Slides for a Course on the Analysis and Design of Algorithms

Chapter 6: State-Space Search Methods

Stephen J. Hegner
Department of Computing Science
Umeå University
Sweden
hegner@cs.umu.se
http://www.cs.umu.se/~hegner

6. State-Space Search Methods

6.1 Basic Concepts

6.1.1 The basic setting

Setting: The setting is that of problems whose solutions may be expressed in the form

$$(x_1, x_2, \ldots, x_n) \in S_1 \times S_2 \times \ldots \times S_n$$

in which the S_i's are fixed, finite sets.

Examples:

discrete-knapsack problem:

- $S_i = \{0, 1\}$, $1 \le i \le n$.
- 0 $\Rightarrow$ exclude object; 1 $\Rightarrow$ include object.

sum-of-subsets problem:

Given: weights: $\{w_1, w_2, \ldots, w_n\}$
goal: M

Find: all $A \subseteq \{w_1, w_2, \ldots, w_n\}$ with $\sum A = M$.

- $S_i = \{0, 1\}$, $1 \le i \le n$.
- Solutions $(x_1, x_2, \ldots, x_n) \in \{0, 1\}^n$ are such that

$$\sum_{i=1}^{n} x_i \cdot w_i = M$$

n-queens problem:

- Place n queens on a $n \times n$ "chessboard" in such a fashion that no queen can take another.
- Put $S_i = \{1, \ldots, n\} \times \{1, \ldots, n\}$.
- S_i represents the row and column of the i^{th} queen.

6.1.2 State-space size and reduction

- Consider the eight-queens problem as a specific example.
- The solution space for the representation given in 1.1.1 has the following size:

$$\prod_{i=1}^{8} \mathsf{Card}(S_i) = (8^2)^8 = 2^{48}$$

- One way to reduce the size of the state space is to redefine it.
- A simple by sizeable reduction is realized by building into the representation the fact that each queen must lie in a distinct row.
- Thus, define:

$$\mathsf{Value}(S_i) = \text{column of the queen in row } i$$

- In this case,

$$S_i = \{1, 2, \ldots, 8\} \quad \text{for each i}$$

  and the size of the new solution space is:

$$\prod_{i=1}^{8} \mathsf{Card}(S_i) = 8^8 = 2^{24} = 16777216.$$

- An much improved reduction is possible if S_{i+1} is allowed to depend upon S_i.
- For example, observe that in any solution, distinct queens must lie not only in distinct rows, but in distinct columns as well.
- Thus,

$$S_i = \begin{cases} \{1,2,\ldots,8\} & \text{if } i = 1 \\ S_{i-1} \setminus (\text{position of the queen in } S_{i-1}) & \text{otherwise} \end{cases}$$

- The size of the solution space is now:

$$\prod_{i=1}^{8} \mathsf{Card}(S_i) = 8! = 40320$$

- It is necessary to be somewhat careful in specifying state-space restriction.
- An extreme but useless criterion might be to require that $(x_1, x_2, \ldots, x_8)$ already be a solution.
- Clearly, the definition of a state-space element must be simple.
- For the most part, in these notes, situations in which the S_i's do not depend upon one another will be considered.
- From a graphical perspective, the solution space may be viewed as a tree, with each S_i a level in that tree.
- The possible final solutions are represented by the leaves.
- A solution is obtained by determining a value (choice) for each level.

- Further questions:
  - When is a partial solution doomed to failure?
  - In other words, when may a subtree be pruned away?
  - When are two partial solutions essentially the same?
  - Which vertices should be expanded first?

6.1.3 A general formulation for vertex classification and search strategies

- In that which follows, the solution graph will be generated, one vertex at a time, top down.
- The following terminology is relevant in the context of a search tree.

(a) A *dead vertex* is is one for which either:

  (i) all children have been generated; or

  (ii) further expansion is not necessary (because the entire subtree of which it is a root may be pruned).

(b) A *live vertex* is one which has been generated, but which is not yet dead.

(c) The *E-vertex* is the parent of the vertex which is currently being generated.

(d) A *bounding function* is used to kill live vertices via some evaluation function which establishes that the vertex cannot lead to an optimal solution, rather than by exhaustively expanding all of its children.

- The following two *search strategies* are those which will be considered.

Backtracking:

- Vertices are kept in a stack.
- The top of the stack is always the current E-vertex.
- As children of the current E-vertex are generated, they are pushed onto the top of the stack.
- In particular, as soon as a new child *w* of the current E-vertex is generated, *w* becomes the new E-vertex.
- Vertices which are popped from the stack are dead.
- A bounding function is used to kill live vertices (*i.e.*, remove them from the stack) without generating all of their children.

Branch-and-bound:

- Vertices are kept in a *vertex pool*, which may be a stack, queue, priority queue, or the like. Variation is possible.
- As children of the current E-vertex are generated, they are inserted into the vertex pool.
- However, once a vertex becomes the E-vertex, it remains so until it dies.
- The "next" element in the vertex pool becomes the new E-vertex when the current E-vertex dies.
- Vertices which are removed from the vertex pool are dead.
- A bounding function is used to kill live vertices (*i.e.*, remove them from the stack) without generating all of their children.

6.2 Backtracking

- In this subsection, the principles of backtracking will be illustrated via application to the discrete knapsack problem.
- Recall the notational conventions of this problem from 4.3.1:
  - A knapsack with weight capacity M.
  - n objects $\{\mathsf{obj}_1, \mathsf{obj}_2, \ldots, \mathsf{obj}_n\}$, each with a weight w_i and a value v_i.
  - M, the w_i's, and the v_i's are all taken to be positive real numbers.

6.2.1 Bounding functions

- Effective use of backtracking requires a good bounding function.
- In the context of the discrete knapsack problem, a *bounding function* provides a simple-to-compute upper bound on the amount of additional profit which may be obtained by taking a leaf of the subtree of the current vertex as a solution.
- An extremely simple bounding function is obtained by adding the profits of all objects which are yet to be considered.
  - If $(x_1, x_2, \ldots, x_i) \in \{0,1\}^i$ have already been chosen, then

$$\text{bound} = \sum_{j=i+1}^{n} \mathsf{v}_j$$

- A better bounding function is obtained as follows.

  (i) Generate the solution of an associated continuous knapsack problem; specifically

  (ii) If $(x_1, x_2, \ldots, x_i) \in \{0,1\}^i$ has already been chosen as a partial solution, let A be the continuous knapsack problem corresponding to (see 4.3.2) $\mathsf{Knap}(i+1, n, M - \sum_{j=1}^{i} x_j \cdot \mathsf{v}_j)$; *i.e.*, with

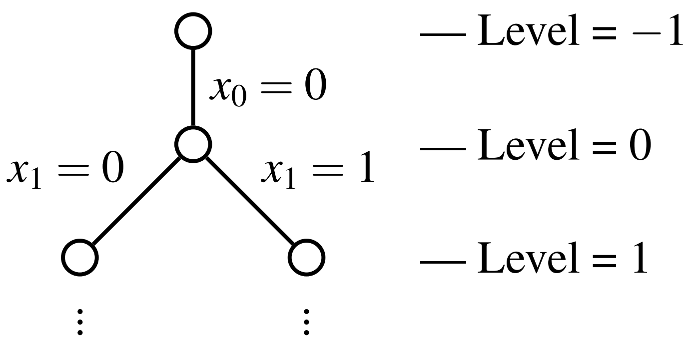
$$\begin{aligned} \text{capacity} &= M - \sum_{j=1}^{i} x_j \cdot \mathsf{v}_j \\ \text{objects} &= \{\mathsf{obj}_{i+1}, \mathsf{obj}_{i+2}, \ldots, \mathsf{obj}_n\} \end{aligned}$$

  - Solve this problem using a greedy-style method and take the profit of that solution to be the bound.
  - Note that the profit of the solution to the continuous knapsack problem will always yield a profit at least as large as that for its discrete counterpart, so this computation does provide an upper bound.

6.2.2 Pseudocode description of the algorithm

```
/*  The major data structures:  */
profit :     array[0..n] of real;
weight :     array[0..n] of real;
best_sol :   array[0..n] of {0,1};
tent_sol :   array[0..n] of {0,1};
/*  0 is a dummy object.  */

/*  The top-level program:  */
⟨ best_profit ← 0;
  path_profit ← 0;
  path_weight ← 0;
  level ← −1;
  try(0);
⟩
```

- Note that -1 labels a dummy top level with only one choice ($x_0 = 0$).



