ADS Report Water Pumping Route

Matthijs Muis

December 2023

Contents

1 Notation

The **natural numbers** are denoted as $\mathbb{N} = \{0, 1, ...\}$.

For an edge $e = (v, w, d) \in V \times V \times \mathbb{N}$ in a **directed, weighted multigraph**, we denote its **source node** $\alpha(e) = v$ and **target node** $\omega(e) = w$ and its **cost/travel time/weight** $c(e) = d$.

2 Overview

This report explains our algorithm for finding the optimal pump-value given a weighted, undirected multigraph (V, E) (for the definition of a weighted, undirected graph multigraph, see below) with a set of water pumps $W \subset V$ and a time limit $t \in \mathbb{N}$. The discussion is organized as follows:

- After formulating the problem, we will first discuss how to reduce the problem (V, W, E, t) to a new problem $(W \cup \{v_0\}, W, E', t)$ with a reduced graph $\overline{G} = (W \cup \{v_0\}, E')$ that has the same solution.
- Then, we describe a recursion relation that will lead us to a correct, top-down recursive implementation of the algorithm.
- This recursive implementation does a lot of double work, since many of the subproblems that are solved in recursive calls, show **overlap**. This motivates a dynamic programming approach, where we use a map to memoize already computed solutions for subproblems: it leads to a worst-case polynomial-time algorithm.
- Finally, we discuss an additional **branch-and-bound** mechanism: in each recursive step, we calculate an **upper bound** for the maximum possible water pumped if we continue down this strategy. In a global variable, we also keep the best **attained value**, and if the upper bound of our current partial solution is below this attained value, we can **prune** this partial solution, since it will never lead to a better solution than the one we already have.

The algorithm with Dynamic Programming + Branching and Bounding, achieved the best times on the visible 34 test cases, making this our final submission.

3 Problem Formulation

Formally, we are given a set of road intersections V and a set of pumping stations $W \subset V$. The set E consists of (bidirectional) roads which are represented by triples (v_i, v_j, d) with $v_i, v_j \in V$ and $d \in \mathbb{N}$ a natural number representing the number of minutes required to travel from road intersection v_i to v_j .

Moreover, we are given a natural number t representing the time in minutes that we are given to pump away as much water as possible. It takes $\Delta t = 10$ minutes to change the direction of the water at a pumping station. Each pumping station can pump 200 m^3 water out of the polder per minute. This means that if we decide to start reversing a water pump $w \in W$ at time t, this will result in an increase of P, - the total value of m^3s water pumped - by $(t - 10) \cdot 200\ m^3$.

Let G denote the graph given in the problem, n_v the number of road intersections, n_w the number of pumps, W the set of all pumps, and e the number of roads. With this notation introduced, we are ready to explain the algorithm.

3.1 Pumping Station Route, about reversing pumps immediately

We say that a *pumping station route* is a sequence of road intersections. It is clear that:

When we are, at some time-point t, at a water pump, then it is always better to start reversing this pump right at time t rather than to first travel somewhere else, then come back to reverse it.

We can formally *prove* this as follows: Suppose that we are at a pump w_0 and we can choose between:

(I) **Travel away** via some edge with cost d_1, then *optimally* start reversing the pumps $w_1, ..., w_k$ at times $t_1 \geq ... \geq t_k$, where $t_1 \leq t - d_1$. Some of the w_i may, w.l.o.g. be w_0 itself.

(II) **First turn on this pump**. On the one hand, this will pump $200 \cdot (t - 10)$ extra water out of the polder immediately. On the other hand, this gives us 10 fewer minutes to turn on the other pumps, meaning that we may fail to turn on w_k. But for the rest, the route along $w_1, ..., w_{k-1}$ would remain optimal, by optimality of the route $w_1, ..., w_k$ in the case of strategy (I). We would in the worst case be able to start reversing $w_1, ..., w_{k-1}$ at $t_1 - 10, ..., t_{k-1} - 10$.

Let P_i denote the water-pump value of strategy i. Then, strategy (I) will have $P_I = 200 \cdot \sum_{i=1}^{k}(t_i - 10)$, while strategy (II) will have $P_{II} = 200 \cdot (t - 10) + 200 \cdot \sum_{i=1}^{k}(t_i - 20)$, which equals $200 \cdot (t-10) + 200 \cdot \sum_{i=1}^{k-1}(t_i - 10) - 200 \cdot 10(k-1) = 200 \cdot (t - 10k) + 200 \cdot \sum_{i=1}^{k-1}(t_i - 10)$. Using the time inequality $t_k + k \cdot 10 + d_1 \leq t$ in strategy I, which **holds** since there are at least (ignoring potential intermediate travel times) $10k + 10$ minutes between starting to turn on w_0 at t and starting to turn on t_k, plus d_1 cost for traveling out of w_0, it now easily follows $(t - 10k) - d_1 \geq t_k$, therefore

$$\frac{P_I}{200} = \sum_{i=1}^{k}(t_i - 10) = (t_k - 10) + \sum_{i=1}^{k-1}(t_i - 10) \leq (t - 10k) + \sum_{i=1}^{k-1}(t_i - 10) = \frac{P_{II}}{200}$$

We conclude that **if we are at a pump $w \in W$ that has not been reversed, then the optimal way to proceed is to turn on this pump immediately before travelling anywhere else**. This justifies why we only look at *pumping station routes* rather than also specifying whether we turn on a pump if we pass it.

4 Explanation of the Algorithm

4.1 Graph Representation

Note that the above problem statement can be reformulated in terms of an undirected, weighted multigraph G, together with a subset W of its vertices, a starting node v_0 and a time limit t. A **weighted multigraph** is a tuple $G = (V, E)$ where $E \subset V \times V \times \mathbb{N}$ is the set of **triples** (v_i, v_j, d) denoting edges (v_i, v_j) and their travel costs d.

