). Thus,

$$z_0 = 5 - \tilde{x}_4 - \tilde{x}_5$$

must hold, and together with the implicit restrictions $x_4, x_5 \geq 0$ this implies $z_0 \leq 5$. The tableau even delivers a proof that the BFS we have computed is the unique optimal solution to the problem: $z = 5$ implies $x_4 = x_5 = 0$, and this determines the values of the other variables. Ambiguities occur only if some of the nonbasic variables have zero coefficients in the z-row of the final tableau. Unless a specific optimal solution is required, the simplex algorithm in this case just reports the optimal BFS it has at hand.

3.2.3 Geometric Interpretation

Consider the standard form LP (3.1). For each constraint

$$\begin{aligned} \sum_{j=1}^{n} a_{ij}x_j &\leq b_i \quad \text{or} \\ x_j &\geq 0, \end{aligned}$$

the points $\tilde{x} \in \mathbb{R}^n$ satisfying the constraint form a closed *halfspace* in $\mathbb{R}^n$. The points for which equality holds form the boundary of this halfspace, the *constraint hyperplane*.

The set of feasible solutions of the LP is therefore an intersection of halfspaces, which is by definition a (possibly empty) *polyhedron* P. The facets of P are induced by (not necessarily all) constraint hyperplanes. The nonnegativity constraints $x_j \geq 0$ restrict P to lie inside the positive orthant of $\mathbb{R}^n$. The following correspondence between basic feasible solutions of the LP and vertices of P justifies the geometric interpretation of the simplex method as an algorithm that traverses a sequence of vertices of P until an optimal vertex is found.

Fact 3.3 *Consider a standard form LP with feasible polyhedron P. The point $\tilde{x}' = (\tilde{x}_1, \ldots, \tilde{x}_n)$ is a vertex of P if and only if the vector $\tilde{x} = (\tilde{x}_1, \ldots, \tilde{x}_{n+m})$ with*

$$\tilde{x}_{n+i} := b_i - \sum_{j=1}^{n} a_{ij}\tilde{x}_j, \;\; i = 1, \ldots, m$$

is a basic feasible solution of the LP.

Two consecutive tableaus constructed by the simplex method have $n-1$ nonbasic variables in common. Their BFS thus share $n-1$ zero variables. Equivalently, the corresponding vertices lie on $n-1$ common constraint hyperplanes, and this means that they are adjacent in P. The feasible solutions obtained in the process of continuously increasing the value of a nonbasic variable until it becomes basic correspond to the points on the edge of P connecting the two vertices.

Here we are content with checking the correlations in case of Example 3.2. The LP consists of five constraints over two variables, therefore the feasible region is a polygon in $\mathbb{R}^2$. Every constraint hyperplane defines a facet, so we get a polygon with five edges and five vertices. In the previous subsection we were going through a sequence of four tableaus until we discovered an optimal BFS. The picture below shows how this corresponds to a sequence of adjacent vertices. The optimization direction c is drawn as a fat arrow. Since the objective function value gets higher in every iteration, the path of vertices is monotone in direction c (Figure 3.2).

BFS	vertex	$z = x_1 + x_2$
$(0, 0, 1, 3, 4)$	$(0, 0)$	0
$(0, 1, 0, 3, 1)$	$(0, 1)$	1
$(1, 2, 0, 2, 0)$	$(1, 2)$	3
$(3, 2, 2, 0, 0)$	$(3, 2)$	5

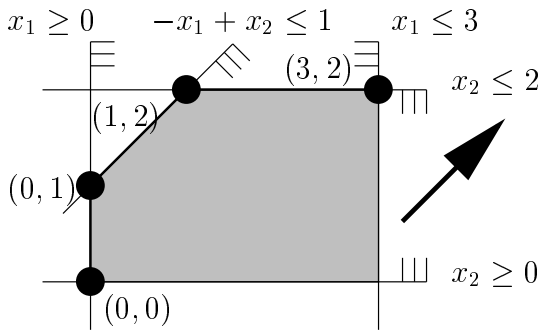


Figure 3.2: Geometric interpretation of the simplex method

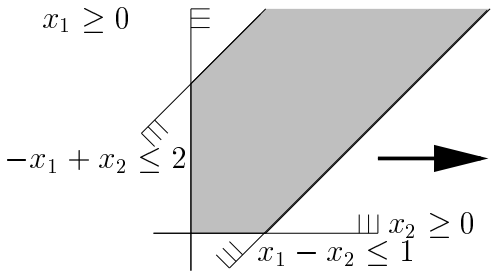
3.2.4 Exception Handling

So far our outline of the simplex method went pretty smooth. This is in part due to the fact that we have only seen one very small and trivial example of the way it works. On the other hand, the method *is* simple, and we will just incorporate some 'exception handling' and do a little fine tuning, again basically by example.

Unboundedness

During a pivot step, we make the value of a nonbasic variable just large enough to get the value of a basic variable down to zero. This, however, might never happen. Consider the example

$$\begin{array}{lrcl} \text{maximize} & x_1 & & \\ \text{subject to} & x_1 - x_2 & \leq & 1, \\ & -x_1 + x_2 & \leq & 2, \\ & x_1, x_2 \geq 0. & & \end{array}$$



with initial tableau

$$\begin{array}{rcrcrcr} x_3 & = & 1 & - & x_1 & + & x_2 \\ x_4 & = & 2 & + & x_1 & - & x_2 \\ \hline z & = & & & x_1 & & \end{array}$$

After one pivot step with x_1 entering the basis we get the tableau

$$\begin{array}{rcrcrcr} x_1 & = & 1 & + & x_2 & - & x_3 \\ x_4 & = & 3 & & & - & x_3 \\ \hline z & = & 1 & + & x_2 & - & x_3 \end{array}$$

If we now try to bring x_2 into the basis by increasing its value, we notice that none of the tableau equations puts a limit on the increment. We can make x_2 and z arbitrarily large – the problem is unbounded. By letting x_2 go to infinity we get a feasible halfline – starting from the current BFS – as a witness for the unboundedness. In our case this is

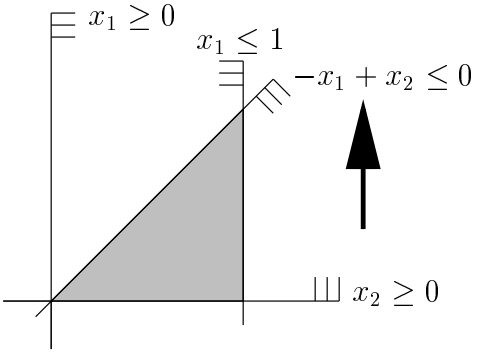
the set of feasible solutions

$$\{(1,0,0,3) + x_2(1,1,0,0) \mid x_2 \geq 0\}.$$

Such a halfline will typically be the output of the algorithm in the unbounded case. Thus, unboundedness can quite naturally be handled with the existing machinery. In the geometric interpretation it just means that the feasible polyhedron P is unbounded in the optimization direction.

Degeneracy

While we can make some nonbasic variable arbitrarily large in the unbounded case, just the other extreme happens in the degenerate case: some tableau equation limits the increment to zero so that no progress in z is possible. Consider the LP



$$\begin{array}{lrcl} \text{maximize} & x_2 & & \\ \text{subject to} & -x_1 + x_2 & \leq & 0, \\ & x_1 & \leq & 2, \\ & x_1, x_2 \geq 0, & & \end{array} \tag{3.15}$$

with initial feasible tableau

$$\begin{array}{rcrcr} x_3 &=& & x_1 - x_2 \\ x_4 &=& 2 - x_1 & & \\ \hline z &=& & & x_2 \end{array}$$

The only candidate for entering the basis is x_2, but the first tableau equation shows that its value cannot be increased without making x_3 negative. This may happen whenever in a BFS some basic variables assume zero value, and such a situation is called *degenerate*. Unfortunately, the impossibility of making progress in this case does not imply optimality, so we have to perform a 'zero progress' pivot step. In our example, bringing x_2 into the basis results in another degenerate tableau with the same BFS.

$$\begin{array}{rcrcr} x_2 &=& & x_1 - x_3 \\ x_4 &=& 2 - x_1 & & \\ \hline z &=& & x_1 - x_3 \end{array}$$

Nevertheless, the situation has improved. The nonbasic variable x_1 can be increased now, and by entering it into the basis (replacing x_4) we already obtain the final tableau

$$\begin{array}{rcrcr} x_1 &=& 2 & & - x_4 \\ x_2 &=& 2 - x_3 & - & x_4 \\ \hline z &=& 2 - x_3 & - & x_4 \end{array}$$

with optimal BFS $x = (x_1, \ldots, x_4) = (2, 2, 0, 0)$.

In this example, after one degenerate pivot we were able to make progress again. In general, there might be longer runs of degenerate pivots. Even worse, it may happen that

a tableau repeats itself during a sequence of degenerate pivots, so the algorithm can go through an infinite sequence of tableaus without ever making progress. This phenomenon is known as *cycling*. If the algorithm does not terminate, it must cycle. This follows from the fact that there are only finitely many different tableaus.

Fact 3.4 *The LP (3.1) has at most* $\binom{m+n}{m}$ *tableaus.*

To prove this, we show that any tableau $\mathcal{T}$ is already determined by its basis variables. Write $\mathcal{T}$ as

$$\begin{array}{rcrcr} x_B & = & \beta & - & \Lambda x_N \\ \hline z & = & z_0 & + & \gamma^T x_N, \end{array}$$

and assume there is another tableau $\mathcal{T}'$ with the same basic and nonbasic variables, i.e. $\mathcal{T}'$ is the system

$$\begin{array}{rcrcr} x_B & = & \beta' & - & \Lambda' x_N \\ \hline z & = & z'_0 & + & \gamma'^T x_N, \end{array}$$

By the tableau properties, both systems have the same set of solutions. Therefore

$$\begin{aligned} (\beta - \beta') - (\Lambda - \Lambda')x_N &= 0 \text{ and} \\ (z_0 - z'_0) + (\gamma^T - \gamma'^T)x_N &= 0 \end{aligned}$$

must hold for all n-vectors x_N, and this implies $\beta = \beta', \Lambda = \Lambda', \gamma = \gamma'$ and $z_0 = z'_0$. Hence $\mathcal{T} = \mathcal{T}'$.

There is a standard way to avoid cycling, by using symbolic perturbation. One perturbs the right-hand side vector b of the LP by adding powers of a symbolic constant ε (assumed to be infinitesimally small). The LP then becomes

$$\begin{array}{lll} \text{maximize} & \sum_{j=1}^n c_j x_j & \\ \text{subject to} & \sum_{j=1}^n a_{ij} x_j \le b_i + \varepsilon^i & (i = 1, \ldots, m), \\ & x_j \ge 0 & (j = 1, \ldots, n), \end{array} \tag{3.16}$$

and if the original LP (3.1) is feasible, so is (3.16). A solution to (3.1) can be obtained from a solution to (3.16) by ignoring the contribution of ε, i.e. by setting ε to zero. Moreover, any valid tableau for (3.16) reduces to a valid tableau for (3.1) when the terms involving powers of ε are disregarded.

In case of (3.15), the initial tableau of the perturbed problem is

$$\begin{array}{rcrcrcr} x_3 & = & \varepsilon & + & x_1 & - & x_2 \\ x_4 & = & 2 + \varepsilon^2 & - & x_1 & & \\ \hline z & = & & & & & x_2 \end{array}$$

Pivoting with x_2 entering the basis gives the tableau

$$\begin{array}{rcrcrcr} x_2 & = & \varepsilon & + & x_1 & - & x_3 \\ x_4 & = & 2 + \varepsilon^2 & - & x_1 & & \\ \hline z & = & \varepsilon & + & x_1 & - & x_3 \end{array} \tag{3.17}$$

This is no longer a degenerate pivot, since x_2 (and z) increased by ε. Finally, bringing x_1 into the basis gives the tableau

$$\begin{array}{lcrcrcl} x_1 & = & 2+\varepsilon^2 & & & - & x_4 \\ x_2 & = & 2+\varepsilon+\varepsilon^2 & - & x_3 & - & x_4 \\ \hline z & = & 2+\varepsilon+\varepsilon^2 & - & x_3 & - & x_4 \end{array} \tag{3.18}$$

with optimal BFS $x = (2+\varepsilon^2, 2+\varepsilon+\varepsilon^2, 0, 0)$. The optimal BFS for (3.15) is recovered from this by ignoring the additive terms in ε. In general, the following holds, which proves nondegeneracy of the perturbed problem.

Fact 3.5 *In any BFS of (3.16), the values of the basic variables are nonzero polynomials in ε, of degree at most m. The tableau coefficients at the nonbasic variables are unaffected by the perturbation.*

To find the leaving variable, polynomials in ε have to be compared. This is done lexicographically, i.e.

$$\sum_{k=1}^{m} \lambda_k \varepsilon^k < \sum_{k=1}^{m} \lambda'_k \varepsilon^k$$

if and only if $(\lambda_1, \ldots, \lambda_m)$ is lexicographically smaller than $(\lambda'_1, \ldots, \lambda'_m)$. The justification for this is that one could actually assign a very small numerical value to ε (depending on the input numbers of the LP), such that comparing lexicographically is equivalent to comparing numerically, for all polynomials that turn up in the algorithm.

In the perturbed problem, progress is made in every pivot step. Cycling cannot occur and the algorithm terminates after at most $\binom{m+n}{m}$ pivots.In the associated feasible polyhedron, degeneracies correspond to 'overcrowded vertices', which are vertices where more than n of the constraint hyperplanes meet. There are several ways to represent the same vertex as an intersection of exactly n hyperplanes, and a degenerate pivot switches between two such representations. The perturbation slightly moves the hyperplanes relative to each other in such a way that any degenerate vertex is split into a collection of nondegenerate ones very close together.

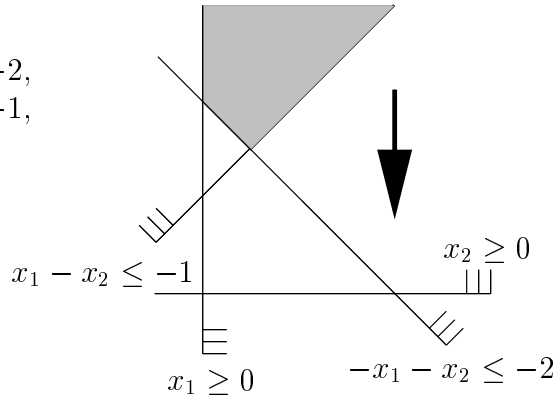
Infeasibility

To start off, the simplex method needs some feasible tableau. In all examples considered so far such a tableau was readily available since the initial tableau was feasible. We say that the problem has a *feasible origin*. This is equivalently expressed by the fact that the right-hand side vector b of the LP is nonnegative. If this is not the case, we first solve an auxiliary problem that either constructs a BFS to the original problem or proves that the original problem is infeasible. The auxiliary problem has an additional variable x_0 and is defined as

$$\begin{array}{ll} \text{minimize} & x_0 \\ \text{subject to} & \sum_{j=1}^{n} a_{ij} x_j - x_0 \le b_i \quad (i = 1, \ldots, m), \\ & x_j \ge 0 \qquad\qquad\qquad\quad (j = 0, \ldots, n). \end{array}$$

This problem is feasible (choose x_0 big enough), and it is clear that the original problem is feasible if and only if the optimum value of the auxiliary LP is zero. Let us do an example and consider the problem

$$\begin{array}{lrcl} \text{maximize} & -x_2 & & \\ \text{subject to} & -x_1 - x_2 & \leq & -2, \\ & x_1 - x_2 & \leq & -1, \\ & x_1, x_2 \geq 0. & & \end{array}$$



with initial tableau

$$\begin{array}{rcrcrcr} x_3 & = & -2 & + & x_1 & + & x_2 \\ x_4 & = & -1 & - & x_1 & + & x_2 \\ \hline z & = & & & & - & x_2 \end{array}$$

This problem has an infeasible origin, because setting the right-hand side variables to zero gives $x_3 = -2, x_4 = -1$. The auxiliary problem (written in maximization form to avoid confusion and with the objective function called w in the tableau) is

$$\begin{array}{lrcl} \text{maximize} & -x_0 & & \\ \text{subject to} & -x_1 - x_2 - x_0 & \leq & -2, \\ & x_1 - x_2 - x_0 & \leq & -1, \\ & x_0, x_1, x_2 \geq 0. & & \end{array}$$

with initial tableau

$$\begin{array}{rcrcrcrcr} x_3 & = & -2 & + & x_1 & + & x_2 & + & x_0 \\ x_4 & = & -1 & - & x_1 & + & x_2 & + & x_0 \\ \hline w & = & & & & & & - & x_0 \end{array}$$

The auxiliary problem has an infeasible initial tableau, too, but we can easily construct a feasible tableau by performing one pivot step. We start increasing the value of x_0, this time not with the goal of *maintaining* feasibility but with the goal of *reaching* feasibility. To get $x_3 \geq 0$, x_0 has to increase by at least 2, and this also makes x_4 positive. By setting $x_0 := 2$ we get $x_3 = 0$ and $x_4 = 1$. Solving the first tableau equation for x_0 and substituting from this into the remaining equations as usual gives a new *feasible* tableau with x_0 basic and x_3 nonbasic.

$$\begin{array}{rcrcrcrcr} x_0 & = & 2 & - & x_1 & - & x_2 & + & x_3 \\ x_4 & = & 1 & - & 2x_1 & & & + & x_3 \\ \hline w & = & -2 & + & x_1 & + & x_2 & - & x_3 \end{array}$$

The simplex method can now be used to solve the auxiliary problem. In our case, by choosing x_2 as the entering variable, we accomplish this in one step. The resulting tableau is

$$\begin{array}{rcrcrcrcr} x_2 & = & 2 & - & x_1 & + & x_3 & - & x_0 \\ x_4 & = & 1 & - & 2x_1 & + & x_3 & & \\ \hline w & = & & & & & & - & x_0 \end{array}$$

Since all coefficients of nonbasic variables in the w-row are nonpositive, this is an optimal tableau with BFS $x = (x_0, \ldots, x_4) = (0, 0, 2, 0, 1)$. The associated zero w-value asserts that the LP we originally wanted to solve is actually feasible, and we can even construct a feasible tableau for it from the final tableau of the auxiliary problem by ignoring x_0 and expressing the original objective function z in terms of the nonbasic variables; from the first tableau equation we get in our case $z = -x_2 = -2 + x_1 - x_3$, and this gives a valid feasible tableau

$$\begin{array}{rcrcrcr} x_2 & = & 2 & - & x_1 & + & x_3 \\ x_4 & = & 1 & - & 2x_1 & + & x_3 \\ \hline z & = & -2 & + & x_1 & - & x_3 \end{array}$$

with corresponding BFS $x = (x_1, \ldots, x_4) = (0, 2, 0, 1)$ for the original LP. For this to work, x_0 should be nonbasic in the final tableau of the auxiliary problem which is automatically the case if the problem is nondegenerate. (To guarantee this in the general situation, choose x_0 as the leaving variable whenever this is a possible choice.)

If the optimum value of the auxiliary problem is nonzero, we can conclude that the original LP is infeasible and simply report this fact.

3.2.5 Tableaus from bases

We have argued earlier that a tableau is uniquely determined by the set of basic variables. But how do we compute the corresponding tableau? It will have the form

$$\begin{array}{rcrcr} x_B & = & \beta & - & \Lambda x_N \\ \hline z & = & z_0 & + & \gamma^T x_N . \end{array} \tag{3.19}$$

Formally, if G is some subset

$$G = \{j_1, \ldots, j_k\} \subseteq [n + m]$$

of variable subscripts, then

$$x_G := (x_{j_1}, \ldots x_{j_k})^T \tag{3.20}$$

is the vector of all the variables with subscripts in G. Any tableau is therefore uniquely specified by its basic subscript set B. We will make this specification explicit, i.e. we show how the entries β, γ, Λ and z_0 relate to B.

Consider the LP in compact equality form (3.4), which is

$$\begin{array}{lll} \text{(LP)} & \text{maximize} & \underline{c}^T x \\ & \text{subject to} & \underline{A} x = b, \\ & & x \geq 0, \end{array} \tag{3.21}$$

with $\underline{A}$ and $\underline{c}$ defined according to (3.5) resp. (3.6).

Let $\underline{A}_j$ denote the j-th column of $\underline{A}$. For subscript set $G = \{j_1, \ldots, j_k\}$ let

$$\underline{A}_G := (\underline{A}_{j_1}, \ldots, \underline{A}_{j_k}) \tag{3.22}$$

collect the k columns corresponding to the variables with subscripts in G. Then the equations of (3.21) read as

$$\underline{A}_B x_B + \underline{A}_N x_N = b. \tag{3.23}$$

Since (3.19) has by definition of a tableau the same set of solutions as (3.23), the former is obtained by simply solving (3.23) for x_B, which gives

$$x_B = \underline{A}_B^{-1} b - \underline{A}_B^{-1} \underline{A}_N x_N, \tag{3.24}$$

and therefore

$$\begin{array}{rcl} \beta & = & \underline{A}_B^{-1} b, \\ \Lambda & = & \underline{A}_B^{-1} \underline{A}_N. \end{array} \tag{3.25}$$

By similar reasoning we compute γ and z_0. For $G = \{j_1, \ldots, j_k\}$ let

$$\underline{c}_G := (\underline{c}_{j_1}, \ldots, \underline{c}_{j_k})^T \tag{3.26}$$

collect the entries corresponding to variables with subscripts in G. Then the equation for z in (3.4) reads as

$$z = \underline{c}_B^T x_B + \underline{c}_N^T x_N. \tag{3.27}$$

Again by the tableau property, the last row of (3.19) is equivalent to (3.27), and the former is obtained by simply substituting from (3.24) into (3.27), which gives

$$z = \underline{c}_B^T \underline{A}_B^{-1} b + (\underline{c}_N^T - \underline{c}_B^T \underline{A}_B^{-1} \underline{A}_N) x_N, \tag{3.28}$$

and therefore

$$\begin{array}{rcl} z_0 & = & \underline{c}_B^T \underline{A}_B^{-1} b, \\ \gamma^T & = & \underline{c}_N^T - \underline{c}_B^T \underline{A}_B^{-1} \underline{A}_N. \end{array} \tag{3.29}$$

These formulas show that it is not necessary to rewrite the whole tableau in every pivot step (this affects $\Theta(nm)$ entries). Rather, it is sufficient to maintain and update the current BFS, the basic subscript set B as well as the *inverse* of the matrix $\underline{A}_B$. All necessary information can be retrived directly from this. Note that before the first iteration, $\underline{A}_B = \underline{A}_B^{-1} = E$. Because this matrix has only m^2 entries, this is a saving for large values of n.

