Align-BDD: encoding j-UFLP with BDDs.

Alexey Bochkarev

June 22, 2022

1 TBD: Summary

In this note I discuss a new way to create BDDs for the uncapacitated facility location poroblem (UFLP) that yields more compact diagrams. Moreover, it sort of gives understanding where does the BDD size for UFLPs come from. I would propose to use this as a starting point to design a special class of instances.

I tried some numerical experiments with the type of instances we had (a brief note on that in the very end), but I think we could at least try to do better than that.

Context: We have been constructing BDDs for the CPP using auxiliary, smaller MIPs. It is faster than a naive MIP, but seems to allow very limited sensitivity analsysis. Also, it turned out it is quite often slower than CPP MIP. Therefore, we decided to revert to full BDDs (where one layer = one decision variable, x_i) that would not require solving any auxiliary MIPs. This required a better way to construct BDDs for UFLP, so I think I have found one.

2 Encoding UFLP with BDDs

2.1 Creating a BDD for a given order of variables

Consider a single UFLP instance, which is equivalent to the following MIP:

$$\min \sum_{i=1}^{N} \Big(c_i x_i + f_i(a_i) \Big) \tag{1a}$$

$$\text{s.t. } a_i = \sum_{j \in S_i} x_i \qquad \text{for all } i = 1, \ldots, N, \tag{1b}$$

$$x_i \in \{0, 1\} \qquad \text{for all } i = 1, \ldots, N, \tag{1c}$$

$$\tag{1d}$$

This is the same formulation as before, for a single UFLP.

where S_i is a list of points adjacent to $\textcircled{i}$ (including itself, by assumption). We seek to build a BDD where every layer would represent a single facility location decision, x_i. Technically, each node of the BDD will be associated with a *state* $\sigma = (\sigma_1, \ldots, \sigma_N)$, where $\sigma_i \in \{1, 0, *\}$ represents whether a facility has been located ($\sigma_i = 1$) at $\textcircled{i}$ or not ($\sigma_i = 0$). The third value, $\sigma_i = *$, indicates that the decision on point $\textcircled{i}$ does not affect costs, and hence, does not matter at this moment of the BDD construction, or simply not yet known. Each node state is unique within the respective layer. As we incorporate more decisions and the corresponding costs to the diagram, some previous decisions become irrelevant and the layer width can both increase or decrease. Let us try to find a good order of decisions to keep the BDD size small.

We do not impose any hard constraints, so, nothing can affect *feasibility*, only costs are relevant.

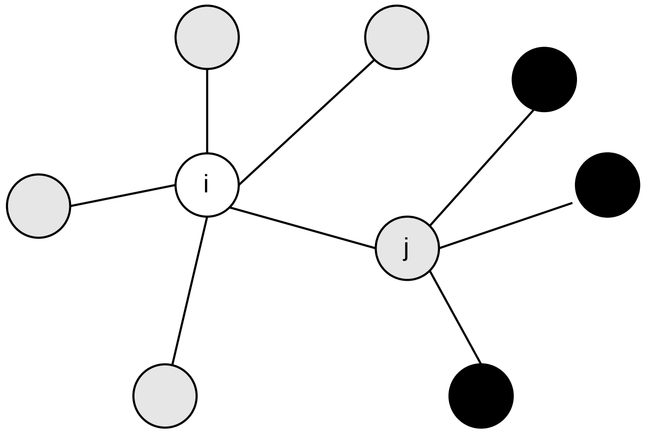

Figure 1: Several points of the original graph.

