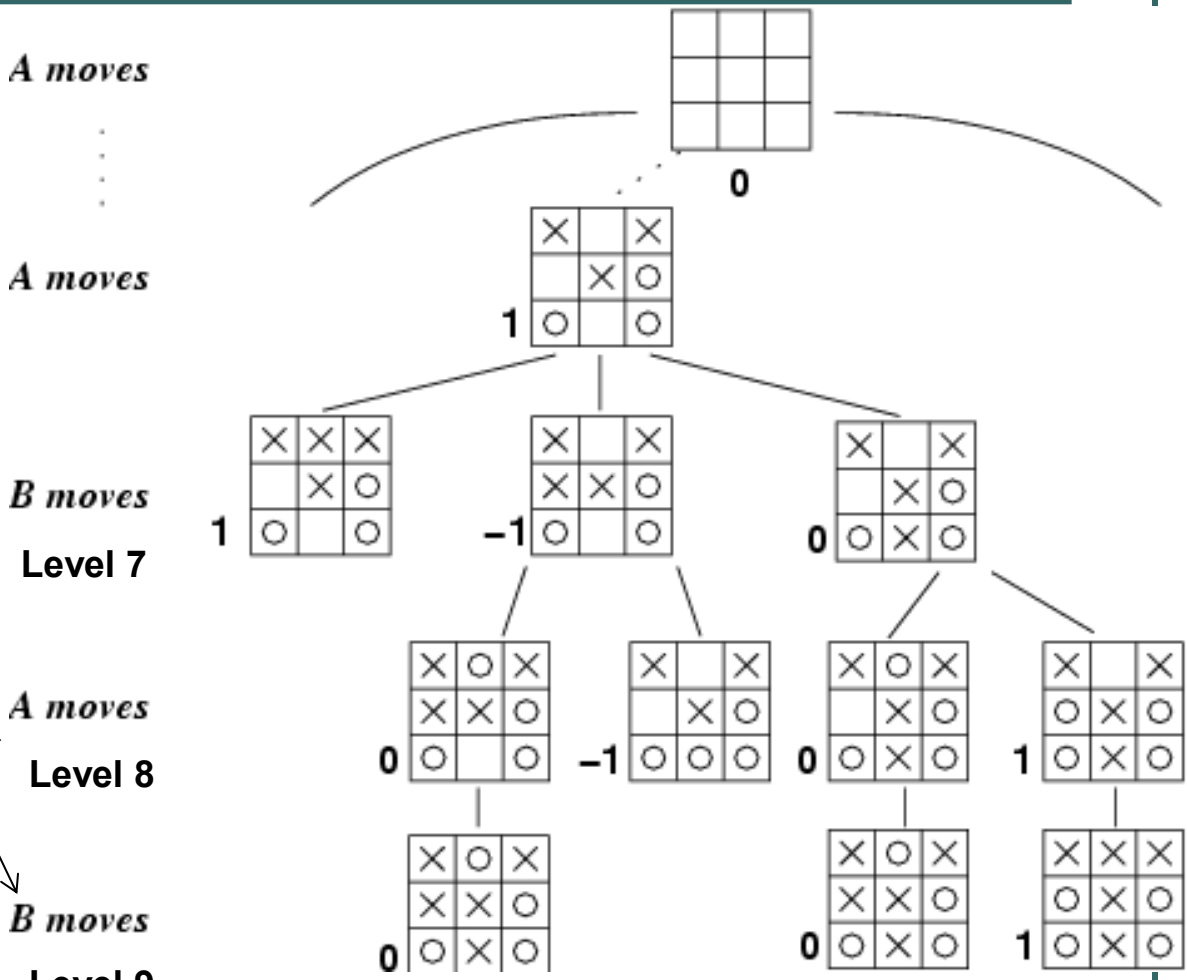
Algorithm Design Techniques (III)

Minimax. Alpha-Beta Pruning. Search Tree Strategies (backtracking revisited, branch and bound). Local Search.

Tic-Tac-Toe Game Tree (partly)

- Leaves that are wins for B – player using "'o"' to mark squares – get value -1, draws are 0, and wins for A, +1.
- We proceed up the tree. On level 8,there's one empty square and it's A's turn, the values for the unresolved board are the "'maximum"' of one child at level 9.
- On level 7, it is B's move and we take the value for an interior node the minimum of the values of its children.
- If the root has value +1, player A has a winning strategy, if 0 neither of them has, if -1, player B has one.



Minimax Algorithm

- Helps find the best move, by working backwards from the end of the game.
- At each step it assumes that
  - player A is trying to maximize the chances of A winning, while on the next turn
  - player B is trying to minimize the chances of A winning (i.e., to maximize B's own chances of winning).
- At terminal node (leaf):
  - The minimax value of the terminal node is given by the evaluation function (–1=loss, 0=draw, 1=win)
- At a non-terminal MAX node:
  - Calculate the minimax values of its successor nodes.
  - The minimax value of the MAX node is the maximum of the minimax values of its successor nodes
- At a non-terminal MIN node:
  - Calculate the minimax values of its successor nodes.
  - The minimax value of the MIN node is the minimum of the minimax values of its successor nodes
- For chess: cca 10^{40} different legal positions; 35^{100} nodes in average game tree

Alpha-beta pruning

- This algorithm is for heuristic search of best moves on game trees of games of two players, assuming that both players apply the best tactics, i.e. they do as good moves as possible on the basis of the available knowledge. It is a concretization of the branch and bound search.

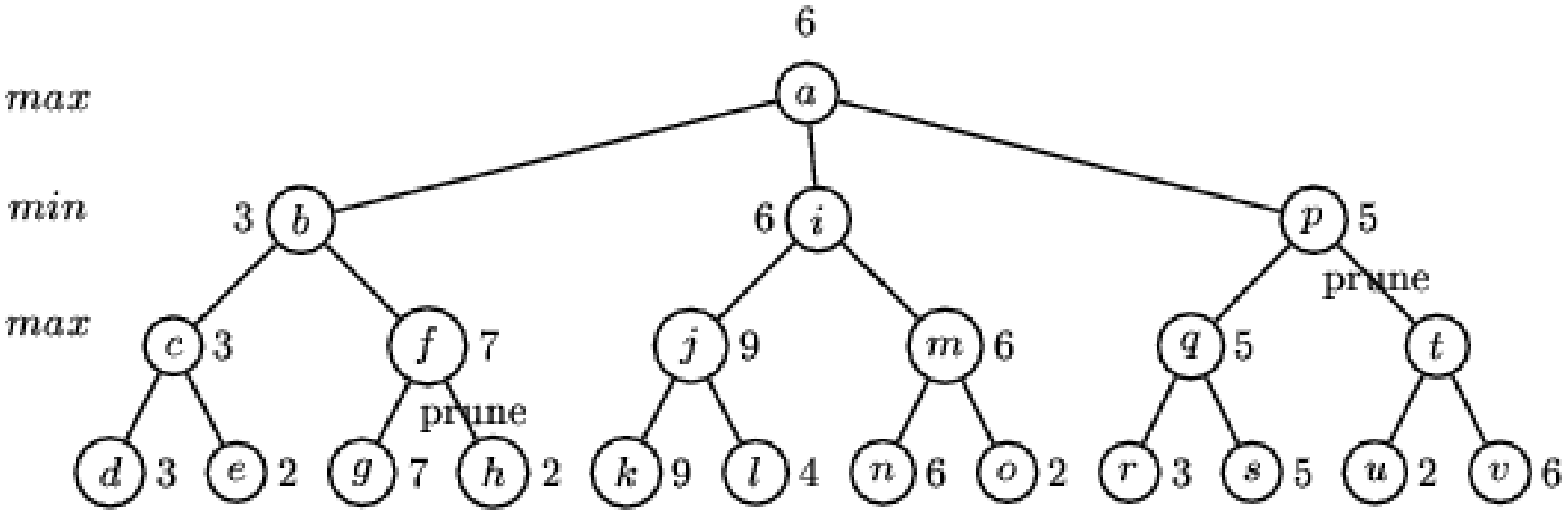
- This algorithm gives the same result as the **minimax** procedure for the games of two players. This algorithm lies in the ground of many game programs for **chess** and other games.

