Lab 1 — Centralized Coordination Algorithms

Multi Agent Systems (TDDE13), Linköping University
Fredrik Präntare (fredrik.prantare@liu.se), Autumn Semester 2019

Directions: Solve the exercises below. You should submit your source code and a 2-page report with your results to your TA before the deadline.

Deadline: 08:00, 6 December 2019.

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.

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

Exercise 1: 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 $N = \{a_1, a_2, a_3\}$. Note that we often omit the notion "over N" for brevity/clarity.

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 $\mathcal{NP}$-hard—the number of solutions grows in $\mathcal{O}(|N|^{|N|})$—and there are no known algorithms to solve it in polynomial time. However, the problem has been widely studied, and many different algorithms have been presented to solve it for various settings.

In light of these observations, your **first exercise** is to solve this problem with two different non-optimal (i.e., that return feasible solutions to the problem) algorithms:

- a *random search* algorithm (Algorithm 1), which randomly generates a number of solutions and returns the best it finds; and
- a polynomial time greedy algorithm (Algorithm 2), that works by forming the coalitions that would locally maximize the system's total welfare.

For this purpose, we are using `C++`—however, you don't need to know much of it to complete the lab. In more detail, Algorithm 1 and Algorithm 2 should be implemented in the file `lab-shell-task1.cpp`, which is included in the lab "shell" that you can download from the course's website (https://www.ida.liu.se/ TDDE13/index.en.shtml). Before this, you must also implement the function `csg::evaluate_coalition_structure` according to the coalition structure generation problem's definition. To summarize, you should:

1. implement `csg::evaluate_coalition_structure`;
2. implement `csg::random_search_CSG` (Algorithm 1);
3. implement `csg::greedy_CSG` (Algorithm 2); and
4. benchmark the two algorithms on the problems with prefix `csg` in the folder `/problems`.

You should plot and discuss your results in the report—e.g., what is the solution quality of the different algorithms, and what type of problems are they suitable for?

To **compile** the code and your implementations, use the terminal, and make sure that you are in the folder where you decompressed the lab files. Then use the command `make`. If run successfully, this will create a file called `lab`, which you can use and execute as follows:

```
lab "problem/<problem set filename>"
```

where `<problem set filename>` is one of the the following problem sets (inside the `/problems` folder) that you shall benchmark your algorithms with. The problem sets are:

- `csg-15-npd-1.prob`: 15 agents and normally distributed values;
- `csg-15-npd-2.prob`: 15 agents and normally distributed values;
- `csg-20-npd-1.prob`: 20 agents and normally distributed values; and
- `csg-20-npd-2.prob`: 20 agents and normally distributed values.

Algorithm 1 : `random_search_CSG`$(N, v, k_{samples})$
Output: A coalition structure over N.

$S^* \leftarrow \{\{N\}\}$
for $1, ..., k_{samples}$ **do**
    $S \leftarrow \{\}$
    for $i \in N$ **do**
        $j \leftarrow \mathcal{U}(0, |S|)$ ▷ j gets a random integer in the interval $[0, |S|]$.
        if $j = 0$ **then** $S \leftarrow S \cup \{\{i\}\}$
        else $S[j] \leftarrow S[j] \cup \{i\}$
    if $V(S) > V(S^*)$ **then** $S^* \leftarrow S$
return S^*

Algorithm 2 : `greedy_CSG`(N, v)
Output: A coalition structure over N.

$S \leftarrow \{\}$
for $i \in N$ **do**
    $S^* \leftarrow S \cup \{\{i\}\}$
    for $C \in S$ **do**
        $S' \leftarrow (S \setminus C) \cup (C \cup \{i\})$
        if $V(S') > V(S^*)$ **then** $S^* \leftarrow S'$
    $S \leftarrow S^*$
return S

For implementing Algorithm 1, it can be helpful to generate random numbers pseudo-uniformly. This can be done using the following code:

```
srand(time(NULL)); // Call this once in the start of your function.
int i = rand() % max_value; // Yields an integer in the range [0, max_value).
```

In `C++`, we often use the `vector` to store elements. For example, you can retrieve the i^{th} coalition (represented as a vector of players) from a coalition structure `cstructure` (represented as a vector of coalitions) and add player `p` (represented as an integer) to it as follows:

```
coalition & c = cstructure[i]; // Store a reference to the i:th coalition to c.
c.push_back(p); // Append player p to c (the i:th coalition of cstructure).
```

or alternatively:

```
cstructure[i].push_back(p); // Append p to the i:th coalition in cstructure.
```

The size (cardinality) of a vector (e.g., coalition or coalition structure) can be retrieved with:

```
vector.size(); // Returns vector's size, e.g., number of agents in a coalition.
```

You can evaluate a coalition C with a value function v as follows: `evaluate_coalition(C,v)`.

Exercise 2: Coordinating coalitions

Suppose now 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 T. 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 Christmas together. To make Christmas Eve enjoyable, they have to:

1. clean the house;
2. cook Christmas dinner; and
3. buy and decorate a Christmas tree.

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.

The *assignment problem* revolves around this question of deciding on whom should perform which task. In the following sections, we shall look at two different classes of this problem: one in which we assume only one person is allowed to be assigned to a task, and one in which we assume any number of agents can work together to complete a task.

Prerequisite: The linear assignment problem

In the *linear assignment problem*, we assume that:

1. there's a cost function $c : N \times T \mapsto \mathbb{R}$ 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.

