Segment 8

Recurrences & Backtracking

Recurrences

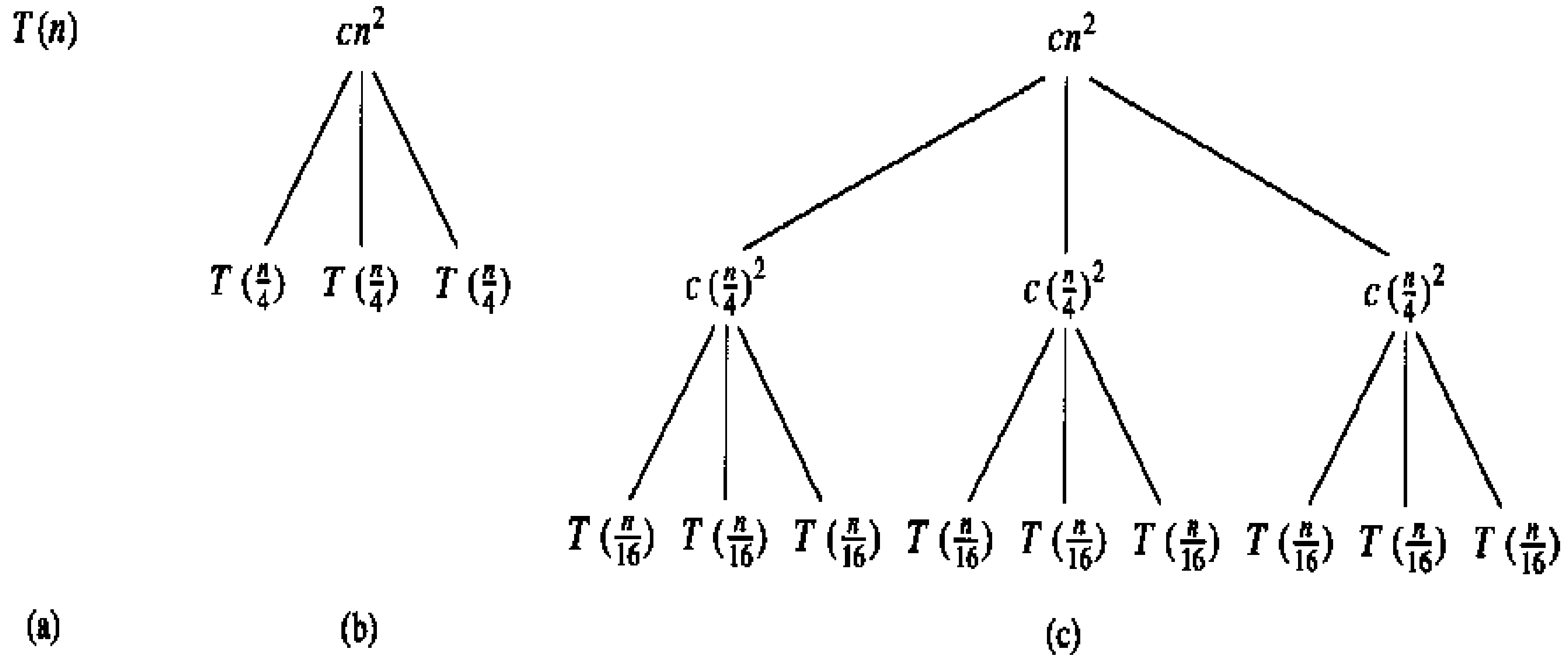
- A *recurrence* is an equation or inequality that describes a function in terms of its value of smaller inputs.
- We have studied the recurrence in detail.
- We also known about three methods that are used to solve the recurrences.
- Here we will know another method to solve the recurrences, i.e. Recurrence tree method.

Recurrence Trees

- Allow you to visualize the process of iterating the recurrence
- Allows you make a good guess for the substitution method
- Or to organize the bookkeeping for iterating the recurrence
- Convert the recurrence into a tree:
  - Each node represents the cost of a single subproblem in the set of recursive function invocation.
  - Sum up the costs within each level of the tree to obtain a set of pre-level costs.
  - Then sum all the pre-level costs to determine the total of all levels of the recursion.

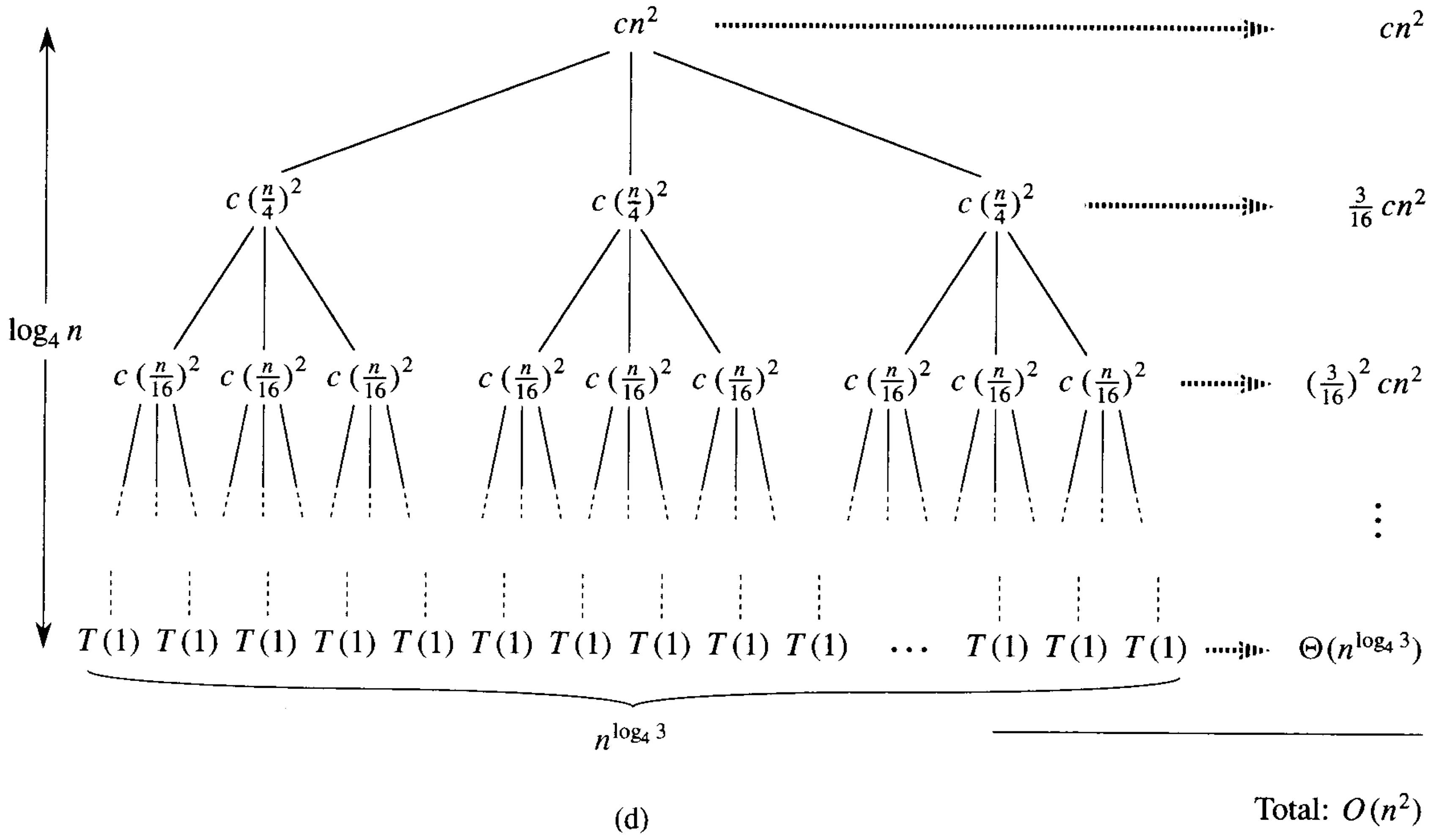
Recursion-tree: Example 1

$$T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$$



$T(n)$
cn^2
$T(\frac{n}{4})$ $T(\frac{n}{4})$ $T(\frac{n}{4})$
cn^2
$c(\frac{n}{4})^2$ $c(\frac{n}{4})^2$ $c(\frac{n}{4})^2$
$T(\frac{n}{16})$ $T(\frac{n}{16})$ $T(\frac{n}{16})$ $T(\frac{n}{16})$ $T(\frac{n}{16})$ $T(\frac{n}{16})$ $T(\frac{n}{16})$ $T(\frac{n}{16})$ $T(\frac{n}{16})$
(a)
(b)
(c)

