11. Backtracking Algorithms

Introduction to backtracking

Backtracking algorithms are often used to solve *constraint satisfaction* problems. The 0—1 Knapsack Problem is an example problem that can be solved by backtracking. (There are other approaches (greedy, dynamic programming) for this problem.)

Conceptually, backtracking performs a depth-first search of a tree.

Problem 5: 0—1 Knapsack Problem

Given a knapsack with capacity C*, and a list of provisions (an inventory)* $\langle p_k : k \in \mathbb{N} \rangle$ *(the list could be unbounded) A provision* p *is a 3–tuple*

$$p :: (\mathrm{String}, \mathrm{Num}, \mathrm{Num}) = (\textit{name}, \textit{weight}, \textit{value})$$

Decision Problem: Given a value V*, is there a subset* $\mathbb{I}$ *of provisions such that*

$$\sum_{p \in \mathbb{I}} p_{\textit{weight}} \leq C, \quad \textit{and}$$
$$\sum_{p \in \mathbb{I}} p_{\textit{value}} \geq V$$

This decision problem is NP*-complete.*
Function Problem: Find the maximum value of V *over all* feasible *subsets of provisions. A subset is feasible if the sum of its weights does not exceed the capacity* C *of the knapsack. This function problem is* NP*-hard.*

The Partition problem: Can a set integers be partitioned into two non-empty subsets that have equal sums over their values. It is an NP-complete problem. Partition is a special case of the Knapsack problem: Assume each items weight equals its value and $C = V = 1/2 \sum$ weights. Solving Knapsack in this special case solves Partition.

A useful abstraction is to consider both summations to be over the entire inventory (list of provisions). This is accomplished using a a bit vector

$$\langle b_0, b_1, b_2, b_3, \ldots \rangle$$

where a particular bit b_k is set to 1 or 0 depending on whether or not provision p_k is or is not placed in the knapsack. There are several things this reveals.

- There are 2^n bit vectors of length n (or subsets of an n element set

$\mathbb{Z}_n = \{0, 1, \ldots, (n-1)\}$. Exhaustive search will take exponential time.

- Not all subsets need to be explored: Once a subset is *infeasible* so are all of its supersets.

Here is a functional algorithm for the problem. An example inventory is given. The `combs` function that searches over all combinations of provisions from the inventory (subsets of it). The `main` function specifies the input and output.

From The Haskell code is from Rosetta Code

The `combs` function maps a list of provisions and a capacity to and ordered pair: a value and its feasible list of provisions.

As an initial condition, if the provision list is empty, then for any capacity, the returned value is 0 and the empty list.

Given an non-empty list of provisions, there are two possibilities: (1) If the weight of the provision at the head of the list is less than the capacity, then including the provision is feasible, otherwise (2) the provision cannot be in a feasible solution.

In the second case, return the result from the rest of the list and the given capacity. This is the case where b_p, the bit representing the provision, is 0. In the first case, provision may or may not be in the optimal feasible solution. So, compute both cases and return the one that is largest.

Listing 32: Functional 0 –1 Knapsack

```
-- A provision is its name, its weight, and its value
data Provision = (String, Num, Num)

-- Here is a sample inventory of provisions
inventory = [("map",9,150), ("compass",13,35), ("water",153,200),
             ("sandwich",50,160), ("glucose",15,60), ("tin",68,45),
             ("banana",27,60), ("apple",39,40), ("cheese",23,30),
             ("beer",52,10), ("cream",11,70), ("camera",32,30),
             ("tshirt",24,15), ("trousers",48,10), ("socks",4,50),
             ("umbrella",73,40), ("towel",18,12), ("book",30,10),
             ("trousers",42,70), ("overclothes",43,75),
             ("notecase",22,80), ("sunglasses",7,20)]

 -- The combs function searches over feasible solutions to find one
 -- that maximizes the value of provisions
 combs [] _ = (0, [])
 combs ((n,w,v):rest) cap
    | w <= cap  = max (combs rest cap)
                      (prepend (n,w,v) (combs rest (cap - w)))
    | otherwise = combs rest cap
    where prepend (n,w,v) (value, list) = (value + v, (n,w,v):list)

main = do print (combs inventory 400)
```

Given a list of n provisions combs always calls itself again with a list of size $n-1$, and sometimes calls itself twice on the tail of the list with

different capacities. Prepending a triple onto the current optimal value and list takes constant time. Therefore, in the worst case, the code's time complexity can be modeled by the famous Mersenne recurrence

$$T(n) = 2T(n-1) + 1,\ T(0) = 0$$

which has solution $T(n) = 2^n - 1$.