To represent such a graph, we use an **unordered map** (implemented as a hash table), called `graph`, which maps v_i to a list of (v_j, d)-pairs such that for each j, (v_i, v_j, d) or (v_j, v_i, d) is in E. Some of the test cases we have seen, indeed have multiple edges between the same nodes (v_i, v_j). This means that (v_i, v_j) is **not a key for** d, and this also motivates our choice **not to implement the graph as follows**:

- use an unordered map to map each v_j to its neighbours v_j.

- use another unordered map `cost` to map each pair (v_i, v_j) that has an edge to a d stored in `cost[(v_i,v_j)]` (this cannot be done, as (v_i, v_j) is not a key for d).

We *could* in principle use a separate `cost` dictionary, provided that we use it to store the **minimum** cost min $(v_i, v_j, d) \in Ed$ among roads starting or ending in v_i. But we will perform a graph reduction anyway (see 4.3), so we do not care *too much* about space complexity of the initial representation

There are two asides here: first, the above neighbour-list representation is at first glance meant for **directed graphs**, where we map a **source node** $\alpha(e)$ of a directed edge e to a **target node** $\omega(e)$ of that edge. In an undirected graph, we also want a fast lookup of all $\alpha(e)$ given $\omega(e)$. That is why we also store (v_i, d) in the list `graph[vj]`, whenever (v_j, d) is stored in `graph[vi]`. This leads to an invariant we need to maintain and some redundancy in the representation, but it is the price we have to pay if we want fast bidirectional lookup of neighbours.

4.2 Reading and storing Input Information

We store the problem (V, W, E, t) in an object which is an instance of the class `WaterProblem`:

```
class WaterProblem {

    // Stores |V|, the cardinality of V.
    int v;

    // Stores V as a vector of integers.
    // We number the intersections n = 0, 1, ... , v-1 .
    vector<int> intersections;

    // Stores |W|, the cardinality of W
    int w;

    // Stores W, a list of v in V that are waterpumps
    vector<int> waterpumps_list;

    // For w = waterpumps_list[i], we define
    // index_in_waterpumps_list[w] = i
    // This means that we can enumerate
    // the waterpumps 1 to w-1 and also
    // quickly look up the enum value of any waterpump w.
    unordered_map<int,int> index_in_waterpumps_list;

    // This is t from the problem statement (V,W,E,t),
    // but we capitalize it to distinguish from local use
    // of t in subproblems.
    int T;

    // This stores for every v in V, a list of (v',d)
    // such that (v,v',d) in E or (v',v,d) in E
    // This means we treat the undirected multigraph as a
    // directed multigraph, but
    // with v -> v' iff v' -> v exists. This allows
    // us to quickly look up neighbours in both directions
    // along an edge (v,v',d), which will be needed for
    // Dijkstra's Algorithm
    unordered_map<int,vector<pair<int,int>>> graph;

}
```

The constructor reads the input of the described format and stores it in the following data structures:

```
    public:
    WaterProblem() {

        // The natural number e (number of edges)
        // as in the assignment:
        int e;

        // Read the first line of input into
        // the described constants:
        cin >> v >> w >> e >> T;

        // read the consecutive w lines "w" and
        // push these waterpumps into the vector.
        // also store the index in the vector i in
        // the map index_in_waterpumps_list[w].
        for (int i = 0; i < w; i++) {
            int waterpump;
            cin >> waterpump;
            waterpump -= 1;                          // 0 based numbering
            waterpumps_list.push_back(waterpump);
            index_in_waterpumps_list[waterpump] = i;
        }

        // for each v in V, initialize graph[vertex]
        // as an empty list for
        // elements of type (int, int), where the
        // first integer will be v' (a neighbour) and the
        // second d, for (v,v',d) or (v',v,d) in E.
        for (int vertex = 0; vertex < v; vertex++) {
            intersections.push_back(vertex);
            graph[vertex] = vector<pair<int,int>>{};
        }


        // read the consecutive v lines "v  v' d" and
        // store two edges: v gets neighbour (v',d)
        // and v' gets (v,d)
        for (int _ = 0; _ < e; _++) {
            int v1, v2, d;
            cin >> v1 >> v2 >> d;
            graph[v1-1].push_back(pair<int,int>{v2-1,d}); // 0 based numbering
            graph[v2-1].push_back(pair<int,int>{v1-1,d});
        }
    }
```

The amortized complexity of inserting/pushing into the maps and vectors is $\mathcal{O}(1)$, and therefore, reading input is easily seen to have running time $\Theta(|V| + |W| + |E|) = \Theta(|V| + |E|)$ since $|W| \in \mathcal{O}(|V|)$.

4.3 Graph Reduction using Dijkstra's Algorithm

It is logical to think that the larger G is, the more we will have to compute during the dynamic programming phase in order to solve the problem. We will see in section 4.4 that this is indeed the case. Because of this, we wish to reduce the size of G before we start the second phase of the algorithm.

We note that we do not care about travel times between every road intersection, but merely about the times between intersections containing a water pump. Because of this, it makes sense to eliminate all of the other nodes and simply consider the pumps and the shortest paths between them (we also keep the starting node, since this is where we are initially). To this end, we construct a new graph $\overline{G}$ as follows: The vertices of $\overline{G}$ are the pumps and the starting node. The edges $(u, v) \in \overline{G}$ represent the shortest path between u and v, and have that path length as their distance $d(u, v)$. For an example of a graph G and the corresponding graph $\overline{G}$, see figure 1. To construct $\overline{G}$, we run Dijkstra's algorithm from every pump the determine the shortest paths to every other pump in the network. Note that if we know the shortest path from every pump to pump w, we know the shortest path from w to every pump in the network. This means we only have to run Dijkstra n_w times, $n_w - 1$ times for all pumps except one, and once for the starting node.

