Αλγόριθμοι και Προχωρημένες Δομές Δεδομένων
Δυναμικός Προγραμματισμός

Πανεπιστήμιο Ιωαννίνων, Τμήμα Πληροφορικής και Τηλεπικοινωνιών,
Μεταπτυχιακό Πληροφορικής & Δικτύων (2019-2020)

Γκόγκος Χρήστος

Νοέμβριος 2019

Τι είναι ο Δυναμικός Προγραμματισμός;

- Ο Δυναμικός Προγραμματισμός (ΔΠ) είναι μια αλγοριθμική προσέγγιση που εκμεταλλεύεται, εφόσον μπορούν να εντοπιστούν, ιδιαίτερα χαρακτηριστικά του προβλήματος έτσι ώστε να υπολογιστεί αποδοτικά η βέλτιστη λύση. Συνδυάζει αναδρομή, αποθήκευση ενδιάμεσων λύσεων (memoization) και συνδυασμό βέλτιστων λύσεων υποπροβλημάτων για τη βέλτιστη επίλυση του συνολικού προβλήματος.

Ο δυναμικός προγραμματισμός ως μια διαδικασία τριών βημάτων

- Εντοπισμός μιας σχετικά μικρής ομάδας υποπροβλημάτων στα οποία μπορεί να διασπαστεί το πρόβλημα.
- Εντοπισμός γρήγορου τρόπου επίλυσης μεγαλύτερων προβλημάτων δεδομένων των λύσεων μικρότερων προβλημάτων.
- Εντοπισμός του τρόπου που η τελική λύση μπορεί να προκύψει από τις λύσεις των υποπροβλημάτων.

Αρχή της βελτιστότητας

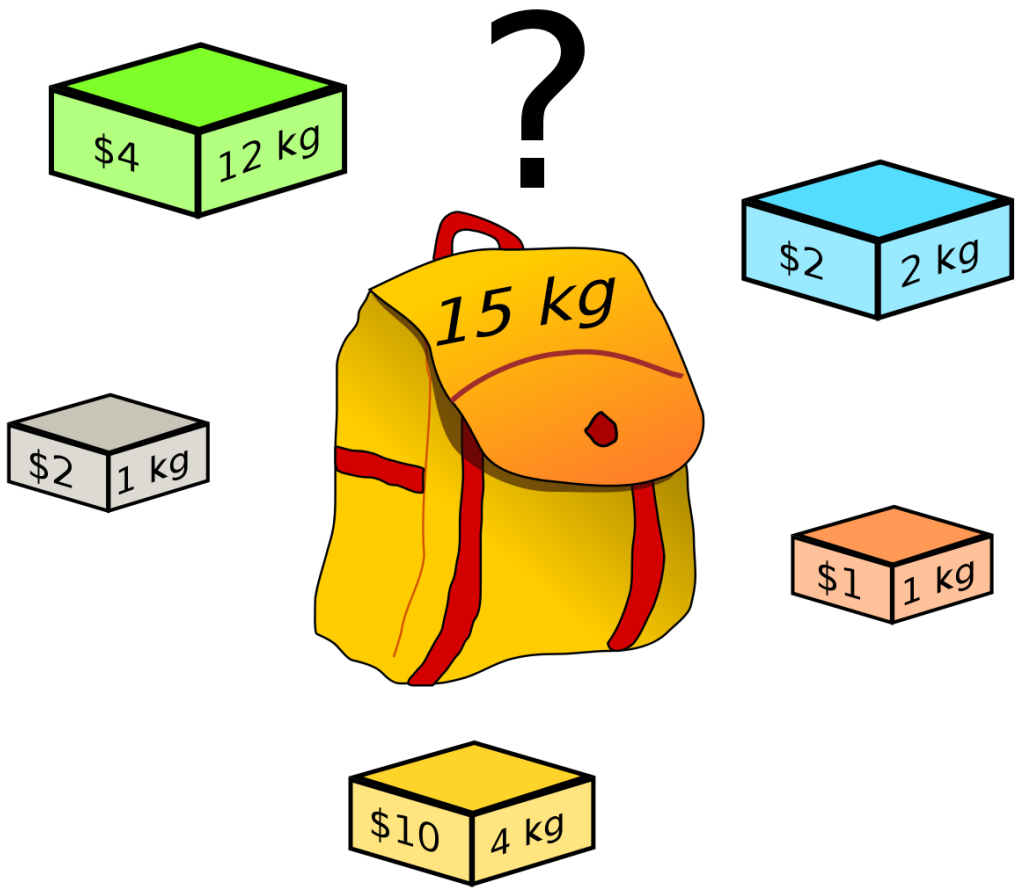
- Ανεξάρτητα από την πρώτη απόφαση οι υπόλοιπες αποφάσεις θα πρέπει να είναι βέλτιστες σε σχέση με την κατάσταση που προκύπτει μετά την πρώτη απόφαση