Here is a C (pseudo-code) implementation backtracking for the problem.

A more general model would be

$$T(n) = 2T(n-1) + c,\ T(0) = a$$

which has solution

$$T(n) = 2^n a + (2^n - 1)c$$

I believe comparing the functional and imperative codes clearly shows the difference between understanding the problem versus understanding the problem and the machine.

Listing 33: Imperative 0 –1 Knapsack Backtracking Algorithm

```
#include <stdio.h>
#include <stdlib.h>

typedef struct {
  char *name;
  int weight;
  int value;
} provision;

provision items[] = {
  {"map",          9, 150},
  {"compass",     13,  35},
  {"water",      153, 200},
  ...,
  {"sunglasses",  7,  20}
};

int cap;             // capacity of knapsack
int n;               // number of items
int X[n];            // current array of bits
int optBits[n];      // optimal array of bits
int optValue = 0;

bool isFeasible(provision *items) {
  int sum = 0;
  for (int i = 0; i < n; i++) {
    sum = sum + (X[i] * items[i].weight);
  }
  if (sum <= cap) { return true; }
  else {return false;}
}

bool betterValue(provisions *items) {
  int sum = 0;
  for (int i = 0; i < n; i++) {
    sum = sum + (X[i] * items[i].value);
  }
  if (sum > optValue) {
    optValue = sum;
    return true;
```

```
  }
  else {return false;}
}

int knapsack(provisions *items, int level) {
  if (level == n) {
    if (isFeasible(items)) {
          if (betterValue(items)) {
            optBits = X;
          }
        }
      }
      else {
        X[level] = 1;
        knapsack(items, level + 1);
        X[level] = 0;
        knapsack(items, level + 1,);
      }
    }
```

The above code has time complexity described by

$$\begin{aligned}
T(n) &= 2T(n-1) + n \\
&= 2[2T(n-2) + (n-1)] + n \\
&\vdots \\
&= 2^k T(n-k) + \sum_{i=0}^{k-1} 2^i (n-i) \\
&= \sum_{i=0}^{n-1} 2^i (n-i) \\
&= n(2^n - 1) - \sum_{i=0}^{n-1} 2^i i \\
&= n(2^n - 1) - (2 + 2^n(n-2)) \\
&= 2^{n+1} - n - 2
\end{aligned}$$

Here is another imperative C algorithm for the $0-1$ Knapsack. It uses dynamic programming and comes from Rosetta Code. It makes me not want to be a C programmer. This algorithm is pseudo-polynomial, that is, its time complexity is $T(n) = O(wn)$ where w is the capacity of the knapsack.

Pseudo-polynomial means the time complexity depends as a polynomial in value of a number, the capacity w in this case, but the input size depends on the $\lfloor w \rfloor + 1$, and w is exponential in this size.

Listing 34: Imperative Dynamic Programming 0–1 Knapsack Algorithm

```
#include <stdio.h>
#include <stdlib.h>

typedef struct {
    char *name;
    int weight;
    int value;
} provision;

provision items[] = {
    {"map",                      9,   150},
    {"compass",                 13,    35},
    {"water",                  153,   200},
    ...,
    {"sunglasses",               7,    20}
};

int *knapsack(provision *items, int n, int w) {
    int i, j, a, b, *mm, **m, *s;
    mm = calloc((n + 1) * (w + 1), sizeof (int));
    m = malloc((n + 1) * sizeof (int *));
    m[0] = mm;
    for (i = 1; i <= n; i++) {
        m[i] = &mm[i*(w+1)];
        for (j = 0; j <= w; j++) {
            if (items[i-1].weight > j) {
                m[i][j] = m[i-1][j];
            }
            else {
                a = m[i-1][j];
                b = m[i-1][j-items[i-1].weight]+items[i-1].value;
                m[i][j] = a > b ? a : b;
            }
        }
    }
    s = calloc(n, sizeof (int));
    for (i = n, j = w; i > 0; i--) {
        if (m[i][j] > m[i-1][j]) {
            s[i-1] = 1;
            j -= items[i-1].weight;
        }
    }
    free(mm);
    free(m);
    return s;
}

int main () {
    int i, n, tw = 0, tv = 0, *s;
    n = sizeof (items) / sizeof (provision);
    s = knapsack(items, n, 400);
    for (i = 0; i < n; i++) {
```

```
      if (s[i]) {
        printf("%-22s %5d %5d\n", items[i].name,
                                  items[i].weight,
                                  items[i].value);
        tw += items[i].weight;
        tv += items[i].value;
      }
    }
    printf("%-22s %5d %5d\n", "totals:", tw, tv);
    return 0;
  }
```

