Solutions to Exercises

Exercises from Sect. 2.1: Paths in Graphs

2.1 In terms of the balls-and-strings model, such a path exists if and only if the stringuv has no slack.

In terms of the distances computed by Dijkstra's algorithm, if $\ell(uv)$ is the length of uv, we must have $D[v] = D[u] + \ell(uv)$ or vice versa.

Note that whenever the edge uv satisfies this condition, we can pick t to be either u or v, more precisely the one of those two that is farther from s.

2.2 In terms of the balls-and-strings model, we are asking whether any of the balls drop farther down if we cut the string that represents uv.

If the stringuv has some slack, the answer is no. Otherwise, without loss of generality let us assume that v is the deeper of the two balls. Obviously, if there are some balls that would drop after we cut uv, ball$v$ must be among them. Therefore, it is sufficient to check whether this ball drops any farther. And to check that we just need to examine the strings that currently lead from v upwards.

In terms of the distances computed by Dijkstra's algorithm, we need to check whether there is a vertex $x \neq u$ such that $D[x] + \ell(xv) = D[v]$.

2.3 We can easily design graph classes in which the total number of paths between two given vertices grows exponentially in the number of vertices. One such graph class is shown in Fig. 1. Therefore, the best we can hope for in terms of time complexity is a solution that is linear in the output size.

Before actually generating the paths, we will run Dijkstra's algorithm twice: once from s, computing the distances $D_s[\cdot]$, the other time from t, computing the distances $D_t[\cdot]$. Using these two sets of distances, we can easily write a recursive algorithm that uses backtracking to generate all paths from s to t, and nothing else. The main observation that helps us avoid all the other paths: a vertex x lies on some paths from s to t if and only if $D_s[x] + D_t[x]$ equals the shortest distance between s and t (i.e., $D_s[t]$).

2.4 In terms of the balls-and-strings model, note that when the model is hanging by the vertex s, all the edges in all paths from s to t go straight downwards. If we take

M. Forišek and M. Steinová, *Explaining Algorithms Using Metaphors*, SpringerBriefs in Computer Science DOI: 10.1007/978-1-4471-5019-0,

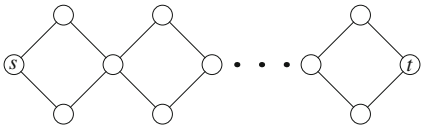


Fig. 1 A graph class with an exponential number of paths. (All edges have unit lengths.)

the original graph, leave only those edges, and direct them accordingly, we would get a directed acyclic graph. In Exercise 2.3, we have shown how to generate all paths through such graph using backtracking. Now we need to count those paths.

We can easily add memoization to the algorithm from Exercise 2.3: for each vertex v we will compute and store the number of $s - t$ paths that pass through v—or, equivalently, the number of paths from v to t. When processing a vertex v, we find all its possible successors on the path, recursively find the number of paths for each of them, and sum those counts to get the total number of paths from v to t.

An equivalent solution using dynamic programming starts by computing the distances from s to all other vertices, and then processes the vertices ordered by distance, starting with t and ending with s.

2.5 Let uv be the edge we are going to shorten. In terms of the balls-and-strings model, consider the amount of slack the corresponding string has. If we are going to shorten it by a strictly greater amount, some of the distances will change—the new shorter string will pull the ball on its lower end upwards from its current location. Otherwise, there will clearly be no changes to the graph.

In terms of the distances computed by Dijkstra's algorithm, we need to check whether $|D[u] - D[v]|$ is more than the new length of uv.

2.6 Without loss of generality, let v be the vertex displaced by shortening the edge uv. In the balls-and-strings model, the ball$v$ may pull a set of other balls upwards. We need to discover those balls and recompute the new distances for them. This can be done by marking v as the (currently) only unfinished vertex, resuming Dijkstra's algorithm and marking vertices as unfinished if their distance gets improved.