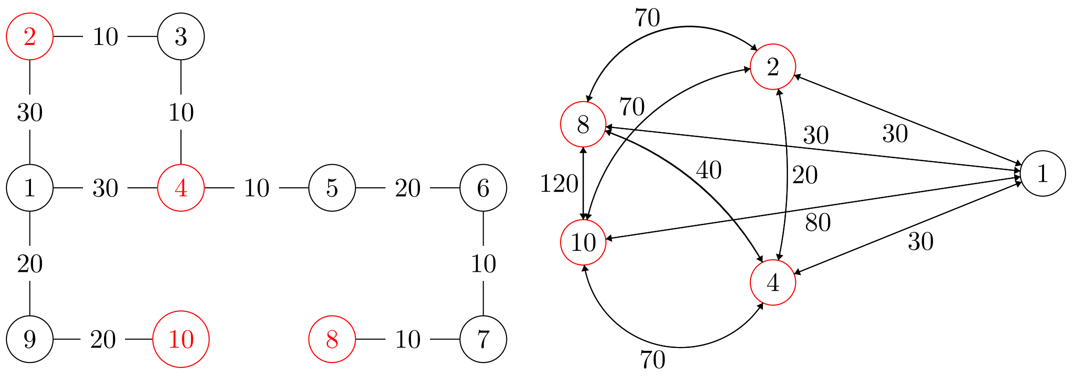

Figure 1: A pumping problem (left) and the corresponding reduced graph (right). The water pumps are indicated in red.

The implementation is a function within the class `WaterProblem`, which will replace the original `graph` with the reduced graph representation since we don't need the original anymore.

```
vector<pair<int,int>> dijkstra(
    unordered_map<int,vector<pair<int,int>>> graph,
    int source_vertex,
    const unordered_set<int>& final_vertices
) {
    // Returns a vector of (v,d) :: pair<int,int> where v is
    // a vertex in final vertices and d is the length of the
    // shortest path from source_vertex to v in the given
    // multigraph. The multigraph is assumed to be represented
    // by the map graph, a mapping from vertices (int)
    // to vectors of (neigbour, d) pairs.


    // initialize the result vector of travel costs:
    vector<pair<int,int>> travel_costs;

    // initialize the cost dictionary, where we initially
    // set all path lengths to other nodes to "infinity" = 10000
    vector<int> cost(graph.size(), 10000);

    // Create the priority queue
    priority_queue<pair<int,int>, vector<pair<int,int>>, greater<pair<int,int>>> Q;

    // Pute the source node in the queue:
    Q.emplace(pair<int,int>{0,source_vertex});

    // create a hash set of finished nodes:
    unordered_set<int> finished;

    // Canonical Dijkstra loop
    while (!Q.empty()) {
        int total_cost = Q.top().first;
        int top = Q.top().second;
        Q.pop();
        cost[top] = total_cost;

        if (final_vertices.find(top) != final_vertices.end()
        && finished.find(top) == finished.end())
            // if the current top node is in final_vertices,
            // we should emplace it in the result
            // adjacency vector.
            travel_costs.push_back(pair<int,int>{top, total_cost});

        finished.emplace(top);
        for(pair<int,int> nbr_dt : graph[top]) {
            int dt = nbr_dt.second;
            int neighbour = nbr_dt.first;
            if (finished.find(neighbour) == finished.end()
            && cost[neighbour] > dt + total_cost) {
                Q.emplace(pair<int,int>{dt + total_cost, neighbour});
                cost[neighbour] = dt + total_cost;
            }
        }
    }
    return travel_costs;
}

unordered_map<int,vector<pair<int,int>>> reduce (
    const unordered_map<int,vector<pair<int,int>>>& graph,
    const vector<int>& final_vertices_list
) {

    // Reduce a graph (V,E) to (V',E') where V' are the vertices in
    // final_vertices_list and (v,v', d) in E' if and only if there
    // is a shortest path from v to v' in (V,E) of length d.

    // Implemented as: run Dijkstra |W|+1 times on every node in W and v_0,
    // Create adjacency lists [(v',d)] for all v in V', and
    // this replaces our original graph.

    unordered_set<int> final_vertices_set;
    unordered_map<int,vector<pair<int,int>>> reduced_graph;

    for (int vertex : final_vertices_list)
        final_vertices_set.emplace(vertex);

    for (int vertex : final_vertices_list) {
        vector<pair<int,int>> reduced_neigbhours = dijkstra(graph, vertex, final_vertices_set);
        reduced_graph[vertex] = reduced_neigbhours;
    }
    return reduced_graph;
}
```

From inside the class WaterProblem, we can call reduce_graph, which almost directly calls reduce except that it handles a small edge case: either $v_0 \in W$ or $v_0 \notin W$, and depending on this, we need to add v_0 to V to create V':

```
     void reduce_graph() {

        // replace V with W:
        copy(waterpumps_list.begin(), waterpumps_list.end(), intersections.begin());
        intersections.resize(waterpumps_list.size());

        // add v0 to V or not if it is already in W:
        if (!is_water_pump(0))
            intersections.push_back(0);
        graph = reduce(graph, intersections);
    }
```

A very important notice: Since Dijkstra finishes each vertex in $W \cup \{v_0\} = V'$ once from each other node in V', the reduced graph is a **complete graph** with **no multiple edges**. Therefore, each pair $u, v \in V'$ has a well-defined, unique distance $d(u, v)$. This will ease our notation in what follows.

We stick to the original representation, although it is not strictly speaking necessary. Note however, that this representation now has a space complexity for graph of $\Theta(|E'|) = \Theta(|V'|^2) = \Theta(|W|^2)$, which would be the same space complexity as a map that only stores neigbouring nodes, and an extra map that stored $d(u, v)$ for each $(u, v) \in E'$

4.4 Recursion Relation

For the dynamic programming phase, we observe the following recursion: define $\mathcal{F}(t, v, W')$ **the maximum amount of water that can still be pumped if you have t time left, are in node v and W' are the pumps that have been turned on**. The problem that we need to solve is to compute $\mathcal{F}(t, v_0, \emptyset)$.

