# Lab 1 — Centralized Coordination Algorithms

Multi-Agent Systems (TDDE13), Linköping University
*Fredrik Präntare (fredrik.prantare@liu.se)*

**Directions:** Solve the problems below and submit your solutions to the online judge.

# Coordinating Agents with Centralized Algorithms

So far we've mainly focused on modeling agents from a game-theoretic perspective. We've built mathematical models with the aim to analyze the agents' behaviors, strategies, and decision-making capabilities; and discussed mechanisms to e.g., make systems of agents behave in certain ways.

In this lab, we shall instead look at a few different ways to coordinate and organize multiple agents in a centralized (non-distributed) fashion, so that they can work together to solve problems. Doing so can significantly affect a multi-agent system's performance—the agents can, in many instances, be organized and coordinated so that they can solve tasks more efficiently, and consequently benefit collectively and individually.

In more detail, we shall look at two well-studied optimization problems in multi-agent systems and operations research, namely:

- the *linear assignment* problem (Task 1); and
- the *coalition structure generation* problem for *characteristic function games* (Task 2),

and implement solutions for them. We will use the online judge *Kattis* to automatically correct your implementations and lab submissions. Although we (strongly) recommend using the code base implemented in Java (for Task 1) and C++ (for Task 2) that you can download from the course's webpage, you decide yourself which of the supported programming languages to use (students have previously used e.g., Python and C). See `https://liu.kattis.com/help` for more information on supported languages and how to get started. (If you are new to online judges, we strongly recommend that you first solve a few of the easier problems at `https://open.kattis.com/problems` before proceeding!)

## Task 1: Coordinating Agents with an Assignment Algorithm

Suppose that a set of agents (or players) $N$ have already decided to work together (e.g., in a coalition $C = N$) to achieve some goal, and that to achieve this goal, the agents have to solve a set of tasks $M$. A question we might ask is: How should we divide the labour (i.e., the tasks) between the agents to maximize their joint productivity?

For example, suppose that Joe, Ada and Sam are celebrating a special holiday together. To make this special occasion enjoyable, they have to:

1. clean the house;
2. cook dinner; and

3. decorate the dining table.

If Joe, Ada and Sam have different skills, proclivities and preferences, it is reasonable to assume that there is also a preferred division of labour that maximizes the group's welfare and efficiency (i.e., its potential payoff).

The *assignment problem* revolves around this question of deciding on whom should perform which task—or more generally, finding a cost-minimizing bijection between two sets (can you see why this is an equivalent problem to finding a value-**maximizing** bijection?). In this lab, we shall model this type of situation as a *linear assignment problem*, in which we assume that:

1. there's a cost function $c : N \times M \mapsto \mathbb{R}$ that represents the incurred cost for a specific agent-to-task assignment;

2. the agents are required to perform as many tasks as possible by assigning at most one agent to each task, and at most one task to each agent; and

3. we want to minimize the *total cost* (i.e., the aggregated sum) of all agent-to-task assignments.

For simplicity and clarity during this lab, we assume that $n = |N| = |M|$. This is also without loss of generality, since if $|N| \neq |M|$, we can add "dummy" agents or jobs so that $|N| = |M|$.

More formally, the goal of the linear assignment problem is then to find a bijection $f : N \mapsto M$ that minimizes:

$$\sum_{i \in N} c(i, f(i)).$$

This is a fundamental problem in the field of combinatorial optimization, and it has many real-world applications that are important to multi-agent systems and operations research.

A naïve solution to solve this problem is to evaluate all possible bijections, and return one with the lowest total cost (this bijection is called *optimal*). However, while there is a factorial number of such bijections (since there are $n!$ permutations of $N$), there are algorithms that can solve the problem much more efficiently, for example the famous *Hungarian algorithm* [Kuhn, 1955] that solves it in polynomial time. It is also possible to solve this problem efficiently in polynomial time by first representing it as a *weighted flow network*, and then using e.g., *Edmond–Karp*'s algorithm (more on this later) to find a *minimum cost maximum flow* (i.e., a maximum flow with lowest aggregated cost), which can then be *transformed*[1] into an optimal bijection. This is the approach we shall discuss and use in this lab.