Pruning and Bounding Functions

The functional backtracking algorithm for Knapsack 32 *prunes* the search space by only exploring branches where `w <= cap`. The imperative backtracking algorithm ?? explores the entire search space, but the search can be *pruned* by a guard before each recursive call: Does the current weight plus the weight of the next provision not exceed the capacity?

Bounding functions provide more general approaches to pruning.

Let Let $\vec{X} = \langle x_0, x_1, \ldots, x_{k-1}, \ldots \rangle$ be a k-bit string representing a (partial) solution to a constraint satisfaction problem. Let

$$C(\vec{X}) = \left(\sum_{i=0}^{k-1} v_i x_i \right) + \max \left\{ \left(\sum_{i=k}^{n} v_i x_i : \sum_{i=0}^{n-1} w_i x_i \right) \leq C \right\}$$

That is, $C(\vec{X})$ is the maximum value of *feasible descendants (extensions)* of $\vec{X}$. It is the value of the currently selected provisions plus the largest value over the set of feasible extensions (descendants) of $\vec{X}$.

In particular, if $\left|\vec{X}\right| = n$, then $\vec{X}$ is a feasible solution, $C(\vec{X})$ is its value, but $\vec{X}$ may not be optimal. Also, if $\left|\vec{X}\right| = 0$, then $C(\vec{X})$ is the optimal value of the problem.

Definition 12: Bounding Function Properties

A bounding function B *is any function defined on variable length bit strings such that*

- *If* $\vec{X}$ *is a* feasible *solution, the* $C(\vec{X}) = B(\vec{X})$
- *For all* partial feasible *solutions,* $C(\vec{X}) \leq B(\vec{X})$

If such a bounding function $B()$ can be found, and if at any point of the computation $B(\vec{X}) \leq$ `optValue` holds, then no extensions of $\vec{X}$ can lead to an optimal solution. And, searching descendants of $\vec{X}$ can be pruned.

Computing $C(\vec{X})$ is expensive when $\vec{X}$ has many descendants. The bounding function $B()$ should be much easier to compute. And, we want $B()$ be be a good approximation of $C()$.

One trick that can lead to discovering a bounding function is to find a simpler, easier to solve, related problems. A natural approximation to the 0–1 Knapsack problem is the Rational Knapsack problem (RK)

Recall, the greedy approach to the RK problem: Given a list of items, sort them in descending order in their value-to-weight ratios. An item with value 10 and weight 3 is worth more than an item with value 10 and weight 4. This sort only needs to be done once. Assume its time complexity is $O(n \lg n)$.

The greedy algorithm places items in the knapsack in order, one at a time as long as they fit. Some fraction of the last item might need to be used to fill, but not overfill, the knapsack. The time complexity is $O(n)$. See the notes on Greedy algorithms.

Let $\vec{X} = \langle x_0, x_1, \ldots, x_{k-1}, \ldots \rangle$ be a string of k-bit strings representing a (partial) solution to a Knapsack problem. Let $R(k, C')$ be the optimal solution to the Rational Knapsack problem with capacity C', over all rational descendants of $\vec{X}$, that is, $\langle x_k, x_{k+1}, \ldots, x_{n-1} \rangle$ where the values of $x_j \in \mathbb{Q}$, the set of rationals.

Define a bounding function by

$$B(\vec{X}) = \sum_{i=0}^{k-1} x_i p_i + R\left(k, C - \sum_{i=0}^{k-1} x_i w_i\right) = CV + R(k, C - CW)$$

where CV is the current value and CW is the current weight.

That is, $B(\vec{X})$ is the value selected provision from 0 to $k-1$, plus the value that can be gained from the remaining provisions using the remaining capacity and rational x's. When all of the x's are restricted to 0 or 1, then $C(X) = B(X)$. Also, since rational x's yield more freedom (choices), $C(X) \leq B(X)$

Here is an example from (Stinson, 1987).