2.7 Whenever we improve the distance to a vertex, we remember the edge used to do so. When the algorithm terminates, we have a set of $n - 1$ remembered edges (one for each vertex except for s). These edges clearly form a tree of paths. The other $m + n - 1$ edges can be safely removed. Removing more edges would leave the graph disconnected, thus this number of removed edges has to be optimal.

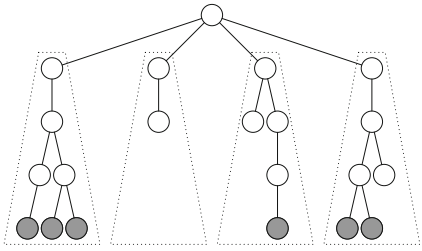


Fig. 2 A tree with an even diameter, rooted at its center. *Shaded* vertices are the ends of all longest paths. There are exactly 11 distincts longest paths in this tree

2.8 Fix a particular path. For each string on it: cut it, measure the new depth of t, and then glue the string back. The smallest new depth of t is the length of the second path. This algorithm can be implemented directly, and its running time is clearly polynomial in the number of vertices. (However, note that there are more efficient algorithms for this problem.)

Exercises from Sect. 2.2: Longest Paths in a Tree

2.11 In the described case, it is better to visualize the tree hanging by the center vertex, as shown in Fig. 2. Let d_i be the number of vertices in the i-th subtree that have the globally maximal depth. Each longest path is uniquely determined by picking two of these vertices as its endpoints, with one condition: the chosen two endpoints must not be in the same subtree. The number of longest paths can easily be counted by counting all possible pairs and subtracting the incorrect ones: $\left((\sum d_i)^2 - \sum d_i^2\right)/2$.

2.12 Yes, it does. Exactly the same visualization as in the unweighted version can be used to prove it. (For integer edge lengths you can convince yourself easily by subdividing each edge into edges of length 1 by inserting additional vertices.)

Moreover, it can be implemented in exactly the same way—as the path between any pair of vertices is unique, any tree traversal can be used to compute path lengths and find the farthest vertex.

2.13 No, the algorithm does not work. The improved version given in the next exercise does not work either, we will give a counterexample below.

2.14 On a dense graph, a single breadth-first search takes $\Theta(n^2)$ time. In each iteration, the distance between the current pair of vertices increases. And as the diameter cannot exceed $n - 1$, there will be at most $n - 1$ executions of breadth-first search, for a total time complexity bound $O(n^3)$.

(At this point, there comes a next exercise: is this upper bound actually tight?)

2.15 A counterexample is shown in Fig. 3. Suppose that the algorithm started in the vertex a. The only vertex at distance 3 is b, all the others are strictly closer. In the next step, we start at b and discover that a is the farthest vertex at distance 3. The algorithm now terminates. However, the conclusion is false—the distance between vertices x and y is obviously 4.

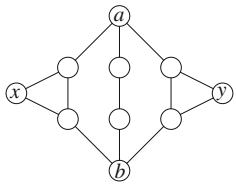


Fig. 3 The tricky longest path algorithm may fail to find the diameter of a general graph

2.16 It is quite common to confuse the diameter and the length of the longest simple path. In trees these two coincide, but in general graphs (even if the edges all have unit length) these are two different concepts. Verifying whether there is a simple path of a given length k is NP-complete—checking for the presence of a Hamiltonian path is a special case of this problem.

On the other hand, the diameter can easily be computed in polynomial time: we can just compute the distance for each pair of vertices, and then output the maximum of these values.

Exercises from Sect. 3.1: 2D Shortest Path

3.1 Yes in both cases. The rubber band metaphor still applies. For a circular obstacle the rubber band will necessarily form a tangent of the circle. Hence, when building the visibility graph, we need to consider tangents from a vertex of a polygon to a given circle, and common tangents of a pair of circles. There is only a constant number of these for any given pair of objects, and they can be found in constant time. See Fig. 4 for an example.