There are a few key reasons why we use this approach here: a) Minimum cost maximum flow algorithms can be used to solve many different types of assignment problems if you can find a correct "flow representation" of them, which is often much easier and less complicated than implementing a new type of algorithm; and b) the efficient solution that we devise here (i.e., Edmond–Karp's algorithm) can be broken down into a number of simpler parts (e.g., *Dijkstra*) that can be understood, discussed, implemented and tested separately.

[1]In complexity theory, we call this type of a transformation a *polynomial-time reduction.*

Against this background, and in more detail, we shall solve the linear assignment problem with the following 4 steps:

1. Represent the linear assignment problem as a minimum cost maximum flow problem.
2. Implement Edmond–Karp's algorithm for maximum flow.
3. Make a minor adjustment to Edmond–Karp's algorithm so that it can solve minimum cost maximum flow problems. (This step involves implementing Dijkstra's algorithm for finding single source shortest paths.)
4. Use our modified algorithm (from step 3) together with our minimum cost maximum flow problem representation (from step 1) to solve the linear assignment problem.

For now, these steps may seem both unintuitive and disconnected. But as we proceed, they will (hopefully) appear more rational and become easier to grasp. Note that while you may choose to implement all of these steps yourself, we strongly recommend you to use the Java-based code base that you can download from the **course's webpage** which contains a working implementation of Edmond–Karp's algorithm for minimum cost maximum flow problems (i.e., you only have to complete Step 1 and Step 4). In any case, you need to understand the main intuition behind all four steps to complete the lab, so you should at least read and understand the following sections.

## Step 1: Linear Assignment as Minimum Cost Maximum Flow

Suppose we have a *weighted flow network* $(G, f_{cap}, f_{cost}, s, t)$, where $G = (V, E)$ is a directed graph (with $V$ being its set of vertices and $E$ its set of edges), $f_{cap} : E \mapsto \mathbb{N}^+$ its *capacity function*, $f_{cost} : E \mapsto \mathbb{R}$ its *cost function*, $s \in V$ its *source* and $t \in V$ its *sink*. The question is: How can we design a flow network of this type for which a *minimum cost maximum flow* can efficiently be transformed into an *optimal bijection* for a given linear assignment problem? (We urge you to think about this question and try to answer it before reading on!)

The answer to this question is shown in Figure 1. Here, we let there exist $|N|$ vertices that represent the agents (the "agent vertices"), together with $|M|$ vertices that represent the tasks (the "task vertices"). Then, we let there be an edge with capacity 1 and cost 0 from the source to every agent vertex. (These edges represent that each agent can be assigned to exactly one task.) Moreover, we form $|N||M| = n^2$ edges between the agent and task vertices—one for every possible agent-to-task assignment. Each such edge is given a capacity equal to 1, and a cost equivalent to the corresponding agent-to-task assignment's cost (given by the linear assignment problem's cost function). Finally, there is an edge with capacity 1 and cost 0 from every task vertex to the sink.

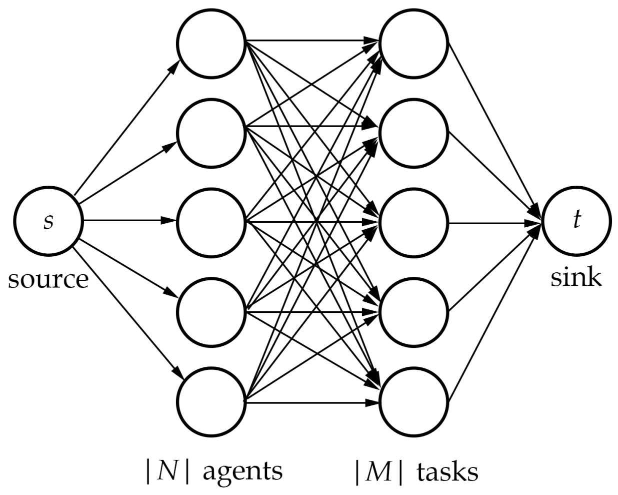

Figure 1: A linear assignment problem represented as a flow graph.