Παραδείγματα προβλημάτων στα οποία μπορεί να εφαρμοστεί ο ΔΠ

- 0-1 σακίδιο (0-1 knapsack)
- Μέγιστη κοινή υποακολουθία (Longest Common Subsequence)
- Άθροισμα υποσυνόλου (Subset Sum Problem)
- Αλυσιδωτός πολλαπλασιασμός πινάκων (Matrix Chain Multiplication)
- Συντομότερο μονοπάτι σε κατευθυνόμενο ακυκλικό γράφημα (Shortest path in a DAG)
- Μακρύτερη αυξανόμενη υποακολουθία (Longest increasing subsequence problem)
- Απόσταση επεξεργασίας (Edit distance problem)
- Συντομότερα αξιόπιστα μονοπάτια (Shortest reliable paths)
- Συντομότερα μονοπάτια για όλα τα ζεύγη κορυφών (All-pairs shortest paths)

0-1 σακίδιο: ορισμός προβλήματος

- n αντικείμενα
- Αξίες αντικειμένων: $v_1, v_2, v_3, \dots, v_n$ (ακέραιες θετικές τιμές, αλλά μπορεί να είναι και πραγματικές θετικές τιμές)
- Βάρη αντικειμένων: $w_1, w_2, w_3, \dots, w_n$ (ακέραιες θετικές τιμές)
- Χωρητικότητα σάκου: C (ακέραια θετική τιμή)
- Ζητείται να βρεθεί το σύνολο $S \subseteq \{1,2,3, \dots, n\}$ και για το οποίο ισχύει ότι:
  - $\max V = \sum_{i \in S} v_i$
  - $\sum_{i \in S} w_i \leq C$



Βέλτιστη υποδομή

- Αν θεωρήσουμε ότι ήδη γνωρίζουμε τη βέλτιστη λύση $S \subseteq \{1,2,3,\dots,n\}$ με συνολική αξία $V = \sum_{i \in S} v_i$ τότε η λύση αυτή είτε θα περιέχει είτε δεν θα περιέχει το τελευταίο αντικείμενο n:
  - Αν η βέλτιστη λύση δεν περιέχει το n τότε η $S^* = \{1,2,3,\dots,n-1\}$ θα είναι η βέλτιστη λύση στο υποπρόβλημα των πρώτων $n-1$ αντικειμένων. Αν δεν ίσχυε αυτό τότε θα υπήρχε καλύτερη λύση και για το πρόβλημα των n αντικειμένων
  - Αν η βέλτιστη λύση περιέχει το n τότε θα πρέπει $w_n \leq C$ και η απομένουσα λύση των πρώτων $n-1$ αντικειμένων θα αποτελεί τη βέλτιστη λύση στο υποπρόβλημα επιλογής από τα πρώτα $n-1$ αντικείμενα με διαθέσιμη χωρητικότητα $C - w_n$. (γιατί ισχύει αυτό το συμπέρασμα;)

Γιατί ισχύει το προηγούμενο συμπέρασμα;

- Για την περίπτωση που η βέλτιστη λύση $S \subseteq \{1,2,3,\dots,n\}$ και $n \epsilon S$ ισχύει ότι:
  - Δεσμεύοντας w_n χωρητικότητα προκειμένου να συμπεριληφθεί στη λύση το αντικείμενο n παραμένει διαθέσιμη $C - w_n$ χωρητικότητα.
  - Αν υπάρχει καλύτερη λύση $S^* \subseteq \{1,2,3,\dots,n-1\}$ με συνολική αξία $V^* > V - v_n$ για μέγιστη χωρητικότητα $C - w_n$ τότε η $S^* \cup \{n\}$ θα είχε συνολική αξία $V^* + v_n > V - v_n + v_n = V$.
  - Όμως το συμπέρασμα $V^* + v_n > V$ είναι άτοπο διότι υποθέσαμε ότι η λύση S είναι βέλτιστη.