Suppose we wish to compute $\mathcal{F}(t, v, W')$, there are then two possible cases: Either $v \in W \setminus W'$ or $v \notin W \setminus W'$. In the first case, we already saw that it is best to **turn on the pump immediately** rather than to travel anywhere else first.

If $v \notin W \setminus W'$, i.e. the node we are visiting is not a pump or is a pump that is already on, we can increase the total amount of water pumped only by travelling to other nodes and seeing what we can do there. So the maximum amount of water that can be pumped is the maximum amount that can be pumped if we travel to a neighbor w of the current node and obtain $\mathcal{F}(t - d(w, v), w, W')$. We can without loss of generality assume that $w \in W \setminus W'$, since we have computed the shortest paths to all other waterpumps, so we do not have to consider intermediate steps via v_0, and **if we cannot travel to any $w \in W \setminus W'$**, this would by **completeness of the graph $\overline{G}$** mean that $W \setminus W' = \emptyset$, i.e. **we have already turned on all pumps**. This means that we cannot improve the amount, and we hit a base case, and $\mathcal{F}(t, v, W') = 0$. The other base case occurs when $t \leq 10$, in which case we have no time left to reverse any pump. This yields the following recurrence relation for the entries of $\mathcal{F}$:

$$\mathcal{F}(t, v, W') = \begin{cases} 0 & \text{if } t \leq 10 \text{ or } W' = W \\ \mathcal{F}(t - 10, v, W' \cup \{v\}) + 200 \cdot (t - 10) & v \in W \setminus W' \\ \max \{\mathcal{F}(t - d(w, v), w, W') \mid w \in \text{neighbors}[v] \cap W \setminus W'\} & \text{otherwise} \end{cases} \tag{1}$$

4.4.1 Nota bene: when there are edges with zero cost

Finally, we need to be mindful about the following: recursion may not terminate, i.e. **cycle** if there are edges with $d(v,w)=0$. Such cases are part of the problem statement and of the test cases. Even though the recursive relation would mathematically still be correct, the computation would not terminate: we compute $\mathcal{F}(t,v,W')$, thereby explore $\mathcal{F}(t-0,v,W')$ for some $w \in \text{neighbors}[v]$, with $d(v,w)=0$, therefore explore $\mathcal{F}(t-0,v,W')$ since $v \in \text{neighbors}[w]$ which leads to infinite regress because t is never decreased.

The above description can solve this problem easily due to the fact that we **look only at neighbours in**

$$\text{neighbors}[v] \cap W \setminus W'$$

This implies that any neighbour we explore, is a not-yet-turned-on water pump, which the next recursive call will turn on immediately, thereby decreasing the time. If $\text{neighbors}[v] \cap W \setminus W' = \emptyset$, then since **the reduced graph is complete**, this means $W = W'$, so we have already covered this in a base case.

We do not have to consider any $w \in \text{neigbours}[v]$ with $d(v,w)=0$ that are not in $W \backslash W'$: this would be necessary in the not-reduced graph, because there may be a pump that is only reachable via w (i.e., we have to look *behind* w to find all possible recursion relations).

For example, consider a sligth alteration of the problem in Figure 1: If we would slightly augment the problem and set $d(1,4)=0$, then we would still have to consider to travel to 4 from 1, even if 4 is already turned on, because we may be able to travel to 8 within the time limit. Now, in the reduced graph, *if* we can reach a node w from v via any path in the original graph, there will be a connecting edge: this means that there is no need to *look behind* neigbours with distance $d(v,w)=0$, and we can ignore these if they are not waterpumps. And otherwise, we can immediately decide to turn them on, because they require no travel time. To summarize, our algorithm does the following: while we are going over the neighbors of v, if we find a pump w that has not been turned on such that $d(v,w)=0$, we travel to w and turn it on. We represent this by setting

$$\mathcal{F}(t,w,W') = \mathcal{F}(t-10,w,W') + 200 \cdot (t-10),$$

and adding w to W'.

4.4.2 Nota bene 2: representing sets with integers

We would like the arguments of $\mathcal{F}(\cdot,\cdot,\cdot)$ to all be integers: this is also useful once we implement a memoization map, which should map (t,v,W') as a key to the memoized value $\mathcal{F}(t,v,W')$. t and v are already integers, but W is a **mathematical set**. It leads to very inefficient computations when we would W' as a C++ `unordered_set`, because in particular, we should be able to check equality of sets W', W'' fast, and to compute hashes of them for indexing in the memoization map. These are two operations that *"programming language" sets* are not meant for.

To represent a set of nodes W', we use a *set number*, which TL;DR is an integer $d(W')$ that uniquely identifies a set as a binary number with $n_w = |W|$ binary digits $d_0 d_1 ... d_{n_w - 1}$, where the i-th digit $d_i = 1$ if and only if `waterpumps_list[i]`$\in W'$.

Here, we enumerate the nodes in W as $w_0, w_1, w_2 ...$, then use the canonical bijection between the powerset of W and sequences of $\{0,1\}$ of length $|W|$: $2^W \sim \{0,1\}^{|W|}$. For the sequence s representing some set $W' \in W$, we set $s_i = 0$ if $w_i \notin W'$ and $s_i = 1$ if $w_i \in W'$. Using this bijective correspondence, we can then identify sets with binary sequence, which are in turn identified with integers: for example, (suppose $|W| = 5$), $W' = \{0, 1, 5\} \mapsto (1, 1, 0, 0, 1) \mapsto 19$. We can use such a *set number* to represent sets as integers: $2^{|W|} - 1$ then represents W, so we can check with some simple arithmetic whether an element is in a set. We can now quickly check equility between W' and W'' since they are equal if and only if $d(W') = d(W'')$. And we can use $d(W')$ as an integer that can be hashed to an index in the memoization map.