Example: Use of bounding function

Assume there are 5 items with weights

$$\vec{W} = \langle 11, 12, 8, 7, 9 \rangle$$

and values

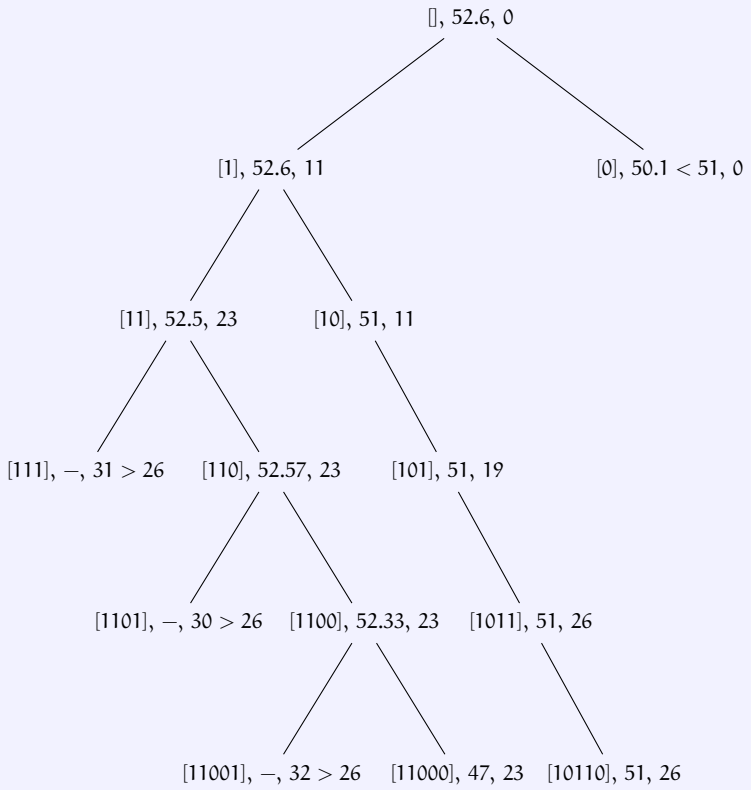
$$\vec{V} = \langle 23, 24, 15, 13, 16 \rangle$$

Pretend the knapsack's capacity of $C = 26$. *The weights and values are sorted by value-to-weight ratio:*

$$\langle 23/11, 24/12, 15/8, 13/7, 16/9 \rangle \approx \langle 2.09, 2, 1.875, 1.857, 1.77 \rangle$$

The search space tree shown below and explained after the diagram. A node is a triple $([xs], B, CW)$, *a list* $[xs]$ *of previously*

set bits, the value of the bounding function B, *and the current weight of the included items.*



Assume that the positive $x = 1$ branch is explored first. The greedy algorithm first computes x's: 1, 1, 3/8 to fill the knapsack. The bounding value is

$$B([]) = 23 + 24 + \frac{3}{8} \cdot 15 = 52.625$$

Now explore the 1 branch:

$$B([1]) = 23 + 24 + \frac{3}{8} \cdot 15 = 52.625, \quad CW = 11$$

- $B([11]) = 23 + 24 + \frac{3}{8} \cdot 15 = 52.625$, $cw = 23$
  - $B([111])$ is infeasible: $cw = 31 > 26 = C$, prune this branch
  - $B([110]) = 23 + 24 + \frac{3}{7} \cdot 13 \approx 52.57$, $cw = 23$ (The left (down) branch [1101] is infeasible: its weight is $CW = 30$)

    The search follows $[110] \mapsto [1100] \mapsto [11000]$: a feasible solution. Along this branch B is updated: $52.57 \mapsto 52.33 \mapsto 47$ and a potential optimal value `optValue=47` is set.
- Now explore the [10] branch. $B([10]) = 23 + 15 + 13 \approx 51$, $cw = 26$

- $B([101]) = 51$, $CW = 26$
  - $B([1011]) = 51$, $CW = 26$
    - $[10111]$ is infeasible: $CW = 11 + 8 + 7 + 9 = 35 > 26 = C$
    - $[10110]$ is feasible: $B([10110]) = 51$, $CW = C$, and a new, better, potential optimal value `optValue=51` is set.
  - $B([100]) = (23) + 13 + \frac{2}{3}16 = 46.6 < 51$, $CW = 11$. Since this value of B is less than the previously computed optimal value 51 this branch can be pruned.