Διατύπωση του συμπεράσματος της βέλτιστης υποδομής

Έστω S η βέλτιστη λύση στο 0-1 σακίδιο. Τότε:

- Το S είναι η βέλτιστη λύση για τα πρώτα $n-1$ αντικείμενα και χωρητικότητα C.
- Το S αποτελείται από **το αντικείμενο $\boldsymbol{n}$** και **τη βέλτιστη λύση για το υποπρόβλημα των πρώτων $\boldsymbol{n-1}$ αντικειμένων και χωρητικότητα $\boldsymbol{C-w_n}$.**

Υποπροβλήματα i, j

- Τα υποπροβλήματα ορίζονται με βάση δύο παραμέτρους, το πλήθος των i πρώτων αντικειμένων και τη χωρητικότητα του σακιδίου j.
- Το «υποπρόβλημα» n, C είναι το αρχικό πρόβλημα.
- Αν $V_{i,j}$ είναι η μέγιστη συνολική αξία του υποσυνόλου των i πρώτων αντικειμένων με συνολική χωρητικότητα το πολύ j τότε ισχύει ότι για κάθε $i = 1,2, \dots, n$ και κάθε $j = 0,1,2, \dots, C$:

$$V_{i,j} = \begin{cases} V_{i-1,j} & \text{εάν } w_i > j \\ \max\{V_{i-1,j}, V_{i-1,C-w_i} + V_i\} & \text{εάν } w_i \leq j \end{cases}$$

Αλγόριθμος δυναμικού προγραμματισμού για το πρόβλημα 0-1 σακιδίου

```
V = (n+1)*(C+1) // δισδιάστατος πίνακας – εκκίνηση αρίθμησης από το 0
-----------------------------------------------------------------
for j = 0 to C do
      V[0,j] = 0
for i = 1 to n do
      for j = 0 to C do
            if w[i] > j then
                  V[i,j] = V[i-1,j]
            else
                  V[i,j] = max(V[i-1,j], V[i-1,j-w[i]] + v[i]
return V[n,C]
```

Αλγόριθμος εντοπισμού αντικειμένων που συμμετέχουν στη λύση

```
S = {}
j = C
for i = n downto 1 do
      if w[i] <= j && V[i-1,j-w[i]] + v[i] >= V[i-1,j] then
            S = S ∪ {i}
            j = j – w[i]
return S
```

Απόδοση αλγορίθμου

- Ο αλγόριθμος αποτελείται από δύο φάσεις, τη συμπλήρωση του πίνακα V και τον εντοπισμό των αντικειμένων που συμμετέχουν στη λύση στο συμπληρωμένο πίνακα:
  - Για τη συμπλήρωση του πίνακα ο αλγόριθμος διαθέτει $O(1)$ χρόνο για την επίλυση καθενός από τα $(n+1) * (C+1) = O(nC)$ προβλήματα. Συνεπώς, εκτελείται σε $O(nC)$ χρόνο.
  - Για τον εντοπισμό των αντικειμένων που συμμετέχουν στη λύση ο αλγόριθμος διαθέτει $O(1)$ χρόνο σε κάθε επανάληψη. Συνεπώς, εκτελείται σε $O(n)$ χρόνο.
- Άρα, καθώς η πολυπλοκότητα του αλγορίθμου καθορίζεται από τη φάση με την υψηλότερη πολυπλοκότητα, η πολυπλοκότητα του αλγορίθμου είναι $\boldsymbol{O(nC)}$.

Εκτέλεση αλγορίθμου (1/6)

Αντικείμενο	Αξία	Βάρος
1	3	4
2	2	3
3	4	2
4	4	3
n=4, C=6		

	0	1	2	3	4	5	6
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**
1							
2							
3							
4							

Η πρώτη γραμμή του πίνακα (γραμμή-0) συμπληρώνεται με μηδενικά.

