

CS_232 Algorithms and Complexity

Week 09, Friday, 28/11/2008

Oliver Kullmann

Department of Computer Science
Swansea University
Swansea, SA2 8PP, UK

e-mail: O.Kullmann@Swansea.ac.uk

<http://cs.swan.ac.uk/~csoliver>

December 1, 2008

Additional lecture held on Friday, November 28, 2008, in the Keir Hardie Building, Room 303, from 12:00 - 13:00.

I Solutions for Coursework I

I Graph isomorphisms

1) The meaning

Explain in your own words what it means for two graphs to be isomorphic, and what additionally has to be observed for general graphs.

Consider two general graphs G_1, G_2 . Roughly speaking:

- G_1, G_2 are isomorphic iff we can rename the vertices and edges of G_1 such that the renamed graph actually is equal to G_2 .
- Visualising graph isomorphism, we draw G_1 and G_2 using “anonymous” vertices and edges, and then G_1 is isomorphic to G_2 iff we can move the edges and vertices of G_1 so that the drawing of G_1 becomes the same as the drawing of G_2 .

Using a bit more precision: G_1 is isomorphic to G_2 iff there is a bijection α from the set $V(G_1)$ of vertices of G_1 to the set $V(G_2)$ of vertices of G_2 and a bijection

β from the set $E(G_1)$ of edges of G_1 to the set $E(G_2)$ of edges of G_2 such that

for every edge $e \in E(G_1)$ connecting vertices $v, w \in V(G_1)$ in G_1 it connects $\beta(e)$ the vertices $\alpha(v), \alpha(w) \in V(G_2)$ in G_2 .

If we just consider graphs, then we don't need to rename the edges (they don't have names!), so an isomorphism between graphs (possibly with loops) is just given by a bijection of the vertex sets which transports the edge set of the one graph into the edge set of the other graph.

Further background: Consider two finite graphs G_1, G_2 . We want to decide whether G_1 is isomorphic to G_2 or not.

If $|V(G_1)| \neq |V(G_2)|$ or $|E(G_1)| \neq |E(G_2)|$ then G_1, G_2 are not isomorphic. So assume $|V(G_1)| = |V(G_2)|$ and $|E(G_1)| = |E(G_2)|$ in the sequel.

To make our life a bit easier, we can assume that $V(G_1) = V(G_2) = \{1, \dots, n\}$ for $n := |V(G_1)|$ (i.e., using indices for the vertices of G_1 and G_2).

Now we enumerate all permutations $\pi \in S(\{1, \dots, n\})$, that is all bijections π from $\{1, \dots, n\}$ into itself, and for each π we check

Is for every edge $\{v, w\} \in E(G_1)$ the edge $\{\pi(v), \pi(w)\} \in E(G_2)$?

If one π has this property, then π is an isomorphism from G_1 to G_2 (and thus G_1 and G_2 are isomorphic), while if none of the permutations $\pi \in S(\{1, \dots, n\})$ has this property, then G_1 and G_2 are not isomorphic.

(Note that due to $|E(G_1)| = |E(G_2)|$ we do not need to check whether all edges of G_2 are actually of the form $\{\pi(v), \pi(w)\}$ for edges $\{v, w\} \in E(G_1)$, since this is now automatically guaranteed (using the fact that π is a bijection).)

Let's consider the example $n = 3$. We have $3! = 6$ permutations here, namely 123, 132, 213, 231, 312, 321. The permutation 231 for example denotes the bijection $\pi : \{1, 2, 3\} \rightarrow \{1, 2, 3\}$ given by $\pi(1) = 2$, $\pi(2) = 3$, $\pi(3) = 1$. Given for example this bijection, we check whether it is actually an isomorphism from G_1 to G_2 by running through all edges $\{u, v\} \in E(G_1)$ and checking

whether the edge $\{\pi(u), \pi(v)\}$ is an edge of G_2 (for example, if $\{1, 2\} \in E(G_1)$, then we check whether $\{2, 3\} \in E(G_2)$) — if this is the case for all edges $\{u, v\} \in E(G_1)$, then π actually is an isomorphism of G_1 and G_2 , otherwise not.

Enumerating all elements of S_n (the bijections from $\{1, \dots, n\}$ into itself) can be done “lazily” by using a simple recursive approach, or it can be done in a more complicated but also more (space-) efficient iterative way (for example running through all permutations in lexicographical order as in the above example).

2) A general decision procedure

A powerful general technique for hard problems is backtracking. Explain in a paragraph the general idea (of course, cite your sources!). Describe how to decide whether two graphs are isomorphic via backtracking. State the time complexity of your algorithm using the Big Oh notation.

Given some decision problem (that is, only “yes” and “no” answers are sought), where we are looking for a “solution” which has the form of an assignment of certain values to some “variables”, and where we can evaluate already *partial* assignments, the general **backtracking** approach is as follows:

- (a) The main function $f(P)$ calls the recursive function $f'(P, \varphi)$ with the empty assignment $\varphi := \emptyset$.
- (b) The recursive function $f'(P, \varphi)$ determines whether the given partial assignment φ can be extended to a solution of problem P .
- (c) $f'(P, \varphi)$ works as follows:

- (a) If φ is a solution for P then output “yes” (and perhaps φ), and the whole process is finished.
- (b) If on the other hand we recognise P together with φ as unsolvable, then return “no” and backtrack (recursively) to the previous level.
- (c) Choose a (still) open variable v and sort the possible values $\varepsilon_1, \dots, \varepsilon_k$ of v . Run through the values $\varepsilon_1, \dots, \varepsilon_k$, extending φ by the assignment $v \rightarrow \varepsilon_i$, obtaining φ_i , and applying $f'(P, \varphi_i)$.
- (d) When backtracking, we always try the next not yet chosen value ε_i . If none is left, then we return “no” and backtrack (recursively) to the previous level.

