Decision Diagram-Based Approach to 0/1 Knapsack Problem

2024학년도 겨울학기 Logistics Lab 연구참여

지도 교수	**김병인**
사 수	**나진우**
성 명	**권순영**
학 번	**20220742**
학 과	**산업경영공학과**
기 간	**2024/12/30 ~ 2025/02/10**

목차

1. 서론

본 과제에서는 조합 최적화의 대표적인 문제 중 하나인 0/1 Knapsack Problem 을 해결하는 데에 있어 Decision Diagram(DD) 방법론을 적용했다. 최적화보다 논리 회로의 구성이나 디지털 시스템 설계에 사용됐던 DD 의 개념은 David Bergman, Willem-Jan van Hoeve, 등 다양한 연구자의 도전으로 확장됐다. 이들의 연구를 바탕으로 집필된 <Decision Diagrams for Optimization>은 DD 를 최적화에 활용하기 위한 기초 개념 정립, 다양한 최적화 문제를 바탕으로 한 활용 예시, 그리고 후속 연구 및 발전을 위한 토대를 만든다. 본 과제는 위 서적을 바탕으로 하였으나, 서적에서는 참고하지 않는 문제인 0/1 Knapsack Problem 에 DD 방법론을 적용했다.

과제의 목표에는 크게 세 가지가 있었다:

1) 첫째, 서적에서 다루는 주제 중 일부를 구현함으로써 실제 문제 적용에서 고려할 사항을 실험적으로 확인한다.

2) 둘째, 0/1 Knapsack Problem 의 다양한 test case 를 확인하여 DD 방법론의 장단점 및 유용하게 활용될 문제 상황을 정리한다.

3) 셋째, 기존의 최적화 방법론(LP Relaxation, 등)과 비교를 하여 DD 방법론의 실용성을 실험적으로 확인한다.

본 과제는 DD 방법론이 조합 최적화에 있어 어떤 역할을 가질 수 있는지 확인하는 과정이다. 이를 확인하기 위해 크게 네 가지 용도의 DD 를 확인한다: Exact DD, Relaxed DD, Restricted DD, Branch-and-Bound Algorithm(General Solver). 또한, 각 용도에 있어 DD 의 실용성에 큰 영향을 주는 요인을 실험적으로 확인하여 결과 및 해석을 정리하였다.

위 서적에서 자주 강조되는 내용은 DD 방법론의 robustness 이다. 다양한 문제에 적용이 가능한 방법론이자, 다른 최적화 방법론과 결합하여 이를 보강할 수 있는 도구라 제안되지만, 실제로 이를 분석하는 과정은 소개되지 않으며 크게 직관적이지도 않다. 직접 DD 방법론을 적용해보며, 다른 방법론과 비교하고, 실험적으로 확인하는 과정을 통해 DD 방법론이 어떤 문제 상황에 강한지, 어떠한 조합 최적화 문제를 접근할 때 DD 방법론을 적용하는 방법, 그리고 이러한 robustness 를 직접 확인하고자 기존의 주제에서 확장된 연구 주제를 선정했다.

본 과제를 통해 DD 방법론을 사용하고자 하는 공학도에게는 적용 방법에 대한 기초적인 직관을, 그리고 더욱 심층적으로 연구하고자 하는 산업공학도가 활용할 수 있는 기초적인 토대를 만들고자 한다.

2. 이론적 배경

2.1 Decision Diagram

Decision Diagram(DD)의 개념은 switching circuit 과 Boolean logic 에서부터 비롯된다. Lee(1959)[1]에서 binary-decision program 의 개념이 처음 제안되며 node 를 통한 decision representation 이 처음 제시되었다. 이에서 나아가, Akers(1978)[2]과 Bryant(1986)[3]의 연구를 통해 data structure 이자 graphical representation 의 형태를 가진 DD 가 제시되었다. 다양한 Boolean function 을 나타내는 도구로서 DD 는 지금까지 사용되고 있다.

최적화에서의 DD 는 solution space 에 대한 representation 으로 처음 제시되었다. Solution counting 의 도구로 처음 사용됐으나, branch-and-bound scheme 을 사용한 Lai, Pedram and Vrudhula(1994)[4]의 연구나 relaxation 의 개념을 적용한 Behle 의 연구, 등을 통해 점차 다양한 용도를 가지게 되었다.

최근 조합 최적화에서 제일 활발하게 활용되는 형태의 DD 이자, 0/1 Problem 에 제일 적합한 형태는 reduced ordered binary decision diagram(RO-BDD)이다. 0/1 Problem 에 대한 canonical representation 이자, 다양한 최적화 방법론을 적용할 수 있으며 현실적으로 구현 가능한 데이터 구조다.

2.1.1 Exact Decision Diagram

Exact Decision Diagram 은 어떠한 조합 최적화 문제의 solution space 를 완전하게 나타내는 DD 의 일종이다. Exact DD 의 root node 로부터 terminal node 까지 가는 path 는 하나의 solution 에 대응되며, Exact DD 의 경우에는 root node 로부터 terminal node 로 향하는 모든 path 의 set 은 set of all feasible solutions 과 동일하다.

Exact DD 의 layer 은 각각 하나의 variable 을 나타낸다고 보면 직관적으로 이해할 수 있다. Layer 을 거듭하며 constraint 에 따라 가능한 feasible solution 들이 표현되며, construction 이 끝난 이후에 root node 과 terminal node 를 연결하는 path 중 shortest path, 또는 longest path 를 구하여 최적해를 구할 수 있다.

Fig. 1 에서 이러한 Exact DD 의 기본적인 모습을 확인할 수 있다.

[1] C. Y. Lee. Representation of switching circuits by binary-decision programs. Bell Systems Technical Journal, 38:985–999, 1959.
[2] S. B. Akers. Binary decision diagrams. IEEE Transactions on Computers, C-27:509–516, 1978.
[3] R. E. Bryant. Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers, C-35:677–691, 1986.
[4] Y.-T. Lai, M. Pedram, and S. B. K. Vrudhula. EVBDD-based algorithms for integer linear programming, spectral transformation, and function decompo sition. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 13:959–975,1994.

Item	*Profit*	*Weight*
1	8	3
2	7	3
3	6	4
4	14	6
Capacity: 6		

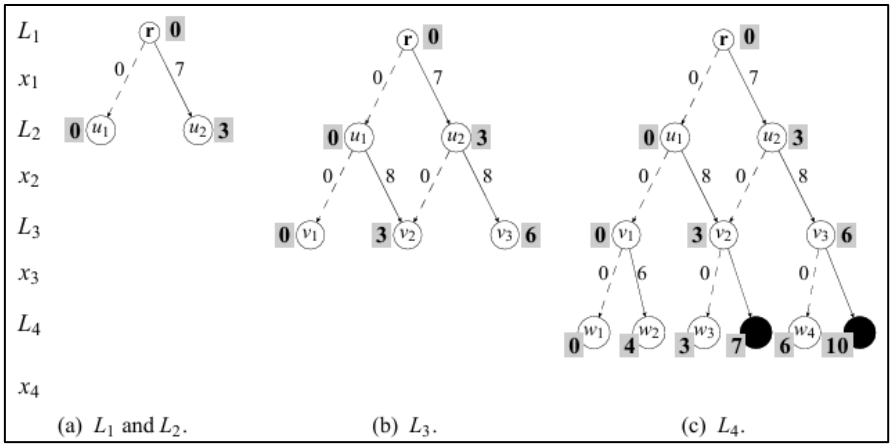

***Fig. 1.1**: Exact DD representation of a knapsack problem (Reference from Bergman et al.)*

Exact DD 를 만드는 과정은 조합 최적화 문제의 dynamic programming(DP) formulation 을 활용한다. DP formulation 의 recursive 한 특성을 사용하여, Exact DD 의 연속된 두 layer 간의 관계를 state transition function 으로 나타낸다. 따라서 Exact DD 의 construction 과정은 DP formulation 을 통한 state space, transition function, transition cost function, 그리고 root value 의 정의에 따라 top-down 으로 이루어진다. 위 4 개의 요소는 문제에 따라 formulation 을 진행해야 한다. 또한, formulation 의 방식에 따라 DD 의 효율이 극도로 좌지우지 됨을 주의할 필요가 있다. 이러한 Exact DD 의 top-down construction 에 대한 pseudocode 는 Algorithm 1 에서 확인할 수 있다.

Algorithm 1 Exact DD Top-Down Compilation

```
Create node r = r̂ and let L_1 = {r}
for j = 1 to n do
    let L_{j+1} = ∅
    for all u ∈ L_j and d ∈ D_j do
        if t_j(u,d) ≠ Ô then
            let u' = t_j(u,d), add u' to L_{j+1}, and set b_d(u) = u', v(u,u') = h_j(u,u')
Merge nodes in L_{n+1} into terminal node t
```

Top-down construction 말고 constraint propogation 을 바탕으로 한 Exact DD 생성 알고리즘도 서적에서는 제시되나, DP formulation 이 존재하는 0/1 Knapsack problem 의 경우에는 활용할 필요가 없다고 판단했다.

매우 직관적으로 받아들일 수 있는 부분은 Exact DD 의 비효율성이다. 결론적으로는 solution space 에 대한 full counting 의 일종이기 때문에 크게 실용적이지 않을 뿐더러, state explosion 에 매우 취약하다. 그러나, exact 한 optimum solution 을 제공한다는 점에서는 특징이 확고한 방식이고, construction 중 같은 layer 에서 동일한 state 가 자주 발생할 경우 메모리 효율성과 computational time 모두 향상시킬 수 있다. 이는 problem structure 에 따라, 그리고 해당 problem 의 test case 에서도 적합성을 확인한 후 사용되어야 한다.

2.1.2 Relaxed Decision Diagram

Relaxed Decision Diagram 은 Exact DD 와 유사하지만 조합 최적화 문제의 relaxation 을 나타내며, 이러한 relaxation 으로 인해 Exact DD 와는 다르게 몇 개의 path 는 infeasible solution 을 나타내기도 한다. 이에 따라 Relaxed DD 는 solution 에 대한 upper bound 를 제공하며, solution space 에 대한 superset 를 나타낸다.

Relaxed DD 에서 이러한 relaxation 을 행하는 방법은 node merging 이다. Construction 이전에 width parameter 을 설정한 뒤, 특정 layer 의 node 수가 width 를 초과하면 node 일부를 병합한다. 이러한 과정을 통해 node 의 전체적인 개수는 줄어들지만, 가용 가능한 path 의 수는 늘어나며 infeasible solution 이 발생할 수도 있다. Node 의 수가 줄어듦으로서 computational time 을 줄일 수 있고, node merging 의 효율적인 구현을 통해 Exact DD 보다 빠르게 construction 과정을 마칠 수 있다. Exact DD 와 Relaxed DD 의 비교는 Fig. 2 에서 확인할 수 있다.

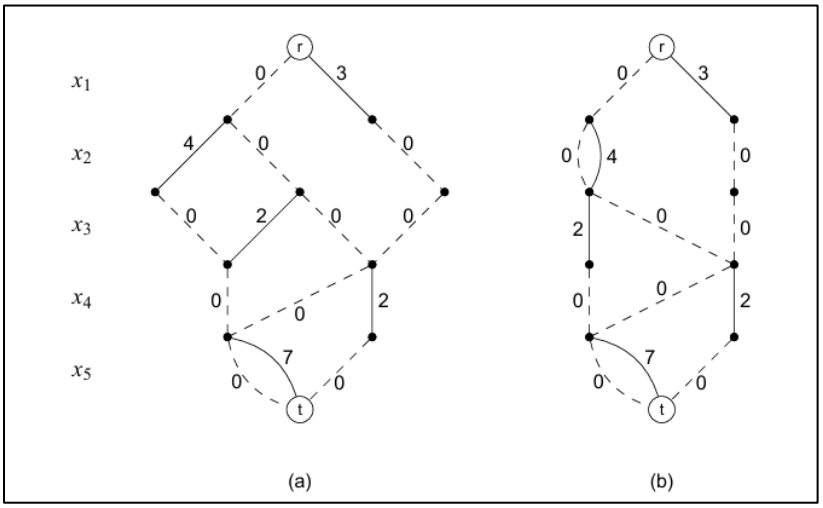

Fig. 2: Comparison of an Exact DD and Relaxed DD representation.