Recursion-tree method



(d)

Recursion-tree method

- Subproblem size for a node at depth i, $\frac{n}{4^i}$
- Subproblem size hits n=1 when $\frac{n}{4^i}$ =1 or when $i = \log_4 n$
- Total level of tree , $\log_4 n + 1$ (0,1,2,… $\log_4 n$.)
- Number of nodes at depth i , 3^i
- Cost of each node at depth i , $c(\frac{n}{4^i})^2$
- Total cost over all nodes at depth i , $3^i c(\frac{n}{4^i})^2 = (\frac{3}{16})^i cn^2$
- Last level, at depth $\log_4 n$, has $3^{\log_4 n} = n^{\log_4 3}$ nodes

  and cost is $n^{\log_4 3} T(1)$, which is, $\Theta(n^{\log_4 3})$

Recursion-tree method

Prove of $3^{\log_4 n} = n^{\log_4 3}$

$$\begin{aligned}\log_4 3^{\log_4 n} &= (\log_4 n)(\log_4 3)\\ &= (\log_4 3)(\log_4 n)\\ &= \log_4 n^{\log_4 3}\end{aligned}$$

We conclude, $3^{\log_4 n} = n^{\log_4 3}$

The cost of the entire tree

$$T(n) = cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \ldots + \left(\frac{3}{16}\right)^{\log_4 n - 1} cn^2 + \Theta(n^{\log_4 3})$$

$$= \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta\left(n^{\log_4 3}\right)$$

$$= \frac{(3/16)^{\log_4 n} - 1}{(3/16) - 1} cn^2 + \Theta(n^{\log_4 3}).$$

The cost of the entire tree

Backing up one step and applying equation, $\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$

$$T(n) = \sum_{i=0}^{\log_4 n-1} \left(\frac{3}{16}\right)^i cn^2 + \Theta\left(n^{\log_4 3}\right)$$

$$< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta\left(n^{\log_4 3}\right)$$

$$= \frac{1}{1-(3/16)} cn^2 + \Theta\left(n^{\log_4 3}\right)$$

$$= \frac{16}{13} cn^2 + \Theta(n^{\log_4 3})$$

$$= O(n^2)$$

Recurrence Tree: Example 2

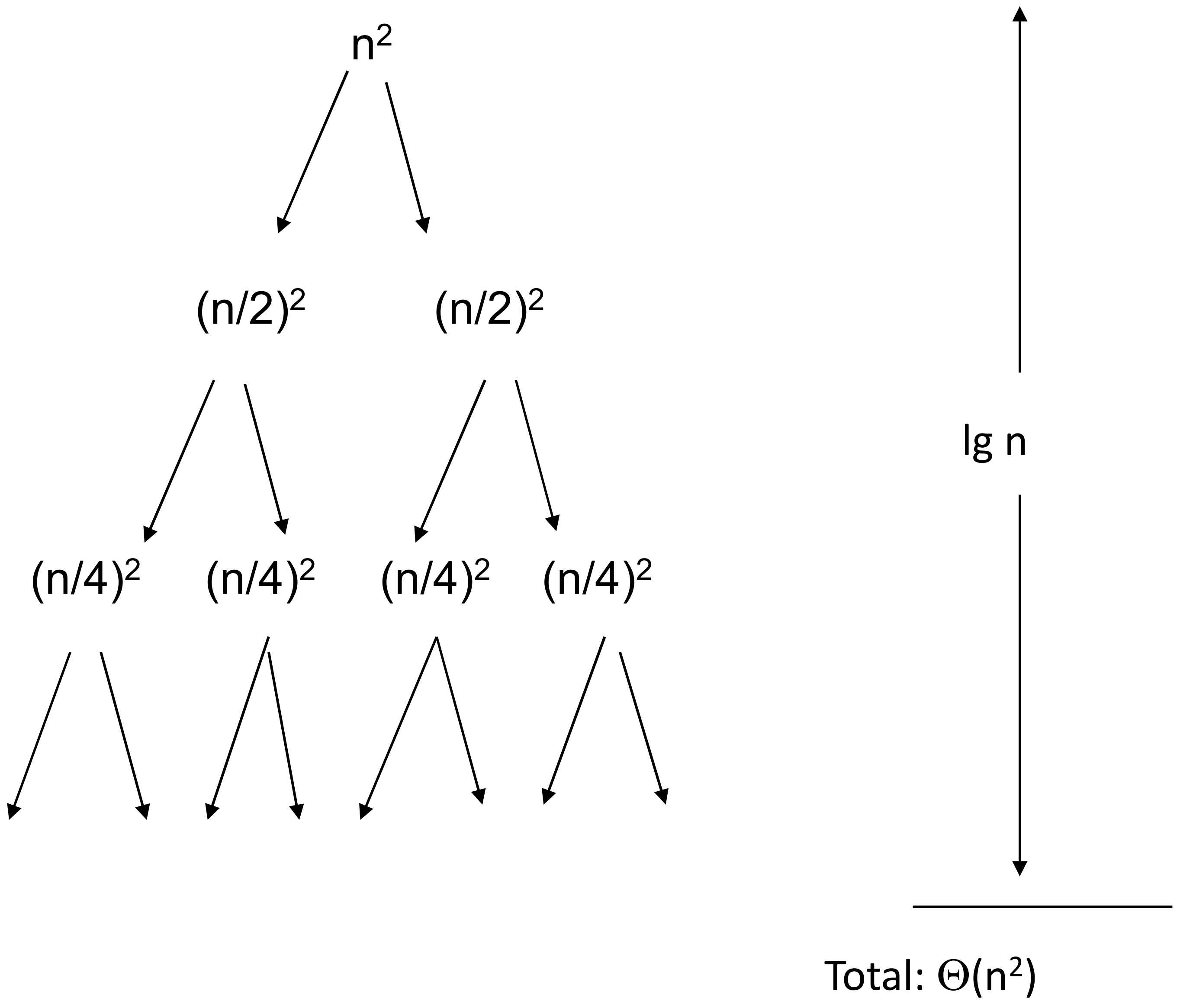
$$W(n) = 2W(n/2) + n^2$$

$$\begin{tikzcd}
& n^2 \arrow[dl] \arrow[dr] & \\
W(n/2) & & W(n/2)
\end{tikzcd}$$

$$\begin{tikzcd}
& n^2 \arrow[dl] \arrow[dr] & & \\
(n/2)^2 \arrow[d] \arrow[dr] & & (n/2)^2 \arrow[d] \arrow[dr] & \\
W(n/4) & W(n/4) & W(n/4) & W(n/4)
\end{tikzcd}$$

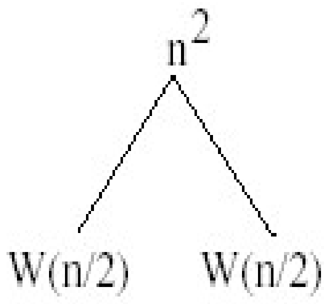
$$\begin{tikzcd}
& & & n^2 \arrow[dll] \arrow[drr] & & & & \\
& (n/2)^2 \arrow[dl] \arrow[dr] & & & & (n/2)^2 \arrow[dl] \arrow[dr] & & \\
(n/4)^2 \arrow[d] \arrow[dr] & & (n/4)^2 \arrow[d] \arrow[dr] & & (n/4)^2 \arrow[d] \arrow[dr] & & (n/4)^2 \arrow[d] \arrow[dr] & \\
{} & {} & {} & {} & {} & {} & {} & {}
\end{tikzcd}$$