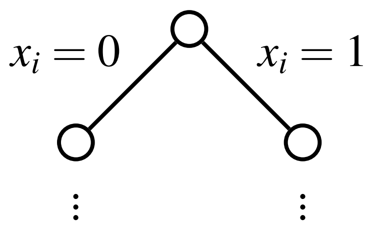
```
procedure try(choice : {0,1})
  ⟨ level ← level + 1;
    tent_sol[level] ← choice;
    if (choice = 1)
      then ⟨ path_weight ← path_weight + weight[level];
              path_profit ← path_profit + profit[level];
           ⟩
    if path_weight ≤ M
      then solve();
    if (choice = 1)
      then ⟨ path_weight ← path_weight − weight[level];
              path_profit ← path_profit − profit[level];
           ⟩
    level ← level − 1;
  ⟩

procedure solve();
  ⟨ if level = n
      then process_leaf
       else if bound() + path_profit > best_profit
              then ⟨ try(0); try(1) ⟩
  ⟩

procedure process_leaf();
  ⟨ if path_profit > best_profit
      then ⟨ best_sol ← tent_sol; best_profit ← path_profit; ⟩
  ⟩
```

Improvement: If *bound* uses a solution to the continuous knapsack problem which also happens to solve the extant discrete knapsack problem, a solution has been found, so the computation may be stopped.

6.2.3 An alternate approach – dynamic state-space solution

- The idea is as follows:

  1. Generate a solution $(x_1, x_2, \ldots, x_n)$ to the associated continuous knapsack problem.
     - Note that $x_i \in \{0,1\}$ must hold for all except possibly on value of i.
  2. If all $x_i \in \{0,1\}$, a solution to the discrete knapsack problem has been found.
  3. If $0 < x_i < 1$, use x_i as the branch value for the next level.



  4. Solve the associated problem for each subtree.

- Experiments have shown (perhaps surprisingly) that this approach is inferior to that which uses a static representation.

6.2.4 Solution of two examples

- The example problem introduced in 3.1.3, and solved in 4.3.3 using dynamic programming, is solved here using backtracking.
- For completeness, the data of the example are restated.
- Let $M = 8$; $n = 4$.
- In this case, for variation, the solution will be found for two distinct orderings of the objects, as shown in the tables below.

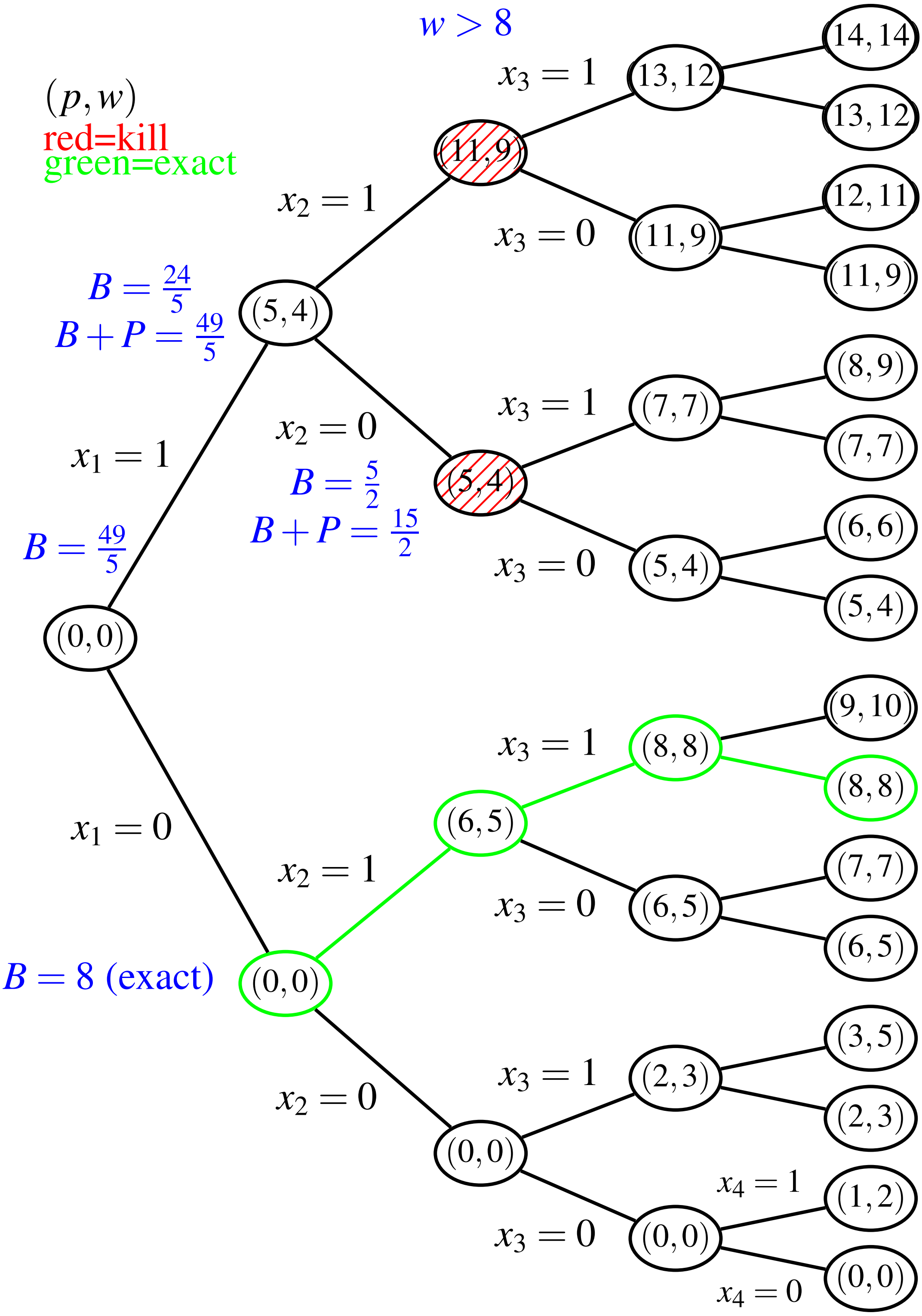
(a):

i	1	2	3	4
v_i	5	6	2	1
w_i	4	5	3	2

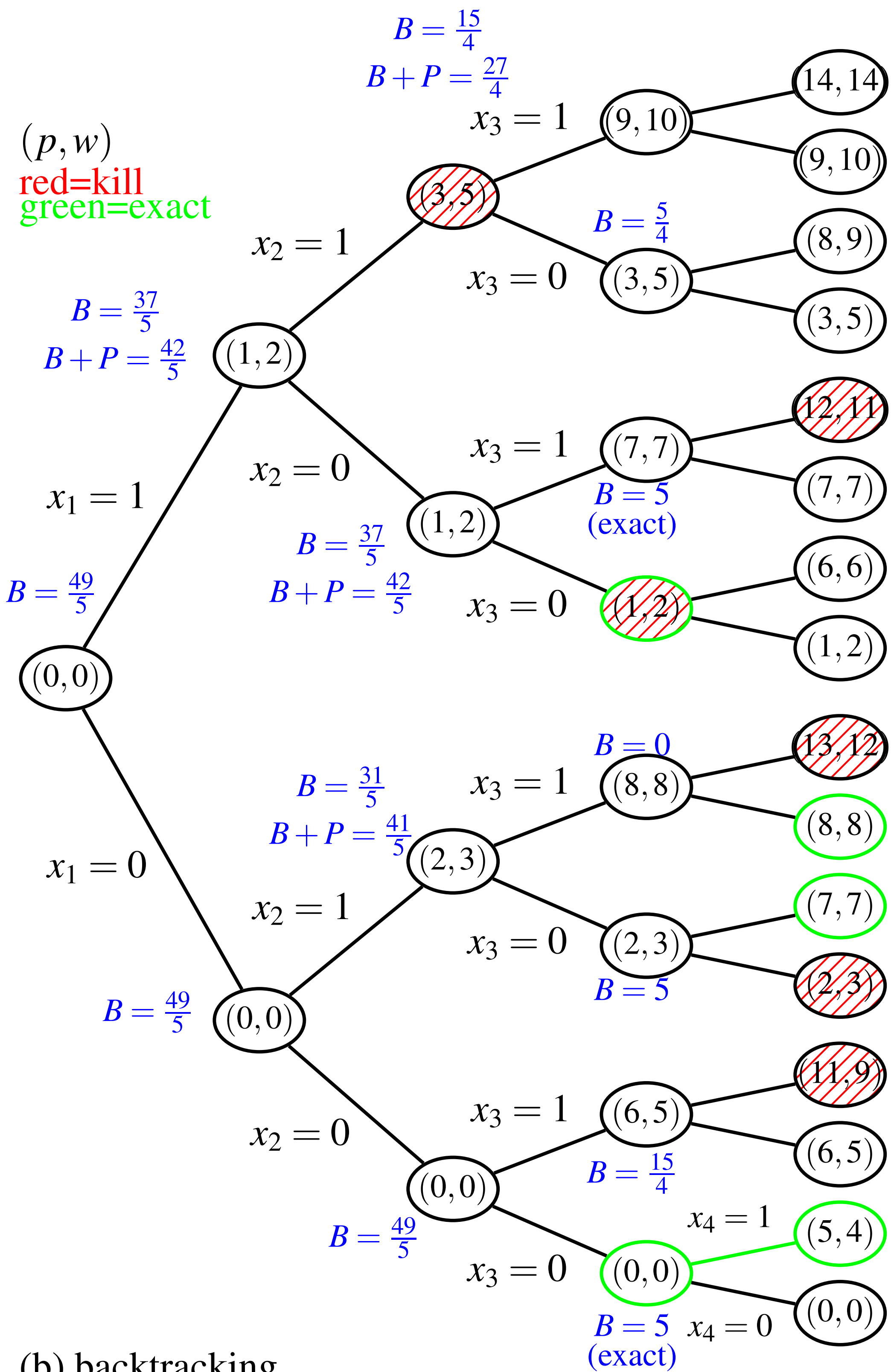
(b):

i	1	2	3	4
v_i	1	2	6	5
w_i	2	3	5	4

- The solution to the continuous knapsack problem on the remaining subproblem will be used as the bounding function.
- The heuristic employed is that if the solution to the continuous knapsack problem is also a solution to the extant discrete problem, the subtree has been solved optimally.
- Note that the algorithm works regardless of the order of the objects, but that in any case the p/w ordering on the remaining objects must be used to obtain the continuous knapsack problem required for the bounding function.



