# TIME-DEPENDENT KNAPSACK PROBLEM

GROUP 2

# BACKGROUND

Time-dependent knapsack problem (TDKP) with irregular availability is an advanced variant of the classic 0/1 knapsack problem

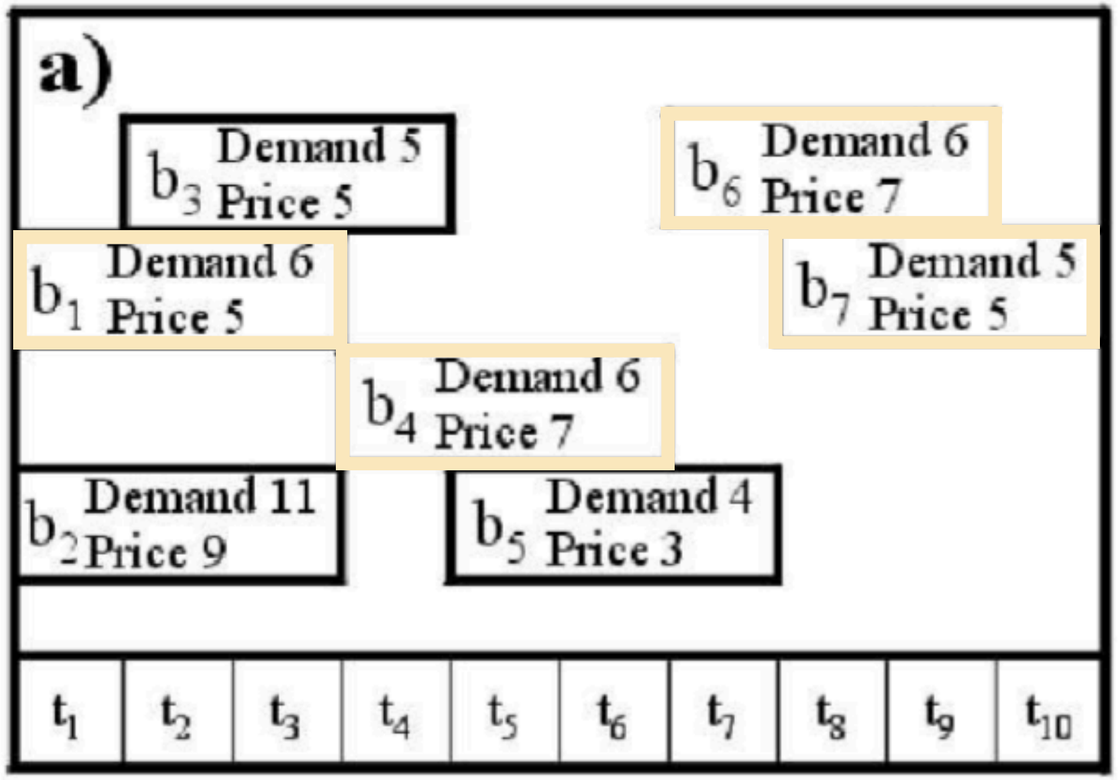
# BACKGROUND

Given a set of items, each with a price, demand, and time window it can be in the knapsack

Maximize the total price of the items such that the sum of the demands doesn't exceed the capacity of the knapsack at any given time.

# SAMPLE PROBLEM

**Maximum capacity: 10**




(Bartlett et al., 2005)

# BACKGROUND

As the 0/1 knapsack problem is NP-hard, the Time-dependent knapsack problem is also NP-hard.

The 0/1 knapsack problem can be solved in pseudo-polynomial time using dynamic programming, but its DP formulation doesn't work when items have to be removed from the knapsack.

# METHODS IMPLEMENTED AND COMPARED

**A. Brute Force (Baseline)**

- Tries all $2^n$ combinations.
- Good for small n only.
- We used this as the baseline to check correctness.

**B. Simulated Annealing — CPU Version**

- The first heuristic method we built.
- Uses randomization to transition between knapsack states.
- Runs multiple independent instances to increase the chances of finding better solutions.
- First-ever use of the heuristic on the problem, to our knowledge.
- Uses generic hyperparameters from CP-Algorithms.

# METHODS IMPLEMENTED AND COMPARED

**C. Simulated Annealing — GPU CUDA (Parallel)**

- We parallelized:
  - Many SA instances (400)
  - Each instance consists of multiple worker threads (32), which parallelize certain computations.
- Main idea:
  - CPU SA = 1 SA chain at a time, can be extended to tens through multithreading
  - CUDA SA = hundreds of SA chains at once
- Optimizations:
  - Page creation and prefetching
  - Memory coalescing so that one cacheline can serve multiple SA instances.
  - Worker threads write demand change events (if an item is selected) of an instance to shared memory in parallel.
  - Single-threaded demand accumulation via prefix array-style computation.

