1 Matroids

A matroid is a general mathematical object, that was originally axiomatized to provide an abstraction for *greedy algorithms*. Here, we present the axiomatization and propose a transition system which models the algorithm for solving problems with greedy solutions.

We first define a matroid: a matroid over a set S is a set of subsets I of S such that:

1. *Non-degenerate*: The empty set is a member of I
2. *Downward closed*: For any set $S \in I$, all subsets of S are in I.
3. *Exchange*: For any two sets $A, B \in I$ such that $|A| < |B|$, there exists an element $e \in B$, $e \notin A$ (e for *extend*) such that $A \cup \{e\} \in I$.

We presume there exists a *weight function* $w : S \to \mathbb{N}$. We extend this weight function to work on subsets by evaluating the sum of the individual elements:

$$w : 2^S \to \mathbb{N}; \quad w(ss) = \sum_{s \in ss} w(s).$$

Now, the *greedy algorithm for matroids* allows us to solve the following question: For a matroid $M \equiv (S, I)$ and weight function $w : S \to \mathbb{N}$, produce the set:

$$i^\star \equiv \arg\max_{i \in I} w(i).$$

The algorithm is a typical greedy algorithm:

```
def solve_greedy(S, I, w):
  istar = [] # start with empty set
  Ssorted = sort(S, key=lambda i:w(i)) # sort in descending order of weight.
  # greedily pick 'x' as long as it remains independent
  for s in Ssorted: if (istar + [s]) in I: istar.append(s)
  return istar
```

We show how to implement the above algorithm as a transition system.

2 Transition systems

A transition system is a 5-tuple $(X, X_0, U, \to, Y, H)$ where:

1. X is the state space
2. $X_0 \subseteq X$ is the set of potential initial states
3. U is the set of inputs to the system

4. $\rightarrow: X \times U \rightarrow 2^X$ is a non-deterministic transition relation
5. Y is the output space
6. $H : X \rightarrow Y$ is the projection / view from the state to the output space.

3 Matroids as a transition system

We assume that we are given a matroid $M \equiv (S, I)$ and we are attempting to build a transition system that models the algorithm `solve_greedy` as outlined above.

Clearly, in our problem, we have two transition systems that need to be composed together:

1. *`sort`*: Produces the element a_i: the ith element of the list of elements of S ordered in descending order by w.
2. *`choose`*: Decides whether element a_i belongs to $i^\star$.

3.1 `sort` as a transition system

We assume that we already know how to perform sorting as a transition system, since this has been covered in class. We can use the bubble-sort encoding of the transition system.

3.2 `choose` as a transition system

1. The state space is the possible states of $i^\star$, which is subsets of S: $X \equiv 2^S$.
2. The initial state starts with $i^\star$ as empty: $X_0 \equiv \emptyset$.
3. The set of inputs to the system are the elements of S: $U \equiv S$. This models the current $s \in S$ we are attempting to add into $i^\star$.
4. The transition relation decides to add $s \in S$ based on whether $i^\star \cup \{s\} \stackrel{?}{\in} I$:

$$\rightarrow: X \times U \rightarrow X \qquad \rightarrow: 2^S \times S \rightarrow 2^S$$

$$i^\star \xrightarrow{s} \begin{cases} i^\star \cup \{s\} & i^\star \cup \{s\} \in I \\ i^\star & \text{otherwise} \end{cases}$$

5. The output space is the state as the state space: $Y \equiv X = 2^S$
6. The projection function is the identity: $H : X \rightarrow Y; H(i) = i$

3.3 Composition of transition systems: `matroid` $\equiv$ `sort` $\rtimes$ `choose`

Note that we need an adaptor between `sort` and `choose` : one that feeds the output list of `sort` as a sequence of inputs to `choose`. We can define this combined system, `matroid`, as a *composition* of transition systems. We assume that the output space of `sort` is the sequence of sorted elements. We also assume that we have a predicate *sorted*? : $X(\texttt{sort}) \to \texttt{bool}$, which on being fed a state of `sort` tells us if it is sorted or not. We assume that the transition space of `sort` is a single word $U(\texttt{sort}) \equiv \{\texttt{next!}\}$, which moves the sorting algorithm forward.

1. $X \equiv X(\texttt{sort}) \cup (X(\texttt{sort}) \times \mathbb{N} \times X(\texttt{choose}))$. We are either sorting, or we are choosing the ith element of the sorted list.
2. $X_0 \equiv X_0(\texttt{sort})$. We start from the sorting algorithm.
3. $U \equiv \{\texttt{next!}\}$. The algorithm is fully driven: the sorting algorithm was already driven by `next!`. We plug the `sort` automata with the `choose` automata to make `choose` automatically driven as well.
4. The transition relation uses `next!` when we are still sorting. Once we are done sorting, it feeds the sorted list element-by-element to its `choose` state.

$$arr \xrightarrow{\texttt{next!}} \begin{cases} arr' & arr \xrightarrow{\texttt{next!}} arr' \in \texttt{sort} \\ (arr, 0, X_0(\texttt{choose}) = \emptyset) & \textit{sorted}?(arr) = \texttt{true} \end{cases}$$

$$((arr, ix, i^\star)) \xrightarrow{\texttt{next!}} (arr, ix+1, i'^\star); \quad i^\star \xrightarrow{arr[ix]} i'^\star \in \texttt{choose}$$

5. $Y \equiv Y(\texttt{choose}) = 2^S$. The output space is the output of the `choose` automata since it is the final independent set we are interested in.
6. The projection returns the empty set if we are still sorting. If we are done sorting, it returns the current independent set.

$$H(x) \equiv \begin{cases} i^\star & x = (\_, \_, i^\star) \\ \emptyset & \text{otherwise} \end{cases}$$

4 A discussion of alternate designs

note that we don't, in fact, need the fully sorted list. Rather, what we really want is the *sequence* of elements of S ordered by w. It is possible to consider other *streaming* combinators that combine two automata in a streaming fashion, where we do not hold the entire state in memory.

In contrast, the current implementation is a *buffered* implementation, where we buffer the sorted array into memory and then operate on it to build the matroid. It would be interesting to formally implement and compare these systems to each other.