In order to enumerate the nodes $w_0, ..., w_{n_w-1} \in W$ and quickly look up for which i holds $w_i = w$, which is necessary to compute $d(W')$, we need an additional map which we already discussed, namely `index_in_waterpumps_list`. The following utility functions check for elementhood in a set number and can be used to add a waterpump to a set number, i.e. compute $d(W' \cup \{w\})$ given $d(W')$ and $w \in W \setminus W'$:

```
bool is_elem(int w, int set_number) {
    // returns whether w in W', where set_number = d(W')
    return pth_binary_digit(set_number, index_in_waterpumps_list[w]) == 1;
}

bool is_water_pump(int w) {
    // returns whether w is a water pump
    return index_in_waterpumps_list.find(w) != index_in_waterpumps_list.end();
}

int add_to_set(int w, int set_number) {
    // returns d(W'+w) given set_number = d(W') and w
    return set_number + (1 << index_in_waterpumps_list[w]);
}

bool is_full_set(int set_number) {
    // returns whether W' == W where set_number = d(W')
    return set_number + 1 == (1<<w);
}
```

Where we compute the p-th binary digit of an unsigned integer using some quick arithmetic:

```
int pth_binary_digit(int n, int p) {
    // computes the pth binary digit of positive integer n >= 0
    return (n % (1<<(p+1))) / (1<<p);
}
```

With these details arranged for, we can now give a very simple, naive implementation for $\mathcal{F}$, which for clarity is called `max_water_pumped` in the code.

4.4.3 Naive implementation

```
int max_water_pumped(
       int t,
       int current_node,
        int set_number,
            // The three variables (t, current_node, set_number)
            // completely describe the state.
    ) {

    if (t <= 10 || is_full_set(set_number)) {
        // We cannot turn on any pumps in this problem.
        return 0;
    }

    else if (is_water_pump(current_node)
        && !is_elem(current_node, set_number)) {

        // If the current_node is a water pump, but not yet turned on,
        // we should turn it on immediately

        // Hence add current_node to W'
        int set_number_plus_current_node = add_to_set(current_node, set_number);

        return max_water_pumped(t - 10, current_node, set_number_plus_current_node);
    }

    else {
        int max_water = 0;
        for(pair<int,int> w_dt : graph[current_node]) {
            int w = w_dt.first;
            int dt= w_dt.second;
            if (is_water_pump(w)
                && !is_elem(w, set_number)
                && t - dt > 10)

                // for each neighbour w that is a water pump that is
                // not-yet turned on, such that it can be reached in dt
                // time such that t - dt > 10 (i.e. there is time left to
                // turn the pump on), recursively compute max_water_pumped(t - dt, w, set_numbe
                // and return the maximum over all such w.

                max_water = max(max_water, max_water_pumped(t - dt, w, set_number));
        }
        return max_water;
    }
}
```

It is terribly slow and exponential in time; this is because it computes $\mathcal{F}(t', v', W')$ every time, for eacg problem where $\mathcal{F}(t', v', W')$ occurs in the expanded recursive relation somewhere. This creates a tree of recursion that branches at least into two subtrees at every subproblem, hence exponential time in $|V'|$ at least. A bit handwavy, this argument is; but we will not consider it for too long and rather move on to the idea to memoize solutions of subproblems: this is called ***Dynamic Programming***, and it will make the algorithm polynomial time.

4.5 Dynamic Programming

We solve this recurrence, but solving it using the and use memoization; we store $\mathcal{F}(t, w, W')$ in a map under the key (t, v, W) once it is computed, and when we need some value of $\mathcal{F}$ again, we first look it up in this map. This avoids computing anything twice.

We compute the recurrence relation top-down using a recursive function, but this recursive function also is passed a memoization map `memo` with key $(t, v, d(W')$, which it first checks for the key t, v, W that it should compute. Because this is C++, we don't index with a triple of integers, but rather nest a map within a map within a map, each level indexed with one of the integers $t, v, d(W')$

If an earlier recursive call already calculated this value, it stored this in the memo map. This ensures that no computation is executed twice. On the other hand, there are also no redundant computations of values for (t, v, W) that are never used in the computation of $(0, v_0,)$: this **would** be the case in a bottom-up dynamic programming approach, so we save a few unnecessary computations, although it does not change the worst-case complexity compared to a bottom-up approach since in the worst case we still have to compute $\mathcal{F}(t, v, W')$ for every possible state (t, v, W')

This leads to the following implementation, where each case (t, v, W') is either:

1. a **base case** where $W' = W$ or $t \leq 10$.
2. a recursive case, but we already computed and memoized it in `memo`; in that case, we look it up and return it.
3. a recursive case, but not pre-computed, so either we turn on a pump immediately, or we recursively explore the possibility to travel to neighbours.

With the **extra step** that if we could not look up the value in `memo`, we also have to insert it after computation for lookup in subsequent calls to this specific state.

```
int max_water_pumped(
        int t,
        int current_node,
        int set_number,
            // The three variables (t, current_node, set_number)
            // completely describe the state.

        unordered_map<int,unordered_map<int,unordered_map<int,int>>>& memo,
            // the memoization table
    ) {

    if (t <= 10 || is_full_set(set_number)) {
        // We cannot turn on any pumps in this problem.
        return 0;
    }
```

```
else if (memo.find(t) != memo.end()
    && memo[t].find(current_node) != memo[t].end()
    && memo[t][current_node].find(set_number) != memo[t][current_node].end()) {

    // This means that we have already done the calculation
    // for the state (t, current_node, set_number)

    return memo[t][current_node][set_number];
}

else if (is_water_pump(current_node)
    && !is_elem(current_node, set_number)) {
    // The value for the state (t, current_node, set_number) is not
    // present in the memoization table

    // If the current_node is a water pump, but not yet turned on,
    // we should turn it on immediately

    // Hence add current_node to W'
    int set_number_plus_current_node = add_to_set(current_node, set_number);

    memo[t][current_node][set_number] = max_water_pumped(
        t - 10,
        current_node,
        set_number_plus_current_node,
        memo);

    return memo[t][current_node][set_number];
}

else {
    int max_water = 0;
    for(pair<int,int> w_dt : graph[current_node]) {
        int w = w_dt.first;
        int dt= w_dt.second;
        if (is_water_pump(w)
            && !is_elem(w, set_number)
            && t - dt > 10)

            // for each neighbour w that is a water pump that is
            // not-yet turned on, such that it can be reached in dt
            // time such that t - dt > 10 (i.e. there is time left to
            // turn the pump on), recursively compute max_water_pumped(t - dt, w, set_numbe
            // and return the maximum over all such w.

            max_water = max(max_water, max_water_pumped(t - dt, w, set_number, memo));
            }
            memo[t][current_node][set_number] = max_water;
            return memo[t][current_node][set_number];

        }
    }
```