In the case at hand, the “graph isomorphism problem”, where for two graphs G_1, G_2 we need to decide whether G_1 is isomorphic to G_2 , the “variables” are the vertices of G_1 , while the “values” are the vertices of G_2 .

More specifically, we can proceed as follows: The recursive algorithm takes an injection j with domain and image contained in $\{1, \dots, n\}$ as input (for the first call, the empty mapping is supplied). If the domain of j is all of $\{1, \dots, n\}$, then j actually is a

bijection from $\{1, \dots, n\}$ to $\{1, \dots, n\}$, and we check whether j is a graph isomorphism — if it is, we found G_1 and G_2 to be isomorphic, and we can exit the algorithm, while otherwise the algorithm backtracks. If the domain of j is not all of $\{1, \dots, n\}$, then a vertex $v \in \{1, \dots, n\}$ not in the domain of j is chosen, and all possible images $j(v)$ not in the current image of j are tried out as extensions of j .

Reminder: That j is an *injection* means, that for all vertices v, w in the domain of j (that is, for which j is actually defined) with $v \neq w$ we have $j(v) \neq j(w)$. In other words, j is injective iff different vertices get mapped to different vertices.

Pseudocode (in C/C++ style) is as follows, where we use $\text{dom}(j) \subseteq \{1, \dots, n\}$ for the *domain* of j (the set of arguments for which j is defined), and $\text{rg}(j)$ for the *range* of j , the set $\{j(v) : v \in \text{dom}(j)\}$ of values of j . Furthermore we use INJ for a type, whose values are injections j where both $\text{dom}(j)$ and $\text{rg}(j)$ are subsets of $\{1, \dots, n\}$. To simplify the code, the inputs G_1 and G_2 are global variables as well as n (recall that we already checked that both input graphs have the same number of vertices and edges).

```
bool isomorphism_extension(INJ j) {
  if (dom(j) = {1, ..., n}) {
    for all {v, w} edge of G1 {
      if (not {j(v), j(w)} edge of G2)
        return false;
    }
    return true;
  }
  else {
    choose v in {1, ..., n} minus dom(j);
    for all v' in {1, ..., n} minus rg(j) {
      j' = j plus (v maps to v');
      if (isomorphism_extension(j'))
        return true;
    }
    return false;
  }
}
```

G1 and G2 are isomorphic iff

`isomorphism_extension(\emptyset)`

returns true, where \emptyset is the empty map.

The “main complexity” of this algorithm is $\tilde{O}(n!)$, ignoring some polynomial factor for carrying out computation of the permutations and checking whether we found an isomorphism.

Actually, the operations per “round” (per new permutation π) can be done in quadratic time in the size $s := \|G_1\| + \|G_2\|$, and thus the running time of the algorithm is

$$O(n! \cdot s^2).$$

(Estimating here n with s , giving the upper bound $O(n! \cdot s^2) \leq O(s! \cdot s^2)$, is typically not done: estimating a linear factor n by s might be considered as acceptable, since linear functions don’t grow fast, but estimating n in a fast-growing function like $n!$ by s would make a huge difference. Thus big terms like 2^n or $n!$ use some refined parameters like the number of vertices if possible; on the other hand, in such a context, where we have exponential growth, for the polynomial factors often one doesn’t care so much. In fact sometimes notations like “ \tilde{O} ” are used to abstract away from the polynomial part altogether. Of course, such considerations are only applied to algorithms with exponential behaviour. For algorithms with linear or quadratic running time one takes a much closer look.)

Further refinements: Adding further “consistency checks” is **essential** for acceptable performance. For example if $j(v)$ has not the same degree as v then we do not need to consider $j(v)$.

The main advantage of a backtracking approach over the brute force approach (running through all permutations) is only realised if we do not need to wait until j is “full” (has all of $V(G_1)$ as its domain), but we can detect earlier that j cannot be extended to an isomorphism. So, as outlined in our general approach at the beginning, we need to evaluate *partial assignments*.

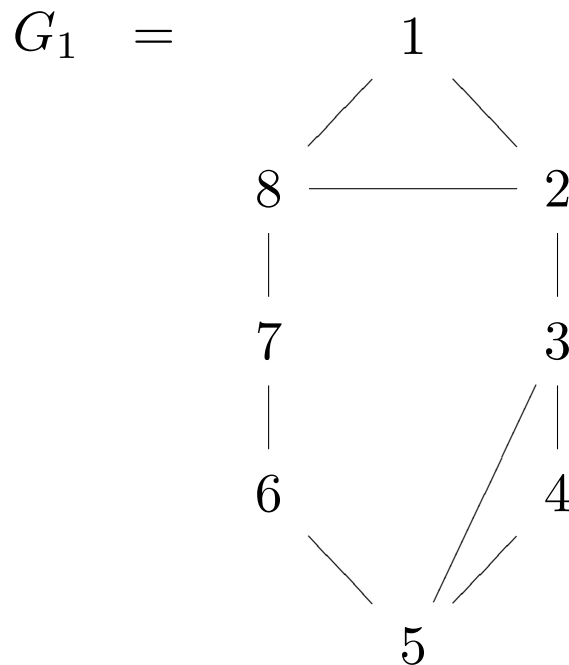
The first step thus to exploit the potential savings of the backtracking approach is to check for every edge $\{v, w\} \in E(G_1)$ as soon as both v, w are in the domain of j , whether $\{j(v), j(w)\} \in E(G_2)$ — if this is not the case, then already at this point we can backtrack!