3.2 As suggested by the hint in the statement, there can be situations where the rubber band touches an edge of a polyhedral obstacle. Why is this a problem? Because in such cases we cannot easily tell where exactly the rubber band will touch the edge. In general, there is an uncountable number of possible points where the touch can occur, so the technique of producing a finite visibility graph fails.

However, this should not seen as a shortcoming of our metaphor. Neither should it be seen as just our inability to come up with the right formula to compute the point where the rubber band touches the edge. It has been proved in [15] that in three dimensions the path problem with polyhedral obstacles is NP-hard.

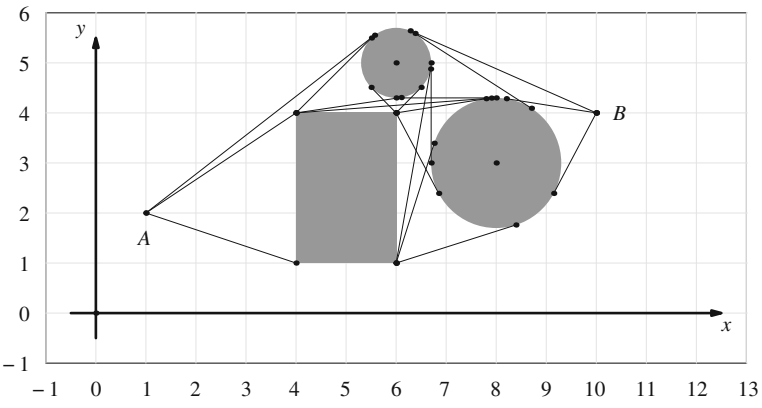
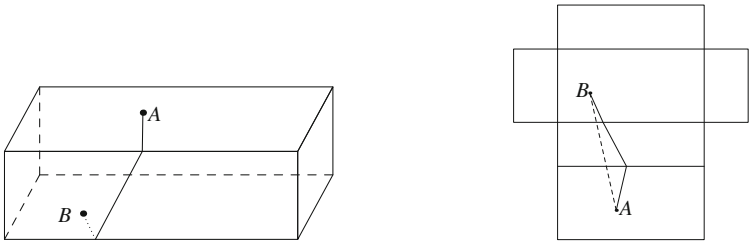


Fig. 4 The visibility graph for polygonal and circular obstacles



(a) A path from A to B on the surface of a box

(b) The same box unfolded into a plane, showing the same path (full) and a shorter path (dashed)

Fig. 5 The path problem on two-dimensional surfaces in three dimensions

3.3 The path we seek clearly lies entirely on the surface of the box. Moreover, on each face of the box the path has to be a straight line segment.

Figure 5 shows a sample box and one possible way how it can be unfolded to form a planar surface. Clearly, for the optimal path there must be a way of unfolding the box such that the optimal path turns into a straight line segment that connects the two given points. As for a box there are only a few ways to unfold it, we can try them all and pick the best solution overall.

The unfolding technique was later used in the design of more general algorithms. For example, [16] give a quadratic time algorithm to find the path on the surface of a general (not necessarily convex) polyhedron.

3.4 Basically, the instability occurs around any instance with multiple distinct optimal paths. Consider the instance shown in Fig. 6. In this instance there are two optimal solutions: one goes above the rectangle, the other one below. If we shifted point A slightly upwards or downwards, we would get two very similar instances, each with only one solution.