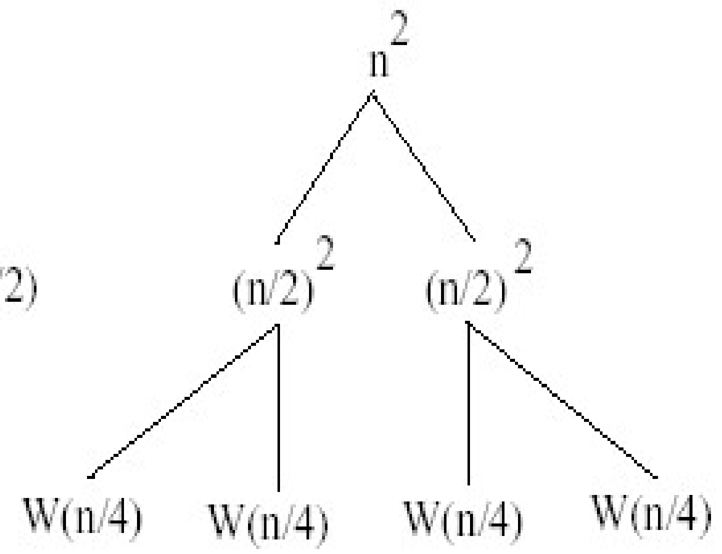
Recurrence Tree Example



Total: $\Theta(n^2)$

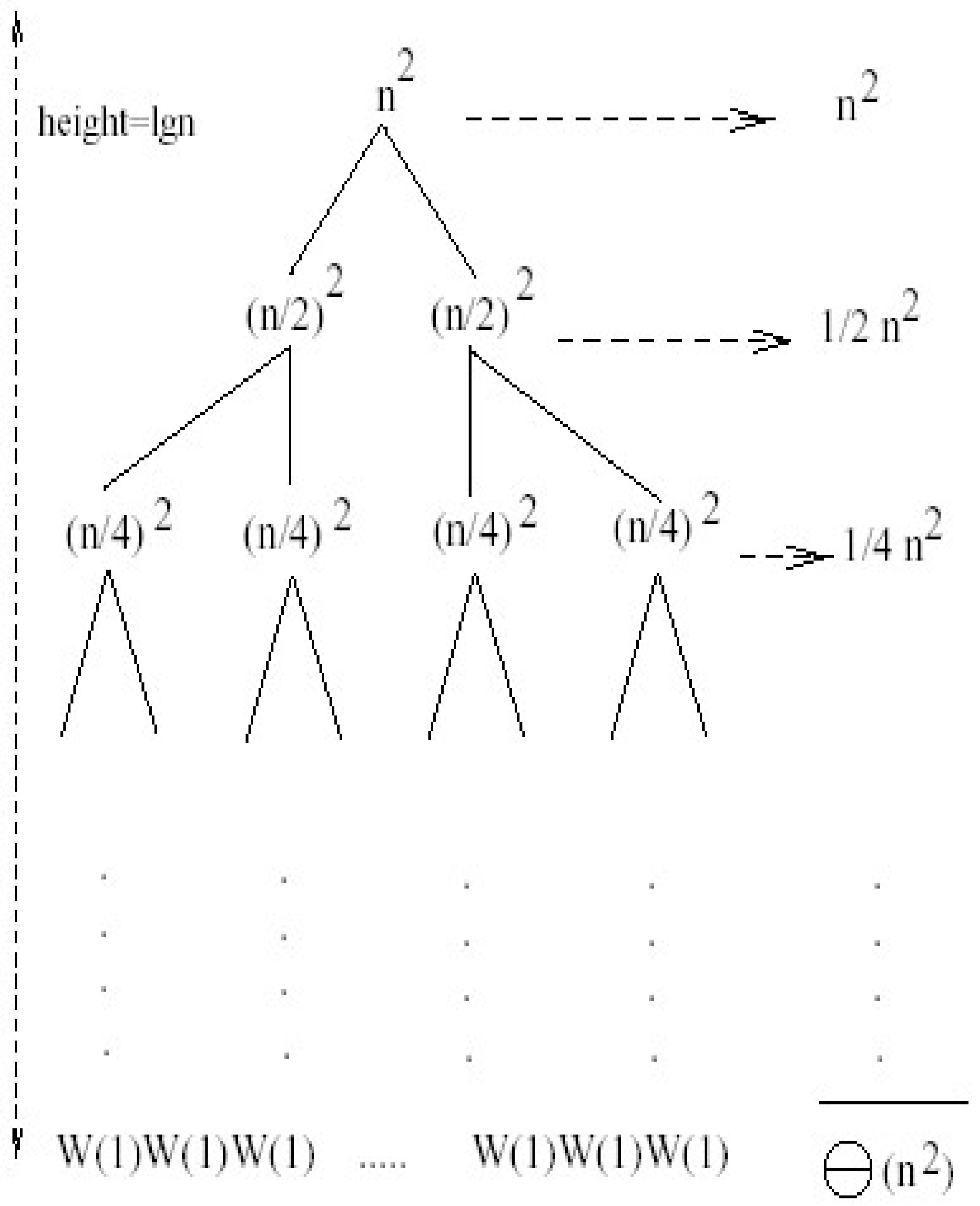
11





$W(n/2)=2W(n/4)+(n/2)^2$

$W(n/4)=2W(n/8)+(n/4)^2$



The cost of the entire tree

- Subproblem size at level i is: $n/2^i$
- Subproblem size hits 1 when $n/2^i=1 \Rightarrow i = \log n$
- Total level of tree , $\log n+1$ (0,1,2,… logn.)
- Cost of the problem at level i = $(n/2^i)^2$
- No. of nodes at level $i = 2^i$
- Total cost:

$$W(n)=\sum_{i=0}^{\lg n-1}\frac{n^2}{2^i}+2^{\lg n}W(1)=n^2\sum_{i=0}^{\lg n-1}\left(\frac{1}{2}\right)^i+n\le n^2\sum_{i=0}^{\infty}\left(\frac{1}{2}\right)^i+O(n)=n^2\frac{1}{1-\frac{1}{2}}+O(n)=2n^2$$

$\Rightarrow W(n) = O(n^2)$

Example 3

$$T(n) = T(n/4) + T(n/2) + n^2$$

$$\begin{tikzcd}
 & n^2 \arrow[dl] \arrow[dr] & \\
T(n/4) & & T(n/2)
\end{tikzcd}$$

$$\begin{tikzcd}
 & & n^2 \arrow[dl] \arrow[dr] & & \\
 & (n/4)^2 \arrow[dl] \arrow[d] & & (n/2)^2 \arrow[d] \arrow[dr] & \\
T(n/16) & T(n/8) & & T(n/8) & T(n/4)
\end{tikzcd}$$

n^2

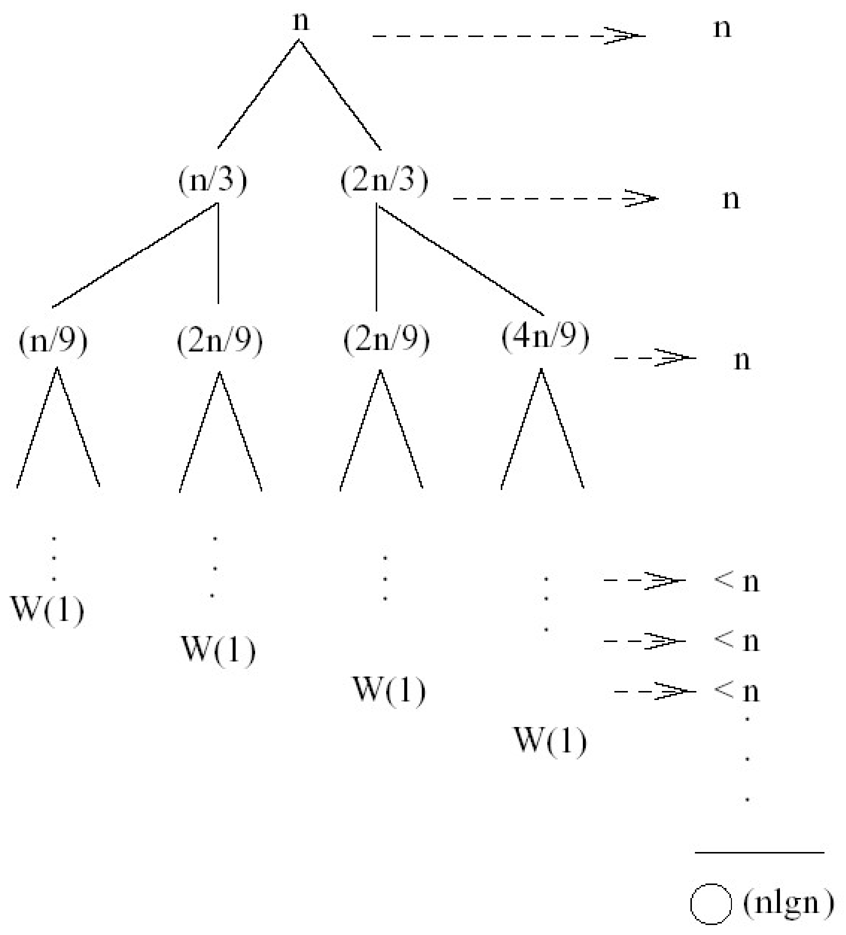
$5/16\ n^2$

$25/256\ n^2 = (5/16)^2 n^2$

Since the values decrease geometrically, the total is at most a constant factor more than the largest term and hence the solution is $\Theta(n^2)$