Relaxed DD 는 Exact DD 의 기본적인 구조를 활용하지만, 추가적으로 설정할 부분이 세 가지 있다. 첫째, 위에서 언급한 width parameter 이다. 해당 변수는 곧 relaxation 의 aggresiveness 를 의미하기도 하며, width 를 줄일수록 optimal path 가 relax 될 확률은 늘어나지만 computational time 을 줄일 수 있다. 반대로, width 를 높이면 exact representation 에 가까워지지만 구현의 효율성은 떨어진다. 둘째, node merging 에서 어떤 node 를 merge 할지 선택하는 node select heuristic 이 있다. 구현 방식에는 여러 가지가 있지만, optimal path 를 되도록 relaxation 하지 않는 게 bound quality 에 좋으므로, 문제의 state formulation 에 따라 suboptimal 한 node 를 merge 하도록 구현하면 성능이 올라간다. 셋째, node merge 를 어떻게 할 것인지 설정하는 node merge operator 이 있다. 해당 연산자는 사용자가

설정한 state formulation 에 따라 적절하게 정의하면 된다. 이러한 요소들이 결합된 Relaxed DD 의 construction 과정을 나타낸 pseudocode 를 Algorithm 3 에서 확인할 수 있다.

Algorithm 3 Relaxed DD Top-Down Compilation for Maximum Width W

Create node $r = \hat{r}$ and let $L_1 = \{r\}$
for $j = 1$ to n **do**
  while $|L_j| > W$ **do**
    let $M = \text{node\_select}(L_j)$, $L_j \leftarrow (L_j \setminus M) \cup \{\oplus(M)\}$
    for all $u \in L_{j-1}$ and $i \in D_j$ with $b_i(u) \in M$ **do**
      $b_i(u) \leftarrow \oplus(M)$, $v(a_i(u)) \leftarrow \Gamma_M(v(a_i(u)), b_i(u))$
  let $L_{j+1} = \emptyset$
  for all $u \in L_j$ and $d \in D(x_j)$ **do**
    if $t_j(u,d) \neq \hat{0}$ **then**
      let $u' = t_j(u,d)$, add u' to L_{j+1}, and set $b_d(u) = u'$, $v(u,u') = h_j(u,u')$
Merge nodes in L_{n+1} into terminal node t

2.1.3 Restricted Decision Diagram

Restricted Decision Diagram 은 Exact DD 와 유사하지만 조합 최적화 문제에 대한 primal heuristic 을 나타내며, restriction 과정으로 인해 손실되는 path 로 인해 모든 path 가 feasible path 이지만 기존의 optimal path 가 없어질 수도 있다. 이에 따라 Restricted DD 는 solution 에 대한 lower bound 를 제공하며, solution space 에 대한 subset 를 나타낸다.

Restricted DD 에서는 node exclusion 이 활용된다. 기존 Relaxed DD 에서는 node merging 을 통해 path 를 유지한 반면, node exclusion 에서는 node 와 함께 이를 지나는 모든 path 를 지운다. Relaxed DD 와 마찬가지로 width parameter 을 넘을 때, 해당 layer 에서 node 를 지우며 이를 지나는 path 가 모두 지워진다. Node 의 수가 줄어듦으로서 computational time 을 줄일 수 있다. Exact DD 와 Restricted DD 의 비교는 Fig. 3 에서 확인할 수 있다.

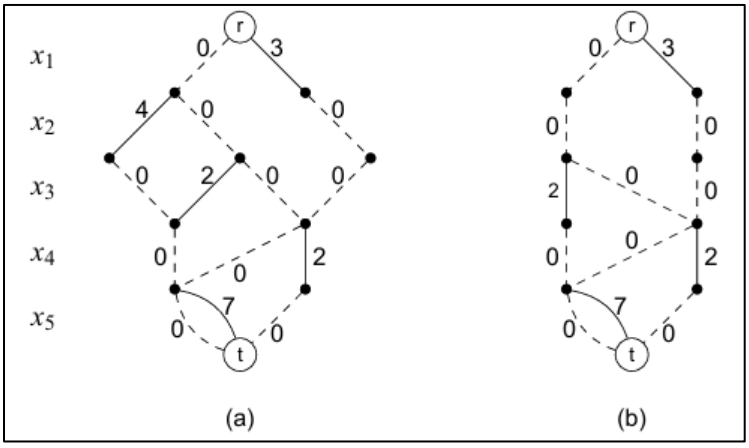

Fig 3: Comparison of an Exact DD and Restricted DD respresentation.

Restricted DD 또한 Relaxed DD 와 유사하게 설정할 부분이 두 가지 있다. 첫째, 위와 동일한 width parameter 을 설정할 수 있다. 둘째, 위와 유사한 node selection heuristic 이 있다. 이는 Relaxed DD 와 마찬가지로, problem structure 을 바탕으로 sub-optimal node 를 얼마나 공격적으로 exclude 하는 지에 따라 구현의 성능이 결정된다. Relaxed DD 와 다르게 operator 이 큰 문제가 되지 않는 이유는 valid relaxation 의 조건 때문이다. Relaxed DD 의 merge operator 은 superset 을 나타내야 하기에, valid 한 relaxation 이어야 하지만, exclusion 의 경우에는 그러하지 않기에 구현이 상대적으로 간단하다. Restricted DD 의 구현 과정은 Relaxed DD 와 거의 동일하며, node merging 대신 node exclusion 을 한다는 차이밖에 없다.

2.1.4 Branch-and-Bound Algorithm

본 서적에서 제시하는 Branch-and-Bound(BB) Algorithm 은 integer programming(IP)의 branch-and-bound scheme 를 바탕으로 한다. General IP solver 과 같은 역할을 하며, 서적에서는 이를 DD 방법론만을 사용했다는 점에서 의의를 높게 평가한다.

본 알고리즘은 크게 세 가지 원리를 바탕으로 구현된다. 첫째, Relaxed DD 와 Restricted DD 는 모두 solution space 에 대한 변형을 내제하지만, Exact DD 에서의 optimal path 가 merge/exclude 되지 않는다면 Relaxed/Restricted DD 는 optimal solution 을 반환한다. 둘째, Relaxed DD 와 Restricted DD 의 solution quality 를 construction 중에 확인할 수는 없지만, 각각은 optimal solution 에 대한 upper/lower bound 를 나타낸다. 따라서, construction 중에 bound property 를 만족하지 못한다면, 해당 diagram 속에는 optimal path 가 없음을 확인할 수 있다. 마지막으로, Relaxed DD 와 Restricted DD 의 구현은 Exact DD 보다 computational time 이 짧기에, solution space 에 대한 exact representation 을 최소화 하는 게 제일 효율적인 방법이다.

이를 바탕으로 BB Algorithm 은 다음과 같은 과정으로 실행된다. 첫째, Relaxed DD 를 만든 뒤, solution space 에 대한 relaxed representation 이 되는 layer 부터 탐색을 진행한다. 이는 general IP branch-and-bound 에서 linear relaxation 등으로 upper bound 를 구하는 과정과 같다. 둘째, relax 된 node 를 root node 로 하여, Restricted DD 를 만든다. 이는 어느 한 variable 에서 integer constraint 를 적용하여 possiblle solution 을 구하는 것과 같다. Restricted DD 에서 제시된 lower bound 가 여태까지의 solution 중 제일 크다면 optimal path 의 후보가 된다. 만약 lower bound 가 더 낮다면 bound 가 적용되어, node 탐색이 pass 된다. 셋째, 해당 node 를 root node 로 하여, Relaxed DD 를 만든다.

이는 어느 한 variable 값을 바탕으로 branch 하는 과정과 같다. Relaxed DD 의 solution space 가 어디까지 exact 한지 확인한 후, 위 과정을 반복한다. 한 path 가 exact 하면 알고리즘은 종결되며, optimal solution 이 제시된다. 해당 알고리즘에 대한 pseudocode 를 Algorithm 6 에서 확인할 수 있다.

Algorithm 6 Branch-and-Bound Algorithm

initialize $Q = \{r\}$, where r is the initial DP state
let $z_{\text{opt}} = -\infty$, $v^*(r) = 0$
while $Q \neq \emptyset$ **do**
  $u \leftarrow \text{select\_node}(Q)$, $Q \leftarrow Q \setminus \{u\}$
  create restricted BDD B'_{ut} using Algorithm 1 with root u and $v_r = v^*(u)$
  if $v^*(B_{ut}) > z_{\text{opt}}$ **then**
    $z_{\text{opt}} \leftarrow v^*(B')$
  if B'_{ut} is not exact **then**
    create relaxed BDD $\bar{B}_{ut}$ using Algorithm 1 with root u and $v_r = v^*(u)$
    if $v^*(\bar{B}_{ut}) > z_{\text{opt}}$ **then**
      let S be an exact cutset of $\bar{B}_{ut}$
      for all $u' \in S$ **do**
        let $v^*(u') = v^*(u) + v^*(\bar{B}_{uu'})$, add u' to Q
return z_{opt}

2.1.5 Variable Ordering

Exact DD 의 구현에서 매우 직관적으로 확인 가능한 부분이지만, DD 의 top-down construction 과정 중 variable ordering 은 DD 의 overall size 에 큰 영향을 끼친다. 같은 문제에 대한 Exact DD representation 으로, variable ordering 하나만 바뀌어도 생기는 영향을 Fig. 4 에서 확인할 수 있다.

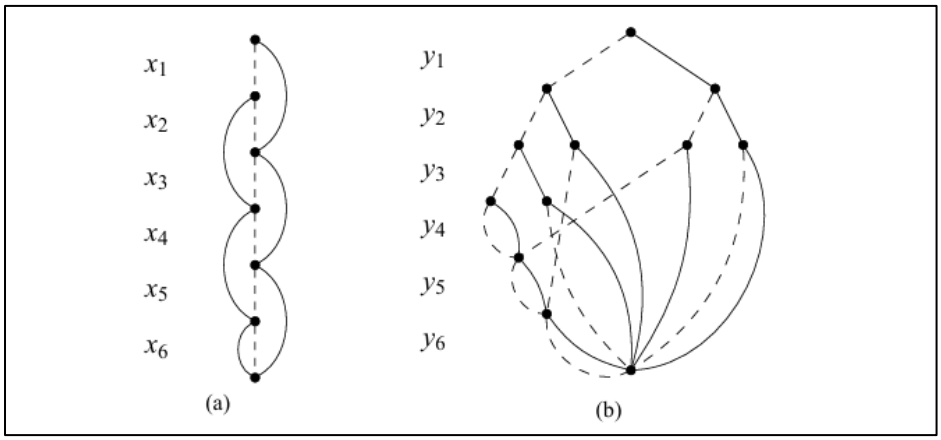

Fig. 4: Comparison of identical representation using different variable ordering.

그러나, 최적의 variable ordering 을 구하는 것 또한 NP-hard problem 이다.

2.2 Knapsack Problem

최적화에서의 Knapsack Problem 은 흔히 볼 수 있는 문제 유형이다. 다양한 용도를 가지고 있을 뿐더러, Knapsack Problem 계열에서 제일 간단한 형태 중 하나인 0/1 Knapsack Problem 에는 exact method 과 approximation method 모두 다양하게 적용되고 있다. 또한, dynamic programming formulation 을 가지고 있다는 점에서 본 과제에 적합한 최적화 문제라고 평가했다.