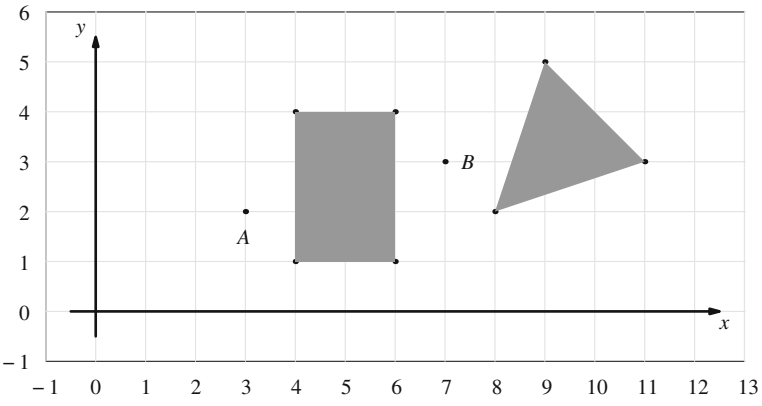


Fig. 6 An instance with two distinct paths

3.5 After we finished the path algorithm on the visibility graph, we can use the computed information to reconstruct all possible paths—or, more precisely for each edge incident to the target vertex we know whether there is a path that uses this edge.

If there is only one such edge (this is usually the case), we have to move B farther away in the direction of the edge. If we leave B in any other direction, the rubber band will be able to contract. Hence it is sufficient to check one position of B. To do this check, we could recompute the path from A to the new B, and compare the distance to the one we expect to find. However, it is sufficient to compute the distances from the new B to the directly visible vertices of the visibility graph. For each of them we already know their distance from A.

If there is more than one optimal direction from which we can reach B in the original graph, the answer is always negative.

Here is an additional exercise. Consider the following simpler algorithm: If all paths reach B using the same edge, move B in the direction of that edge and give a positive answer, otherwise give a negative answer.

Does this algorithm work? Or is the additional check we introduced above really necessary?

Exercises from Sect. 3.2: Distance Between Segments

3.6 Yes, for line segments the two statements are actually equivalent—whenever we have a configuration where neither of the cars moves, nor their distance is not only locally, but also globally minimal. See the answer to the next exercise for more.

3.7 Almost always there is a single configuration where the distance is locally minimal—the globally minimal one. The only situation with more than one local minimum is the one shown in Fig. 3.13 (on p. 41): with two parallel segments there can be multiple optimal configurations.

3.8 For two circles the rubber band metaphor can again be used to easily analyze the possible cases. The railroad car on a circular track will not move if and only if the force pulls it exactly toward the center, or exactly away from it. Hence, if the circles do not intersect, in the optimal configuration the two cars both lie on a line that connects the centers of the two circles. (What happens in the case when the two circles share the same center?)

For the circle-segment case we again get finitely many cases to check: in the optimal configuration, the rubber band has to be orthogonal to the circle, and either orthogonal to the segment, or at its endpoint.

Note that when one of the objects is a circle, in addition to the stable equilibrium that corresponds to the global optimum there is also an unstable equilibrium for each car: the opposite end of the circle.

3.9 The proposed algorithm does not work. The boy in Fig. 3.17 (on p. 47) is outside of the polygon, yet he cannot see the outside in any direction.

Exercises from Sect. 3.3: Winding Number

3.11 Clearly, all points that have a nonzero winding number are also enclosed by the polyline. However, the opposite inclusion does not have to hold. As shown in Fig. 7, it is possible to have a part of the plane that has a winding number zero but it is still enclosed by the polyline.

3.12 We can easily find the leftmost vertex of the polyline (any of them, if there are multiple). Starting at this vertex, we can walk around the polyline in counterclockwise direction and construct its boundary. (At each self-intersection we change direction to follow the next edge in counterclockwise order.)

The result of this procedure is a new polyline that encloses the same set of points. The new polyline may touch or overlap itself, but it never crosses itself. Hence we can use the winding number or the ray casting algorithm to test whether the given point lies inside this new polyline.

As an example, the boundary of the closed polyline in Fig. 7 is a new closed polyline with 13 vertices.

3.14 For each edge of the polyline, the boy turns by less than 180° (half a circle) while the girl traverses that edge. Hence, the number of full circles the girl can make has to be strictly less than $n/2$. For each $n \geq 3$ we can construct a point and a polyline such that the winding number of the point with respect to the polyline will be $\lfloor (n-1)/2 \rfloor$, which is necessarily optimal.

