Laboratory for Concurrent Computing Systems
Technical Report 31-038

Version 1.1 22 October 1992

Five Ways to Fill Your Knapsack

Professor A. P. W. Böhm
Colorado State University
USA

Professor G. K. Egan
Swinburne University of Technology
Australia

Abstract:

We compare five solutions to the zero-one knapsack problem: *Divide and Conquer, Depth First with Bound, Dynamic Programming, Memo Functions, and Branch and Bound.* Our programs are written in Sisal and run on the CSIRAC II dataflow machine. Two of the algorithms, Memo Functions and Branch and Bound, benefit from non deterministic extensions of Sisal, *put* and *get*. We introduce these extensions and compare the performance of the five algorithms using knapsacks of 20 and 40 objects. We measure the performance of our programs in S_1: the number of instructions executed, $S\infty$: the critical path length, and $\pi = [S_1/S\infty]$: the adverage parallelism. It turns out that the Branch and Bound algorithm performs best in terms of S_1, especially for the harder test cases.

LABORATORY FOR CONCURRENT COMPUTING SYSTEMS
COMPUTER SYSTEMS ENGINEERING
School of Electrical Engineering
Swinburne University of Technology
John Street, Hawthorn 3122, Victoria, Australia.

FIVE WAYS TO FILL YOUR KNAPSACK

Wim Böhm, Colorado State University, USA
Greg Egan, Swinburne University of Technology, Australia

October 22, 1992

ABSTRACT

We compare five solutions to the zero-one knapsack problem: *Divide and Conquer, Depth First with Bound, Dynamic Programming, Memo Functions, and Branch and Bound.* Our programs are written in Sisal and run on the CSIRAC II dataflow machine. Two of the algorithms, Memo Functions and Branch and Bound, benefit from non deterministic extensions of Sisal, *put* and *get.* We introduce these extensions and compare the performance of the five algorithms using knapsacks of 20 and 40 objects. We measure the performance of our programs in S_1: the number of instructions executed, S_∞: the critical path length, and $\pi = \lfloor S_1/S_\infty \rfloor$: the average parallelism. It turns out that the Branch and Bound algorithm performs best in terms of S_1, especially for the harder test cases.

FIVE WAYS TO FILL YOUR KNAPSACK

Just blow up the stack Jack
Make a bad call Paul
Hit the wrong key Lee
Set your pointers free

Just mess up the bus Gus
Don't need to recurse much
Just listen to me

....

Kind of by Paul Simon
Courtesy of the net

1 INTRODUCTION

The zero-one knapsack problem is defined as follows. Given n objects with positive weights W_i and positive profits P_i, and a knapsack capacity M, determine a subset of the objects represented by a bit vector X with elements X_1 to X_n, such that

$$\sum_{i=1}^{n} X_i W_i \leq M \quad and \quad \sum_{i=1}^{n} X_i P_i \quad maximal$$

We assume the objects to be sorted by profit weight ratio, as solutions are often close to the *greedy approximation*: grab objects with a maximal profit weight ratio until the knapsack cannot be filled any further.

The knapsack problem gives rise to a search space of 2^n combinations of objects, which can be depicted as a binary tree, where the root represents an empty knapsack, and going from $level_i$ in the tree to $level_{i+1}$ represents either picking $object_i$ (going left down) or not picking $object_i$ (going right down). Given a partial solution (a choice for objects 1 .. i), a lower bound for the best total solution can be computed in linear time by adding objects with maximal profit weight ratio (i.e. objects i+1, i+2, ...) until an object exceeds the knapsack capacity, while an upper bound can be computed by adding part of the object that exceeded the knapsack capacity, such that the knapsack is filled to capacity.

In this paper we compare five solutions to the zero-one knapsack problem: *Divide and Conquer, Depth First with Bound, Dynamic Programming, Memo Functions,* and *Branch and Bound,* written in Sisal and run on the CSIRAC II dataflow machine. Two of these algorithms, Memo Functions and Branch and Bound, need non deterministic extensions of Sisal, *put* and *get.* We introduce these extensions. We compare the performance of the five algorithms using knapsacks of up to 40 objects. We measure the performance of our programs in S_1: the number of instructions executed, S_∞: the critical path length, and $\pi = \lfloor S_1/S_\infty \rfloor$: the average parallelism.

