Appendix for AAMAS 2026 Rebuttal – Paper #177

Contents

1 Introduction

This appendix consolidates the extended technical material supporting Reviewer EH5s (R3)'s request for enhanced pseudocode clarity, deeper insight into underlying data structures, and transparency of our implementation. While the main paper presents the core logic of VERA in a concise, space-efficient form, the material below provides a more explanatory and reproducible view of the system:

Section 1 elaborates the full pseudocode with line-by-line explanation, along with the associated data structures, memoization strategy, and DP-pruning rules employed by VERA. These details make explicit how intermediate states are stored, how redundant branches are avoided, and how the recursive structure resolves optimal decisions efficiently.

Section 2 presents the complete formulation and implementation of the OPT() routine—our primary recursive operator. We detail its interfaces, termination conditions, internal invariants, and how it integrates with the broader VERA workflow.

Together, these sections provide the expanded algorithmic transparency requested by R3 and directly align with the GitHub resources that will be released publicly.

2 Section 1: Pseudo Code and it's Textual Description

2.1 Function SolveMCKP()()

Algorithm 3 SolveMCKP(R, providers)

Require: $R \geq 0$; providers (list of provider–tier lists)
Ensure: allocation list; total_profit

```
n ← len(providers)
if n = 0 or R ≤ 0 then
    return ([], 0.0)
end if
// Preprocess tiers
proc_tiers ← [ ]
for each provider p in providers do
    arr ← [ ]
    for each tier in p do
        units ← int(tier.units)
        price ← float(tier.price)
        tid ← int(tier.tier)
        append (units, price, tid, tier) to arr
    end for
    append arr to proc_tiers
end for
IMPOSSIBLE ← −10^18
memo ← {} map (j, d) ↦ (best_value, chosen_idx)
best_val ← IMPOSSIBLE
best_d ← None
for d = 0 to R do
    v ← OPT(n, d, proc_tiers, memo, IMPOSSIBLE)
    if v ≠ IMPOSSIBLE and v > best_val then
        best_val ← v
        best_d ← d
    end if
end for
if best_val = IMPOSSIBLE or best_d = None then
    memo.clear()
    ([ ], 0.0)
end if
Reconstruct allocation
allocation ← [ ]
d ← best_d
for j = n 1 do
    (val, chosen) ← memo[(j, d)]
    if chosen = None then
        continue
    end if
    (units, price, tid, orig) ← proc_tiers[j − 1][chosen]
    append { mec_index : j − 1, tier_id : tid, units :
    units, price : price } to allocation
    orig fields are included when forming the final dictionary in
    code
    d ← d − units
end for
reverse(allocation)
total_profit ← best_val
memo.clear()
(allocation, total_profit)
```

2.1.1 Function Inputs

- Amount surplus resource in the beginning of the round *R*
- List of client Operator Platforms (OPs)
-

2.1.2 Output

When triggered by VERA, the function returns both the maximum attainable welfare and the corresponding optimal allocation of the available resources ***R*** across client OPs.

2.1.3 Line By Line Explanation

The SolveMCKP() procedure begins by determining the number of participating consumer OPs (Line 1) and immediately returns an empty allocation when either no OPs exist or the available resource capacity is zero

(Lines 2–4). It then preprocesses every OP's cumulative bid tiers into a lightweight tuple form (Lines 5–16), normalizing the units, valuations, and tier identifiers so that the subsequent recursion can operate efficiently. After initializing a large negative sentinel to represent infeasible DP states and creating the memoization table that will store optimal subproblem results (Lines 17–18), the procedure prepares to search across all exact capacities from 0 to R (Lines 19–20). For each such capacity d, it invokes the recursive OPT(n,d), which computes the best welfare achievable using all OPs under an exact remaining capacity d (Line 22). Whenever a better feasible welfare is obtained, both the best value and the corresponding capacity are updated (Lines 23–26). If no feasible allocation exists across any capacity, the function clears the memo table and exits (Lines 28–31). Once the best exact-capacity solution is identified, the procedure reconstructs the final welfare-maximizing allocation by tracing backward from provider nnn to 111 (Lines 32–44). At each step, it reads from the memoized records to determine whether the current OP was skipped or which tier was chosen (Lines 36–39), appending the selected tier's information while reducing the remaining capacity accordingly (Lines 40–43). After reversing the reconstructed list to restore natural OP order (Line 45), the procedure assigns the total welfare (Line 46), clears the memo table for safety (Line 47), and returns the final allocation and welfare pair (Line 48). Thus, SolveMCKP() serves as the global welfare maximizer, identifying both the optimal set of tiers and the maximum achievable social welfare.

