Computing

Archiv
für elektronisches
Rechnen

Archives
for Electronic
Computing

Herausgegeben von:
Edited by:

E. Bukovics, Wien
R. Inzinger, Wien
W. Knödel, Stuttgart
C. C. Elgot, Yorktown Heights

Vol 5 / 1970

Springer-Verlag
Wien/New York

Computing 5, 324—331 (1970)

Algorithmus 13

Lösung eines Optimum-Mix-Problemes

Von

R. Albrecht und B. Buchberger, Innsbruck

(Eingegangen am 17. September 1969 in verbesserter Fassung am 2. Januar 1970)

```
procedure optmix (m, n, job, jend, a, h, limit, rmax, f, epsilon, strat, r,
  cost, bmin, furth, pi);
  value m, n, limit, rmax, epsilon;
  integer m, n, limit, rmax, r, furth, pi;
  real cost, bmin, epsilon;
  integer array job, jend, h, a, strat; real procedure f;

comment
```

ALGOL-Prozedur zur Lösung des Optimum-Mix-Problemes bei nichtlinearen, monotonen, konvexen Kostenfunktionen nach R. ALBRECHT: Gegeben sind m Aufgaben und n Hilfsmitteltypen. Für gewisse Paare (i, k) sei die i-te Aufgabe durch a [i, k] Einheiten von Hilfsmitteln des k-ten Hilfsmitteltyps lösbar. Die a [i, k] sind die definierten Elemente der Zuordnungsmatrix. Jede Aufgabe sei durch Hilfsmittel mindestens eines Typs lösbar. Für jeden Hilfsmitteltyp ist eine mit zunehmender Anzahl von Einheiten des Hilfsmittels streng monoton zunehmende nach oben konvexe Kostenfunktion gegeben.
Alle Aufgaben sollen so gelöst werden, daß die Gesamtkosten minimal werden.

Benötigte Unterprogramme

f — real procedure, abhängig von 2 Parametern, wobei der erste (vom Typ integer) den Hilfsmitteltyp, der zweite (vom Typ integer) den Argumentwert der Kostenfunktion für diesen Hilfsmitteltyp angibt.

Eingangsparameter

m — Zeilenanzahl der Zuordnungsmatrix = Anzahl der Aufgaben.

n — Spaltenanzahl der Zuordnungsmatrix = Anzahl der Hilfsmitteltypen.

limit — Ist die Anzahl der nach der Reduktion verbleibenden Strategien größer als limit, dann wird eine Näherungslösung bestimmt, andernfalls wird durch systematische Prüfung eine optimale Lösung ermittelt.

rmax — Gibt an, wieviele optimale Strategien maximal gespeichert werden sollen, falls mehr als eine optimale Strategie gefunden wird.

epsilon — Relative Maschinengenauigkeit (= 5 * 10 power − (p − 1). p gibt die Anzahl der Stellen in der Mantisse an).

Übergangsparameter

a [1 : k] — Enthält alle definierten Elemente der Zuordnungsmatrix, zeilenweise gespeichert. k = Anzahl aller dieser Elemente.

h [1 : k] — Enthält die zu den im Feld a gespeicherten Elementen gehörigen Nummern der Hilfsmitteltypen.

job [1 : m] — Die Werte in job geben für jede Zeile der Zuordnungsmatrix an, an welcher Stelle im Feld a das erste Element der Zeile steht.

jend [1 : m] — Die Werte in jend geben für jede Zeile der Zuordnungsmatrix an, an welcher Stelle im Feld a das letzte Element der Zeile steht.

Ergebnisse

r — Anzahl der abgespeicherten optimalen Strategien. Wird eine Näherungsstrategie berechnet, so erhält r den Wert 1.

cost — Kosten, die eine optimale bzw. eine Näherungsstrategie verursacht.

bmin — Untere Schranke für cost.

furth — Gesamtzahl der optimalen Strategien − rmax, falls die Gesamtanzahl größer als rmax ist, sonst 0. Falls nur eine Näherungsstrategie berechnet wurde, erhält furth den Wert − 1.

pi — Gibt die Anzahl der nach der Reduktion noch verbleibenden möglichen Strategien an.

strat [1 : rmax, 1 : m] — Enthält maximal rmax optimale Strategien. Die i-te Spalte gibt die Nummer des Hilfsmitteltyps zur Lösung der i-ten Aufgabe an.