1.1 THE CSIRAC II DATAFLOW MACHINE

The CSIRAC II dataflow computer [1], used in this study, is characterised by random allocation of workload at the node level as distinct from a code block or procedure level; generic node functions; strongly typed, variable length tokens; loop unravelling as well as re-entrant code support using a single undifferentiated colour tag combined with the ability to preserve temporal ordering of tokens without tag manipulation overheads, tokens on any given arc with the same colour being maintained in strict FIFO order; imbedded storage functions for local state information; heterogeneous streams; integrated input/output and error mechanisms. More recent refinements to the architecture have included the addition of vector and compound token types and extensions to matching functions for streams.

2 THE ALGORITHMS

In the following programs W denotes the array of weights, P denotes the array of profits, M the knapsack capacity, n the number of objects, i the level in the search tree, and cp the profit gathered at a particular point in the search tree.

2.1 DIVIDE AND CONQUER

The Divide and Conquer solution to the knapsack problem is better seen as an executable specification. Apart from checking whether the weight of an object exceeds the remaining capacity of the knapsack, the Divide and Conquer algorithm does not prune the search space. As the left-down and right-down searches are independent, this algorithm is highly parallel. But, as is often the case when there is abundant parallelism, a lot of unnecessary work is performed.

```
function knapdc(W,P: array[integer]; i,M,n: integer  returns integer)
if M < W[i] then
if i<n then knap(W,P,i+1,M,n) else 0 end if
else if i<n then
let l := knapdc(W,P,i+1,M-W[i],n)+P[i];
r := knapdc(W,P,i+1,M,n)
in if l > r then l else r end if
end let
else P[i] end if
end if
end function
```

The main function initializes W, P, M and n and calls $knapdc(W, P, 1, M, n)$.

2.2 DEPTH FIRST WITH BOUND

The Depth First with Bound solution computes, in a certain point of the search space, the upper bound given the partial solution, and if this upper bound is less than the best solution found so far, the sub-tree under the partial solution is not further explored. This avoids large amounts of work, but causes the search to proceed depth first left to right, and consequently looses almost all parallelism in the algorithm. It also forces the search to go down the "greedy" path, which may not always be favourable. The Branch and Bound algorithm in section 2.6 deals with these problems. Note that, when going left down (taking $object_i$), the upperbound does not need to be recomputed as it does not change.

```
forward function knapb(W,P: array[integer]; i,cp,M,n,best: integer; returns integer)

function knap(W,P: array[integer]; i,cp,M,n: integer; returns integer)
if (M<W[i]) then
if i<n then knapb(W,P,i+1,cp,M,n,cp) else cp end if
else if i<n then
let l := knap(W,P,i+1,cp+P[i],M-W[i],n);
r := knapb(W,P,i+1,cp,M,n,l)
in max(l,r)
end let
else cp+P[i]
end if
end if
end function

function knapb(W,P: array[integer]; i,cp,M,n,best: integer; returns integer)
let bound :=
for initial
b := cp; cm := M; j := i
repeat
b,cm,j :=
if (old cm >= W[old j])
then old b + P[old j], old cm - W[old j], old j + 1
else old b + (old cm * P[old j])/W[old j],0,n+1
end if
until j > n
returns value of b
end for
in if bound <= best then best else knap(W,P,i,cp,M,n) end if
end let
end function
```

The main function initializes W, P, M and n and calls $knap(W, P, 1, 0, M, n)$.

2.3 DYNAMIC PROGRAMMING

The dynamic programming solution to the knapsack problem combines solutions of sub-problems bottom-up, saving answers to sub-problems in a vector V_i. At $stage_i$ in the computation, V_i contains solutions to problems with knapsack capacity 0 to M using objects 1 to i only. An element of V_i can be expressed in terms of elements of vector V_{i-1}:

$$V_i[j] = max(V_{i-1}[j], P[i] + V_{i-1}[j - W[i]])$$

The term $V_{i-1}[j]$ represents the choice of not taking $object_i$, the term $P[i]+V_{i-1}[j-W[i]]$ represents the choice of picking $object_i$.

```
function knapdp (W,P: array[integer]; M,n: Integer returns integer)
let FinalV :=
for initial
i := 0;  V := array_fill (0,M,0);
repeat
i := old i + 1; Pi := P[i]; Wi := W[i];
V := for v1 in old V at j
nv := if j >= Wi
then let v2 := old V[j-Wi] + Pi in max(v1,v2) end let
else v1
end if
returns array of nv
end for
until i = n
returns value of V
end for
in FinalV[M]
end let
end function
```

The main function initializes W, P, M and n and calls $knapdp(W, P, M, n)$. The algorithm computes $M * n$ values, each value takes constant time to compute, so knapdp has an S_1 complexity of $O(M * n)$. Also, this is the only algorithm with potential for vectorization.