(a) backtracking



(b) backtracking

Selection order in continuous-knapsack approximation is by p/w.

6.3 Branch and Bound

6.3.1 Overview of branch and bound

- Recall the general strategy: generate all children of the current E-vertex before selecting a new E-vertex.
- Strategies for selecting a new E-vertex:

  LIFO order: depth first, using a stack.

  FIFO order: breadth first, using a queue.

  Intelligent order: use a priority queue.

- In each case, a bounding function is also used to kill vertices.

6.3.2 Example – the 8-puzzle

- Eight tiles move about nine squares.
- The goal configuration is shown below:

1	2	3
4	5	6
7	8	

- Tiles are moved from an initial configuration to reach the goal configuration.

1	2	3
4		5
7	8	6

$\rightsquigarrow$

1	2	3
4	5	
7	8	6

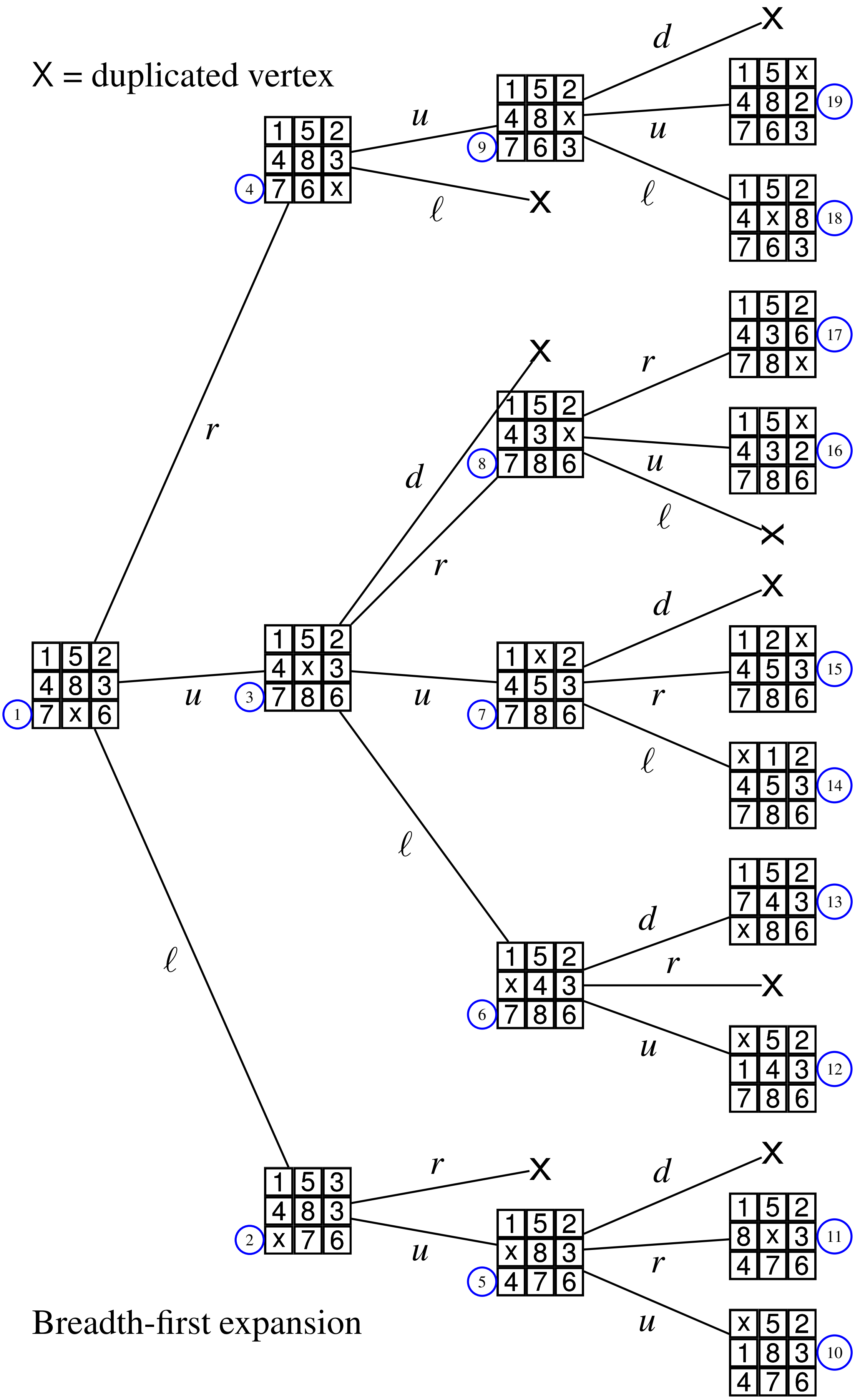
$\rightsquigarrow$

1	2	3
4	5	6
7	8	

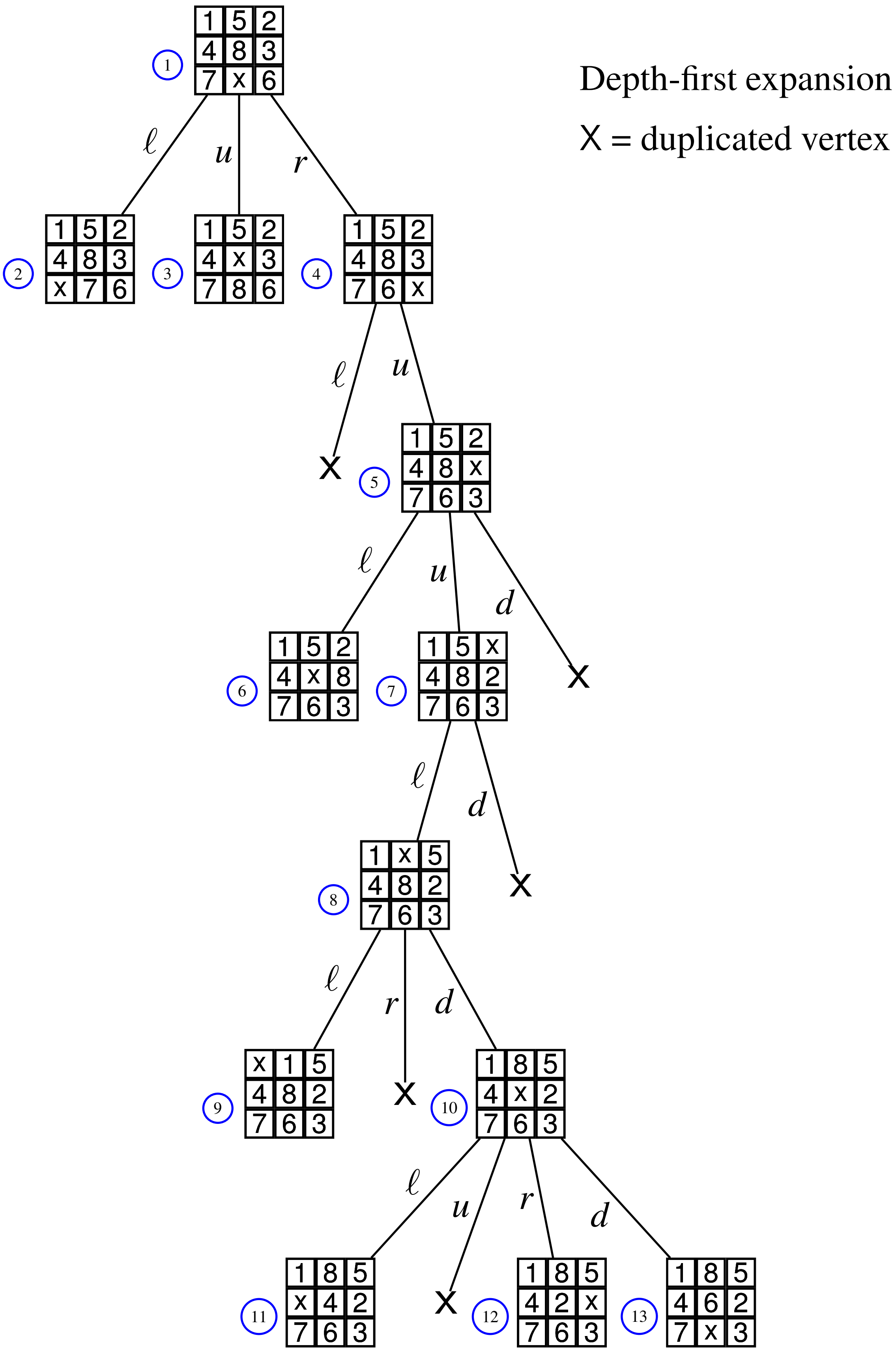
- Notation for directions to "move" the open slot:

$$\begin{array}{lll} \ell = \text{left} & & u = \text{up} \\ r = \text{right} & & d = \text{down} \end{array}$$

- No bounding function is used here.
- Examples of LIFO and FIFO order are shown on the following two slides.