alpha-beta Algorithm

- ***alpha*** = best (largest) minimax value MAX is guaranteed to reach
- ***beta*** = best (smallest) minimax value MIN is guaranteed to reach
- Call: ***MaxValue***(root, $-\infty$, $+\infty$)

```
MaxValue(state, alpha, beta)
if node is a terminal node (or to be treated like one) then
    return the value of the evaluation function for that node
else
    for each successor state s of state
      do alpha ← max(alpha, MinValue(s, alpha, beta))
      if alpha ≥ beta then return beta
return alpha
MinValue(state, alpha, beta)
if node is a terminal node (or to be treated like one) then
    return the value of the evaluation function for that node
else
    for each successor state s of state
      do alpha ← min(alpha, MaxValue(s, alpha, beta))
      if alpha ≥ beta then return alpha
return alpha
```

Example of Alpha-Beta Pruning



- http://www.ndsu.nodak.edu/instruct/juell/vp/cs724s01/alpha/alpha.html
- http://www.emunix.emich.edu/~evett/AI/AlphaBeta_movie/sld001.htm

minimaxAB Algorithm

```
/* alpha is the best score for max along the path to state
   beta  is the best score for min along the path to state
   returns VALUE of best operator
   need additional code to return operator */

int minimaxAB(state, player, depth, alpha, beta)
   if (depth == limit or state is terminal)
        return the static evaluation of state
   if (player is min)
        until all successors, s, are examined or alpha > beta
                val=minimaxAB(s, max, depth+1, alpha, beta)
                if (val < beta) beta = val
        return beta
   if (player is max)
        until all successors, s, are examined or alpha > beta
                val=minimaxAB(s, min, depth+1, alpha, beta)
                if (val > alpha) alpha = val
        return alpha
```

Negamax – minimaxAB simplified

- variant formulation of minimax search that relies on the zero-sum property of a two-player game:

  max(a, b) = -min(-a, -b)

```
function negamax(node, depth, α, β, color)
        if node is a terminal node or depth = 0
                return color * the heuristic value of node
        else
                foreach child of node
                        α := max(α, -negamax(child, depth-1, -β, -α, -color))
                {the following if statement constitutes alpha-beta pruning}
                        if α≥β
                                break
        return α
```

http://en.wikipedia.org/wiki/File:Minmaxab.gif

Terminology for Search Tree Strategies

- If the solution graph is generated, one vertex at a time, top down, the following terminology is relevant in the context of a search tree.
  - A ***dead vertex*** is one for which either:
    1. all children have been generated; or
    2. further expansion is not necessary (because the entire subtree of which it is a root may be pruned).
  - A ***live vertex*** is one which has been generated, but which is not yet dead.
  - The ***E-vertex*** is the parent of the vertex which is currently being generated.
  - 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.

Search Strategies

- **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.
  - 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.

Search Strategies

- **Branch-and-bound**:
  - Vertices are kept in a ***vertex pool***, which may be a stack, queue, priority queue, or the like.
  - As children of the current ***E***-vertex are generated, they are inserted into the vertex pool.
  - 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 vertex pool) without generating all of their children.

Effective BackTracking

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

$$bound = \sum_{j=i+1}^{n} v_j$$

Effective Backtracking (2)

- Better bounding function :
  - Generate the solution of an associated fractionary knapsack problem; specifically
  - If $(x_1, x_2, \ldots, x_j) \in \{0, 1\}^j$ have already been chosen as a partial solution, let A be the fractionary knapsack problem with

$$capacity = W - \sum_{i=1}^{j} x_i w_i$$

$$items = \{item_{j+1}, item_{j+2}, \ldots$$

  - Solve this problem using a greedy-style method and take the profit of that solution to be the bound.
  - The profit of the solution to the fractionary 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.

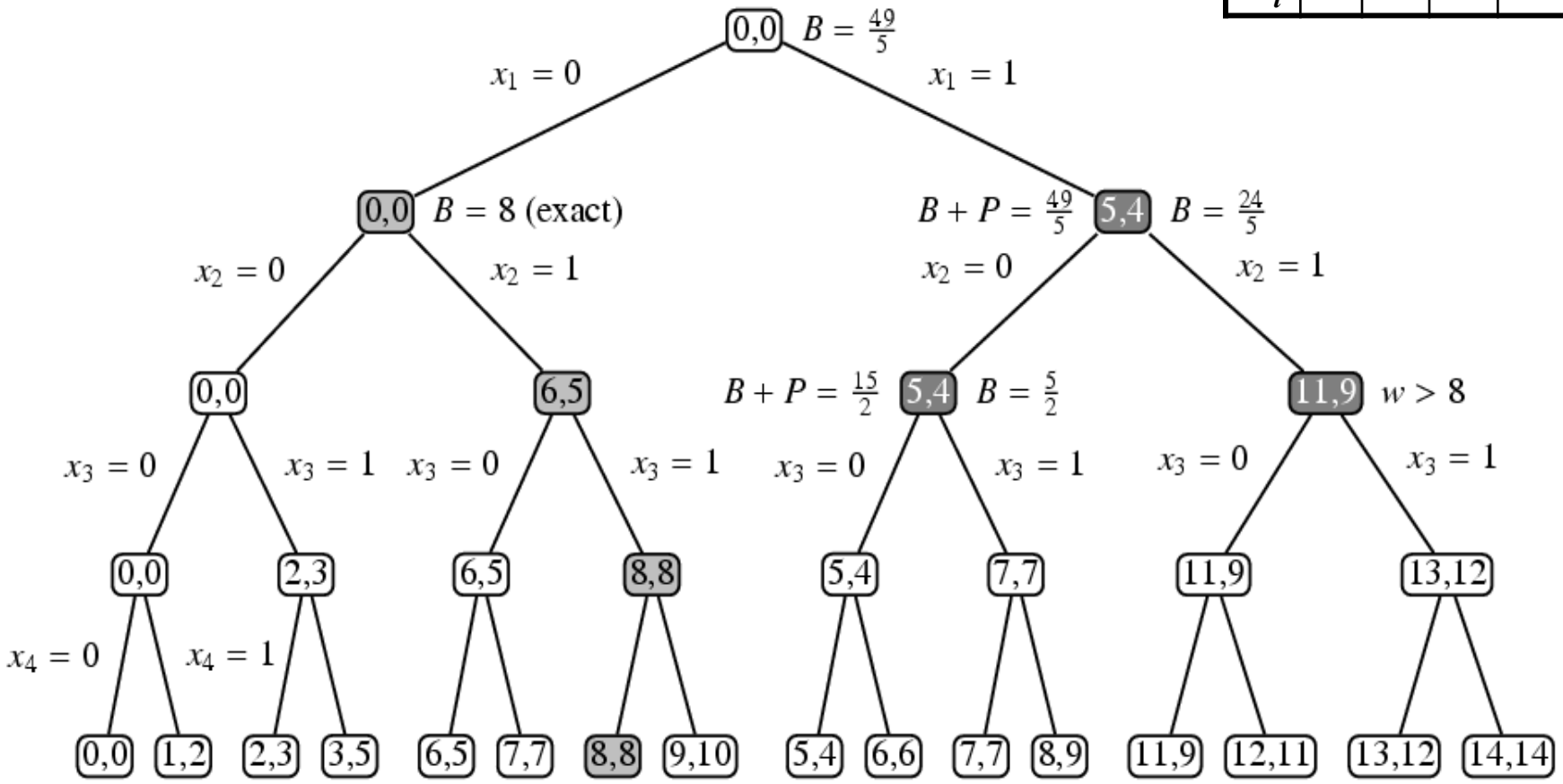
Example

- 0–1 Knapsack for capacity $W=8$
  - Two distinct orderings of items
  - Solution to the fractionary knapsack problem used as bounding function (value of B)
  - Heuristic: if the solution to the fractionary (continuous) problem = solution to 0-1 problem, the subtree has been solved optimally
  - In any case the profit per weight ratio ordering on the remaining objects must be used to obtain the fractionary knapsack solution required for the bounding function