2.4 TAGGED MEMORY, LOCKS, and NON DETERMINISM

In a dataflow machine, asynchronous structure accessing is implemented using split-phase read and write operations and storage cells augmented with two *tag* bits: a *presence bit* P and a *defer bit* D.

```
READ ( cell: storage-cell returns number):
if cell.P
then return cell.VAL
else cell.D := True;
enqueue the READ request using cell.VAL as a pointer
end if

WRITE ( cell: storage-cell, val: number):
if cell.P then ERROR
else if CELL.D
then honour ALL requests in the defer queue
end if;
cell.P := True;
cell.VAL := val
end if;
```

Until now we have been able to express our algorithms in standard Sisal. The implementations of the Memo Functions and Branch and Bound algorithms require non determinism and can therefore not be expressed in pure Sisal. We will use the *side effecting* operations *put* and *get* for this. The combination of *put* and *get* provides for light weight locks, using the presence and defer bits of tagged memory. *Get* reads a value from a storage cell and resets the presence bit, in other words, it reads and wipes out a value from storage. *Put* writes a value in a storage cell, in such a way that only one *get* can "get" it. If there are no deferred accesses, put just performs a write. If there are deferred accesses, put honours *one* request, leaves the cell empty and the rest of the accesses deferred.

```
GET ( cell: storage-cell returns number):
if cell.P
then cell.P := False; return cell.VAL
else cell.D := True;
enqueue the GET request using cell.VAL as a pointer
end if;

PUT ( cell: storage-cell, val: number ):
if cell.P then ERROR
else if cell.D
then honour ONE request in the defer queue
else cell.P := True; cell.VAL := val
end if;
```

The *get* and *put* functions are supported directly by the structure-read-and-reset (srr) and the structure store-write-read-once (srw) instructions of the CSIRAC II. These instructions have been used for some time in the runtime resource management library for CSIRAC II [1]. The following is the implementation of *put* and *get* in the intermediate code i2 [2] used in the Sisal to CSIRAC II compiler.

```
define __get(index)->value;
begin
srr(index) -> value;
end;

define __put(index, value)->acknowledge;
begin
srw(value index) -> pip_gate;
pip(value pip_gate) -> acknowledge;
end;
```

where pip stands for "pass if present".

2.5 MEMO FUNCTIONS

The memo function solution to knapsack combines the divide-and-conquer and dynamic programming methods. A table is maintained containing all sub-solutions. The table elements are initialized to -1 to indicate that computation of the solution to the particular sub-problem has not been started.

```
function knapm(Pad,W,P: array[integer]; i,M,n: integer returns integer)
let Pos:=M*n+i; PP := get(Pad,Pos);
BP := if OldP ~= -1 then PP
else if M < W[i] then
if i<n then knapm(Pad,W,P,i+1,M,n) else 0 end if
else if i<n then
let l := knapm(Pad,W,P,i+1,M-W[i],n)+P[i];
r := knapm(Pad,W,P,i+1,M,n)
in max(l,r)
end let
else P[i]
end if
end if
end if
in put(Pad,Pos,BP)
end let
end function
```

The main function creates a table *Pad* containing $(M+1)*n$ elements initialized to -1. The semantics of *put* and *get* ensure that only one process at the time will get the value of a certain table element. If it is -1 the process will compute the solution to the particular sub-problem and put the solution back. If the element is not -1, it has been computed, so the process puts it back in the table. Other processes needing this solution will be deferred until the solution is put back. As in the dynamic programming algorithm, the total amount of work is $O(M*n)$. $O(M*n)$ table

elements are computed. As the number of non deferred processes going down from $level_i$ to $level_{i+1}$ is at most M, at most $O(M * n)$ processes can get deferred.