3.2.6 Pivot Rules

A *pivot rule* is any scheme for choosing the entering variable in case there is more than one candidate for it (which is the typical situation). The number of pivot steps that are necessary to solve the LP crucially depend on the pivot rule (this can already be verified

in our running example; depending on whether we choose x_1 or x_2 in the first step, we need 2 or 3 steps). The problem is of course that we don't know beforehand which choice will turn out most profitable in the end. Therefore, a pivot rule is always a *heuristic*.

Here are some classical rules. The term 'improving variable' refers to any nonbasic variable with positive coefficient in the z-row.

LARGEST COEFFICIENT. Enter the improving variable with largest coefficient in the z-row. This is the rule originally proposed by Dantzig, the inventor of the simplex method.

LARGEST INCREASE. Enter the improving variable which leads to the largest *absolute* improvement in z. This rule is computationally more expensive than the LARGEST COEFFICIENT rule but locally maximizes the progress.

STEEPEST EDGE. Enter the improving variable which maximizes the z-improvement per *normalized* unit increase of the entering variable (this is the increase necessary to translate the current feasible solution by a vector of unit length). Geometrically, this corresponds to choosing the steepest upward edge starting at the current vertex of the feasible polyhedron.

SMALLEST SUBSCRIPT. Enter the improving variable with smallest subscript. This is Bland's rule, and it is of theoretical interest because it avoids cycling, as mentioned in Subsection 3.2.4.

RANDOM EDGE. Enter an improving variable, chosen uniformly at random from all candidates. This rule (and randomized rules in general) are of quite some relevance, when it comes to the worst-case complexity of the simplex method. We will come back to this in a later section.

3.3 Duality

Assume the discotheque *Kaufleuten* in Zürich wants to organize a party, for which it has 2000 units of champagne, 5000 units of beer, and 6000 units of mineral water in stock.

There are two categories of guests, VIPs and fillers, with different drinking demands. Each VIP drinks 3 units of champagne, 3 units of beer, and 2 unit of mineral water; each filler consumes one unit of champagne, 4 units of beer, and 5 units of mineral water.

Moreover, the *Kaufleuten* charges entrance fees of CHF 20 per VIP and CHF 10 per filler, but the drinks are free.[1]

Of course, the *Kaufleuten* wants to maximize its profit; so it asks for the number v of VIPs and the number f of fillers that must be admitted such that the entrance fees are maximized under the conditions that the drinks are sufficient.

[1]Knowing Zürich prices, these entrance fees are slightly unrealistic, but this is a theory lecture anyway.

This can be formulated as a linear program in 2 nonnegative variables v and f, and three inequality constraints, see Figure 3.3.

$$
\begin{array}{llrcrcrl}
(LP) & \text{maximize} & 20v & + & 10f & & & \text{(entrance fee)} \\
 & \text{subject to} & 3v & + & f & \leq & 2000 & \text{(enough champagne)} \\
 & & 3v & + & 4f & \leq & 5000 & \text{(enough beer)} \\
 & & 2v & + & 5f & \leq & 6000 & \text{(enough mineral water)} \\
 & & v & & & \geq & 0 & \text{(nonnegative VIP number)} \\
 & & & & f & \geq & 0 & \text{(nonnegative filler number)}
\end{array}
$$

Figure 3.3: The *Kaufleuten* LP

Strictly speaking, we have an ILP here, because both v and f must be integer, but let's ignore this issue for now.

There is another way, the *Kaufleuten* could make its money, namely if it does not charge entrance fees but sells the drinks instead. Let C, B and M be the prices the *Kaufleuten* charges per unit of champagne, beer and mineral water. As long as

$$3C + 3B + 2M \geq 20 \tag{3.30}$$

and

$$C + 4B + 5C \geq 10, \tag{3.31}$$

the profit made from selling drinks to VIPs and fillers compensates the loss in entrance fees. It follows that under these constraints, the *Kaufleuten* will not make less money by selling drinks than it will make by charging entrance fees. In particular, as it cannot sell more drinks than it has, the value

$$2000C + 5000B + 6000M$$

is an upper bound for the profit that is made by selling drinks at prices C, B, M; this again is an upper bound for the profit that is made by charging entrance fees, provided the constraints (3.30) and (3.31) hold. If we want to have a good upper bound for the profit we can make from charging entrance fees, we should try to make the value $2000C + 5000B + 6000M$ as small as the constraints (3.30) and (3.31) allow it.

The best upper bound we can obtain with these considerations is therefore given by the optimal value of the LP in Figure (3.4).

What we have just derived is the statement

$$\text{opt(LP)} \leq \text{opt(LP}^{\Delta}\text{)}. \tag{3.32}$$

(LP$^{\Delta}$) is called the *dual* of (LP). In general, if we have a problem

$$
\begin{array}{lll}
\text{(LP)} & \text{maximize} & c^T x \\
 & \text{subject to} & Ax \leq b, \\
 & & x \geq 0
\end{array}
$$

$$
\begin{array}{lllrcrcrcr}
(LP^{\Delta}) & \text{minimize} & 2000C + 5000B + 6000M & & & & & & & \\
 & \text{subject to} & & 3C & + & 3B & + & 2M & \geq & 20 \\
 & & & C & + & 4B & + & 5M & \geq & 10 \\
 & & & C & & & & & \geq & 0 \\
 & & & & & B & & & \geq & 0 \\
 & & & & & & & M & \geq & 0
\end{array}
$$

Figure 3.4: The dual of the *Kaufleuten* LP

in n nonnegative variables and m additional inequality constraints, its dual is the problem

$$
\begin{array}{lll}
(\mathrm{LP}^{\Delta}) & \text{minimize} & b^T y \\
 & \text{subject to} & A^T y \geq c, \\
 & & y \geq 0
\end{array}
$$

in m nonnegative variables and n additional inequality constraints. In the *Kaufleuten* case, we have

$$
A = \begin{pmatrix} 3 & 1 \\ 3 & 4 \\ 2 & 5 \end{pmatrix}, \quad b = \begin{pmatrix} 2000 \\ 5000 \\ 6000 \end{pmatrix}, \quad c = \begin{pmatrix} 20 \\ 10 \end{pmatrix}, \quad x = \begin{pmatrix} v \\ f \end{pmatrix}, \quad y = \begin{pmatrix} C \\ B \\ M \end{pmatrix}.
$$

Let's prove the statement (3.32) in the general case. The proof also shows that nothing really magic happens here.

Theorem 3.6 *Consider a linear program*

$$
\begin{array}{lll}
(LP) & \textit{maximize} & c^T x \\
 & \textit{subject to} & Ax \leq b, \\
 & & x \geq 0
\end{array}
$$

and its dual

$$
\begin{array}{lll}
(LP^{\Delta}) & \textit{minimize} & b^T y \\
 & \textit{subject to} & A^T y \geq c, \\
 & & y \geq 0.
\end{array}
$$

For all feasible solutions $\tilde{x}$ of (LP) and all feasible solutions $\tilde{y}$ of (LP^{Δ}),

$$
c^T \tilde{x} \leq b^T \tilde{y}
$$

holds.

Proof. Let $\tilde{y} \geq 0$ and assume $\tilde{x}$ feasible for (LP). Then we have

$$
\tilde{y}^T A \tilde{x} \leq \tilde{y}^T b,
$$

because inequalities are preserved under multiplication with nonnegative numbers. If in addition $\tilde{y}^T A \geq c^T$ holds (i.e. $\tilde{y}$ is feasible for (LP^Δ)), then we get

$$\tilde{y}^T b \geq \tilde{y}^T A \tilde{x} \geq c^T \tilde{x},$$

as claimed. □

The amazing fact is that a statement stronger than (3.32) holds, namely that (under suitable conditions)

$$\mathrm{opt}(\mathrm{LP}) = \mathrm{opt}(\mathrm{LP}^\Delta).$$

This means, the original (primal) LP has the same optimal value as the dual LP. You might want to check this in case of the *Kaufleuten* LP.

Theorem 3.7 *Consider a* feasible *and* bounded *linear program*

$$\begin{array}{lll} \textit{(LP)} & \textit{maximize} & c^T x \\ & \textit{subject to} & Ax \leq b, \\ & & x \geq 0. \end{array}$$

Then its dual

$$\begin{array}{lll} \textit{(LP}^\Delta\textit{)} & \textit{minimize} & b^T y \\ & \textit{subject to} & A^T y \geq c, \\ & & y \geq 0 \end{array}$$

is feasible and bounded, and there are feasible solutions $\tilde{x}$ of (LP) and $\tilde{y}$ of (LP$^\Delta$) such that

$$c^T \tilde{x} = b^T \tilde{y}$$

holds. In particular, $\tilde{x}$ is optimal for (LP) and $\tilde{y}$ is optimal for (LP$^\Delta$).

Proof. If $c^T \tilde{x} = b^T \tilde{y}$ holds, then $\tilde{x}$ and $\tilde{y}$ must be optimal, because $c^T \tilde{x} \leq b^T y$ holds for all feasible solutions y of (LP^Δ) and $c^T x \leq b^T \tilde{y}$ holds for all feasible solutions x of (LP) by Theorem 3.6. It remains to prove the existence of suitable $\tilde{x}$ and $\tilde{y}$.

For this, consider (LP) after introducing slack variables, where it appears in the form

$$\begin{array}{lll} \mathrm{(LP)} & \text{maximize} & \underline{c}^T x \\ & \text{subject to} & \underline{A} x = b, \\ & & x \geq 0, \end{array}$$

and let $\tilde{x}$ be an optimal BFS computed by the simplex method, B the corresponding set of basic indices, $z_0 = \underline{c}^T \tilde{x}$ the optimal objective function value of (LP). We claim that the vector $\tilde{y}$ defined by

$$\tilde{y}^T := \underline{c}_B^T \underline{A}_B^{-1}$$

is feasible for (LP^Δ) and satisfies $b^T \tilde{y} = z_0$ – this proves the Theorem. As we know by (3.29) that

$$z_0 = \underline{c}_B^T \underline{A}_B^{-1} b = \tilde{y}^T b,$$

the last part is easy. For feasibility, we must show that

$$\tilde{y}^T A \geq c^T \quad (\Leftrightarrow \tilde{y}^T A_j \geq c_j, \quad j \in \{1, \ldots, n\}), \tag{3.33}$$

and that

$$\tilde{y} \geq 0 \quad (\Leftrightarrow \tilde{y}^T \mathbf{e}_j \geq 0, \quad j \in \{1, \ldots, m\}). \tag{3.34}$$

Here, $\mathbf{e}_j$ is the j-th unit vector. After writing (3.34) in this strange way, it becomes clear that (3.33) and (3.34) together are equivalent to

$$\tilde{y}^T \underline{A} \geq \underline{c}^T \quad (\Leftrightarrow \tilde{y}^T \underline{A}_j \geq \underline{c}_j, \quad j \in \{1, \ldots, n+m\}).$$

Assume $B = \{j_1, \ldots, j_m\}$. If $j = j_\ell \in B$, then

$$\tilde{y}^T \underline{A}_j = \underline{c}_B^T \underline{A}_B^{-1} \underline{A}_j = \underline{c}_B^T \mathbf{e}_\ell = \underline{c}_{j_\ell} = \underline{c}_j.$$

Here we have used $\underline{A}_B^{-1} \underline{A}_B = E_m$, the unit matrix of dimensions $(m \times m)$. If $j \in N$, we get

$$\tilde{y}^T \underline{A}_j = \underline{c}_B^T \underline{A}_B^{-1} \underline{A}_j = \underline{c}_j - \gamma_j \geq \underline{c}_j$$

by (3.29) and because $\gamma_j \leq 0$ (recall that γ_j is the coefficient of the variable x_j in the z-row of the final, optimal tableau). This completes the proof of the duality theorem.

□

Chapter 4

Complexity of the Simplex Method

In the exercise we have seen that an LP with n variables and m inequality constraints might have up to

$$\binom{n+m}{m}$$

basic feasible solutions (BFS). Namely, each choice of m variables out of the $n+m$ (original plus slack) variables might give us a basis. This can indeed happen in the degenerate case, otherwise this bound is not attainable.

In this chapter we will be concerned with the case $m \ll n$; you can actually imagine m to be a constant, say 5. In this case,

$$\binom{n+m}{m} = \Theta(n^m),$$

and there are indeed LPs with this many BFS. (As an exercise, you might try to find an LP with 2 inequality constraints which has $\Theta(n^2)$ BFS; if you fail, try to find an LP with 1 inequality constraint that has $\Theta(n)$ BFS.)

For $m = 5$, for example, this means that the simplex method cannot take more than $O(n^5)$ pivot steps to solve the LP. Although this is a polynomial bound, it is far from satisfactory. Unfortunately, for most pivot rules (schemes to choose the entering variable in case there are several choices, cf. previous chapter), nothing better can be proved; the situation is even worse: for some pivot rules, there are linear programs for which this number of steps is actually needed.

Luckily, there is *randomization*. In this chapter we will discuss a randomized pivot rule which solves an LP with constantly many constraints in expected time $O(n)$, which is best possible. Actually, the runtime is of the form $O(f(m) \cdot n)$, where f is a function which grows exponentially with m. For any fixed m, however, $f(m)$ is fixed as well, and we get a linear-time solution.

4.1 An expected linear-time algorithm

Consider an LP in standard form; after introducing slack variables, it assumes the form

$$\begin{array}{lll} \text{(LP)} & \text{maximize} & \underline{c}^T x \\ & \text{subject to} & \underline{A}x = b, \\ & & x \geq 0, \end{array} \tag{4.1}$$

where we have $n+m$ variables indexed with $Q = \{1, \ldots, n+m\}$ and m equality constraints. To solve the problem means to find an *optimal* basis $B(Q) \subseteq Q$, given *some* basis $B \subseteq Q$. The simplex method does this by going through a sequence of bases, starting with B, until some basis is found which cannot be improved anymore – this basis will be output as $B(Q)$. For the complete chpater, we assume that LP is *nondegenerate*, cf. previous chapter. Below we will see why this assumption is necessary.

We will do this in a randomized fashion now, using a recursive algorithm. This algorithm will solve *subproblems* of (4.1). Namely, for $R \subseteq Q$, consider the restricted problem

$$\begin{array}{lll} \text{(LP}(R)\text{)} & \text{maximize} & \underline{c}_R^T x_R \\ & \text{subject to} & \underline{A}_R x_R = b, \\ & & x_R \geq 0, \end{array} \tag{4.2}$$

which arises from (4.1) by removing all variables with indices not in R. We get that LP=LP(Q); moreover, (4.2) can equivalently be written as

$$\begin{array}{lll} \text{(LP}(R)\text{)} & \text{maximize} & \underline{c}^T x \\ & \text{subject to} & \underline{A}x = b, \\ & & x_{Q \setminus R} = 0, \\ & & x \geq 0. \end{array} \tag{4.3}$$

How can we solve the restricted problem? In the formulation (4.3) it becomes clear that we can do this with the simplex method again, provided we have a basis $B \subseteq R$. The corresponding BFS is feasible for LP(R), because all variables with subscripts not in B have value zero – in particular the ones that are not in R. Moreover, we can do pivot steps as we used to do them, as long as we never let any variable with index not in R become nonzero. But this is easy: we never admit such variables as candidates for the entering variable. Then, all variables in $Q \setminus R$ will stay nonbasic throughout and thus will have value zero. Therefore, any BFS we go through is feasible for LP(R).

Doing this, we will reach at some point a basis B' with tableau

$$\begin{array}{rcccc} x_{B'} & = & \beta' & - & \Lambda' x'_N \\ \hline z & = & z'_0 & + & \gamma'^T x'_N, \end{array} \tag{4.4}$$

where we get stuck, because we have

$$\gamma'_j \leq 0 \quad \forall j \in R \setminus B'.$$

This means, there is no candidate for an entering variable, because we excluded $Q \setminus R$ from consideration. Of course, we might have $\gamma'_j > 0$ for some $j \in Q \setminus R$, but as we are not allowed to make that variable basic, we cannot proceed.

Luckily, this means that we have solved $\mathrm{LP}(R)$ to optimality:

Claim 4.1 *$B' = B(R)$, an optimal basis for LP(R). In other words, z'_0 is the optimal objective function value for LP(R).*

Proof. Let $\tilde{x} = (\tilde{x}_{B'}, \tilde{x}_{N'})$ be any feasible solution to $\mathrm{LP}(R)$. The associated objective function value $z = \underline{c}^T \tilde{x}$ satisfies

$$z = z'_0 + \gamma'^T \tilde{x}_{N'} \tag{4.5}$$

by definition of a tableau. Moreover,

$$\gamma'_j \tilde{x}_j = \begin{cases} 0, & \text{if } j \in Q \setminus R, \\ \leq 0, & \text{if } j \in R \setminus B \text{ (because } \gamma'_j \leq 0, \tilde{x}_j \geq 0). \end{cases} \tag{4.6}$$

(4.5) and (4.6) together imply

$$z \leq z'_0,$$

and it follows that z'_0 is the optimal objective function value. As it is attained by the BFS corresponding to B', the claim follows. □

The implication of this is that if we have a basis $B \subseteq R$, we can solve the restricted $\mathrm{LP}(R)$ just as easily as we would have solved $\mathrm{LP} = \mathrm{LP}(Q)$. The following algorithm solves $\mathrm{LP}(R)$, by employing a special pivot rule which restricts the problem to a still more restricted, randomly chosen, subproblem in a first step. `solve`(R, B) outputs an optimal basis $B(R)$ of R, given some basis $B \subseteq R$.

Algorithm 4.2

```
solve(R, B): (* R ⊆ Q, B ⊆ R some basis *)
  IF R = B THEN
     RETURN B
  ELSE
     choose j ∈ R \ B at random
     B' := solve(R \ {j}, B)
     IF B' = B(R) THEN        ← γ'_j ≤ 0
        RETURN B'
     ELSE
        B'' :=pivot(B', j)
        RETURN solve(R, B'')
     END
  END
```

Some comments are in order: if $R = B$, we need to solve the restricted problem $\mathrm{LP}(B)$, where B is the current basis. But then, *no* variable is allowed to enter the basis, and we are done – B is the optimal basis of this subproblem. Otherwise, we mark another variable (index) j as "forbidden" and recursively solve the subproblem $\mathrm{LP}(R \setminus \{j\})$. This gives us $B' = B(R \setminus \{j\})$, an optimal basis for $\mathrm{LP}(R \setminus \{j\})$. If B' turns out to be optimal for $\mathrm{LP}(R)$ as well, we are done and can return B'. How can we test this? We need to check whether the previously forbidden variable j can still lead to an improvement, i.e. whether it is a candidate for the entering variable now. In any case, it is the *only* candidate, because $\gamma'_k \leq 0$ for all $k \in R \setminus (B \cup \{j\})$ with the notation of (4.4). So, the test '$B' = B(R)$?' amounts to the test '$\gamma'_j \leq 0$?'

In case of $\gamma'_j > 0$, we can still improve our solution, and we do so by entering j into the basis. This is indicated by the statement '$B'' :=\mathrm{pivot}(B', j)$', which returns the new and improved basis. With this basis, we simply continue.

From this description, it should be clear, that `solve` is a plain simplex algorithm, with a special rule to choose the candidate for the entering variable that is considered next in any given situation.

It is also clear that `solve` terminates, because in the first recursive call, the problem gets smaller, and in the second one, we have an improved basis. Note, however, that this requires nondegeneracy, otherwise we cannot argue that we make progress in the objective function during the pivot step.

For the analysis of the algorithm `solve` we need some definitions.

Definition 4.3

(i) $z(B)$ *is the objective function value associated with the basis* B. $opt(R) := z(B(R))$ *is the optimal objective function value of* $LP(R)$.

(ii) $j \in R$ *is* enforced *in* (R, B) *if* $z(B) > opt(R \setminus \{j\})$.

(iii) $\dim(R, B) := m - |\{j \in R \mid j \text{ is enforced in } (R, B)\}|$ *is called the* hidden dimension *of* (R, B).

(iv) $t_k(r)$ *is the maximal expected number of* optimality tests '$B' = B(*)$' *that occur in* `solve`(R, B) *(and all its recursive calls), over all sets* R *and bases* $B \subseteq R$ *such that* $|R| = r$ *and* $\dim(R, B) \leq k$.

What does it mean that an element is enforced? Assume R is a set of people, and the B's are possible parties of m people. As you want to find the party which provides the most fun, you rank all B's by some fun value $z(B)$. Assume $j =$ Franz. Then the condition

$$z(B) > \mathrm{opt}(R \setminus \{j\})$$

means that B is a party which guarantees more fun than the best party to which Franz is not invited. This can only mean that Franz $\in B$. But more is true: as you find still better and better parties, Franz must always be on the guest list; in particular, Franz must be

invited to the optimal party. Franz is *enforced* in (R, B), because you must invite Franz to every party which is at least as good as B.

Then, $\dim(R, B)$ is simply the number of free slots you still have on your guest list: $m - \dim(R, B)$ guests are already enforced, i.e. they must be invited. From this it is intuitive that the problem is easier to solve if $\dim(R, B)$ is small, and that's the reason why we want to analyze a call to `solve`(R, B) in terms of its *size* r and its hidden dimension k, by finding good bounds for $t_k(r)$.