Example 4 (simpler proof)

$W(n) = W(n/3) + W(2n/3) + n$



- The longest path from the root to a leaf is:

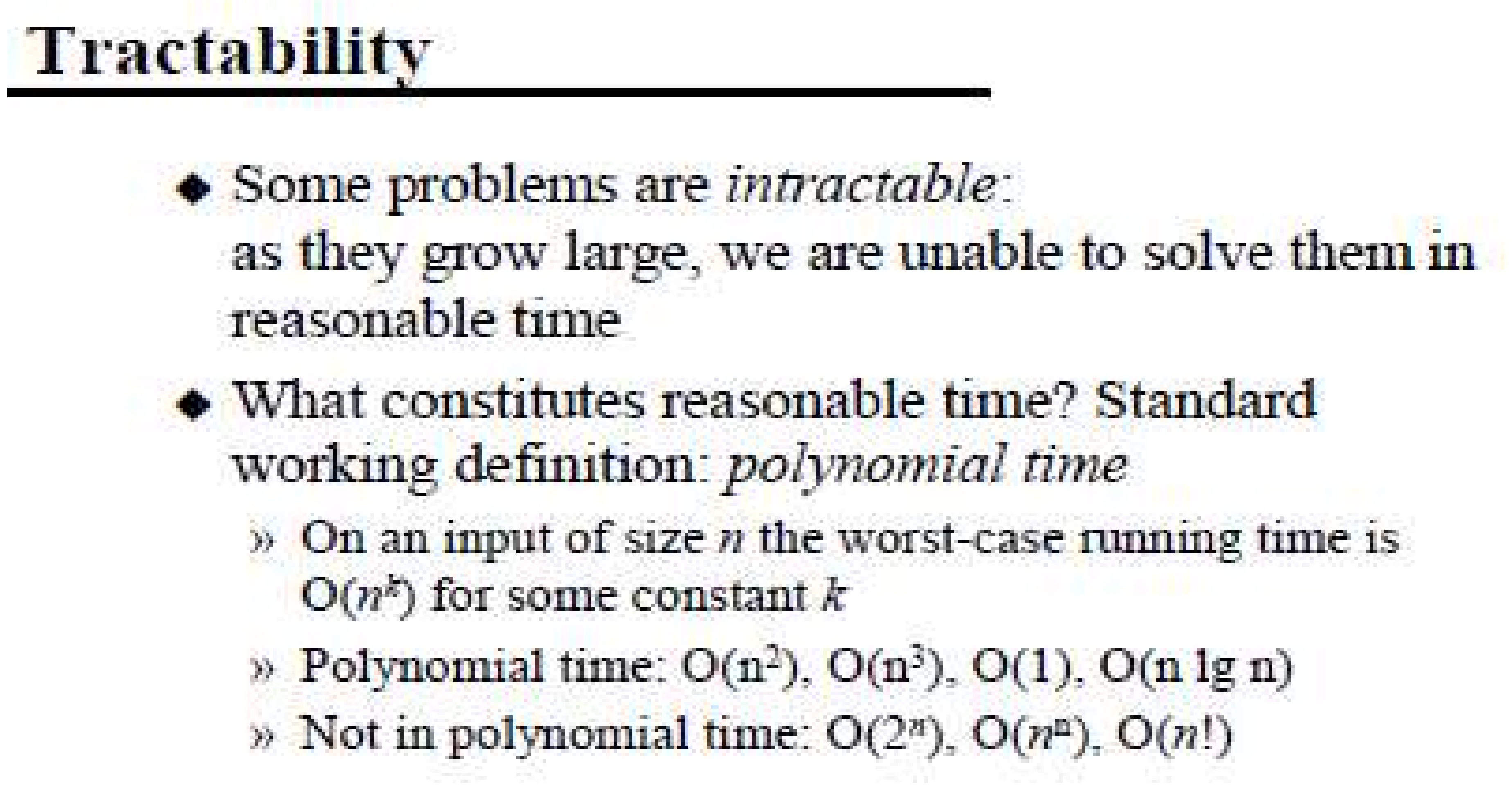
  $n \to (2/3)n \to (2/3)^2\, n \to \dots \to 1$
- Subproblem size hits 1 when

  $1 = (2/3)^i n \Leftrightarrow i=\log_{3/2} n$
- Cost of the problem at level i = n
- Total cost:

$$W(n) < n + n + \dots = n(\log_{3/2} n) = n\frac{\lg n}{\lg\frac{3}{2}} = O(n\lg n)$$

$\Rightarrow W(n) = O(n\lg n)$

P, NP, NP HARD AND NP – COMPLETE PROBLEMS



Tractability

- Some problems are *intractable*: as they grow large, we are unable to solve them in reasonable time
- What constitutes reasonable time? Standard working definition: *polynomial time*
  - » On an input of size *n* the worst-case running time is $O(n^k)$ for some constant *k*
  - » Polynomial time: $O(n^2)$, $O(n^3)$, $O(1)$, $O(n \lg n)$
  - » Not in polynomial time: $O(2^n)$, $O(n^n)$, $O(n!)$

Polynomial-Time Algorithms

- Are some problems solvable in polynomial time?
  - » Of course: every algorithm we've studied provides polynomial-time solution to some problem
- Are all problems solvable in polynomial time?
  - » No: Turing's "Halting Problem" is not solvable by any computer, no matter how much time is given
- Most problems that do not yield polynomial-time algorithms are either optimization or decision problems.

Optimization/Decision Problems

- Optimization Problems:
  - An optimization problem is one which asks, "What is the optimal solution to problem X?"
  - Examples:
    - 0-1 Knapsack
    - Fractional Knapsack
    - Minimum Spanning Tree
- Decision Problems
  - An decision problem is one which asks, "Is there a solution to problem X with property Y?"
  - Examples:
    - Does a graph G have a MST of weight $\leq W$?

Optimization/Decision Problems

- An optimization problem tries to find an optimal solution
- A decision problem tries to answer a yes/no question
- Many problems will have decision and optimization versions.
  - Eg: Traveling salesman problem
    - optimization: find hamiltonian cycle of minimum weight
    - decision: find hamiltonian cycle of weight $< k$
- Some problems are decidable, but *intractable*:
  as they grow large, we are unable to solve them in reasonable time
  - *What constitutes "reasonable time"?*
  - *Is there a polynomial-time algorithm that solves the problem?*

The Class *P*

P: the class of decision problems that have polynomial-time deterministic algorithms.

» That is, they are solvable in $O(p(n))$, where $p(n)$ is a polynomial on n

» A deterministic algorithm is (essentially) one that always computes the correct answer

Why polynomial?