2.2.1 Problem Formulation

0/1 Knapsack Problem 은 여러 종류의 item 이 있음을 가정하며, 이러한 item 은 각각 weight 와 value 를 가짐을 가정한다. 각 item 은 copy 가 1 개씩 밖에 없어, knapsack 에 넣는 경우(1-arc), 또는 넣지 않는 경우(0-arc), 총 두 가지밖에 없다. Knapsack 의 capacity 를 넘지 않으면서, 총 value 의 값을 최대화하는 item 의 조합을 구하는 문제다. 이에 대한 mathematical model formulation 은 다음과 같다.

For a set of n total items numbered from 1 to n, each with a weight w_i and a value v_i, and a knapsack with a total capacity of W

$$Maximize \sum_{i=1}^{n} v_i x_i$$

$$(s.t.) \sum_{i=1}^{n} w_i x_i \leq W$$

$$x_i \in \{0, 1\} \text{ (binary variable)}$$

0/1 Knapsack Problem 의 dynamic programming formulation 은 current capacity 를 바탕으로 한다. 이는 item 에서의 0-arc 와 1-arc 에 따라 바뀌는 current capacity 를 바탕으로 recursive 하게 정의된다. 이러한 DP solution 은 *O(nW)* time 으로, pseudopolynomial time 을 가진다. 해당 알고리즘을 직접적으로 사용하는 것은 Section 3.1 이지만, 이러한 dynamic programming formulation 을 바탕으로 DD 의 state formulation 을 했다. 둘이 꼭 일치할 필요는 없지만, 0/1 Knapsack Problem 의 경우에는 item 을 넣는 order 이 무관하기에 state formulation 을 단순하게 잡는 게 유리하다. 엄밀한 formulation 은 다음과 같다:

$$\mathrm{m}(0, \mathrm{w}) = 0$$

$$m(i, w) = m(i-1, w) \; if \; w_i > W - w \qquad (0-arc)$$

$$m(i, w) = \max\left(m(i-1, w), m(i-1, w-w_i) + v_i \; if \; w_i \leq W - w\right) \qquad (1-arc)$$

2.2.2 State Formulation

본 연구에서는 0/1 Knapsack Problem 의 dynamic programming formulation 을 바탕으로 state formulation 을 했다. DD formulation 에 필요한 4 가지 요소는 다음과 같다:

1) State Space:

$state\ space\ S : space\ of\ all\ possible\ capacities$
$root\ state\ \hat{r} : empty\ knapsackterminal\ states$
$\widehat{t_k} : capacity\ of\ knapsack\ after\ adding\ n\ items\ in\ feasible\ state$
$\hat{0} : created\ when\ total\ weight\ of\ added\ items\ exceeds\ capacity$

2) Transition Functions

$t_j : S_j\ X\ D_j \rightarrow S_{j+1}$
$t_j(s_{j+1},\ d_j) = s_j - w_j\ \ if\ s_j - w_j \geq 0\ else\ \hat{0}$

3) Transition Cost Functions

$h_j : S\ X\ D_j \rightarrow \mathbb{R}$

$h_j(d_j) = v_j\ \ if\ s_j - w_j \geq 0\ else\ 0$

4) Root Value

$v_r : 0$

위 formulation 에 따라서 각 layer 을 하나의 item 으로 정의하며, 이에 따른 0-arc 와 1-arc 는 각각 item 을 넣지 않는 경우와 넣는 경우를 나타낸다. State formulation 에 따라 두 node 의 current capacity 가 같고, 같은 layer 에 있다면 둘은 하나의 node 로 취급해도 된다(Reduced Decision Diagram).

(continued below)

3. 연구 과정

3.1 Exact DD vs Dynamic Programming

Exact DD 는 위에서 언급한 듯, exact solver 으로 optimal solution 을 반환한다. 따라서 본 과제에서는 기존의 dynamic programming formulation 과 비교하여 computational time performance 를 비교하고자 한다. 또한, 문제의 크기나 특성에 따른 computational time 영향을 확인하여 Exact DD methodology 의 장단점을 확인한다.

3.1.1 Problem Size vs Computational Time

본 실험에서는 문제의 크기; Knapsack Problem 의 item 종류 수에 따른 Exact DD 의 computational time 을 확인하고자 한다. 각 item 의 weight 는 1 에서 20 사이, value 는 10 에서 100 사이, 그리고 knapsack 의 capacity 는 item weight 총합의 80%로 설정하여 random instance 를 생성했다. 이러한 instance 의 예시는 Table 1 에서 제시된다.

Problem Size	Capacity	Item Index	Weight	Value
5	38	0	8	47
		1	4	83
		2	8	84
		3	11	56
		4	17	94

Table 1: Example test case of knapsack instance

실험에 사용된 problem size 는 n=5, 10, 20, 50, 100, 200, 500, 1000 이다. 각 변인에서 위와 같은 instance 를 10 개 생성하여 computational time 을 기록했다.

3.1.2 Exact DD vs DP – Varying Problem Size

본 실험에서는 Exact DD methodology 와 dynamic programming 의 computational time 을 비교하고자 한다. Problem size 가 결과에 미치는 영향을 확인하고자 다양한 problem size 에서 비교를 진행했다. 위와 같은 방식으로 random instance 를 생성했으며, problem size 가 n=20, 100, 500, 1000 일 경우에서 instance 를 10 개씩 생성했다.

3.1.3 Exact DD vs DP – Varying Capacity

본 실험에서는 두 방법론의 비교에 있어 knapsack capacity 가 결과에 미치는 영향을 확인하고자 다양한 capacity 에서 비교를 진행했다. 위에서는 capacity 를 item weight 총합의 80%로 정한 반면, 본 실험에서는 capacity 를 생성된 instance 의 item weight 총합의 60%(Low), 90%(Mid), 1000%(High), 그리고 2500%(Extreme)로 실험을 진행했다. Problem size 가 n=20, 100, 1000 일 경우에서 각 capacity 변인에 해당되는 instance 를 10 개씩 생성했다.

3.2 Relaxed DD vs LP Relaxation

Relaxed DD 는 문제에 대한 relaxation 으로, original problem 에 대한 upper bound 를 제공한다. 본 과제에서는 이를 PuLP 라이브러리의 LP Relaxation 과 비교하여 computational time 이랑 bound quality 를 확인하고자 한다. 또한, 문제의 크기나 특성에 따른 bound quality/computational time 을 확인하여 Relaxed DD methodology 의 장단점을 확인한다.

3.2.1 Width vs Bound Quality/Computational Time

Relaxed DD 의 설정된 width 에 따라 relaxation 의 강도가 정해지며, 이로 인해 bound quality 와 computational time 모두 영향을 받는다. 이 관계를 확인하고자 본 실험을 진행한다.

본 실험에서는 3.1 과 같은 방식으로 instance 를 생성한다. Width 의 값이 W=2, 5, 10, 20, 50, 100 일 때, problem size 가 n=100 일 경우에서 각 weight 변인에 해당되는 instance 를 10 개씩 생성했다.

3.2.2 Node Select Heuristic vs Bound Quality/Computational Time

본 실험에서는 Relaxed DD 의 생성에 있어 node select heuristic 의 영향을 확인하고자 한다. 'random' heuristic 은 특정 layer 에서 node 를 무작위적으로 선택하여 merge 한다. 'minLP' heuristic 은 layer 중 state 값(remaining capacity)을 기준으로 하여 node 를 sorting 한 다음, 기준 width 이내로 들어오도록 state 가 작은 node 를 merge 한다. 'maxLP' heuristic 은 minLP 와 같은 방식으로 작동하지만, 반대로 state 가 큰 node 를 merge 한다.

본 실험에서는 3.1 과 같은 방식으로 instance 를 생성하며, random, minLP, maxLP 3 가지 heuristic 에 대하여 problem size 가 n=100, width 는 W=20 일 경우에서 instance 를 10 개씩 생성했다.

3.2.3 Relaxed DD vs LP Relaxation – Varying Problem Size

본 실험에서는 Relaxed DD methodology 와 LP relaxation 의 bound quality/computational time 을 비교하고자 한다. Width 와 node select heuristic 은 각각 3.2.1 과 3.2.2 의 결과를 참고하여 W=20, minLP heuristic 으로 설정했다. 이외 실험 방식은 3.1.2 와 동일하다.

3.2.4 Relaxed DD vs LP Relaxation – Varying Capacity

본 실험에서는 width 와 node select heuristic 은 각각 3.2.1 과 3.2.2 의 결과를 참고하여 W=20, minLP heuristic 으로 설정했다. 이외 실험 방식은 3.1.3 와 동일하다.

3.3 Restricted DD vs Greedy Heuristic

Restricted DD 는 문제에 대한 primal heuristic 으로, feasible solution/lower bound 를 제공한다. 본 과제에서는 이를 greedy + local search heuristic 과 비교하여 computational time 이랑 bound quality 를 확인하고자 한다. 또한, 문제의 크기나 특성에 따른 bound quality/computational time 을 확인하여 Restricted DD methodology 의 장단점을 확인한다.

본 실험에서 사용한 greedy + local search heuristic 에 대한 pseudocode 는 Algorithm 4 에서 확인할 수 있다.

3.3.1 Width vs Bound Quality/Computational Time

Restricted DD 의 설정된 width 에 따라 heuristic 의 강도가 정해지며, 이로 인해 bound quality 와 computational time 모두 영향을 받는다. 이 관계를 확인하고자 본 실험을 진행한다.

본 실험에서는 3.1 과 같은 방식으로 instance 를 생성한다. Width 의 값이 W=2, 5, 10, 20, 50, 100 일 때, problem size 가 n=100 일 경우에서 각 weight 변인에 해당되는 instance 를 10 개씩 생성했다.

3.3.2 Restricted DD vs Greedy Heuristic

본 실험에서는 Restricted DD methodology 와 greedy heuristic 의 bound quality/computational time 을 비교하고자 한다. 실험 방식은 3.2.3 와 동일하다.

3.4 Branch-and-Bound Algorithm

서적에서는 DD 기반 branch-and-bound algorithm 을 general IP solver 과 비교하여 bound quality/computational analysis 차원에서의 강점을 선보였다. 그러나, 해당 알고리즘을 구현하는 과정에서 비교 자체가 무의미할 정도로 general IP methodology 에 뒤쳐졌기에, 본 보고서에서는 구현 관련 특이사항을 구체화하고자 한다.

3.5 Variable Ordering

Variable ordering 은 DD 구현의 틀에 있어 중요한 요소로, Relaxed DD 의 node select heuristic 과 같이 문제 구조를 바탕으로 DD methodology 의 전반적인 효율성을 높일 수 있는 방안이다. 본 실험에서는 greedy heuristic 을 바탕으로 variable ordering 을 진행했다. DD 의 initialization 과 함께 input 으로 받은 item list 를 $\frac{value}{weight}$에 대하여 sort 를 진행한다. 이처럼 가치가 높을 것으로 판단되는 item(해에 들어갈 확률이 높을 item)을 DD 구현 앞쪽에 취급하여, 전체적인 node 의 수도 줄이며 Relaxed/Restricted DD 에서는 merging/excluding 에 휘말릴 확률을 줄여준다. 본 실험에서는 무작위적인 layer 순서를 나타내는 'random', 그리고 위와 같은 방식으로 ordering 을 한 'greedy' 두 가지를 비교한다.

본 실험에서는 3.1 과 같은 방식으로 instance 를 생성하여, problem size n=5, 10, 20, 50, 100, 200, 500, 1000 일 경우에서 random, greedy 각각에서 instance 10 개씩 생성하여 Exact DD 의 computational time 을 비교한다.

4. 결과

4.1 Exact DD vs Dynamic Programming

4.1.1 Problem Size vs Computational Time

Exact DD 는 problem size 에 따른 state explosion 으로 인해 큰 문제에 대한 적용이 어렵다. Problem size 와 computational time 사이의 관계는 Table 2, Figure 5 에서 확인할 수 있다.