- V[0,0] = 0
- V[0,1] = 0
- V[0,2] = 0
- V[0,3] = 0
- V[0,4] = 0
- V[0,5] = 0
- V[0,6] = 0

Εκτέλεση αλγορίθμου (2/6)

Αντικείμενο	Αξία	Βάρος
1	3	4
2	2	3
3	4	2
4	4	3
n=4, C=6		

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	**0**	**0**	**0**	**0**	**3**	**3**	**3**
2							
3							
4							

Ο υπολογισμός των αποτελεσμάτων γίνεται γραμμή προς γραμμή

- V[1,0] = 0
- V[1,1] = 0
- V[1,2] = 0
- V[1,3] = 0
- V[1,4] = max(V[0,4], V[0,4-4]+v[1]) = max(0, 0+3) = 3
- V[1,5] = max(V[0,5], V[0,5-4]+v[1]) = max(0, 0+3) = 3
- V[1,6] = max(V[0,6], V[0,6-4]+v[1]) = max(0, 0+3) = 3

Εκτέλεση αλγορίθμου (3/6)

Αντικείμενο	Αξία	Βάρος
1	3	4
2	2	3
3	4	2
4	4	3
n=4, C=6		

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	0	0	3	3	3
2	**0**	**0**	**0**	**2**	**3**	**3**	**3**
3							
4							

- V[2,0] = 0
- V[2,1] = 0
- V[2,2] = 0
- V[2,3] = max(V[1,3], V[1,3-3]+v[2]) = max(0, 0+2) = 2
- V[2,4] = max(V[1,4], V[1,4-3]+v[2]) = max(3, 0+2) = 3
- V[2,5] = max(V[1,5], V[1,5-3]+v[2]) = max(3, 0+2) = 3
- V[2,6] = max(V[1,6], V[1,6-3]+v[2]) = max(3, 0+2) = 3

Εκτέλεση αλγορίθμου (4/6)

Αντικείμενο	Αξία	Βάρος
1	3	4
2	2	3
3	4	2
4	4	3
n=4, C=6		

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	0	0	3	3	3
2	0	0	0	2	3	3	3
3	**0**	**0**	**4**	**4**	**4**	**6**	**7**
4							

- V[3,0] = 0
- V[3,1] = 0
- V[3,2] = max(V[2,2], V[2,2-2]+v[3]) = max(0, 0+4) = 4
- V[3,3] = max(V[2,3], V[2,3-2]+v[3]) = max(2, 0+4) = 4
- V[3,4] = max(V[2,4], V[2,4-2]+v[3]) = max(3, 0+4) = 4
- V[3,5] = max(V[2,5], V[2,5-2]+v[3]) = max(3, 2+4) = 6
- V[3,6] = max(V[2,6], V[2,6-2]+v[3]) = max(3, 3+4) = 7

Εκτέλεση αλγορίθμου (5/6)

Αντικείμενο	Αξία	Βάρος
1	3	4
2	2	3
3	4	2
4	4	3
n=4, C=6		

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	0	0	3	3	3
2	0	0	0	2	3	3	3
3	0	0	4	4	4	6	7
4	**0**	**0**	**4**	**4**	**4**	**8**	**8**

- V[4,0] = 0
- V[4,1] = 0
- V[4,2] = V[3,2] = 4
- V[4,3] = max(V[3,3], V[3,3-3]+v[4]) = max(4, 0+4) = 4
- V[4,4] = max(V[3,4], V[3,4-3]+v[4]) = max(4, 0+4) = 4
- V[4,5] = max(V[3,5], V[3,5-3]+v[4]) = max(6, 4+4) = 8
- V[4,6] = max(V[3,6], V[3,6-3]+v[4]) = max(7, 3+4) = 8

Εκτέλεση αλγορίθμου (6/6)

Αντικείμενο	Αξία	Βάρος
1	3	4
2	2	3
3	4	2
4	4	3
n=4, C=6		

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	0	0	3	3	3
2	0	0	0	2	3	3	3
3	0	0	4	4	4	6	7
4	0	0	4	4	4	8	8

Εντοπισμός λύσης: Ξεκινώντας από την κάτω δεξιά γωνία του πίνακα (`i=4,j=6`), ο αλγόριθμος ελέγχει σε κάθε βήμα ποια περίπτωση της αναδρομής χρησιμοποιήθηκε:

- Αν **`w[i]>j ή V[i-1,j-w[i]]+v[i]<V[i-1,j]`** τότε το αντικείμενο `i` δεν συμπεριλαμβάνεται στη λύση.
- Αν **`w[i]<=j και V[i-1,j-w[i]]+v[i]>=V[i-1,j]`** τότε το αντικείμενο `i` συμπεριλαμβάνεται στη λύση και το νέο `j` μειώνεται κατά το βάρος του αντικειμένου `i`.