With this representation in mind, we leave it to the reader to derive (and subsequently implement) how to convert this weighted flow network's *minimum cost maximum flow* into an optimal bijection for a given linear assignment problem. Moreover, for implementing a flow graph in this lab, we strongly recommend using an *adjacency list* (instead of e.g., an *adjacency matrix*)—see for example the Java code base provided on the course's webpage.

## Step 2: Edmond–Karp for Maximum Flow

Edmond–Karp's algorithm for maximum flow is an implementation of the more general *Ford–Fulkerson method*. In fact, it is identical to it, except that it also defines *how to find* augmenting paths, which it uses breadth-first search for. So, how does Ford–Fulkerson's method work? We shall give the main gist of it, and then, after some thought—and perhaps together with a few online resources—you should be able to implement it yourself.

First, recall that a *maximum flow* is a network flow for which the aggregated flow on the edges adjacent to the sink vertex is maximum. In other words, there is no other way of routing more flow from the source vertex to the sink vertex. Also, note that an *augmenting path* is a path (an ordered set of edges) from the source to the vertex for which we can push additional flow (i.e., the path has "available" net capacity on all of its edges).

Second, suppose we designed and implemented Algorithm 1. This method works as follows: As long as there is at least one augmenting path, we route as much flow as we can through it. This process is shown in Figure 2. Perhaps surprisingly, this very simple algorithm is not only efficient, but it also *almost* works.

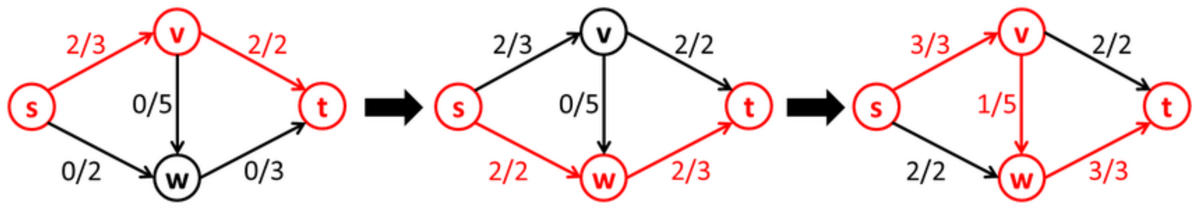

Figure 2: An example of finding augmenting paths and routing as much flow as possible through them. Here we find three different augmenting paths before we stop.

**Algorithm 1** : `ProblematicAugmentingPathMaxFlowAlgorithm`

```
1: maximum_flow ← 0
2: while there is an augmenting path do
3:     augmenting_path ← FindAugmentingPath()
4:     min_cap ← FindMinimumCapacityAlongPath(augmenting_path)
5:     push min_cap flow along augmenting_path
6:     maximum_flow ← min_cap
7: return maximum_flow
```

However, there is one critical flaw to it, which appears when one of the augmenting paths we choose “blocks” a series of other more “fruitful” (i.e., with a larger total flow) augmenting paths—see Figure 3 for an example of when this can occur. Consequently, the method may return a (suboptimal) local optimum.