Breadth-first expansion



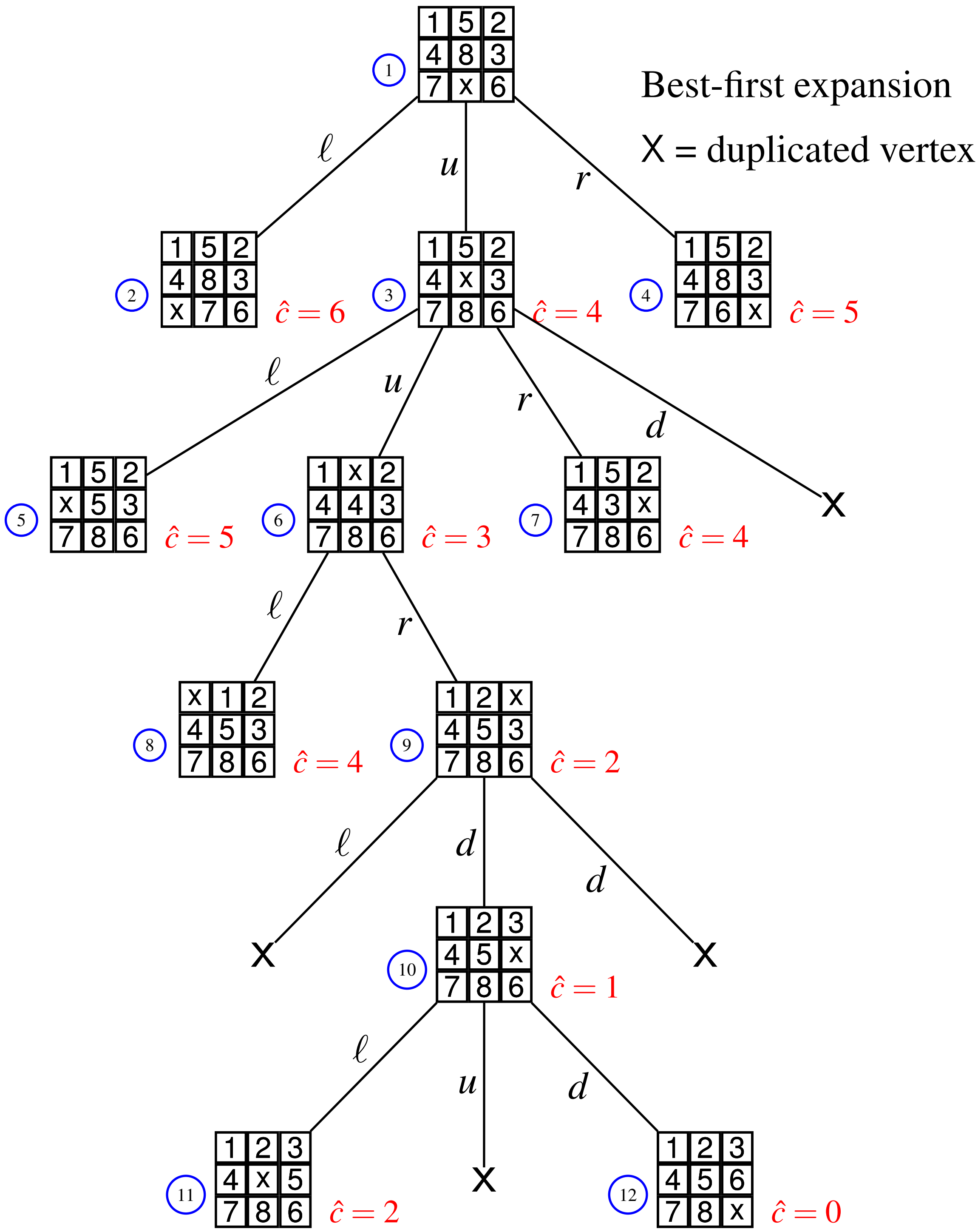
Depth-first expansion
X = duplicated vertex

6.3.3 Best-first search with branch and bound

- Associated with a best-first strategy is a *cost function* c.
- $c(x)$ is the cost of finding a solution from vertex x.
- A reasonable measure of cost might be the number of additional vertices which must be generated in order to obtain a solution.
- The problem is that $c(x)$ is very difficult to compute, in general, without generating the solution first.
- Therefore, an approximation $\hat{c}$ is used.
- In the 8-puzzle, an appropriate $\hat{c}$ might be the following:

$$\hat{c}(x) = \text{number of tiles which are out of place}$$

- In this measure, the empty slot is not considered to be a tile.
- On the next slide, the best-first expansion of the extant example for the 8-puzzle is shown.



Best-first expansion
X = duplicated vertex
1
2
3
4
5
6
7
8
9
10
11
12
ℓ
u
r
d
$\hat{c} = 6$
$\hat{c} = 4$
$\hat{c} = 5$
$\hat{c} = 5$
$\hat{c} = 3$
$\hat{c} = 4$
$\hat{c} = 4$
$\hat{c} = 2$
$\hat{c} = 1$
$\hat{c} = 2$
$\hat{c} = 0$

6.3.4 Desirable properties for $\hat{c}$

- The two key properties are the following:
  - (a) $\hat{c}$ should approximate c in a "nice" fashion.
  - (b) $\hat{c}$ should be easy to compute.
- An oft-used form for $\hat{c}$ is the following:

$$\hat{c}(x) = \hat{g}(x) + k(x)$$

- in which:
  - $\hat{g}$ is an estimate of the cost to reach a solution vertex from x.
  - $k(x)$ is a weighted function of the cost to reach vertex x from the root.
- In the 8-puzzle example:
  - $\hat{g}(x)$ is the number of tiles which are out of place.
  - $k(x) = 0$.

Argument for $k(x) = 0$: A cost which has already been incurred should not enter into the evaluation.

Argument for $k(x) > 0$:

- $k = 0$ adds a bias in favor of deep searches.
- If $|\hat{g}(x) - c(x)|$ is large, the wrong path may be expanded to a very deep level.
- $k(x)$ adds a breadth-first component.

- A possible choice for $k(x)$ for the 8-puzzle is the length of the path from the root to x.

6.3.5 Properties of $\hat{c}$ for general search problems

- For a general search problem such as 8-puzzle, in which there is no distinction between feasible solutions and optimal ones, further properties on $\hat{c}$ are not generally necessary for correctness.
- Note, however, that a mechanism for avoiding visiting the same vertex repeatedly is necessary to avoid loops in the search process.