2.2 Function OPT()

Algorithm 4 OPT(j, d, *proc_tiers*, *memo*, *IMPOSSIBLE*)

Require: $0 \le j \le n, 0 \le d \le R$
Ensure: best_value for first j providers with exact capacity d

```
key ← (j, d)
if key ∈ memo then
    memo[key].value
end if
if j = 0 then
    if d = 0 then
        memo[(0,0)] ← (0.0, None)
        0.0
    else
        memo[(0,d)] ← (IMPOSSIBLE, None)
        IMPOSSIBLE
    end if
end if
Option A: skip provider j − 1
best ← OPT(j − 1, d, proc_tiers, memo, IMPOSSIBLE)
chosen ← None
Option B: choose one tier from provider j − 1
for each (t_idx, (units, price, tid, orig)) in proc_tiers[j − 1] do
    if units ≤ d then
        prev ← OPT(j − 1, d − units, proc_tiers, memo, IMPOSSIBLE)
        if prev ≠ IMPOSSIBLE then
            val ← prev + price
            if val > best or (val = best and (chosen = None or t_idx < chosen)) then
                best ← val
                chosen ← t_idx
            end if
        end if
    end if
end for
memo[key] ← (best, chosen)
best
```

2.2.1 Function Inputs

- j → represents the number of consumer Operator Platforms (OPs) considered in the current subproblem. This function computes the maximum welfare achievable using only the first j OPs. This creates the DP structure where:
    - Decreasing j explores smaller prefixes, and
    - Base case j=0, corresponds to "no OP available"

  Role in DP: It controls which OP is being evaluated at this recursion level, enabling the options:
    - Skip OP j-1, or
    - Pick exactly one of OP j-1's tiers
- d → It denotes the exact amount of resource capacity available in the current subproblem. OPT() must compute the best welfare achievable while filling exactly d units of resources.
  This reflects the exact-fill semantics of classical MCKP, consistent with Equation (4) in the paper.
  Role in DP:
    - Determines which tiers are feasible (only those with units ≤d).
    - Guides recursive calls
    - Base case OPT(0,d) is feasible only if d=0, infeasible otherwise

- proc_tiers → Preprocesse tier structure. It is a list of lists storing the tiered bids in a normalized, computation-friendly structure.
  For each provider *I*, its entry is: [(units,price,tier_id,orig),…]
  Why needed:
  - DP recursion repeatedly access tier units and valuations
  - Preprocessing ensures integer units, floating valuations, and deterministic tier IDs
  - Avoid heavy dictionary lookups inside the hot loop, improving performance

  Role in DP: Used to iterate over all feasible tiers of OP j-1, when exploring Option B (choosing tier)
- memo → Memoization table. It stores already-computed DP states to avoid exponential recomputation. A typical entry:
  memo[(j,d)]=(best_value,chosen_tier_index)
  Why necessary:
  - Converts exponential recursion into pseudo-polynomial DP (O(n.R.k) complexity).
  - Records not only the best value but also the chosen tier, enabling reconstruction in the top-level function.

  Role in DP:
  - On entering a state, OPT first checks if (j,d) is memorized
  - On exiting a state, OPT writes the computed results before returning
- IMPOSSIBLE – Sentinel for Infeasible States. This variable represents the DP states that cannot be satisfied under exact-fill rules. Typically set to a large negative constant (e.g.. -10^{18}).
  Why needed:
  - Avoids the use of negative infinity, which is slower in Python float comparisons
  - Prevents accidental acceptance of infeasible paths during max-comparisons
  - Ensures base conditions match the MCKP recurrence:
    - OPT(0,0)=0
    - OPT(0,d>0)=IMPOSSIBLE

  Role in DP: It is used to:
  - Flag subproblems with insufficient capacities
  - Exclude invalid choices from max comparisons
  - Maintain correctness of recurrence.