# METHODS IMPLEMENTED AND COMPARED

**C. Simulated Annealing — GPU CUDA (Parallel)**

- Optimizations:
  - Kernel synchronization to alternate between single-threaded and multithreaded portions.
  - Allow knapsack items to exceed knapsack limits with a penalty (configurable hyperparameter) instead of explicitly correcting to avoid branching.
  - Correcting violations in the final set of items is still needed.
    - Performed in the CPU (using CPU baseline) due to excessive branching.

**D. Linear Programming (LP) as a benchmark**

- Implementation uses OR-Tools, a free and open-source linear programming library by Google.
- Used as the ground-truth "best possible answer," given that OR-Tools is a mature platform.
- Helps us measure how close our heuristics are to the values from other solvers.

# OTHER ATTEMPTS

**Dynamic Programming (Exact)**

- Maximizes the price of items in the knapsack given the set of items in the knapsack at the current point in time.
- The DP version of this problem still ends up exponential in time, as there is a need to explore all possible sets of items for each time point to guarantee correctness.
- Unless there exists a non-exponential DP formulation, it behaves like brute force in the worst case.
- However, it is still better if the number of items that could occupy each point in time is small, or if the time interval of each item is fixed, but such constraints aren't guaranteed.
- Because of this risk, exact DP was not practical or scalable for our project.

# RESULTS

## TIME (MS)

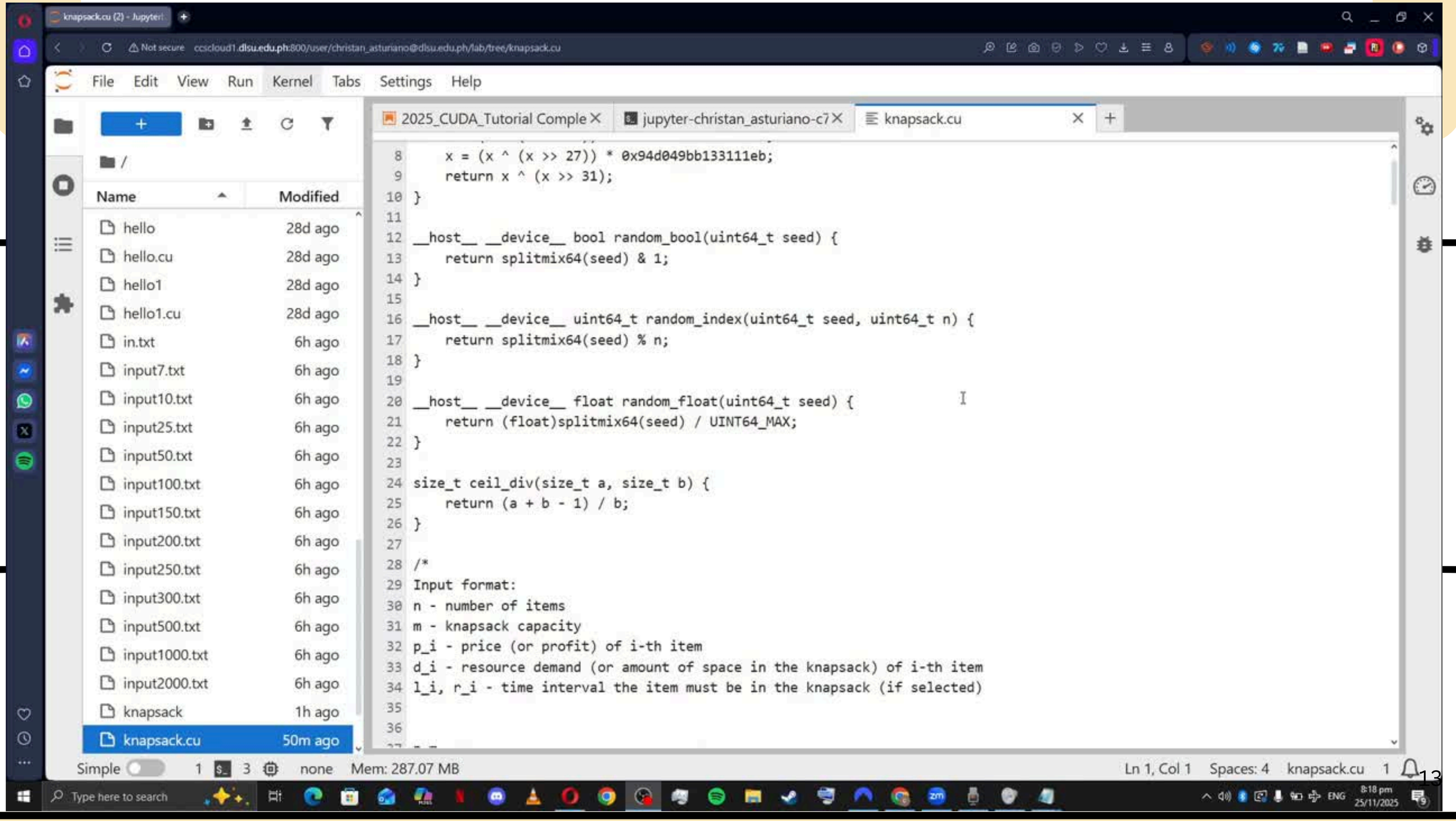
| n | SA (CPU) | SA (GPU) | SPEEDUP |
|---|---|---|---|
| **10** | 2008.344 | 15.83418 | **126x** |
| **25** | 3246.888 | 18.44622 | **176x** |
| **50** | 6652.54 | 25.00182 | **266x** |
| **100** | 9588.222 | 35.5514 | **270x** |
| **200** | 16804.56 | 57.66832 | **291x** |
| **250** | 23425.46 | 70.99128 | **330x** |
| **500** | 37419.44 | 115.0966 | **325x** |
| **1000** | 69236.3 | 195.242 | **355x** |
| **2000** | 123274.2 | 313.7148 | **393x** |