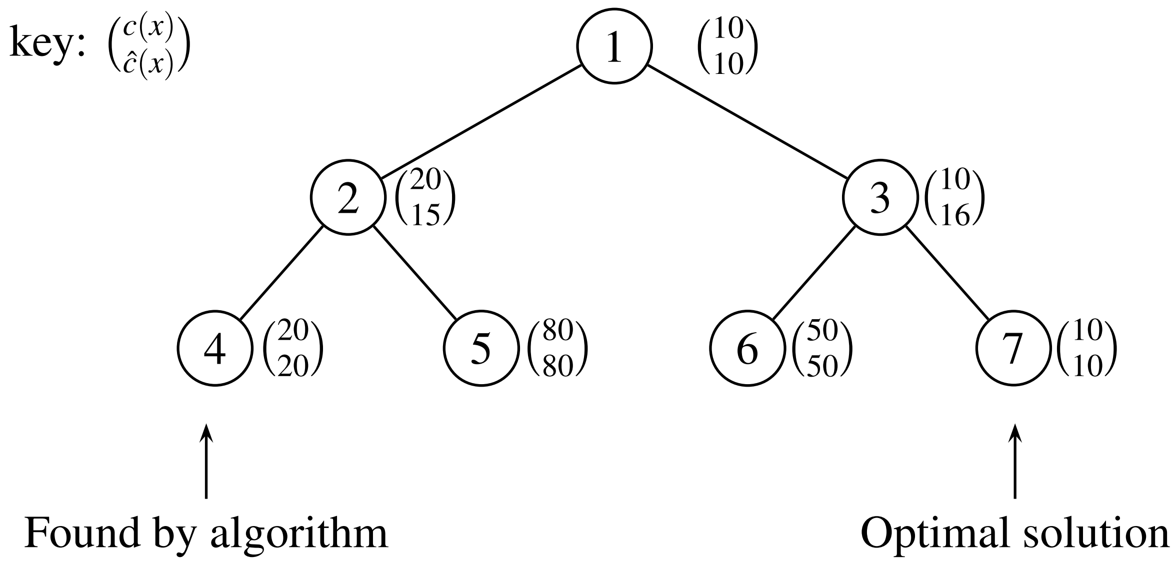
6.3.6 Important properties of $\hat{c}$ for optimization problems

- For $\hat{c}$ to function correctly in a best-first search process, it must satisfy certain formal properties if an optimal solution is to be found.
- Consider in particular optimization problems such as discrete knapsack and travelling salesman.
- It is important to know whether a leaf vertex which has been reached in the search process is an optimal solution.

- To see the difficulties, consider a minimization problem with:

  $c(x)$ = value of the best leaf beneath vertex x.

  $\hat{c}(x)$ is as shown in the graph below.



- The problem:
  - $c(3) < c(2) \Rightarrow$ the optimal solution is below vertex 3.
  - $\hat{c}(3) > \hat{c}(2) \Rightarrow$ the algorithm looks below vertex 2.

(a) Call an approximate cost function $\hat{c}$ *ideal* if the following condition holds for all pairs of vertices (x, y):

$$\hat{c}(x) < \hat{c}(y) \Leftrightarrow c(x) < c(y)$$

6.3.7 Theorem *Let c (resp. $\hat{c}$) be the actual (resp. approximate) cost function for a minimization problem to be solved by branch-and-bound search. The first leaf vertex to be reached is the optimal solution iff $\hat{c}$ is ideal.* $\Box$

- The conditions of 1.3.7 are very difficult to establish in practice.
- A weaker but far more useful result is the following.

6.3.8 Definition Call an approximate cost function $\hat{c}$ *admissible* if the following two conditions are satisfied.

(a) $\hat{c}(x) \leq c(x)$ for all vertices x.

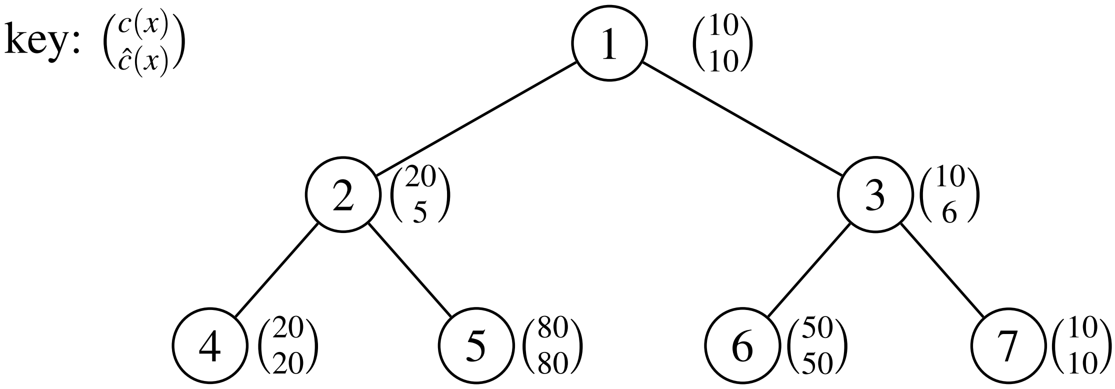
(b) $\hat{c}(x) = c(x)$ for all answer vertices (*i.e.*, all leaf vertices which represent feasible solutions).

6.3.9 Theorem (informal statement) *If the approximate cost function $\hat{c}$ is admissible, then under branch-and-bound solution, the first answer vertex to become an E-vertex is an optimal solution.*

PROOF: This result will be stated more rigorously and proven in 1.3.11 below. □

6.3.10 Example

- Consider the following search tree, which is a modification of that of 1.3.6, altered so that $\hat{c}(x) \leq c(x)$ for all nodes x.



- The evolution of the priority queue of vertices is as follows:

$$\begin{array}{lllllll} 1(10) & \leadsto & 2(5) & \leadsto & 3(6) & \leadsto & 7(10) \\ & & 3(6) & & 4(20) & & 4(20) \\ & & & & 5(80) & & 6(50) \\ & & & & & & 5(80) \end{array}$$

- Vertices 4 and 5 are the first answer vertices to be placed in the queue.
- However, Vertex 7 is the first which becomes the E-vertex, so it is an optimal solution and the search may be halted.

6.3.11 Theorem (formal statement) *Let $T = (V, E, g)$ be a finite rooted tree, and let*

$$c : V \to \mathbb{R}$$

be an evaluation function on the vertices of T which is fixed on the leaves of T and which satisfies

$$c(x) = \min(\{c(y) \mid y \textit{ is a leaf descendant of } x\})$$

for all non-leaf vertices. Let

$$\hat{c} : V \to \mathbb{R}$$

be an admissible approximate cost function with respect to c. Then, if a least-cost branch-and-bound expansion of the tree is performed with respect to $\hat{c}$, the first E-vertex which is also a leaf is a minimum-cost leaf.

PROOF: Let x be the current E-vertex, and suppose further that x is a leaf and that no previous E-vertex has been a leaf. Let y be any other leaf vertex, and let w be the youngest (*i.e.*, furthest from the root) ancestor of y which has been generated. Then $\hat{c}(x) \leq \hat{c}(w)$, else w would have been an E-vertex before x, and have generated descendants. Also, $c(w) \leq c(y)$, since $c(w)$ is the minimum value over all of its descendants. Hence $c(x) = \hat{c}(x) \leq \hat{c}(w) \leq c(w) \leq c(y)$. □

6.3.12 Remark

- Branch-and-bound search with an admissible $\hat{c}$ is called *A*-search* in the artificial intelligence literature.

6.3.13 Solution of the discrete knapsack problem

- The discrete knapsack examples of 1.2.4 will now be solved using branch and bound.
- A leaf vertex x is identified with the solution vector $(x_1, x_2, \ldots, x_n)$ which defines the path from the root to x.
- Since this is a maximization problem, the inequalities must be reversed; *i.e.*, $\hat{c}(x) \geq c(x)$.
- The following definition of $c(x)$ is used:

$$c(x) = \begin{cases} \sum_{i=1}^{n} \mathsf{v}_i \cdot x_i & \text{for a feasible answer (leaf) vertex } x \\ -\infty & \text{for an illegal leaf vertex (too much weight)} \\ \max\left(\left\{\begin{array}{c} c(\mathsf{LeftChild}(x)) \\ c(\mathsf{RightChild}(x)) \end{array}\right\}\right) & \text{for a non-leaf} \end{cases}$$

- The following approximation function is used for a vertex x at level j in the tree (with the root at level 0):

$$\hat{c}(x) = \sum_{i=1}^{j} \mathsf{v}_i \cdot x_i + \mathsf{Profit}(\mathsf{CKnap}(j+1, n, M - \sum_{i=1}^{j} \mathsf{w}_i \cdot x_i))$$

  in which $\mathsf{Profit}(\mathsf{CKnap}(p, q, W))$, with $p \leq q$, denotes the profit obtained in the solution of the continuous knapsack problem with objects $\{\mathsf{obj}_k \mid p \leq k \leq q\}$ and capacity W.

