New Directions for Solving Integer Optimization Problems
Richard Barr

Outline

• Optimization
  – Current research
  – Parallel B&B, B&C
  – Integer simplex
• Heuristics
  – Approximation algorithms
  – Ant-colony optimization
  – Scatter search
• Constraint programming
Integer-Programming Optimization

Integer Optimization Research

- More cutting-plane/branch & bound hybrids
- Specialty algorithms, for problems with special structure
  - Network-related problems
  - 0-1 integer programs
Parallel Branch-and-Bound

Simultaneous Optimization of Different Candidate Problems

Parallel Processing

• The use of multiple processors (CPUs) for a single application

• Goals
  – Smaller solution times
  – Linear speedup (or better): $S_p = p$, where

$$S_p = \frac{\text{Solution time using 1 processor}}{\text{Solution time using } p \text{ processors}}$$
Relative Speedups

![Relative Speedups Graph]

Parallel Architectures

- Shared memory
- Distributed processing

![Parallel Architectures Diagram]
Designing Parallel Programs

- Successful parallel-processing applications
  - Decompose work into large subtasks
  - Distribute work evenly across all processors
    - Have good load balancing
  - Minimize overhead
    - Resource contention, inter-task communication
  - Minimize dependencies
    - Starting task A depends on completion of task B

Parallel B&B, B&C

- A natural application of parallel processing
  - Both tightly coupled and distributed designs
- Decomposition: by (CP)
- Load balancing: trivial with shared memory, can be accomplished in distributed systems
- Overhead: minimal with shared memory
- Dependencies: none, really
Current Trends

- Distributed architecture the *de facto* standard
  - Loosely coupled workstations
  - Networks of PCs
- Development of fault-tolerant software
  - Minimal impact of lost processor(s)
- Branch & cut, heavy pre-processing heuristics
- Most applications show near-linear speedup
- Parallel commercial B&B/B&C solvers

Future Research

- Cuts in B&C: where, when, what type?
- Hybrid architectures
  - Network of shared-memory PCs
- Diverse processing strategies
  - Different branching and separation rules on different processors work better than universal rules (Barr and Stripling, 1996)
  - Benefits from each strategy’s strengths
    - Depth-first
    - Best-bound, simultaneously
Integer Simplex

Consider the following IP problem:

P1: \[ \min cx \]
subject to: \[ Ax \geq 1 \]
\[ x \in \{0,1\} \]

where
\[ A \] is a matrix of 1s and 0s, and
\[ 1 \] is a vector of 1s
Integer Simplex

- Using the simplex method,
  - If the pivot element for an incoming variable is always a 1
  - Then
    - The nonbasic variable will enter with a value of 1
    - There is no division performed in executing the pivot
    - The basic variables will remain 0 or 1

- If a basic solution with all $c_j - z_j \geq 0$ can be found via with this integer pivoting rule (IPR), it is optimal for the IP
- If no improving pivots can be performed via the IPR and some $c_j - z_j < 0, j \in J$
  - The optimal solution to P1 can be found by solving a series of integer subproblems, one for each $j \in J$, requiring $x_j = 1$
Set-Covering Example

- 0-1 integer program
- All constraint coefficients 0 or 1
- All RHS are 1
- Minimize cost of covering all constraints

Minimize \[ \sum c_j x_j \]

subject to: \[ \sum a_{ij} x_j \geq 1 \]

\[ x_j \in \{0,1\} \]

Initial Tableau (\( y_i = \text{artificial} \))

<table>
<thead>
<tr>
<th>Basis</th>
<th>( x_B )</th>
<th>( x_1 )</th>
<th>( x_2 )</th>
<th>( x_3 )</th>
<th>( x_4 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( y_1 )</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>( y_2 )</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>( y_3 )</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>( c_j - z_j )</td>
<td>0</td>
<td>-9</td>
<td>-19</td>
<td>-19</td>
<td>-19</td>
</tr>
</tbody>
</table>
Phase II (Local) Optimum

<table>
<thead>
<tr>
<th>Basis</th>
<th>( x_B )</th>
<th>( y_1 )</th>
<th>( y_2 )</th>
<th>( y_3 )</th>
<th>( x_4 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x_1 )</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-1</td>
<td>2</td>
</tr>
<tr>
<td>( x_2 )</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>( x_3 )</td>
<td>0</td>
<td>0</td>
<td>-1</td>
<td>1</td>
<td>-1</td>
</tr>
<tr>
<td>( c_j - z_j )</td>
<td>-2</td>
<td>9</td>
<td>9</td>
<td>10</td>
<td>-1</td>
</tr>
</tbody>
</table>

For a Global Optimum

- Subproblems are formed by eliminating rows and columns meeting a consistency requirement
- Setting previously un-pivotable variables (due to the IPR) to 0 or 1
- Reoptimizing using the integer simplex, which may yield other subproblems
- Best integer solution found is globally optimum
Heuristics