4.6 Branching and Bounding

The dynamic programmin approach is already quite good (its complexity is discussed below). In fact, adding a Branch-and-Bound (search pruning) mechanism does theoretically not improve the worst-case running time of our algorithm compared to dynamic programming. The practice is, however, that it did give interesting speedups on some of the larger test cases, and therefore it is part of our final algorithm.

In general, branch-and-bound-methods apply in a combinatorial optimization problem setting, where S is a (large, difficult to enumerate) solution set and we are looking for $s \in S$ where a function $f : S \to \mathbb{R}$ assumes its maximum. S is large and takes time to enumerate (in our case, S consists of all *water pumping routes* through the graph G that take less than t minutes, including the switching of pumps: the number of paths is exponential in $|W|$ i.e. "grows like a tree" if you consider how it can branch in many possible directions from each intersection), but we have a *heuristic* $h : D \to \mathbb{R}$ at our disposal, which is a partial function on the power set (i.e. $D \subset 2^S$) such that

$$\forall D' \in D : \forall s \in D' : f(s) \leq h(D')$$

That is, we can bound $f(s)$ that we can achieve on any of the $s \in D'$ by $h(D')$. In our case, we regard S as the set of all possible water pumping routes. Our D is t**he set of all routes that can be created** from a *partial solution.* A *partial solution* consists of a list of traveled edges that we have already traversed up until some state (t, v, W'), when $t > 10$ and $W' \neq W$ yet. That is, a partial solution is the "state of our strategy" at the start of some recursive call, where we also count the current water pumped (that is, all $(t_i - 10) \cdot 200\ M = m^3$ for all timestamps $t_1, ..., t_k$ at which we have already turned on a pump).

What is an upper bound for the total amount of water that can be pumped at such an intermediate step, in other words, what should we define $h(D)$ for the set D of solutions that can stem from this intermediate step? We can construct such an estimate by considering the following **relaxation** of the problem:

All edges have $d = 0$

In that case, if we are time t and $|W'| = N$ while $|W| = n_w$, we can turn on at most

$$\min\{\lfloor \frac{t}{10} \rfloor, n_w - N\}$$

pumps in the remaining time (since each pump needs 10 minutes to turn on, and there are at most $n_w - N$ pumps that can be turned on).

If we could travel between nodes at cost $d = 0$, then the remaining $\mathcal{F}(t, v, W')$ becomes easy to calculate: it is just

$$\sum_{i=1}^{m} (t - 10i) \cdot 200, \quad \text{where } m = \min\{\lfloor \frac{t}{10} \rfloor, n_w - N\}$$

Some simple arithmetic shows that this equals:

$$200 \cdot m \cdot t - 1000 \cdot m \cdot (m+1), \quad \text{where } m = \min\{\lfloor \frac{t}{10} \rfloor, n_w - N\}$$

So this gives us a good upper bound h for $\mathcal{F}(t, v, W')$, and it can be used to give an upper bound of how much we could achieve if we continue down this searching branch: it is simply how much water we already pumped, plus the upper bound for $\mathcal{F}(t, v, W')$. Then, if we store globally the best possible value of f that has already been achieved, we can immediately stop exploring this partial solution further: it will always come out at a lower total water_pumped value.

4.6.1 Implementation of the Heuristic

Implementationwise, we now have an upper bound function for a state (t, v, W') that is a $\mathcal{O}(()\,1)$ closed-form calculation, which only depends on the values t and $n_w - |W'|$, the latter we will logically name `nr_available_pumps`:

```
int upper_bound(int t, int nr_available_pumps) {
    // returns an upper bound for how much total water we can
    // still pump away given that we are in a state with
    // timelimit t and nr_available_pumps pumps that are
    // not yet turned on.

    const int m = nr_available_pumps;
        // this is how many pumps can still be turned on in the
        // remaining time, if we could travel between
        // intersections in 0 minutes
    return 200 * m * t - 1000 * m * (m + 1);

}
```

The function `max_water_pumped` is upgraded again: it is given three extra arguments:

1. `nr_available_pumps`: this number is $|W'|$. While it could technically be calculated from `set_number`, this is tedious and also time-consuming to do (it involves counting zeroes and ones in a binary number of size n_w, adding a factor n_w to the complexity if we would do it in this way). Instead, we pass this n_w in the top-most call, and if pumps are turned on during the search, we decrease it in recursive calls.

2. `current_water_pumped`: via this argument, a calling state can pass information on how much water is already ***secured***: that is, when we turned on a pump at a time t earlier, the grand total water that will be pumped by this pump (i.e. $200 \cdot (t - 10)$ is added to `current_water_pumped` when it is passed to a recursive call.

3. `attained_lower_bound`: a reference to an integer holding the best attained water_pumped value over all completely explored water pumping routes. In the initial, top-most call, this is passed a reference to an int `lower_bound = 0`. Any base case (leaf call) is allowed to update this value if managed to achieve a better water_pumped value.