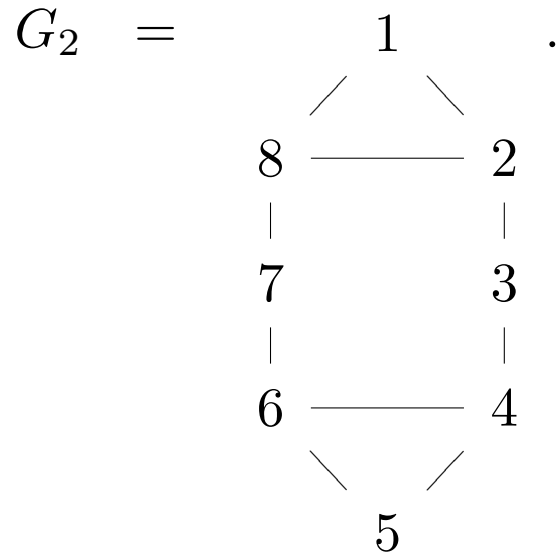
So if v is the newly assigned vertex, then we check exactly all edges $\{v, w\} \in E(G_1)$ for which w is already in the domain of j (in this way we also avoid double-checking).

3) Deciding isomorphism

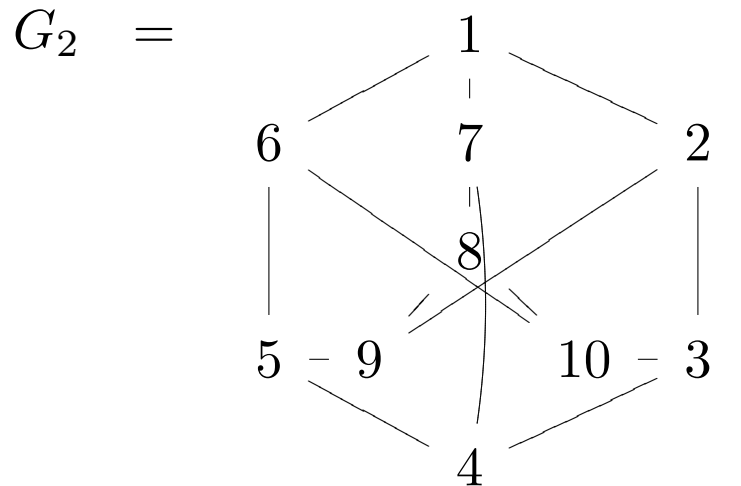
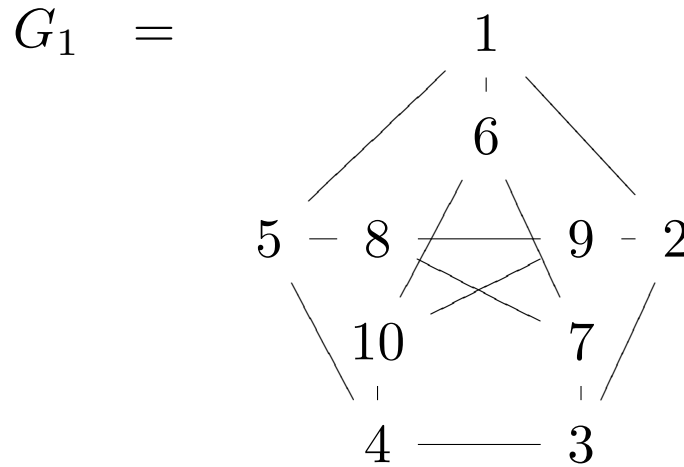
For each of the following pairs of graphs, decide whether G_1 is isomorphic to G_2 or not; in case they are isomorphic show how to rename the vertices of one of them to get the other graph, while in case they are not isomorphic give a criterion that separates them, and argue, that this criterion is invariant under isomorphisms (that is, can be used to show that two graphs are non-isomorphic).

(a)





(b)



Below you find *possible answers* (in general there are many possibilities here).

Part a) The number of vertices and edges, and also the (sorted) degree sequences of both graphs are the same, so that doesn't help us here. Furthermore the general structure is also the same: Both graphs are cycles of length 8 with two additional "chords", and those two chords create two triangles.

The natural conjecture should be that these two graphs are not isomorphic, since the relative position of the two triangles seems to be "essentially" different. What is needed is a "good" criterion to distinguish the two graphs (a criterion which is invariant under isomorphisms, that is, which doesn't depend on the specific way we draw the graphs, or on the specific naming conventions).

A sensible criterion is to take the minimal distance between the two triangles (note that both graphs have exactly two triangles), which is 1 for the first graph and 2 for the second graph.

Part b)

Both graphs are “cubic”, that is, each vertex has degree 3, so here vertex degrees again don’t help.

A feature of G_1 is the cycle of length 5 given by the vertex sequence $C_1 := (1, 2, 3, 4, 5)$. In G_2 we find the 5-cycle $C'_1 := (7, 8, 9, 5, 4)$. In G_1 we have another 5-cycle, disjoint to C_1 , namely $C_2 := (6, 7, 8, 9, 10)$. And similarly, in G_2 we have another 5-cycle, disjoint to C'_1 , namely $C'_2 := (1, 2, 3, 10, 6)$. So perhaps both graphs are isomorphic? Let’s try to construct an isomorphism π from G_1 to G_2 .

Let’s try to map the cycle C_1 to the cycle C'_1 , that is

$$\pi_1 : 1 \mapsto 7, 2 \mapsto 8, 3 \mapsto 9, 4 \mapsto 5, 5 \mapsto 4.$$

Yet it seems fine, since π_1 is an isomorphism from the induced subgraph of G_1 given by the vertex set $\{1, 2, 3, 4, 5\}$ to the induced subgraph of G_2 given by the vertex set $\{7, 8, 9, 4, 5\}$.

We can try the same with the other 5-cycles we found:

$$\pi_2 : 6 \mapsto 1, 7 \mapsto 2, 8 \mapsto 3, 9 \mapsto 10, 10 \mapsto 6.$$

Again, π_2 is an isomorphism from the induced subgraph of G_1 given by the vertex set $\{6, 7, 8, 9, 10\}$ to the induced subgraph of G_2 given by the vertex set $\{1, 2, 3, 6, 10\}$.