# OUTPUT

| n | LP | SA (CPU) | SA (GPU) |
|---|---|---|---|
| **10** | 104 | 104 | 104 |
| **25** | 281 | 281 | 281 |
| **50** | 613 | 613 | 613 |
| **100** | 1188 | 1188 | 1188 |
| **200** | 2574 | 2553 | 2553 |
| **250** | 3100 | 2990 | 2990 |
| **500** | 6332 | 5601 | 5601 |
| **1000** | 12462 | 9077 | 9077 |
| **2000** | 24903 | 13662 | 13662 |

RESULTS

# PROJECT DEMO


knapsack.cu (2) - Jupyter
Not secure ccscloud1.dlsu.edu.ph:800/user/christan_asturiano@dlsu.edu.ph/lab/tree/knapsack.cu
File Edit View Run Kernel Tabs Settings Help
2025_CUDA_Tutorial Comple
jupyter-christan_asturiano-c7
knapsack.cu
/
Name
Modified
hello 28d ago
hello.cu 28d ago
hello1 28d ago
hello1.cu 28d ago
in.txt 6h ago
input7.txt 6h ago
input10.txt 6h ago
input25.txt 6h ago
input50.txt 6h ago
input100.txt 6h ago
input150.txt 6h ago
input200.txt 6h ago
input250.txt 6h ago
input300.txt 6h ago
input500.txt 6h ago
input1000.txt 6h ago
input2000.txt 6h ago
knapsack 1h ago
knapsack.cu 50m ago
x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
return x ^ (x >> 31);
}
__host__ __device__ bool random_bool(uint64_t seed) {
return splitmix64(seed) & 1;
}
__host__ __device__ uint64_t random_index(uint64_t seed, uint64_t n) {
return splitmix64(seed) % n;
}
__host__ __device__ float random_float(uint64_t seed) {
return (float)splitmix64(seed) / UINT64_MAX;
}
size_t ceil_div(size_t a, size_t b) {
return (a + b - 1) / b;
}
/*
Input format:
n - number of items
m - knapsack capacity
p_i - price (or profit) of i-th item
d_i - resource demand (or amount of space in the knapsack) of i-th item
l_i, r_i - time interval the item must be in the knapsack (if selected)
Simple 1 3 none Mem: 287.07 MB
Ln 1, Col 1 Spaces: 4 knapsack.cu 1
Type here to search
ENG 8:18 pm 25/11/2025

# DISCUSSION

- Our implementation can obtain the global maximum when tested with small sample sizes (n=5, 26), where the global maximum is computable in a reasonable amount of time through brute-force.
- Our implementation remains competitive with OR-Tools when tested with moderately high n ($n \leq 1000$).
  - This mirrors the findings of Hashim et al. (2024) on 0/1 knapsack that simulated annealing's performance decreases for larger sample sizes.
- Future work can explore combining our optimized simulated annealing with other heuristics, such as linear programming (hybrid SA-LP).
  - Explored in other problems, such as coordinating directional overcurrent relays (Kida et al., 2020), and beam angle in radiation therapy (Bertsimas et al., 2012)

# THANK YOU