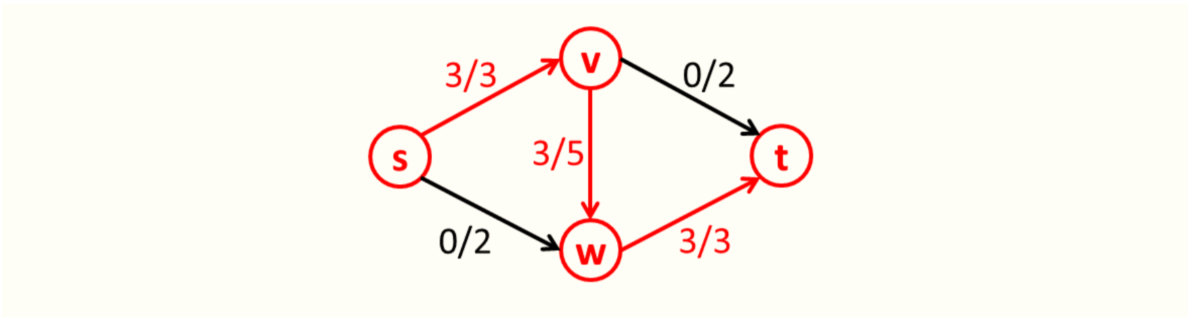

Figure 3: An example of a locally optimal “blocking” flow. We first have to ”revert” some of the current flow to route more flow from the source to the sink along an augmenting path.

So, can we remedy this flaw, or do we have to think about a different method to solve the problem? Luckily, we only have to add *residual edges* (also called *back edges*) to fix it. These residual edges mirror the original edges by pointing in the opposite direction. Moreover, they have a capacity which is equal to the corresponding original edge’s *current flow*. So, when there is no flow, the back edges have a capacity equal to zero. Then, their capacities are continuously updated as the network’s flow is altered. More importantly, these edges allow us to partially “undo previous flow”. See Figure 4 for an example.

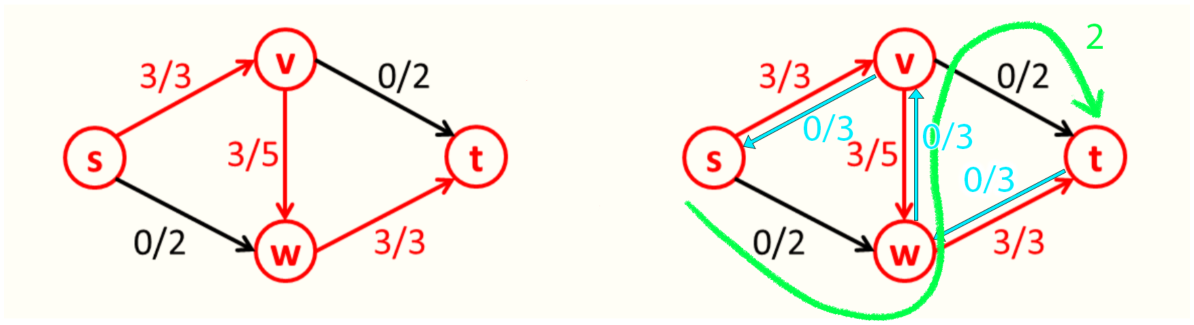

Figure 4: Adding residual edges (coloured cyan in the graph) makes it possible for us to find new augmenting paths (visualized by the green path) by partially undoing previous flow.

With this in mind, pseudocode for Ford–Fulkerson's method is displayed in Algorithm 2. As previously mentioned, this algorithm is equivalent to Edmond–Karp's algorithm if breadth-first search is used to find augmenting paths.

**Algorithm 2** : `FordFulkersonMaxFlowAlgorithm`

```
1: initialize residual (back) edges
2: maximum_flow ← 0
3: while there is an augmenting path do
4:     augmenting_path ← FindAugmentingPath()        ▷ Edmond–Karp's uses BFS here.
5:     min_cap ← FindMinimumCapacityAlongPath(augmenting_path)
6:     push min_cap flow along augmenting_path
7:     maximum_flow ← min_cap
8: return maximum_flow
```

You can now try to implement Edmond–Karp's version of this algorithm, and test your implementation using `https://liu.kattis.com/problems/maxflow` before proceeding. (Implementing this algorithm might take a few hours, but it thankfully also constitutes the main part of this lab. An implementation of this algorithm can then be used to solve a wide range of different matching problems! Alternatively, you can choose to use the Java code base that you can download from the course's webpage.)