Approximation Algorithm

- Polynomial-time algorithm
- Computes feasible solutions to NP-hard problems
- Is within a known “distance” from the optimal solution
Binary Knapsack Algorithm

• Binary knapsack problem:

\[ \text{Max } px, \text{ s.t. } ax \leq b, \ x = \{0, 1\} \]
where all \( a_i \geq 0 \)

• Greedy algorithm:
  – Choose \( x_i = 1 \) in decreasing \( p_i/a_i \) order as long as the constraint is not violated
  – Set remaining \( x_i = 0 \)

Approximation Analysis

• Define

\[ z_{Gr} = \text{value of solution found with the greedy} \]
\[ z_H = \max(z_{Gr}, \text{largest } p_i) \]
\[ z^* = \text{optimal solution to the IP} \]

• It can be shown that:

\[ \frac{z^*}{z_H} < 2 \]
Minimum Metric TSP

- Consider the traveling-salesman problem that
  - Is symmetric
  - Observes the triangle inequality
- For the nearest-neighbor algorithm for the $n$-city MMTSP
  \[
  \frac{z_{NN}}{z^*} \leq \frac{1}{2} \left( \lceil \log_2 n \rceil + 1 \right)
  \]

New Metaheuristics

- Immune-system methods
  - Emulate the human immune system
- Memetic algorithms
  - Combine genetic algorithms with local search
- Ant-colony optimization
- Scatter search
Ant-Colony Optimization

- ACO is based on observing ant colonies and their search for food
- ACO algorithms use a colony of ants to solve optimization problems

Ant-Colony Optimization

- Problems are organized as networks to be explored
- Ants leave chemical *pheromone* on their paths to and from food sources
- Each arc \((i,j)\) has variable \(\tau_{ij}\), intensity of artificial pheromone trail, initially 0
Ant-Colony Optimization

- At decision points, they \textit{probabilistically} choose the path with the most pheromone.
- Shorter paths are updated more quickly than longer ones, increasing their probability of selection.

\begin{align*}
\text{Probability of ant } k \\
\text{at node } i \text{ moving to node } j \text{ on } (i,j) \text{ is:} \\
\quad \quad \quad \quad \quad \quad p_{ij}^k &= \frac{\tau_{ij}}{\sum_{j \in N_j} \tau_{ij}} \\
\text{After use, deposit of pheromone on } (i,j) \text{ updated as:} \\
\quad \quad \quad \quad \quad \quad \tau_{ij}(t) &= \tau_{ij}(t) + \Delta \tau
\end{align*}
Memory & Diversification

- Individual ants remember their most recent trail, for backtracing
- Ants are not adaptive, but modify problem presentation
- Communicate indirectly with others
- Pheromone trails evaporate over time using an exponential
  \[ \tau \leftarrow (1 - \rho)\tau, \rho \in (0,1] \]
  at each iteration
- Avoids quick convergence to sub-optimal path

Natural Parallelism

- Steps of the algorithm can be run by parallel processors
- Little synchronization needed

ACO Outline:
1. Initialize incumbent
2. Until termination rule reached:
   1. Create newAnt(s)
   2. Evaporate pheromone
3. Return incumbent
ACO Outline

newAnt()

1. Initialize at start
2. while (current state NE target state)
   a) Determine neighborhood
   b) Compute move probabilities based on pheromone
   c) Move across selected arc and update arc pheromone
3. Compute value of solution found
4. Compare with incumbent, possibly updating
5. Return to start along previous path, updating pheromone based on solution value
6. Die

Scatter Search
Scatter-Search Overview

1. Generate a set of diverse solution vectors via heuristics
2. Repeat until termination:
   1. Pick best to be reference solutions
   2. Create new points by linear combinations of current reference solutions, inside & outside the convex regions formed
   3. Apply generalized rounding procedures to create integer values as needed

Diverse Solution Generator
Elite Reference Set Selection

Linear-Combination Solutions
New Directions in IP

Interior Solution

Exterior Solutions
New Directions in IP

New Elite Reference Set

Linear Combinations
Linear Combinations

Interior/Exterior Combos
Constraint Programming

An alternative to optimization

• Often, users are simply interested in identifying feasible solutions to difficult problems
• Optimality is not as important
• The *Constraint Satisfaction Problem* (CSP):
  – Set of variables, any type
  – Set of constraints
  – Requires: find 1/all feasible solution
Map Coloring Problem

definitions

enum Country
{Belgium, Denmark, France, Germany, Netherlands, Luxembourg};
enum Colors {blue, red, yellow, gray};
var Colors color[Country];
solve {
    color[France] <> color[Belgium];
    color[France] <> color[Luxembourg];
    color[France] <> color[Germany];
    color[Luxembourg] <> color[Germany];
    color[Luxembourg] <> color[Belgium];
    color[Belgium] <> color[Netherlands];
    color[Belgium] <> color[Germany];
    color[Germany] <> color[Netherlands];
    color[Germany] <> color[Denmark];
};