As a simple example, for odd $n \geq 5$, the set of all n longest diagonals of a regular n-gon forms such a polyline.

Exercises from Sect. 3.4: Triangulation

3.15 Among the polygons with the smallest number of vertices, every triangle and every concave polygon has a unique triangulation. For any $n \geq 3$, we can construct an example polygon with a unique triangulation by starting with a triangle and repeatedly adding new triangles such that each new triangle shares and edge with the previously constructed polygon, and the third vertex of the new triangle is placed in such a way that it has no inner diagonal leaving it. See Fig. 8 for a small example.

3.16 Each polygon has a triangulation, therefore each polygon with n vertices has to have at least $n-3$ inner diagonals. We now claim that the triangulation of a

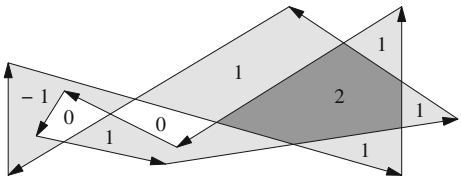


Fig. 7 Some of the finite regions of the plane may have winding number equal to 0

polygon is unique if and only if it has *precisely* $n-3$ inner diagonals. This is because as soon as a polygon has more than $n-3$ inner diagonals, it has a triangulation and an inner diagonal$d$ that is unused in that triangulation. If we now use the diagonal d to split the polygon into two smaller ones and triangulate each of them, we will obtain a different triangulation of the same polygon.

The number of inner diagonals can easily be computed in polynomial time. The simplest algorithm runs in $\Theta(n^3)$: for each diagonal and each edge, check whether they intersect.

There are also more efficient algorithms to check whether a polygon has a unique triangulation. Mirzaian [17] shows that it is possible to check whether a given triangulation is unique in $O(n)$ time: it is sufficient to check whether the triangulation contains a diagonal such that the quadrilateral formed by its two incident triangles is convex. Together with Chazelle's $\Theta(n)$ algorithm [18] to find one triangulation this solves our problem in linear time.

3.17 We will construct the coloring recursively. For a triangle the solution is simple, Now suppose we have a polygon with $n > 3$ vertices. As there are $n-2$ triangles and the polygon has n sides, clearly there has to be a triangle that contains two sides of the polygon. Also, those sides clearly have to be adjacent. If we remove such a triangle, we obtain a polygon with $n-1$ vertices. We color the smaller polygon recursively. Then we reattach the removed triangle T_1. It shares a diagonal with exactly one other triangle T_2. We look at the color of T_2 and color T_1 using the opposite color.

3.18 Note the triangles labeled 1 through 9 in Fig. 9. Each of these triangles must have a different color from its two neighbors, and that clearly cannot be done with just two colors. In other words, the dual graph is not bipartite, as these nine triangles correspond to an odd-length cycle in the dual graph. Therefore, two colors are not sufficient to color some of the general triangulations.

For an upper bound, we can use the same recursive technique as in Exercise 3.17 to produce a coloring that only uses a few colors. In any triangulation of a polygon with holes there is a triangle that shares at least one side with the boundary of the polygon (including the boundaries of holes, if any). We can look for such a triangle, remove it, and recursively color the rest of the polygon. When we reattach the removed triangle, it becomes adjacent to at most two colored triangles; therefore, we surely have at least one available color for the new triangle.

3.19 Yes, it can. The only exception is the case where the outside polygon is convex, but in that case finding an inner diagonal is easy.

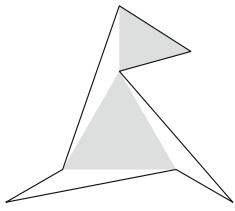


Fig. 8 A 7-vertex polygon with a unique triangulation

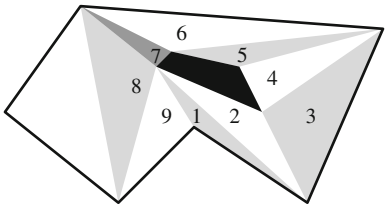


Fig. 9 The proof that *three colors* is required to color this triangulation.