Lemma 4.4

$$\begin{aligned} t_k(m) &= 0, \\ t_0(r) &\leq t_0(r-1) + 1, \\ t_k(r) &\leq t_k(r-1) + 1 + \frac{k}{r-m} t_{k-1}(r), \quad r > m, k > 0. \end{aligned}$$

Proof. If $|R| = m$, we have $R = B$, so no optimality tests occur at all. If $\dim(R, B) = 0$, you already have found the optimal basis, because all elements in it are enforced. This means, there can never be another pivot step, equivalently, there cannot be a second recursive call in `solve`. This means, there are just the tests in the first recursive call (where we have hidden dimension 0 again), plus the one test we do to find out that B is indeed optimal. This explains the second inequality of the lemma.

At this point it should be pointed out why the parameter k is called *hidden* dimension: namely, the algorithm does not know it. In particular, it does not know that the first recursive call and also the subsequent optimality test are not necessary in case of $\dim(R, B) = 0$.

If $r > m$ and $k > 0$, the first recursive call solves a problem of size $r - 1$ and hidden dimension at most k again. This follows from the fact that if $i \in R \setminus \{j\}$ is enforced in (R, B), it is also enforced in $(R \setminus \{j\}, B)$, because

$$z(B) > \mathrm{opt}(R \setminus \{i\}) \geq \mathrm{opt}(R \setminus \{i, j\}).$$

Put another way, if B is better than the best party without $i =$ Susan, then B is of course better than the best party without Susan and $j =$ Franz. When we come back from the first recursive call, we perform one optimality test.

In case we actually get into the second recursive call, we know that j is enforced now, because

$$z(B'') > z(B') = \mathrm{opt}(R \setminus \{j\}).$$

This is obvious: we have just computed (recursively) the best party B' without $j =$ Franz, and B'' is better. On the other hand, all elements enforced in (R, B) are still enforced in (R, B''), because $z(B'') > z(B)$ – enforeced elements can never leave the basis again. This means, the second recursive call has size r again, but hidden dimension at most $k - 1$.

Moreover, the second recursive call happens only with probability at most $k/(r - m)$, and that's the punchline of the whole randomized approach here. Why? Let $B(R)$ be the

optimal basis (which the algorithm doesn't know, of course). If the index j chosen in the first recursive call is not in $B(R)$, then $B(R) = B(R \setminus \{j\})$, because then $B(R) \subseteq R \setminus \{j\}$ implies

$$z(B(R)) = \text{opt}(R) \geq \text{opt}(R \setminus \{j\}) \geq z(B(R)) \quad \Rightarrow z(B(R)) = \text{opt}(R \setminus \{j\}).$$

This means, the second recursive call can *only* happen if $j \in B(R)$. Because j was chosen randomly from $r - m$ elements, this probability is at most $m/(r-m)$. But more is true: as we only have at most k free slots left in $B(R)$ ($m - k$ elements of $B(R)$ are already enforced, i.e. they are in B and do not qualify as the index j), the probabilty is actually at most $k/(r-m)$, and this explains the third inequality of the lemma. □

In this lemma, we have silently used some easy probability theory. The quantity $t_k(r)$ is actually the expectation of a random variable. In expressing it as a combination of other expectations, we have used the linearity of expectation. Moreover, the value $t_{k-1}(r)$ is actually a bound for the expected performance of the second recursive call, conditioned on the event that a j causing such a call was chosen. Thus, it is a conditional expectation, and to get the true expectation, we have multiplied it with the probabilty that this event occurs.

Theorem 4.5

$$t_k(r) \leq \sum_{i=0}^{k} \frac{1}{i!} k!(r-m) \leq e\,k!(r-m),$$

where e is Euler's constant $e = 2.71828\ldots$.

Thus, `solve` only needs an expected linear number of optimality tests to solve an LP with a constant number m of constraints.

Proof. By induction, where the bound obviously holds for $r = m$ (where we get 0) and $k = 0$ (where we get $r - m$). For $r > m, k > 0$ we inductively get

$$\begin{aligned}
t_k(r) &\leq t_k(r-1) + 1 + \frac{k}{r-m} t_{k-1}(r) \\
&\leq \sum_{i=0}^{k} \frac{1}{i!} k!(r-1-m) + 1 + \frac{k}{r-m} \sum_{i=0}^{k-1} \frac{1}{i!} (k-1)!(r-m) \\
&= \sum_{i=0}^{k} \frac{1}{i!} k!(r-1-m) + 1 + \sum_{i=0}^{k-1} \frac{1}{i!} k! \\
&= \sum_{i=0}^{k} \frac{1}{i!} k!(r-1-m) + \sum_{i=0}^{k} \frac{1}{i!} k! \\
&= \sum_{i=0}^{k} \frac{1}{i!} k!(r-m).
\end{aligned}$$

□

This bound can still be improved, by the following idea: so far we have used only that j is enforced in the second recursive call, but on average, much more new elements get enforced. To see this, consider the (at most k, as we know) elements $j_1, \ldots, j_k$ whose choice as j would lead to a second recursive call. Assume we have ordered them such that

$$\mathrm{opt}(R \setminus \{j_1\}) \leq \mathrm{opt}(R \setminus \{j_2\}) \leq \cdots \leq \mathrm{opt}(R \setminus \{j_k\}).$$

If $j = j_\ell$ is chosen, we get that not only j_ℓ, but also $j_1, \ldots, j_{\ell-1}$ are enforced in (R, B''), because

$$z(B'') > z(B') = \mathrm{opt}(R \setminus \{j_\ell\}) \geq \mathrm{opt}(R \setminus \{j_t\}),$$

for all $t = 1, \ldots, \ell$. In this case, $\dim(R, B'') \leq \dim(R, B) - \ell \leq k - \ell$. This means, the third inequality in Lemma 4.4 can be replaced by the inequality

$$t_k(r) \leq t_k(r-1) + 1 + \frac{1}{r-m} \sum_{\ell=1}^{k} t_{k-\ell})(r),$$

and one can prove (again by an easy induction) that this leads to an improved bound of

$$t_k(r) \leq 2^k (r - m).$$

Summarizing this section, we get the following

Theorem 4.6 *Any linear program $LP = LP(Q)$ with $n+m$ variables and constantly many inequalities can be solved in expected time $O(n)$, by a call to the algorithm* `solve`(Q, B).

Proof. Convince yourself that the overall number of steps done by the algorithm is proportional to the number of optimality tests. This holds for constant m, because then also a pivot step can be implemented in constant time ($O(m^3)$, actually, or even $O(m^2)$ if one does it slighly more cleverly and just updates the basis inverse A_B^{-1} upon a pivot step). Then the runtime is bounded by $O(t_m(n+m)) = O(n)$. □

This result is remarkable, because only in the eighties, an $O(n)$ algorithm for LP was first discovered. This was a determinstic (and actually quite complicated) algorithm whose runtime is of the order

$$2^{2^m} n,$$

which is really slow, even for small m. Only using randomization, it was possible to find an easier and also faster algorithm.

As m grows, even the algorithm we have developed here becomes slow, as we have an exponential factor of 2^m in the runtime. This is a general problem: no simplex variant is known whose runtime is polynomial in both parameters n and m. To find such an algorithm is a major open problem in the theory of linear programming.

4.2 LP-type problems

Reconsidering Algorithm 4.2, we find that not much of the linear programming structure was used. The party example we have been pulling out of the hat didn't have much to do with LP, and yet we were able to explain some principles with it. What we actually used was that we have a ground set Q, subsets R of which have optimal values $\text{opt}(R)$. We also have bases B, which were small subsets, and we have bases of sets R, which were small subsets B of R such that $\text{opt}(B) = \text{opt}(R)$. Then we have an optimality test which certifies whether a basis is the basis of some set, and a pivot step to find an improved basis, in case the basis was not optimal yet. In fact, there are problems more general than LP, for which all these notions make sense, and for which the algorithm `solve` will work.

Definition 4.7 *Let Q be a finite set, $w : 2^Q \mapsto \mathbb{R} \cup \{-\infty\}$ a function that assigns values to subsets of Q. The pair (Q, w) is called an* LP-type problem *if the following axioms are satisfied.*

(i) Monotonicity: $w(R) \leq w(S)$ for all $R \subseteq S \subseteq Q$

(ii) Locality: if we have $B \subseteq R \setminus \{j\} \subseteq Q$ such that $w(B) = w(R \setminus \{j\}) \neq -\infty$, and $w(R) > w(R \setminus \{j\})$, then also $w(B \cup \{j\}) > w(B)$.

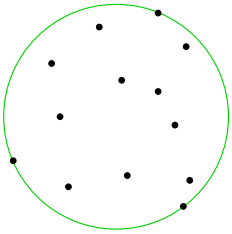
If we have an LP over variables with indices in Q, and we set $w(R) = \text{opt}(R)$ for all R such that $|R| \geq m$ and $\text{LP}(R)$ has a solution ($w(R) = -\infty$ otherwise), we get an LP-type problem. The monotonicity property is quite obvious: $\text{LP}(S)$ has less restrictions than $\text{LP}(R)$, so its maximum objective function value can only be higher.

Locality is more subtle. As its name indicates, this property guarantees that we can check optimality *locally*. To understand what this means, look at the algorithm `solve` again. When we come back from the first recursive call, we have $\text{opt}(B') = \text{opt}(R \setminus \{j\})$, and we want to know whether $\text{opt}(R) > \text{opt}(R \setminus \{j\})$. Locality tells us that this can only be the case if also $\text{opt}(B' \cup \{j\}) > \text{opt}(B')$. In fact, this is exactly what we exploited in the algorithm: If B' is not optimal for R, we find a new and improved basis $B'' \subseteq B' \cup \{j\}$, so B' was not even optimal for $B' \cup \{j\}$. Equivalently, $\text{opt}(B' \cup \{j\}) > \text{opt}(B)$.

This means, we check whether B' is optimal for R by checking whether it is optimal for $B' \cup \{j\}$. It might be obvious to you that this is the same thing – after all, that's how the simplex algorithm is designed. However, if the LP is degenerate, locality might fail! Namely, if entering j into the basis after the first recursive call does not lead to progress in the objective function, we get $\text{opt}(B'') = \text{opt}(B' \cup \{j\}) = \text{opt}(B')$, even though we might have $\text{opt}(R) > \text{opt}(R \setminus \{j\})$. In order to make progress again, we need to consider more elements than just the ones in $B' \cup \{j\}$.

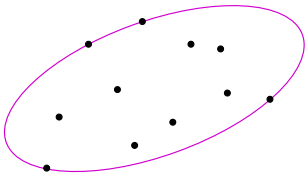
Therefore, only a nondegnerate LP gives rise to an LP-type problem. What are other LP-type problems? I promised that it does not stop with LP. Here are a few examples.

Smallest enclosing ball. Given n points in $\mathbb{R}^d$, find the ball of smallest volume covering all the points.



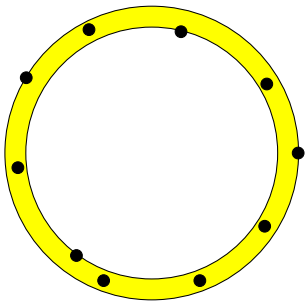
$\delta \leq d + 1$

Smallest enclosing ellipsoid. Given n points in $\mathbb{R}^d$, find the ellipsoid of smallest volume covering all the points.



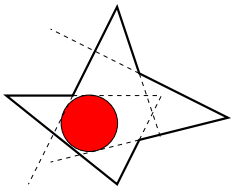
$\delta \leq d(d+3)/2$

Smallest enclosing annulus Given n points in the plane, find the annulus of smallest area covering all the points.



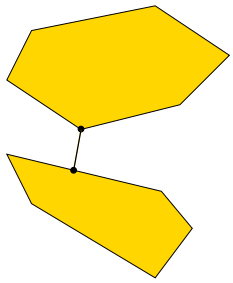
$\delta = 4$

Largest disk in kernel. Given a starshaped polygon with n edges, find the largest disk in the kernel of the polygon.



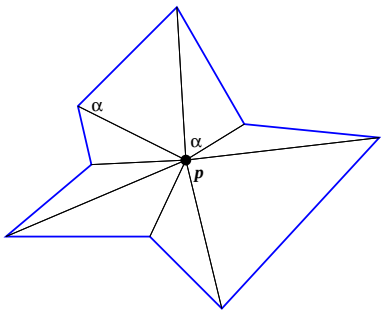
$\delta = 3$

Distance of polytopes. Given two polytopes defined by overall n halfspaces (or n points), find the shortest distance between them.



$\delta \leq d + 2$

Angle-optimal placement. Given a starshaped polygon with n edges, find the point p in its kernel such that the smallest angle in the graph consisting of the polygon edges and the connections between p and the vertices is maximized.



$\delta = 3$

To explain what the δ-value in these figures mean, here is some more notation, motivated by the LP case.

Definition 4.8 *Let (Q, w) be an LP-type problem.*

(i) $B \subseteq Q$ *is a* basis *if* $w(B) > w(B \setminus \{j\})$ *for all* $j \in B$. *Thus, a basis is an inclusion-minimal set which defines some value.*

(ii) $B \subseteq R$ *is* basis of R *if* B *is a basis and* $w(B) = w(R)$.

(iii) $\delta := \max_{B \text{ Basis}} |B|$ *is called the* combinatorial dimension *of* (Q, w).

The combinatorial dimension of an LP with m constraints is therefore m, because all bases have size m. In other LP-type problems, bases can have different sizes. If you look at the smallest enclosing ball problem, a basis of a set is (in case of general position of the points) exactly the set of points on the boundary of the smallest enclosing ball. In the picture above, this number is three, but it can as well be two, if the two boundary

points form the diameter of the smallest enclosing ball. In arbitrary dimension d, the combinatorial dimension is at most $d+1$, because this many points determine a unique ball.

For the other LP-type problems I have listed, we also have bounds for δ which are related to the dimension of the space the problem 'lives' in. The crucial insight here is that usually, this dimension is constant. After all, we are dealing with geometric problems here which mostly live in 2 or 3 dimensions. However, if the combinatorial dimension is constant, we may use the algorithm `solve` for our LP-type problem, and we will get expected $O(n)$ runtime.

This presumes that we know how to perform the optimality test and the pivot step, which are the only problem-specific primitives that remain. For most problems mentioned above, this is not difficult, and even if it is, we can still afford a brute-force solution. The primitive operations only involve constant-size problems, so they can be implemented in constant time.

Luckily, there are usually even efficient solutions. Let me discuss just one example. How do we perform the optimality test in case of smallest enclosing balls? The situation is that we have recursively found the smallest enclosing ball of the set $R \setminus \{j\}$, and we want to know whether the ball will get larger when we add j. It is intuitively clear (but must be proved, of course), that this is the case if and only if point j lies outside the smallest enclosing ball determined by $R \setminus \{j\}$. This test is very simple to execute in all dimensions.

Summarizing the discussion here, we obtain the following

Theorem 4.9 *An LP-type problem of constant combinatorial dimension δ over an n-element set Q can be solved in expected time $O(n)$ using the algorithm `solve`. In particular, we obtain $O(n)$ algorithms for all the problems mentioned above.*

A little care is in order: while the fact that bases may have different sizes does not affect the correctness of the algorithm `solve` it affects the runtime (Why? Exercise!). We still get a linear bound, though, only the dependence of the bound on the combinatorial dimension δ changes. If δ is constant, this makes no difference.

Theorem 4.9 is even more remarkable than the corresponding Theorem 4.6 for LP. Namely, for many LP-type problems (including smallest enclosing ellipsoids, for example), the algorithm `solve` was the *first* algorithm to achieve linear runtime. Given the fact that this algorithm uses hardly any problem-specific structure, this becomes even more amazing. The crucial insight here is that randomization really helps.

Chapter 5

The Primal-Dual Method

When we started out with Linear Programming, this was motivated by the SOLA problem; in fact, we had observed (or, rather, proved) that the problem of finding a weight-minimal perfect matching in a weighted complete bipartite graph over $2n$ vertices can be formulated as a linear program in n^2 variables (one for each edge) and $2n$ equality constraints (one for each vertex). Solving this problem using the simplex method will give us a solution, but the runtime is not satisfactory, at least from a theoretical point of view. The results of the previous chapter don't help much, because the number m of equality constraints is not constant. Using the procedure `solve`, we could compute an optimal asignment with an expected *exponential* number of roughly

$$2^{2n}n^2$$

steps. This is not what we were looking for.

This chapter introduces a method, the *primal-dual method*, which is capable of solving the assignment problem (and many other problems which can be formulated as LP) efficiently. Just like simplex, the primal-dual method is a general method for solving LP, but certain substeps may become much easier when the LP has some structure we understand well.

Assume we have a minimization LP in equality form, with n variables and m equality constraints, given as

$$\begin{array}{lll} \text{(LP)} & \text{minimize} & c^T x \\ & \text{subject to} & Ax = b, \\ & & x \geq 0. \end{array} \tag{5.1}$$

The primal-dual method can actually be described for LP in any form, but this form is practical, because it is the one assumed by the assignment problem. Its dual is given by

$$\begin{array}{lll} \text{(LP}^\Delta\text{)} & \text{maximize} & b^T y \\ & \text{subject to} & A^T y \leq c. \end{array} \tag{5.2}$$

This is not the standard form duality we have derived earlier, but it can easily be proved from it (as we have done in an exercise already).

The primal-dual method is based on the *complementary slackness theorem*, which we have also addressed in an exercise already, for the case of standard form LP. Let us redo it here for our formulations of (LP) and (LP$^\Delta$).

Theorem 5.1 *Let $\tilde{x}, \tilde{y}$ be feasible solutions to (LP) and (LP$^\Delta$). $\tilde{x}$ and $\tilde{y}$ are optimal solutions to (LP) and (LP$^\Delta$), respectively, if and only if the following* complementary slackness condition *holds:*

$$\tilde{x}_j = 0 \text{ or } (A^T\tilde{y})_j = c_j, \quad \forall j = 1, \ldots, n.$$

This means, if the j-th variable in the primal problem "has slack" (meaning that $\tilde{x}_j > 0$), then the j-th inequality in the dual must not have slack ($(A^T\tilde{y})_j > c_j$), and vice versa. This explains the name of the Theorem.

Proof. If $\tilde{x}$ and $\tilde{y}$ are feasible, it follows that

$$c^T\tilde{x} \geq \tilde{y}^T A\tilde{x} = \tilde{y}^T b. \tag{5.3}$$

Moreover, the inequality holds componentwise, i.e.

$$c_j\tilde{x}_j \geq \tilde{y}^T A_j\tilde{x}_j = (A^T\tilde{y})_j\tilde{x}_j. \tag{5.4}$$

Now it is easy to see that we have equality in (5.4) for all j (equivalently, equality in (5.3)) if and only if the complementary slackness condition holds. But $c^T\tilde{x} = \tilde{y}^T b$ is equivalent to simultaneous optimality of $\tilde{x}$ and $\tilde{y}$. It is necessary, because the optimal solutions of primal and dual must be equal, but it is also sufficient: $c^T x \geq \tilde{y}^T b$ and $c^T\tilde{x} \geq y^T b$ holds for all feasible solutions x and y to the primal and dual, respectively, so $\tilde{x}$ and $\tilde{y}$ are best possible solutions in both cases. □

5.1 Description of the Method

Assume we have some feasible solution $\tilde{y}^{(0)}$ to (LP$^\Delta$). Then define

$$J := \{j \mid (A^T\tilde{y}^{(0)})_j = c_j\}.$$

J contains the inidices of the dual inequalities which have no slack at $\tilde{y}$. By the complementary slackness theorem, if we find a feasible solution $\tilde{x}$ to the primal such that $\tilde{x}_j = 0$ for all $j \notin J$, then both $\tilde{x}$ and $\tilde{y}$ are optimal for their respective problems. Such an $\tilde{x}$ exists if and only if the system

$$\begin{aligned} A_J x_J &= b, \\ x_J &\geq 0 \end{aligned} \tag{5.5}$$

has a solution, where A_J contains the columns of A (and x_J the entries of x) corresponding to the variables $x_j, j \in J$. To test this, we solve the *restricted primal*

$$\begin{array}{lll} \text{(RP)} & \text{minimize} & \sum_{i=1}^m \lambda_i \\ & \text{subject to} & A_J x_J + E\lambda = b, \\ & & x_J \geq 0, \\ & & \lambda \geq 0. \end{array} \tag{5.6}$$

Here, $\lambda = (\lambda_1, \ldots, \lambda_m)^T$ is an auxiliary vector of m variables, one of which appears in every equality constraint (E is the $(m \times m)$ unit matrix). Assuming that $b \geq 0$ (which we can achieve by multiplying equalities in (LP) by -1 if necessary), we see that (RP) is feasible. Moreover, it has optimal solution 0 if and only if the system (5.5) has a feasible solution.

This means, if we solve (RP) and find that its optimal value is 0, our original problems (LP) and (LP$^\Delta$) are solved. If opt$(RP) > 0$, we have failed. Then, however, we can improve our original dual solution $\tilde{y}^{(0)}$ and repeat. To see this, consider the dual of (RP),

$$\begin{array}{lll} \text{(RP}^\Delta\text{)} \quad \text{maximize} & b^T y & \\ \text{subject to} & (A^T y)_j \leq 0, & j \in J \\ & y_i \leq 1, & i = 1, \ldots, m \end{array} \tag{5.7}$$

We have

$$\text{opt}(RP) > 0 \Leftrightarrow \text{opt}(RP^\Delta) > 0.$$

Let $\tilde{y}$ be an optimal solution to (RP$^\Delta$) such that $b^T \tilde{y} > 0$, and let us define an updated dual solution $\tilde{y}^{(1)}$ by

$$\tilde{y}^{(1)} = \tilde{y}^{(0)} + \Theta \tilde{y}, \quad \Theta > 0.$$

We get

$$b^T \tilde{y}^{(1)} = b^T \tilde{y}^{(0)} + \Theta b^T \tilde{y} > b^T \tilde{y}^{(0)},$$

so $\tilde{y}^{(1)}$ has indeed a better objective function value in (LP$^\Delta$) than $\tilde{y}^{(0)}$. It remains to show that $\tilde{y}^{(1)}$ is feasible for (LP$^\Delta$), and this holds at least for small values of Θ. Namely,

$$(A^T \tilde{y}^{(1)})_j = (A^T \tilde{y}^{(0)})_j + \Theta (A^T \tilde{y})_j \begin{cases} = c_j + \Theta (A^T \tilde{y})_j & \leq c_j, \quad j \in J, \text{ by (RP}^\Delta\text{)}, \\ & \leq c_j \quad j \notin J, \Theta \text{ small enough.} \end{cases} \tag{5.8}$$

The second inequality holds for Θ small enough because $(A^T \tilde{y}^{(0)})_j < c_j$ for $j \notin J$. The value Θ is chosen as large as possible subject to the requirement that $\tilde{y}^{(1)}$ is feasible for (LP$^\Delta$). It may happen that there is no upper bound for Θ, in which case (LP) must be infeasible: if there is no upper bound for the objective function value in (LP$^\Delta$), (LP) can have no feasible solution $\tilde{x}$, because then $c^T \tilde{x}$ would be an upper bound for the dual objective function value.