» if not, very inefficient

» nice closure properties

» machine independent in a strong sense

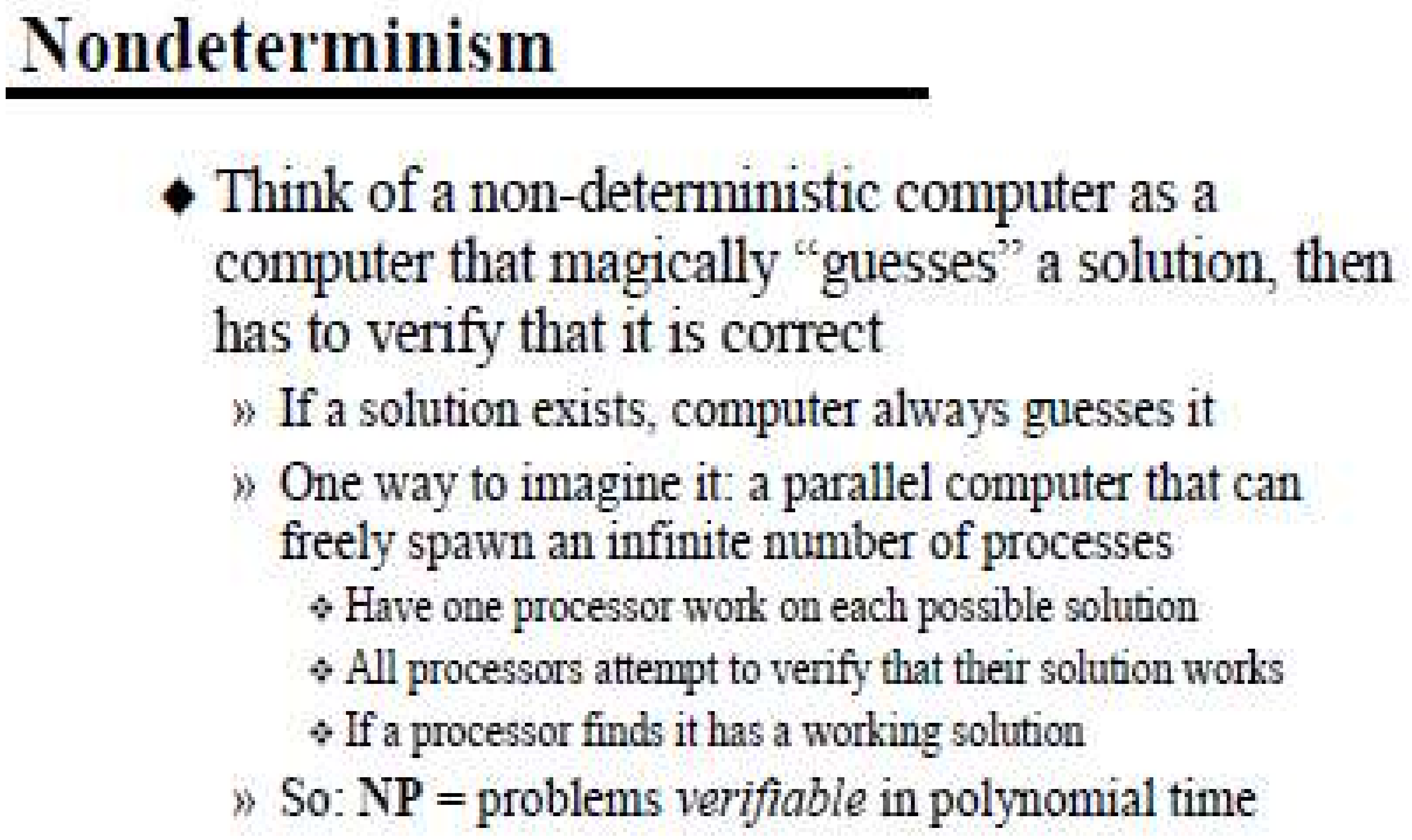
Sample Problems in P

- Fractional Knapsack
- MST
- Single-source shortest path
- Sorting
- Others?

The class *NP*

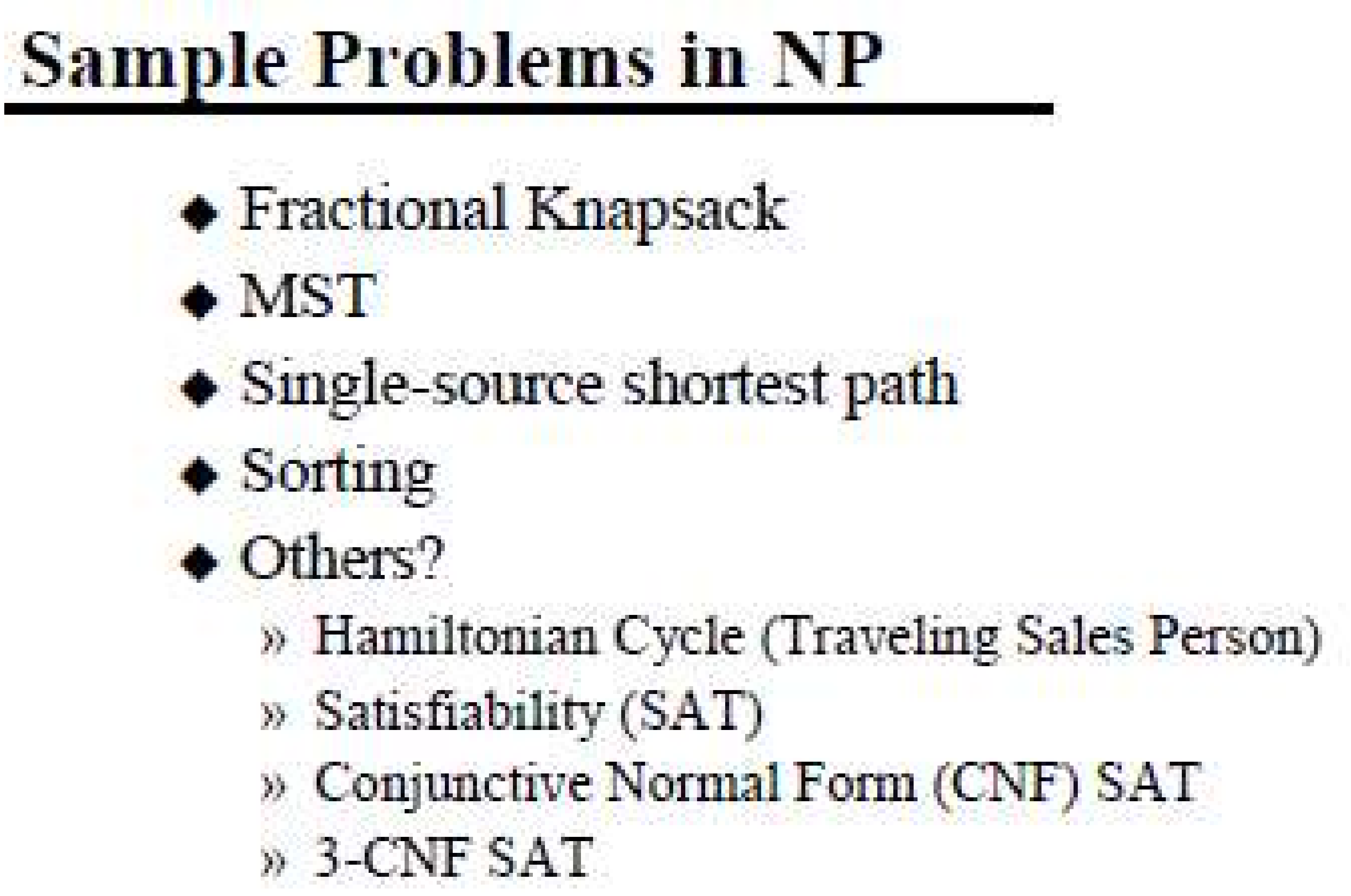
NP: the class of decision problems that are solvable in polynomial time on a *nondeterministic* machine (or with a non-deterministic algorithm)

- (A *determinstic* computer is what we know)
- A *nondeterministic* computer is one that can "guess" the right answer or solution
  - » Think of a nondeterministic computer as a parallel machine that can freely spawn an infinite number of processes
- Thus *NP* can also be thought of as the class of problems
  - » whose solutions can be verified in polynomial time; or
  - » that can be solved in polynomial time on a machine that can pursue infinitely many paths of the computation in parallel
- Note that *NP* stands for "Nondeterministic Polynomial-time"