Knapsack Example (1)

Dark grey=**killed**
Light grey=**exact**
Nodes = ***profit, weight***

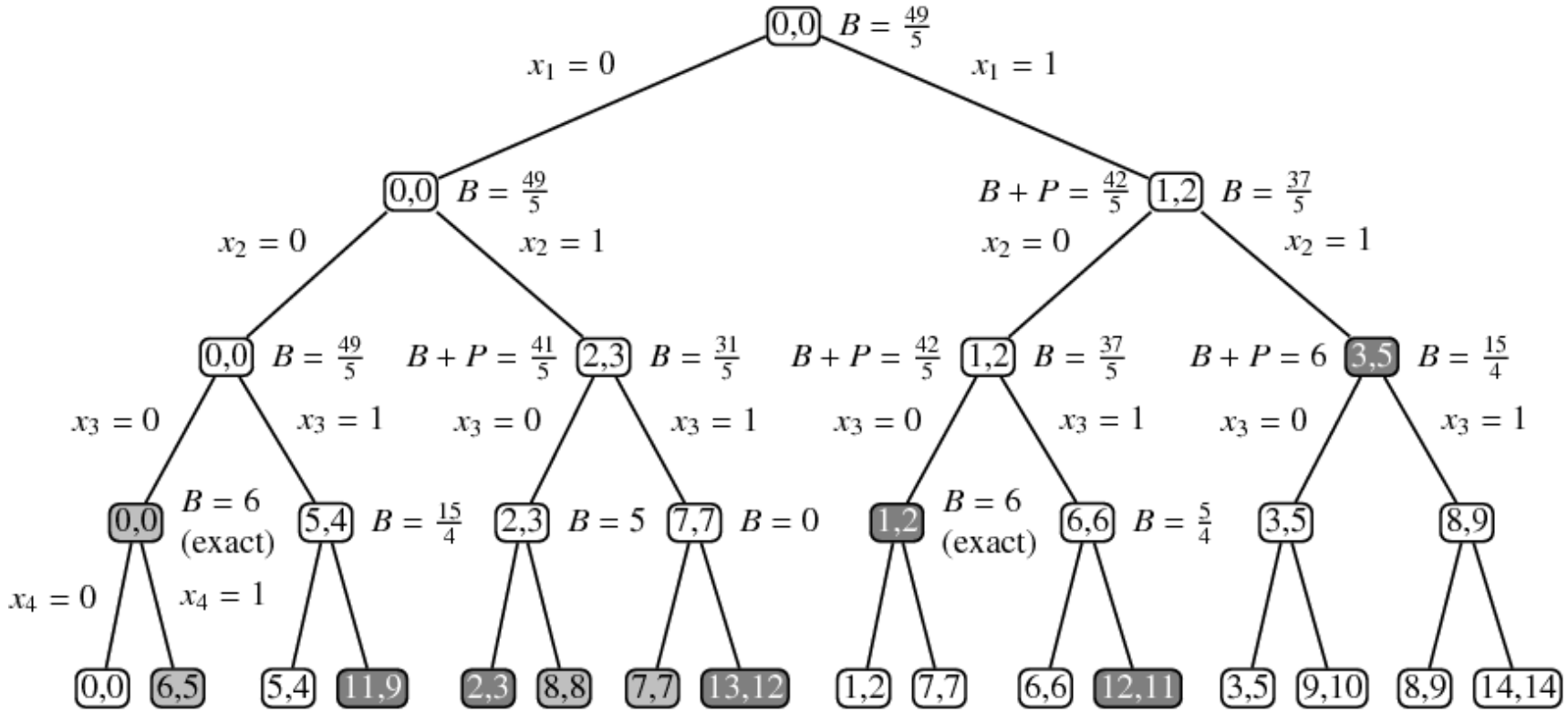
i	1	2	3	4
v_i	5	6	2	1
w_i	4	5	3	2



Knapsack Example(2)

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

Dark grey=**killed**
Light grey=**exact**
Nodes = ***profit, weight***



Binary search as branch-and-bound

- The functions *left(p), right(p)* and *val(p)* are for selecting the left or right subtree and taking the value of the node *p* (of the root of the tree).

binsearch(*x*, *p*)

if *empty*(*p*) **then** failure

else if *val*(*p*) = *x* **then** success

else if *val*(*p*) < *x* **then** *binsearch*(*x*, *right*(*p*))

else *binsearch*(*x*, *left*(*p*))

Branch and Bound

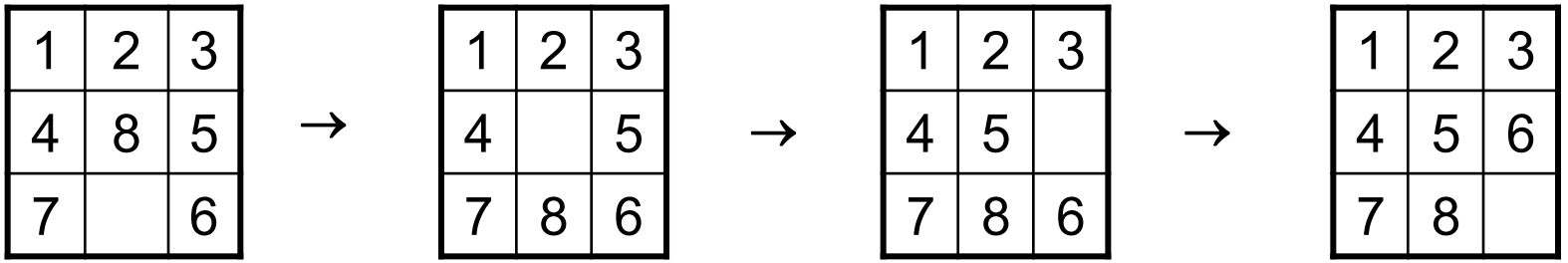
- General strategy: generate all children of the current ***E***-vertex before selecting a new ***E***-vertex.
  - Strategies for selecting a new ***E***-vertex:
    - LIFO order: depth first, using a stack.
    - FIFO order: breadth first, using a queue.
    - Best-first order: use a priority queue.
  - In each case, a bounding function is also used to kill vertices.