There are two key observations relevant to the BDD construction. First, we can **incorporate costs** associated with point i, $c_i x_i + f_i(a_i)$, as soon as we incorporate all decisions $x_j \in S_i$. For the situation depicted in Figure 1, we can calculate costs associated with $\textcircled{i}$ only when we incorporate all decisions regarding the gray points and $\textcircled{i}$ itself. After that we can derive x_i and a_i (to compute $f_i(a_i)$ from each BDD node state. However, incorporating all points from S_i is not enough to discard the decision regarding point $\textcircled{i}$ (marking it $*$ in all further node states). For example, the cost associated with point $\textcircled{j}$ in Figure 1 depends on the decisions regarding points $\textcircled{i}$, $\textcircled{j}$, and all the points shaded with dark color. Therefore, we will have to keep the decision on $\textcircled{i}$ in the state until we incorporate all the points depicted in the figure (assuming no other points are present). This is the second key observation regarding the BDD construction:

Any point $\textcircled{i}$ can be marked **"irrelevant"** in the node state (σ_i set to $*$) as soon as we have incorporated the decisions regarding all points $j \in S_i^* = \{j : \ d(i,j) \leq 2\}$ into the diagram.

These two observations yield an algorithm to construct the diagram given an order of points to incorporate, and suggest a way to find a good order of decisions. Let us start with the former.

Algorithm 1 constructs the diagram encoding problem (1), assuming pre-defined order of variables O. For each point $\textcircled{i}$, we sustain two values. First, its *cost uncertainty* Q_i is the number of points we would need to incorporate into the diagram to be able to calculate the cost associated with $\textcircled{i}$. Second, its *value* V_i is the number of points which costs depend on the decision on point $\textcircled{i}$. Every time we incorporate new point $\textcircled{i}$ into the diagram, we decrement Q_j for all points from its neighborhood S_i. After this step we can calculate costs associated with all points j such that $Q_j = 0$. Further, each time we calculate cost for some point j we can decrement V_k for all points k from its neighborhood S_j. Every time some node value V_k drops to zero, we mark this node, $\textcircled{k}$, as irrelevant by setting the corresponding state marker to $*$.

Algorithm 1 create-UFLP-BDD

Input: Graph G, given by $S_i, i = 1, \dots, N$; costs $f_i(\cdot), c_i$ for $i = 1, \dots, N$; points order O.
Output: BDD encoding the problem.

```
 1: $Q_j \leftarrow |S_j|$ for all $j = 1, \dots, N$   // Costs "uncertainty" for each node
 2: $V_j \leftarrow |S_j^*|$ for all $j = 1, \dots, N$   // Node "value".
 3: next-layer $\leftarrow \{(*, \dots, *)\}$   // Root node only.
 4: $k \leftarrow 1$   // Current layer number.
 5: while $k < N$ do
 6:     $i \leftarrow Q_k$   // The next point being incorporated into the diagram.
 7:     $P \leftarrow \varnothing$   // Set of points to calculate costs for.
 8:     decrement $Q_j$ for all $j \in S_i$
 9:     $P \leftarrow P \cup \{j : \ Q_j = 0, \ j \in S_i\}$
10:     decrement $V_j$ for all $j \in \cup_{t \in P} S_t$
11:     current-layer $\leftarrow$ copy(next-layer)
12:     if $k = N$ then
13:         next-layer $\leftarrow \{(*, \dots, *)\}$   // True terminal.
14:     else
15:         next-layer $\leftarrow \varnothing$.
16:     end if
17:     for $\sigma = (\sigma_1, \dots, \sigma_N) \in$ current-layer do
18:         set next-state: $s_k \leftarrow *$ if $V_k = 0$, and $\sigma_k$ otherwise for $k = 1, \dots, N$.
19:         if next-state $\notin$ next-layer then
20:             create node next-state in next-layer
21:         end if
22:         calculate $a_j \leftarrow \sum_{k \in S_j} s_k$ for all $j \in P$
23:         calculate arc cost $C \leftarrow \sum_{j \in P} \left(c_j s_j + f_j(a_j)\right)$
24:         add arc LO($\sigma$) $\leftarrow$ next-state (with cost $C$)
25:         update next-state: $s_i \leftarrow 1$ if $V_i \neq 0$.
26:         if next-state $\notin$ next-layer then
27:             create node next-state in next-layer
28:         end if
29:         recalculate $a_j \leftarrow \sum_{k \in S_j} s_k$ for all $j \in P$
30:         recalculate arc cost $C \leftarrow \sum_{j \in P} \left(c_j s_j + f_j(a_j)\right)$
31:         add arc HI($\sigma$) $\leftarrow$ next-state (with cost $C$)
32:     end for
33:     $k \leftarrow k + 1$
34: end while
```

I think We can claim that given the order, this algorithm gives us the smallest possible diagram:

I need this valid for all costs part: imagine all the costs are just zero. Then, I can encode it as a BDD of width one. Like, whatever I do, I'll have zero costs. That would not capture the graph structure and would seriously restrict my sensitivity analsysis.