Exercises from Sect. 4.1: Stacks and Queues

4.1 The main difference is in the direction of the "pointers" stored in the data structure. In a linked list implementation of a queue each item is stored along with a pointer to the *next* item, whereas in the waiting room everyone remembers the *previous* patient.

This is a consequence of a more basic difference between the two scenarios: while the items in the linked list are inanimate (and therefore have to be manipulated by an external mechanism), the patients are active agents. From this point of view, the waiting room is a good metaphor for event-driven object oriented programming.

4.2 There are many correct answers to this exercise. One nice example: Each Christmas, I tend to receive some books as presents. And whenever I have a new book, many of my family members and friends are also interested in reading it. This usually evolves into a linked list-like queue: each person remembers an instruction of the type "when you finish reading the book, give it to Deny".

Exercises from Sect. 4.2: Median

4.3 The algorithm works exactly in the same way: we are looking for a location such that there is one half of all people on each side of the location. To prove its correctness, simply assume that each person has their own house. The only difference between this task and the unweighted version is that now there can be multiple houses at the same location.

4.4 Again, we can use the same reasoning. Let v be any vertex of the tree. Assume that we currently have the bus stop in the vertex v. The vertex v has some outgoing edges. There are two possible cases:

(a) The vertex v has a neighbor w such that more than one half of all vertices of the tree lies on the w side of the edge vw. In this case, people living on the w side would manage to pull the bus stop from v to w. In other words, w is a better place for the bus stop, because when moving it from v to w, we are decreasing the walking distance for a majority of people.

(b) The vertex v has no such neighbor. In that case the bus stop will not move from v: regardless of the direction, at least one half of all people will oppose it.Therefore, when looking for the globally optimal location for the bus stop, we are looking for vertices of the second type. The last observation we need is that there can be at most two such vertices in the entire tree, and if there are two, they have to be neighbors. This can easily be shown: if u is a vertex of the second type and v is its neighbor, at most one half of all vertices is on the v side of the edge uv—and therefore at least one half of them is on the u side. Any vertex on the v side other than v will therefore have more than one half of all vertices attached to one of its neighbors.

This gives us a simple greedy algorithm: Place the bus stop into any vertex. While the current vertex is of the first type, move the bus stop to the "large" neighbor. Once this process stops, you have found an optimal location. (The set of all optimal locations either contains this vertex only, or it contains the entire edge leading into one of its neighbors.)

Note that, as in the version on a line, edge lengths do not matter, the optimal location of the bus stop is determined by the shape of the tree only. Also note that the same reasoning can be used to solve the weighted version, we will just use the sum of vertex weights (i.e., the number of people) instead of the count of vertices.

This algorithm can be implemented in $O(n)$ if we first run a depth-first search on the tree to precompute subtree sizes. There is also a different algorithm that solves the problem in $O(n)$: during the first depth-first search we can also compute the sum of walking distances if the bus stop happens to be in the root of the depth-first search. Then, we run a second depth-first search from the same vertex. During this search we take the bus stop with us and update the sum of walking distances as we move along the graph.

4.5 The key to solving this task is to realize that we can break it down into two independent one-dimensional problems. Any solution has to consist of two independent parts: The result of north/south movements of all people has to be that they all end up on the same latitude. At the same time, all their east/west movements have to bring them to the same longitude.

Hence, the optimal meeting place can be determined by computing their median latitude and longitude. (For odd n, there will be a single optimal meeting place. For even n there may be a rectangle of optimal meeting places.)

4.6 Clearly, in the optimal solution there will be some k such that the people from the first k houses in the village use the first bus stop, and people from the other $n - k$ houses use the second bus stop. We can try all possibilities for k and pick the best one. Given k, we can find the optimal placement for each bus stop separately using the original algorithm. This can be done in constant time, but then we have to spend linear time to compute the sum of distances the people who use the bus stop have to walk. (We need this in order to compare the solutions for different k.) The time complexity of this algorithm is $\Theta(n^2)$.