2.6 BRANCH AND BOUND

The Branch and Bound algorithm exploits parallelism to implement branching, which means that the state space is searched breadth first. This avoids the drawbacks of the depth first with bound algorithm. Notice the absence of explicit queueing in the algorithm. Sub-trees are cut by estimating the upperbound of a partial solution and comparing it to a *shared variable GLow* containing the current best lower bound, maintained with *puts* and *gets*, ensuring that only one process can get GLow, use it and write an updated value back.

```
function knapbb (GLow, W, P: Vector; i, cp, M, n: integer returns integer)
if i > n | M=0  then cp
else
let L, U :=
for initial % compute lower and upper bound
cl := cp;  cu := cp;  cm := M; j  := i
repeat
cl, cu, cm, j :=
if old cm >= W[old j]
then old cl + P[old j],  old cu + P[old j],
old cm - W[old j],  old j + 1
else old cl, old cu + ((old cm * P[old j]) / W[old j]),
old cm, n+1
end if
until j > n
returns value of cl value of cu
end for;
GL := get(GLow,1);
GB := put(GLow,1,max(GL,L));
in
if U < GB
then 0
else if M >= W[i]
then let l := knapbb(GLow, W, P, i+1, cp+P[i], M-W[i], n);
r := knapbb(GLow, W, P, i+1, cp, M, n)
in if l > r then l
else r end if
end let
else knap (GLow, W, P, i+1, cp, M, n)
end if
end if
end let
end if
end function
```

2.7 Make that six: FUNCTIONAL BRANCH AND BOUND

We can make the above Branch and Bound algorithm functional by going down the tree breadth first, creating a set of "viable tasks" for the next level down, using the same lower and upperbound computation, but comparing this not to a global shared variable, but to the best solution found in the previous level. A task is represented by two integers: a *current profit* and a *capacity left*, and a next level in the tree is therefore represented by two arrays of integers.

```
type Vector = array[integer];
function bstep( W, P, profits, capacities: Vector; i,n, best: integer
returns integer,vector,vector)
for pr in profits dot m in capacities
lwb, prfs, caps :=
if i > n   then  best, array vector [], array vector []
else let L, U :=
for initial             % Greedy algorithm
cl := pr;  cu := pr;  cm := m; j  := i
repeat  cl, cu, cm, j :=
if old cm >= W[old j]
then old cl + P[old j], old cu + P[old j], old cm - W[old j], old j + 1
else old cl, old cu + ((old cm * P[old j]) / W[old j]), old cm, n+1
end if
until j > n
returns value of cl value of cu
end for
in if U < best then  best, array vector [], array vector []
else   if m >= W[i]
then L, array vector[1: pr+P[i],pr], array[1: m-W[i],m]
else L, array vector[1: pr], array[1: m]
end if end if
end let
end if
returns value of greatest lwb    value of catenate prfs   value of catenate caps
end for
end function

function main (returns integer)
let n := ... ; M := ... ;   W := array[1: ... ];   P := array[1: ... ];
InitProfit := Array vector[1:0];    InitCap := Array vector[1:M];
in for initial   profits := InitProfit;  capacities := InitCap; i := 1 ; best := 0
while i <= n repeat
best, profits, capacities :=  bstep(W,P,old profits,old capacities,old i,n, old best);
i := old i + 1
returns value of best
end for
end let
end function  % main
```

$n = 20$			S_1				
Off	Var	M	BB	FBB	DP	MF	DF
20	20	499	36325	47703	188329	485926	*15089*
20	16	432	212014	278080	163284	417389	*98286*
20	12	407	178552	225455	153944	378371	*21781*
20	8	384	142188	177388	145349	358291	*103834*
20	4	344	113578	141805	130394	194516	*48186*
40	20	819	87822	107098	307929	651997	*11731*
40	16	752	28089	37673	273544	584245	*10027*
40	12	727	28161	38007	273544	493647	*9553*
40	8	704	47822	64345	264949	384940	*10358*
40	4	64	55459	72243	264949	212993	*10557*