Stable Marriage

definitions

enum Women ...
enum Men ...
int rankWomen[Women, Man] = ...
int rankMen[Man, Women] = ...
var Women wife[Man];
var Men husband[Women];
solve {
    forall(m in Men)
        husband[wife[m]] = m;
    forall(w in Women)
        wife[husband[w]] = w;
    forall(m in Men & o in Women)
        rankMen[m, o] < rankMen[m, wife[m]] =>
            rankWomen[o, husband[o]] < rankWomen[o, m];
    forall(w in Women & o in Men)
        rankWomen[w, o] < rankWomen[w, husband[w]] =>
            rankMen[o, wife[o]] < rankMen[o, w];
};
Marriage Model

```plaintext
enum Women ...;
enum Men ...

int rankWomen[Women, Men] = ...;
int rankMen[Men, Women] = ...;

var Women wife[Men];
var Men   husband[Women];

solve {
    forall(m in Men)
        husband[wife[m]] = m;

    forall(w in Women)
        wife[husband[w]] = w;

    forall(m in Men & o in Women)
        rankMen[m, o] < rankMen[m, wife[m]] =>
            rankWomen[o, husband[o]] < rankWomen[o, m];

    forall(w in Women & o in Men)
        rankWomen[w, o] < rankWomen[w, husband[w]] =>
            rankMen[o, wife[o]] < rankMen[o, w];
}
```

Marriage Data

```plaintext
Men = {Richard, James, John, Hugh, Greg};
Women = {Helen, Tracy, Linda, Sally, Wanda};

rankWomen =
    #[
        Helen: #[Richard:1, James:2, John:4, Hugh:3, Greg:5 ]#,
        Tracy: #[Richard:3, James:5, John:1, Hugh:2, Greg:4 ]#,
        Linda: #[Richard:5, James:4, John:2, Hugh:1, Greg:3 ]#,
        Sally: #[Richard:1, James:3, John:5, Hugh:4, Greg:2 ]#,
        Wanda: #[Richard:4, James:2, John:3, Hugh:5, Greg:1 ]#
    ]#

rankMen =
    #[
        Richard: #[Helen:5, Tracy:1, Linda:2, Sally:4, Wanda:3 ]#,
        James : #[Helen:4, Tracy:1, Linda:3, Sally:2, Wanda:5 ]#,
        John : #[Helen:5, Tracy:3, Linda:2, Sally:4, Wanda:1 ]#,
        Hugh : #[Helen:1, Tracy:5, Linda:4, Sally:3, Wanda:2 ]#,
        Greg : #[Helen:4, Tracy:3, Linda:2, Sally:1, Wanda:5 ]#
    ]#;
```
Optimized CSP

• When an objective is given, a constraint programming solver will:
  – Expect a program for exploring all possible solutions
  – Examine all solutions in a self-determined order

• Output:
  – List of all optimal solutions
range Boolean 0..1;
int fixed = ...;
int nbStores = ...;
enum Warehouses ...
range Stores 0..nbStores-1;
int capacity[Warehouses] = ...;
int supplyCost[Stores,Warehouses] = ...;
int maxCost = max(s in Stores, w in Warehouses) supplyCost[s,w];

var Boolean open[Warehouses];
var Boolean supply[Stores,Warehouses];
var Warehouses supplier[Stores];
var int cost[Stores] in 0..maxCost;

minimize with linear relaxation
  sum(w in Warehouses) fixed * open[w] +
  sum(w in Warehouses, s in Stores) supplyCost[s,w] * supply[s,w]
subject to {
  forall(s in Stores)
    sum(w in Warehouses) supply[s,w] = 1;
  forall(w in Warehouses, s in Stores)
    supply[s,w] <= open[w];
  forall(w in Warehouses)
    sum(s in Stores) supply[s,w] <= capacity[w];

  forall(s in Stores)
    cost[s] = supplyCost[s,supplier[s]];
  forall(s in Stores)
    open[supplier[s]] = 1;
  forall(w in Warehouses)
    sum(s in Stores) (supplier[s] = w) <= capacity[w];

  forall(s in Stores)
    supply[s,supplier[s]] = 1;
  forall(s in Stores)
    cost[s] = sum(w in Warehouses) supplyCost[s,w] * supply[s,w]
};

search {
  forall(s in Stores ordered by decreasing regretmin(cost[s]))
    tryall(w in Warehouses ordered by increasing supplyCost[s,w])
      supplier[s] = w;
};