KØBENHAVNS UNIVERSITET

Introduktion til diskret matematik og algoritmer: Problem Set 4

Due: Wednesday March 25 at 12:59 CET.
Submission: Please submit your solutions via *Absalon* as a PDF file. State your name and e-mail address close to the top of the first page. Solutions should be written in LaTeX or some other math-aware typesetting system with reasonable margins on all sides (at least 2.5 cm). Please try to be precise and to the point in your solutions and refrain from vague statements. Never, ever just state the answer, but always make sure to explain your reasoning. *Write so that a fellow student of yours can read, understand, and verify your solutions.* In addition to what is stated below, the general rules for problem sets stated on *Absalon* always apply.
Collaboration: Discussions of ideas in groups of two to three people are allowed—and indeed, encouraged—but you should always write up your solutions completely on your own, from start to finish, and you should understand all aspects of them fully. It is not allowed to compose draft solutions together and then continue editing individually, or to share any text, formulas, or pseudocode. Also, no such material may be downloaded from or generated via the internet to be used in draft or final solutions. Submitted solutions will be checked for plagiarism.
Grading: A score of 120 points is guaranteed to be enough to pass this problem set.
Questions: Please do not hesitate to ask the instructor or TAs if any problem statement is unclear, but please make sure to send private messages—sometimes specific enough questions could give away the solution to your fellow students, and we want all of you to benefit from working on, and learning from, the problems. Good luck!

1 (90 p) When Jakob has international visitors, he needs to give them travel directions from Kastrup Airport to the Department of Computer Science (DIKU) at Universitetsparken. Jakob is aware of the following relevant public transport options in Copenhagen with travel times as stated (in either direction, and with time for switching transport mode included):

- Between Kastrup Airport and Kongens Nytorv by metro: 13 minutes.
- Between Kastrup Airport and København H by train: 20 minutes.
- Between København H and Nørreport by train: 3 minutes.
- Beween København H and Kongens Nytorv by metro: 4 minutes.
- Between Kongens Nytorv and Vibenshus Runddel by metro: 10 minutes.
- Between Vibenshus Runddel and Universitetsparken by foot: 9 minutes.
- Between Nørreport and Universitetsparken by bus: 8 minutes.

What is not so clear to Jakob is how he should use this information to find as fast a route as possible between the airport and DIKU to suggest to his visitors.

1a (20 p) Help Jakob by modelling this problem as a graph. Explain what the vertices and edges represent and what other information you need to add to the graph. Make sure to show concretely what graph you obtain for Jakob's problem above.

Solution: We create a graph with 6 vertices:

- vertex A for Kastrup Airport,
- vertex K for Kongens Nytorv,

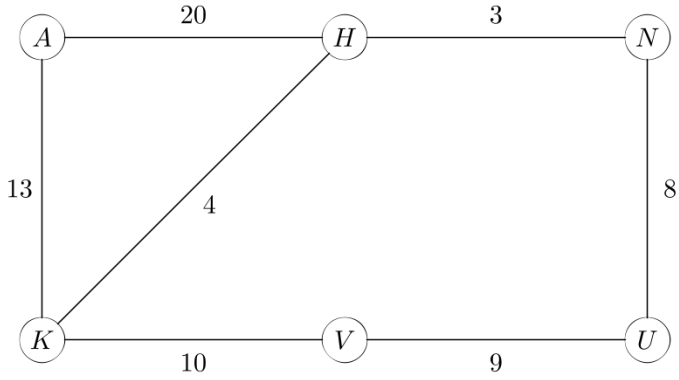


Figure 1: Graph modelling public transport options from Kastrup Airport to DIKU in Problem 1.

- vertex H for København H,
- vertex N for Nørreport,
- vertex V for Vibenshus Runddel,
- vertex U for Universitetsparken and DIKU.

Between these vertices we add edges as specified by the travel options with weights equal to the travel times, and these edges are undirected since the time given is for travel in either direction:

- edge (A, K) with weight 13,
- edge (A, H) with weight 20,
- edge (H, N) with weight 3,
- edge (H, K) with weight 4,
- edge (K, V) with weight 10,
- edge (V, U) with weight 9,
- edge (N, U) with weight 8.

This yields the graph in Figure 1.

1b (70 p) Propose a suitable graph algorithm to solve Jakob's problem. Explain what this algorithm is and why it is the right choice for this problem. Make a dry-run of the algorithm and explain the relevant steps in the execution (similarly to what has been done in class and in the lecture notes).