The above solution can be improved to $\Theta(n)$. For each bus stop we will store the current sum of distances. As we change k, we simulate moving the bus stops and change the sum of distances accordingly.

4.7 Consider any two kids that fill their buckets immediately after one another. If the second kid has a smaller bucket than the first one, we can swap them. (The sum of their waiting times decreases, and none of the other waiting times is influenced by this change.)

Hence, if the sequence in which the kids fill their buckets is *not* sorted according to bucket size, the sum of all waiting times is *not* optimal. This leaves us with just a single candidate for the optimal order.

4.8 Consider the distance between the first and the last house in the village. If this distance exceeds 2 km, it is obvious that the bus stop cannot be within 1 km from each of them at the same time. On the other hand, if the distance between these two houses is 2 km or less, we can find a valid placement for the bus stop by placing it halfway between the first and the last house in the village. (This placement does satisfy the 1 km constraint for each house, but will usually turn out to be worse than optimal in terms of average walking distance.)

Let us take a closer look at how the new constraint limits the possible choices for the bus stop location. For each house, we have a 2 km long interval that has to contain the bus stop. The intersection of all these intervals is the set of all valid locations for the bus stop. Clearly, the intersection of all those intervals is again an interval, and we can compute it as the intersection of just two intervals: the one for the first house and the one for the last house.

If there is an optimal location of the bus stop (computed without the 1 km constraint) that lies within the allowed interval, we are lucky—we can choose that location and be certain that our choice is optimal.

The other possibility is shown in Fig. 10. As the globally optimal solution (i.e., the median coordinate) lies outside of the allowed interval, we can make the following conclusion: If we place the bus stop at any allowed location, there will always be more houses on one side than the other. Hence, the villagers will always want to move the bus stop closer toward the median. Therefore, clearly the best allowed location is at that endpoint of the allowed interval that is the closest to the median.

4.9 There is an efficient greedy solution. Order the houses according to their coordinate. Consider the leftmost house in the village. This house must have a bus

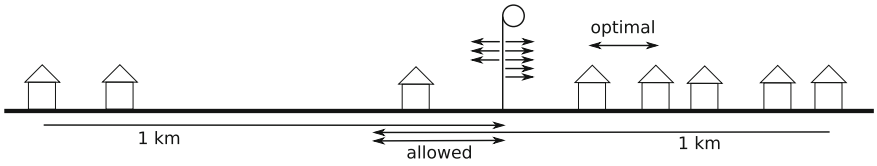


Fig. 10 The additional constraint on 1 km proximity limits the possible locations of the bus stop to an interval. If the globally optimal bus stop location happens to be outside that interval, the optimal allowed solution is at the closer endpoint of the allowed interval

stop within 1 km. Clearly, the optimal place for that bus stop is exactly 1 km in the direction in which the rest of the village lies—no other placement can "cover" more other houses. We repeat the process (pick the first house that is not covered, place a bus stop 1 km behind it) until no more houses remains.

This algorithm can be implemented with time complexity $O(n \log n)$. The slowest part is the sorting; the covering phase can be done in linear time.

Exercises from Sect. 4.3: Substring Search

4.10 Six of the seven lemmings have survived the previous step. Therefore, there must be at least six different pits labeled x in the gadget. The conclusion: N contains at least six occurrences of the letter x. (Note that it may well contain more than 6 xs.)

4.11 The answer is $n - S[n]$. In words: the smallest period is the distance between the two rightmost lemmings in the gadget at the moment when the rightmost lemming reached the end of the gadget.

We are looking for the smallest p such that for each valid i we have $N[i] = N[i + p]$. In other words: Let $k = n - p$. We are looking for the largest k such that the string formed by the *first* k letters of N is the same as the string formed by the *last* k letters of N. (See Fig. 11.)