Problem Size	Average Computational Time (s)
5	0.00872
10	0.00351
20	0.0189
50	0.17306
100	0.76745
200	3.36704
500	25.88351
1000	124.82017

Table 2: Experiment data for 4.1.1

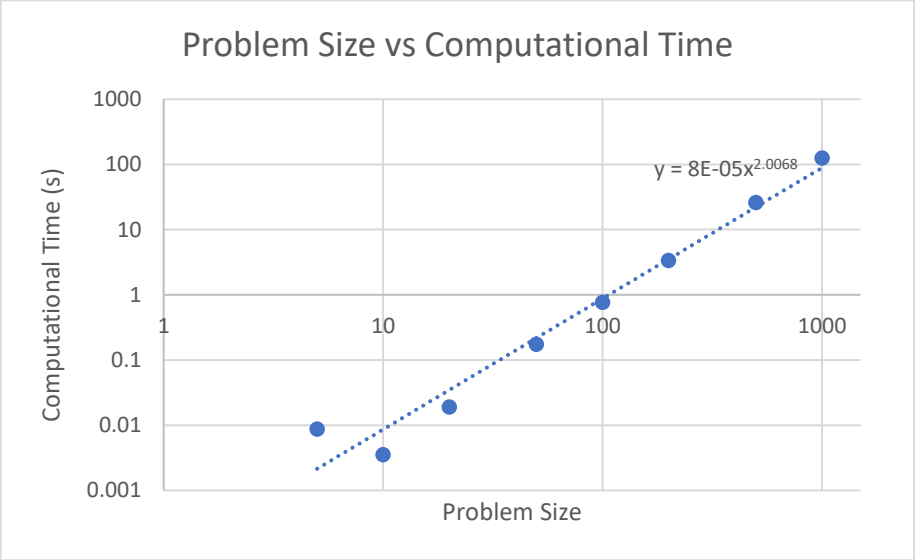

Figure 5: Graphical representation of 4.1.1 experiment results

동일한 조건에서 problem size 에 따른 computational time 이 빠르게 증가함을 확인할 수 있다. 특히, problem size n=50 에서부터 n=1000 까지는 이차함수 꼴의 관계를 확인할 수 있다.

4.1.2 Exact DD vs DP – Varying Problem Size

Problem size 에 따른 Exact DD methodology 와 DP formulation 의 computational time 을 확인했을 때, 현재 설정값으로는 DP 가 더 빨랐음을 확인할 수 있다. Problem size 에 의한 state explosion 은 Exact DD methodology 가

덜한다고 해도, 전체적인 알고리즘의 time complexity 가 DP formulation 보다 비효율적이기에 위와 같은 현상이 나타난다고 해석한다. 구체적인 결과는 Table 3, Figure 6 에서 확인할 수 있다.

Problem Size	Exact DD Average Time (s)	DP Average Time (s)
20	0.020533333	0.007772727
100	0.54247	0.06497
500	24.29322	1.58896
1000	98.88433	6.47301

Table 3: Experiment data for 4.1.2

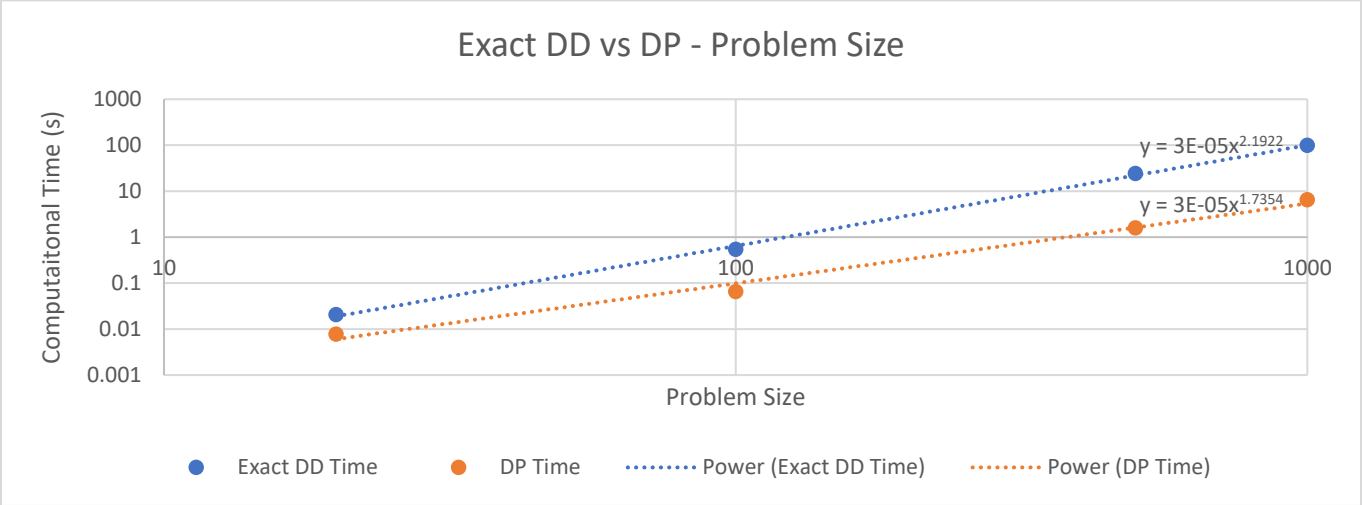

Figure 6: Graphical representation of 4.1.2 experiment results

해당 문제 구조에 한해서 DP formulation 이 더 효율적임을 확인할 수 있다. 이러한 결과는 두 방법론 모두 full state search 이라는 점과 더불어 Exact DD methodology 알고리즘의 복잡도를 바탕으로 해석할 수 있다.

4.1.3 Exact DD vs DP – Varying Capacity

Capacity 에 따른 Exact DD methodology 와 DP formulation 의 computational time 차이는 문제 특성에 따른 알고리즘 취급 방식으로 인해 발생한다. 본 실험은 어떠한 문제 구조에서 DD methodology 가 강점을 가질 지 보여준다. 구체적인 결과는 Table 4, Figure 7 에서 확인할 수 있다.

Problem Size	Capacity Type	Exact DD	DP
20	Low	0.021211	0.002233

100	Low	0.61008	0.06613
1000	Low	106.1718	5.51329
20	Mid	0.02569	0.00307
100	Mid	0.69844	0.08804
1000	Mid	109.9733	8.17432
20	High	0.017789	0.0289
100	High	0.75572	0.73011
1000	High	85.25631	81.44012
20	Extreme	0.02735	0.07499
100	Extreme	0.76413	1.85856
1000	Extreme	94.97833	233.6725