Table 1: S_1 for $n = 20$

$n = 20$			S_∞					π				
Off	Var	M	BB	FBB	DP	MF	DF	BB	FBB	DP	MF	DF
20	20	499	3412	4805	4053	*823*	1585	10	10	46	*590*	9
20	16	432	5458	15194	3517	*819*	5006	38	18	46	*509*	7
20	12	407	4123	10577	3328	*819*	2683	43	21	46	*462*	8
20	8	384	4132	9675	3154	*819*	11884	34	18	46	*437*	8
20	4	344	3900	8294	2845	*821*	5715	29	17	45	*236*	8
40	20	819	3521	5636	6528	*815*	992	25	19	47	*800*	12
40	16	752	3192	4117	6001	*811*	695	9	9	47	*720*	14
40	12	727	3320	4117	5800	*811*	470	8	9	47	*609*	20
40	8	704	3473	5317	5619	*807*	655	14	12	47	*477*	16
40	4	64	3444	5377	5305	*821*	661	16	13	47	*259*	16

Table 2: S_∞ and π for $n = 20$

3 EVALUATION

The knapsack problem instances are created by a C program with the following input parameters:

```
N   - number of candidate items
P   - capacity of knapsack as percentage of total weights
Off - minimal weight and profit of an object
Var - variance in weight and profit of an object
Off + Var is the maximal weight and profit of an object
S   - random number seed
```

The arrays are sorted with highest profit to weight ratio first. The *Off* and *Par* parameters allow to vary the discrepancy between the objects with the highest and lowest profit weight ratio.

$n = 40$			S_1		S_∞		π	
Off	Var	M	BB	DF	BB	DF	BB	DF
40	20	1585	85658	*35859*	10076	*2009*	9	*18*
40	16	1517	*1086312*	15705956	*22142*	2505189	*49*	6
40	12	1440	*987012*	1221636	*22561*	223835	*44*	5
40	8	1391	1223943	*435916*	*21814*	80978	*56*	5
40	4	1328	*5528756*	25377336	*74800*	3663355	*74*	7

Table 3: S_1, S_∞ and π for $n = 40$

With a small *Off* parameter and a large *Var* parameter it is possible to have a knapsack with "diamonds" and "bricks" at the same time.

We have run our programs for several knapsacks with 20 and 40 objects. The capacity of the knapsacks is always 80% of the total weight of the objects. Even for n = 20, the divide and conquer algorithm is unbearably inefficient, so we will not include its results in our tables. In the tables BB stands for Branch and Bound, FBB for functional Branch and Bound, DP for Dynamic Programming, MF for Memo Functions, and DF for Depth First with Bound. The winning value in a certain category is emphasized. For n = 20 we have used five knapsacks with an *Off* parameter of 20, and five with an *Off* parameter of 40 in table 1 and table 2, varying *Var* from 20 down to 4 with steps of 4. FBB is less efficient than BB for two reasons: the bound in FBB is not as good as in BB because it only takes points in the search space on a previous level into account, and FBB uses an expensive reduction operator: *value of catenate*. The case of *Off* = 40 shows the dependence on capacity in the case of the Dynamic Programming and Memo Functions algorithms.

Notice that, in the case of n = 20, the Depth First with Bound algorithm performs well in terms of total work, and that the Memo Functions algorithm exposes the most parallelism. For n = 40, the Functional Branch and Bound, Dynamic Programming and Memo Functions algorithms execute too many instructions to finish in a reasonable amount of time. The only algorithms that are efficient enough in terms of S_1 are Branch and Bound and Depth First with Bound. Table 3 shows the results for n = 40.

The trend seems to be that the harder the problem becomes, the better the Branch and Bound performs in terms of S_1, even though it does not show too much parallelism.

4 CONCLUSION

We have studied a number of algorithms solving the zero-one knapsack problem. These algorithms are written in Sisal and run on the CSIRAC II dataflow machine. Two of these algorithms use non deterministic extensions *put* and *get*, which allow for locking and updating of storage cells. One of these, the Branch and Bound algorithm, performs the best in terms of S_1, for a number of our test

cases. Also, the most parallel algorithm, the divide and conquer algorithm performs the worst in terms of S_1.

References

[1] Egan, G.K., N.J. Webb and A.P.W. Böhm , 'Some Features of the CSIRAC II Dataflow Machine Architecture', in Advanced Topics in Data-Flow Computing, Prentice-Hall 1991, pp143-173.

[2] Egan, G.K., Rawling, M.J. and Webb, N.J., 'i2: An Intermediate Language for the CSIRAC II Data Flow Computer', Technical Report 31-002, Laboratory for Concurrent Computing Systems, School of Electrical Engineering, Swinburne Institute of Technology,1990