## Step 3: Edmond-Karp Minimum Cost Maximum Flow

If you have a working Edmond–Karp implementation, this step is relatively simple: Instead of just finding any augmenting path, we want to find the path with the lowest aggregated flow cost. This can, in a straightforward fashion, be achieved by using *Dijkstra's algorithm* instead of breadth-first search. (Other single source shortest path algorithms can also be used.) See Algorithm 3 for pseudocode. Note that since the back edges have a negative cost, your Dijkstra implementation must allow revisiting nodes when a cheaper path (containing a negative cost edge) is found. See Figure 5 for an example of this issue. Since the residual graph will not contain negative-weight cycles, Dijkstra's algorithm will still terminate. We recommend you to now implement Dijkstra's algorithm (separately!), and test your implementation with `https://liu.kattis.com/problems/shortestpath1`. Then, combine it

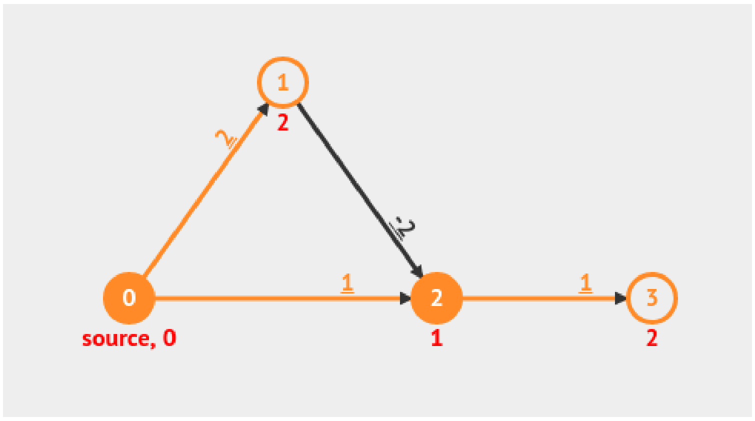

Figure 5: Example of using Dijkstra's algorithm to find the shortest path in a graph containing a negative weight edge. Node 2 has already been expanded using the path 0-2 with cost 1, but there exists a cheaper path 0-1-2 with cost 0. When this path is found, node 2 must be revisited. Since no negative weight cycle exists, when node 2 and 3 have been revisited, the algorithm will terminate. Image source: `https://visualgo.net/en/sssp`.

with your previous maximum flow implementation, and test the modified algorithm with `https://liu.kattis.com/problems/mincostmaxflow`.

**Algorithm 3** : `MinCostMaxFlowAlgorithm`

```
1: initialize residual (back) edges
2: maximum_flow ← 0
3: while there is an augmenting path do
4:     augmenting_path ← FindAugmentingPathWithMinimumCost()   ▷ Use e.g., Dijkstra.
5:     min_cap ← FindMinimumCapacityAlongPath(augmenting_path)
6:     push min_cap flow along augmenting_path
7:     maximum_flow ← min_cap
8: return maximum_flow
```

### Step 4: Solving Linear Assignment with Edmond-Karp

This step is straightforward. First, implement the weighted flow network from step 1. Then, use the "modified" algorithm you implemented in step 3 to find the network's *minimum cost maximum flow*. Finally, check the flow on the edges between the agent and task vertices to extract an optimal bijection (i.e., a solution to the linear assignment problem) from the weighted flow network.

### Final Notes on Task 1

When you have made a correct implementation, submit it here: `https://liu.kattis.com/problems/liu.assignmenteasy`. You need to be registered (you can register for free) and logged in with your *liu.kattis account* (i.e., not your *open.kattis account*). Information about how to format your output and handle the input can be found there (you get this part for free if you use the code base provided on the course's webpage). Note that you are allowed to submit as many solutions as you want—there is no limit, and erroneous submissions are not penalized! Moreover, to pass the lab, you only need to have at least one accepted submission. Finally, keep in mind that, while the code quality is not judged, exceptionally bad and unreadable code will not be graded (even if Kattis accepts it).