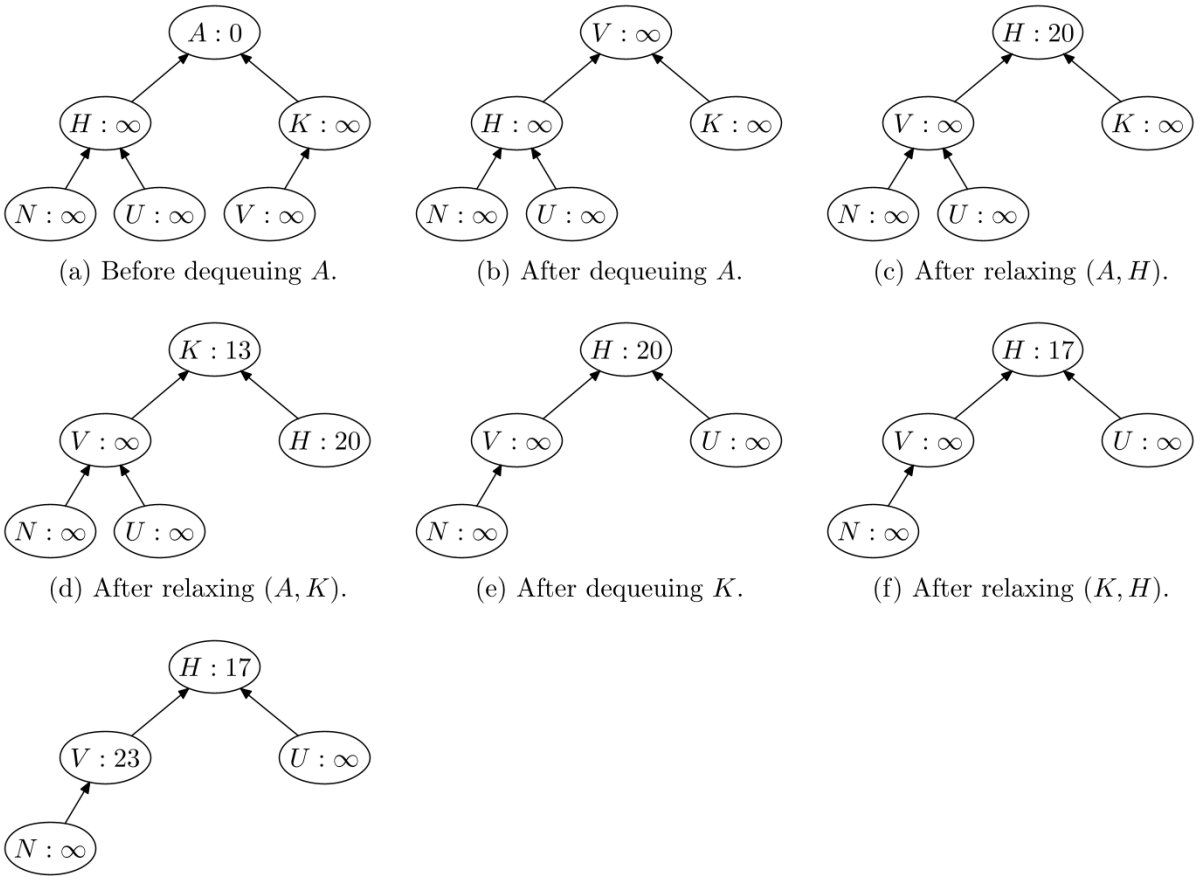
If your algorithm uses any auxiliary data structures, then explain in detail for the first two vertices processed how these data structures change. For the rest of the algorithm execution, just report what the relevant outputs of the data structures are without going into any details.

What travel directions for Jakob's visitors does your algorithm produce?

Solution: What Jakob is looking for is the fastest route from Kastrup to DIKU, i.e., the shortest path in a graph where the edge weights are the travel times between different locations. We have learned in the course to compute such shortest paths by using Dijkstra's algorithm.

Following the instructions in the problem statement, when running Dijkstra's algorithm we illustrate in Figure 2 how the heap used for the priority queue changes during algorithm execution, using the notation $v : k$ for a vertex v with key value k. At the outset, the vertex A corresponding to Kastrup Airport has key 0 and all other vertices have key ∞ as in Figure 2a.

1. After vertex A has been dequeued, vertex V is moved to the top of the heap and we have the configuration in Figure 2b. Relaxing the edge (A, H) gives value 20 to H (i.e., the



(a) Before dequeuing A. (b) After dequeuing A. (c) After relaxing (A, H).