- `w[4]<=6 && V[3,6-3]+v[4]>=V[3,6]` ➔ `3<=6 && 8>=7` (OK και νέο `j=6-3=3`)
- `w[3]<=3 && V[2,3-2]+v[3]>=V[2,3]` ➔ `2<=3 && 4>=0` (OK και νέο `j=3-2=1`)
- `w[2]<=1` ➔ `3<=1` (NOT OK)
- `w[1]<=1` ➔ `4<=1` (NOT OK)
- Άρα, η λύση είναι η $S = \{3,4\}$ με συνολική αξία 8 και συνολικό βάρος 5

Προγραμματιστική λύση

Αντικείμενο	Αξία	Βάρος
1	3	4
2	2	3
3	4	2
4	4	3
n=4, C=6		

```
#include "../ortools/include/ortools/algorithms/knapsack_solver.h"
namespace operations_research {
     void KnapsackExample() {
         KnapsackSolver solver(KnapsackSolver::KNAPSACK_DYNAMIC_PROGRAMMING_SOLVER,
                               "SimpleKnapsackExample");
         std::vector<int64> values = {3, 2, 4, 4};
         std::vector<std::vector<int64>> weights = {{4, 3, 2, 3}};
         std::vector<int64> capacities = {6};
         solver.Init(values, weights, capacities);
         int64 computed_value = solver.Solve();
         std::cout << "Total value: " << computed_value << std::endl;
         std::vector<int> packed_items;
         for (std::size_t i = 0; i < values.size(); ++i)
             if (solver.BestSolutionContains(i))
                 std::cout << "Item " << i + 1 << " is in the bag" << std::endl;
     }
} // namespace operations_research
int main() {
    operations_research::KnapsackExample();
    return EXIT_SUCCESS;
}
```

https://github.com/chgogos/uoi_algorithms_and_complexity ortools_dp_solver_knapsack_demo.cc

Μεταγλώττιση και εκτέλεση (ORTools + Windows + VS 2017)

```
> tools\make run SOURCE=..\src\ortools_dp_solver_knapsack_demo.cc
```

Η εντολή μεταγλώττισης και εκτέλεσης δίνεται από τον κατάλογο ortools χρησιμοποιώντας το τερματικό x64 Native Tools Command Prompt for VS 2017

```
cl /EHsc /MD /nologo /D_SILENCE_STDEXT_HASH_DEPRECATION_WARNINGS -nologo /O2 -DNDEBUG /D__WIN32__
/DNOMINMAX /DWIN32_LEAN_AND_MEAN=1 /D_CRT_SECURE_NO_WARNINGS /DGFLAGS_DLL_DECL=
/DGFLAGS_DLL_DECLARE_FLAG= /DGFLAGS_DLL_DEFINE_FLAG= /Iinclude\\src\\windows /Iinclude /I. -DUSE_CBC -DUSE_CLP
-DUSE_BOP -DUSE_GLOP  -c ..\src\ortools_dp_solver_knapsack_demo.cc  /Foobjs\\ortools_dp_solver_knapsack_demo.obj
ortools_dp_solver_knapsack_demo.cc

cl /EHsc /MD /nologo /D_SILENCE_STDEXT_HASH_DEPRECATION_WARNINGS -nologo /O2 -DNDEBUG /D__WIN32__
/DNOMINMAX /DWIN32_LEAN_AND_MEAN=1 /D_CRT_SECURE_NO_WARNINGS /DGFLAGS_DLL_DECL=
/DGFLAGS_DLL_DECLARE_FLAG= /DGFLAGS_DLL_DEFINE_FLAG= /Iinclude\\src\\windows /Iinclude /I. -DUSE_CBC -DUSE_CLP
-DUSE_BOP -DUSE_GLOP objs\\ortools_dp_solver_knapsack_demo.obj  lib\\ortools.lib psapi.lib ws2_32.lib
/Febin\\ortools_dp_solver_knapsack_demo.exe

bin\\ortools_dp_solver_knapsack_demo.exe
Total value: 8
Item 3 is in the bag
Item 4 is in the bag
```

```
# ΟΡΓΑΝΩΣΗ ΣΕ ΚΑΤΑΛΟΓΟΥΣ
ortools
|
→ αποσυμπιεσμένη η εγκατάσταση του ORTOOLS
src
|
→ ortools_dp_solver_knapsack_demo.cc
```


https://github.com/chgogos/uoi_algorithms_and_complexity

Πηγές

- Algorithms, S. Dasgupta, C. H. Papadimitriou, and U. V. Vazirani, 2006
- Algorithms Illuminated, Part 3: Greedy Algorithms and Dynamic Programming, Tim Roughgarden, 2019
- https://developers.google.com/optimization