If there is a bound for Θ, we get at least one index $j = j_{new} \notin J$ such that $A^T \tilde{y}^{(1)}_j = c_j$, so we have a new dual inequality without slack. We repeat the above steps (which we call an *iteration* of the method), this time starting with

$$J := \{j \mid (A^T \tilde{y}^{(1)})_j = c_j\}.$$

Theorem 5.2 *The primal-dual method terminates in a finite number of iterations, if some care is taken.*

"Some care" means that we cannot just take *any* optimal solution $\tilde{y}$ of (RP$^\Delta$) to perform the update, in case this optimal solution is not unique. The fact that the dual objective

funtion value strictly increases in every iteration does not mean that we are done after finitely many iterations. For example, if (LP^Δ) is unbounded, it is conceivable that we are able to increase the value infinitely often. But even if (LP^Δ) is bounded, it may happen that the dual solution converges to a value *strictly* smaller than the true optimal value. Although such pathological examples require irrational coordinates, we cannot ignore them in theoretical considerations. After these words of warning, the reader may appreciate the care we are going to take to make Theorem 5.2 true.

Proof. We will argue that we can always choose an optimal solution $\tilde{y}$ to (RP^Δ) in such a way that no set J appears twice during the algorithm. Because there are only finitely many sets J, this proves the Theorem.

For this, consider the unique lexicographically largest optimal solution $\tilde{y}$ to (5.7).[1] Because all variables y_i are bounded by 1, $\tilde{y}$ exists. Equivalently, $\tilde{y}$ is the unique optimal solution to the *perturbed* problem where we replace b with

$$\hat{b} = b + (\varepsilon, \varepsilon^2, \ldots, \varepsilon^m)^T,$$

ε standing for an arbitrarily small positive constant. Recall that in the description of the simplex method, we have used the same manouvre to get rid of degeneracies in the primal problem (RP).

Now define

$$B := \{j \mid (A^T\tilde{y})_j = 0\}.$$

Then $\tilde{y}$ is also the lexicographically largest optimal solution to the problem

$$\begin{array}{lllll} (\mathrm{RP}^\Delta(\mathrm{B})) & \text{maximize} & b^T y & & \\ & \text{subject to} & (A^T y)_j \le 0, & j \in B & \\ & & y_i \le 1, & i = 1, \ldots, m, & \end{array} \tag{5.9}$$

because inequalities with slack at the optimal solution can be removed without affecting the optimum. (Equivalently, if $\tilde{x}_j = 0$ at an optimal solution in (RP), the variable can be removed from the problem without affecting the optimum.)

Now consider the next set J' we get after the dual solution update $\tilde{y}^{(0)} \to \tilde{y}^{(1)}$. We have $J' \supset B$, as we easily deduce from (5.8). This implies $b^T\tilde{y}' \le b^T\tilde{y}$, for $\tilde{y}'$ the optimal solution of (RP^Δ) in the next iteration, because we will have at least the constraints of (5.9). But J' also contains the new index $j_{new} \notin J$. By construction of j_{new}, we have

$$(A^T\tilde{y})_{j_{new}} > 0,$$

which means that $\tilde{y}$ is not feasible for (RP^Δ) in the next iteration. This again implies $\hat{b}^T\tilde{y}' < \hat{b}^T\tilde{y}$, because either $b^T\tilde{y}' < b^T\tilde{y}$ already holds, or $\tilde{y}'$ is lexicographically smaller than $\tilde{y}$.

The conclusion is that $\hat{b}^T\tilde{y}$ *strictly* decreases in every iteration, which means that no set J can appear twice. □

A couple of questions remain:

[1] $\tilde{y}$ is lexicographically larger than $\tilde{y}'$ if the first index i where they differ satisfies $\tilde{y}_i > \tilde{y}'_i$.

1. How do we solve (RP), or (RP$^\Delta$)? It seems that we have reduced (LP) to another linear program, so where is the gain? The gain is that (RP) is substantially simpler: it usually has less variables, and it does no longer depend on the original objective function vector c. In the applications, (RP) typically has a nice interpretation as a problem we already know how to solve without using LP techniques.

2. How many iterations does the method take? We have only shown that these are finitely many. Again, in the concrete applications we ususally get a quite good bound on the number of iterations. The next section will show the method in action and answer these questions at least for the (by now notorious) SOLA problem.

Also, there is another important observation: In certain types of problems, in order to update the dual solution, we may not necessarily need an optimal solution to (RP$^\Delta$) – a suitable solution with objective function value larger than 0 will do. Such a solution might be much easier to find than the actual optimal one.

5.2 The SOLA problem revisited again

Let us apply the primal-dual method to the assignment problem now and see what happens. We have the following primal and dual linear programs, given the complete bipartite graph $G = (V, E)$.

$$\begin{array}{lll} \text{(LP)} \quad \text{minimize} & \sum_{e \in E} \omega_e x_e & \\ \text{subject to} & \sum_{e \ni v} x_e = 1, & \forall v \in V, \\ & x_e \geq 0, & \forall e \in E. \end{array}$$

The constraints $x_e \leq 1$ we originally had are redundant because of $\sum_{e \ni v} x_e = 1$. Recall that ω_e is the weight of the edge e, and that an optimal (0/1)-solution to the problem is a weight-minimal perfect matching (assignment).

$$\begin{array}{lll} \text{(LP}^\Delta\text{)} \quad \text{maximize} & \sum_{v \in V} y_v & \\ \text{subject to} & y_v + y_w \leq \omega_e, & \forall \{v, w\} = e \in E. \end{array}$$

For a feasible solution $\tilde{y}$ to (LP$^\Delta$), we define

$$J := \{\{v, w\} \in E \mid \tilde{y}_v + \tilde{y}_w = \omega_e\},$$

labeling the inequalities directly with their defining edges. We then get

$$\begin{array}{lll} \text{(RP)} \quad \text{minimize} & \sum_{v \in V} \lambda_v & \\ \text{subject to} & \sum_{J \ni e \ni v} x_e + \lambda_v = 1, & \forall v \in V, \\ & x_e \geq 0, & \forall e \in J. \end{array}$$

and the dual problem

$$\begin{array}{lll} \text{(RP}^\Delta\text{)} \quad \text{maximize} & \sum_{v \in V} y_v & \\ \text{subject to} & y_v + y_w \leq 0, & \forall \{v, w\} \in J, \\ & y_v \leq 1, & \forall v \in V. \end{array}$$

Now we already see an important feature of the primal-dual method: when we apply it to a concrete problem, we often have a nice interpretation of (RP) or (RP$^\Delta$). This is also the case here.

Lemma 5.3

(i) Both (RP) and (RP$^\Delta$) have integer optimal solutions.

(ii) opt(RP) = opt(RP$^\Delta$) is the number of free *(unmatched) nodes in a matching of largest cardinality in (V, J).*

This means, the primal-dual method in this case reduces a *weighted* matching problem to a series of *unweighted* ones.

Proof. The fact that we have optimal integer solutions can again be proved 'manually', as we first did it for (LP) back in Chapter 3. However, this time we simply invoke the fact that the constraint matrices of both problems are TUM (totally unimodular), see exercises and solutions to them. This guarantees the integer solutions. As the matrices are in both cases vertex-edge incidence matrices of a bipartite graph (with some extra columns or rows which contain only one entry), the TUM property is particularly easy to check.

An optimal solution $(\tilde{x}, \tilde{\lambda})$ to (RP) is then obviously a 0/1-vector. The edges M with $\tilde{x}_e = 1$ define a matching, because every vertex is incident to at most one of them by the constraint

$$\sum_{J \ni e \ni v} \tilde{x}_e \leq \sum_{J \ni e \ni v} \tilde{x}_e + \tilde{\lambda}_v = 1.$$

Moreover, exactly the vertices with $\tilde{\lambda}_v = 1$ are free; there is no matching with less free vertices, because

$$\sum_{v \in V} \tilde{\lambda}_v$$

is minimal under the given restrictions. Therefore, M is a largest matching, and the objective function value is the number of free vertices with respect to M. □

We see that if M is a perfect matching, our problem is solved. In this case, M is a weight-minimal perfect matching. Otherwise, we have to update our dual solution.

In the proof of Theorem 5.2, we have seen that it matters which optimal solution $\tilde{y}$ to (RP$^\Delta$) we will use in the update. We will use a special one which we can directly read off, once we know the largest matching M. For this, we need some more terminology.

Definition 5.4 *Let $M \subseteq J \subseteq E$ be a matching.*

(i) A path $P \subseteq J$ is called alternating *with respect to M, if it alternates between edges in M and in $J \setminus M$ (or vice versa), and if one of its endpoints is a free vertex.*

(ii) An alternating path P is called augmenting *if both its endpoints are free vertices.*

Then we have the following

Lemma 5.5 *Let M be a largest matching with edges in J and v a matched vertex. Either all alternating paths starting at v have even length, or all these paths have odd length.*

Proof. Assume that for some vertex, we have paths P_1 of even and P_2 of odd length. P_1 starts with a matching edge, while P_2 does not. Moreover, the paths share no vertex w other than v, because such a common w would have two edges of M incident to it, unless it is the last (free) vertex on one of the paths. Then, w must also be the last vertex on the other path, because it has no edge of M incident to it. Then, however, P_1 and P_2 would have the same length modulo two, because we have a bipartite graph.

Now, combining P_1 and P_2 gives an augmenting path P. This path has one more edge in $J \setminus M$ than in M, and by "inverting" the path (interchanging matching- and non-matching-edges), we obtain a larger matching than M, a contradiction. □

Now we can define our special optimal solution $\tilde{y}$ to (RP$^\Delta$).

Lemma 5.6 *Let M be a largest matching with edges in J and define $\ell(v)$ as the length of a shortest (or of some) alternating path starting at v ($\ell(v) := \infty$ in case no such path exists). Then the vector $\tilde{y}$ defined by*

$$\tilde{y}_v = \begin{cases} 0, & \text{if } \ell(v) = \infty \\ (-1)^{\ell(v)}, & \text{otherwise} \end{cases} \tag{5.10}$$

is an optimal solution to (RP^Δ).

Remark: Here $\ell(v) = 0$ is also allowed. Therefore a free vertex v always has $\tilde{y}_v = 1$. Because even if it is isolated in J, it has an alternating path of length 0 to itself.

Proof. From Lemma 5.5, we know that $\tilde{y}$ is well-defined. To prove that $\tilde{y}$ is feasible, consider an edge $e = \{v, w\} \in J$. If $\tilde{y}_v, \tilde{y}_w \leq 0$, the inequality for e is satisfied, so assume without loss of generality that $\tilde{y}_v = 1$. This means, from v, there is an alternating path P of even length, starting with a matching edge (or v is a free vertex).

Case (a) $e \in M$**.** Then $P \setminus \{e\}$ is an alternating path of odd length, starting at w, so we get $\tilde{y}_w = -1$, and the inequality for e holds.

Case (b) $e \notin M$**.** Then $P \cup \{e\}$ is an alternating path of odd length, starting at w, so again we get $\tilde{y}_w = -1$.

Optimality of $\tilde{y}$ is easy now: for every free vertex v, we get $\tilde{y}_v = 1$. For every matching-edge $\{v, w\}$, we either have no alternating paths from both vertices, or alternating paths of even length from one of them and odd length from the other. In any case, we get $\tilde{y}_v + \tilde{y}_w = 0$. Let F be the set of free vertices. Because every vertex is either free or matched exactly once, we obtain

$$\sum_{v \in V} \tilde{y}_v = \sum_{v \in F} \tilde{y}_v + \sum_{\{v,w\} \in M} (\tilde{y}_v + \tilde{y}_w) = \sum_{v \in F} \tilde{y}_v = |F| = \text{opt}(RP) = \text{opt}(RP^\Delta).$$

□

Now consider the update $\tilde{y}^{(0)} \to \tilde{y}^{(1)}$, and let $e = \{v, w\} \notin J$ be some edge such that

$$\tilde{y}_v^{(1)} + \tilde{y}_w^{(1)} = w_e = \underbrace{(\tilde{y}_v^{(0)} + \tilde{y}_w^{(0)})}_{<w_e} + \Theta \underbrace{(\tilde{y}_v + \tilde{y}_w)}_{\geq 1}.$$

There are two cases:

Case (a) $\tilde{y}_v = 1, \tilde{y}_w = 1$**.** Then, alternating paths P_1 and P_2 of even length start at both v and w. As above, these paths can be combined to an augmenting path P using the new edge $\{v, w\}$. Moreover, all edges $e = \{t, u\} \neq \{v, w\}$ on P satisfy $\tilde{y}_t + \tilde{y}_u = 0$ (the signs alternate along P_1 and P_2), which means that $P \subseteq J'$, J' the edge set of the next iteration. This follows from (5.8). Also, M, the currently largest matching is contained in J', because we have derived $\tilde{y}_v + \tilde{y}_w = 0$ for all matching-edges. In other words, the next set J' contains M and an augmenting path P for M, so the largest matching M' in the next iteration will have one more edge. This means, the optimal objective function values of (RP) and (RP$^\Delta$) will decrease by two in the next iteration.

Case (b) $\tilde{y}_v = 1, \tilde{y}_w = 0$ **(or vice versa).** In this case, the new edge joins some vertex with a vertex that was not connected to any free vertex along an alternating path. In particular, the new edge cannot be part of any augmenting path. This, however, means that the largest matching will not change in the next iteration (exercise). This is the case that we have found to be potentially dangerous in the previous section. Namely, if opt(RP^Δ) does not decrease throughout the iterations, the whole method might not terminate. In this case, however, there is no danger of cycling, because the number of zeroes in the solution vector $\tilde{y}$ will strictly go down (by two at least), so that we must be in case (a) after at most n iterations.

It may happen that more than one new edge will appear in J'. Then we can process them one at a time in any order.

It follows that the method will take at most n^2 iterations: the matching can get larger only n times, and between two such augmentation steps, we can have up to n non-augmenting iterations. To analyze the cost of a single iteration, let us summarize the steps we have to do. We may assume that all weights ω_e are positive and that our initial dual solution is $\tilde{y}^{(0)} = 0$ which entails $J = \emptyset, M = \emptyset$ and $\tilde{y}_v = 1, v \in V$, for the solution $\tilde{y}$ to (RP$^\Delta$) in the first iteration. Then we have to perform the following computations in each step.

(i) Compute Θ, update the dual variables, and compute the set J' of edges for the next iteration. If J' contains an augmenting path for M, augment M along this path.

(ii) Compute the solution to (RP$^\Delta$) according to (5.10).

Figure 5.1 shows how the primal-dual method works for a small example. To the left, we see the input graph. The left column of the table shows the varying optimal solutions to

(RP$^\Delta$) throughout the iterationss, also indicating the edges of the current set J (edges in the matching M appear in bold).

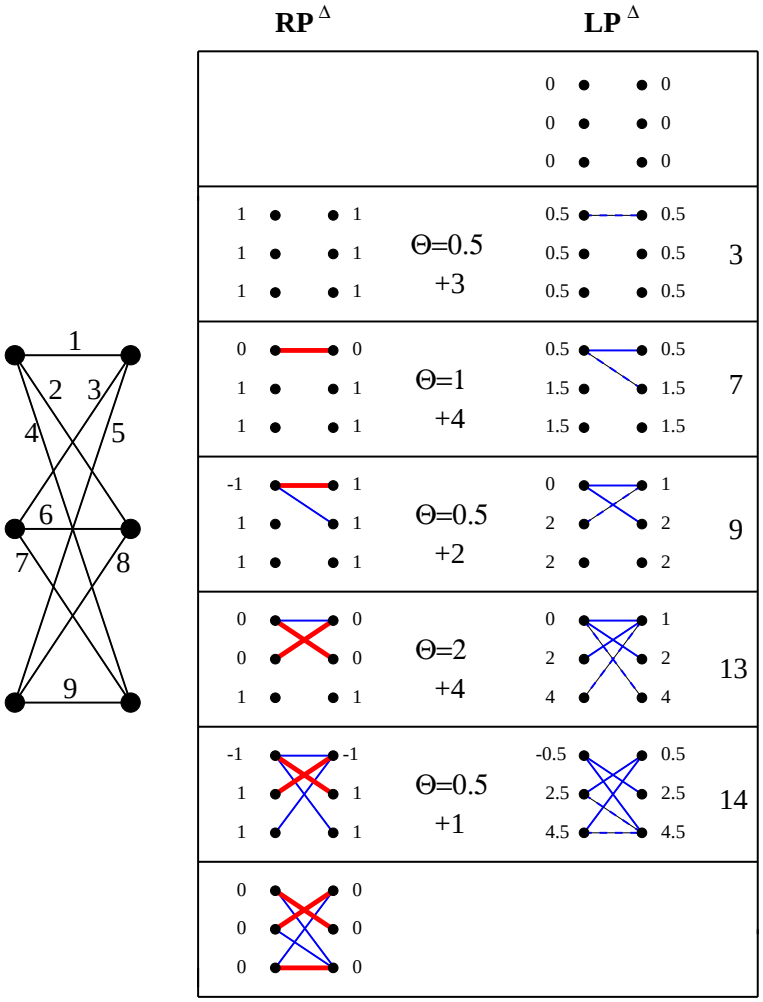


Figure 5.1: The primal-dual method for the assignment problem

On the right, we always see the updated dual solution, along with the next set of edges J' (new edges dashed), and its objective function value (strictly increasing in every iteration). The value Θ used for the update, along with the resulting increase in the dual objective function value is indicated for all iterations. The top entry in the right column shows the initial dual solution $\tilde{y}^{(0)}$, while the bottom entry in the left column corresponds to the final instance of (RP$^\Delta$) with objective function value 0 (all vertices are matched).

Step (i) above can be obviosuly performed in $O(n^2)$ time, by going through all edges (for finding Θ and J') and all vertices (for updating the dual solution). Augmenting the matching M, if possible, can also be done within these time bounds, because we can easily find the augmenting path, by tracing it to both its ends from v and w, the two vertices such that $\tilde{x}_v = \tilde{x}_w = 1$.

To implement step (ii), we must compute $\tilde{y}$ according to (5.10), given the (possibly updated) matching M. This means, for every vertex we need the shortest (or any, in fact) alternating path starting at that vertex. This can be found by a graph search in an auxiliary graph. To find the shortest alternarting paths to free vertices on the left, we introduce a new vertex s, which is connected by directed edges to all free vertices on the left. Every edge in J' also receives a direction: it points to the right for non-matching edges, and to the left for matching edges. Then all directed paths in this directed graph that start at s correspond to alternating paths with respect to M. Using breadth-first search (BFS), we can find the shortest alternating paths (and in particular their lengths) to all vertices in $O(n^2)$ time. Repeating the same procedure with an additional vertex t connected to all free vertices on the right, we find the overall shortest augmenting paths for all vertices. This gives us the desired solution $\tilde{y}$ in time $O(n^2)$.

This means, we have at most n^2 iterations, each of which can be performed in $O(n^2)$ time, so that we get

Theorem 5.7 *The primal-dual method can be used to solve the assignment problem over a complete weighted bipartite graph with $2n$ vertices in time $O(n^4)$.*

An improvement to $O(n^3)$ is possible, by observing that a non-augmenting iteration can actually be implemented in $O(n)$ time, using some data structures to find Θ faster, and showing that the next $\tilde{y}$ can be obtained in amortized time $O(n)$, because coordinates of value zero that have been processed in a non-augmenting step disappear afterwards. The resulting method is known as the *Hungarian Method* for the assignment problem.

Chapter 6

Complexity

Throughout this lecture, we have been talking about "efficient" algorithms, and about "inefficient" ones, where we have never formally defined what we mean by this. For example, the first algorithm we had for the SOLA (assignment) problem was just the simplex algorithm applied to the LP formulation of the problem, which has n^2 variables and $2n$ constraints. Unfortunately, the complexity result we have proved for simplex is only good if there are few constraints; in this case, we get a runtime of at least

$$2^{2n}n^2$$

steps, which is not efficient. On the other hand, the primal-dual method of the previous chapter allows us to solve the problem in time

$$O(n^4),$$

which might not be utterly efficient, either, but much better than the exponential bound.

In this chapter, we want to give a definition of efficiency; we will call an algorithm efficient, if its runtime is bounded by a polynomial function in the input size. Whether this agrees with the concept of practical efficiency that you may have in mind, is another issue we are not going to discuss here. Under this definition, the primal-dual method for solving the assignment problem is efficient, and we would also call an $O(n^{1000})$ algorithm efficient. Luckily, typical efficient algorithms we encounter in practice have small exponents in their runtime bound (see for example breadth-first search or Dijkstra's algorithm, and even the improvement to $O(n^3)$ that is possible in case of the assignment problem).

We also want to consider the question which problems allow efficient algorithms and which don't.

6.1 The classes P and NP

Every *optimization* problem has an associated *decision* problem. For example, the problem of finding a largest independent set in a graph has the following decision variant.