(d) After relaxing (A, K). (e) After dequeuing K. (f) After relaxing (K, H).

(g) After relaxing (K, V).

Figure 2: Heap configurations for priority queue in Problem 1.

weight of the edge), and and the min-heap property is restored by letting H bubble up above V, yielding Figure 2c. Relaxing (A, K) decreases the key of K to 13 and makes H bubble up above K, yielding Figure 2d. These are all edges incident to A that we need to consider for relax operations.

2. Since vertex K is at the top of the heap, it is dequeued next. This adds the edge (A, K) to the spanning tree, which we indicate in Figure 3. When K is removed, U is moved to the top. This violates the min-heap property, since the key of U is not smaller than or equal to those of its children. Since H has smaller key than V, we swap U and H. This restores the heap property (since the left subtree of the root was not changed, and U is now the root of a singleton subheap), and so the heap after removal of K looks as in Figure 2e.

   Relaxing the edge (K, H) decreases the key value of H to 17. Since H is already the root, the heap does not change except for this key update and now looks like in Figure 2f. Relaxing (K, V) decreases the key value of V to 23, but again does not lead to any structural changes in the heap since V still has a larger key than its parent H (see Figure 2g). There are no further edges incident to K to relax. According to the instructions in the problem statement, we do not need to provide any further heap illustrations from this point on.

3. Since H is now the vertex with the smallest key value it is dequeued next, and the

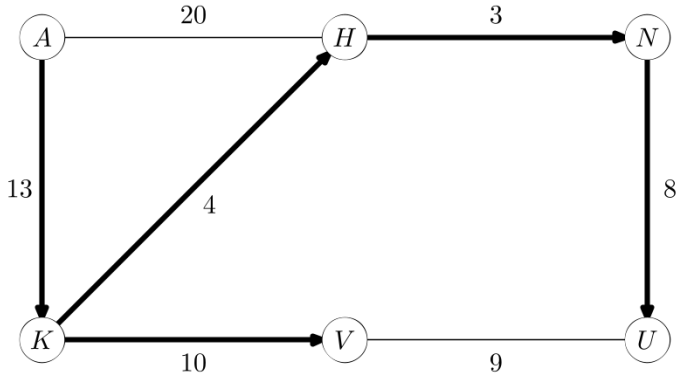


Figure 3: Shortest path tree for public transport from Kastrup Airport to DIKU in Problem 1.

edge (K, H) is added to the spanning tree (since the latest update of the key value of H was when relaxing the edge (K, H)). When the edge (H, N) is relaxed the key of N is updated to 20.

4. Vertex N currently has the smallest key 20 and is dequeued next. This adds the edge (H, N) to the spanning paths tree, since the key of N was last updated when (H, N) was relaxed. Relaxing (N, U) updates the key of U to 28.

5. Now vertex V has the smallest key 23 and so is dequeued, adding (K, V) to the spanning tree. When we relax (V, U) nothing happens since $23 + 9 \geq 28$.

6. Finally, vertex U is dequeued, adding the edge (N, U) to the spanning tree. The queue is now empty, and there is nothing to relax.

The shortest path spanning tree computed as described above is indicated by the bold edges in Figure 3. From this tree we can read off that the fastest route for Jakob's guests is Kastrup Airport – Kongens Nytorv – København H – Nørreport – Universitetsparken/DIKU.

2 (120 p) Consider the graph in Figure 4 and the following ordered sequences listing the vertices in this graph:

1. a, c, e, b, h, g, f, d.
2. a, b, c, e, d, h, g, f.
3. a, c, b, e, h, g, f, d.
4. a, b, d, c, e, h, g, f.

For each of the sequences above, determine whether it can be the result of:

(a) a breadth-first search with vertices listed in order of visits;
(b) a depth-first search with vertices listed in order of discovery;
(c) a shortest-path computation with vertices listed in the order they are removed from the priority queue.

Each sequence above is the output of at most one of the algorithms, but since there are four sequences there could be a sequence that cannot be produced by any of the algorithms.

Partial credit is given for matching algorithms and vertex sequences correctly. For full credit, you need to give an overview of how and why the algorithms process the vertices in the given order (such as explaining the order of recursive calls, edges processed, or similar), or why none of the proposed algorithms could yield the sequence in question, but you do not have to provide detailed information about any auxiliary data structures used in the algorithms.