- The vertex-killing function at level j which is used is the following:

$$u(x) = \sum_{i=1}^{j} \mathsf{v}_i \cdot x_i + \mathsf{Profit}(\mathsf{Greedy}(p,q,W))$$

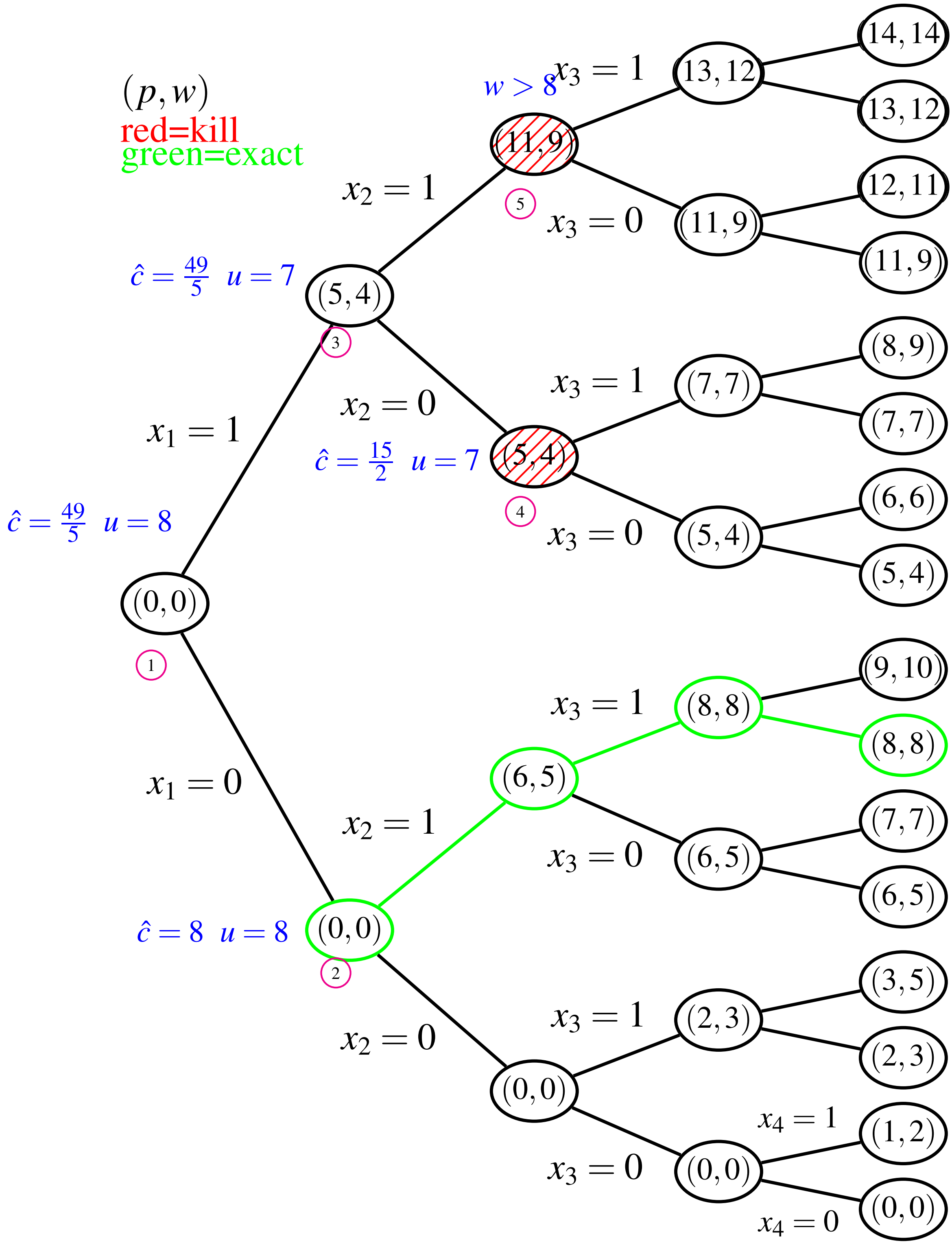
in which $\mathsf{Profit}(\mathsf{Greedy}(p,q,W))$, with $p \leq q$, is the value obtained by applying a greedy-style procedure, with the objects $\{\mathsf{obj}_k \mid p \leq k \leq q\}$, ordered by profit, for a knapsack problem with capacity W.

- The following global value is maintained:

$$U = \max(\{u(x) \mid x \text{ has been generated}\})$$

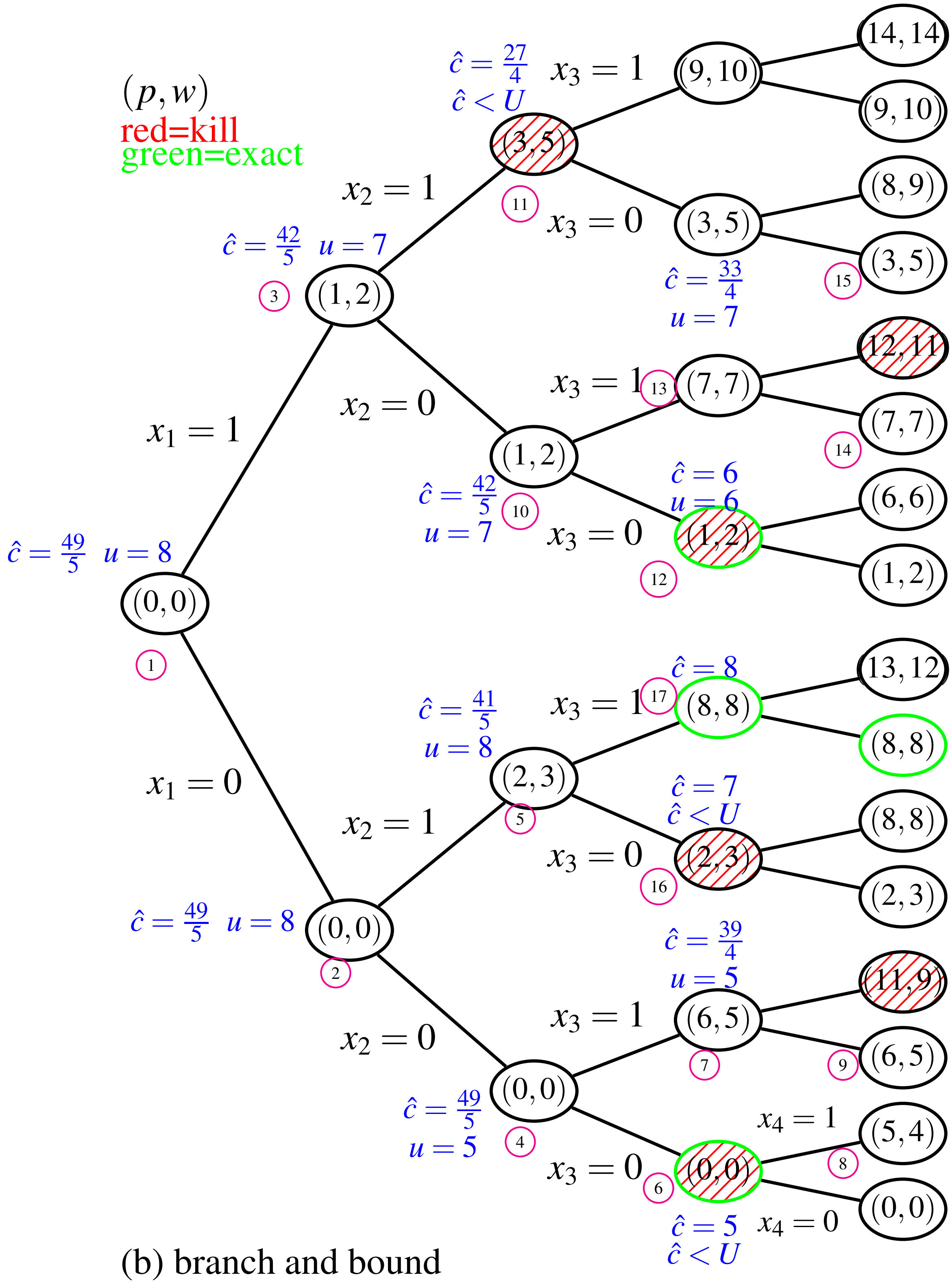
- The vertex x is killed whenever $\hat{c}(x) < U$.

- Evaluation is also halted if the computation of $\hat{c}(x)$ results in an exact solution of the continuous knapsack problem, as in 1.2.4.



(a) branch and bound

Vertex 2 is regarded as a leaf, because of the exact solution.



(b) branch and bound

Selection order in both knapsack approximations is by p/w.