(IS) Given a graph $G = (V, E)$ and a number k, does G have an independent set of size k?

The significance of decision problems lies in the fact that they can be written as *formal languages*; this allows us to consider a decision problem as a rigorously defined mathematical object.

For us, a formal language will be a set of finite bitstrings. Let $\{0,1\}^*$ denote the set of all bitstrings of finite length. Then any $L \subseteq \{0,1\}^*$ is a language. To define the language associated with (IS), we agree on some encoding of graph-number pairs (G, k) in form of bitstrings. Then L_{IS} is the set of syntactically correct encodings of pairs (G, k) with the property that G has an independent set of size k. In view of this, we can also refer to languages as decision problems.

An algorithm A for a decision problem L is then a Turing machine (or a `C++` program, `Oberon` program, assembler code, ...) with the following input/output behavior:

A :
Input: $w \in \{0,1\}^*$
Output "yes", if $w \in L$, "no" otherwise.

Definition 6.1 P *is the class of all languages L for which a Turing machine A and a constant c exist such that the runtime of A is of the order $O(|w|^c)$ for all $w \in \{0,1\}^*$. Here, $|w|$ is the length of w, and an $O(|w|^c)$ Turing machine is said to run in polynomial time.*

If you don't know what a Turing machine is, you may replace the term with `C++` program, `Oberon` program, assembler code, or any other concrete code, and you will get the same class **P**. This is because a Turing machine can simulate any program of any programming language, and to simulate a single step of the program, the Turing machine only needs polynomial time. In the following, we will simply refer to Turing machines as algorithms.

For us, **P** is the class of efficiently solvable decision problems. Is (IS) in **P**? Until today, nobody knows the answer, but there is a strong suspicion that this is not the case.

At least, (IS) is in a larger class of problems which are the ones that have efficient *verifiers*.

Definition 6.2 *A verifier for a language L is an algorithm*
V
Input: $w \in \{0,1\}^*$ and $b \in \{0,1\}^*$ (a "proof" for $w \in L$)
Output: $V(w, b) \in \{$"yes", "no"$\}$,

with the following two properties:

(a) for all $w \in L$, there exists a proof b such that $V(w, b) =$ "yes",

(b) for all $w \notin L$, $V(w, b) =$ "no", for all b.

A verifier can therefore certify $w \in L$, provided an appropriate proof b is available, but no proof can fool V into accepting a word $w \notin L$.

This leads us to the class **NP**.

Definition 6.3 NP *is the class of all languages L for which a verifier V and a constant c exist such that $V(w,b)$ is computed in time $O(|w|^c)$, for all $w, b \in \{0,1\}^*$.*

In particular, such a verifier can only look at polynomially many bits of the proof b.

Theorem 6.4 *(IS)*$\in$ **NP**.

Proof. We need to establish a polynomial-time verifier. On input w, b, the verifier first checks whether w is a syntactically correct encoding of a graph-number pair (G,k), and whether b is a syntactically correct encoding of a vertex set U of size k. Finally, it checks whether U is an independent set and delivers the answer. If $w \in L_{IS}$, such an independent set U exists, and its encoding is a proof b which leads to a "yes" answer. If $w \notin L$, b can never be the encoding of an independent set of size k, and the answer will always be "no". □

The largest open question in theoretical computer science is whether **P** = **NP**. We obviously have **P** $\subseteq$ **NP**, because a polynomial-time algorithm to solve a problem can be considered as a polynomial-time verifier which simply ignores b.

6.2 Polynomial-time reductions

Here we want to develop statements of the form: problem X is at least as hard as problem Y. The tool is the following

Definition 6.5 *Let $L_1, L_2 \subseteq \{0,1\}^*$ be languages and R an algorithm which for every input bitstring w produces an output bitstring $R(w)$. R is called a* polynomial-time reduction *from L_1 to L_2 if the following two properties hold.*

(a) $w \in L_1 \Leftrightarrow R(w) \in L_2$, and

(b) R runs in polynomial time.

If a polynomial-time reduction from L_1 to L_2 exists, we write $L_1 \leq_P L_2$.

The significance of this definition is expressed by the following

Lemma 6.6 *If $L_2 \in$ **P** and $L_1 \leq_P L_2$, then $L_1 \in$ **P**.*

Proof. Given some w for which we want to decide membership in L_1, we compute $R(w)$ in polynomial-time, and then decide membership of $R(w)$ in L_2, which we can also do in polynomial-time since $L_2 \in$ **P**. By property (a) above, we then have the desired answer for w. Since the composition of polynomial-time algorithms is a polynomial-time algorithm, it follows that $L_1 \in$ **P**. □

As an example for polynomial-time reducibility, we show that (3-SAT)$\leq_P$(IS), where (3-SAT) is the following problem.

(3-SAT) Given a boolean formula ϕ in conjunctive normal form, where every clause has exactly three literals coming from distinct variables, decide whether ϕ has a satsifying truth assignment.

Such a formula could look like this:

$$\phi = (x_1 \vee x_2 \vee \overline{x_3}) \wedge (x_1 \vee x_4 \vee \overline{x_2}) \wedge (\overline{x_1} \vee \overline{x_4} \vee x_2).$$

In this case, a satisfying truth assignment can be obtained for example by setting $x_1 = x_2 =$ true and the other two variables in an arbitrary way.

Lemma 6.7 *(3-SAT)$\leq_P$(IS).*

Proof. We need to construct a polynomial-time reduction from (3-SAT) to (IS). This means, given a formula ϕ, we need to construct a graph-number pair (G, k) such that ϕ has a satisfying truth assignment if and only if G has an independent set of size k.

G consists of triples of vertices, one triple for each clause of ϕ. The vertices within one triple are pairwise connected and labeled by the literals in the corresponding clause. Between vertices in different triples, we have an edge if and only if the vertex labels are opposite literals coming from the same variable. Figure 6.1 shows how G looks like for our example formula ϕ.

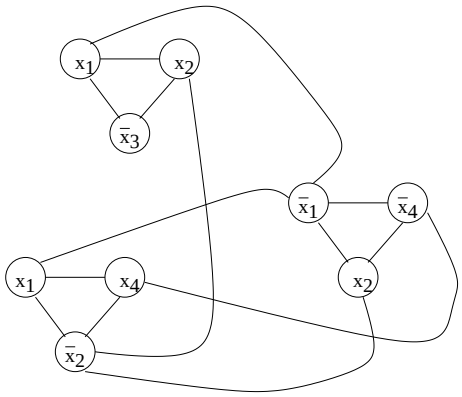


Figure 6.1: Reduction from (3-SAT) to (IS)

Finally, the number k will be the number of clauses of ϕ; in our example we get $k = 3$.

Now assume ϕ is satisfiable. Then there is one literal in every clause which is set to true in some satisfying assignment. The corresponding vertices in G (one from each triple) form an independent set of size k: no edge can connect two of them, because we cannot have both x_i and $\overline{x_i}$ equal to true for some i. On the other hand, if G has an independent set U of size k, this set must have exactly one vertex from every triple: there are only k triples, and no two vertices can be in the same triple, because then there would be an edge between them. Each vertex in U has an associated literal in ϕ, and when we set all of

them to true, we obtain a satisfying assignment. Namely, the assignment is well-defined, because we will never try to set both x_i and $\overline{x_i}$ to true, and it will have one satisfied literal in every clause. ◻

The consequence of this reduction is that if we could prove (IS)$\in \mathbf{P}$, we would automatically get (3-SAT)$\in P$. For both problems, no polynomial-time algorithms have ever been found.

6.3 NP-completeness and NP-hardness

The most difficult problems in **NP** are those to which *all* other problems in NP are polynomial-time reducible.

Definition 6.8 *A language $L \in \mathbf{NP}$ is called* **NP**-complete *if $L' \leq_P L$ for all languages $L' \in \mathbf{NP}$.*

Assuming that there exists some **NP**-complete problems L at all (which is not obvious), it has the property that if somebody finds a polynomial-time algorithm for L (i.e. proves $L \in \mathbf{P}$), then we get $\mathbf{P} = \mathbf{NP}$, i.e. *all* problems in **NP** are solvable in polynomial-time then. This follows directly from Lemma 6.6. In this sense, the **NP**-complete problems are indeed the most difficult ones in **NP**.

It has been shown by Cook and Levin in the seventies that there are NP-complete problems.

Theorem 6.9 *(3-SAT) is* **NP***-complete.*

The proof is not too difficult (once you have the idea), but quite technical. It employs an alternative (actually, the original) definition of **NP** as the class of problems that can be solved in polynomial time using a *nondeterministic* turing machine. Then it proves that any such machine can be "implemented" as a boolean formula.

To show that other languages are **NP**-complete is now much easier. In fact, we have already proved the following

Corollary 6.10 *(IS) is* **NP***-complete.*

Namely, we know that $(IS) \in \mathbf{NP}$. Moreover, every language $L \in \mathbf{NP}$ satisfies $L \leq_P (IS)$, because $L \leq_P (3-SAT)$ by Cook and Levin's Theorem, and $(3-SAT) \leq_P (IS)$ by Lemma 6.7. Moreover, the polynomial-time reducibility relation is transitive (as before: composition of polynomial-time algorithms is polynomial-time), so we get $L \leq_P (IS)$ for all $L \in \mathbf{NP}$, as desired.

This means, to prove that a problem is **NP**-complete, you have to show that it is in **NP**, and that there is some **NP**-complete problem which is polynomial-time reducible to it.

This has in fact been done for a host of problems (a few thousands, if you want a non-educated guess). For none of these problems, a polynomial-time algorithm could be found. The general belief is that no such algorithms exist and that $\mathbf{P} \neq \mathbf{NP}$.

So far, we have been talking about decision problems, but this course deals with optimization problems. How can we classify these with respect to **P** and **NP**? There is a couple definitions, differing in details. For our purposes, the following will do:

Definition 6.11 *An optimization problem is* **NP**-hard *if all problems in* **NP** *are polynomial-time reducible to the corresponding decision problem, and this decision problem is called* **NP**-*hard, too.*

Note that this does not require the corresponding decision problem to be **NP**-complete. There are some subtleties here which we might address later in the course. In any case, by this definition, the problem of finding the largest independent set in a graph is **NP**-hard, because its corresponding decision problem (IS) is **NP**-complete; in particular, the required polynomial-time reductions from all problems in **NP** exist.

6.4 Integer Linear Programming is NP-hard

Much of discrete optimization (in particular later parts of this course) are concerned with integer linear programs (ILP). ILP is **NP**-hard, so we cannot expect to find a polynomial-time algorithm that can solve any ILP instance.

Theorem 6.12 *(ILP) is* **NP**-*hard.*

Proof. We need to find a polynomial-time reduction from some **NP**-complete problem to the decision variant of (ILP). But we have already done this in Section 3.1.4.

Namely, any instance (G, k) of (IS) can be mapped to an (ILP) instance (I, k) such that G has an independent set of size k if and only if the ILP I has a feasible solution of value k. For $G = (V, E)$, we have used the following ILP:

$$\begin{array}{lllll} (\mathrm{ILP}_{IndepSet}(G)) & \text{maximize} & \sum_{v \in V} x_v & & \\ & \text{subject to} & x_v + x_w \leq 1, & \forall \{v, w\} \in E, \\ & & x_v \in \{0, 1\}, & \forall v \in V. \end{array}$$

□

6.5 How to deal with NP-hard problems

The fact that an optimization problem is **NP**-hard is no reason to give up, and in practice, one simply cannot afford to give up. The world of today demands solutions to many challenging **NP**-hard problems. They arise in the design of flight schedules, in microchip layouts, factory design, route planning, etc. So what can we do? There are basically two lines of attack:

(1) Approximation algorithms. These are efficient algorithms that do not necessarily find the optimal solution for all problem instances, but in any case a solution that has some *provable* quality. Below we will see such an algorithm for a special class of ILPs. This means, in order to gain efficiency, one is willing to sacrifice optimality. Often, the quality gurantees achieved by approximation algorithms are not good enough for practical applications.

(2) Heuristics. These are algorithms that try to find a (near) optimal solution, knowing that this might take long (or even too long). The art here is to carefully analyse the problem and the concrete instances at hand, and to find ways to exploit their structure in such a way that the solution is obtained fast. After all, the fact that a problem is **NP**-hard does not mean that every instance of it is hard. Heuristics try to make the class of instances that can be handled efficiently in practice as large as possible. Heuristics don't come with an a priori quality guarantee, but they typically deliver upper and lower bounds for the optimal solution that are very close together or even equal. We will discuss heuristics later in the course.

6.6 Approximation algorithms

Let us consider the *hitting set* (HS) problem: we are given a ground set H and a set $\mathcal{T}$ of subsets $T_1, \ldots, T_m \subseteq H$. The elements of H have positive weights $w_a, a \in H$. We are looking for a subset $A \subseteq H$ such that

- $A \cap T_i \neq \emptyset, i = 1, \ldots, m$, and
- $\sum_{a \in A} w_a$ is minimal

Such a set A is a weight-minimal *hitting set*, because it "hits" all subsets T_i.

This problem is a generalization of many known problems. Let us just give two examples:

1. *Vertex cover.* Given a graph $G = (V, E)$, find a smallest subset $C \subseteq V$ such that each edge $e \in E$ has one of its vertices in C. To formulate this a a hitting set instance, we set $H := V$, $\mathcal{T} := E$ and all weights equal to one.

2. *Minimum spanning tree.* Given a connected graph $G = (V, E)$ with positive edge weights $w_e, e \in E$, find a spanning tree with minimal sum of edge weights. In hitting set language, we are looking for a weight-minimal subset of $H := E$ which hits all *cuts* in the graph. A cut is defined by a subset $S \subseteq V$ of vertices, and consists of all the edges that connect elements of S with elements of $V \setminus S$. If a set of edges hits all cuts, it defines a connected, spanning subgraph. Minimizing the sum of edge weights ensures that we actually get a tree – the minnimum spanning tree.

Here is an ILP formulation of (HS). We have $x_a = 1$ if a appears in the hitting set, $x_a = 0$ otherwise.

$$\begin{array}{lll} (\mathrm{ILP}_{HS}) \quad \text{minimize} & \sum_{a \in H} w_a x_a & \\ \text{subject to} & \sum_{a \in T_i} x_a \geq 1 & i = 1, \ldots, m, \\ & x_a \in \{0, 1\}, & \forall a \in H. \end{array}$$

It can be shown that (HS) is **NP**-hard. We can even sketch the proof here. Let (VC) be the vertex cover problem (decisision version) as defined above. Obviously, we have (VC)$\leq_P$(HS), because (VC) is a special case of (HS). Moreover, (IS)$\leq_P$(VC), because in any graph, the complement of an independent set is a vertex cover, and vice versa. Because (IS) is **NP**-complete, it follows that (HS) is **NP**-hard.

We will develop an approximation algorithm for (HS) now. It will run in polynomial time, but will not necessarily deliver a weight-optimal hitting set. The algorithm is a variant of the primal-dual method that we already know. The primal-dual method is quite useful in approximation algorithms in general, and this is the reason why we discuss this application here. Specifically for the vertex cover problem, there are simpler approximation algorithms with the same provable quality.

Let us consider the LP-relaxation of the ILP above.

$$\begin{array}{lll} (\mathrm{LP}_{HS}) \quad \text{minimize} & \sum_{a \in H} w_a x_a & \\ \text{subject to} & \sum_{a \in T_i} x_a \geq 1 & i = 1, \ldots, m, \\ & x_a \geq 0, & \forall a \in H. \end{array}$$

The constraints $x_a \leq 1$ are redundant: if we want to minimize $\sum_{a \in H} w_a x_a$, it makes no sense to have $x_a > 1$ for some element. Now consider the dual of this LP.

$$\begin{array}{lll} (\mathrm{LP}^{\Delta}_{HS}) \quad \text{maximize} & \sum_{i=1}^{m} y_i & \\ \text{subject to} & \sum_{i:T_i \ni a} y_i \leq w_a & a \in H, \\ & y_i \geq 0, & i = 1, \ldots, m. \end{array}$$

Assume $\tilde{y}$ is a feasible solution for this dual. Primal-dual-like, we then consider the set of constraints which are satisfied with equality at $\tilde{y}$. These are constraints corresponding to elements $a \in J$ with

$$J := \{a \in H \mid \sum_{i:T_i \ni a} \tilde{y}_i = w_a\}$$

and constraints corresponding to indices

$$I := \{i \in \{1, \ldots, m\} \mid \tilde{y}_i = 0\}.$$

In the "classical" primal-dual method, we would now search for a feasible solution $\tilde{x}$ to the primal problem such that

$$\begin{array}{rcll} \tilde{x}_a & = & 0, \quad a \notin J, & \text{(primal slackness conditions)} \\ \sum_{a \in T_i} \tilde{x}_a & = & 1, \quad i \notin I, & \text{(dual slackness conditions)} \end{array}$$

If such a vector $\tilde{x}$ exists, $\tilde{x}$ and $\tilde{y}$ are both optimal in their respective problems, by complementary slackness. Unlike in our previous description of the primal-dual method, primal and dual LP are in standard form here, so we need to apply the complementary slackness theorem for standard-form LP (see exercise 5-2).

The problem is that we are not looking for a solution $\tilde{x}$ to (LP$_{HS}$) but to (ILP$_{HS}$). In an attempt to satisfy at least the primal slackness conditions, we define a 0/1-vector $\tilde{x}$ by

$$\tilde{x}_a := \begin{cases} 0, & \text{if } a \notin J \\ 1, & \text{if } a \in J. \end{cases}$$

Lemma 6.13 *If $\tilde{x}$ is not feasible for (ILP$_{HS}$), then* no *feasible solution $\tilde{x}'$ to (LP$_{HS}$) satisfies the primal slackness conditions.*

The test whether $\tilde{x}$ is feasible for (ILP$_{HS}$) plays the role of the restricted primal here. If $\tilde{x}$ is not feasible, we know that the restricted primal has nonzero optimum (by the Lemma), and therefore we can improve the dual solution $\tilde{y}$. On the other hand, if $\tilde{x}$ is feasible, we have no information about the solution of the restricted primal. However, it turns out that we can still stop in this case (like we would have done it in case the restricted primal solves to 0), and $\tilde{x}$ will be an approximation of the true optimal solution to (ILP$_{HS}$). But let us prove the lemma first.

Proof. If $\tilde{x}$ is not feasible for (ILP$_{HS}$), there exists a set T_k which is not hit, i.e.

$$\sum_{a \in T_k} \tilde{x}_a = 0.$$

We must have $T_k \cap J = \emptyset$, because $\tilde{x}_a = 1$ for $a \in J$. Now let $\tilde{x}'$ be any feasible solution for (LP$_{HS}$). Then

$$\sum_{a \in T_k} \tilde{x}'_a \geq 1$$

holds, which implies $\tilde{x}'_a > 0$ for some $a \in T_k \subseteq H \setminus J$, so the primal slackness conditions do not hold. □

To improve the dual solution, we proceed as follows: we only change (increase) the entry $\tilde{y}_k$ corresponding to the set T_k which was not hit:

$$\tilde{y}_k^{(1)} := \tilde{y}_k^{(0)} + \Theta.$$

We need to argue that this is still dual feasible for Θ small enough. For $a \in J$ we have

$$\sum_{i: T_i \ni a} \tilde{y}_i^{(1)} = \sum_{i: T_i \ni a} \tilde{y}_i^{(0)} = w_a,$$

because $a \notin T_k$, so the entry that changed does not appear. In particular, the set J will not get smaller in the next iteration. For $a \notin J$, we get

$$\sum_{i: T_i \ni a} \tilde{y}_i^{(1)} \leq w_a$$

for Θ small enough, because the inequality was strict before. If we choose Θ as large as possible without violating any dual constraints, we will end up with a set $J' := J \cup F$ in the next iteration, where F is the new set of elements whose dual constraints are satisfied with equality at $\tilde{y}^{(1)}$.

After at most $|H|$ iterations, $\tilde{x}$ must be feasible, because eventually, all elements will appear in J. The first J which leads to a feasible solution will be output as the set A. Let $\tilde{y}^{(\ell)}$ be the dual solution at that time. We know that

$$w(A) := \sum_{a \in A} w_a = \sum_{a \in A} \sum_{i: T_i \ni a} \tilde{y}^{(\ell)} = \sum_{i=1}^{m} |A \cap T_i| \tilde{y}_i^{(\ell)} \leq M \sum_{i=1}^{m} \tilde{y}_i^{(\ell)}, \tag{6.1}$$

where $M := \max_i |T_i|$ is the largest set that appears. On the other hand,

$$\sum_{i=1}^{m} \tilde{y}_i^{(\ell)} \leq \text{opt}(LP_{HS}^{\Delta}) = \text{opt}(LP_{HS}) \leq \text{opt}(ILP_{HS}). \tag{6.2}$$

(6.1) and (6.2) together imply

$$w(A) \leq M \text{opt}(ILP_{HS}).$$

This means, the algorithm computes a hitting set A whose weight is at most M times the weight of an optimal hitting-set. This is the kind of quality guarantee that approximation algorithms usually deliver: the solution is worse than the optimal solution only by a constant factor. In case of the vertex cover problem, we get $M = 2$, because all sets to hit are edges of a graph. This implies the following theorem which concludes this chapter.

Theorem 6.14 *For any graph G, the primal-dual method as described above computes a vertex cover whose size is at most twice the size of a smallest vertex cover.*

Chapter 7

Integer Polyhedra

In the last chapter, we have shown that the ILP problem is **NP**-hard, i.e. there is probably no polynomial-time algorithm that is able to solve any ILP instance. Still, there are easy instances, and we have already come across them. Namely, if the LP relaxation has an optimal integral solution $\tilde{x}$, this solution $\tilde{x}$ is at the same time an optimal ILP solution, and in this case, the solution is obtained in polynomial time. An example was the ILP for the assigment problem, where we have seen that for any objective function, there is an optimal integral solution.

In this chapter, we want to investigate this issue in some more depth and develop conditions under which a system of linear inequalities has integral optimal solutions for every objective function. One such condition has already been given (and proved) in the exercises.