Example – the 8 puzzle

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

1	2	3
4	5	6
7	8	

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

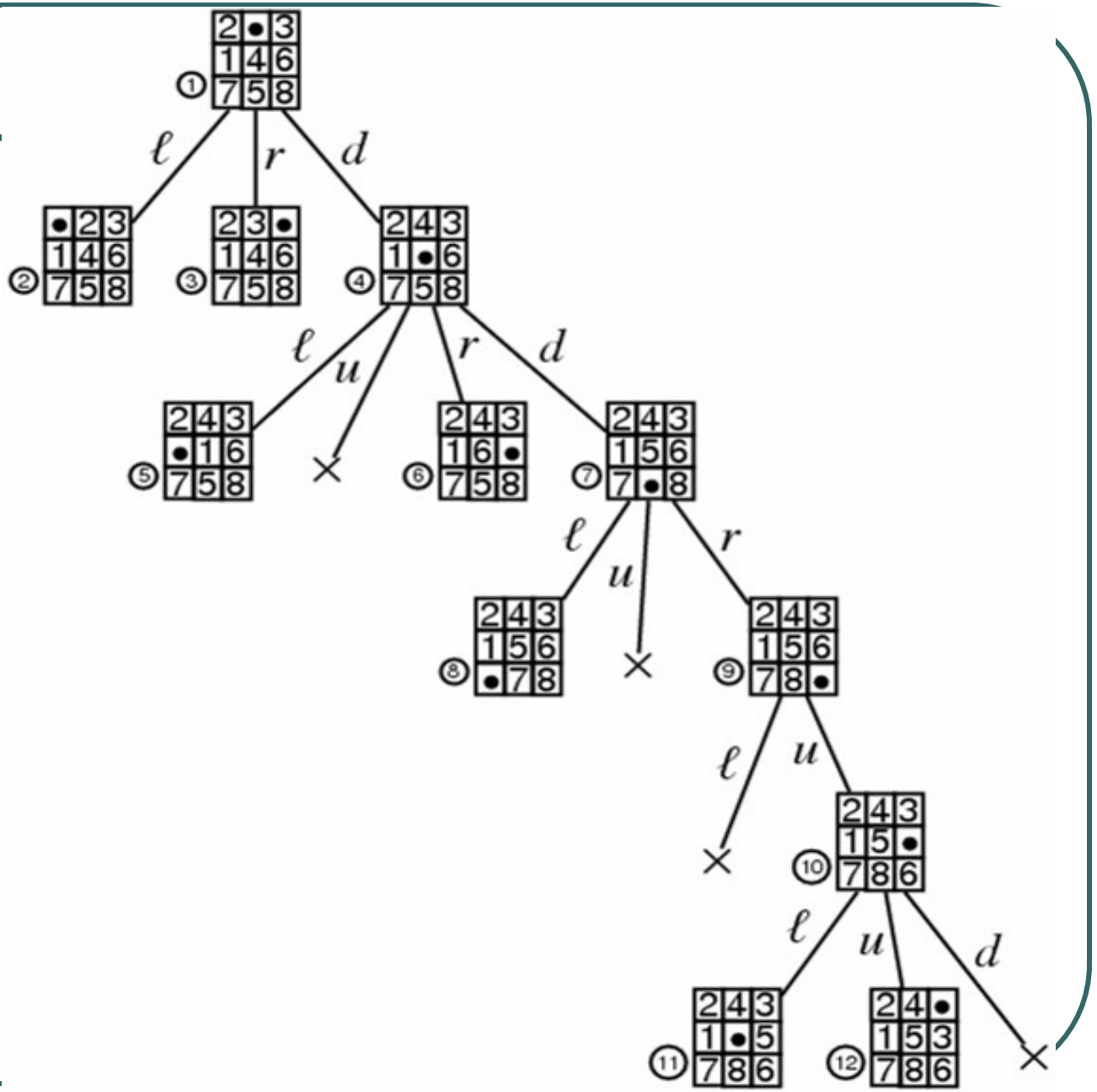


- Notation for direction where the empty slot "moves":