- The priority queue history is as follows:

order (a): $1\left(\frac{49}{5}\right) \rightsquigarrow \begin{matrix} 2(8)\text{X} \\ 3\left(\frac{49}{5}\right) \end{matrix} \rightsquigarrow 3\left(\frac{49}{5}\right) \rightsquigarrow$ done

order (b): $1\left(\frac{49}{5}\right) \rightsquigarrow \begin{matrix} 2\left(\frac{49}{5}\right) \\ 3\left(\frac{42}{5}\right) \end{matrix} \rightsquigarrow \begin{matrix} 4\left(\frac{49}{5}\right) \\ 3\left(\frac{42}{5}\right) \\ 5\left(\frac{41}{5}\right) \end{matrix} \rightsquigarrow \begin{matrix} 7\left(\frac{39}{4}\right) \\ 3\left(\frac{42}{5}\right) \\ 5\left(\frac{41}{5}\right) \end{matrix} \rightsquigarrow$

$\begin{matrix} 3\left(\frac{42}{5}\right) \\ 5\left(\frac{41}{5}\right) \\ 8(6)L \end{matrix} \rightsquigarrow \begin{matrix} 10\left(\frac{42}{5}\right) \\ 5\left(\frac{41}{5}\right) \\ 8(6)L \end{matrix} \rightsquigarrow \begin{matrix} 13\left(\frac{33}{4}\right) \\ 5\left(\frac{41}{5}\right) \\ 8(6)L \end{matrix} \rightsquigarrow \begin{matrix} 5\left(\frac{41}{5}\right) \\ 14(7)L \\ 8(6)L \end{matrix} \rightsquigarrow \begin{matrix} 17(8)X \\ 14(7)L \\ 8(6)L \end{matrix} \rightsquigarrow$ done

key: Entries are of the form $v(\hat{c}(v))$[type] with:

v	=	vertex number
L	$\Rightarrow$	leaf vertex
X	$\Rightarrow$	exact solution; behaves as a leaf vertex

6.4 The Travelling-Salesman Problem and Branch-and-Bound

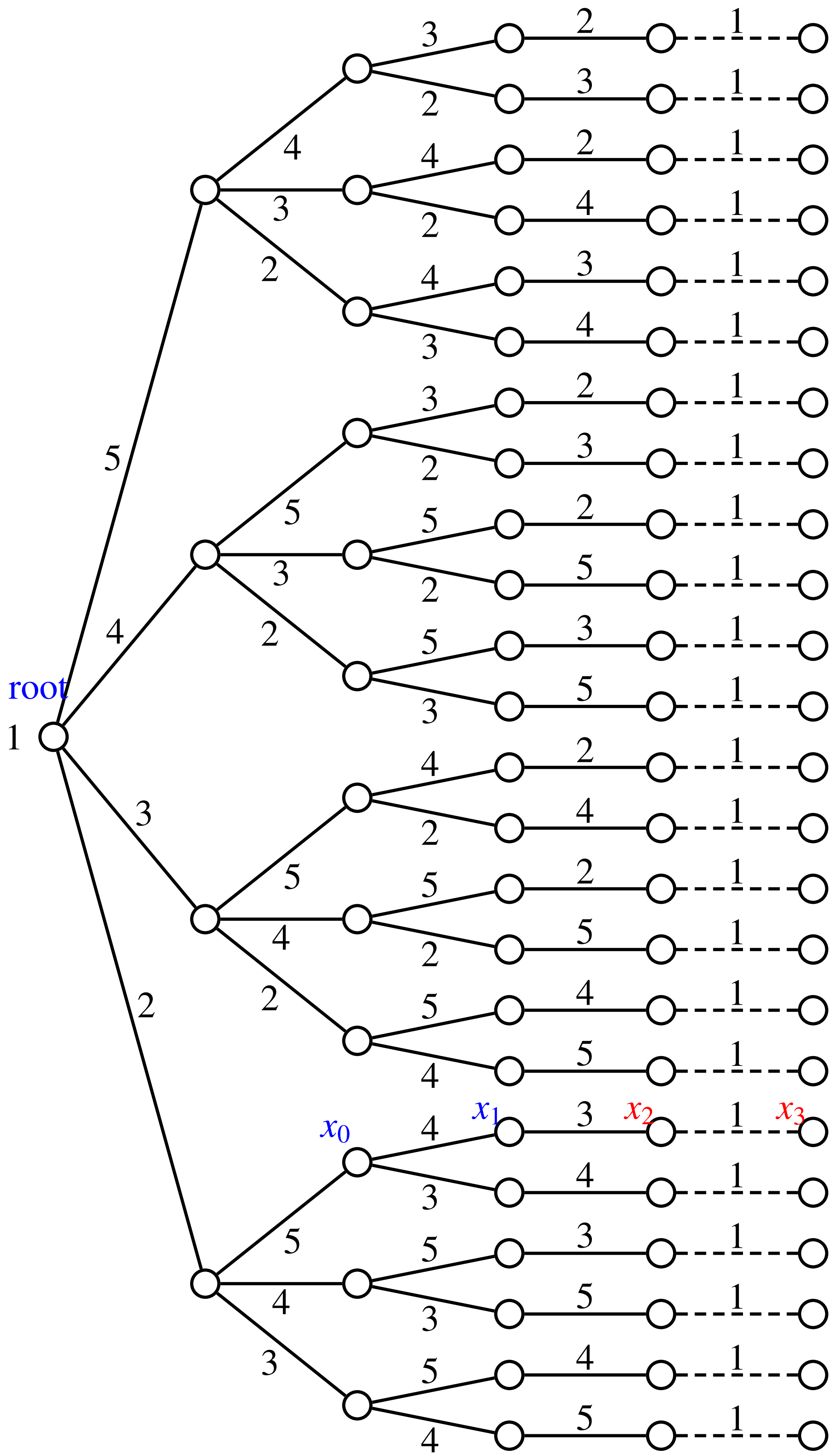
6.4.1 Formulation of the problem

- The (directed) graph G is represented as a *cost matrix*.

Example:

$$M = \begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix}$$

- Vertices are numbered $\{1, 2, \ldots, n\}$, with $n = 5$ in this example.
- M_{ij} is the cost of the edge $i \rightsquigarrow j$.
- $M_{ij} = \infty$ means that there is no edge $i \rightsquigarrow j$.
- The associated state-space tree starts at vertex 1, and reflects the sequence of choices.
- The tree for $n = 5$ is shown on the next slide.


root
1
x_0
x_1
x_2
x_3

6.4.2 Conventions for the state-space tree

- Every vertex is labelled with the sequence beginning with 1, and followed by the sequence of labels of the associated edges.
- For a vertex x of the state-space tree, this label is denoted by $\mathsf{PathOf}(x)$.

  $\mathsf{PathOf}(\text{root}) = \langle 1 \rangle$ $\mathsf{PathOf}(x_2) = \langle 1,2,5,4,3 \rangle$
  $\mathsf{PathOf}(x_0) = \langle 1,2,5 \rangle$ $\mathsf{PathOf}(x_3) = \langle 1,2,5,4,3,1 \rangle$
  $\mathsf{PathOf}(x_1) = \langle 1,2,5,4 \rangle$

- Call a vertex x of the state-space tree a *decision vertex* if it has at least two ancestors.
- Call a vertex x of the state-space tree a *near leaf* if $\mathsf{PathOf}(x)$ includes all vertices except one.
- In the tree on the previous page, x_1 is a near leaf, while x_2 and x_3 are not.
- Once a near leaf is reached, all decisions regarding the tour have been made. No further decision can be made.
- Thus, the near leaves will be treated as leaves in the search process.
- Call a vertex x of the state-space tree *nonredundant* if it is either a decision vertex or a near leaf.
- For a near leaf x, define $\mathsf{Tour}(x)$ to be $\mathsf{PathOf}(x) \cdot \langle x', 1 \rangle$ with x' the sole vertex not in $\mathsf{PathOf}(x)$.
- For example, $\mathsf{Tour}(x_1) = \mathsf{PathOf}(x_3)$ in the graph on the previous page.

- For an actual cost function on the nonreduncant vertices of the state-space tree, the following is used:

$$c(x) = \begin{cases} \mathsf{CostOf}(\mathsf{Tour}(x)) & \text{if } x \text{ is a near leaf} \\ \min(\{c(y) \mid y \in \mathsf{Children}(x)\} & \text{otherwise} \end{cases}$$

- Note that $\mathsf{CostOf}(\mathsf{rootvertex})$ is the cost of an optimal tour.
- A simple choice for $\hat{c}$ is the cost along the path from the root to x. If x is a near leaf, the cost of travelling to the final new vertex and then back to the root (along a single edge) must be added on.
- There are much better choices for $\hat{c}$, which are now developed.

6.4.3 Row minimization Let M be the cost matrix for a travelling-salesman problem of size n, and let $i \in \{1,2,\ldots,n\}$.

(a) $$\mathsf{RowMin_i}(M) = \begin{cases} \min(\{M_{ij} \mid 1 \leq j \leq n\}) & \text{if some } M_{ij} < \infty \\ 0 & \text{if } M_{ij} = \infty \text{ for all } j,\ 1 \leq j \leq n \end{cases}$$

(b) $\mathsf{Reduction}(M, \mathsf{row}, i)$ is the matrix obtained by subtracting $\mathsf{RowMin_i}(M)$ from each entry in row i.

Note: In the context of this computation, $\infty - a = \infty$ for any finite number a.

6.4.4 Theorem – row reduction *Let T be the travelling-salesman problem defined by matrix M, and let* $\mathsf{Reduction}(T,\mathsf{row},i)$ *be the travelling-salesman problem defined by the matrix* $\mathsf{Reduction}(M,\mathsf{row},i)$. *Then*

$$\begin{aligned}&\mathsf{CostOf}(\mathsf{MinTour}(T)) = \\ &\quad \mathsf{CostOf}(\mathsf{MinTour}(\mathsf{Reduction}(T,\mathsf{row},i))) + \mathsf{RowMin_i}(M)\end{aligned}$$

PROOF: Each tour must include exactly one entry from row i, since each tour contains exactly edge which begins at vertex i. From this the result follows immediately. □

- Completely similar ideas apply to columns.

