CS 312: Algorithm Analysis

Lecture #33: Branch and Bound, Job Assignment

Slides by Eric Ringger, with contributions from Mike Jones, Eric Mercer, and Sean Warnick

Objectives

- Understand the difference between backtracking and branch and bound
- Understand bounding functions
- Develop a branch and bound algorithm for the Job Assignment Problem

State-space Search

- **Backtracking**
  - **Purpose:** Existence, Enumeration
  - **Answer:** Yes/No, Count, Set
  - **Idea:** Avoid searching the entire state-space
  - **Main tool:** Feasibility function

- **Branch and Bound**
  - **Purpose:** Existence, Enumeration, Optimization
  - **Answer:** Yes/No, Count, Set (Good, better, best)
  - **Idea:** Both avoid searching the entire state-space
  - **Main tool:** Feasibility function, Bounding function

Bounding Function

Given some state $s$ in the search space, compute a bound $B(s)$ on the cost/goodness of all solutions that descend from that state.

**BSSF = Best Solution So Far**

- cost of actual solution is somewhere in here
- Better worse
- Cost of BSSF bound $B(s)$

"From this state $s$, I can do no better than $B(s)$."
Pruning Scenario #1

<table>
<thead>
<tr>
<th></th>
<th>Cost of BSSF</th>
<th>bound B(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>better</td>
<td></td>
<td></td>
</tr>
<tr>
<td>worse</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Would you explore this state s with bound B(s)?

BSSF = “Best Solution So Far”

Pruning Scenario #2

<table>
<thead>
<tr>
<th></th>
<th>bound B(s’)</th>
<th>Cost of BSSF</th>
</tr>
</thead>
<tbody>
<tr>
<td>better</td>
<td></td>
<td></td>
</tr>
<tr>
<td>worse</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Would you explore this state s’ with bound B(s’)?

BSSF = “Best Solution So Far”

Job Assignment Problem

- Given n tasks and n agents.
- Each agent has a cost to complete each task.
- Assign each agent one task; each task one agent;
- A 1:1 mapping
- Minimize cost

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>11</td>
<td>12</td>
<td>18</td>
<td>40</td>
</tr>
<tr>
<td>B</td>
<td>14</td>
<td>15</td>
<td>13</td>
<td>22</td>
</tr>
<tr>
<td>C</td>
<td>11</td>
<td>17</td>
<td>19</td>
<td>23</td>
</tr>
<tr>
<td>D</td>
<td>17</td>
<td>14</td>
<td>20</td>
<td>28</td>
</tr>
</tbody>
</table>

Could solve with LP, But it makes a good example for B&B.

Minimization

- The cost of the BSSF
- Is an Upper Bound on the optimal solution
- B(): Bounding function for evaluating any state s
- Is a Lower Bound on potential solutions reachable from s
- Usually involves solving a relaxed form of the problem
- s0: Initial state
- LB = B(s0)
- Is a Lower Bound on all potential solutions reachable from the initial state
- Tight bounds:
- Why would you want the upper and lower bounds to be tight?

Example

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>11</td>
<td>12</td>
<td>18</td>
<td>40</td>
</tr>
<tr>
<td>B</td>
<td>14</td>
<td>15</td>
<td>13</td>
<td>22</td>
</tr>
<tr>
<td>C</td>
<td>11</td>
<td>17</td>
<td>19</td>
<td>23</td>
</tr>
<tr>
<td>D</td>
<td>17</td>
<td>14</td>
<td>20</td>
<td>28</td>
</tr>
</tbody>
</table>

How to represent a state?
How about an initial state?
Example

First, generate a solution (not necessarily optimal) and call that your best solution so far (BSSF). How?