Solution: Let us consider the different algorithms in the order listed and try to find sequences that would match their outputs. Since we are promised that each sequence is the output of at most one algorithm, we are done with each sequence as soon as we find a match.

Sequence 2 is the result of breadth-first search starting with vertex a. To see this, consider how the different vertices are enqueued in and dequeued from the queue if we start the breadth-first search in vertex a:

Dequeued vertex	**Enqueued vertices**	**Queue**
—	—	(a)
a	$\{b, c, e\}$	(b, c, e)
b	$\{d\}$	(c, e, d)
c	—	(e, d)
e	$\{h\}$	(d, h)
d	—	(h)
h	$\{g\}$	(g)
g	$\{f\}$	(f)
f	—	$()$

Sequence 4 is the result of depth-first search starting with vertex a. Below follows an illustration of in which orders vertices are discovered and finished:

Discover a — undiscovered neighbours b, c and e
- Discover b — undiscovered neighbour d
  - Discover d — undiscovered neighbour c
    - Discover c — no undiscovered neighbours
    - Finish c, since no undiscovered neighbours
  - Finish d, since no undiscovered neighbours left
- Finish b, since no undiscovered neighbours left
- Discover e — undiscovered neighbour h
  - Discover h — undiscovered neighbour g
    - Discover g — undiscovered neighbour f
      - Discover f — no undiscovered neighbours
      - Finish f, since no undiscovered neighbours
    - Finish g, since no undiscovered neighbours left
  - Finish h, since no undiscovered neighbours left
- Finish e, since no undiscovered neighbours left

Finish a, since no undiscovered neighbours left

Sequence 1 is the result of a call to Dijkstra's algorithm computing shortest paths from vertex a. To see this, consider in which order the vertices are dequeued from the priority queue and how vertex key values are updated as edges are relaxed:

Dequeued	**Updates**	**Priority queue (after relaxations)**
—	—	$\{(a:0), (b:\infty), (c:\infty), (d:\infty), (e:\infty), (f:\infty), (g:\infty), (h:\infty)\}$
a	$\{b, c, e\}$	$\{(c:2), (e:4), (b:6), (d:\infty), (f:\infty), (g:\infty), (h:\infty)\}$
c	$\{b\}$	$\{(e:4), (b:5), (d:\infty), (f:\infty), (g:\infty), (h:\infty)\}$
e	$\{h\}$	$\{(b:5), (h:6), (d:\infty), (f:\infty), (g:\infty)\}$
b	$\{d\}$	$\{(h:6), (d:12), (f:\infty), (g:\infty)\}$
h	$\{g\}$	$\{(g:7), (d:12), (f:\infty)\}$
g	$\{f\}$	$\{(f:9), (d:12)\}$
f	$\{d\}$	$\{(d:10)\}$
d	—	$\{\}$

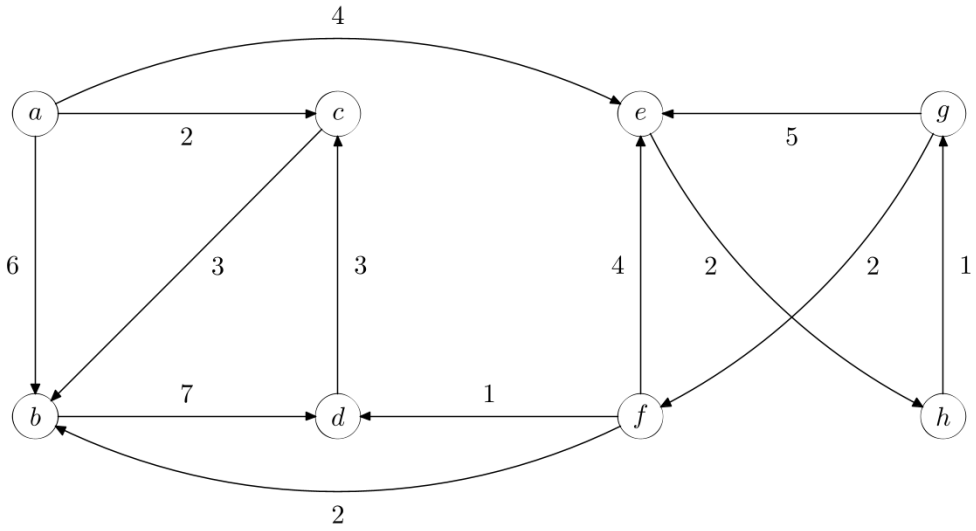


Figure 4: Graph for Problem 2

Sequence 3, finally, is a spoiler sequence that cannot be the output of any of the algorithms listed. To see this, consider the following case analysis:

- If the sequence were the result of breadth-first search, then it would list vertices in order of the number of directed edges required to get to the vertex from the start vertex a, but the first deviation from this is that g is listed before d.
- A depth-first search starting with a, c, b would discover d next and not e.
- Finally, the sequence cannot be the output of Dijkstra's algorithms, since the vertices are not listed in increasing order of distance from a. The first deviation here is that e should come before b in a listing in increasing order of distance.