Table 4: Experiment data for 4.1.3

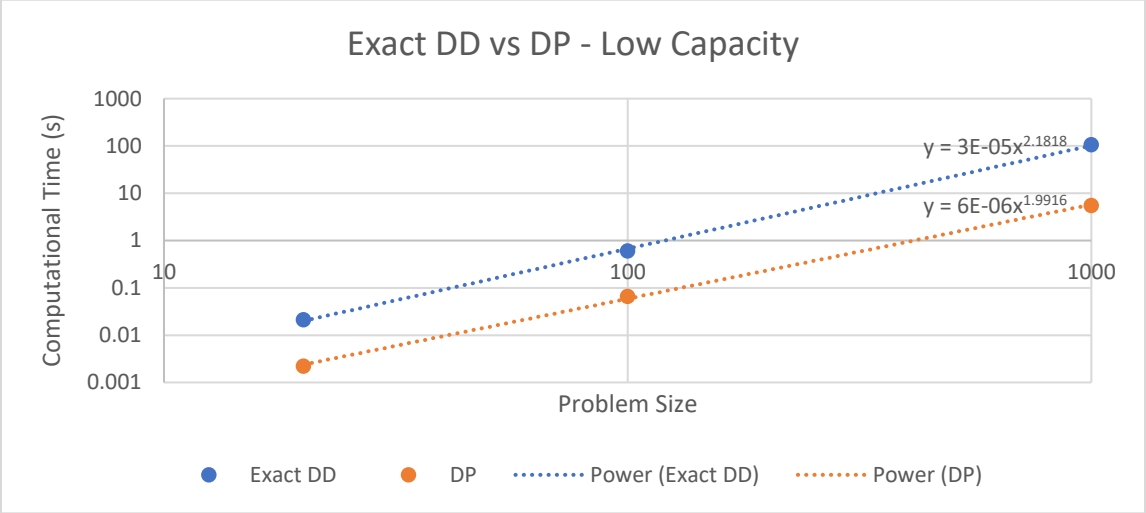

Figure 7-1: Graphical plot of 4.1.3 at low capacity

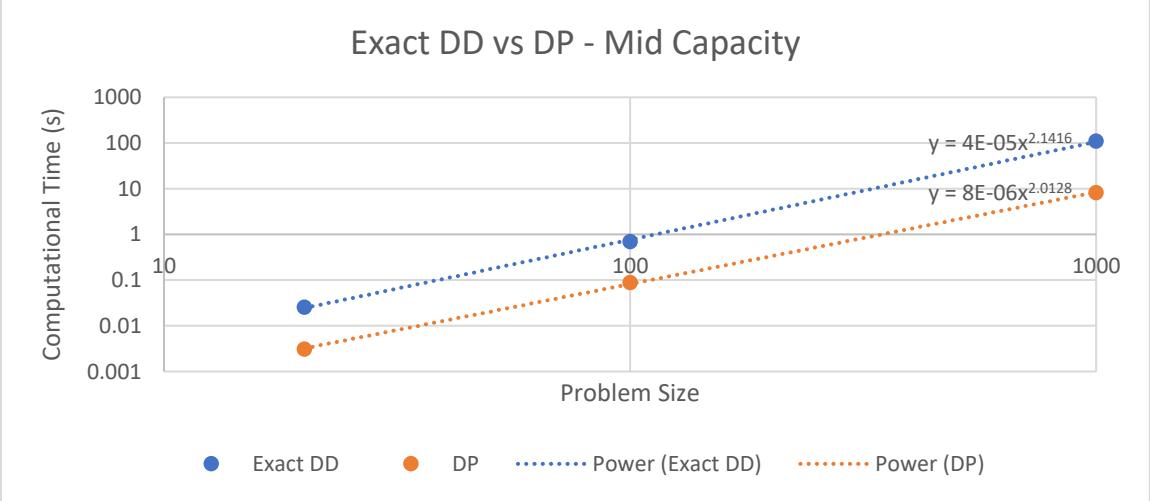

Figure 7-2: Graphical plot of 4.1.3. at mid capacity

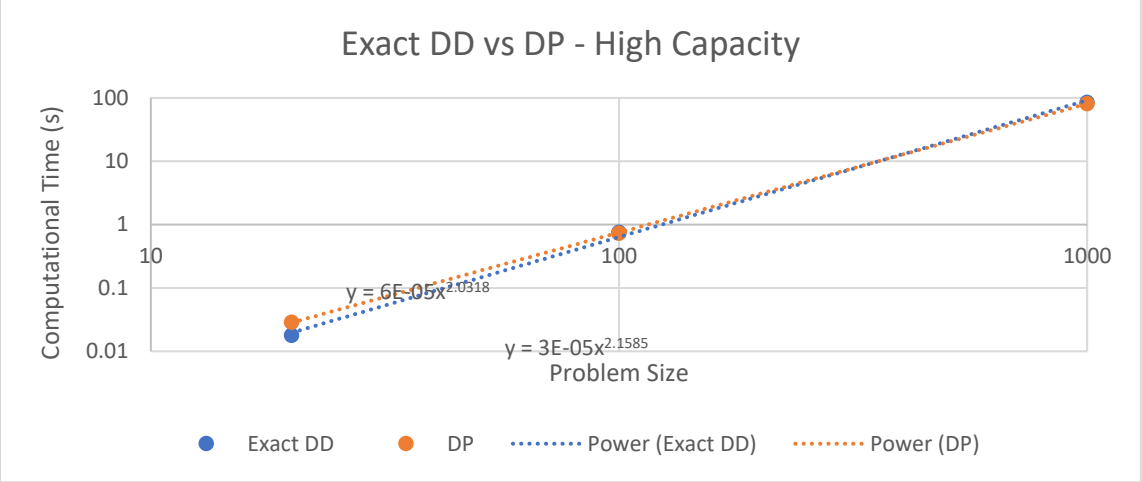

Figure 7-3: Graphical plot of 4.1.3 at high capacity

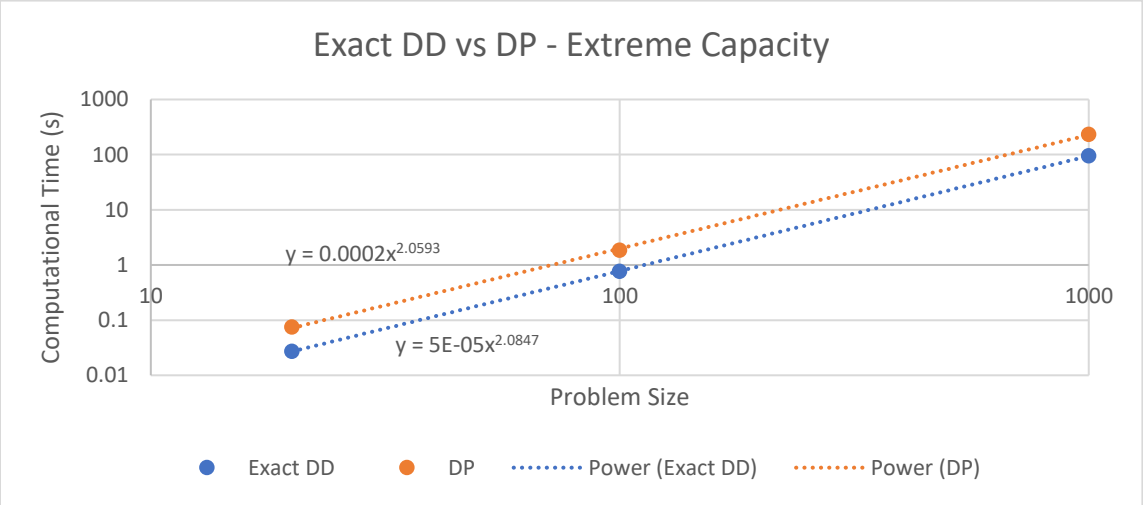

Figure 7-4: Graphical plot of 4.1.3 at extreme capacity

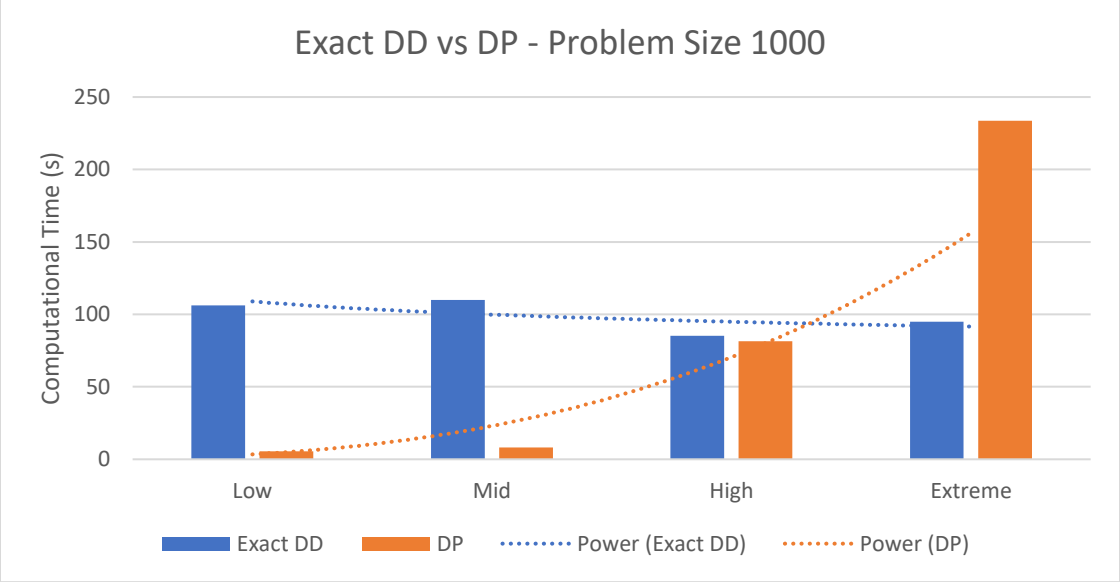

Figure 7-5: Graphical plot of 4.1.3 at problem size 1000

본 실험의 결과는 Exact DD 의 활용 가능성을 보여준다. DD 의 top-down construction 과정 중에서 state 가 중복되는 node 는 중첩되기에, capacity 가 높은 문제 상황에서는 일반적인 DP 보다 작은 representation 을 solution space 를 나타낸다. Figure 7-5 에서 이러한 점을 재차 확인할 수 있다. DP 는 모든 state 를 iteration 으로 다루기에 capacity 가 늘어나는 경우에서 computational time 이 같이 커지지만, DD 는 capacity 에 따른 computational time 의 유의미한 변화가 없었다.

Knapsack Problem 에서는 이러한 형태로 나타나지만, 이와 같이 state representation 을 쉽게 줄이지 못하는 DP formulation 을 가진 문제에서 Exact DD 를 활용할 이유로 보인다. TSP 와 같은 문제에서 또한, state representation 을 쉽게 중첩시킬 수 있는 representation 으로 활용할 방안이 있을 것으로 예상된다.

4.2 Relaxed DD vs LP Relaxation

4.2.1 Width vs Bound Quality/Computational Time

Relaxed DD 는 width 에 따라 node merging 이 진행되며, optimal solution 을 나타내는 path 위 node 가 merge 되면 해에 대한 upper bound 가 제공된다. 따라서, width 가 작을수록 node 의 수는 적어지며 computational time 에는 유리하지만 bound quality 는 줄어들 수 있음을 직관할 수 있다. 설정되는 width 와 bound quality/computational time 사이 관계는 Table 5, Figure 8 에서 확인할 수 있다.

Width	Absolute Relative Gap	Computational Time
2	0.05473	0.01037
5	0.04842	0.02146
10	0.05259	0.04862
20	0.04544	0.12056
50	0.03424	0.17372
100	0.00936	0.28126

Table 5: Experiment data for 4.2.1

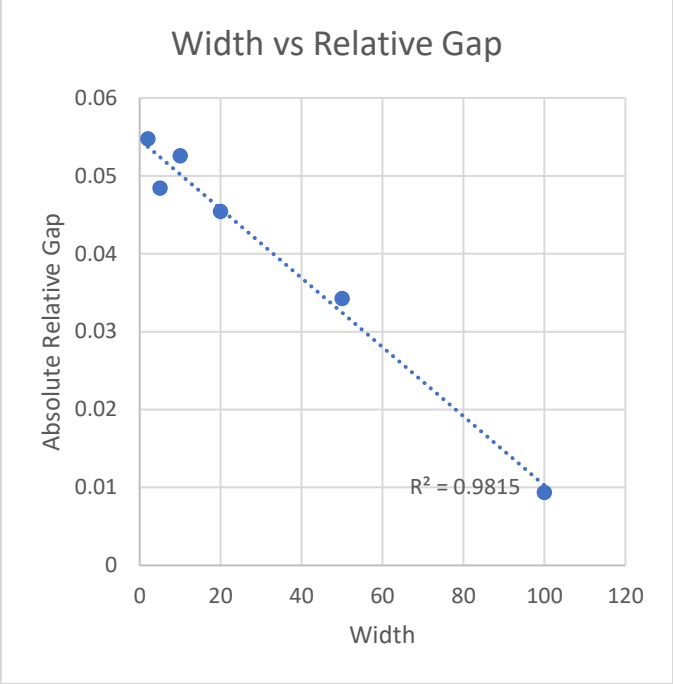

Figure 8-1: Graphical plot of 4.2.1 in relation to relative gap

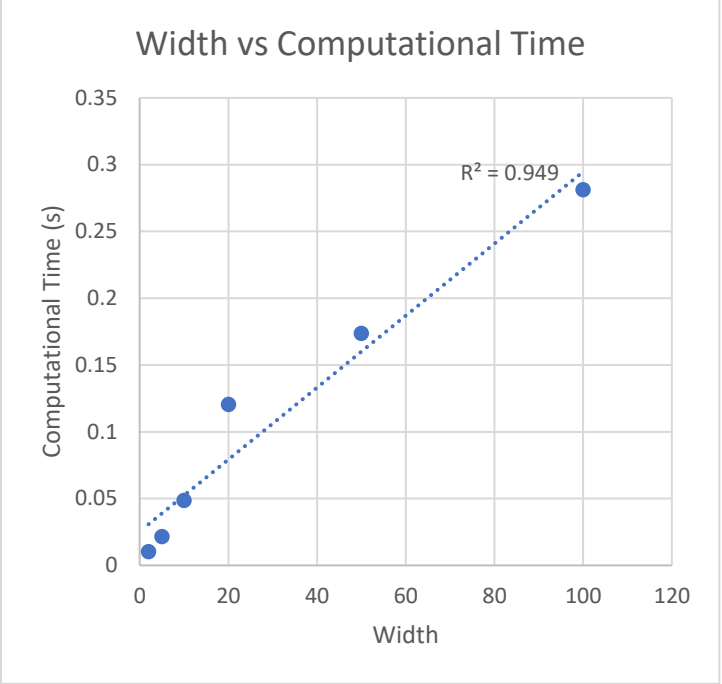

Figure 8-2: Graphical plot of 4.2.1 in relation to computational time

이처럼 width 가 커질수록 bound quality 는 높아지며, computational time 은 점점 커짐을 확인할 수 있다. 또한, absolute relative gap 가 < 0.01 범위로 들어오는 시점이 width 랑 problem size 와 같아지는 점임을 확인하여 이를 지표로 삼을 수 있다.

4.2.2 Node Select Heuristic vs Bound Quality/Computational Time

Node select heuristic 의 영향을 Table 6, Figure 9 에서 확인할 수 있다.

Heuristic	Absolute Relative Gap	Computational Time
random	0.05201	0.10107
maxLP	0.05201	0.09311
minLP	0.04335	0.0867

Table 6: Experiment data for 4.2.2

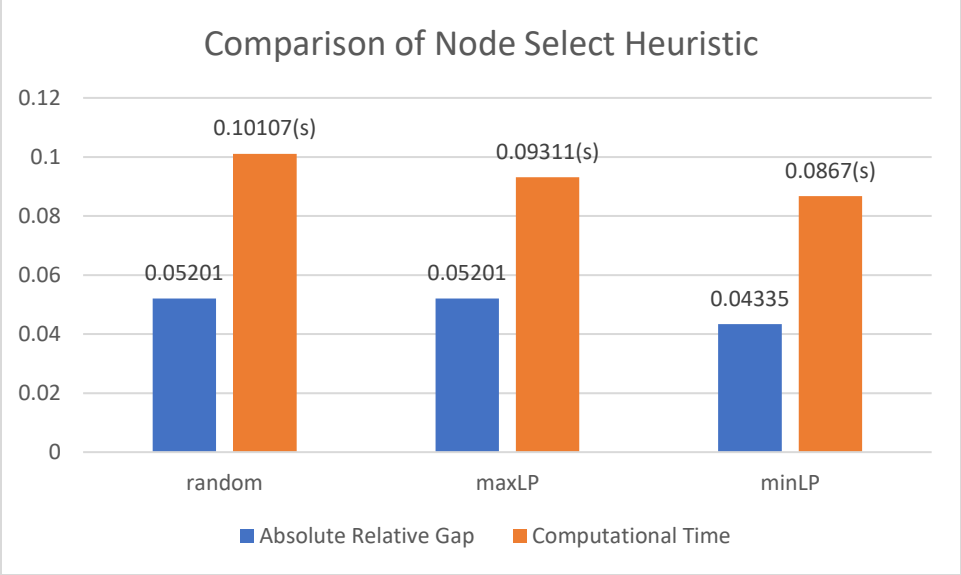

Figure 9: Graphical plot of 4.2.2

Bound quality 와 computational time 모두 minLP 가 제일 유리하다는 점과 더불어, node select heuristic 이 bound quality 보다 computational time 에 큰 영향을 미친다는 점 또한 확인할 수 있다. Node select heuristic 에 따라 optimal path 의 merging 여부가 결정되는 것은 맞기에 bound quality 에 영향을 끼친다는 결과를 해석할 수 있으나, 이보다도 전반적인 node 의 개수에 큰 영향을 미쳐 computational time 에 대한 영향 또한 확인할 수 있다.