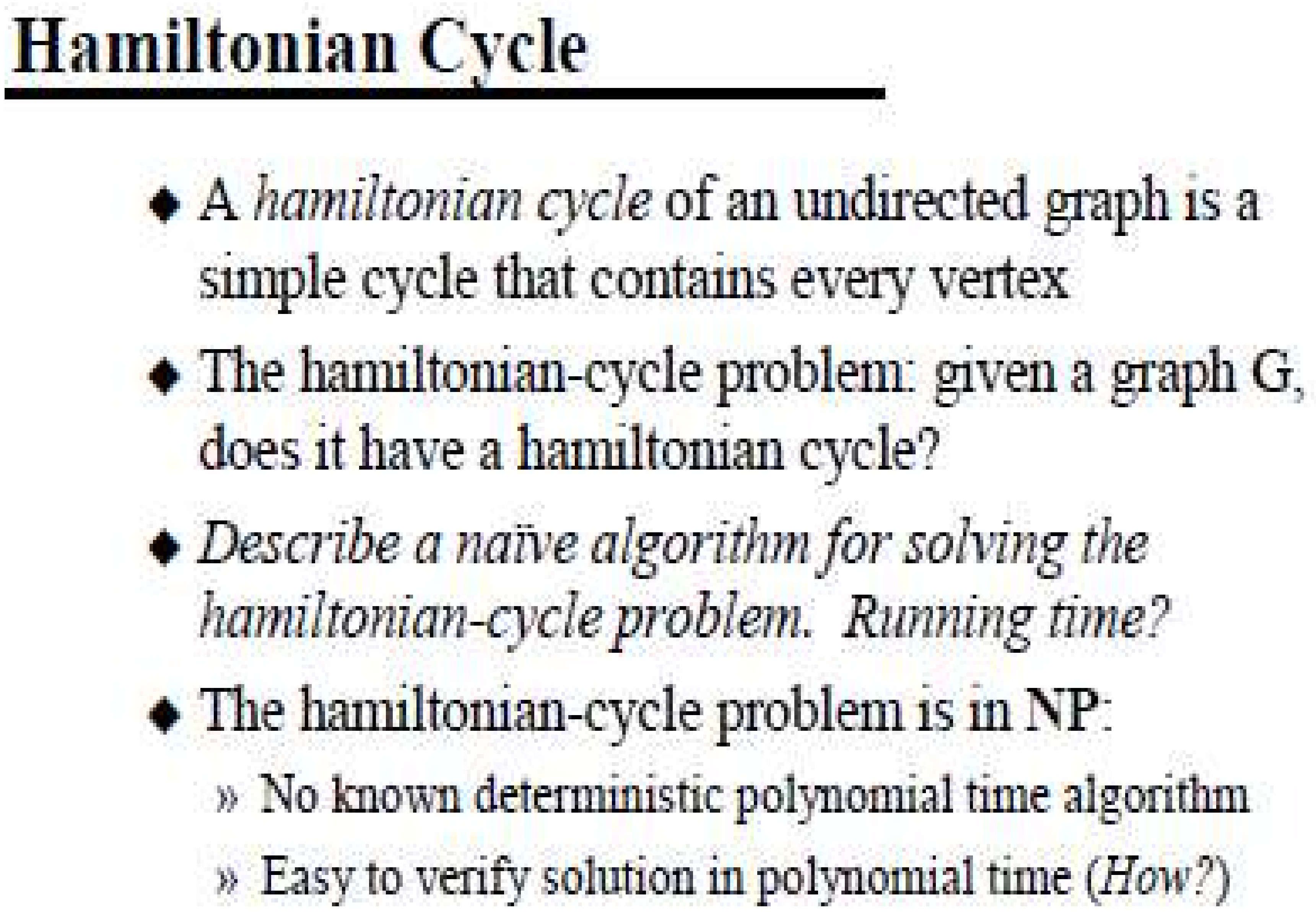
Nondeterminism

- Think of a non-deterministic computer as a computer that magically "guesses" a solution, then has to verify that it is correct
  - » If a solution exists, computer always guesses it
  - » One way to imagine it: a parallel computer that can freely spawn an infinite number of processes
    - Have one processor work on each possible solution
    - All processors attempt to verify that their solution works
    - If a processor finds it has a working solution
  - » So: **NP** = problems *verifiable* in polynomial time



Sample Problems in NP

- Fractional Knapsack
- MST
- Single-source shortest path
- Sorting
- Others?
  - » Hamiltonian Cycle (Traveling Sales Person)
  - » Satisfiability (SAT)
  - » Conjunctive Normal Form (CNF) SAT
  - » 3-CNF SAT

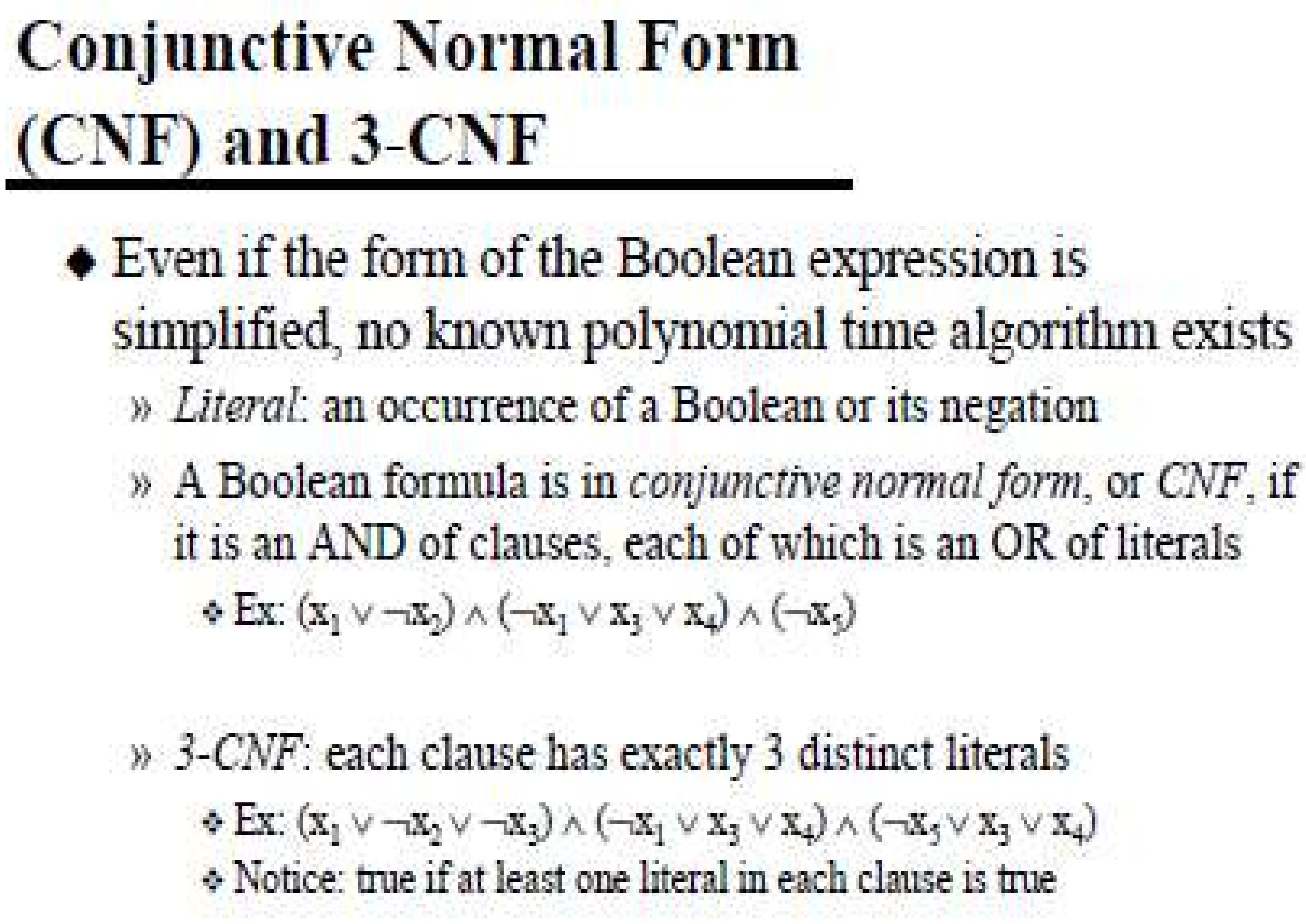


Hamiltonian Cycle

- A *hamiltonian cycle* of an undirected graph is a simple cycle that contains every vertex
- The hamiltonian-cycle problem: given a graph G, does it have a hamiltonian cycle?
- *Describe a naïve algorithm for solving the hamiltonian-cycle problem. Running time?*
- The hamiltonian-cycle problem is in NP:
  - » No known deterministic polynomial time algorithm
  - » Easy to verify solution in polynomial time (*How?*)

The *Satisfiability* (SAT) Problem

- *Satisfiability* (SAT):
  - » Given a Boolean expression on *n* variables, can we assign values such that the expression is TRUE?
  - » Ex: $((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$
  - » Seems simple enough, but no known deterministic polynomial time algorithm exists
  - » Easy to verify in polynomial time!