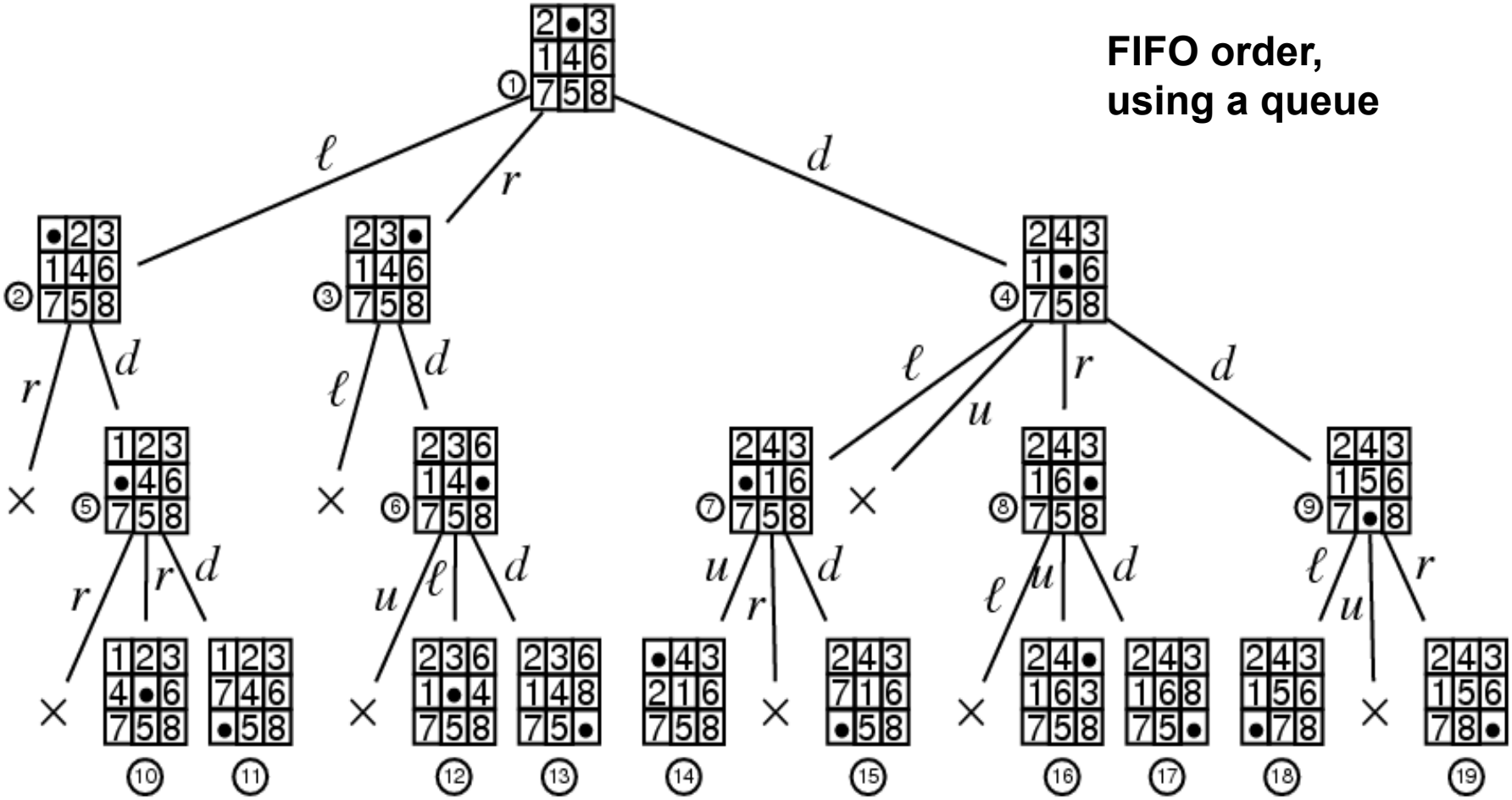
ℓ = left, r = right, u=up, d=down

Depth-first Expansion for the 8-puzzle

LIFO order, using a stack



Breadth-first Expansion for the 8-puzzle



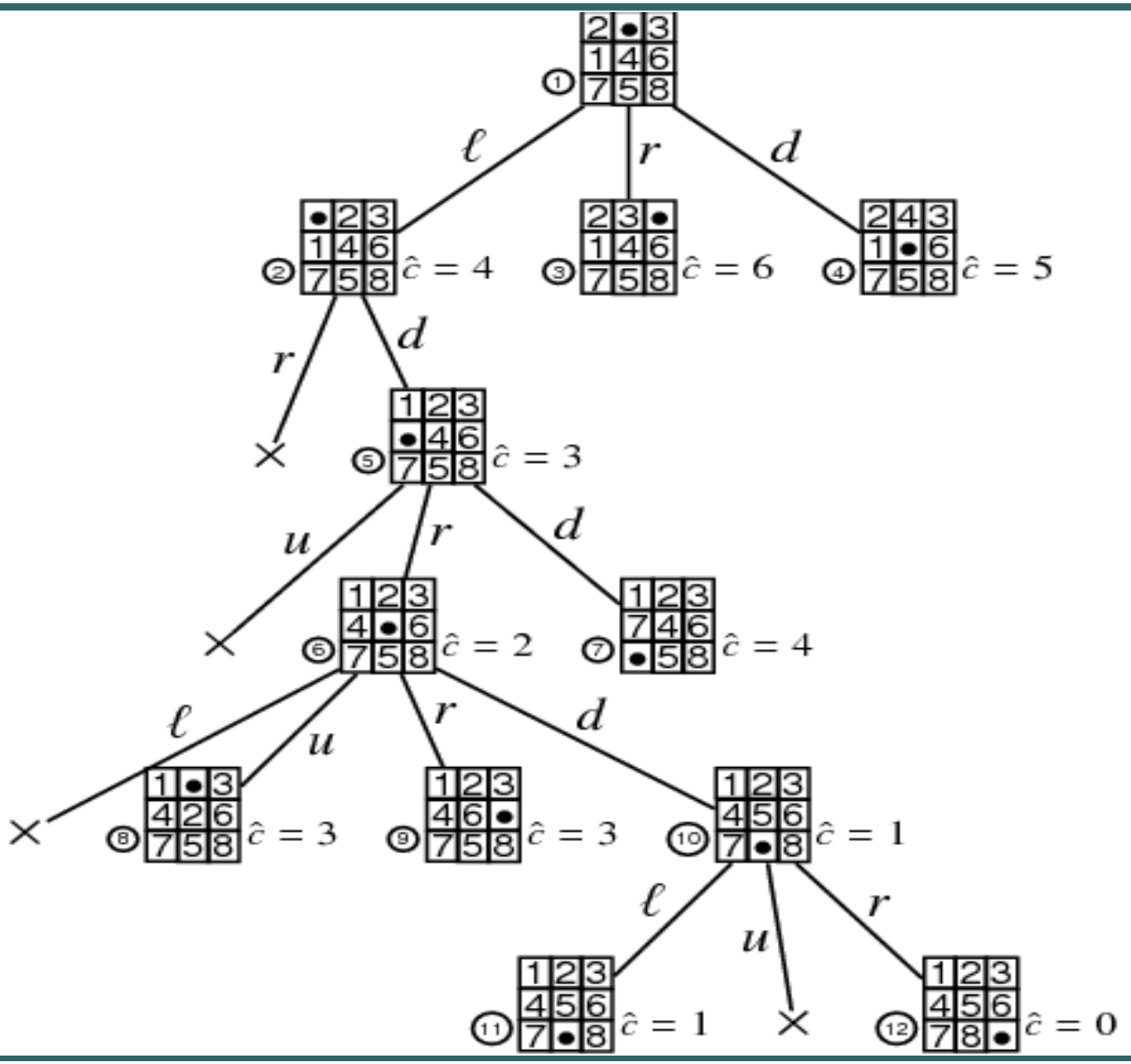
Best-first Search with Branch and Bound

- A *cost function,* c is associated with a best-first strategy
- $c(v)$ is the cost of finding a solution from vertex v
- Reasonable measure of cost: number of additional vertices which must be generated in order to obtain a solution
- Problem: $c(v)$ is very difficult to compute, in general (without generating the solution first)
  - Therefore, an approximation $\hat{c}$ is used
  - For the 8-puzzle, an appropriate approximation might be:

    $\hat{c}$ = number of tiles which are out of place
  - Here, the empty slot is not considered as a tile

Best-first Expansion for the 8-puzzle

Best-first order, using a priority queue



Desirable Properties for $\hat{c}$

- Key properties:
  - $\hat{c}$ should be easy to compute
  - $\hat{c}$ should give a good approximation for c
- Commonly used form for $\hat{c}$:

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

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

Selection of $k(v)$

- $k(v)=0$
  - A cost already incurred into should not enter into the evaluation
- $k(v)>0$
  - $k(v)=0$ favors deep searches
  - If $|\hat{g}(v)-c(v)|$ is large, then at very deep level the wrong path may get expanded
  - $k(v)$ adds a breadth first component
- Possible choice of $k(v)$ for the 8-puzzle: length of path from the root to v

Important properties for $\hat{c}$

- It is important to know whether a leaf vertex reached in the process is an optimal solution
- An approximate cost function $\hat{c}$ must be *admissible*, i.e. for all vertex pairs (v, w)

$$\hat{c}(v) < \hat{c}(w) \Leftrightarrow c(v) < c(w)$$

where $c(v)$ is the value of the best leaf beneath vertex v

- Hard in practice. Weaker condition:
  - $\hat{c}(v) \leq c(v)$ for all vertices
  - $\hat{c}(v) = c(v)$ for all answer vertices

0-1 Knapsack with Branch and Bound

- Leaf vertex v is identified with the solution vector $(x_1, x_2, \ldots, x_n)$ which defines a path from the root to v
- Its'a a maximization problem, reverse inequalities
- Definition for cost

$$c(x) = \begin{cases} \sum_{i=1}^{n} v_i x_i & \text{for a feasible answer vertex on the path of } x \\ -\infty & \text{for an illegal leaf vertex (too much weight)} \\ \max\left(\begin{Bmatrix} c(\text{LeftChild}(x)) \\ c(\text{RightChild}(x)) \end{Bmatrix}\right) & \text{for a non-leaf} \end{cases}$$

0-1 Knapsack with Branch and Bound

- Approximation function for a vertex x at depth j in the tree

$$\hat{c}(x) = \sum_{i=1}^{j} v_i x_i + \text{Profit}(\text{FKnap}(j+1, n, W - \sum_{i=1}^{j} w_i x_i))$$

where $\text{Profit}(\text{FKnap}(p, q, W))$ denotes the profit obtain in the solution of the fractionary knapsack problem with items $\{item_{j+1}, item_{j+2,} \ldots, item_n\}$ and capacity W

- Vertex-killing function:

$$u(x) = \sum_{i=1}^{j} v_i x_i + Greedy(\{item_{j+1}, \ldots, item_n\})$$

0-1 Knapsack with Branch and Bound

- $\text{Greedy}(\{item_{j+1}, \ldots, item_n\})$ is the value obtained by applying a greedy strategy, with items ordered by profit for a knapsack with capacity