이러한 node select heuristic 의 성능은 implementation 을 통해 더욱 증폭할 수 있다. 특히, variable ordering 이나 top-down construction 차원에서 dynamic node select 를 실행할 수 있도록 자료 구조형을 설정하면 더욱 효율성 높게 사용할 수 있다.

4.2.3 Relaxed DD vs LP Relaxation – Varying Problem Size

Problem size 에 따른 Relaxed DD methodology 와 LP relaxation 의 성능을 비교했을 때 전반적으로 LP relaxation 이 우위에 있음을 확인할 수 있다. 구체적인 결과는 Table 7, Figure 10 에서 확인할 수 있다.

Problem Size	DD Gap	LP Gap	DD Time	LP Time
20	0	0.00847	0.03589	0.06585
100	0.00792	0.00095	0.27351	0.08282
500	0.04416	0.00001	1.59758	0.12939

1000	0.05086	0	2.99754	0.19716

Table 7: Experiment data for 4.2.3

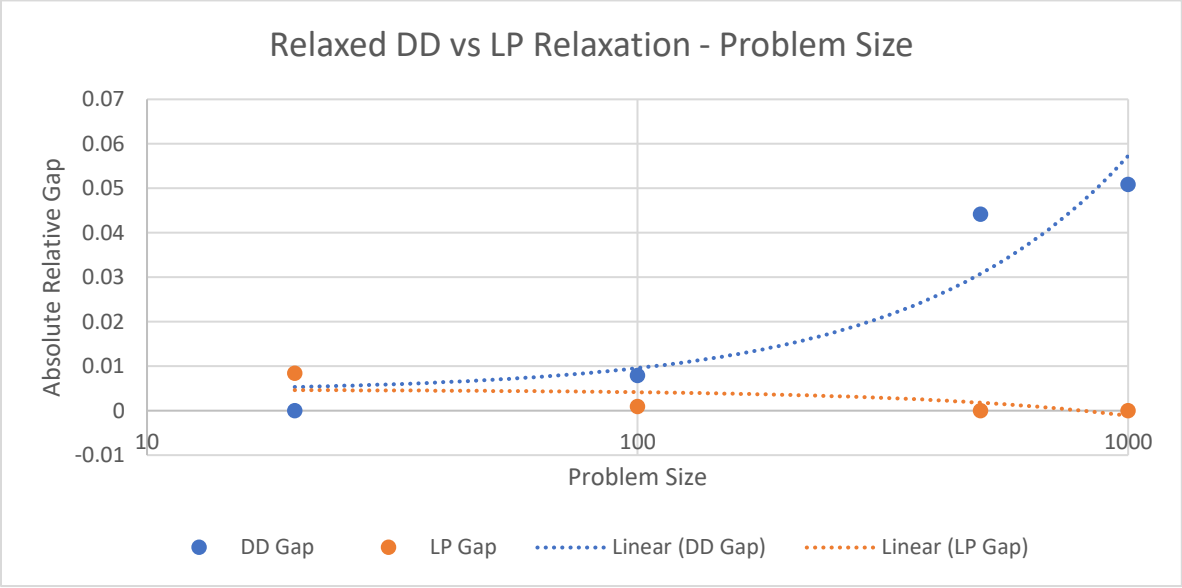

Figure 10-1: Graphical plot of 4.2.3 in relation to problem size

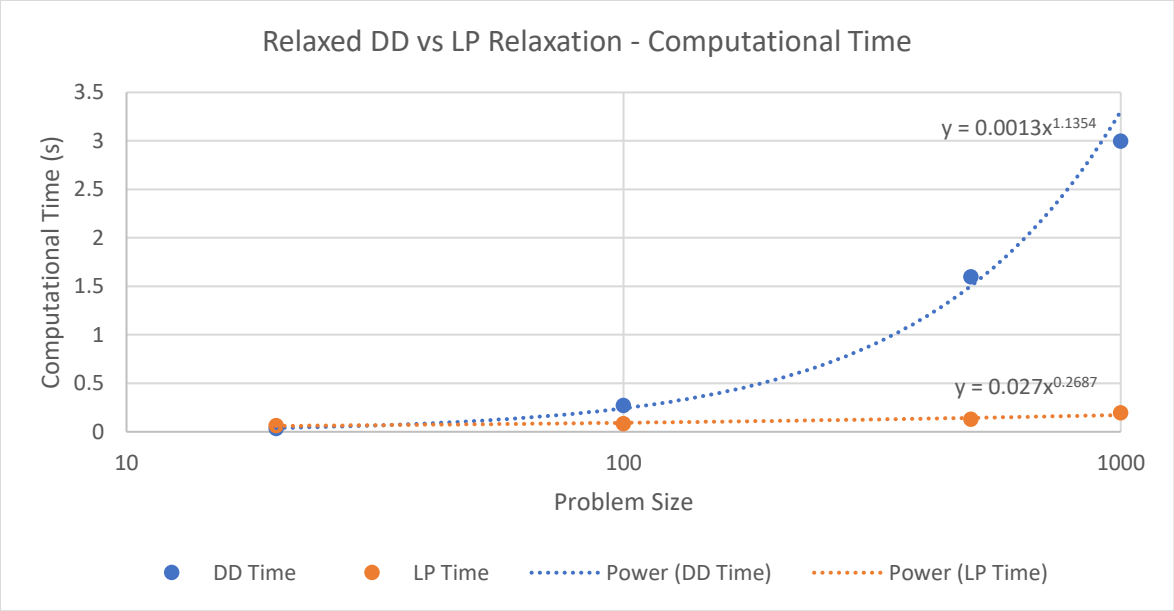

Figure 11-2: Graphical plot of 4.2.3 in relation to computational time

4.1.2 의 DP formulation 과는 다르게 LP relaxation 은 problem size 가 커질수록 bound quality 가 올라감을 확인할 수 있다. 제일 작은 problem size 에서는 LP relaxation 이랑 Relaxed DD 의 bound quality 순위가 역전되며, 작은 problem size 범위에서의 심층적인 탐구가 필요함을 제시한다.

Computational time 차원에서는 state explosion 으로 인한 악효과를 Relaxed DD 가 더 받는다는 점을 확인할 수 있다.

4.2.4 Relaxed DD vs LP Relaxation – Varying Capacity

기존의 Exact DD 와 같이 capacity 차원에서 DD methodology 의 성능이 오르지는 않음을 본 실험에서 확인할 수 있다. 그러나, 작은 problem size 에서는 Relaxed DD 가 LP relaxation 보다 성능이 좋음을 확인할 수 있다. 이는 LP relaxation 에서 fractional solution 이 나올 경우 필요한 rounding 의 과정에 반면, Relaxed DD 는 작은 problem size 에서 거의 exact 한 solution space 를 나타내기에 발생하는 현상으로 보인다. 구체적인 결과는 Table 8, Figure 11 에서 확인할 수 있다.

Capacity Type	DD Relative Gap	LP Relative Gap	DD Time	LP Time
Low	0	0.00713	0.01637	0.04818
Mid	0	0.0069	0.02489	0.04506
High	0	0	0.02625	0.04478
Extreme	0	0	0.01774	0.04431

Table 8: Experiment data for 4.2.4

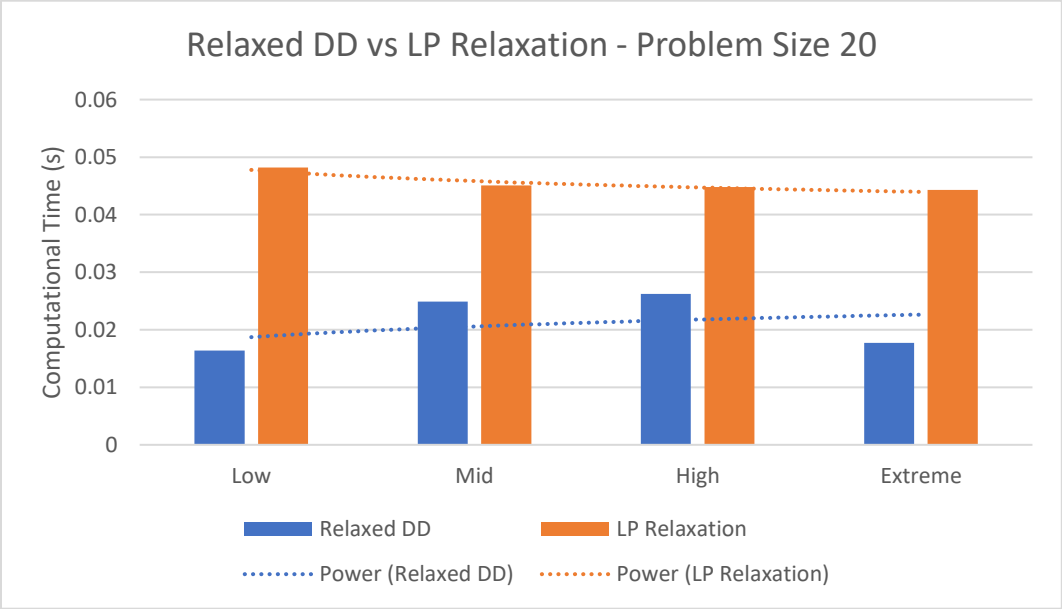

Figure 11: Graphical plot of 4.2.4 in relation to capacity type

Capacity 와 관련된 problem structure 은 아니어도 Relaxed DD 가 LP relaxation 을 outperform 할 경우는 몇가지 고안할 수 있다. 첫째, item 의 weight 와 value 가 correlation 을 가질 경우, 공격적인 node select heuristic/variable ordering 전략으로 Relaxed DD 의 성능을 높힐 수 있다. 둘째, 기존의 0/1 Knapsack Problem 에서 추가적인 constraint 가 있을 경우(ex. conflict constraint, etc)에도 Relaxed DD 가 더욱 효율적으로 문제를 접근할 수 있다. 이처럼 general 한

경우에서는 LP Relaxation 의 성능이 더 높지만 더욱 복합적인 instance 에 대해서는 추가 실험이 필요하다는 점을 확인할 수 있다.

4.3 Restricted DD vs Greedy Heuristic

4.3.1 Width vs Bound Quality/Computational Time

Restricted DD 는 width 에 따라 node excluding 이 진행되며, optimal solution 을 나타내는 path 위 node 가 exclude 되면 해에 대한 lower bound 가 제공된다. 따라서, Relaxed DD 와 마찬가지로 width 가 작을수록 node 의 수는 적어지며 computational time 에는 유리하지만 bound quality 는 줄어들 수 있음을 직관할 수 있다. 설정되는 width 와 bound quality/computational time 사이 관계는 Table 9, Figure 12 에서 확인할 수 있다.