More formally, we want to find a bijection $b : N \mapsto T$ that minimizes:

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

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. A naïve solution to solve it is to evaluate all possible bijections, of which the one with the lowest total cost is deemed *optimal*. However, while there is a factorial number of such bijections (since there are $n!$ permutations of A), there are algorithms that can solve the problem much more efficiently, for example the famous *Hungarian algorithm* (Kuhn, 1953) that solves it in polynomial time.

Simultaneous coalition structure generation and assignment

From an algorithmic perspective, coalition structure generation and assignment are two fundamental coordination processes that are typically treated as two separate paradigms. In cooperative instances for which coordination of multiple coalitions is important, we can improve the quality of our coalitions by taking our goals/tasks into consideration when forming them. One way to do so is to first generate a coalition structure, assign its coalitions to different tasks, evaluate a number of such configurations, and choose the best. If we are to make informed decisions on which configurations to evaluate, this would typically require two different functions for expressing a coalition's "value": one for deciding on which coalitions to form (analogous to the characteristic function in the coalition structure generation problem), and one for assigning/allocating them to goals/tasks (analogous to the cost function in the assignment problem). This is potentially disadvantageous, since it is often complicated to create good utility functions (or to generate realistic performance measures), and it is not necessarily a simple task to predict how the two functions influence the quality of generated solutions.

In light of these observations—and since there are many settings and scenarios in which the utility of a team not only depends on its members and the environment, but also on the task/goal it is assigned to—we define the *simultaneous coalition structure generation and assignment problem* as follows:

Input: A *coalitional game with alternatives* (N, T, v); where N is a set of players (agents), $T = \{1, ..., |T|\}$ is a set of tasks/alternatives (represented as integers), and $v : 2^N \times T \mapsto \mathbb{R}$ corresponds to the value (e.g., potential/expected utility) of assigning a coalition $C \subseteq N$ to a task $t \in T$.

Output: A size-$|T|$ ordered coalition structure CS over N that maximizes CS's total (aggregated) value $V(CS) = \sum_{i=1}^{|T|} v(C_i, i)$.

Your **second exercise** is to solve this problem with two different non-optimal algorithms that are very similar to those you implemented during the first part of this lab:

- a *random search* algorithm (Algorithm 3), which randomly generates a number of solutions and returns the best it finds; and
- a polynomial time greedy algorithm (Algorithm 4), that works by forming the coalitions that would locally maximize the system's total welfare.

You should only need **minor** adjustments to your code from the previous exercise to implement them. These algorithms should integrate coalition-to-task assignment into the formation of coalitions by generating *ordered* coalition structures, for which each possible enumeration of coalitions correspond (bijectively) to a specific assignment of tasks. The ordered coalition structure $\langle C_1, ..., C_{|T|} \rangle$ thus corresponds to assigning the coalition C_i to the task $i \in T$ for $i = 1, ..., |T|$. Hence, you should only generate $|T|$-size coalition structures.

Algorithm 3 and Algorithm 4 should be implemented in the file `lab-shell-task2.cpp`. Before this, you must also implement `scsga::evaluate_ordered_coalition_structure` according to the simultaneous coalition structure generation and assignment problem's definition.

To summarize, you should:

1. implement `scsga::evaluate_ordered_coalition_structure`;
2. implement `scsga::random_search_SCSGA` (Algorithm 3);
3. implement `scsga::greedy_SCSGA` (Algorithm 4); and
4. benchmark the two algorithms on the problems with prefix `scsga` (found in `/problems`).

You should plot and discuss your results in the lab report from benchmarking the algorithms with the following problem sets (inside the `/problems` folder):

- `scsga-15-10-npd-1.prob`: 15 agents, 10 tasks and normally distributed values;
- `scsga-15-10-npd-2.prob`: 15 agents, 10 tasks and normally distributed values;
- `scsga-15-30-npd-1.prob`: 15 agents, 30 tasks and normally distributed values; and
- `scsga-15-30-npd-2.prob`: 15 agents, 30 tasks and normally distributed values.

Algorithm 3 : `random_search_SCSGA`$(N, T, v, k_{samples})$
Output: A size-$|T|$ ordered coalition structure over N.

$S^* \leftarrow \emptyset_{|T|}$ ▷ S^* is initialized to a list of $|T|$ empty coalitions.
for $1, ..., k_{samples}$ **do**
  $S \leftarrow \emptyset_{|T|}$
  for $i \in N$ **do**
    $C \leftarrow$ random element from S
    append agent i to C
  if $S^* = \emptyset_{|T|}$ or $V(S) > V(S^*)$ **then** $S^* \leftarrow S$
return S^*

Algorithm 4 : `greedy_SCSGA`(N, T, v)
Output: A size-$|T|$ ordered coalition structure over N.

$S \leftarrow \emptyset_{|T|}$ ▷ S is initialized to a list of $|T|$ empty coalitions.
for $i \in N$ **do**
  $S^* \leftarrow S$
  for $j = 1, ..., |T|$ **do**
    $S' \leftarrow S$
    $S'[j] \leftarrow S'[j] \cup \{i\}$ ▷ Agent i joins coalition C_1.
    if $S^* = S$ or $V(S') > V(S^*)$ **then** $S^* \leftarrow S'$
  $S \leftarrow S^*$
return S

Note that we use 1-based numbering in our pseudo-code and definitions, while `C++` is zero-indexed—in other words, our notation $S[1]$ corresponds to `cstructure[0]` in `C++`.