Using this data, we can immediately cut the branch once we observe that

```
current_water_pumped + upper_bound(t, nr_available_pumps) <= attained_lower_bound
```

And this is the key idea of the final implementation of `max_water_pumped`:

```
int max_water_pumped(
    int t,
    int current_node,
    int set_number,
        // The three variables (t, current_node, set_number)
        // completely describe the state.

    int nr_available_pumps,
        // This number can in principle be calculated
        // from set_number, but since that adds O(|W|) complexity
        // to each call to this function, we keep it
        // as an extra argument.


    unordered_map<int,unordered_map<int,unordered_map<int,int>>>& memo,
        // the memoization map

    int current_water_pumped,
        // an extra variable to keep track of how much water has
        // been pumped by parent calls along the currently explored
        // branch.

    int& attained_lower_bound
        // a reference to the value of the best solution
        // that has been found yet.
        // Initially, this reference holds 0.
        // If we are in a leaf call and a better path is found,
        // the reference is updated to current_water_pumped
    ) {

    if (current_water_pumped + upper_bound(t, nr_available_pumps)
                <= attained_lower_bound) {
        // We cannot hope to do better.
        // Might as well return 0 because we don't
        // care about what we do next
        // But DON'T store 0 in the memo map! It breaks the
        // invariance that the memo map contains correct values.
        return 0;
    }

    else if (t <= 10 || is_full_set(set_number)) {
        // If here, we know that we have found a better solution,
        // otherwise control would have gone to the first if-branch.
        // And that we also have reached a base case:
        // Therefore, it is a good idea to update the
        // attained_lower_bound to a new optimal value.

        attained_lower_bound = current_water_pumped;
        return 0;
    }

    else if (memo.find(t) != memo.end()
    && memo[t].find(current_node) != memo[t].end()
    && memo[t][current_node].find(set_number) != memo[t][current_node].end()) {

        // This means that we have already done the calculation
        // for the state (t, current_node, set_number)

        return memo[t][current_node][set_number];
    }


    else if (is_water_pump(current_node)
        && !is_elem(current_node, set_number)) {

        // The value for the state (t, current_node, set_number) is not
        // present in the memoization table

        // If the current_node is a water pump, but not yet turned on,
        // we should turn it on immediately

        // Hence add current_node to W'

        int set_number_plus_current_node = add_to_set(current_node, set_number);
        nr_available_pumps -= 1;
        current_water_pumped += 200 * (t - 10);
            // add the grand total immediately;
            // this is fine if we do it consistently.


        memo[t][current_node][set_number] = max_water_pumped(
            t - 10,
            current_node,
            set_number_plus_current_node,
            nr_available_pumps,
            memo,
            current_water_pumped,
            attained_lower_bound) + 200 * (t - 10);
        return memo[t][current_node][set_number];
    }



    else {
        int max_water = 0;

        // for each neighbour w that is a water pump that is
        // not-yet turned on, such that it can be reached in dt
        // time such that t - dt > 10 (i.e. there is time left to
        // turn the pump on), recursively compute
        // max_water_pumped(t - dt, w, set_number)
        // and return the maximum over all such w.


        for(pair<int,int> w_dt : graph[current_node]) {
            int w = w_dt.first;
            int dt= w_dt.second;
            if (is_water_pump(w)
                && !is_elem(w, set_number)
                && t - dt > 10)
                max_water = max(max_water,
                    max_water_pumped(
                        t - dt,
                        w,
                        set_number,
                        nr_available_pumps,
                        memo,
                        current_water_pumped,
                        attained_lower_bound));
        }
        memo[t][current_node][set_number] = max_water;
        return memo[t][current_node][set_number];
    }
}
```

5 Correctness

5.1 Dynamic Programming

The correctness of the dynamic programming phase follows from the correctness of the recurrence relation which we have already discussed in great detail in section 4.4.

5.2 Graph Reduction and Dijkstra

We focus on the correctness of the reduction phase: in particular, Dijkstra indeed finds the shortest paths, even in a multigraph, since it always considers **all outgoing edges**, so when it encounters the same neighbour v' of v twice by exploring both (v', d) and (v', d') in the adjacency list of v, it will automatically put the shorter d above the other in the priority queue, effectively ignoring the longer edge. Thereby, it is practically equivalent to Dijkstra on a **simple** (meaning not more than one edge between two nodes), directed graph, and since our graph representation accounts for the bidirectionality of the edges by storing each (v, v', d) as a $(v', d) \in \text{neigbours}[v]$ and a $(v, d) \in \text{neigbours}[v']$, it effectively treats the graph as a bidirectional graph. Since all edges have nonnegative weights, Dijkstra's Algorithm is correct on such graphs, and thus we conclude that it indeed finds the shortest paths).

Why does the reduced graph have an equivalent solution? Well, we already explained that we need only care about which pumps we visit, and not necessarily about which nodes we visit in between. The earlier a pump is turned on, the more water it can pump before the time limit is reached. So we look for the fastest route available when traveling between pumps. We have already seen in the lecture that Dijkstra's algorithm correctly computes these shortest paths. Since we only care about the pumps we visit and not about the nodes in between, we can simply consider the shortest path between two pumps as a single edge with the length of the shortest path in the original graph G. Therefore we conclude that our reduction of G does not affect the optimal solution found by the algorithm.

5.3 Branch and Bound

Cutting off a search branch and returning just 0 as a value prematurely can only be done if we are very-very certain that we cannot indeed find an optimal solution if we look a bit further. We have already discussed that we have an ***admissible heuristic*** for the upper bound computation, since this heuristic (i.e. the function `upper_bound`) essentially computes the maximum value for a relaxed problem (namely, the problem with zero travel times between nodes, which are always at most the travel times than the ones we encounter in the actual given graph). So by admissibility of the heuristic, pruning is correct. As long as we don't store the "just 0" that is returned in the memoization table, it is fine to return just 0. Note however that the memoization table should always only contain correctly computed values $\mathcal{F}(t, v, W')$, since we need it to look up these values.