Methode

Iteratives Reduktionsverfahren, das darin besteht, aufgrund des Zutreffens gewisser Kriterien möglichst viele Elemente der Zuordnungsmatrix als für eine optimale Strategie nicht in Frage kommend auszuscheiden. Die nach der Reduktion verbleibenden Strategien werden entweder systematisch durchprobiert oder es wird, falls deren Anzahl größer als die vorgegebene Schranke limit ist, eine Näherungsstrategie berechnet;

```
begin
  integer repet, k, i, p, p 1, change, str, aux 1, aux 2, aux 3, aux 4, aux 5,
    aux 6;
```

```
  real cost 1, minim, mi, sf 1, sfs 1, aux 7, amin;
  integer array sw, s, l [1 : n], st, sup [1 : m];
  array sf, sfs [1 : n], min [1 : m];
comment
  l (i) enthält die Summe aller Elemente der Zuordnungsmatrix, die in
  der i-ten Spalte stehen und die einzigen definierten Elemente in ihrer
  Zeile sind;

procedure lreg (i 1);
value i 1; integer i 1;
comment
  Globale Größen: m, f, job, jend, min, a, h, l, strat. Die Prozedur lreg
  dient zum eventuellen Neubestimmen der l (i), falls bei der Streichung
  eines Matrixelementes sich die Anzahl der in dieser Zeile definierten
  Elemente auf 1 reduziert. Es wird außerdem geprüft, ob sich die Minima
  der rechten Seiten von (2.2) und (2.4) (siehe [1], S. 256, 257) allenfalls
  verkleinern;

begin
  integer aux 1, aux 2, aux 3, aux 4, i, p;
  real minim;
  if job [i 1] equal jend [i 1] then
  begin
    aux 1 := job [i 1]; aux 2 := h [aux 1];
    l [aux 2] := l [aux 2] + a [aux 1];
    for i := 1 step 1 until m do
    begin
      aux 1 := job [i]; aux 3 := jend [i];
      if aux 1 equal aux 3 then goto a 2;
      for p := aux 1 step 1 until aux 3 do
      if h [p] equal aux 2 then
      begin
        aux 4 := a [p];
        minim := aux 4 * f (aux 2, l [aux 2] + aux 4);
        if minim less min [i] then
        begin
          min [i] := minim;
          strat [l, i] := p
        end
      end;
a 2: end
  end
end lreg;
comment
  Im folgenden Programmteil bis zur Marke iteration werden die An-
```

fangswerte der folgenden Ausdrücke l (k), s (k), sf (k), sfs (k), (k = 1, . . ., n) berechnet:

l (k) Wie oben.

s (k) Enthält die Summe aller Elemente, die in der k-ten Spalte stehen.

sf (k) Gibt zum Argumentwert s (k) den zugehörigen Funktionswert der k-ten Kostenfunktion an.

sfs (k) Enthält die Produkte sf (k) * s (k).