Fact 7.1 *Consider the system* $\{Ax \leq b, x \geq 0\}$*, where* A *and* b *are integral, and* A *is* totally unimodular *(TUM). Then every basic feasible solution* $\tilde{x}$ *of*

$$\begin{array}{lll} \textit{(LP)} & \textit{maximize} & c^T x \\ & \textit{subject to} & Ax \leq b, \\ & & x \geq 0, \end{array}$$

has integral coordinates.

It follows that the simplex algorithm will return an integral solution $\tilde{x}$ wheh applied to (LP).

However, the TUM property is only sufficient for integrality, but not necessary. In the sequel we will discuss another sufficient condition (*total dual integrality* (TDI)), under which we get integral solutions. To motivate this notion, let us consider the problem of maximum weighted matchings in general graphs, which we will not be able to handle using the TUM property.

7.1 Maximum weighted matching

Let $G = (V, E)$ be a graph with nonnegative edge weights $w_e, e \in E$. The goal is to find a matching M in G (set of edges with no common vertices) such that the sum of edge weights in M is as large as possible. This is one of the hardest easy problems, meaning that there exists a polynomial-time algorithm (which is not obvious), and that already slight generalizations of the problem become **NP**-hard

Introducing 0/1-variables x_e for the edges, the matching problem can be formulated as an ILP as follows.

$$\begin{array}{lll} \text{(ILP)} \quad \text{maximize} & \sum_{e \in E} w_e x_e & \\ \text{subject to} & \sum_{e \ni v} x_e \leq 1, & v \in V \\ & x_e \in \{0,1\}, & e \in E. \end{array}$$

In this formulation (and also in the following) we will regard an edge as a set of two vertices. We already know this system from the bipartite case. Here, however, the LP relaxation

$$\begin{array}{lll} \text{(ILP)} \quad \text{maximize} & \sum_{e \in E} w_e x_e & \\ \text{subject to} & \sum_{e \ni v} x_e \leq 1, & v \in V \\ & x_e \geq 0, & e \in E \end{array}$$

may have fractional optimal solutions, and $\text{opt}(LP) \neq \text{opt}(ILP)$ in general. This is due to the fact that the graph is not necessarily bipartite and therefore contains cycles of odd length. For example, if the graph is a triangle (all three edge weights being 1), the maximum matching has weight 1, because only one of the three edges can be in any matching. The LP relaxation, however, has optimum value 3/2, because we may set all variable values to 1/2.

To fix this problem, we will add a family of inequalities to the system which are redundant in the ILP formulation (meaning that they are automatically satisfied by all matchings), but make a difference in the relaxation.

Observation 7.2 *Let $A \subseteq V$ be a set of odd size and consider the set $E_A \subseteq E$ of edges with both vertices in A (equivalently, the set of edges e such that $e \subseteq A$). Then every matching contains at most $(|A| - 1)/2$ edges of E_A.*

This is clear: if the matching would contain more than $(|A| - 1)/2$ edges of E_A, it would match more than $|A| - 1$ vertices of A; this means, restricted to the graph induced by A, it would be a perfect matching covering all $|A|$ vertices. This is impossible if A has odd size. This means, we can add the following "odd-set inequalities" to the ILP above without affecting its solution, obtaining

$$\begin{array}{lll} \text{(ILP')} \quad \text{maximize} & \sum_{e \in E} w_e x_e & \\ \text{subject to} & \sum_{e \ni v} x_e \leq 1, & v \in V \\ & \sum_{e \subseteq A} x_e \leq \frac{|A|-1}{2}, & A \subseteq V, |A| \text{ odd} \\ & x_e \in \{0,1\}, & e \in E. \end{array}$$

With the LP relaxation

$$\begin{array}{lll} \text{(LP')} \quad \text{maximize} & \sum_{e\in E} w_e x_e & \\ \text{subject to} & \sum_{e\ni v} x_e \le 1, & v \in V \\ & \sum_{e\subseteq A} x_e \le \frac{|A|-1}{2}, & A \subseteq V, |A| \text{ odd} \\ & x_e \ge 0, & e \in E, \end{array}$$

one can prove the following Theorem, due to Edmonds.

Theorem 7.3 *$opt(ILP') = opt(LP')$. In particular, (LP') has an integral optimal solution for all weight values $w_e, e \in E$.*

Before we do this, using the notion of TDI-systems, let us remark that this does *not* lead to a polynomial-time matching algorithm right away. It is true that the matching problem can be written as an LP, but the price to pay is an *exponential* number of constraints, one for every odd set. In this situation, even a polynomial-time method for LP will only give us an exponential-time algorithm for the matching problem. At the end of this chapter, we will come back to this issue.

7.2 Total dual integrality

This concept has been introduced to prove integrality of LP solutions, and it is motivated by the matching application.

Definition 7.4 *The inequality system $\{Ax \le b\}$ is called* TDI *(total dual integral), if the linear program*

$$\begin{array}{lll} (LP^\Delta) & \textit{minimize} & b^T y \\ & \textit{subject to} & A^T y = c, \\ & & y \ge 0 \end{array}$$

has an integral optimal solution $\tilde{y}$ for all integral vectors c for which (LP^Δ) is bounded and feasible.

Before we dive into the discussion of the significance of this definition, let us add a word of warning: the TDI property is not very robust. This means, if we for example scale every coordinate in the original system $\{Ax \le b\}$ by the same value $\lambda \ge 0$, we obtain an equivalent system but it might no longer be TDI. For example, the system $\{x \le 1\}$ is TDI because

$$\begin{array}{ll} \text{minimize} & y \\ \text{subject to} & y = c, \\ & y \ge 0 \end{array}$$

has optimal solution $\tilde{y} = c$ for all c such that the problem is feasible. On the other hand, the equivalent system $\{2x \le 2\}$ is not TDI because

$$\begin{array}{ll} \text{minimize} & 2y \\ \text{subject to} & 2y = c, \\ & y \ge 0 \end{array}$$

has no integral optimal solution $\tilde{y}$ if c is odd. Another pitfall is the following: assume you want to know whether a system in standard form $\{Ax \leq b, x \geq 0\}$ is TDI. It might be tempting to apply standard form duality and check whether the LP

$$\begin{array}{ll} \text{minimize} & b^T y \\ \text{subject to} & A^T y \geq c, \\ & y \geq 0 \end{array}$$

has integral solutions $\tilde{y}$ for all integral c. However, if you apply the original definition of TDI, you see that you must also require that the values of the slack variables are integral, i.e. that $\tilde{y}$ is integral *and* $A^T\tilde{y} - c$ is integral. If A happens to be integral, this extra condition is not needed, but in general, it is. Consider the system $\{x_1 + \frac{1}{2}x_2 \leq 1, x_1, x_2 \geq 0\}$. Although the linear program

$$\begin{array}{ll} \text{minimize} & y \\ \text{subject to} & y \geq c_1, \\ & \frac{1}{2}y \geq c_2, \\ & y \geq 0 \end{array}$$

always has an integral solution $\tilde{y} = \max(c_1, 2c_2, 0)$ for integral c_1, c_2, the system $\{x_1 + \frac{1}{2}x_2 \leq 1, x_1, x_2 \geq 0\}$ is not TDI. For example, if $c_1 = 3, c_2 = 1$, the optimal solution is $\tilde{y} = 3$, but $\frac{1}{2}\tilde{y} - 1 = \frac{1}{2} \notin \mathbb{Z}$.

The conclusion is that if you want to prove a system to be TDI, it is quite important in which way it is represented. In the exercises, you are asked to prove that even the removal of redundant inequalities might destroy the TDI property (or, to formulate it positively, the addition of redundant inequalities might give you a TDI system).

Here is the main result of this section.

Theorem 7.5 *If $\{Ax \leq b\}$ is TDI, where A is rational and b is integral, then the linear program*

$$\begin{array}{lll} \textit{(LP)} & \textit{maximize} & c^T x \\ & \textit{subject to} & Ax \leq b \end{array}$$

has an integral optimal solution $\tilde{x}$ for all (not necessarily integral or rational) objective function vectors c which define a bounded and feasible problem.

To simplify the presentation, we only state the following lemma (which we will need below), and postpone its proof to the end of the section.

Lemma 7.6 *Let A, b be rational. The system $\{Ax = b\}$ has an integral solution $\tilde{x}$ if and only if for all rational y, the following implication holds:*

$$y^T A \text{ integral} \quad \Rightarrow \quad y^T b \text{ integral}.$$

This lemma can be seen as an integer version of *Farkas' Lemma* (see exercices).

Proof. (of Theorem 7.5.) Assume there exists some vector c such that (LP) has no integral optimal solutions. From this we will deduce that the system is not TDI.

Let $\tilde{x}$ be a *basic* optimal solution, meaning an optimal solution which satisfies a maximal number of inequalities of $\{Ax \leq b\}$ with equality. Let $\{A'x \leq b'\}$ be this maximal subsystem such that $A'\tilde{x} = b'$. We have the following two properties.

(a) all solutions to $A'x = b'$ satisfy $c^T x = c^T \tilde{x}$. Namely, all solutions to $A'x = b$ in some small neighbourhood of $\tilde{x}$ are feasible for the (LP), which means that $c^T x \leq c^T \tilde{x}$ for all solutions in that neighborhood, because $\tilde{x}$ was optimal. This implies that $c^T x$ must be constant over the system $\{A'x = b'\}$. (Why? Work it out, using the fact that $c^T x$ is a linear function.)

(b) all solutions to $A'x = b'$ are feasible for $Ax \leq b$. Assume there is some $\tilde{x}'$ such that $A'\tilde{x}' = b$ and $\tilde{x}'$ is infeasible. Going along a straight line from $\tilde{x}$ to $\tilde{x}'$, we encounter another optimal solution (by (a)) which satisfies one more inequality of $\{Ax \leq b\}$ with equality, which is a contradiction to the maximality of the susbsystem $\{A'x \leq b'\}$.

In case of systems $\{Ax \leq b, x \geq 0\}$, a basic optimal solution is the unique solution to its subsystem $\{A'x = b', x_i = 0\,(i \in I)\}$ in which case (a) and (b) are obvious.

Using these two properties, we can conclude the proof as follows. As we have assumed that (LP) has no integral optimal solution, the whole system $\{A'x = b'\}$ has no integral solution (because such a solution would be optimal by (a) and (b)). Using the lemma above, we conclude that in this case, there must exist some vector $\tilde{y}$ such that

$$d^T := \tilde{y}^T A' \text{ integral}, \quad \tilde{y}^T b' \text{ not integral}.$$

Without loss of generality, we may even assume that $\tilde{y} \geq 0$; for this, let κ be the least common multiple of all denominators of the entries in A (recall that A is a rational matrix). By adding mutliples of κ to the entries of $\tilde{y}$, we don't change the fact that $\tilde{y}^T A$ is integral (κA is an integer matrix), but we also don't change the property that $\tilde{y}^T b$ is *not* integral (because we add integers to the original non-integer value of $\tilde{y}^T b$). Choosing suitable multiples, we can then achieve $\tilde{y} \geq 0$.

Now we claim that $\tilde{x}$ is an optimal solution to the linear program

$$\begin{array}{lll} \text{(LP')} & \text{maximize} & d^T x \\ & \text{subject to} & Ax \leq b. \end{array}$$

Feasibility is obvious, and to argue that $\tilde{x}$ is optimal, we first compute

$$d^T \tilde{x} = \tilde{y}^T A' \tilde{x} = \tilde{y}^T b'.$$

Moreover, all feasible solutions satisfy $A'x \leq b'$ and therefore

$$d^T x = \tilde{y}^T A' x \leq \tilde{y}^T b',$$

where we have used $\tilde{y} \geq 0$. Since $\tilde{y}^T b' \notin Z$ is the optimal value of (LP'), it is also the optimal (nonintegral) value of its dual,

$$\begin{array}{lll} (\text{LP}'^{\Delta}) & \text{minimize} & b^T y \\ & \text{subject to} & A^T y = d, \\ & & y \geq 0. \end{array}$$

This means, we have found an integral vector d such that (LP'$^{\Delta}$) has no integral optimal solution $\tilde{y}$ (because otherwise the optimal objective function value would have to be integral as well, recalling that b was assumed to be integral). In other words, the system $\{Ax \leq b\}$ is not TDI. □

We still need to show Lemma 7.6.

Proof. If $\{Ax = b\}$ has an integral solution $\tilde{x}$, then $y^T A$ integral implies that $y^T b = y^T A\tilde{x}$ is integral, so one direction is easy. Now assume that $y^T A$ integral implies $y^T b$ integral for all y. Because this property also holds for all subsystems $\{A'x = b'\}$ (set the variables in y corresponding to the removed equalities to zero), we may assume that $\{Ax = b\}$ contains no redundant equalities. This means, whenever $y^T A = 0$ for some $y \neq 0$, then $y^T b \neq 0$. First we show that the rows of A must be linearly independent. Assume, this does not hold; then there is some $\tilde{y} \neq 0$ such that $\tilde{y}^T A = 0$. This implies

$$\tilde{y'}^T A = 0, \tilde{y'}^T b = \frac{1}{2}, \quad \text{for } \tilde{y}' := \tfrac{\tilde{y}}{2\tilde{y}^T b},$$

a contradiction to the requirement that $\tilde{y'}^T b$ must be integral.

Given that A has full row rank, we can use elementary column operations to get A into the form

$$(B \;\; 0),$$

where B is an invertible lower-triangular matrix. This can even be done using so-called *unimodular* transformations, which are

- swapping of two columns
- multiplication of columns with -1,
- subtraction of one column from another.

This is an exercise. It is easy to see that a unimodular transformation applied to A corresponds to a multiplication with a square matrix U (from the right), where U arises from the unit matrix by

- swapping of two columns, or
- replacement of a 1 by a -1, or
- insertion of a single -1 somewhere.

In all three cases, U is integral and $\det(U) \in \{1, -1\}$, so U is a *unimodular* matrix. It is clear that the product of unimodular matrices is again unimodular, and the same is true for the inverse of a unimodular matrix. Now, let W be the unimodular matrix such that

$$(B \;\; 0) = AW$$

and let y^T be any row of B^{-1}. Because of

$$B^{-1}AW = B^{-1}(B \;\; 0) = (E \;\; 0),$$

E the unit matrix, the vector y^TAW is a unit vector. Moreover, W^{-1} is unimodular (in particular, W^{-1} is integral), so the vector $y^TAWW^{-1} = y^TA$ is integral, too. By assumption, y^Tb must then also be integral. Doing this for all rows of B^{-1}, we get that the vector $B^{-1}b$ is integral. Define

$$\tilde{x} := W \begin{pmatrix} B^{-1}b \\ 0 \end{pmatrix}.$$

$\tilde{x}$ is an integral vector, because it is a product of an integral matrix and an integral vector. The claim is that $\tilde{x}$ is a solution to $Ax = b$ which completes the proof. To show this, we compute

$$A\tilde{x} = AW \begin{pmatrix} B^{-1}b \\ 0 \end{pmatrix} = (B \;\; 0) \begin{pmatrix} B^{-1}b \\ 0 \end{pmatrix} = BB^{-1}b = b.$$

□

7.3 The matching system is TDI

Let's come back to matchings. If we can prove that the inequality system

$$\begin{aligned} \sum_{e \ni v} x_e &\leq 1, \quad v \in V \\ \sum_{e \subseteq A} x_e &\leq \frac{|A| - 1}{2}, \quad A \subseteq V, |A| \text{ odd} \\ x_e &\geq 0, \quad e \in E \end{aligned}$$

is TDI, we have proved Edmond's Theorem 7.3, using Theorem 7.5 from the previous section. The proof I will give below is not the one I presented in the lecture; the proof from the lecture (actually the one from the book *Combinatorial Optimization* by Bernhard Korte and Jens Vygen, Theorem 11.15) suffers from the problem that it already uses the fact that the matching polytope is integral. Because we want to derive this integrality from the TDI property, this proof is pointless in our context. The proof below is an alternative version which (hopefully) makes no further assumptions.

To start, let us develop the following equivalent formulation of the TDI property.

Lemma 7.7 *The system $\{Ax \leq b\}$ (A rational, b integral) is TDI if and only if*

$$\max\{c^T x \mid Ax \leq b, x \in \mathbb{Z}^n\} = \min\{b^T y \mid A^T y = c, y \geq 0, y \in \mathbb{Z}^m\} \tag{7.1}$$

holds for all integral c for which the minimum exists.

Proof. Assume $\{Ax \leq b\}$ is TDI, c integral Then we know by definition that

$$\min\{b^T y \mid A^T y = c, y \geq 0, y \in \mathbb{Z}^m\} = \min\{b^T y \mid A^T y = c, y \geq 0\},$$

and by Theorem 7.5 that

$$\max\{c^T x \mid Ax \leq b, x \in \mathbb{Z}^n\} = \max\{c^T x \mid Ax \leq b\}.$$

Then (7.1) follows by LP duality.

For the other diretion, assume (7.1) holds. Using weak duality, we get

$$\begin{aligned} \max\{c^T x \mid Ax \leq b, x \in \mathbb{Z}^n\} &\leq \max\{c^T x \mid Ax \leq b\} \\ &\leq \min\{b^T y \mid A^T y = c, y \geq 0\} \\ &\leq \min\{b^T y \mid A^T y = c, y \geq 0, y \in \mathbb{Z}^m\}. \end{aligned}$$

By (7.1), equality must hold, which proves the TDI property. □

The plan now is to show that (7.1) holds for the matching system. For this, let us consider the dual problem; it has a variable y_v for each vertex v (corresponding to the vertex constraints of the matching system), and a variable z_A for every odd set (corresponding to the odd set inequalities). If $\mathcal{A}$ denotes the set of subsets of V of odd size, the dual problem can be written as

$$\begin{array}{lll} (\mathrm{LP}^\Delta) & \text{minimize} & \sum_{v \in V} y_v + \sum_{A \subseteq \mathcal{A}} \frac{|A|-1}{2} z_A \\ & \text{subject to} & \sum_{v \in e} y_v + \sum_{A \supseteq e} z_A \geq c_e, \quad e \in E, \\ & & y_v, z_A \geq 0, \quad v \in V, A \in \mathcal{A}. \end{array}$$

We have written the dual in standard (inequality) form, although the definition of TDI requires a dual problem in equality form (which we would get after adding slack variables); however, in this case it holds that if we find an integral solution $\tilde{y}, \tilde{z}$ to the standard form dual, we have verified the TDI property. Namely, the constraint matrix and right-hand side are both integral, in which case the values of the slack variables at an optimal integral solution will be integral, too. This has already been observed in the discussion following Definition 7.4.

This means, in order to verify (7.1) we need to prove that for every integral c, the optimum value of

$$\begin{array}{lll} (\mathrm{ILP}) & \text{maximize} & \sum_{e \in E} c_e x_e \\ & \text{subject to} & \sum_{e \ni v} x_e \leq 1, \quad v \in V \\ & & \sum_{e \subseteq A} x_e \leq \frac{|A|-1}{2}, \quad A \subseteq V, |A| \text{ odd} \\ & & x_e \in \{0,1\} \quad e \in E \end{array} \tag{7.2}$$

equals the optimum value of

$$\begin{array}{lll} (\mathrm{ILP}^{\Delta}) & \text{minimize} & \sum_{v\in V} y_v + \sum_{A\subseteq\mathcal{A}} \frac{|A|-1}{2} z_A \\ & \text{subject to} & \sum_{v\in e} y_v + \sum_{A\supseteq e} z_A \geq c_e, \quad e \in E, \\ & & y_v, z_A \geq 0, y_v, z_A \in \mathbb{Z}, \quad v \in V, A \in \mathcal{A}. \end{array} \tag{7.3}$$

In other words, $\mathrm{opt}(\mathrm{ILP}^{\Delta})$ must be equal to the weight of an optimal matching. Some notation: treating c as a paremeter, $\mathrm{ILP}(c)$ and $\mathrm{ILP}^{\Delta}(c)$ denote the problems (7.2) and (7.3). $w(c)$ denotes the weight of an optimal matching with respect to the weight function c, i.e. $w(c) = \mathrm{opt}(ILP(c))$. For any given matching M, $w_c(M)$ is the weight of M with respecto to c. Finally, $\Delta(c) := \mathrm{opt}(ILP^{\Delta}(c))$. By weak duality, $w(c) \leq \Delta(c)$ always holds, and we need to prove $w(c) = \Delta(c)$.

To streamline the presentation, we will make some claims in the following which we will prove only later.

Now assume (7.1) does not always hold. From this, we are going to derive a contradiction. Let $(G = (V, E), c)$ be a graph-weights pair such that $w(c) < \Delta(c)$, and $|V| + |E| + \sum_{e\in E} |c_e|$ is as small as possible. We refer to (G, c) as a *minimal counterexample*. The minimality implies $c_e \geq 1$ for all $e \in E$, because if $c_e \leq 0$, the corresponding dual constraint is redundant; this means, e can be removed from G without changing $w(c)$ and $\Delta(c)$. Because (G, c) was a minimal counterexample, this is impossible. Moreover, we can assume that G is connected, otherwise some connected component of G defines a smaller counterexample. For this, one has to observe that $w(c)$ and $\Delta(c)$ are the sum of the corresponding optimal values of the subgraph ILPs and $\mathrm{ILP}^{\Delta}(c)$s, respectively. Now there are two cases.

(a) G contains a vertex v which is matched in every optimal matching. Let c' arise from c by decreasing the weights of all edges $e \ni v$ by one.

Claim 7.8 $w(c') = w(c) - 1$.

Because (G, c) was a minimal counterexample, (7.1) must hold for (G, c'), so $w(c') = \Delta(c')$ holds, where $\Delta(c')$ is assumed by an integral solution $\tilde{y}, \tilde{z}$ to $\mathrm{ILP}^{\Delta}(c')$. Increasing $\tilde{y}_v$ by one gives an integral feasible solution to $\mathrm{ILP}^{\Delta}(c)$, of value $w(c') + 1 = w(c)$. But this means, $w(c) \geq \Delta(c)$, a contradiction to the assumption $w(c) < \Delta(c)$. Hence, this case cannot occur.