Conjunctive Normal Form (CNF) and 3-CNF

- Even if the form of the Boolean expression is simplified, no known polynomial time algorithm exists
  - » *Literal*: an occurrence of a Boolean or its negation
  - » A Boolean formula is in *conjunctive normal form*, or *CNF*, if it is an AND of clauses, each of which is an OR of literals
    - Ex: $(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3 \vee x_4) \wedge (\neg x_5)$
  - » *3-CNF*: each clause has exactly 3 distinct literals
    - Ex: $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_3 \vee x_4) \wedge (\neg x_5 \vee x_3 \vee x_4)$
    - Notice: true if at least one literal in each clause is true

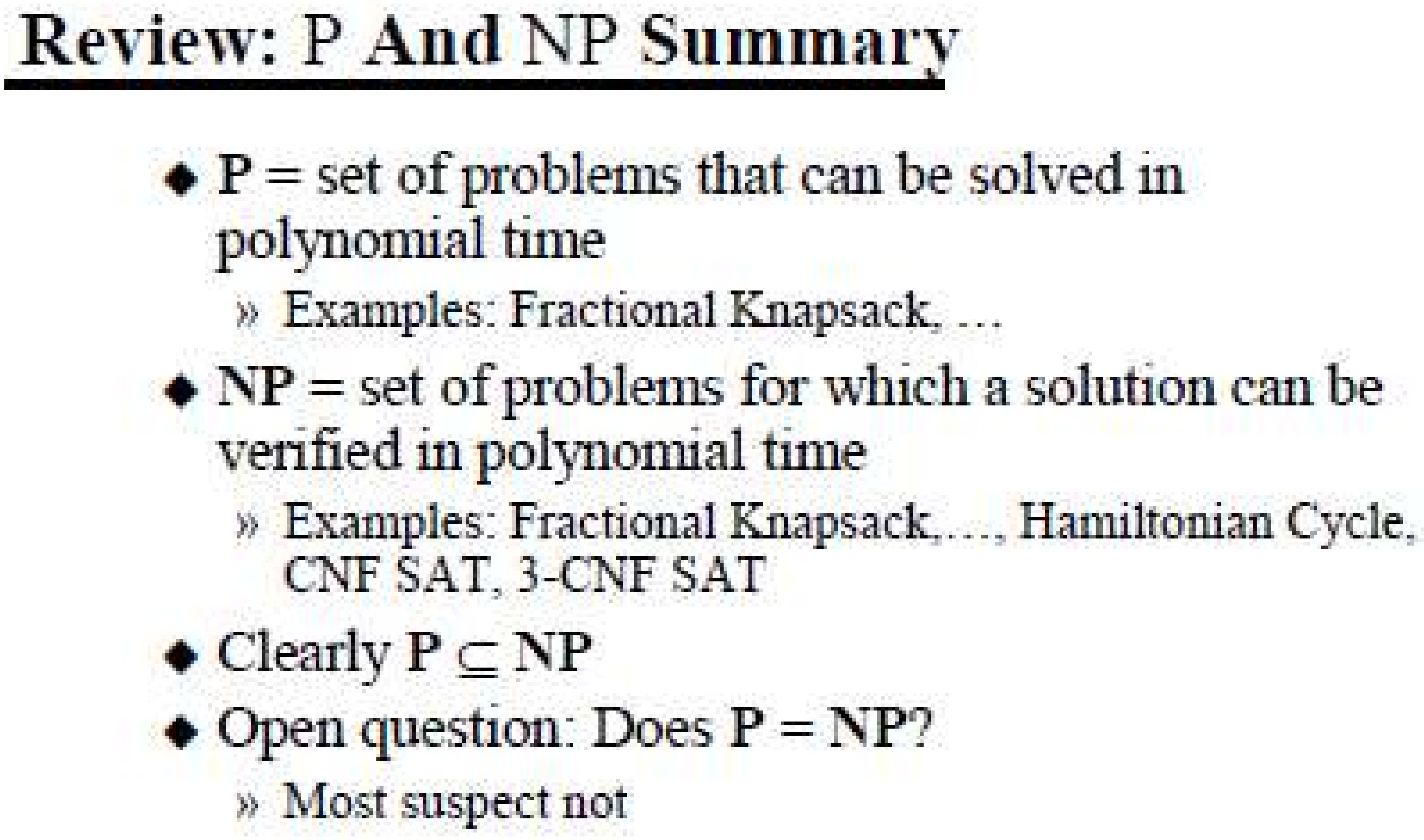
Example: CNF satisfiability

- This problem is in *NP*. Nondeterministic algorithm:
  - » Guess truth assignment
  - » Check assignment to see if it satisfies CNF formula
- Example:

  $(A \vee \neg B \vee \neg C) \wedge (\neg A \vee B) \wedge (\neg B \vee D \vee F) \wedge (F \vee \neg D)$
- Truth assignments:

	A	B	C	D	E	F
1.	0	1	1	0	1	0
2.	1	0	0	0	0	1
3.	1	1	0	0	0	1
4.	... (how many more?)					

Checking phase: $\Theta(n)$



Review: P And NP Summary

- P = set of problems that can be solved in polynomial time
  - » Examples: Fractional Knapsack, …
- NP = set of problems for which a solution can be verified in polynomial time
  - » Examples: Fractional Knapsack,…, Hamiltonian Cycle, CNF SAT, 3-CNF SAT
- Clearly $P \subseteq NP$
- Open question: Does P = NP?
  - » Most suspect not

NP-complete problems

- A decision problem *D* is <u>*NP*-complete</u> iff
  1. $D \in NP$
  2. every problem in *NP* is polynomial-time reducible to *D*

- Cook's theorem (1971): CNF-sat is *NP*-complete

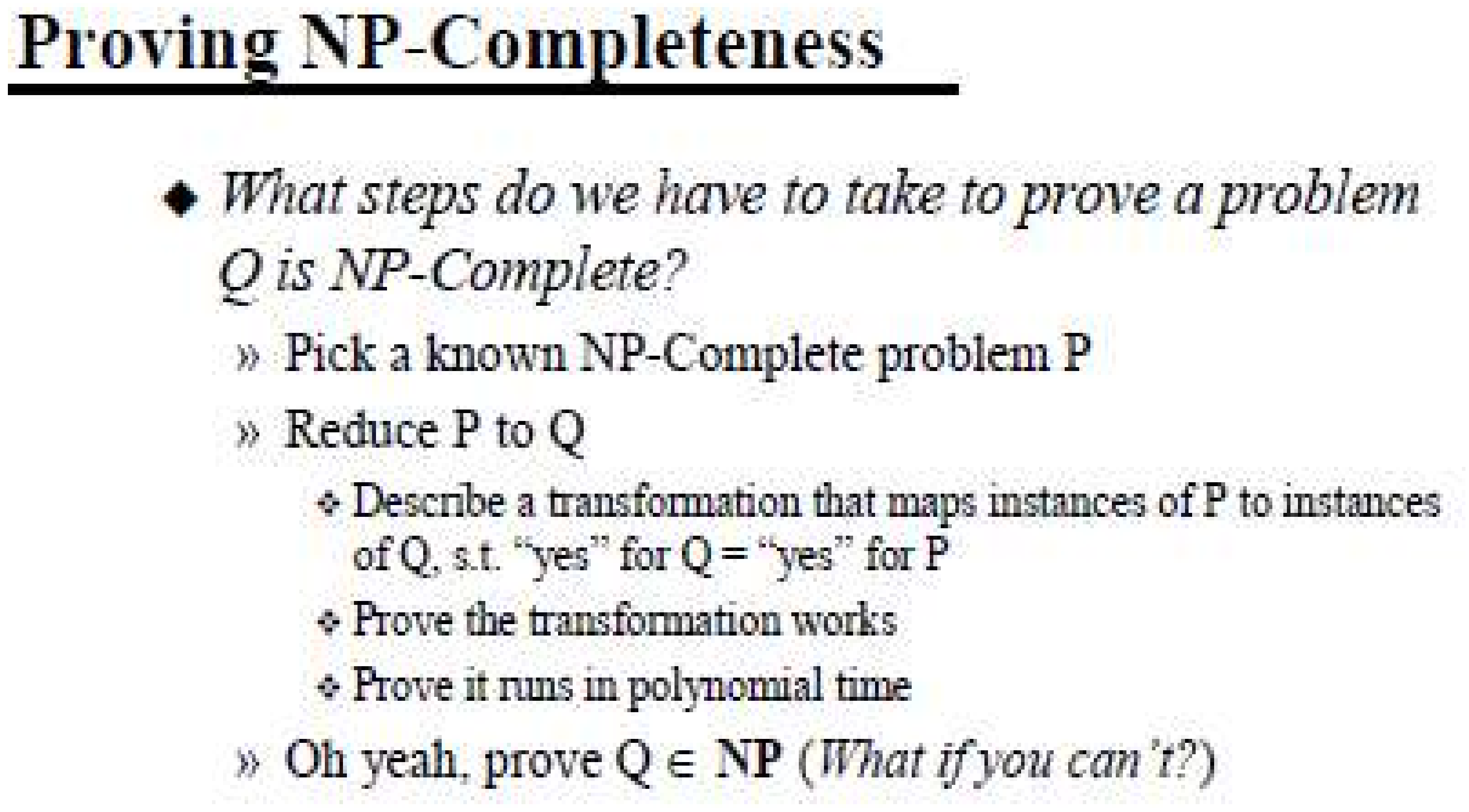
- Other *NP*-complete problems obtained through polynomial-time reductions of known *NP*-complete problems