Proposition 1. *Given the target order of points, Algorithm 1 produces the smallest possible diagram that is valid for all cost parameters c_i and $f_i(\cdot)$.*

I am not providing a proof now, before we are sure we will need it. But I think the key idea would be that the algorithm gives a lower bound on the number of nodes. This is because we need to differentiate at least between the states at each BDD layer that we assign to different nodes in this algorithm. Otherwise there would be two states associated to the same BDD node, and we could construct a situation when the cost would be wrog.

2.2 Finding a good order of variables.

The procedure we have just introduced provides some insight into the size of the resulting diagram. Assume the size (width) of the current, k-th layer is W_k. Then, width of the next layer will be:

$$W_{k+1} = W_k \times 2^{1-F_k} = 2^{k-\sum_{j=1}^{k} F_j},$$

where F_j is the number of points marked irrelevant after step j of Algorithm 1. Therefore, the logarithm of width will look like the one presented in Figure 2. The top (red) line represents step numbers k, the bottom (blue) one represents cumulative number of nodes marked irrelevant. Both lines start at $(0,0)$ and end at (N,N), and certainly $\sum_{j=1}^{k} F_j \leq k$, because at each step only one point is added, and hence no more than k points can be marked irrelevant after k steps.

This representation suggests a simple greedy algorithm, where we would be trying to mark the next node irrelevant as fast as possible. This means we would always add points from a smallest $S_j^* \setminus X$, where X is the set of points already added to the BDD. We illustrate it with Example 1.

Example 1. *Consider a graph depicted in Figure 4. We summarize the steps we take in Table 1. Rows represent distance-two-neighborhoods of respective points. When we incorporate a point (add it to the BDD), we remove it from further consideration by crossing it out from all the rows of the third column (S_i^*) of the table. We keep track of the number of points left in each subset in the right-most five columns of the table. . The algorithm runs as follows:*

For example, when we incorporate points 1,2,3, and 4 into the BDD, we cross them out from all the rows of that column and update the sizes of the subsets that are left. For subset ③ (row 3) we had 1,2,3,4,5,6. After crossing out 1,2,3,4 we are left with 5,6, which gives size 2 in column $k=5$ (after step five). And so on for other rows.

- *The smallest S_i^* corresponds to point 11 (with the single element, 11). Therefore, we add point 11 and mark it irrelevant immediately.*
- *The next candidate subset is S_1^* (of the smallest size 4, ignoring point 11 now). Hence we add points 1,2,3, and 4, and mark ① irrelevant.*