Width	Absolute Relative Gap	Computational Time
2	0.00064	0.00413
5	0.00061	0.01286
10	0.00043	0.01949
20	0	0.03604
50	0	0.08274
100	0	0.15626

Table 9: Experiment data of 4.3.1

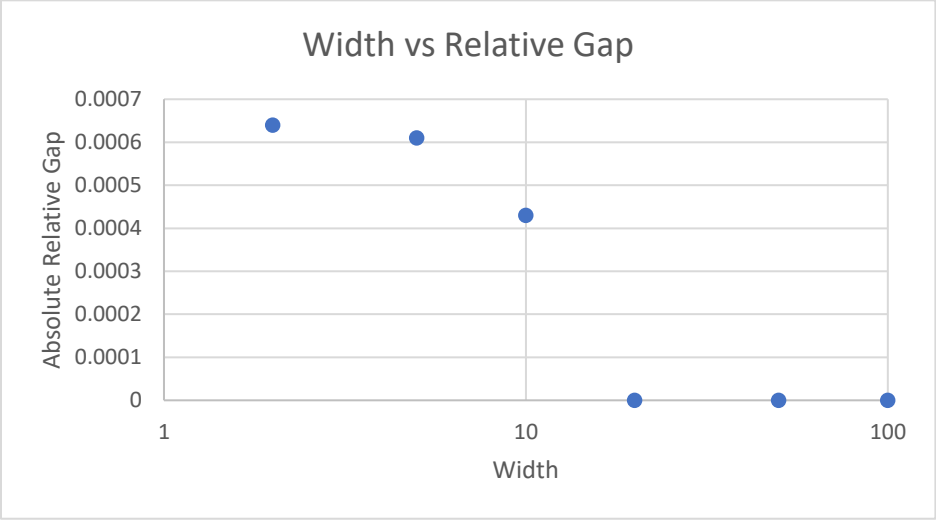

Figure 12-1: Graphical plot of 4.3.1 in relation to absolute relative gap

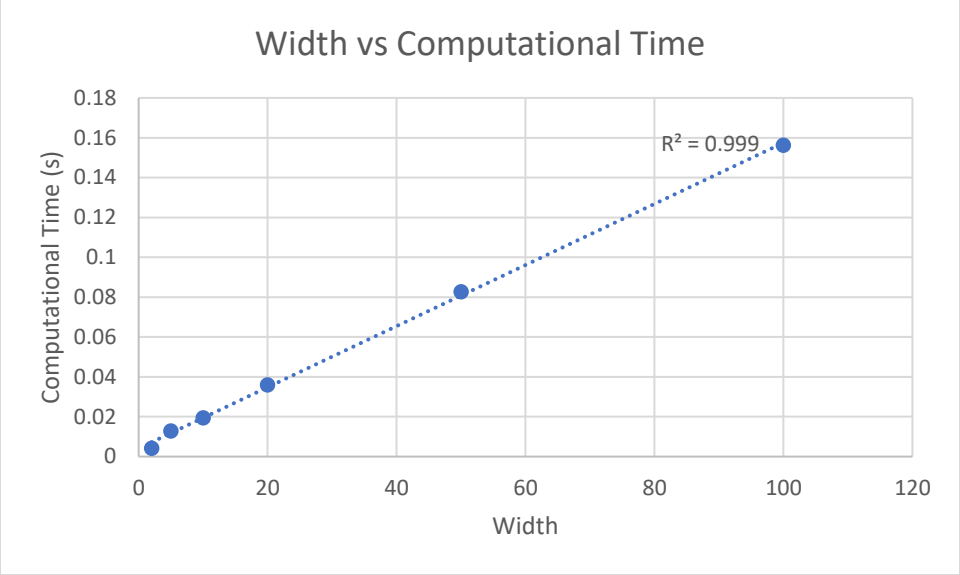

Figure 12-2: Graphical plot of 4.3.1 in relation to computational time

본 실험의 결과를 통해 Restricted DD 의 bound quality 가 Relaxed DD 의 bound quality 보다 width 에 대한 영향을 더 받는다는 점을 확인할 수 있다. 특히, width W=20 부터 Restricted DD 는 optimum 을 반환한다. 이는 큰 instance 에서는 node merging 의 효과가 node excluding 보다 크다는 점으로 해석할 수 있다. Node merging 의 경우에는 해당 node 를 거치는 모든 path 가 relaxation 을 거치지만, excluding 의 경우에는 삭제된 path 만 영향을 받기 때문이다.

Width 와 computational time 의 관계는 Restricted DD 와 Relaxed DD 가 유사함을 볼 수 있다. Width 가 커질수록 excluding function 의 시행 횟수가 늘어나며 점점 커진다.

4.3.2 Restricted DD vs Greedy Heuristic

본 실험의 구체적인 결과는 Table 10, Figure 13 에서 확인할 수 있다.

Problem Size	DD Gap	Greedy Gap	DD Time	Greedy Time
50	0.00008	0.00209	0.01791	0.000001
100	0.00006	0.00061	0.03671	0.000001
200	0.00001	0.00041	0.07892	0.0002
10000	0	0	4.4436	0.01369

Table 10: Experiment data of 4.3.2

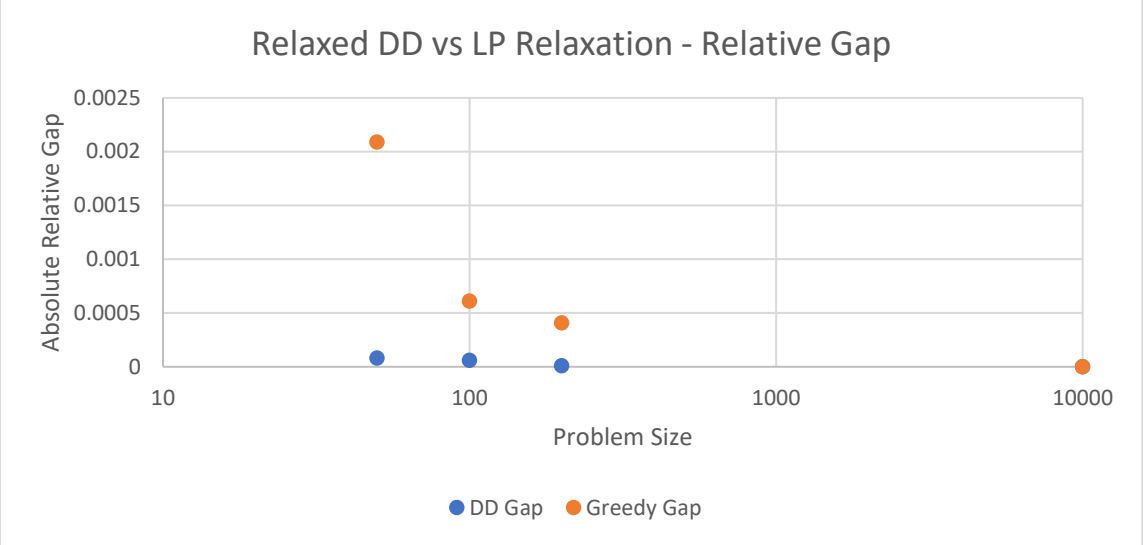

Figure 13-1: Graphical plot of 4.3.2 in relation to absolute relative gap

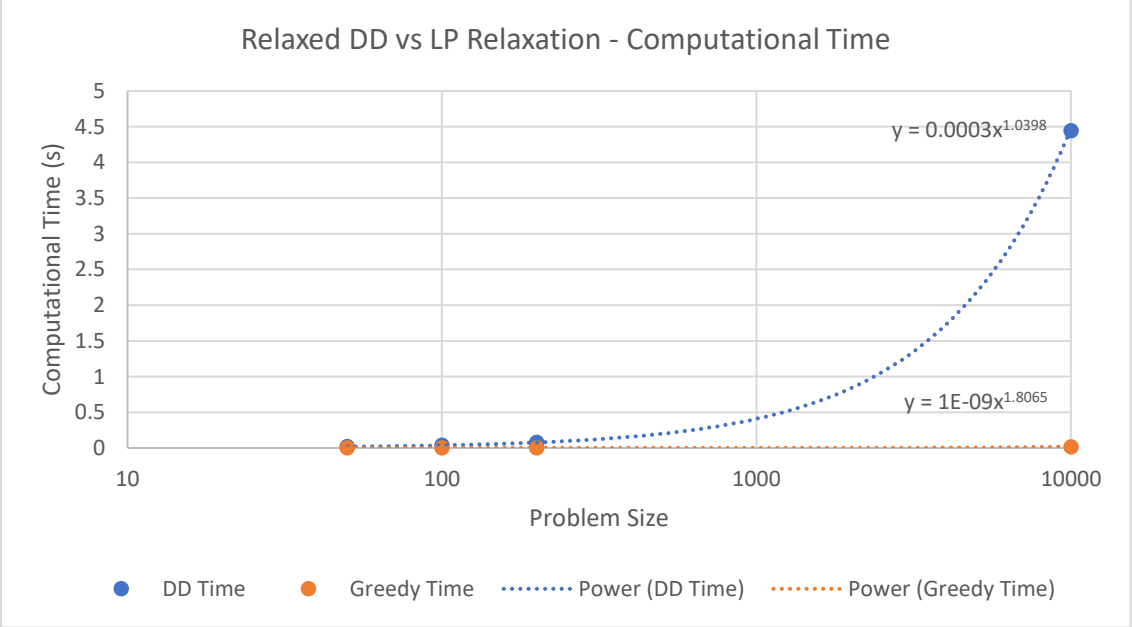

Figure 13-2: Graphical plot of 4.3.2 in relation to computational time

본 실험의 결과는 기존 Relaxed DD 와 비슷한 면모를 보인다. DD methodology 가 bound quality 에 있어 더 유리한 면은 있지만, computational time 에서는 Greedy heuristic 와 100 배 넘게 차이나는 instance 도 있다. 또한, Restricted DD 와 Greedy heuristic 모두 problem size 가 커질수록 bound quality 가 좋아진다는 점에서 Restricted DD 의 활용 방안을 찾기 더욱이 힘들다.

이러한 결과는 본 과제의 Restricted DD implementation 과정으로 인해 발생한 것으로 해석된다. Restricted DD 의 구현에 있어, greedy 형식의 variable ordering 이 제일 효율적임을 확인하여 overhead 로 해당 ordering 을 구한 뒤, DD 를 construct 하는 방식을 활용했다. 그러나, 이는 결국 Greedy heuristic 과 거의 유사한 방법으로, DD 의 top-down

construction 과정을 추가했기에 computational time 차원에서 더 느릴 수밖에 없다. 이러한 방식으로 결과를 해석할 경우, problem size 에 대한 power-regression 에서 Restricted DD 가 Greedy heuristic 보다 작은 exponent 를 가진 점 또한 해석할 수 있다. 결국 Exact DD 와 DP 의 차이처럼, 충분한 사이즈에 도달했을 때는 construction 과정보다 node 중첩의 효과가 더 커지며, 해당 시점에서는 Restricted DD 가 Greedy heuristic 보다 빠를 수도 있음을 시사한다.

이 또한 Relaxed DD 와 마찬가지로 implementation 차원에서의 최적화로 더욱 차별화할 수 있을 것으로 예상된다.

4.4 Branch-and-Bound Algorithm

본래의 연구 계획에서는 서적에서 제공한 BB Algorithm 을 General IP solver 이랑 비교하여 위와 같은 computational analysis 를 진행하고자 했다. 그러나, 본 알고리즘을 구현하는 과정에서 크게 세 가지 문제가 확인됐다. 첫째, BB Algorithm 은 Relaxed/Restricted DD 의 iterative 생성을 활용하기에 두 알고리즘이 비효율적일 경우에는 computational time 이 급격하게 올라간다. 둘째, node structure 을 overhead 로 저장해두지 않고, 매번 새로 생성하는 과정으로 인해 기본적인 computational time 이 매우 길다. 셋째, 0/1 Knapsack Problem 의 formulation 에 있어 작은 width 로 공격적인 relaxation/reduction 이 되어야 빠른 bound 가 제공되지만, 해당 문제에서는 그러한 전략을 적용하기에 어려운 점이 있다.