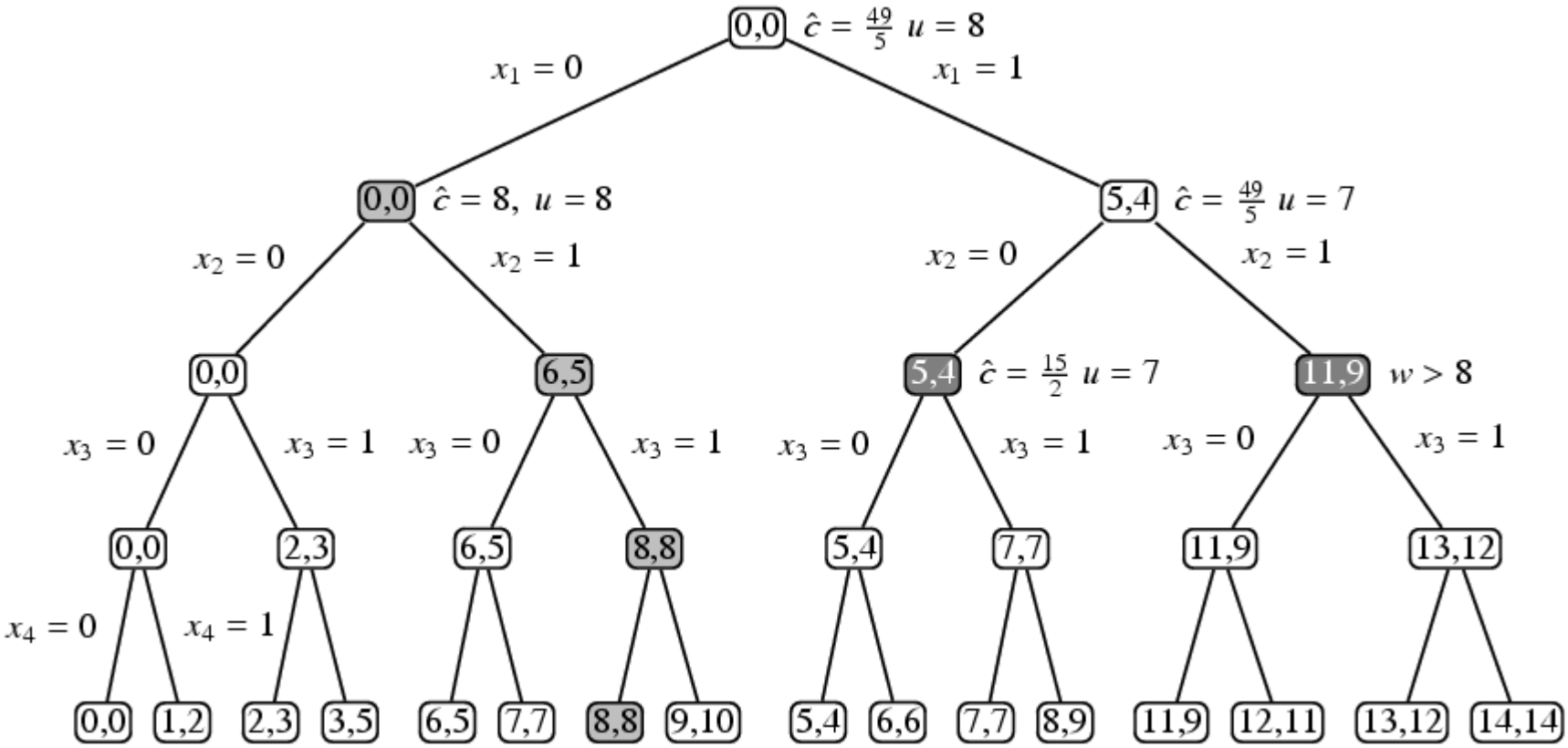
$$W - \sum_{i=1}^{j} w_i x_i$$

- The vertex is killed whenever

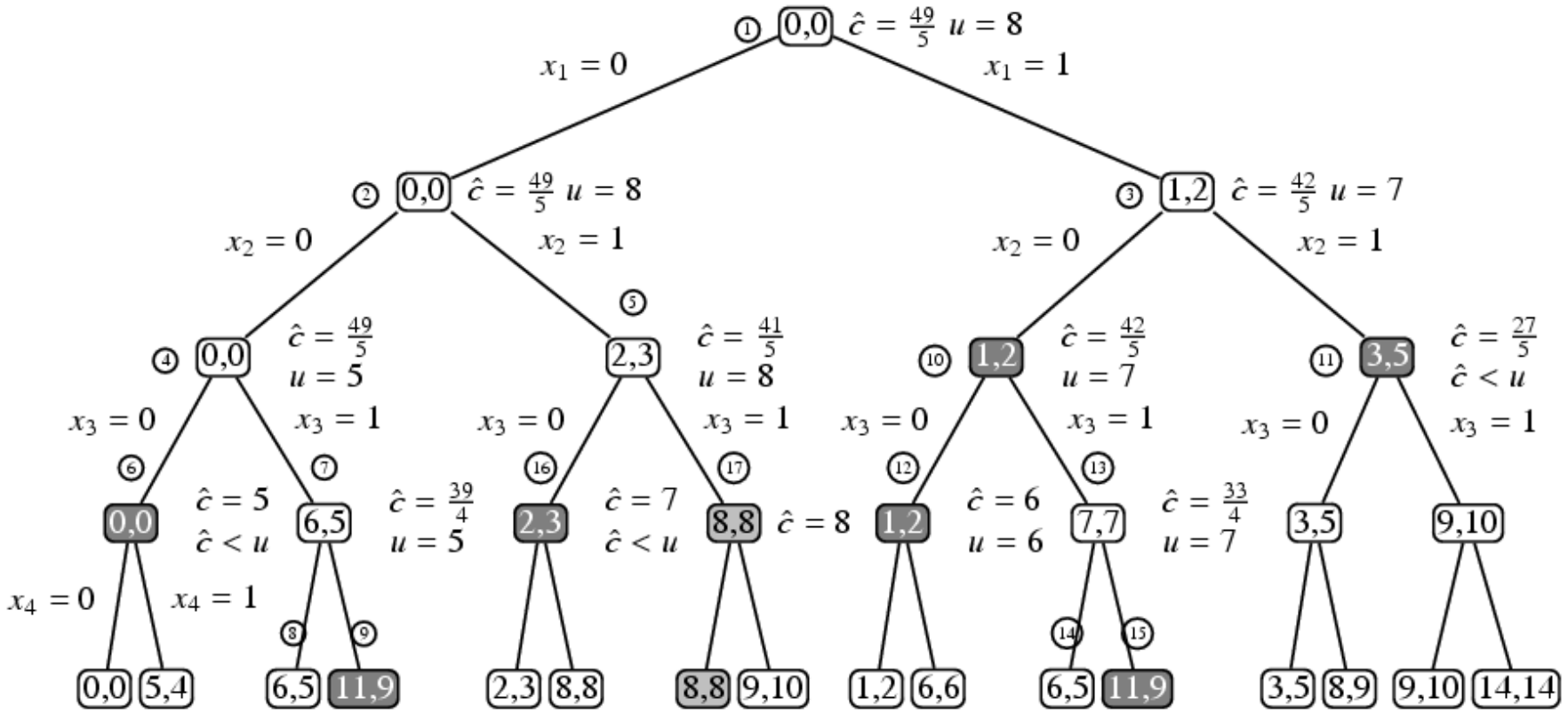
$$\hat{c}(x) < U = \max(\{u(x) \mid x \text{ has been generated}\})$$

- Evaluation also halted when the computation of $\hat{c}(x)$ yields an exact solution of the fractionary knapsack problem.