6 Time complexity

6.1 Reading the input and providing the final answer

Before the algorithm can be executed, our program must read the provided input. This requires reading $1 + n_w + e$ lines. Since we know that $n_w \leq 12$ for all problems, the time complexity of this is $\mathcal{O}(e + n_w)$. Since we make the graph while reading the input, and the graph is programmed as an `unordered_map` from nodes to adjacency lists, which is internally a hash table that provides (amortized) $\mathcal{O}(1)$ insert and access operations, and this unordered map contains *adjacency vectors*, which also have an amortized $\mathcal{O}(() 1)$ access and push back time, inserting and accessing neighbours and costs stored in this representation does not incur additional time complexity.

As for returning the final answer: the answer to the problem is obtained by executing the dynamic programming algorithm and then returning the value in $\mathcal{F}(t, v_0, \emptyset)$, once the necessary parts of the map have been filled out by the algorithm. Returning $\mathcal{F}(t, v_0, \emptyset)$ has time complexity $\mathcal{O}(1)$, the time complexity of the dynamic programming will be discussed below.

6.2 Graph Reduction

We start by discussing the time complexity of the reduction phase. In this phase we run Dijkstra's algorithm n_w times. Dijkstra's algorithm uses a priority queue to prioritize nodes with the lowest current cost to be finished first. As a result discussed in the lecture, this gives a time complexity of $\mathcal{O}((e + n_v)\log(n_v))$. So the total time complexity of the graph reduction phase, considering that we run Dijkstra n_w+1 or n_w times (depending on whether $v_0 \in W$ or not) is $\mathcal{O}(n_w \cdot ((e + n_v)\log(n_v)))$. Storing the new entries of the resulting reduced graph in a new adjacency map representation is accounted for here (this requires constant-time inserts in vectors and unordered maps which are hidden in the complexity of the helper functions).

6.3 Dynamic Programming + Branch and Bound measures

We now discuss the complexity of the dynamic programming phase. Since we eliminate all nodes that are not water pumps or the starting node v_0 during the first phase, $\mathcal{F}$ can be memoized using an at most $t \times (n_w + 1) \times 2^{n_w}$-sized memo map. We get to this specific size by noting that

1. There are $t + 1$ possible time values $0, 1, ..., t$, and each of these values might occur for states that we might come along during the recursion.
2. The range of possible intermediate nodes that we could be in is $|V'|$, the number of vertices of the reduced graph: either $n_w + 1$ or n_w.
3. The number of possible subsets $W' \subset W$ is 2^{n_w}.

Since all triples (t, v, W') in the product of these ranges might occur, we get a worst case of $\mathcal{O}(t \times (n_w + 1) \times 2^{n_w})$ states that we might need to come by during our search for the optimal path. This is often less, especially due to the pruning that the Branch-and-Bound mechanism might give us, **but** there are graphs where we might **never be able to prune** and **need to consider all subsequent states**. An example:

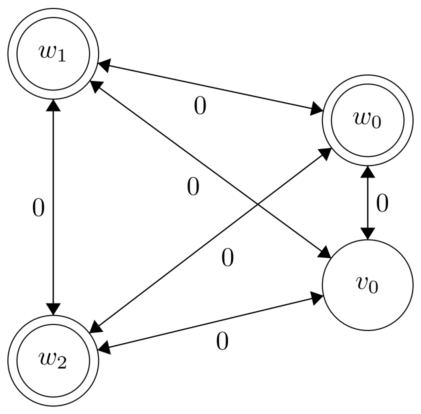

Figure 2: Let t be 40

In this example, where we let $t = 40$*, we cannot conclude that we have indeed found the best value of* $30 + 20 + 10 = 60$ *until we have explored every possible order of traversing the nodes in this graph: the problem is that the heuristic* $h(t, v, W')$ *always equals* $\mathcal{F}(t, v, W')$*, so we can never prune. However, in this case we do only consider discrete timestamps* $0, 10, 30, 40$*, so we have not reached the upper bound on the number of states to consider.*

In the worst case, all entries of this map have to be computed, but if this is not necessary to compute $\mathcal{F}(0, v_0, \emptyset)$, there will be no more computations than necessary **due to the top-down approach**. No value of the map is computed twice thanks to memoization, giving an upper bound of $t \times (n_w + 1) \times 2^{n_w}$ recursive calls. Base cases are computed in constant time (`return 0`), and recursive calls in the worst case **require the execution of the $O(n_w)$ loop in the final `else`-branch**, with $O(n_w)$ `max`-computations (which are constant time) (this analysis disregards the complexity hidden in the recursive calls, which belong netto to those recursive calls), so a call to `max_water_pumped` in the worst case has a complexity of $\mathcal{O}\left(()\, n_w\right)$. In the worst case, we do this for all $t \times n_v \times 2^{n_w}$ possible non-base case states, so we conclude that the dynamic programming phase has time complexity $\mathcal{O}\left(t \cdot (n_w + 1) \cdot 2^{n_w} \cdot n_w\right) = \mathcal{O}\left(t \cdot n_w^2 \cdot 2^{n_w}\right)$.

6.4 Total Complexity

This is the sum of the complexities of each phase, amounting to a total complexity of:

$$\mathcal{O}\left(n_v + e + n_w^2 \cdot t \cdot 2^{n_w} + n_w(e + n_v)\log(n_v))\right)$$

Where:

- The $\mathcal{O}\left(n_w + e\right)$ term comes from reading input and creating the original problem graph.
- The $\mathcal{O}\left(n_w(e + n_v)\log(n_v))\right)$ term comes from the graph reduction phase.
- The $\mathcal{O}\left(n_w^2 \cdot t \cdot 2^{n_w}\right)$ comes from the DP-BB phase.

7 Code

We already gradually described the code (as an alternative for pseudocode) during the development of the idea in section 4. The full code, which has been tested before submission, can be found accompanying the submission.