위 문제점을 참고하여 조금 더 발전된 구현 전략을 제시하고자 한다. 첫째, parallelization 전략과 더불어 DD construction 을 overhead 로 진행하기 위해 node 임시 저장을 구현한다. 반복적인 DD construction 의 computational time 을 줄일 수 있을 것으로 보며, bound quality 의 하락은 Relaxed/Restricted DD 의 width 로 조율할 수 있다. 둘째, DD 자료형 구조의 전반적인 optimization 과 merging/excluding 함수의 optimization 을 통해 computational time 을 줄일 수 있다. 셋째, DD construction scheme 자체를 dynamic ordering 으로 구현하여 생성되는 Relaxed/Restricted DD 의 depth 를 키우고 총 iteration 수를 줄일 수 있다.

이러한 구현 전략을 채택한다고 해도, 해당 알고리즘을 활용하기에는 추가 실험이 필요한 부분들이 있다. 첫째, 탐색할 node 를 저장하는 자료 구조형(알고리즘에서는 "Q"로 표시)를 queue-based(BFS), 또는 stack-based(DFS)로 함에 따라 성능에 차이가 있을 것으로 보인다. 특히, node select heuristic 과 variable ordering 이랑의 조합에 따라 어떤 자료

구조형을 채택하는 지에 큰 영향이 있을 것으로 보인다. 둘째, 위의 실험들과 같이 problem structure 에 따른 구현 전략 또한 확인할 필요가 있다.

4.5 Variable Ordering

본 과제에서 구현한 DD 는 전부 Greedy variable ordering scheme 을 활용했다. 본 실험에서는 greedy scheme 이랑 random scheme 의 차이를 확인하고자 한다. 본 실험의 구체적인 결과는 Table 11, Figure 14 에서 확인할 수 있다.

Problem Size	Random Time	Greedy Time
5	0.00113	0.00086
10	0.00973	0.00945
20	0.07357	0.0374
50	0.43447	0.37772
100	2.01379	1.39811
200	8.24871	6.34376
500	51.49763	39.32221

Table 11: Experiment data for 4.5

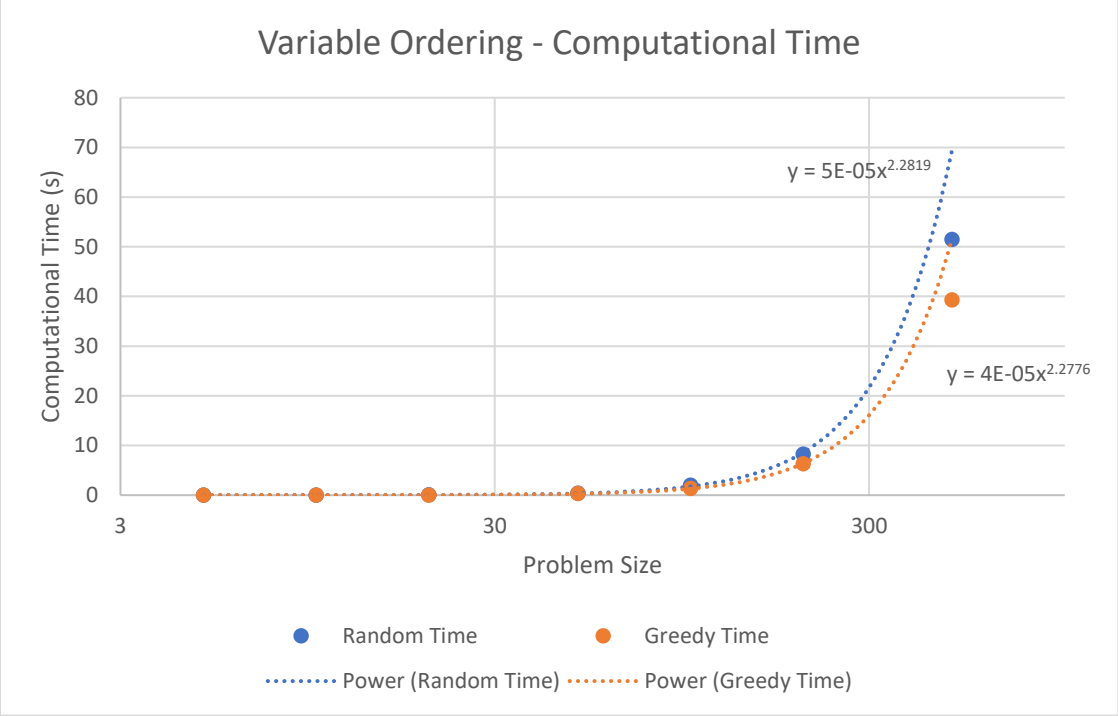

Figure 14: Graphical plot of 4.5 in relation to computational time

Greedy scheme 는 final solution 에 들어갈 node 를 먼저 처리함에 있어, node 중첩을 더 늘릴 수 있고 이에 따라 DD 의 최종 size 를 줄일 수 있다는 점에서 advantage 를 가진다. 본 실험에서도 greedy scheme 이 random scheme 보다 유리한 결과가 나왔으며, 특히 큰 problem size 일수록 state explosion 방지의 효과가 커진다는 점을 확인할 수 있다.

본 주제에 관해서 Relaxed/Restricted DD 의 bound quality/computational time 또한 확인할 필요가 있다. Variable ordering 과 이에 맞는 node select heuristic 을 사용할 경우에 더욱 큰 효과를 보일 것으로 예상된다.

(continued below)

5. 고찰

본 과제를 통해 DD methodology 의 확실한 성능을 확인하는 것보다 어떠한 문제 상황에서, 어떤 방식으로 구현하는 것이 최적인지를 확인할 수 있었다. 위 실험을 통해 확인한 DD 의 특징은 크게 세 가지로 정리할 수 있다:

1) 첫째, 실제 문제를 적용하는 과정에 있어 DD 구현에 중요한 점은 node select heuristic 이나 variable ordering 과 같은 차원에서 problem structure 을 최대한 반영하는 것이다. 이처럼 구체적인 heuristic 의 구현을 통해 solution space 에 대한 representation 을 효과적으로 해야만 기존 방법론보다 좋은 성능을 보인다.

2) 둘째, DD 의 핵심 장점은 customizability 와 문제 상황에 따른 strategic approach 이다. Relaxed DD 나 Restricted DD 의 구현에 있어 width 를 통해 bound quality 와 computational time 사이의 trade-off 를 조율하는 과정이나, BB algorithm 과 같은 방법론에 있어 자료 구조형을 수정하여 문제 상황에 알맞은 방식을 적용할 수 있다. 특히, 기존 방법론에다 DD 방법론을 덧붙였을 때 적용하고자 하는 역할에 맞춰 이를 쉽게 조정할 수 있다.

3) 셋째, 구현 자체의 효율성에 큰 영향을 받는다. Node merging/excluding function 이 실행되는 횟수나 state explosion 으로 인한 node 수를 보아 DD 방법론은 기본적으로 computationally exhaustive 한 방법론이다. 그런 만큼 구현 방식이나 root function 의 효율성에 큰 영향을 받는다.

본 연구 과제의 한계는 구현 과정과 computational analysis 의 방법론에 있다. 구현 차원에서 node merging/excluding function 을 더 최적화할 수 있어, 본 형태의 DD 에서도 더 좋은 성능을 보일 수 있다. 또한, Relaxed DD/Restricted DD 와 Exact DD 의 computational time 을 비교했을 때, root function 의 computational efficiency 가 떨어진다는 점을 확인할 수 있다.

방법론 차원의 한계는 문제 설정과 비교 방식에 있다. 0/1 Knapsack Problem 의 설정 당시에는 computational analysis 에 대한 확실한 계획 없이 진행됐으나, 오히려 DP 구조가 확실하며 LP Relaxation 이 쉬운 문제로 설정하는 바람에 problem structure 에 의존하는 DD methodology 의 성능이 상대적으로 떨어졌다. 오히려 problem structure 을 가용할 수 있는 더욱 복잡한 형태의 문제를 활용했을 경우 DD methodology 의 장단점이 더욱 드러났을 것으로 보인다.

(continued below)

6. 후기

제 두 번째 연구참여를 진행하는 과정에서 제일 크게 와닿았던 점은 최적화 연구를 하기 위해 필요한 스킬셋의 다양성이었습니다. 본 과제를 진행하면서는 크게 세 가지: 코딩 능력, 수학적 직관력, 그리고 연구/문제 해결 능력의 필요성을 더욱 느꼈습니다.

코딩 능력은 사실 첫 번째 연구참여로 인해 이의 필요성을 익히 알고 있었습니다. 첫 연구참여에서 이러한 능력의 부족으로 인해 크게 어려워 했으며, 같은 문제를 겪지 않겠다는 다짐으로 두 번째 연구참여를 시작했음에도 정말 중요한 능력임을 느꼈습니다. 단순히 코딩 언어에 대한 익숙함, 자료 구조형에 대한 이해 뿐만 아니라 문제를 접근하는 방식이 중요하게 느껴졌습니다. 마치 번역 작업처럼, 최적화 차원에서 해석한 문제를 해결하는 도구로써 코드로 번역을 해주는 과정이 필요하며, 이를 적절히 하기 위해 더욱 많은 코딩 경험이 필요하다고 느껴졌습니다.

수학적 직관력은 포항공대 산경과 내 '수리계획(IMEN-361)' 수업에서 처음 필요성을 느꼈습니다. 결국 최적화도 기초 수학에서 파생된 연구 분야이기에, 그리고 자료 구조형이나 알고리즘을 다루는 과정에 있어 수학적인 직관력이 중요하다는 점이 느껴졌습니다. 같은 이론적인 엄밀함이 필요한 것은 아니더라도, 문제 해결의 씨앗이 되어 응용적인 특성이 가까운 물류최적화에도 정말 중요한 스킬셋임을 느꼈습니다.

마지막으로 연구 차원의 능력은 이번에 제가 맡은 과제의 성격으로 인해 더욱이 느껴진 것 같습니다. 제가 사수님의 과제를 섀도잉 하면서 주어진 문제를 해결하는 것이 아닌, 주제 하나를 바탕으로 제 나름의 문제를 찾아가고 이러한 문제를 정의하는 과정이 매우 새롭게 느껴졌습니다. 결국 제 연구참여에서도 문제 정의가 제일 긴 과정이 되었고, 문제를 해결하는 과정에서도 정의를 여러 번 갈아엎었습니다. 어떠한 프로젝트의 scope 를 설정하고, 이것을 체계적으로 분해하여 문제 해결의 도구로써 연구를 사용하는 방법론은 앞으로 더욱 익혀야 한다는 점을 느꼈습니다.

이외에도 제 첫 연구참여랑은 매우 다른 경험이었습니다. 물류 연구실 선배님들의 세미나 준비를 돕는 과정이나, 사수님과 함께 새로운 주제를 탐구하는 과정은 제게 새롭게 다가왔습니다. 제 기존의 경험과는 또다른 연구의 단면을 경험한 것 같습니다.

제 기존의 연구참여 경험이나 학생으로서의 특성을 고려하여 또 새로운 학습의 기회를 주신 김병인 교수님과 매주 시간과 노력을 들여 제 과제를 지도해주신 나진우 사수님께 감사드립니다. 무엇보다 학생으로서 조금 더 발전하여, 조금 더 멋진 연구를 해내고 싶은 마음이 커지는 연구참여 경험이 되었습니다.