(b) No vertex of G is matched in every optimal matching. Then let c' arise from c by decreasing *all* weights by one.

Claim 7.9 $w(c') \leq w(c) - \left\lfloor \frac{|V|}{2} \right\rfloor$.

Again, (G, c') cannot be a counterexample, so $w(c') = \Delta(c')$, with an optimal solution $\tilde{y}, \tilde{z}$ to $\mathrm{ILP}^{\Delta}(c')$. If $|V|$ is odd, we can increase the value of $\tilde{z}_V$ by one, giving us a feasible solution to $\mathrm{ILP}^{\Delta}(c)$, of value

$$w(c') + \frac{|V| - 1}{2} = w(c') + \left\lfloor \frac{|V|}{2} \right\rfloor \leq w(c),$$

showing that $w(c) \geq \Delta(c)$, a contradiction. If $|V|$ is even, consider some subset $A := V \setminus \{v\}$ of odd size. Increasing $\tilde{z}_A$ and $\tilde{y}_v$ by one gives a feasible solution to $\mathrm{ILP}^{\Delta}(c)$, of value

$$w(c') + 1 + \frac{|V| - 2}{2} = w(c') + \frac{|V|}{2} = \left\lfloor \frac{|V|}{2} \right\rfloor \leq w(c),$$

leading to a contradiction again. This means, case (b) cannot occur either, so the original assumption that a counterexample (G, c) exists leads to a contradiction. Therefore, (7.1) must hold for all pairs (G, c).

□

It remains to prove the two claims.

Proof. (Claim 7.8) Every optimal matching with respect to c has weight $w(c) - 1$ in $\mathrm{ILP}(c')$, because it contains an edge incident to v. This shows $w(c') \geq w(c) - 1$. On the other hand, if $w(c') > w(c) - 1$ would hold, this would already imply $w(c') \geq w(c)$ (because all the weights are integral).

Let M be an optimal matching with respect to c'. We have

$$w_c(M) \geq w_{c'}(M) = w(c') \geq w(c) \geq w_c(M),$$

and $w_c(M) = w_{c'}(M) = w(c)$ follows. This means, M is optimal for $\mathrm{ILP}(c)$; this is a contradiction to the fact observed above, namely that $w_{c'}(M) = w(c) - 1$ must hold for every optimal matching with respect to c. □

Proof. (Claim 7.9) Assume that $w(c') > w(c) - \lfloor |V|/2 \rfloor$ and let M be an optimal matching with respect to c', chosen in such a way that $w_c(M)$ is as large as possible. M must have less than $\lfloor |V|/2 \rfloor$ edges, otherwise

$$w(c) \geq w_c(M) \geq w_{c'}(M) + \left\lfloor \frac{|V|}{2} \right\rfloor,$$

where we have just assumed the contrary. It follows that M matches less than $|V|$ vertices if $|V|$ is even, and less than $|V| - 1$ if $|V|$ is odd. In both cases, there are two unmatched vertices, say u and v. Assume M, u, v have been chosen such that the length $\delta(u, v)$ of a shortest path between u and v is as small as possible (such a shortest path exists in all cases, because we have shown above that G can be assumed to be connected). Because $\{u, v\}$ cannot be an edge (otherwise this edge could be added to M, improving $w_c(M)$), we have $\delta(u, v) > 1$, and there is some vertex $t \neq u, v$ on the shortest path.

Because we are in case (b) of the main proof, there is an optimal matching M' with respect to c such that M' does not match t. Now let us consider the symmetric difference $M \triangle M'$ (the set of edges which are in exactly one of M and M'). Because every vertex is incident to at most one edge from each M and M', $M \triangle M'$ defines a graph where every vertex has degree at most two. Such a graph consists of a collection of paths and cycles (isolated vertices, i.e. paths of length zero, may occur). Let P be the component that contains t. P must be a path (starting at v), because t has at most one incident edge (from M) in $M \triangle M'$. Moreover, u and v have also degree at most one in $M \triangle M'$, with a possible incident edge coming from M'. Then P cannot contain both u and v, because it can have only one more degree-one vertex, at its other end. Say P does not contain u.

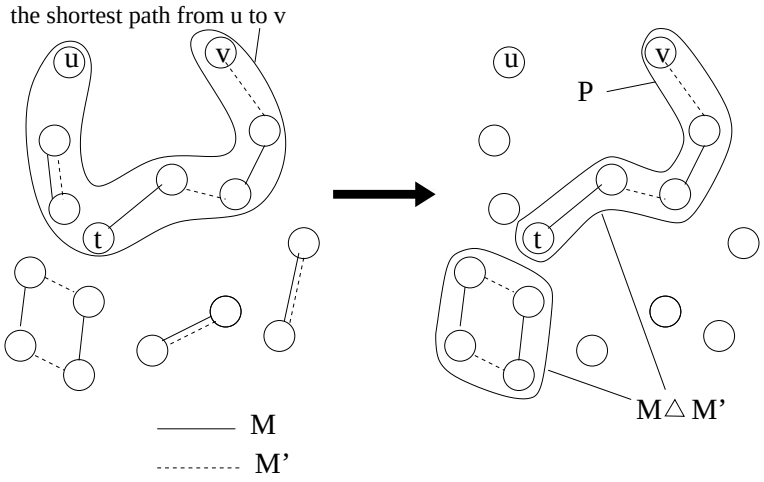


Figure 7.1: Sketch of the proof

Now consider the symmetric difference $M \triangle P$. We have the following two properties.

(i) $M \triangle P$ is a matching. Namely, if some vertex w is incident to two edges in $M \triangle P$, one from M and one from P, then the P-edge must come from M'. But then the M-edge must also be in P (P is a connected component of $M \triangle M'$), so the M-edge is not in $M \triangle P$, a contradiction.

(ii) $|M \triangle P| \leq |M|$. This holds because $|M \triangle P| > |M|$ can only hold if P is an augmenting path. If P is empty, this surely is not the case; if P is nonempty, it starts with t and an incident edge from M. This means, t is not a free vertex with respect to M, so P cannot define an augmenting path. In particular, $M \triangle P$ does not cover t.

We are close to the end now. Because c' has been obtained from c by decreasing all weights by one, we have

$$\begin{aligned} w_{c'}(M \triangle P) - w_{c'}(M) &= (w_c(M \triangle P) - |M \triangle P|) - (w_c(M) - |M|) \\ &= (w_c(M \triangle P) - w_c(M)) + (|M| - |M \triangle P|) \\ &\geq w_c(M \triangle P) - w_c(M) \quad \text{,by property (ii).} \end{aligned}$$

To continue, observe that

$$w_c(M) + w_c(M') = w_c(M \triangle P) + w_c(M' \triangle P). \tag{7.4}$$

This holds, because on the left-hand side of the equation, the weight of every edge in $M \triangle M'$ is counted exactly once, while all edges in $M \cap M'$ contribute twice their weight. The same is true on the right-hand side: an edge $e \in M \setminus M'$ is in $M \triangle P$ if $e \notin P$ and in $M' \triangle P$ if $e \in P$, but not in both. The same holds for edges in $M' \setminus M$. Edges in $M \cap M'$ are not in P and therefore in both $M \triangle P$ and $M' \triangle P$.

Plugging (7.4) into the previous derivation, we can conclude that

$$w_{c'}(M \triangle P) - w_{c'}(M) \geq w_c(M \triangle P) - w_c(M) = w_c(M') - w_c(M' \triangle P) \geq 0,$$

because M' was an optimal matching with respect to c. It follows that $M \triangle P$ must be optimal with respect to c' (because it is no worse than M). However, $M \triangle P$ does not cover u (because neither M nor P cover u), and it also does not cover t (because either *both* M and P or none of them cover t; see the argument for property (ii) above). This means, we have found an optimal matching with respect to c' which does not cover u and t and has $w_c(M \triangle P) \geq w_c(P)$. Because the shortest-path distance between u and t is strictly smaller than between u and v, this is a contradiction to our original choice of M, u, v. □

7.4 The integer hull of a polyhedron

The notions of TUM and TDI establish *sufficient* conditions for a system $\{Ax \leq b\}$ to have integral optimal solutions for all linear objective functions. However, they are not necessary. In this section, we want to characterize the class of systems $\{Ax \leq b\}$ for which integral solutions always exist.

Definition 7.10

(i) Let $\{Ax \leq b\}$ be a system of linear inequalities in n variables. The set $\{x \in \mathbb{R}^n \mid Ax \leq b\}$ is called a polyhedron.

(ii) Let $V \subseteq \mathbb{R}^n$ be a finite set. The convex hull *of V, defined as*

$$conv(V) := \{\sum_{v \in V} \lambda_v v \mid \lambda_v \geq 0 \ \forall v, \sum_{v \in V} \lambda_v = 1\},$$

is called a polytope.

Here is the main theorem of polytope theory.

Theorem 7.11 *Every bounded polyhedron is a polytope and vice versa.*

This means, every bounded polyhedron has two descriptions, one in terms of *halfspaces* (linear inequalities), and one in terms of vertices. This is not unfamiliar to us. For example, a cube is the convex hull of its eight vertices, but at the same time the intersection of the halfspaces bounded by its six facets.

Definition 7.12 *Let P be a polyhedron in $\mathbb{R}^n$. The set*

$$P_I := conv(P \cap \mathbb{Z}^n)$$

is called the integer hull *of P.*

In this definition, we might have to take the convex hull of an infite set of points. Like for linear combinations, this is defined exactly as above, where every sum ranging over infinitely many terms is restricted to a finite number of nonzero terms. If P is bounded, it contains only finitely many points from $\mathbb{Z}^n$, and then P_I is a polytope. If P is *rational*, then P_I will always be at least a polyhedron.

Theorem 7.13 *If A, b consists of rational entries and P is the polyhedron defined by $\{Ax \leq b\}$, then P_I is a polyhedron, too.*

This becomes false if irrational entries are allowed (exercise).

Here comes the main definition of this section.

Definition 7.14 *A polyhedron such that $P = P_I$ is called an* integral *polyhedron.*

As it turns out, integral polyhedra are exactly the polyhedra over which any linear function assumes its maximum at an integral solution (provided an optimal solution exists at all). We will not prove this here (although it would not be too difficult with the machinery we have developed). Therefore, we just conclude with the formal statement.

Theorem 7.15 *$P = \{x \mid Ax \leq b\} \neq \emptyset$ is an integral polyhedron if and only if the linear program*

$$\begin{array}{lll} \textit{(LP)} & \textit{maximize} & c^T x \\ & \textit{subject to} & Ax \leq b \end{array}$$

has an integral optimal solution $\tilde{x}$ for all c for which the maximum exists.

For example, given a graph $G = (V, E)$, the bounded set of solutions satisfying

$$\begin{aligned} \sum_{e \ni v} x_e &\leq 1, \quad v \in V \\ \sum_{e \subseteq A} x_e &\leq \frac{|A| - 1}{2}, \quad A \subseteq V, |A| \text{ odd} \\ x_e &\geq 0, \quad e \in E, \end{aligned}$$

is an integral polytope by this Theorem, the so-called *matching polytope*.

Chapter 8

Cutting Planes

We have seen in the last chapter that the *matching system*

$$\begin{array}{lll} \text{(LP')} \quad \text{maximize} & \sum_{e\in E} w_e x_e & \\ \text{subject to} & \sum_{e\ni v} x_e \le 1, & v \in V \\ & \sum_{e\subseteq A} x_e \le \frac{|A|-1}{2}, & A \subseteq V, |A| \text{ odd} \\ & x_e \ge 0, & e \in E, \end{array}$$

has the TDI property. This implies that the polyehedron P of feasible solutions to this sytem equals its integer hull P_I. The vertices of the latter polyhedron correspond to the (chracteristic vectors of) matchings. In other words, we have an explicit description of P_I in terms of a set of linear inequalities. Although this set is exponentially large in the size of the graph, the fact that an explicit description can be given at all is quite strong. In fact, an optimal matching can be found efficiently, as first shown by Edmonds.

Theorem 8.1 *The matching of maximum weight in a weighted graph can be found in polynomial time.*

The algorithm follows the primal-dual method; below we will encounter a powerful general method to derive results of this type.

In other cases, an explicit description is not known. An example is the *travelling salesperson problem* (TSP): we are given the complete graph K_n on n vertices, with nonnegative edge weights. The goal is to find a *tour* which minimizes the total weight of the edges that are used. Here, a tour is a cycle which goes through every vertex excatly once.

As an ILP, we can formulate the problem as follows. We introduce variables x_e for every edge, with the meaning that

$$x_e = \begin{cases} 1, & \text{if } e \text{ is in the tour,} \\ 0, & \text{otherwise} \end{cases}$$

Then the problem is

$$\begin{array}{lll} \text{(TSP)} \quad \text{minimize} & \sum_{e\in E} w_e x_e & \\ \text{subject to} & \sum_{e\ni v} x_e = 2, & v \in V \\ & \sum_{e\subseteq A} x_e \le |A| - 1, & A \subseteq V, A \neq \emptyset, V \\ & x_e \in \{0,1\}, & e \in E. \end{array}$$

The constraints $\sum_{e \in v} x_e = 2$ ensure that exactly two edges go through every vertex, which is obviously needed to get a tour. However, this would still allow *subtours*, i.e. a set of several disjoint cycles. To rule out these subtours, we need to add the *subtour inequalities* $\sum_{e \subseteq A} x_e \leq |A| - 1$ for $A \neq \emptyset, V$. For a fixed A, the inequality just says that no subtour can be formed by the vertices in A, because such a subtour would have $|A|$ edges.

Replacing the constraints $x_e \in \{0, 1\}$ by $0 \leq x_e \leq 1$, we get the canonical LP-relaxation. However, although the TSP system looks similar to the matching system, the situation is quite different here: the LP relaxation does have fractional optimal solution in general, so the *subtour polytope*

$$P := \left\{ x \;\middle|\; \begin{array}{ll} \sum_{e \ni v} x_e = 2, & v \in V \\ \sum_{e \subseteq A} x_e \leq |A| - 1, & A \subseteq V, \\ 0 \leq x_e \leq 1, & e \in E \end{array} \right\} \tag{8.1}$$

is not integral, meaning that $P \neq P_I$ (P_I is called the *TSP-polytope*). Even worse, TSP is one of the problems for which no explicit description of P_I is known. To be precise: for any fixed n, one can of course compute the finite inequality description of P_I. The statement is that no class of inequalities (parameterized with n) exists such that this class yields a description for all n. It is even NP-hard to decide whether a given inequality is nonredundant for P_I.

In order to optimize a linear function over P_I (and that's what needs to be done to solve TSP), we can therefore not rely on P_I directly. The method of *cutting planes* described in this chapter provides a way around this.

8.1 Outline of the Method

We are given a general ILP of the form

$$\begin{array}{lll} \text{(ILP)} & \text{maxmize} & c^T x \\ & x \in P_I, & \end{array} \tag{8.2}$$

where

$$P = \{x \mid Ax \leq b\}.$$

Then we perform the following steps.

1. Set $Q := P$;
2. Compute an optimal solution $\tilde{x}$ to

$$\begin{array}{lll} \text{(ILP)} & \text{maxmize} & c^T x \\ & x \in Q. & \end{array}$$

3. If $\tilde{x}$ is integral, we are done;

4. If $\tilde{x}$ has fractional coordinates, then $x \notin P_I$ (why?). In this case, there exists an inequality which is satisfied by all $x \in P_I$ but not by $\tilde{x}$. This means, there are d, f such that

$$\begin{aligned} d^T x &\leq f, \quad x \in P_I, \\ d^T \tilde{x} &> f. \end{aligned}$$

Set

$$Q := Q \cap \{x \mid d^T x \leq f\}$$

and repeat with step 2.

Some comments are in order. Q is described by a finite set of inequalities throughout the algorithm. This is true in the beginning (where Q corresponds to the system $\{Ax \leq b\}$), and each round adds one more inequality. Thus, we can in principle optimize over Q by known methods; keep in mind, however, that Q might have a very large inequality description even in the beginning, so that it is not clear whether step 2 can be performed efficiently. We will come back to this below.

The existence of a *separating hyperplane* $d^T x = f$ in case $\tilde{x} \notin P_I$ follows from convexity theory. In general, if C is some convex set and p a point not in C, then there exists a hyperplane which separates C from p, see Figure 8.1.)

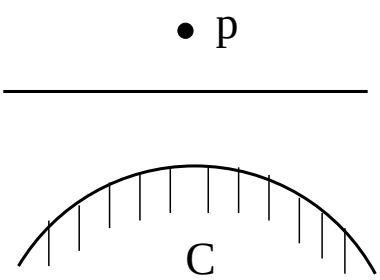


Figure 8.1: A separating hyperplane

Because P_I is a polyhedron, it is in particular convex, so we can apply the general statement.

Thus, what happens in the outline above can be described as "cutting off" fractional optimal solutions by adding another inequality which prevents this solution from reappearing in the next iteration, see Figure 8.2.

Two questions remain:

- How do we find the separating inequality $d^T x \leq f$ (a cut), and
- does this method ever terminate?

We will answer these questions by providing a concrete implementation of the outline above in the next section.

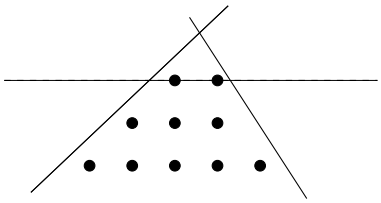


Figure 8.2: Cutting off fractional solution

8.2 Gomory Cuts

Let us address ILP of the general form (8.2), with the additional assumption that the polyhedron $\{Ax \leq b\}$ is bounded, i.e. it is actually a polytope. Moreover, we assume that the system is in standard form, i.e. that it includes the constraints $x \geq 0$, so that we can directly apply the simplex method to it. All this can be done without loss of generality, see the exercises. In addition, we can assume that A, b, c all have integer entries (by scaling them with suitable multiples, if they have rational entries at all).

Then we perform the following procedure, which explains how cuts are found; termination will be adressed later. The cuts used by the method below are called *Gomory cuts*. The method maintains the invariant that Q contains all integral solutions of the system $\{Ax \leq b\}$.

1. $Q := \{x \mid Ax \leq b\}$.

2. find the lexicographically largest basic feasible solution $\tilde{x}$ to

$$\begin{array}{ll} \text{(LP)} & \text{maxmize} \quad c^T x \\ & x \in Q. \end{array}$$

   This can be done by the simplex method, after symbolically perturbing the objective function: instead of $c^T x$, maximize

$$c^T x + \sum_{i=1}^{n} \varepsilon^i x_i,$$

   where $\varepsilon > 0$ is a symbolic constant. This results in the lexicographically largest optimal solution $\tilde{x}$, meaning that among all optimal solutions, $\tilde{x}$ has maximal x_1-coordinate, and among those, $\tilde{x}$ has maximal x_2-coordinate, etc.

3. If the basic part $\tilde{x}_B$ is integral, we are done (because the nonbasic part satisfies $\tilde{x}_N = 0$, which is integral in all cases).

4. If $\tilde{x}_B$ has fractional coordinates, we distinguish between two cases.

(a) $z_0 \notin \mathbb{Z}$, where z_0 is the optimal objective function value $z_0 = c^T\tilde{x}$ of the LP solved in step 2. Then we consider the z-row of the optimal tableau, which reads as

$$z = c^T x = z_0 - \gamma^T x_N, \quad \gamma \geq 0.$$

This implies

$$z \leq z_0 - \lfloor \gamma^T \rfloor x_N,$$

where $\lfloor \gamma^T \rfloor$ arises from γ^T by rounding down each coordinate to the nearest integer. Moreover, if x is any integral solution, then the associated objective function value z is integral. Therefore,

$$\begin{aligned} & z + \lfloor \gamma^T \rfloor x_N \leq z_0 \\ \Rightarrow\ & z + \lfloor \gamma^T \rfloor x_N \leq \lfloor z_0 \rfloor \\ \Leftrightarrow\ & z \leq \lfloor z_0 \rfloor - \lfloor \gamma^T \rfloor x_N \end{aligned}$$

holds for all $x \in Q \cap \mathbb{Z}^n$. This means, the inequality

$$z_0 - \gamma^T x_N \leq \lfloor z_0 \rfloor - \lfloor \gamma^T \rfloor x_N \quad \Leftrightarrow \quad \left(\gamma^T - \lfloor \gamma^T \rfloor\right) x_N \geq z_0 - \lfloor z_0 \rfloor$$

holds for all integral solutions in Q, but *not* (and this is the point!) for the solution $\tilde{x}$ we have computed in step 2. Namely, for that solution, $\tilde{x}_N = 0$, hence

$$\left(\gamma^T - \lfloor \gamma^T \rfloor\right) \tilde{x}_N = 0 < z_0 - \lfloor z_0 \rfloor,$$

because z_0 was not integer. This means, we have found an inequality which separates Q_I from $\tilde{x}$. Adding this inequality to Q ("cutting off" $\tilde{x}$) makes Q smaller, without cutting off any point from Q_I. Thus, we set

$$Q := Q \cap \{x \mid \left(\gamma^T - \lfloor \gamma^T \rfloor\right) x_N \geq z_0 - \lfloor z_0 \rfloor\}$$

and go back to step 2.

(b) $z_0 \in \mathbb{Z}$. Then let j be the smallest index such that $\tilde{x}_j \notin \mathbb{Z}$. Such an index must exist because we have assumed that $\tilde{x}$ is not integral. Moreover, $j \in B$, the set of basic indices, because $\tilde{x}_N = 0$. Then we can consider the tableau equation for x_j which reads as

$$x_j = \beta_j - \lambda^T x_N,$$

and as before we see that all integral solutions satsify

$$x_j \leq \lfloor \beta_j \rfloor - \lfloor \lambda^T \rfloor x_N.$$

In this case, we set

$$Q := Q \cap \{x \mid x_j \leq \lfloor \beta_j \rfloor - \lfloor \lambda^T \rfloor x_N\}$$

and return to step 2.