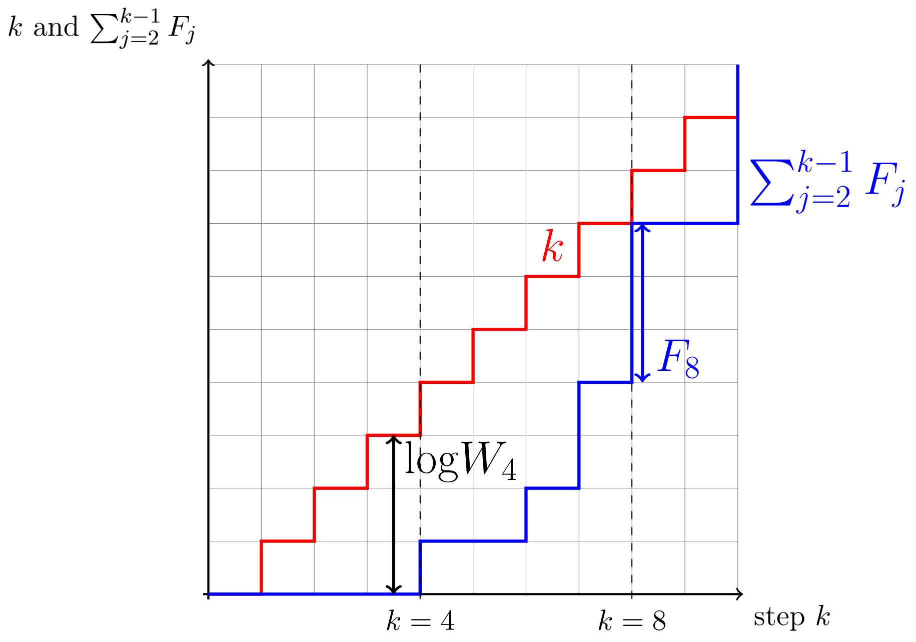

Figure 2: An illustration for the layer width.

- *We update the residual sizes of the length-2 neighborhoods (without elements 1,2,3,4, in addition to 11) in column $k = 5$ and observe that ③ can be marked irrelevant as well (points 1,2,3, and 4 are already added).*
- *The next candidate subset to complete (corresponding to a smallest number in column $k = 5$) is ②. We add points 5 and 6, marking ② as irrelevant.*
- *The next candidate subset corresponds to a smallest number in column $k = 8$. This would be the ones corresponding to ④, ⑤, or ⑥. The first one is ④, so we proceed with incorporating points 7 and 10 into the diagram and marking ④ as irrelevant.*

  We always pick the first subset, as we keep them in a min-heap, keyed by the current size and breaking ties with the subset number.
- *The next candidates correspond to smallest numbers in column $k = 11$, which are rows 5 or 6. Either way we incorporate ⑧ and mark both 5 and 6 as irrelevant. Updated residual length-2 neighborhoods are presented in column $k = 12$.*
- *Finally, we add the last point, ⑨ and mark 7, 8, 9, and 10 as irrelevant, concluding the search for the best order.*

This procedure results in the following solution: 11 | 1,2,3,4 | 5,6 | 7, 10 | 8 | 9 Here the points between the bars can be re-arranged at no additional cost in terms of the BDD size. The corresponding layer width diagram (similar to Figure 2) is presented in Figure 3.

In practice, the presented algorithm provides relatively compact BDD sizes. TBD: In fact, I am almost sure the algorihtm is optimal. Proof