Suppose that we used the gadget for the string N, and that we read the string N to get one of the lemmings to the end of the gadget. Let us take a look at the position k. Does this position currently contain a lemming? During its lifetime the corresponding lemming had to cross the first k pits in the gadget. On the other hand, during its lifetime the corresponding lemming would hear the *last* k letters of N. Therefore, position k contains a lemming if and only if the first k letters of N are the same as the last k letters.

Hence, the position of each lemming defines one possible period. As we want the shortest period, i.e., the largest k, we are interested in the second lemming from the right. And that lemming is located at the position $k = S[n]$.

4.12 Let $N[x..y]$ denote the substring formed by the letters x through y of N. We know that the rightmost two lemmings are at positions 17 and 12, therefore (by the same argument as in the solution of Exercise 4.11) we have $N[1..12] = N[6..17]$. This means that the shortest period of the string $N[1..17]$ is $p = 5$. But then the string also must have the period $2p = 10$, therefore $N[1..7] = N[11..17]$. And from this equality it follows that there will certainly be a lemming at position 7.

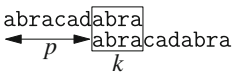


Fig. 11 The shortest period p is determined by the longest k such that the prefix and suffix of length k match

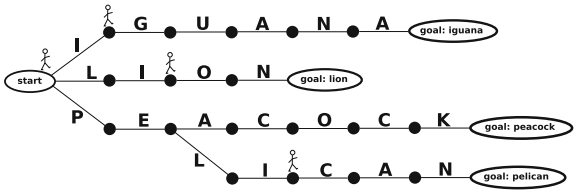


Fig. 12 A schematic view of the Aho-Corasick gadget for the words *IGUANA*, *LION*, *PEACOCK*, and *PELICAN*. From the position of the lemmings we can infer the last four letters read: *PELI*

Now suppose there is a lemming at position 11. As there is another one at position 12, it follows that $N[1..12]$ is periodic with period 1. In other words, $N[1..12]$ consists of 12 copies of the same letter. Then, from $N[1..12] = N[6..17]$ we get that $N[1..17]$ has to consist of 17 copies of that letter. But that contradicts the fact that the lemming at position 12 is the second one from the right. Therefore, there cannot be a lemming at position 11.

4.13 We can extend the KMP gadget into one that can look for multiple patterns at the same time. The main trick is to change the single path into a tree of paths. More precisely, the layout of the gadget will directly correspond to a trie that contains all the words we wish to find. The behavior of lemmings will still be deterministic, we just add a single new rule: If you have multiple options how to go on, and one of them currently has a bridge, pick that option.

Figure 12 shows, in a schematic way, a gadget that can be used to look for four words at the same time.

In order to find all occurrences of all words within a given string H, we can use the same process as before: we read the individual letters of H while following the rightmost lemming. (Note that only one lemming is released each second, therefore there cannot be two lemmings tied for being the rightmost one—or for any other position.)

The precomputed information now has to be generalized as well: now we need to answer the question: "If the rightmost lemming is in this note of the trie, in which node is the second lemming from the right?" Note that the answer node may be in a different branch from the question node—as is the case in Fig. 12.

The construction above is the basis of the Aho-Corasick algorithm.

Note that in the general case there is one more complication we did not have to deal with. In general, it is possible that our set of patterns contains two patterns such that one of them is contained in the other. For example, if we are looking for the strings *BAT* and *ALBATROSS* in the string *ALBAT*, we will follow the rightmost lemming on its way through the "albatross branch" of the gadget, completely missing the fact that one of the other lemmings reached the goal in the "bat branch". In the Aho-Corasick algorithm this is handled by adding output links; we omit the details here.

Index

M. Forišek and M. Steinová, *Explaining Algorithms Using Metaphors*, SpringerBriefs in Computer Science DOI: 10.1007/978-1-4471-5019-0,