6.4.5 Column minimization Let M be the cost matrix for a travelling-salesman problem of size n, and let $i \in \{1,2,\ldots,n\}$.

(a) $\mathsf{ColMin_i}(M) = \begin{cases} \min(\{M_{ji} \mid 1 \leq j \leq n\}) & \text{if some } M_{ji} < \infty \\ 0 & \text{if } M_{ji} = \infty \text{ for all } j,\ 1 \leq j \leq n \end{cases}$

(b) $\mathsf{Reduction}(M,\mathsf{col},i)$ is the matrix obtained by subtracting $\mathsf{ColMin_i}(M)$ from each entry in column i.

6.4.6 Theorem – column reduction *Let T be the travelling-salesman problem defined by matrix M, and let* $\mathsf{Reduction}(T,\mathsf{col},i)$ *be the travelling-salesman problem defined by the matrix* $\mathsf{Reduction}(M,\mathsf{col},i)$. *Then*

$$\begin{aligned}&\mathsf{CostOf}(\mathsf{MinTour}(T)) = \\ &\quad \mathsf{CostOf}(\mathsf{MinTour}(\mathsf{Reduction}(T,\mathsf{col},i))) + \mathsf{RowMin_i}(M)\end{aligned}$$

□

6.4.7 Full reduction Let M be the cost matrix for a travelling salesman problem T consisting of n vertices.

(a) Call M *reduced* if each row and each column consists either entirely of ∞ entries, or else contains at least one zero entry.

(b) Define the *row-column reduction sequence* of M, denoted $\mathsf{RCRed}(M)$, recursively as follows:

$$\begin{array}{rcll} R_0(M) & = & M & \\ R_k(M) & = & \mathsf{Reduction}(R_k, \mathsf{row}, k-1) & 1 \le k \le n \\ R_k(M) & = & \mathsf{Reduction}(R_{k-1}, \mathsf{col}, k-n) & n+1 \le k \le 2n \end{array}$$

(c) Define the *row-column reduction* of M, denoted $\mathsf{RCRed}(M)$, to be $R_{2n}(M)$.

6.4.8 Example

- Let M be as in the example of 1.4.1:

$$M = \begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix}$$

- First do the rows:

$$R_n(M) = \begin{bmatrix} \infty & 10 & 20 & 0 & 1 \\ 13 & \infty & 14 & 2 & 0 \\ 1 & 3 & \infty & 0 & 2 \\ 16 & 3 & 15 & \infty & 0 \\ 12 & 0 & 3 & 12 & \infty \end{bmatrix} \begin{array}{r} 10 \\ 2 \\ 2 \\ 3 \\ \underline{4} \\ 21 \end{array}$$

- Then the columns:

$$R_{2n}(M) = \mathsf{RCRed}(M) = \begin{bmatrix} \infty & 10 & 17 & 0 & 1 \\ 12 & \infty & 11 & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \end{bmatrix}$$

$$\begin{matrix} 1 & 0 & 3 & 0 & 0 & = & 4 \end{matrix}$$

- A lower bound on the cost of a tour is thus 25.
- More generally:

6.4.9 Theorem *Let T be the travelling-salesman problem defined by matrix M, and let $\mathsf{RCRed}(T)$ be the travelling-salesman problem defined by the matrix $\mathsf{RCRed}(M)$. Then*

$$\mathsf{CostOf}(\mathsf{MinTour}(T)) = \mathsf{CostOf}(\mathsf{MinTour}(\mathsf{RCRed}(T))) + \sum_{i=1}^{n} (\mathsf{RowMin_i}(M) + \mathsf{ColMin_i}(R_n(M)))$$

with $R_n(M)$ as defined in 1.4.7. □

6.4.10 Dynamic reduction

- *Dynamic reduction* makes use of the fact that once a choice to follow an edge $i \rightsquigarrow j$ in the tour is made, the i^{th} row and the j^{th} column of the cost matrix M become irrelevant to the cost of extending the partial solution to an optimal tour.
- Since such reductions are applied only to nonredundant vertices of the state-space tree, the entry M_{j1} is also irrelevant, since including it would introduce a cycle into the partial solution.
- These entries may thus be forced to ∞ without affecting the computation of an optimal tour.
- The resulting matrix may be further reduced.
- The details are as follows.

(a) For any $n \times n$ cost matrix M, and any $i, j \in \{1, 2, \ldots, n\}$, define $\mathsf{PreRed}(M, i, j)$ to be the $n \times n$ matrix with

$$\mathsf{PreRed}(M,i,j)_{k,\ell} = \begin{cases} \infty & \text{if } i = k \text{ or } j = \ell \text{ or } (k,\ell) = (j,1) \\ M_{ij} & \text{otherwise} \end{cases}$$

(b) Define

$$\mathsf{DynRed}(M,i,j) = \mathsf{RCRed}(\mathsf{PreRed}(M,i,j))$$

6.4.11 Example

- In this example, $\mathsf{DynRed}(M', 1, 5)$ will be computed for the reduced matrix M' of 1.4.8, which is:

$$M' = \mathsf{RCRed}(M) = \begin{bmatrix} \infty & 10 & 17 & 0 & 1 \\ 12 & \infty & 11 & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \end{bmatrix}$$

- First, row 1, column 5, as well as the (5,1) entry, are set to ∞.

$$\mathsf{PreRed}(M', 1, 5) = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & 2 & \infty \\ 0 & 3 & \infty & 0 & \infty \\ 15 & 3 & 12 & \infty & \infty \\ \infty & 0 & 0 & 12 & \infty \end{bmatrix}$$

- Next, the full reduction of this new matrix is computed.

$$\mathsf{DynRed}(M', 1, 5) = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 10 & \infty & 9 & 0 & \infty \\ 0 & 3 & \infty & 0 & \infty \\ 12 & 0 & 9 & \infty & \infty \\ \infty & 0 & 0 & 12 & \infty \end{bmatrix} \begin{matrix} \\ 2 \\ \\ 3 \\ \\ \hline 5 \end{matrix}$$

- This yields a new lower bound on the least cost tour which begins with $1 \rightsquigarrow 5$.

25	+	1	+	5	=	31
↑		↑		↑		↑
old bound		old (1,5) entry		bound for new reduction		new lower bound

- In the *dynamic path reduction* technique, such a reduction is performed each time a decision to select a new edge for the tour is made.

6.4.12 Formal dynamic path reduction Let M be an $n \times n$ matrix which defines a travelling-salesman problem, and let $s = \langle x_1, x_2, \ldots, x_k \rangle$ be a sequence of distinct elements from $\{1, 2, .., n\}$ representing a nonredundant vertex of the state-space tree.

(a) For $1 \leq i \leq k$, define

$$\mathsf{PathRed}(M, s, x_i) = \begin{cases} \mathsf{RCRed}(M) & \text{if } i = 1 \\ \mathsf{DynRed}(\mathsf{PathRed}(M, s, x_{i-1}), x_{i-1}, x_i) & \text{otherwise} \end{cases}$$

(b) Define

$$k(s) = \sum_{i=1}^{n} (\mathsf{RowMin_i}(\mathsf{PathRed}(M, s, x_k)) + \mathsf{ColMin_i}(R_n(\mathsf{PathRed}(M, s, x_k))))$$

with $R_n(-)$ as defined in 1.4.7.

(c) Define

$$\hat{c}(s) = \begin{cases} k(s) & \text{if } x \text{ is not a near leaf} \\ k(s) + M_{x_k x'} + M_{x' 1} & \text{if } s \text{ is a near leaf and } s \cdot \langle x', 1 \rangle = \mathsf{Tour}(s). \end{cases}$$

- The idea is that, as a path is followed, dynamic reduction is executed for choices already made.

The following is easily verified.

6.4.13 Theorem *Let T be the travelling-salesman problem defined by matrix M, and let $\hat{c}$ be the cost function defined in 1.4.12. Then $\hat{c}$ satisfies the conditions of 1.3.11; i.e.,*

(a) *for all vertices x, $\hat{c}(x) \leq c(x)$;*

(b) *for all leaf vertices x, $\hat{c}(x) = c(x)$.* □

6.4.14 Vertex killing

- A non-leaf vertex may be killed if its reduced matrix contains all ∞ entries, for then no tour is possible.
- Qualitative vertex killing (equivalent to the use of U in the solution of the knapsack problem) is not used in this approach.
  - ➢ It may be added though, upon selection of a suitable means of obtaining such a bound.

6.4.15 Comments on complexity

- Each dynamic reduction may take time $\Theta(n^2)$, with n the number of vertices, although the constant will be small.
- The worst case complexity of this algorithm is $\Theta(n^2 \cdot n!)$, which is worse than the $\Theta(n^2 \cdot 2^n)$ of the dynamic programming approach (4.4.4).
  - ➢ Nevertheless, in practice, the performance often exceeds that of the dynamic-programming approach.