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>11</td>
<td>12</td>
<td>18</td>
<td>40</td>
</tr>
<tr>
<td>B</td>
<td>14</td>
<td>15</td>
<td>13</td>
<td>22</td>
</tr>
<tr>
<td>C</td>
<td>11</td>
<td>17</td>
<td>19</td>
<td>23</td>
</tr>
<tr>
<td>D</td>
<td>17</td>
<td>14</td>
<td>20</td>
<td>28</td>
</tr>
</tbody>
</table>

BSSF

First, generate a solution (not necessarily optimal) and call that your best solution so far (BSSF). How?

BSSF: 73

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>11</td>
<td>12</td>
<td>18</td>
<td>40</td>
</tr>
<tr>
<td>B</td>
<td>14</td>
<td>15</td>
<td>13</td>
<td>22</td>
</tr>
<tr>
<td>C</td>
<td>11</td>
<td>17</td>
<td>19</td>
<td>23</td>
</tr>
<tr>
<td>D</td>
<td>17</td>
<td>14</td>
<td>20</td>
<td>28</td>
</tr>
</tbody>
</table>

Hint

First, generate a solution (not necessarily optimal) and call that your best solution so far (BSSF). How?

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>16</td>
<td>12</td>
<td>18</td>
<td>40</td>
</tr>
<tr>
<td>B</td>
<td>14</td>
<td>15</td>
<td>13</td>
<td>22</td>
</tr>
<tr>
<td>C</td>
<td>11</td>
<td>17</td>
<td>19</td>
<td>23</td>
</tr>
<tr>
<td>D</td>
<td>17</td>
<td>14</td>
<td>20</td>
<td>28</td>
</tr>
</tbody>
</table>

Hint

First, generate a solution (not necessarily optimal) and call that your best solution so far (BSSF). How?

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>16</td>
<td>12</td>
<td>18</td>
<td>40</td>
</tr>
<tr>
<td>B</td>
<td>14</td>
<td>15</td>
<td>13</td>
<td>22</td>
</tr>
<tr>
<td>C</td>
<td>11</td>
<td>17</td>
<td>19</td>
<td>23</td>
</tr>
<tr>
<td>D</td>
<td>17</td>
<td>14</td>
<td>20</td>
<td>28</td>
</tr>
</tbody>
</table>

Lower Bound

Bounding function:
- Easy to evaluate
- True bound

Next, what should we use as our bound function?

Add the smallest entry in each column.

Other alternatives?

Lower Bound

Bounding function:
- Easy to evaluate
- True bound

Next, apply the bound function to the initial state to compute the lower bound on all solutions.

Lower Bound

Bounding function:
- Easy to evaluate
- True bound

Next, apply the bound function to the initial state to compute the lower bound on all solutions.
Next, apply the bound function to the initial state to compute the lower bound on all solutions.

- **LB = 58**

### Initial questions:
- What if the LB == BSSF?
- What if the assignment underlying the LB is a solution?
- What if LB > BSSF?

### Example

Then apply the bounding function:
- add the smallest values in each column.
This is a lower bound on the cost of any solution with job 1 assigned to A.

It's not a solution!
### Example

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>14</td>
<td>11</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>17</td>
<td>15</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>14</td>
<td>20</td>
<td>28</td>
<td></td>
</tr>
</tbody>
</table>

A:1 (60)
A:2 (58)
A:3 (65)
A:4 (78)

BSSF: 73

### Example

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>14</td>
<td>13</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>14</td>
<td>13</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>15</td>
<td>19</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>14</td>
<td>20</td>
<td>28</td>
<td></td>
</tr>
</tbody>
</table>

A:1 (60)
A:2 (58)
A:3 (65)
A:4 (78)

BSSF: 73

What can we say about the last option?