Weiters werden die Minima der rechten Seiten von (2.2) und (2.4) (siehe [1]) bestimmt;

```
epsilon := 1.0 + epsilon;
r := 1;
for k := 1 step 1 until n do l [k] := s [k] := 0;
for i := 1 step 1 until m do if job [i] equal jend [i] then
begin
  aux 1 := job [i];
  aux 2 := h [aux 1];
  l [aux 2] := l [aux 2] + a [aux 1]
end;
for i := 1 step 1 until m do
for p := job [i] step 1 until jend [i] do
begin
  aux 1 := h [p];
  s [aux 1] := s [aux 1] + a [p]
end;
for k := 1 step 1 until n do
begin
  sf [k] := f (k, s [k]);
  sfs [k] := s [k] * sf [k]
end;
for i := 1 step 1 until m do
begin
  p := job [i];
  aux 1 := jend [i];
  aux 2 := h [p];
  aux 5 := a [p];
  minim := aux 5 * f (aux 2, if p equal aux 1 then 1 [aux 2] else
  l [aux 2] + aux 5);
  str := p;
  for p 1 := p + 1 step 1 until aux 1 do
  begin
    aux 2 := h [p 1];
    aux 5 := a [p 1];
    mi := aux 5 * f (aux 2, l [aux 2] + aux 5);
    if mi less minim then
    begin
      minim := mi;
```

```
        str := p 1
      end
    end;
    min [i] := minim;
    strat [1, i] := str
  end;

  comment
    Mit den folgenden drei Laufanweisungen wird geprüft, ob der Satz 2
    von [1] anwendbar ist;

  iteration:
  repet := 0;
  for k := 1 step 1 until n do sw [k] := 0;
  for i := 1 step 1 until m do
  begin
    minim := min [i];
    for p := job [i] step 1 until jend [i] do
    begin
      aux 3 := h [p];
      if sw [aux 3] less 0 then goto a 1;
      if a [p] * sf [aux 3] notgreater minim * epsilon then
      sw [aux 3] := - 1;
  a 1: end
  end;
  for i := 1 step 1 until m do
  begin
    aux 2 := jend [i];
    p := job [i];
  b 1: if sw [h [p]] equal 0 then
    begin
      repet := 1;
      a [p] := a [aux 2];
      h [p] := h [aux 2];
      if aux 2 equal strat [1, i] then strat [1, i] := p;
      jend [i] := aux 2 := aux 2 - 1;
      p := p - 1;
      lreg (i)
    end;
    p := p + 1;
    if p notgreater aux 2 then goto b 1;
  end;

  comment
    Mit der folgenden Laufanweisung wird geprüft, ob Satz 3, Fall (III)
    von [1] anwendbar ist;
```

```
for i := 1 step 1 until m do
begin
  aux 2 := jend [i];
  p := job [i];
b 2: begin
    aux 3 := h [p];
    aux 5 := s [aux 3] — a [p];
    sf 1 := f (aux 3, aux 5);
    sfs 1 := aux 5 * sf 1;
    if sfs [aux 3] — sfs 1 greater min [i] * epsilon then
    begin
      repet := 1;
      a [p] := a [aux 2];
      h [p] := h [aux 2];
      if aux 2 equal strat [1, i] then strat [1, i] := p;
      jend [i] := aux 2 := aux 2 — 1;
      p := p — 1;
      s [aux 3] := aux 5;
      sf [aux 3] := sf 1;
      sfs [aux 3] := sfs 1;
      lreg (i)
    end
  end;
  p := p + 1;
  if p notgreater aux 2 then goto b 2;
end;

comment
  Falls beim letzten Durchgang der bei der Marke iteration beginnenden
  Schleife die Streichung mindestens eines Matrixelements möglich war
  (die Größe repet hat in diesem Fall den Wert 1), so wird der der Marke
  iteration folgende Programmteil wiederholt;

if repet equal 1 then goto iteration;

comment
  Dieser Programmteil dient zur Berechnung einer unteren Schranke
  bmin für die Kosten;

bmin := 0;
for i := 1 step 1 until m do
begin
  aux 4 := job [i];
  aux 1 := h [aux 4];
  amin := a [aux 4] * sf [aux 1];
  if aux 4 equal jend [i] then goto k 1;
  for p := aux 4 step 1 until jend [i] — 1 do
  begin
```

```
    aux 2 := h [p + 1];
    aux 7 := a [p + 1] * sf [aux 2];
    if aux 7 less amin then amin := aux 7;
  end;
k 1: bmin := bmin + amin
end;

comment
  Es folgt die Berechnung der Anzahl der noch verbleibenden Strategien;
pi := 1;
for i := 1 step 1 until m do pi := pi * (jend [i] - job [i] + 1);

comment
  Wenn die Anzahl der noch verbleibenden Strategien kleiner oder gleich
  der angegebenen Schranke limit ist, werden alle Strategien systematisch
  durchprobiert, andernfalls sofort die Kosten der Näherungsstrategie,
  die aufgrund der speziellen Vorgangsweise der Prozedur bereits in
  strat [1, 1 : m] gespeichert ist, berechnet;
for k := 1 step 1 until n do s [k] := 0;
for i := 1 step 1 until m do
begin
  aux 1 := strat [1, i];
  aux 2 := h [aux 1];
  s [aux 2] := s [aux 2] + a [aux 1]
end;
cost := 0;
for k := 1 step 1 until n do
begin
  aux 6 := s [k];
  sf [k] := aux 6 * f (k, aux 6);
  cost := cost + sf [k]
end;
furth := - 1;
if pi greater limit then goto exit;
r := 1; change := m; furth := 0;
for i := 1 step 1 until m do
st [i] := sup [i] := strat [1, i];
cost 1 := cost;
next:
aux 1 := st [change] + 1;
if aux 1 greater jend [change] then
aux 1 := job [change];
if aux 1 notequal sup [change] then
begin
  aux 2 := st [change];
  aux 3 := h [aux 2];
  st [change] := aux 1;
```

```
    aux 4 := h [aux 1];
    aux 5 := s [aux 3] := s [aux 3] − a [aux 2];
    aux 6 := s [aux 4] := s [aux 4] + a [aux 1];
    cost 1 := cost 1 − sf [aux 3] − sf [aux 4];
    sf [aux 3] := aux 5 * f (aux 3, aux 5);
    sf [aux 4] := aux 6 * f (aux 4, aux 6);
    cost 1 := cost 1 + sf [aux 3] + sf [aux 4];
    change := m;
    if cost 1 * epsilon less cost then
    begin
      r := 1; cost := cost 1; furth := 0;
      for i := 1 step 1 until m do
      strat [1, i] := st [i]
    end
    else if cost 1 notgreater cost * epsilon then
    begin
      if r + 1 greater rmax then furth := furth + 1
      else
      begin
        r := r + 1;
        for i := 1 step 1 until m do strat [r, i] := st [i]
      end
    end;
    goto next
end
else
if change − 1 less 1 then goto exit
else
begin
    sup [change] := st [change];
    change := change − 1;
    goto next
end;
exit:
for i := 1 step 1 until r do
for k := 1 step 1 until m do
strat [i, k] := h [strat [i, k]]
end optmix;
```

Literatur

[1] Albrecht, R.: Zum Optimum-Mix-Problem bei nichtlinearen monotonen Kostenfunktionen. Unternehmensforschung **11**, 4, 253–258 (1967).

[2] Burkard, R.: Untersuchungen zum Optimum-Mix-Problem. Erscheint demnächst in Unternehmensforschung.

Prof. Dr. R. Albrecht
Dr. B. Buchberger
Institut für Rechentechnik
der Universität Innsbruck
Innrain 52, A-6020 Innsbruck
Österreich