2.2.2 Output

The OPT function returns the maximum welfare achievable using the first jjj OPs under exact capacity ddd

2.2.3 Line By Line Explanation

Complementing SolveMCKP(), the recursive `OPT(j,d)` function embodies the dynamic-programming core of the Multiple-Choice Knapsack formulation. It begins by forming the DP state key (Line 1) and immediately returns a memoized value if the state has been previously computed (Lines 2–4). The base case occurs when no OPs remain; here the function returns a feasible zero only when the remaining capacity is also zero, and an infeasible sentinel otherwise (Lines 5–12). For non-base cases, `OPT` first evaluates the option of skipping OP j−1, recursively inheriting the best welfare obtainable for the same capacity (Lines 13–16). It then considers the alternative of selecting exactly one of OP j−1's feasible tiers (Lines 17–18). For each tier whose demand does not exceed the remaining capacity (Line 19), the function computes the welfare achievable by combining that tier's valuation with the optimal welfare of the reduced subproblem OPT(j−1,d−$a_{j\alpha}$) (Lines 20–21). Any feasible choice improves the running best value, and ties are broken deterministically to ensure reproducibility (Lines 22–27). After exploring all candidate tiers, the function stores the optimal value along with the chosen tier index in the memo table (Line 29),

ensuring that the driver can later reconstruct the allocation without recomputation, and finally returns the best welfare obtained for state (j,d) (Lines 30–31). In doing so, `OPT` directly implements the recursive structure of the MCKP recurrence, faithfully capturing the optimal social welfare obtainable from the first j OPs under an exact remaining capacity of d.

3 Section 2: CAMARA compliant VERA API's Endpoints

Path	Verb	Purpose	Request	Response	Role
/mecs	POST	Register MEC with capability profile	{name, location, capacityProfile}	{mecId, status}	FRA
/mecs	GET	List registered MECs	-	{mecId, name, capacityProfile}	FRA
/mecs/{mecId}	GET	Inspect MEC profile	-	{mecId, profile}	FRA/MEC
/mecs/{mecId}	DELETE	Deregister MEC (if no commitments)	-	{status}	FRA/MEC
/rounds	POST	Create an allocation round (params)	{capacity, policy, start, end}	{roundId, status}	FRA
/rounds	GET	List rounds / statuses	-	{roundId, status, policy}	FRA/MEC
/rounds/{roundId}	GET	Get round metadata	-	{roundId, status, capacity}	FRA/MEC
/rounds/{roundId}:lock	POST	Deterministic close of bidding	-	{roundId, status=LOCKED}	FRA
/rounds/{roundId}:run	POST	Trigger allocation solver (async op)	-	{opId}	FRA
/bids	POST	Submit a tiered bid for a round	{mecId, roundId, tiers:[{tier, units, price}]}	{bidId, status}	FRA
/bids?roundId={id}	GET	List bids filtered by round	-	{bidId, mecId, tiers}	FRA/MEC (filtered)
/bids/{bidId}	PATCH	Update bid while round OPEN	partial bid	{bidId,status}	MEC
/bids/{bidId}	DELETE	Withdraw bid while OPEN	-	{status}	MEC
/allocations	POST	List allocations (FRA:all; MEC:filtered)	-	{allocationId, assignments, payouts}	FRA/MEC
/allocations/{allocationId}	GET	Inspect allocation details	-	{assignment, payouts, totalValue}	FRA/MEC
/allocations/{allocationId}:settle	POST	Finalize settlement for allocation	-	{allocationId, settled}	FRA

The table given above contains the details of the 16 end points of the VERA API, that we have been made keeping in mind that it is CAMARA compliant. We have used FsatAPI, OpenAPI 3.1.1, and other standardization techniques that are used for the development of any CAMARA compliant API. From security perspective we have implemented OAuth 2.0 authorization along with JWT authentication.