3 (50 p) Back in 2021 during the pandemic, all classes and exams were held virtually, but the Copenhagen Tivoli Gardens amusement park was open for physical visits by the general public. This meant that students frustrated by Zoom lectures and *Digital eksamen* exams were able to head to the *bumper cars* (*radiobiler*) and let go of some steam by trying to bump into as many other cars (or as few cars) as possible.

Suppose that there are $n \geq 2$ bumper cars sharing an area of $m > n^2/2$ square meters.[1] An amazing mathematical fact is that after every bumper car round has finished, there are two different bumper cars that have bumped with exactly the same number of other (distinct) cars. Prove this!

Solution: We start by making three quick observations:

1. The area of course has nothing to do with this, and can just be ignored.

2. "Bumping" is a symmetric relation, in that bumper car A can only have a bump with bumper car B if car B also had a bump with car A. (This is clear from context, since otherwise the claim in the problem statement would be obviously false for two bumper cars.)

[1] In reality, there are 18 bumper cars in the Copenhagen Tivoli Gardens sharing an area of 280 square meters, but this is not too relevant for this problem.

3. Each one of the n bumper cars can bump with at most $n-1$ other cars. If one car bumps with all other cars, then there is no car without bumps, and if there is a car without bumps, then no car can bump with all other cars. (Here we are using the symmetry of the bumping relation.)

Now we can think of each car as being a pigeon (n of them) and each pigeonhole as being labelled by a number of bumps ($n-1$ possibilities, as discussed above, since we cannot have 0 bumps and $n-1$ bumps at the same time). Clearly, by the pigeonhole principle there have to be two bumper cars that have the same number of bumps.

4 (60 p) Another way in which Danish computer science students were able to entertain themselves during the lock-down was to play with legos (or maybe not, but we need a story for this problem, so let us assume they did). In order to make this kind of activity more relevant from the point of view of discrete mathematics, rumour has it that the students often considered different sets of lego pieces (like the one in Figure 5a) and investigated in how many different ways cuboids (rectangular parallelepipeds) can be built from such pieces.

Inspired by this, let us consider the lego pieces in Figure 5a, which as we observe all have different colours. In this particular problem we want to build cuboids that are three layers high. The pieces should be put together in the standard lego way, with the lego studs pointing upwards and being fitted into the holes of any pieces above. All pieces should be used. Constructions like the one in Figure 5b are not valid—this particular construction does have three layers, and the studs point upwards and are fitted into the holes above when possible, but the resulting geometric shape is not a cuboid.

Two cuboids are considered to be the same if they can be rotated so that they become similar. Thus, the constructions in Figures 5c and 5d are both valid, but are considered to be the same cuboid, since rotating Figure 5c yields Figure 5d.

How many *different* cuboids can you build with the lego pieces in Figure 5a satisfying the restrictions described above? Please make sure to explain clearly how you reason to reach your answer.

Solution: Let us start by computing how many cuboids can be built without considering rotational symmetry (so that, for the moment, we consider Figure 5c and Figure 5d to be different cuboids). Once we have this number, it is clear that we should divide by 2 to get the number of constructions as asked for in the problem statement. This is so since it is not possible to build a cuboid in accordance with the rules that is symmetric under rotation. Hence, every cuboid can be paired up with exactly one other different cuboid so that the two are obtained from each other by rotation.

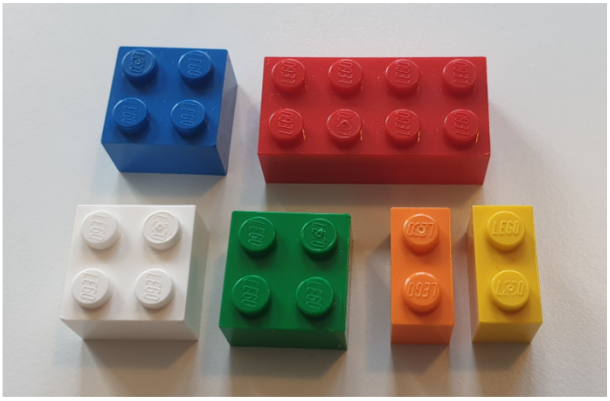
Counting the number of studs, there are:

- one 4×2 brick;
- three 2×2 bricks;
- two 2×1 bricks.