Since the domains and the ranges of π_1, π_2 are disjoint, we can consider the union, and obtain a bijection π from $V(G_1)$ to $V(G_2)$ — is it also an isomorphism? Let's probe some vertices:

- (a) Vertex $1 \in V(G_1)$ is connected to $2, 5, 6 \in V(G_1)$, while vertex $\pi(1) = 7 \in V(G_2)$ is connected to vertices $1, 4, 8 \in V(G_2)$, where $\pi(2) = 8, \pi(5) = 4, \pi(6) = 1$, so that's alright.
- (b) Vertex $7 \in V(G_1)$ is connected to $3, 6, 8 \in V(G_1)$, while vertex $\pi(7) = 2 \in V(G_2)$ is connected to vertices $1, 3, 9 \in V(G_2)$, where $\pi(3) = 9, \pi(6) = 1, \pi(8) = 3$, and again that's alright.
- (c) Vertex $3 \in V(G_1)$ is adjacent to $2, 4, 7 \in V(G_1)$, while $\pi(3) = 9 \in V(G_2)$ is adjacent to $2, 5, 8 \in V(G_2)$, where $\pi(2) = 8, \pi(4) = 5, \pi(7) = 2$. \checkmark
- (d) Vertex $8 \in V(G_1)$ is adjacent to $5, 7, 9 \in V(G_1)$, while $\pi(8) = 3 \in V(G_2)$ is adjacent to $2, 4, 10 \in V(G_2)$, where $\pi(5) = 4, \pi(7) = 2, \pi(9) = 10$. \checkmark

Please check the other six vertices yourself — we actually have obtained an isomorphism from G_1 to G_2 , and thus both graphs are isomorphic.

By the way, here are the adjacency matrices $A(G_1), A(G_2)$ of G_1, G_2 :

$$A(G_1) = \begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \end{pmatrix}$$

$$A(G_2) = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \end{pmatrix}$$

Permuting rows and columns according to π , we obtain $A(G_2)$ from $A(G_1)$.

4) Isomorphism types

Draw all non-isomorphic connected graphs for $n = 0, 1, 2, 3$ vertices. Draw all non-isomorphic trees with $n = 4$ vertices.

First the isomorphism types of connected graphs with at most three vertices:

$n = 0$ Exactly one graph, the graph $K_0 = (\emptyset, \emptyset)$ without vertices.

$n = 1$ Exactly one isomorphism type, the graph $K_1 = (\{1\}, \emptyset)$ with one vertex (and no edges).

$n = 2$ Exactly one isomorphism type, the graph $K_2 = (\{1, 2\}, \{\{1, 2\}\})$ with two vertices and one edge.

$n = 3$ Exactly two isomorphism types, the complete graph K_3 and the path P^2 .

Note that a connected graph with $n \geq 1$ vertices has at least $n - 1$ edges.

Finally, there are exactly two non-isomorphic trees with four vertices, the path P^3 and the triclave $K_{1,3}$.

II graph_traversal validated

Give arguments as good as possible for the following central properties of graph_traversal, where we assume that the input G is connected (argue directly with the pseudo-code given for graph_traversal):

1. *There are exactly $|V(G)| - 1$ tree edges and $|E(G)| - |V(G)| + 1$ back edges.*
2. *When we consider the situation exactly before the new vertex is marked as visited, then the buffer contains exactly two types of edges:
(i) edges connecting vertices already discovered with vertices not yet discovered;
(ii) edges connecting two already discovered vertices.
Regarding type (i), show that the buffer must contain all such “frontier edges.” And regarding type (ii), show that all these edges have been identified already as back-edges and will be dropped from the buffer while searching for a new tree edge.*
3. *Loops are never entered into the buffer but are immediately recognised as back-edges.*

Solution: For Part 1 the basic argumentation is that at the beginning we have one vertex and zero edges in the tree built by `graph_traversal`, and that every new tree edge adds exactly one new (not yet visited) vertex. Since every vertex is visited (due to the assumption that G is connected), we thus find exactly $|V(G)| - 1$ tree edges. Furthermore, since every edge is either a tree edge or a back edge (never both), if we subtract the number of tree edges from the total number of edges we get the number of back edges, which is $|E(G)| - (|V(G)| - 1) = |E(G)| - |V(G)| + 1$.

This argumentation can be refined, discussing in more details the underlying claims. That a graph edge cannot be both a back edge and a tree edge follows from the fact that at the time when a back edge e is recognised (scanning its source v), it must have already been pushed on the buffer before (since we visited its target before), and actually e must still be in the buffer, since if it had been taken out, then it would have been a tree edge (note that at that time the source v must have been unvisited!), and thus it would have been a parent edge leading to v , which is not eligible for recognition as back edges.

We need also to show that actually every edge e of G will be recognised as back edge or tree edge. The first observation is that if

XXX

1) A vertex v is “visited” at the time when v is marked and the visitor action for a new vertex is called (in other words, “visiting” happens directly after the Start label). Now the main observations here are as follows:

1. No vertex is visited which has already been marked as visited (for the start vertex the first instruction of `graph_traversal` is responsible to ensure this, while for new vertices it is the loop condition of the `repeat-until-loop` selecting a new tree edge).
2. Once we visit a vertex, we mark it as visited.
3. A vertex once marked never gets unmarked.

2) There is exactly one place where we push an edge e on the buffer, namely in the `for-loop` where we

run through all edges incident to the vertex currently visited. Since every vertex is visited only once, there are at most two chances for e to be put on the buffer, namely when its two endpoints v, w are visited. One endpoint, say v , will be first (thus w has not been visited yet), then e is pushed on the buffer, and v will be marked as visited, so that when w is visited, the target of the edge e (which now is v (!)) has already been visited. And we don't push an edge on the buffer if its target has already been visited.

3) Here first one has to point out, that if an edge e is pushed on the buffer, then that very *occurrence* of e cannot be sent to the visitor as back edge, but it is later, when we visit the second endpoint w of e , that we possibly can send e to the visitor as back edge: If we reach w via "parent edge" e , then e has been selected as tree edge, and won't be considered for possible back edges, but if we reach w via an edge different from e , then e will be considered as back edge and send to the visitor.