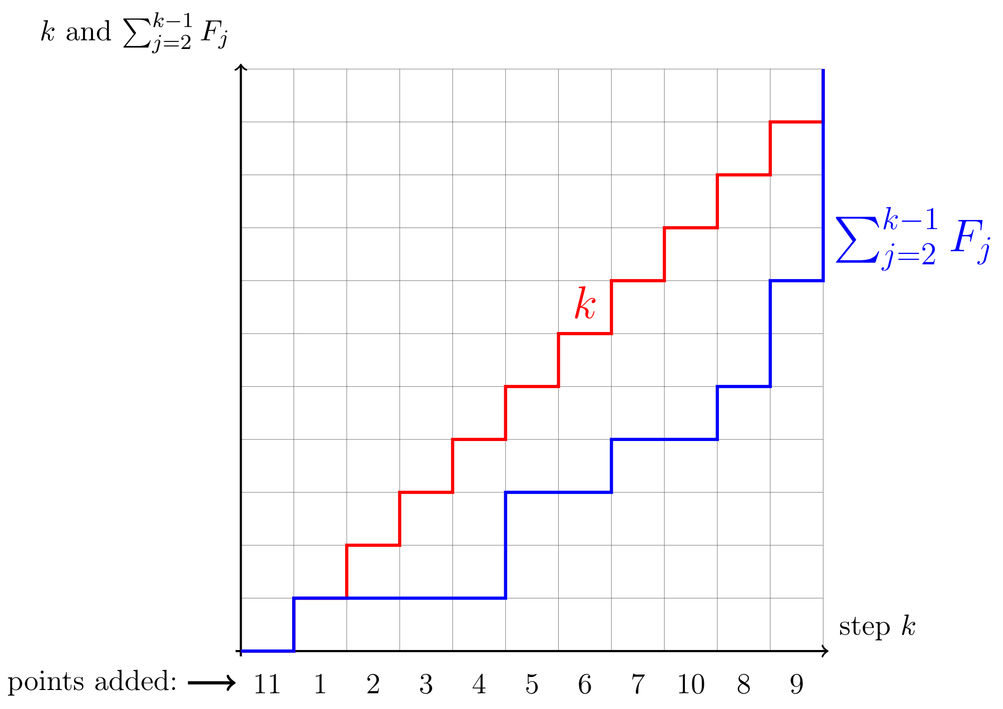

Figure 3: Layer width diagram for the proposed solution.

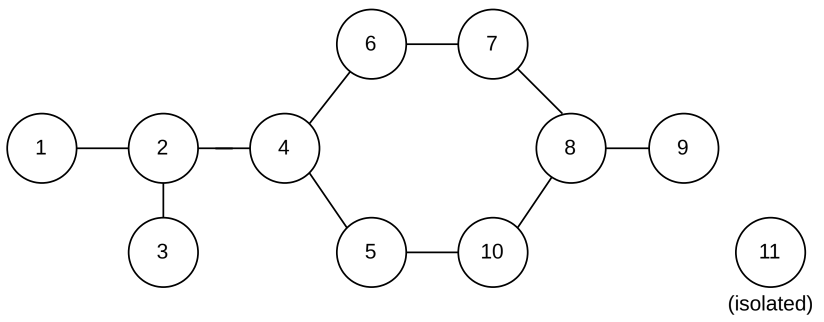

Figure 4: A sample graph G.

sketch: use Proposition 1 to construct a better order. Note that an order is a sequence of subsets corresponding to distance-2-neighborhoods. Pick the first pair of adjacent subsets in the solution such that a greedy algorithm would invert their order, and swap them. I think that looking at a picture similar to Figures 3 one can show that such swap would never make the resulting BDD size worse. Hence, “fixing” all such discrepancies we arrive at a solution that can be obtained with the greedy algorithm. end of TBD

3 TBD: A numerical experiment

I was able to generate a class of instances where the full-DD approach is *somewhat* comparable to the naive MIP and even CPP MIP. In particular,

Table 1: Deriving variable order for the BDD encoding UFLP, for the graph depicted in Figure 4. Highlighted are changes as compared to the previous number.

$\textcircled{i}$	S_i	S_i^*	$\lvert S_i^* \rvert$	$k=1$	$k=5$	$k=8$	$k=11$	$k=12$
1	{1,2}	{1,2,3,4}	4	4	X	X	X	X
2	{1,2,3}	{1,2,3,4,5,6}	6	6	2	X	X	X
3	{2,3}	{1,2,3,4}	4	4	X	X	X	X
4	{2,4,5,6}	{1,2,3,4,5,6,7,10}	8	8	4	2	X	X
5	{4,5,10}	{2,4,5,6,8,10}	6	6	4	2	1	X
6	{4,6,7}	{2,4,5,6,7,8}	6	6	4	2	1	X
7	{6,7,8}	{4,6,7,8,9,10}	6	6	5	4	2	1
8	{7,8,9,10}	{5,6,7,8,9,10}	6	6	6	4	2	1
9	{8,9}	{7,8,9,10}	4	4	4	4	2	1
10	{5, 8, 10}	{4,5,7,8,9,10}	6	6	5	4	2	1
11	{11}	{11}	1	X	X	X	X	X

out of 45 random instances, I had 18 where full-DD was faster than the naive MIP, and 24 – where full-DD was faster than CPP MIP.

This is the 'caveman' type of j-UFLP instances, with $n = 2$ clusters, $M = 11$ points within each one, with sparsity parameter $L = 0.35$. I generate two such UFLP instances and add linking constraints between them (so that all location decisions in one instance affect the decisions in the other one).

But I have a feeling that this result is somewhat fragile and very much dependent on instance parameters. Do you think we could try to design a specific instance class, given our understanding of the algorithm, where our pipeline would be especially beneficial?..