### Example

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>11</td>
<td>12</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>14</td>
<td>15</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>15</td>
<td>17</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>17</td>
<td>19</td>
<td>23</td>
<td></td>
</tr>
</tbody>
</table>

A:1 (60)
A:2 (58)
A:3 (65)
A:4 (78)

BSSF: 73

How to proceed from here? Many options:
- Breadth-first
- Depth-first
- Most promising first

### Example

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>11</td>
<td>12</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>14</td>
<td>15</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>15</td>
<td>17</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>17</td>
<td>19</td>
<td>23</td>
<td></td>
</tr>
</tbody>
</table>

A:1 (60)
A:2 (58)
A:3 (65)
A:4 (78)

BSSF: 73

How to proceed from here? Many options:
- Breadth-first
- Depth-first
- Most promising first – we’ll try this one today
- Other possibilities

### Example

A:1 (60)
A:2 (58)
A:3 (65)
A:4 (78)

### Example

A:2, B:1 (68)
A:2 (58)
A:3 (65)
A:4 (78)
When you reach a solution, update the BSSF if better.

Prune!

That's the basic idea. Many details have been left out. Let's add a few.
Details

- What should each state contain?
  - Just as for backtracking state-space search, we need enough information in each state to generate its children in the state space.
  - The value of the bound on this state.
- How to store the set of states visited but remaining to be explored (i.e., “frontier of the search”)?
  - A set, sometimes called the “agenda” or “open list.”
  - DFS: use a stack.
  - BFS: use a queue.
  - Most Promising First: use a priority queue.
  - Other possibilities (e.g., hybrids).

Recall: Iterative DFS

```c
function DFS(v)
P  empty-stack
visited(v)  true
P.push(v)
while !P.empty() do
  while there exists w adjacent to P.top() (in ascending order) such that !visited(w) do
    visited(w)  true
    P.push(w) // w is the new P.top()
P.pop()
```

Recall: Breadth First Search

```c
function BFS(v)
Q  empty-queue
visited(v)  true
Q.enqueue(v)
while !Q.empty() do
  u  Q.first()
  Q.dequeue()
  for each w adjacent to u (in ascending order) do
    if !visited(w) then
      visited(w)  true
      Q.enqueue(w)
```

B&B using a Priority Queue: Work in Progress

```c
function BandB-draft(v)
Q  empty-priority-queue
visited(v)  true
v.b  bound(v)  // LB
Q.enqueue(v, v.b)
while !Q.empty() do
  u  Q.first()
  Q.dequeue()
  children = generate_children_ascending(u)
  for each w in children do
    if !visited(w) then
      visited(w)  true
      w.b  bound(w)
      Q.enqueue(w, w.b)
```

Use Bound as Priority

```c
function BandB-draft(v)
Q  empty-priority-queue
visited(v)  true
v.b  bound(v)  // LB
Q.enqueue(v, v.b)
while !Q.empty() do
  u  Q.first()
  Q.dequeue()
  for each w adjacent to u (in ascending order) do
    if !visited(w) then
      visited(w)  true
      w.b  bound(w)
      Q.enqueue(w, w.b)
```

Adapt for On-the-fly State-Space Search

```c
function BandB-draft(v)
Q  empty-priority-queue
visited(v)  true
v.b  bound(v)
Q.enqueue(v, v.b)
while !Q.empty() do
  u  Q.first()
  Q.dequeue()
  for each w adjacent to u (in ascending order) do
    if !visited(w) then
      visited(w)  true
      w.b  bound(w)
      Q.enqueue(w, w.b)
```

What's missing?

Remember: a priority queue is not the only possible representation of the agenda.
Critical Elements of a B&B Algorithm

- **BSSF**
- **Bounding function**
- **Representation of the frontier on an agenda**
- **Select state from the agenda**
  - Generate children
  - By copying parent and making one more decision
  - Calculate the bound
  - Handle children that are solutions
  - Place promising children on the agenda
- **Prune the losers**
- **Repeat until**
  - Agenda is empty

An option for Project #7
Completeness and Optimality

Important properties of search:

- **Completeness:**
  - Is your search guaranteed to find a solution when one exists?

- **Optimality:**
  - Does your search find the optimal (i.e., minimum or maximum cost) solution?
  - Does your search only prune sub-optimal states?

Assignment

- **Homework #24**
  - B&B
  - due Thursday

- **Required:** Read B&B Notes & Project #7 instructions!
  - Then you’re ready to solve Project #7