0-1 Knapsack w/ Branch and Bound Example



0-1 Knapsack w/ Branch and Bound Example



TSP Branch and Bound Example

- Consider the traveling salesman problem. A salesman has to visit each of n cities (at least) once each, and wants to minimize total distance traveled.
- Consider the route problem to be the problem of finding the shortest route through a set of cities visiting each city once
- Split the node into two child problems
  - Shortest route visiting city A first
  - Shortest route not visiting city A first
- Continue subdividing similarly as the tree grows

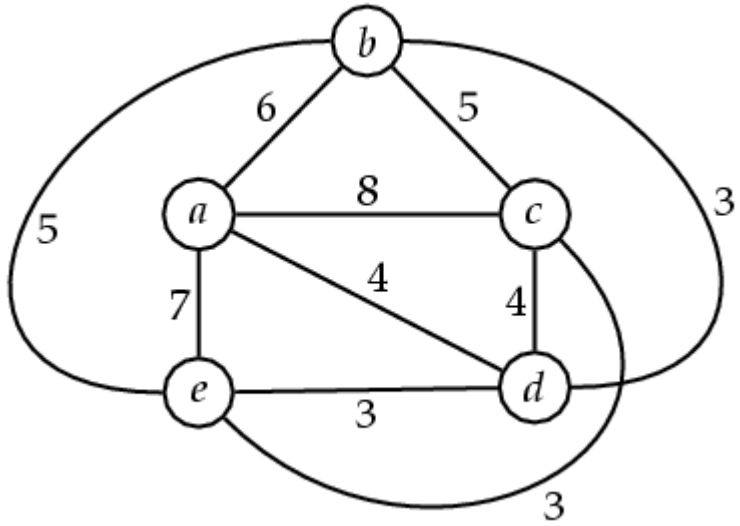
Bounding Heuristics

- A heuristic is a rule of thumb. That is, a heuristic is generally a good idea but some times it doesn't work.
- An example of a heuristic for packing a suitcase might be, put in the big items first such as your shoes, then put the small items in using up the remaining fragmented space.
- For the TSP we might have heuristics for selecting the next city to visit. Maybe go to the nearest?

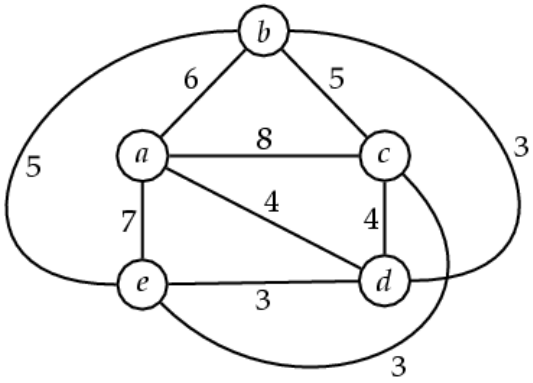
TSP (again)

- Lower bound for cost of any TSP tour:

$$\frac{1}{2} \sum_{\text{all nodes } n} Cost(\text{two tour edges adjacent to } n)$$



TSP example


b
a
c
d
e
6
5
8
3
5
7
4
4
3
3

all tours
tours with (a,b)
tours without (a,b)
tours with (a,b) and (a,c)
tours with (a,b) not (a,c)
tours with (a,c) not (a,b)
tours without (a,b) or (a,c)
tours with (a,b) and (a,c) not (a,d)
tours with (a,b) not (a,c) or (a,d)
tours with (a,b) and (a,d) not (a,c)
tours with (a,c) not (a,b) or (a,d)
tours with (a,c) and (a,d) not (a,b)
tours without (a,b), (a,c) or (a,d)
tours with (a,d) not (a,b) or (a,c)

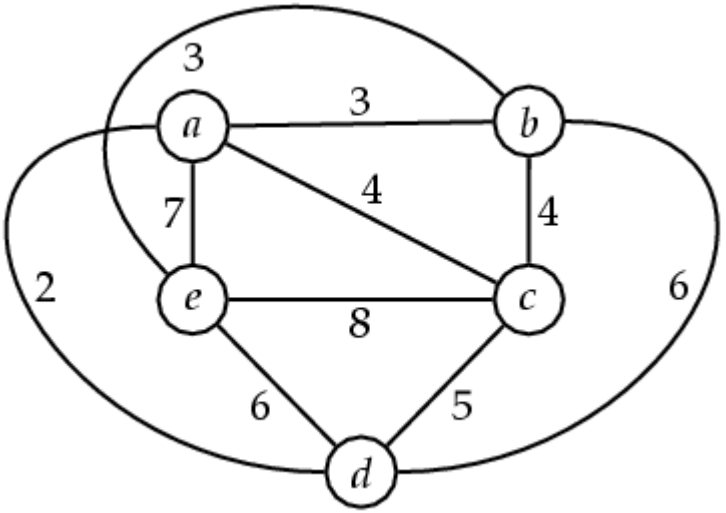
TSP Inferences

- Consider the edges in lexicographic order
- Each time we *branch*, by considering the two children of a node, we try to infer additional decisions regarding which edges must be included or excluded from tours represented by those nodes. Rules for these inferences are:
  - If excluding an edge (x, y) would make it impossible for x or y to have as many as two adjacent edges in the tour, then (x, y) must be included.
  - If including (x, y) would cause x or y to have more than two edges adjacent in the tour, or would complete a non-tour cycle with edges already included, then (x, y) must be excluded.