The termination of this method is summarized by the following

Theorem 8.2 *The above algorithm terminates within a finite number of iterations with an optimal solution $\tilde{x}$ to the system $\{Ax \leq b, x \in \mathbb{Z}^n\}$, or it asserts that no such solution exists.*

Proof. The non-existence of an integral solution is noticed when the LP that is solved in step 2 becomes infeasible, meaning that $Q = \emptyset$. Because we always guarantee that $Q \supseteq \{Ax \leq b, x \in \mathbb{Z}^n\}$, this means that there are no integral solutions.

Now assume that infeasibility does not occur. We will first show that after finitely many iterations, z_0 reaches a fixed integral value. This shows the main idea of the proof, and the fact that also the coordinates of the solution "stabilize" after finitely many steps is easy to derive then.

Consider the sequence $(z_0^{(k)}), k \in \mathbb{N}$, of the z_0-values that are generated in the main loop. The sequence is monotone decreasing (we add a constraint to the LP each time, so the optimum value cannot get larger) and bounded from below, because we have assumed the system $\{Ax \leq b\}$ to be bounded. It follows that the sequence converges to a value w. Because $w < \lfloor w \rfloor + 1$, there is a value ℓ such that

$$z_0^{(\ell)} < \lfloor w \rfloor + 1.$$

If $z_0^{(\ell)} = \lfloor w \rfloor = w$, we are done, because then z_0 must be integral and remains fixed. Otherwise, we have $z_0^{(\ell)} \notin \mathbb{Z}$, and in this iteration we add the inequality

$$z \leq \lfloor z_0^{(\ell)} \rfloor - \lfloor \gamma^T \rfloor x_N, \quad \gamma^T \geq 0.$$

It follows that in the next iteration, this inequality is satisfied, so that we get

$$z = z_0^{(\ell+1)} \leq \lfloor z_0^{(\ell)} \rfloor = \lfloor w \rfloor,$$

because of $\lfloor \gamma^T \rfloor \geq 0, x_N \geq 0$. This means, z_0 is a fixed integer from that iteration on.

Now we prove by induction that also the coordinates $x_1, \ldots, x_n$ stabilize, one after another. Assume that after iteration ℓ_{d-1}, $z, \tilde{x}_1, \ldots, \tilde{x}_{d-1}$ are already fixed integers that do not change anymore (the case $d = 1$ has just been handled with iteration $\ell_0 = \ell$). Then, the sequence $(\tilde{x}_d^{(k)}), k > \ell_d$ is monotone decreasing, because the solutions only get lexicographically smaller throughout the iterations. Because $\tilde{x}_1, \ldots, \tilde{x}_{d-1}$ are already fixed, this means that $\tilde{x}_d$ can only decrease. Because $\tilde{x}_d$ is bounded from below by 0, the sequence $(\tilde{x}_d^{(k)})$ converges to a value $u < \lfloor u \rfloor + 1$. Consequently, there is an index ℓ_d such that

$$\tilde{x}_d^{(\ell_d)} < \lfloor u \rfloor + 1.$$

If $\tilde{x}_d^{(\ell_d)} = \lfloor u \rfloor = u$, we are done, as before, otherwise, $\tilde{x}_d^{(\ell_d)} \notin \mathbb{Z}$, and the algorithm adds the cut

$$x_d \leq \lfloor \beta_1 \rfloor - \lfloor \lambda^T \rfloor x_N,$$

where $x_d = \beta_1 - \lambda^T x_N$ is the tableau row of the variable x_d. Let $\tilde{x}'$ be the next optimal solution, obtained after adding this cut. This solution is a feasible solution to the previous problem and therefore still satisfies

$$\tilde{x}'_d = \beta_1 - \lambda^T \tilde{x}'_N.$$

Because of $\tilde{x}'_d \leq \tilde{x}_d = \beta_1$, we must have $\lambda_i \geq 0$ for all i such that $\tilde{x}'_i > 0$. Because of this,

$$\tilde{x}'_d \leq \lfloor \beta_1 \rfloor - \lfloor \lambda^T \rfloor \tilde{x}'_N \leq \lfloor \beta_1 \rfloor,$$

and $\tilde{x}'_d$ has reached its final integer value $\lfloor \beta_1 \rfloor$ □

8.3 Separation Oracles

The method of Gomory cuts might be useful for general ILP, for which no extra information is given. In this case, the Gomory cuts are "minimal" cuts which are always guaranteed to work. However, the method is quite slow in practice, and it is more of theoretical interest. The approach of cutting planes is more general, and better cuts can be used for special ILP like the TSP (we will get to such cuts below).

In these methods (and even in the Gomory cut method) we have one problem left, which we have already mentioneded above: if the initial inequality description of Q is already very large (for example, exponential in the size of the input instance, like in TSP), step 2 already takes exponential time. However, there is a very general result which provides a way out.

Assume, we have a *separation oracle* with the following specification.

> Given Q and $\tilde{x}$, either certify that $\tilde{x} \in Q$, or find a separating inequality $d^T x \leq f$, meaning that
>
> $$\begin{aligned} d^T x &\leq f, \quad x \in Q, \\ d^T \tilde{x} &> f. \end{aligned}$$

The reason to call this an oracle is that we assume to obtain an answer immediately, but we don't really care how this answer is found. Instead, we bound the runtime of algorithms in terms of the number of oracle calls that are needed. Given a concrete runtime bound for the implementation of an oracle, we then get a concrete runtime bound for an algorithm using the oracle. The following result is due to Grötschel, Lóvasz and Schrijver.

Theorem 8.3 *Let Q be a convex set. An optimal solution to the problem*

$$\begin{aligned} &\textit{maxmize} \quad c^T x \\ &x \in Q. \end{aligned}$$

can be found with a polynomial number of calls to the separation oracle.

I won't specify what this "polynomial number" actually means here; it is not clear what the input really is. Think of this theorem as saying that separation and optimization over convex sets are equally difficult with respect to polynomial-time solvability.

The Gomory cuts described above do not implement a separation oracle, because they only provide cuts for special solutions $\tilde{x}$. In order to be able to optimize, cuts must in principal be provided for all possible $\tilde{x}$, because the optimizing algorithm might just ask for it.

Using this theorem, the following lemma shows that one can optimize over the subtour polytope (8.1) in polynomial time, although it is defined by exponentially many inequalities.

Lemma 8.4 *There exists a polynomial-time separation oracle for the subtour polytope; this means, we can decide in polynomial time, whether a given vector $\tilde{x}$ satisfies all (in)equalities of the system in (8.1), and return a separting inequality if this is not the case.*

Proof. Given $\tilde{x}$, we first check whether $0 \leq \tilde{x}_e \leq 1, e \in E$ and $\sum_{e \ni v} \tilde{x}_e = 2, v \in V$. As these are only polynomially many constraints (polynomial in the size of the graph underlying the TSP problem), this can be done in polynomial time, and every violated (in)equality directly leads to a separating hyperplane.

Now assume $\tilde{x}$ satisfies the above constraints; it is easy to see (exercise) that in this case, the inequalities

$$\sum_{e \subseteq A} \tilde{x}_e \leq |A| - 1, \quad A \neq \emptyset, V$$

are equivalent to

$$\sum_{e : |e \cap A| = 1} \tilde{x}_e \geq 2, \quad A \neq \emptyset, V.$$

Interpreting the $\tilde{x}_e$ as edge weights, the latter class of inequalities is equivalent to the statement that all cuts in the graph have total weight at least 2. To see this, observe that the edges e such that $|e \cap A| = 1$ are exactly the edges that cross the cut defined by A.

Thus, all these inequalities are fulfilled if and only if the minimum cut in the graph (under edge weights $\tilde{x}$) has weight at least 2. Because minimum cuts under nonnegative edge weights can be computed in polynomial time (see also Exercise 5, Problem 1), we either certify in polynomial time that all inequalities hold, or we find a cut A (namely the minimum cut) of weight smaller than 2. The corresponding inequality is a separating inequality. ∎

Chapter 9

The Travelling Salesperson Problem

The TSP problem we have already discussed in the previous chapter to some extent is one of the most important problems in discrete optimization. Many new techniques have been developed with the goal of finding better solutions to large TSP instances. Therefore, this problem deserves its own chapter. I will introduce two important techniques (cutting planes beyond Gomory cuts, Branch & Bound) for the TSP problem, where it will become clear how they are applied in the general situation. I will start with an approximation algorithm for the TSP problem in the special case where the weights satisfy the *triangle inequality*.

9.1 Christofides' Approximation Algorithm

We are given a weighted complete graph on n vertices V, where the weights w_e satisfy the triangle inequality. This means, if p, q, r are vertices of the graph, then the weights satsify

$$w_{\{p,q\}} \leq w_{\{p,r\}} + w_{\{r,q\}}.$$

Interpreting the weights as edge lengths, the triangle inequality stipulates that the direct way from p to q is at least not longer than the detour via a third vertex r. The important case of *Euclidean TSP* is covered by this. Here, the vertices are points in the plane, end the edge weights are actual euclidean lengths.

Here is Christofides' algorithm to find a provably good tour.

1. Compute the *minimum-spanning tree* (MST) T of the graph with respect to the given edge weights. This is a tree whose total weight is as small as possible. It is well-known that an MST can be found in polynomial time (actually, in $O(n^2)$ time), see Figure 9.1.

2. Let V' be the vertices which have odd degree in T. Because the number of odd-degree vertices is even in every graph, $|V'|$ is an even number. In the complete subgraph induced by V', compute a minimum-weight matching M. We have already indicated

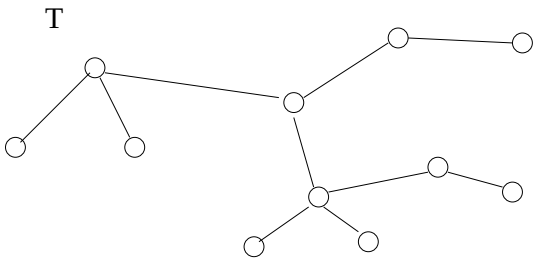


Figure 9.1: Minimum spanning tree T.

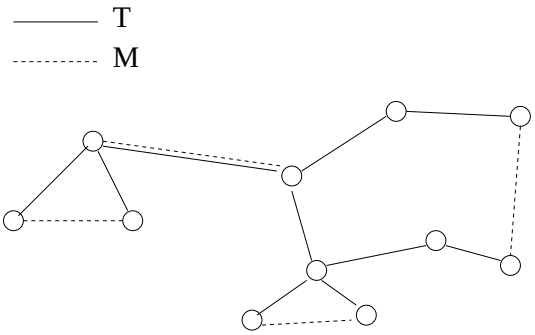


Figure 9.2: T and M.

that this can be done in polynomial time as well, although we have only dealt with the bipartite case in the lecture. Figure 9.2 shows both T and M.

3. Interpreting T and M as edge sets, every vertex has even degree in the graph $(V, T \cup M)$ (which might have multiple edges). It follows that there exist an *Euler tour*, a tour through the edges of the graph which traverses each edge exactly once. Moreover, such an Euler tour can be found in polynomial time (exercise).

4. Traverse the Euler tour and report the vertices in order of apperance along the Euler tour. This order defines the output tour of Christofides' algorithm, see Figure 9.3.

Theorem 9.1 *Let opt be the length of the shortest tour and C the length of the tour computed by Christofides' algorithm. Then*

$$C \leq \frac{3}{2} opt.$$

Proof. It holds that $C \leq w(T) + w(M)$, where $w(T), w(M)$ denote the total weight of T and M, respectively. Namely, $w(T) + w(M)$ is the length of the Euler tour, and the output tour makes only shortcuts with respect to that Euler tour (here we need the triangle inequality). We now show that

$$w(T) \leq \text{opt}, \tag{9.1}$$
$$w(M) \leq \text{opt}/2, \tag{9.2}$$

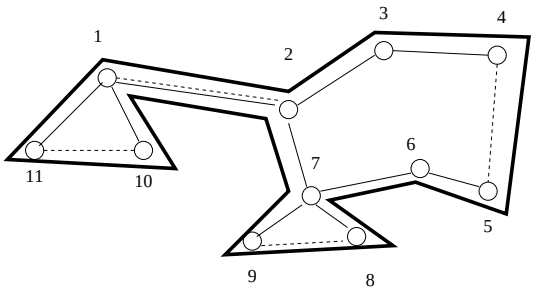


Figure 9.3: Euler tour

which implies the theorem. For this, let S be the edges of an optimal tour and $e \in S$. $S \setminus \{e\}$ is a path and in particular a spanning tree. Because T was the minimum-spanning tree, we get

$$w(T) \leq w(S \setminus \{e\}) \leq w(S) = \text{opt},$$

using the assumption that we have nonnegative edge weights. This shows (9.1). Now, let S' be the edge set of an optimal tour in the complete subgraph induced by V', the set of odd-degree vertices in T. Because $|V'|$ was even, S' has even length and decomposes into two matchings M_1, M_2 which alternate along S'. Because M was the minimum-weight matching, we get

$$w(M) + w(M) \leq w(M_1) + w(M_2) = w(S') \leq w(S) = \text{opt},$$

because the optimal tour in V' cannot be longer than the optimal tour through all vertices. This shows (9.2). □

I want to remark that the triangle inequality is crucial for this result. For general weighted graphs, the length of the optimal tour cannot be approximated up to a constant factor, unless $\mathbf{P} = \mathbf{NP}$.

9.2 Beyond Gomory Cuts

Gomory cuts are the weakest possible cuts, because they don't make use of any special problem structure. For most concrete problems like TSP, much better cuts are known. Intuitively, a cut is better if it cuts off more fractional solutions. The following lemma describes a class of cuts specific to the TSP, which is a better approximation of TSP-polytope than subtour polytope.

Lemma 9.2 *Let $\tilde{x}$ be the characteristic vector of a tour (meaning that $\tilde{x}_e = 1$ iff e is in the tour), $A \subseteq V$, $F \subseteq \delta(A) := \{e \in E \mid |e \cap A| = 1\}$, $|F|$ odd. Then $\tilde{x}$ satsifies the following, so-called* 2-matching inequality.

$$\sum_{e \subseteq A} x_e + \sum_{e \in F} x_e \leq |A| + \frac{|F| - 1}{2}.$$

The proof is simple and given as an exercise. This means, if we are given a fractional solution $\tilde{x}$ and are able to find a violated 2-matching inequality, we can add this inequality to our current set of known inequalities (which initially are just the subtour inequalities), and reoptimize. In fact, if $\tilde{x}$ violates some 2-matching inequality, such an inequality can be found in polynomial time. In other words, there is a polynomial-time separation oracle for the class of 2-matching inequalities.

There are many more known classes of inequalities which are satisfied by every tour, but which are not implied by other inequalities. Only at the point where a fractional solution is obtained which satisfies all these inequalities, the method of cutting planes gets stuck, unless one resorts to the very inefficient Gomory cuts. In this situtaion, the serach for the optimal tour usually continues with *branch & bound*.

In general, much of the work invested into solving real-life problems nowadays is spent in finding classes of inequalities which can be separated in polynomial time (meaning that a polynomial-time separation oracle exists for the polyhedron described by all inequalities in the class).

9.3 Branch & Bound

The branch & bound approach finally does what you might do if you were faced with an ILP: it enumerates all possible integral solutions to pick out the best one. However, it tries to do this in a clever way, so that many solutions need not be considered at all. This clerveness is wasted in the worst case, but in typical situations that occur in practice, it works quite well.

Given the problem

$$\begin{array}{ll} \text{(ILP)} & \text{maximize} \quad c^T x \\ & Ax \leq b, \\ & x \geq 0, \\ & x \in \mathbb{Z}^n, \end{array} \tag{9.3}$$

we first solve the LP relaxtaion by dropping the constraint "$x \in \mathbb{Z}^n$". If the resulting solution $\tilde{x}$ is integral, we are done, otherwise there exists a coordinate i such that $\tilde{x}_i \notin \mathbb{Z}$. Then we *branch*, meaning that we recursively solve the subproblems

$$\begin{array}{ll} \text{(ILP)}_1 & \text{maximize } c^T x \\ & Ax \leq b, \\ & x_i \leq \lfloor \tilde{x}_i \rfloor, \\ & x \geq 0, \\ & x \in \mathbb{Z}^n, \end{array}$$

and

$$\begin{array}{ll} \text{(ILP)}_2 & \text{maximize } c^T x \\ & Ax \leq b, \\ & x_i \geq \lfloor \tilde{x}_i \rfloor + 1, \\ & x \geq 0, \\ & x \in \mathbb{Z}^n, \end{array}$$

and return the better of the two solutions. The crucial observation that makes this work is that every integral solution is feasible in exactly one of the two subproblems, so it cannot be missed. When we do this, we obtain a *computation tree*, where every node corresponds to an ILP, and the edges connecting a node to its children correspond to a branching step.

The computation inside a node terminates if the LP relaxation that is being solved in that node is infeasible or returns an integral solution. The whole computation is guaranteed to terminate if the polyhedron $\{x \mid Ax \leq b, x \geq 0\}$ is bounded, like in case of TSP.

Up to now, this is not really clever. What makes the difference is the following *bound* step. Assume we traverse the computation tree in some order, and we have already solved the LP relaxations at certain nodes. Then we can maintain the quantity

$$z_I,$$

which denotes the best objective function value of any integral solution that has been found in solving the LP relaxations. In the beginning, we have $z_I = -\infty$, but throughout the tree, z_I will increase.

Whenever we solve the LP relaxation at a new node v, we obtain an optimal objective function value $z_{LP}(v)$, in the general case assumed by a fractional solution.

Fact 9.3 *If $z_{LP}(v) \leq z_I$, then no integral solution to the ILP at node v can be optimal for the original ILP (9.3).*

To see this, observe that any integral solution has objective function value at most $z_{LP}(v)$, which is already worse than the best integral solution found in some other branch of the computation. It follows that node v can be *killed*, i.e. we do not need to branch in v.

The hope is that the computation tree stays small, because many nodes are killed early in the computation. Of course, there are important factors that influence the efficiency of this scheme in practice. Among them are

- the order in which tree nodes are processed, and
- the choice of the index i such that $\tilde{x}_i \notin Z$, where $\tilde{x}$ is the solution to the LP relaxation in the node.

It is also important to start off with a value of z_I which is already large. This can ususally be done by computing a good integral solution to start with by some other method. For example, running Christofides' algorithm as a preprosessing stage for a branch& bound computation in the case of TSP leads to a good initial value of z_I already.

9.4 Branch & Bound for the TSP

Branch & Bound is a method which goes beyond ILP. It is applicable whenever we have a problem which can be partioned into subproblems in such a way that the optimal solution to the whole problem is found among the optimal solutions to the subproblems. Moreover,

we need upper bounds for the optimal objective function values in the nodes. In case of ILP, we can obtain them by using the LP relaxation, but this is not the only method which is available, even in case of ILP. For example, in case of TSP, the LP relaxation can be solved in polynomial time, but this is not very efficient in practice. If slightly weaker upper bound are obtainable by much simpler methods, this is preferrable. In general, there is a trade-off between the time it takes to compute the upper bound and its quality.

In the following, we outline an application of Branch & Bound for TSP which makes use of the fact that the *assignment problem* has a simple and fast algorithm. The algorithm is independent from the previous cutting plane approach, and it is even not clear how to combine it with the cutting-plane method directly. Note that the TSP is a minimization problem, so we need *lower* bounds for the optimal objective function values in the nodes, which we compare to a global *upper* bound z_I on the length of a best tour found so far.

Given a tour, we can encode it as a 0/1-vector in a way different from the one that has led us to the subtour polytope. Namely, we can introduce variables $x_{(p,q)}$ for order pairs of nodes, with the interpretation that

$$x_{(p,q)} = \begin{cases} 1, & \text{if } q \text{ is successor of } p \text{ in the tour,} \\ 0, & \text{otherwise.} \end{cases}$$

If $\tilde{x}$ is the characteristic vector of a tour, it must satisfy the equations

$$\sum_q x_{(p,q)} = 1, \forall p, \quad \sum_p x_{(p,q)} = 1, \forall q, \quad x_{(p,q)} \in \{0,1\}, \forall p, q, \tag{9.4}$$

because every vertex has exactly one successor and one predecessor in the tour. Therefore, if we minimize

$$\sum_{p \neq q} w_{\{p,q\}} x_{(p,q)}$$

over the constraints given by (9.4), we obtain a lower bound for the length of the optimal tour. It is only a lower bound, because the conditions of (9.4) are only necessary for a tour, but not sufficient: we might get subtours.

In return, the ILP we get has a nice structure: it is actually an assignment problem over a bipartite graph, where both vertex sets are copies of the vertex set of our original complete graph. Edge weights are the given edge weights, with the exception that $w_{(p,p)} = \infty$ which means that in a minimum-weight assignment, this edge will never be chosen.

Thus, we solve the assignment problem and then branch; again, we do not do it as indicated in the outline above for ILP, but according to some different rule which creates more than two branches. Namely, if the solution to the assignment problem does not give us a tour, there exists a proper subtour of length $k < n$, involving edges $e_i = (p_i, q_i), i = 1, \ldots, k$. Then we branch off into k subproblems, where problem i introduces the additonal constraint $x_{e_i} = 0$. Because the globally optimal tour cannot contain all edges of the subtour, it must be feasible for at least one of the subproblems. The subproblems can be considered as TSP problems again, after resetting $w_{\{p_i,q_i\}} = \infty$ for the i-th subproblem.

9.5 Branch & Cut

I have remarked above that Branch& Bound is usually invoked after all known cuts are exhausted, i.e. if a fractional solution has been obtained which cannot be cut off by any inequality from the classes of inequalities one has at hand for the concrete problem. However, in the subproblems generated during the branching, the cuts may become effective again. Thus, Branch& Cut interleaves Branch steps with Cut steps, applying the latter first whenever possible. With additonal tricks, Branch & Cut codes for the TSP are able to solve very large instances with several tenthousand cities.