- When the $[0]$ branch is explored, we find

$$B([0]) = 0 \cdot 23 + 24 + 15 + \frac{6}{7} \cdot 13 \approx 50.14 < 51$$

  Since this value of B is less than the previously computed potential optimal solution 51 its entire sub-tree can be pruned

The Traveling Salesman & Hamilton Circuit Problem

Bibliography

Agrawal, M., Kayal, N., and Saxena, N. (2002). Primes is in P. http://www.cse.iitk.ac.in/news/primality.html. [page 205]

Aho, A. V. and Corasick, M. J. (1975). Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18(6):333–340. [page 87]

Apostolico, A. and Galil, Z. (1997). *Pattern Matching Algorithms*. Oxford University Press. [page 53]

Bellman, R. (1957). *Dynamic Programming*. Princeton University Press. [page 141]

Bentley, J. (1984). Programming pearls: Algorithm design techniques. *Commun. ACM*, 27(9):865–873. [page 33]

Boyer, R. S. and Moore, J. S. (1977). A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772. [page 71]

Cook, S. A. (1971). The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA. ACM. [page 199]

Corman, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms*. MIT Press, third edition. [page 9], [page 14], [page 18], [page 21], [page 31], [page 33], [page 39], [page 53], [page 93], [page 105], [page 125], [page 130], [page 131], [page 141], [page 163], [page 169], [page 178], [page 187], [page 193], [page 201]

Garey, M. R. and Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of Intractability*. W. H. Freeman. [page 193], [page 203]

Gödel, K. (1930). Die vollst andigkeit der axiome des logischen funktionenkalk uls. *Monatshefte f ur Mathematik und Physik*, 37:349–360. [page 194]

Gödel, K. (1992). *On Formally Undecidable Propositions of Principia Mathematica and Related Systems*. Dover books on advanced mathematics. Dover Publications. [page 194]

Gusfield, D. (1997). *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. EBL-Schweitzer. Cambridge University Press. [page 53]

Hofstadter, D. R. (1999). *Gödel, Escher, Bach: An Eternal Golden Braid. 20th Anniversary Edition*. Basic Books, New York, New York. [page 25]

Kernighan, B. W. and Ritchie, D. M. (1988). *The C Programming Language*. Prentice Hall, second edition. [page 31]

Knuth, D. E. (1984). Literate programming. *The Computer Journal*, 27(2):97–111. [page 30]

Knuth, D. E. (1993). *The Stanford GraphBase: A Platform for Combinatorial Computing*. Addison-Wesley, ACM Press. [page 147], [page 163]

Knuth, D. E., Morris, J. H., and Pratt, V. R. (1977). Fast pattern matching in strings. *SIAM Journal of Computing*, 6(2):240–267. [page 55]

Lipovaca, M. (2011). *Learn You a Haskell for Great Good!: A Beginner's Guide*. No Starch Press, San Francisco, CA, USA, 1st edition. [page 31]

Papadimitriou, C. H. (1994). *Computational Complexity*. Addison-Wesley. [page 193], [page 195], [page 206]

Patterson, D. A. and Hennessy, J. L. (1996). *Computer Architecture: A Quantitive Approach*. Morgan Kaufmann Publishers, Inc., second edition. [page 30]

Polya, G. (1945). *How To Solve It*. Princeton University Press. ISBN 0-692-02356-5. [page 26]

Rabhi, F. A. and Lapalme, G. (1999). *Algorithms: A Functional Programming Approach*. Addison-Wesley. [page 31]

Ramsey, N. (1994). Literate programming simplified. *IEEE Software*, 11(5):97 – 105. [page 30]

Sankoff, D. and Kruskal, J. B., editors (1983). *Time Warps, String Edits, and Macromolecules*. Addison-Wesley, Reading, Massachusetts. [page 146]

Sedgewick, R. (2004). *Algorithms in Java*. Addison-Wesley. [page 31]

Shallit, J. (1994). Origins of the Analysis of the Euclidean Algorithm. *Historia Mathematica*, 21:401–419. [page 44]

Stansifer, R. (1984). Presburger's article on integer airthmetic: Remarks and translation. Technical Report TR84-639, Cornell University, Computer Science Department. [page 194]

Stinson, D. R. (1987). *An Introduction to the Design and Analysis of Algorithms*. The Charles Babbage Research Center, P. O. Box 272, St. Norbert Postal Station, Winnipeg, Manitoba R3V 1L6, Canada, second edition. [page 125]

Turing, A. M. (1936). On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265. [page 197]

Wilf, H. (2006). *Generatingfunctionology: Third Edition*. AK Peters Series. Taylor & Francis. [page 19]

Index