So the only possible problem left is, that when

selecting edges from the buffer for the next tree edge, that then e is just discarded, but not selected as tree edge. This happens iff the target w of e has already been visited, and thus, at the time when w has been visited, it must have been the case that e has been recognised as back edge before (while e can't have been a tree edge, since otherwise it would have been removed from the buffer).

4) Since we can reach new nodes only via edges from visited vertices, by induction one can show that every vertex visited is in the connected component of the start vertex r . The critical thing to show is, that we don't miss a vertex, that is, that every vertex in the connected component of r finally gets visited. To prove this, let's assume that there is a vertex w in the connected component of r not visited. There is a path P from r to w ; let w_0 be the first vertex on this path not being visited. We have $w_0 \neq r$, since we definitely visited r , and thus there is the immediate predecessor v of w_0 in P (where v has been visited). Every edge incident to v and leading to a (then) unvisited vertex has been pushed on the buffer, and thus the edge e

from v to w_0 was pushed on the buffer. As we have seen in (3), e finally gets recognised as back edge or tree edge. If e would have been recognised as back edge, then w_0 had been visited, and so e must have been a tree edge — and thus w_0 would have been visited contradicting the assumption.

5) Consider an edge $e = \{v, w\}$ in the connected component of the start vertex r . By (4) both v and w have been visited; w.l.o.g. v was the first of them. Since then w was unmarked, e was pushed on the buffer.

6) (In Parts 1-5 we didn't make use of the assumption, that G is connected; now we do.) Let t be the number of tree edges, and let b be the number of back edges. By 3) it is $t + b$ equal to the number of edges pushed on the buffer (that is, altogether). By (5) and (2) the number of edges pushed on the buffer is $|E(G)|$, and thus $t + b = |E(G)|$. With every tree edge we visit a new vertex, and thus $t = |V(G)| - 1$ by 4), from which $b = |E(G)| - t = |E(G)| - |V(G)| + 1$ follows.

III Spanning trees

1) All spanning trees

Compute all spanning trees for

$$G = \begin{array}{ccccc} 1 & \text{---} & 2 & \text{---} & 3 \\ | & & | & & | \\ 4 & \text{---} & 5 & \text{---} & 6 \end{array} .$$

(Justify that you got really all spanning trees.)

G is connected (necessary for having a spanning tree) with 6 vertices, and so every spanning tree of G has 5 edges. G has exactly 3 circuits, and since spanning trees do not contain circuits, these circuits have to be “broken” (i.e., one edge has to be removed from each of them).

The elementary circuits (not composable from other circuits) are 1, 2, 5, 4 and 2, 3, 6, 5. Their overlap is 2, 5. Two edges have to be removed: Either we remove one edge from each of the elementary circuits without the overlap, or we remove the overlapping edge, obtaining a graph with exactly one circuit, and then any (single) edge can be removed.

Thus G has exactly $3 \cdot 3 + 6 = 15$ spanning trees.

2) Prim's algorithm

Explain Prim's algorithm for computing MST's in your own words.

The basic idea behind Prim's algorithm for computing MST's is:

- When growing a spanning tree, then all edges connecting some vertex of the current partial spanning tree to some vertex not in the spanning tree (i.e., all potential tree edges) must be in the buffer.
- Now when choosing a new tree edge of minimal weight, then this choice actually is optimal.
- Thus by repeating the process we obtain finally a spanning tree of minimal weight for the connected component of the start vertex.

Rephrased in terms of our generic `graph_traversal` procedure:

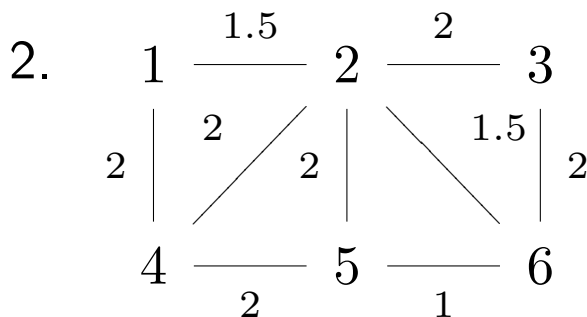
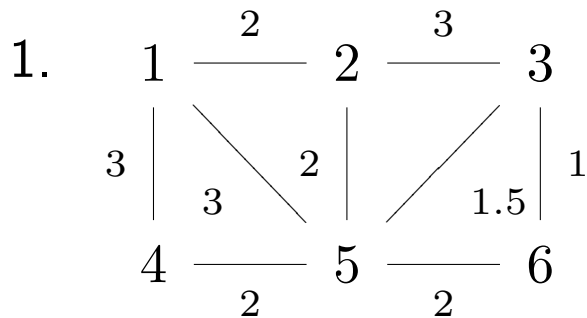
Using `graph_traversal` with a buffer which always returns an edge of minimal weight (among all edges in the buffer) yields an MST for the connected component of the start vertex. In other words:

`graph_traversal` equipped with a minimum priority queue (considering edge weights) computes MST's.

Prim's algorithm is an example, where a greedy strategy is successful (grabbing always the "best" next available edge).

3) Applying Prim's algorithm

Compute MST's for the following weighted graphs, showing the steps of the computations:



In the script 05a a graphical representation of the different states of the (non-deterministic) algorithm is given. Below I use an alternative textual representation.

What is the relevant data to be shown?

- With the processing of `graph_traversal`, one result we are building up (in this context actually the only relevant outcome) is the *spanning tree*, so for each round of the computation we must show the current state.
- The spanning tree represents the (valuable) information we have already extracted from the *buffer*, while its current content represents the “frontier”, and so we must show it.
- Regarding the actions during each round, the natural investigation points are after the first loop (processing the edges incident to the vertices currently visited) and after the second loop (pulling a new edge from the buffer).
- For the very first round, since there is no previous round (as there is no “parent”), the initial state is to be shown — since everything is empty yet, this boils down to the choice of the start vertex.