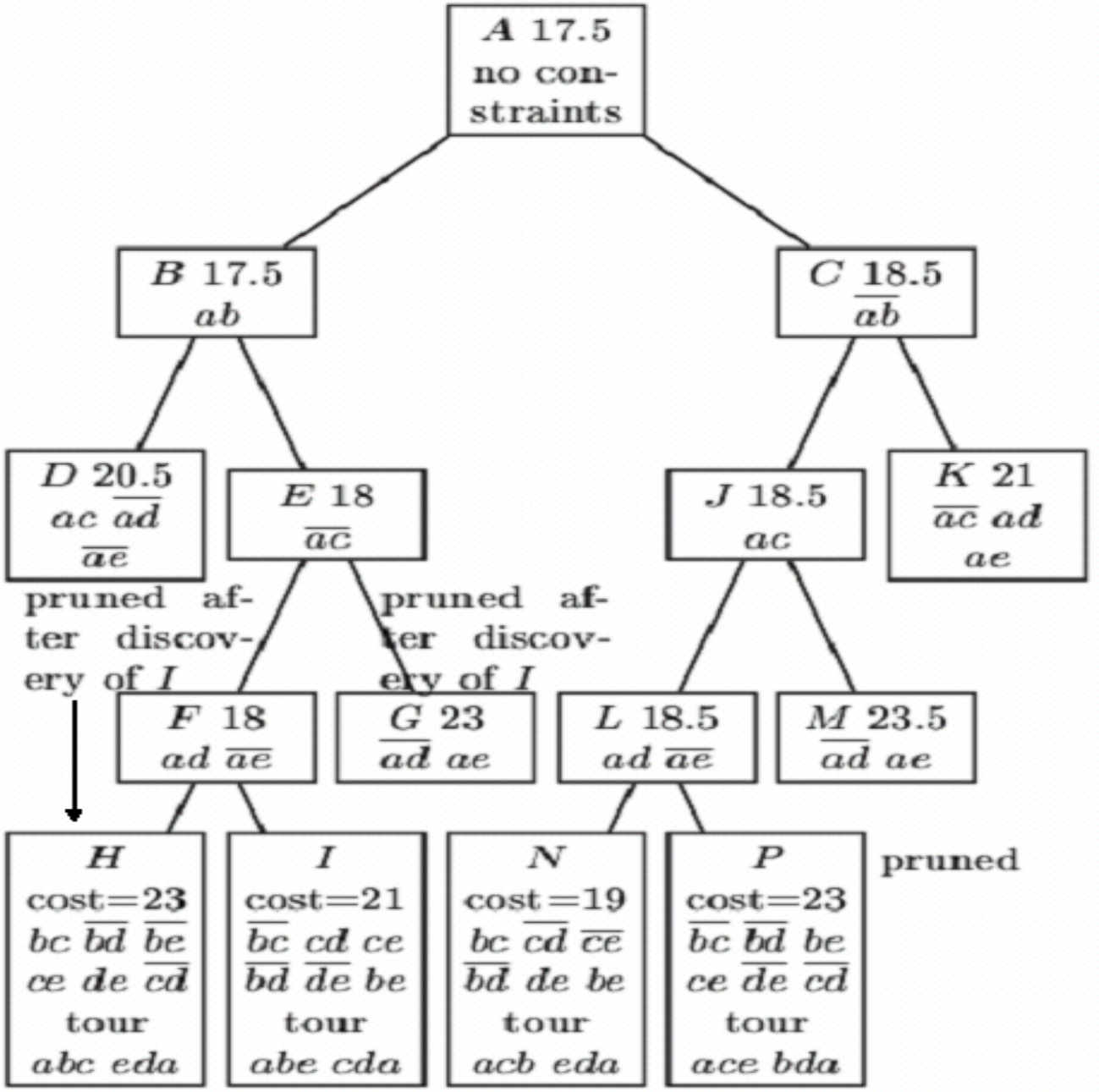
TSP Pruning

- When we branch, after making what inferences we can, we compute lower bounds for both children.
  - If the lower bound for a child is as high or higher than the lowest cost tour found so far, we can *prune* that child and need not construct or consider its descendants.

Another graph for TSP



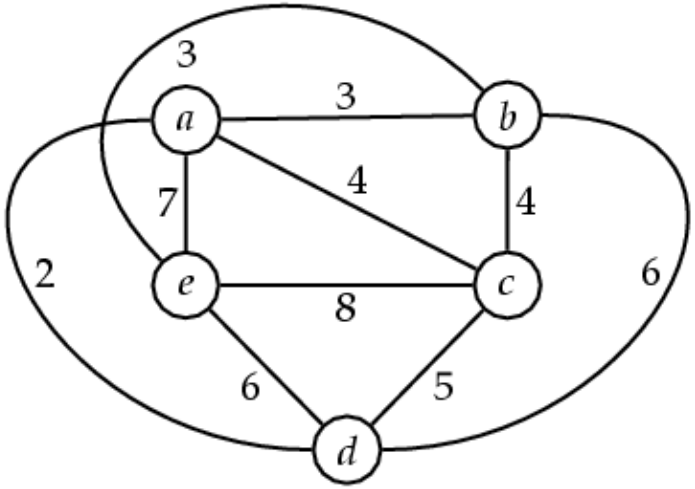
TSP Pruning Example

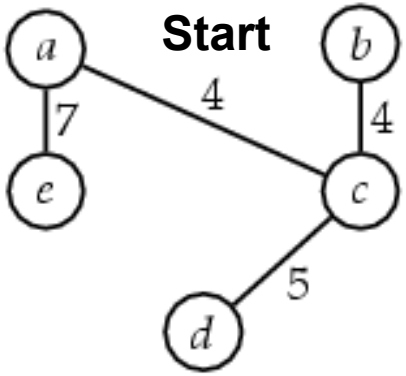


Local Search Algorithms

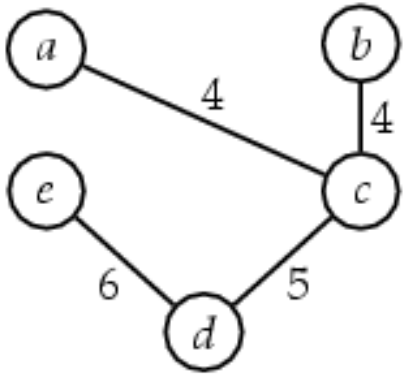
- Sometimes an optimal solution may be obtain if:
  - Start with a random solution.
  - Apply to the current solution a transformation from some given set of transformations to improve the solution. The improvement becomes the new "current" solution.
  - Repeat until no transformation in the set improves the current solution.
- Note: the method makes sense if we we can restrict our set of transformations to a small set, so we can consider all transformations in a short time (e.g. for a problem of size n, $O(n^2)$...$O(n^3)$ transformations)
- Transformations are called ***local transformations***, and the method is called ***local search***

Local Search Example: the MST problem

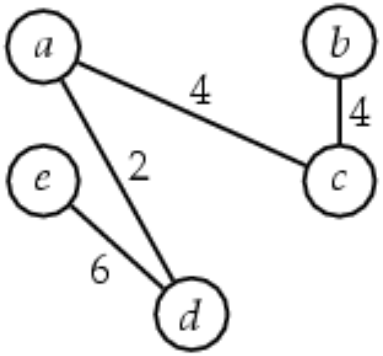




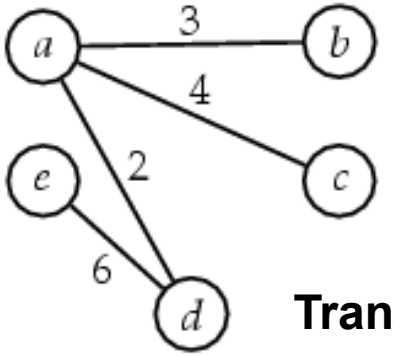
(a)



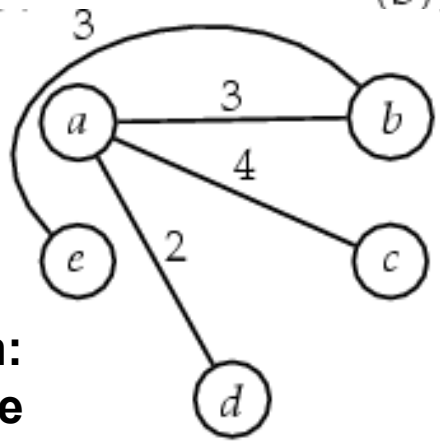
(b)



(c)



(d)



(e)

Transformation: remove an edge and add another to decrease cost

Some conclusions

- **Backtracking**
  - easy to implement
  - little memory required
  - slow to run
- **Branch & Bound**
  - difficult to implement
  - large amounts of memory required
  - maybe faster than backtracking
- **Dynamic Programming**
  - tricky to see the solution construction
  - some memory requirements
  - polynomially fast
  - easy to implement, especially via memoization
- **Greedy Algorithms**
  - polynomially fast
  - little memory required
  - need proofs for optimality of solution, or the solution may be suboptimal

P and NP

- If there's an algorithm to solve a problem that runs in polynomial time, the problem is said to be in set **P**.
- If the outcome of an algorithm to solve a problem can be *verified* in polynomial time, the problem is said to be in the set **NP** (non-deterministic polynomial).
- There is a set of problems in **NP** for which a polynomial solution to one is a solution to all. The set is called **NP-complete**.
- If such a polynomial solution exists, **P=NP**.
- It is not known whether $P \subset NP$ or $P = NP$.

Examples of NP Complete Problems

- **Traveling Salesman Problem**
- **0/1 Knapsack Problem**
- **Graph Coloring Problem**: can you color a graph using $k \geq 3$ colors such that no adjacent vertices have the same color?

Reading

- AHU, chapter 10, sections 4 and 5
- Preiss, chapter: Algorithmic Patterns and Problem Solvers, section Backtracking Algorithms
- CLR, chapter 35, section 3, CLRS chapter 34, sections 1-3
- Notes