***Extra* (optional):** If you want a harder challenge, try implementing the Hungarian algorithm and solve the harder version of this problem that can be found here `https://liu.kattis.com/problems/liu.assignmenthard`. (A correct implementation of the Hungarian algorithm will get accepted on both versions of the problem!)

## Task 2: Coalition Structure Generation

In coalitional game theory, a fundamental algorithmic problem is that of *coalition structure generation*. In this problem, we aim to find a partitioning of the agents into a set of exhaustive and disjoint coalitions called *coalition structures* (Definition 1) that maximizes the system's performance/utility (e.g., social welfare). In particular, we shall focus on coalition structure generation for *characteristic function games* (Definition 2), in which we assume every coalition has a value assigned to it that corresponds to its potential utility.

**Definition 1.** *Coalition structure. A* coalition structure $CS = \{C_1, ..., C_{|CS|}\}$ *over the agents* $N$ *is a set of coalitions with* $C_i \subseteq N \setminus \emptyset$ *for* $i = 1, ..., |CS|$, $C_i \cap C_j = \emptyset$ *for all* $i \neq j$, *and* $\bigcup_{i=1}^{|CS|} C_i = N$.

*For example,* $\{\{a_1, a_3\}, \{a_2\}\}$ *and* $\{\{a_1\}, \{a_2\}, \{a_3\}\}$ *are two different coalition structures over* $N = \{a_1, a_2, a_3\}$.

**Definition 2.** *Characteristic function game. A* characteristic function game *is a coalitional game* $(N, v)$, *where* $N$ *is a set of players (agents), and* $v : 2^N \mapsto \mathbb{R}$ *maps a value to every possible coalition* $C \subseteq N$. $v(\emptyset) = 0$ *is assumed.*

More formally, the conventional coalition structure generation problem for characteristic function games that we shall work with is defined as follows:

---

**Input:** A characteristic function game $(N, v)$.

**Output:** A coalition structure $CS$ over $N$ that maximizes $CS$'s *value* $V(CS) = \sum_{C \in CS} v(C)$.

---

This problem is NP-hard, and the number of solutions grows in $\mathcal{O}(n^n)$, where $n = |N|$. Moreover, to even find an approximate solution guaranteed to be within any bound from optimum, we have to first scan all $2^n$ possible values of the value function! In other words, it is a pretty difficult computational problem. [Rahwan et al., 2015]

In light of these observations, in this task, we "circumvent" this difficult, and instead experiment with the coalition structure generation problem in a *non-exact* fashion—i.e., in a non-optimal way that returns a feasible solution to the problem in polynomial time. There are many different reasonable ways to approach this, including using general-purpose methods such as *local search*, *genetic algorithms*, *backtracking* and *branch-and-bound*. You are free to choose and experiment with as many ideas and methods you want and can think of. In more detail, to complete the task, you need to implement at least one method (not necessarily one listed here), and achieve a score greater than 20 on the following Kattis problem: `https://liu.kattis.com/problems/liu.csg`. (The maximum score is 52.) To get started, we recommend you to just start experimenting with various way to partition the different agents (e.g., greedily or randomly).

As before: When you have made a correct implementation, submit it to Kattis. Information about how to format your output and handle the input can be found there. Note that—while the code quality is not judged—exceptionally bad and unreadable code will not be graded (even if Kattis accepts it).

***Extra*** **(optional):** Try to beat the current student high score of 34.073051!

## References

Harold W Kuhn. The hungarian method for the assignment problem. *Naval research logistics quarterly*, 2(1-2):83–97, 1955.

Talal Rahwan, Tomasz P Michalak, Michael Wooldridge, and Nicholas R Jennings. Coalition structure generation: A survey. *Artificial Intelligence*, 229:139–174, 2015.