Let’s process the first example, using v for the current vertex, (e, k) for the current discovery edge (“tree edge”) with its key, T for the current spanning tree, and B for the current content of the buffer. XXX

1. Initialisation: $v = 1$, $e = \emptyset$, $T = (\{1\}, \emptyset)$, B empty.

(a) After processing v : $B = \{(\{1, 2\}, 2), (\{1, 4\}, 3), (\{1, 5\}, 3)\}$.

(b) After processing B : The set of choices for e is $\{\{1, 2\}\}$. So $e = \{1, 2\}$. Now

i. $T = (\{1, 2\}, \{\{1, 2\}\})$

ii. $B = \{(\{1, 4\}, 3), (\{1, 5\}, 3)\}$.

2. $v = 2$

(a) After v : $B = \{(\{1, 4\}, 3), (\{1, 5\}, 3), (\{2, 3\}, 3), (\{2, 5\}, 2), (\{2, 6\}, 3)\}$.

(b) After B : The set of choices for e is $\{\{2, 5\}\}$. So $e = \{2, 5\}$. Now

i. $T = (\{1, 2, 5\}, \{\{1, 2\}, \{2, 5\}\})$,

ii. $B = \{(\{1, 4\}, 3), (\{1, 5\}, 3), (\{2, 3\}, 3), (\{2, 6\}, 3)\}$.

3. $v = 5$

(a) After v : $B = \{(\{1, 4\}, 3), (\{1, 5\}, 3), (\{2, 3\}, 3), (\{2, 6\}, 3), (\{5, 4\}, 2), (\{5, 6\}, 2)\}$.

(b) After B : The set of choices for e is $\{\{5, 4\}, \{5, 6\}\}$. Let's choose $e = \{5, 4\}$. Now

i. $T = (\{1, 2, 5, 4\}, \{\{1, 2\}, \{2, 5\}, \{5, 4\}\})$

ii. $B = \{(\{1, 4\}, 3), (\{1, 5\}, 3), (\{2, 3\}, 3), (\{2, 6\}, 3), (\{5, 6\}, 2)\}$.

4. $v = 4$

(a) After v : Back edge $\{4, 1\}$ discarded. $B = \{(\{1, 4\}, 3), (\{1, 5\}, 3), (\{2, 3\}, 3), (\{2, 6\}, 3), (\{5, 6\}, 2)\}$.

(b) After B : The set of choices for e is $\{\{5, 6\}\}$. So $e = \{5, 6\}$. Now

i. $T = (\{1, 2, 5, 4, 6\}, \{\{1, 2\}, \{2, 5\}, \{5, 4\}, \{5, 6\}\})$

ii. $B = \{(\{1, 4\}, 3), (\{1, 5\}, 3), (\{2, 3\}, 3), (\{2, 6\}, 3)\}$.

5. $v = 6$

(a) After v : Back edge $\{6, 5\}$ discarded. $B = \{(\{1, 4\}, 3), (\{1, 5\}, 3), (\{2, 3\}, 3), (\{2, 6\}, 3), (\{6, 3\}, 1)\}$.

(b) After B : The set of choices for e is $\{\{6, 3\}\}$. So $e = \{6, 3\}$. Now

i. $T = (\{1, 2, 5, 4, 6, 3\}, \{\{1, 2\}, \{2, 5\}, \{5, 4\}, \{5, 6\}, \{6, 3\}\})$

ii. $B = \{(\{1, 4\}, 3), (\{1, 5\}, 3), (\{2, 3\}, 3), (\{2, 6\}, 3)\}$.

6. $v = 3$

(a) After v : Back edge $\{3, 2\}$ discarded. $B = \{(\{1, 4\}, 3), (\{1, 5\}, 3), (\{2, 3\}, 3), (\{2, 6\}, 3)\}$.

(b) After B : All edges discarded. Buffer empty, thus termination.

1. Initialisation: $v = 1$, $e = 0$, $T = (\{1\}, \emptyset)$, B empty.

(a) After v : $B = \{(\{1, 2\}, 3), (\{1, 4\}, 2)\}$.

(b) After B : The set of choices for e is $\{\{1, 4\}\}$. So $e = \{1, 4\}$. Now

i. $T = (\{1, 4\}, \{\{1, 4\}\})$

ii. $B = \{(\{1, 2\}, 3)\}$.

2. $v = 4$

(a) After v : $B = \{(\{1, 2\}, 3), (\{4, 2\}, 2), (\{4, 5\}, 2)\}$.

(b) After B : The set of choices for e is $\{\{4, 2\}, \{4, 5\}\}$. Choose $e = \{4, 2\}$. Now

i. $T = (\{1, 4, 2\}, \{\{1, 4\}, \{4, 2\}\})$

ii. $B = \{(\{1, 2\}, 3), (\{4, 5\}, 2)\}$.

3. $v = 2$

(a) After v : $B = \{(\{1, 2\}, 3), (\{4, 5\}, 2), (\{2, 3\}, 2), (\{2, 5\}, 2), (\{2, 6\}, 3)\}$. Edge $\{2, 1\}$ was detected as back edge.

(b) After B : The set of choices for e is $\{\{4, 5\}, \{2, 3\}, \{2, 5\}\}$. Choose $e = \{2, 3\}$. Now

i. $T = (\{1, 4, 2, 3\}, \{\{1, 4\}, \{4, 2\}, \{2, 3\}\})$

ii. $B = \{(\{1, 2\}, 3), (\{4, 5\}, 2), (\{2, 5\}, 2), (\{2, 6\}, 3)\}$.

4. $v = 3$

(a) After v : $B = \{(\{1, 2\}, 3), (\{4, 5\}, 2), (\{2, 5\}, 2),$

$(\{2, 6\}, 3), (\{3, 6\}, 2)\}$.