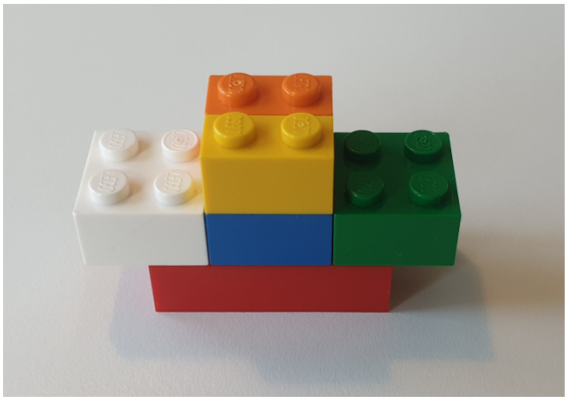
Let us analyse how the cuboids can be constructed.

Once the 4×2 brick is placed somewhere, all the other pieces have to be used in other layers, since otherwise we cannot get a cuboid. The layer with a single 4×2 brick can be chosen in 3 ways.

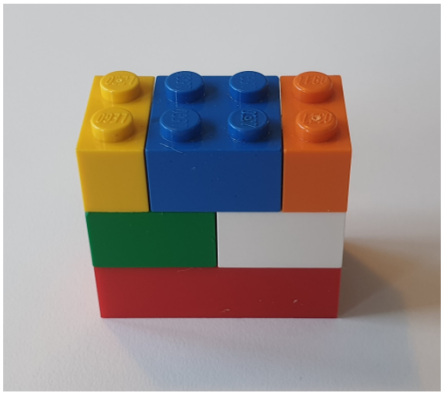
Then there will be a layer with two 2×2 bricks. Once we have decided on the 4×2 layer, the position for the layer with two 2×2 bricks can be chosen in 2 ways. Since there is a total of



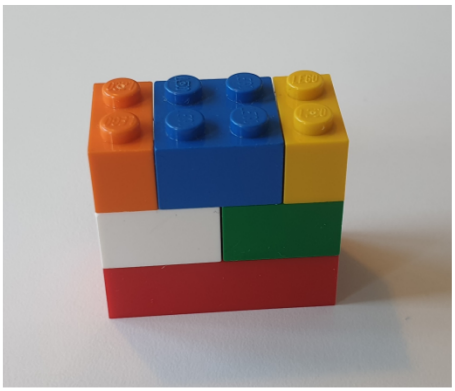
(a) A set of lego pieces.



(b) Invalid construction (not a cuboid).



(c) Valid construction of 3-layer cuboid.



(d) Symmetric version of the same cuboid.

Figure 5: Lego pieces and constructions (and non-constructions) of cuboids for Problem 4.

three 2×2 bricks, the left 2×2 brick can be chosen in 3 ways, after which the right 2×2 brick can be chosen in 2 ways.

So far, we have built two layers in our cuboid in a total of $3 \cdot (2 \cdot 3 \cdot 2) = 36$ ways.

For the third layer, we obtain a case analysis as follows, considering the lego bricks from left to right:

1. A 2×1 brick, then a 2×2 brick, and finally a 2×1 brick (as in Figure 5c and Figure 5d). Given the choices already made, there are 2 choices for this alternative, namely in which order the 2×1 bricks appear. (Note that there is only one 2×2 brick left by now, so there is no flexibility there.)

2. As in case 1 above, but with both 2×1 bricks appearing farthest to the left in the same orientation as in Figure 5c. There are again 2 choices for this alternative, depending on in which order the 2×1 bricks appear.

3. As in case 2, but both 2×1 bricks appearing rotated as in Figure 5b. This gives 2 more choices, depending on in which order the 2×1 bricks appear.

4. As in case 2, but with the 2×1 bricks appearing farthest to the right: 2 more choices.

5. Finally, as in case 3 but with the 2×1 bricks appearing farthest to the right: 2 more choices.

We see that we get a total of 10 ways of constructing the third layer, yielding a total number of 360 constructions. Since we do not distinguish between cuboids that are rotationally symmetric, however, we should divide by 2 to get the final answer that there are $360/2 = 180$ different cuboids that can be constructed.