(b) After B : The set of choices for e is $\{\{4, 5\}, \{2, 5\}, \{3, 6\}\}$. Choose $e = \{4, 5\}$. Now

i. $T = (\{1, 4, 2, 3, 5\}, \{\{1, 4\}, \{4, 2\}, \{2, 3\}, \{4, 5\}\})$

ii. $B = \{(\{1, 2\}, 3), (\{2, 5\}, 2), (\{2, 6\}, 3), (\{3, 6\}, 2)\}$.

5. $v = 5$

(a) After v : $B = \{(\{1, 2\}, 3), (\{2, 5\}, 2), (\{2, 6\}, 3), (\{3, 6\}, 2), (\{5, 6\}, 1)\}$. Edge $\{5, 2\}$ was discovered as back edge.

(b) After B : The set of choices for e is $\{\{5, 6\}\}$. So $e = \{5, 6\}$. Now

i. $T = (\{1, 4, 2, 3, 5, 6\}, \{\{1, 4\}, \{4, 2\}, \{2, 3\}, \{4, 5\}, \{5, 6\}\})$

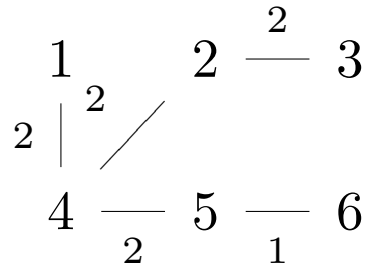
ii. $B = \{(\{1, 2\}, 3), (\{2, 5\}, 2), (\{2, 6\}, 3), (\{3, 6\}, 2)\}$.

6. $v = 6$

(a) After v : $B = \{(\{1, 2\}, 3), (\{2, 5\}, 2), (\{2, 6\}, 3), (\{3, 6\}, 2)\}$. Edges $\{6, 2\}, \{6, 3\}$ discovered as back edges.

(b) After B : All edges discarded. Buffer empty, thus termination.

Again, cleaning up the buffer only happened at the end here. The obtained MST is



which has an (optimal) weight of 9.

4) Dijkstra's algorithm

Explain Dijkstra's algorithm for computing SPT's in your own words.

Dijkstra's algorithm can be explained as follows:

We use an additional data structure

$$d : V(G) \rightarrow \mathbb{R}_{\geq 0} \cup \{+\infty\},$$

which assigns to every vertex v the length $d(v)$ of a shortest path found until now. At the beginning, $d(r) = 0$, where r is the start vertex, while $d(v) = +\infty$ for all other vertices $v \neq r$ (since until now no path has been discovered to v).

Now when exploring a new vertex v , that is, when running through all edges e incident to v and either notifying the visitor, that e is a back edge, or otherwise adding e to the buffer, then we update d in case a shorter path to the target t of e has been found, that is, in case of

$$d(v) + w(e) < d(t)$$

we set $d(t) := d(v) + w(e)$.

Dijkstra's algorithm is another example for a (successful) greedy algorithm:

- The next edge e taken from the buffer is one where the distance $d(\text{source}(e)) + w(e)$ is minimal, that is, where the length of the shortest path discovered until now and using e to the target of e is minimal.
- In other words, the buffer is a minimum priority queue with respect to the labelling $d(\text{source}(e)) + w(e)$ of the edges.

An important point here is, that for edges e in the buffer the distance $d(\text{source}(e))$ *will not change once the edge is put in the buffer*; this is due to the fact, that the source of edges in the buffer are necessarily vertices of the spanning tree grown until now, and for vertices the spanning tree grown until now contains already a shortest path (this was proven in the script).

Thus the “key” of an element in the buffer, the number needed by the priority queue, so that it can

return an element with a minimal key, is fixed upon the entry of the element (the edge), which is necessary for the working of the priority queue.

Using such a priority queue, `graph_traversal` will always return a shortest path tree for the start vertex (containing shortest paths (w.r.t. the *whole graph*) from the root to any other vertex reachable from it).

5) Applying Dijkstra's algorithm

Compute SPT's for the first graph from Part 3) and all six choices for the root, showing the steps of the computations. Derive the distance matrix (for the whole graph, containing all pairwise distances).

We log the run of `graph_traversal` as before, but with the following modifications:

- Since now the root is important, we show the rooted tree T (as a pair of a tree and a vertex (the root)).
- Together with the spanning tree T we show the distance $d(v)$ to the root of each vertex $v \in V(T)$.
- The key of an edge $\{a, b\}$ entered into the buffer now is $d(a) + w(\{a, b\})$ (where $w(\{a, b\})$ is the weight of edge $\{a, b\}$).

Let's process root 1. XXX

1. Initialisation: $v = 1$, $e = 0$, $T = ((\{1\}, \emptyset), 1)$,
 $d(1) = 0$, B is empty.

(a) After v : $B = \{(\{1, 2\}, 2), (\{1, 4\}, 3), (\{1, 5\}, 3)\}$.

(b) After processing B : The set of choices for e is
 $\{\{1, 2\}\}$. So $e = \{1, 2\}$. Now

i. $T = ((\{1, 2\}, \{\{1, 2\}\}), 1)$

ii. $d(1) = 0$, $d(2) = 2$

iii. $B = \{(\{1, 4\}, 3), (\{1, 5\}, 3)\}$.

2. $v = 2$

(a) After v : $B = \{(\{1, 4\}, 3), (\{1, 5\}, 3), (\{2, 3\}, 5),$
 $(\{2, 4\}, 4), (\{2, 6\}, 5)\}$.

(b) After B : The set of choices for e is
 $\{\{1, 4\}, \{1, 5\}\}$. Let's choose $e = \{1, 4\}$. Now

i. $T = ((\{1, 2, 4\}, \{\{1, 2\}, \{1, 4\}\}), 1)$

ii. $d(1) = 0$, $d(2) = 2$, $d(4) = 3$

iii. $B = \{(\{1, 5\}, 3), (\{2, 3\}, 5), (\{2, 5\}, 4),$
 $(\{2, 6\}, 5)\}$.

3. $v = 4$

(a) After v : $B = \{(\{1, 5\}, 3), (\{2, 3\}, 5), (\{2, 5\}, 4),$
 $(\{2, 6\}, 5), (\{4, 5\}, 5)\}$.

(b) After B : The set of choices for e is $\{\{1, 5\}\}$. So
 $e = \{1, 5\}$. Now

- i. $T = ((\{1, 2, 4, 5\}, \{\{1, 2\}, \{1, 4\}, \{1, 5\}\}), 1)$
- ii. $d(1) = 0, d(2) = 2, d(4) = 3, d(5) = 3$
- iii. $B = \{(\{2, 3\}, 5), (\{2, 5\}, 4), (\{2, 6\}, 5), (\{4, 5\}, 5)\}.$

4. $v = 5$

(a) After v : $B = \{(\{2, 3\}, 5), (\{2, 5\}, 4), (\{2, 6\}, 5), (\{4, 5\}, 5), (\{5, 6\}, 5)\}.$ Back edges $\{5, 2\}, \{5, 4\}$ discarded.

(b) After B : Edges $\{2, 5\}, \{4, 5\}$ discarded, then the set of choices for e is $\{\{2, 3\}, \{2, 6\}, \{5, 6\}\}.$ Let's choose $e = \{5, 6\}.$ Now

- i. $T = ((\{1, 2, 4, 5, 6\}, \{\{1, 2\}, \{1, 4\}, \{1, 5\}, \{5, 6\}\}), 1)$
- ii. $d(1) = 0, d(2) = 2, d(4) = 3, d(5) = 3, d(6) = 5$
- iii. $B = \{(\{2, 3\}, 5), (\{2, 6\}, 5)\}.$

5. $v = 6$

(a) After v : $B = \{(\{2, 3\}, 5), (\{2, 6\}, 5), (\{6, 3\}, 6)\}.$ Back edge $\{6, 2\}$ discarded.

(b) After B : Edge $\{2, 6\}$ discarded, then the set of choices for e is $\{\{2, 3\}\}.$ So $e = \{2, 3\}.$ Now

- i. $T = ((\{1, 2, 4, 5, 6, 3\}, \{\{1, 2\}, \{1, 4\}, \{1, 5\}, \{5, 6\}, \{2, 3\}\}), 1)$

$$\text{ii. } d(1) = 0, \quad d(2) = 2, \quad d(4) = 3, \quad d(5) = 3, \\ d(6) = 5, \quad d(3) = 5$$

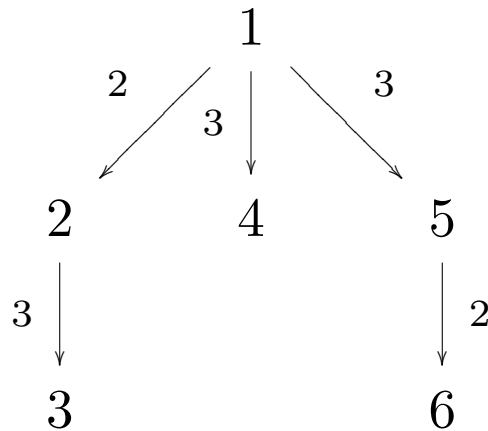
$$\text{iii. } B = \{(\{6, 3\}, 6)\}.$$

6. $v = 3$

(a) After v : $B = \{(\{6, 3\}, 6)\}$. Back edge $\{3, 6\}$ discarded.

(b) After B : All edges discarded. Buffer empty, thus termination.

The extracted SPT with root 1 is



Using the algorithm for all other 5 possible root vertices we obtain the distance matrix

$$\begin{pmatrix} 0 & 2 & 5 & 3 & 3 & 5 \\ 2 & 0 & 3 & 4 & 2 & 3 \\ 5 & 3 & 0 & 5 & 3 & 1 \\ 3 & 4 & 5 & 0 & 2 & 2 \\ 3 & 2 & 3 & 2 & 0 & 2 \\ 5 & 3 & 1 & 2 & 2 & 0 \end{pmatrix}.$$

6) About spanning trees

Assume edge e is part of every spanning tree for graph G — can you characterise such edges so that that we can identify them efficiently?

A useful notion here is the notion of a “bridge”: An edge in a graph is called a *bridge* if after its removal the two endpoints are no longer connected by some walk.

In other words, a bridge of a (general) graph G is an edge $e \in E(G)$ such that after removal of e (while keeping the endpoints) the new graph has more connected components than G .

Equivalently, e is a bridge of G iff e is not part of some circuit in G .

Assume now that G is connected (otherwise G wouldn't have a spanning tree). Thus $e \in E(G)$ is a bridge iff after removal of e the graph gets disconnected. Since a spanning tree T of G is a

connected subgraph of G with $V(G) = V(T)$, it follows $e \in E(T)$, that is, a bridge of G must occur in every spanning tree of G .

The main point now is that also the reverse holds: If some edge $e \in E(G)$ occurs in every spanning tree T of G then e is a bridge of G . To show this, let's assume that e wouldn't be a bridge. So $G' := (V(G), E(G) \setminus \{e\})$ is connected, and thus has a spanning tree T' , where we have $V(T') = V(G') = V(G)$, and thus T' is also a spanning tree of G — but $e \notin E(T')$ contradicting our assumption.

To summarise: An edge $e \in E(G)$ is a bridge of a connected graph G if and only if for every spanning tree T of G we have $e \in E(T)$.

IV Depth-first search

1) DFS

Describe a simplified algorithm for a “depth-first only” form of `graph_traversal`, not using a general buffer, but only a stack, and this only implicitly. Use a simple recursive approach (“to explore a vertex, visit recursively its neighbours”), which offers the same event points for the visitor as `graph_traversal` itself.

XXX

2) Discussion

Discuss the differences in traversing graphs between this recursive approach and `graph_traversal`.