Review: Reduction

- A problem P can be *reduced* to another problem Q if any instance of P can be rephrased to an instance of Q, the solution to which provides a solution to the instance of P
  - » This rephrasing is called a *transformation*
- Intuitively: If P reduces in polynomial time to Q, P is "no harder to solve" than Q

NP-Hard and NP-Complete

- If P is *polynomial-time reducible* to Q, we denote this $P \leq_p Q$
- Definition of NP-Hard and NP-Complete:
  - » If all problems $R \in NP$ are reducible to P, then P is *NP-Hard*
  - » We say P is *NP-Complete* if P is NP-Hard and $P \in NP$
- If $P \leq_p Q$ and P is NP-Complete, Q is also NP-Complete



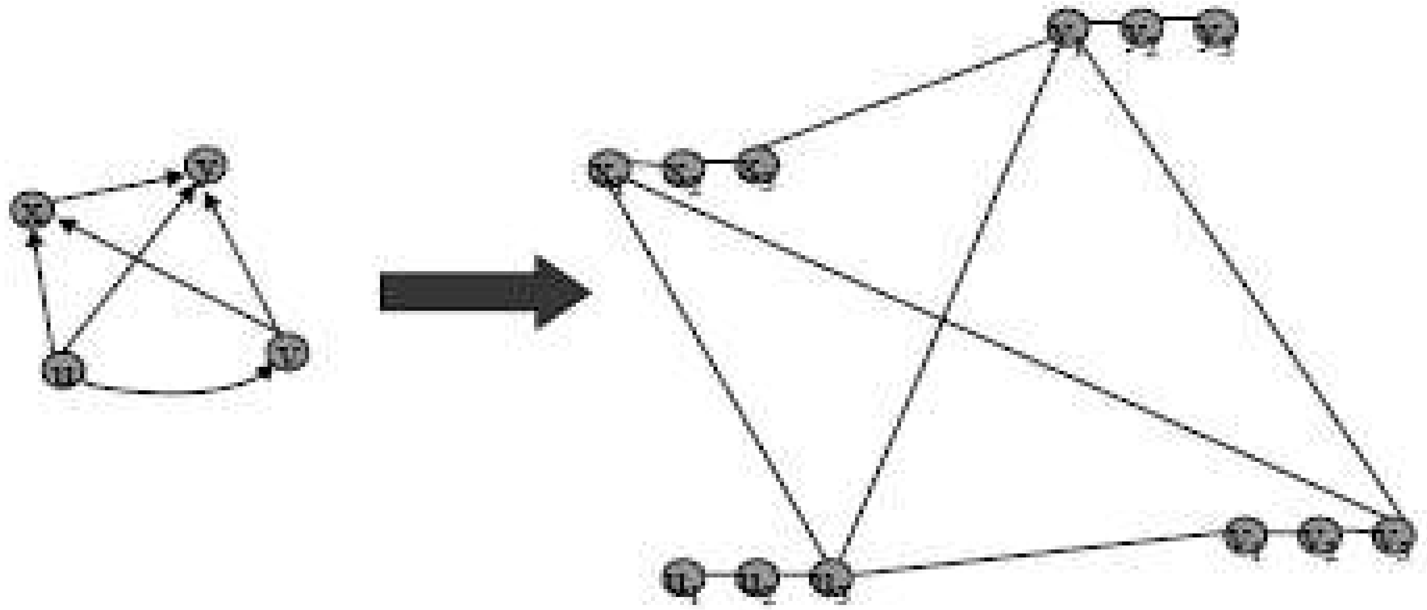
Proving NP-Completeness

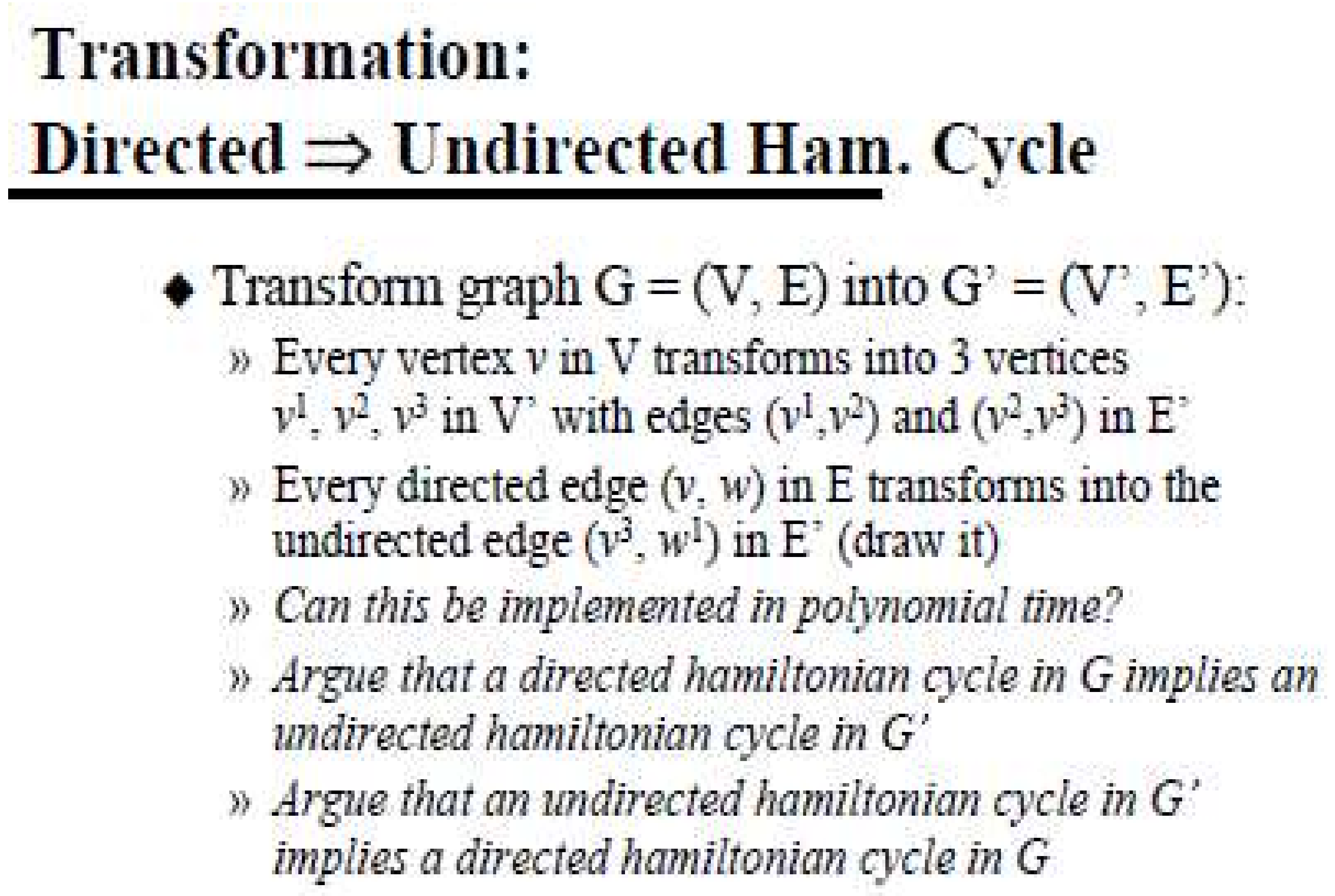
- *What steps do we have to take to prove a problem Q is NP-Complete?*
  - » Pick a known NP-Complete problem P
  - » Reduce P to Q
    - Describe a transformation that maps instances of P to instances of Q, s.t. "yes" for Q = "yes" for P
    - Prove the transformation works
    - Prove it runs in polynomial time
  - » Oh yeah, prove Q ∈ **NP** (*What if you can't?*)

Directed Hamiltonian Cycle ⇒ Undirected Hamiltonian Cycle

- *What was the hamiltonian cycle problem again?*
- For my next trick, I will reduce the *directed hamiltonian cycle* problem to the *undirected hamiltonian cycle* problem before your eyes
  - » *Which variant am I proving NP-Complete?*
- Given a directed graph G,
  - » *What transformation do I need to effect?*

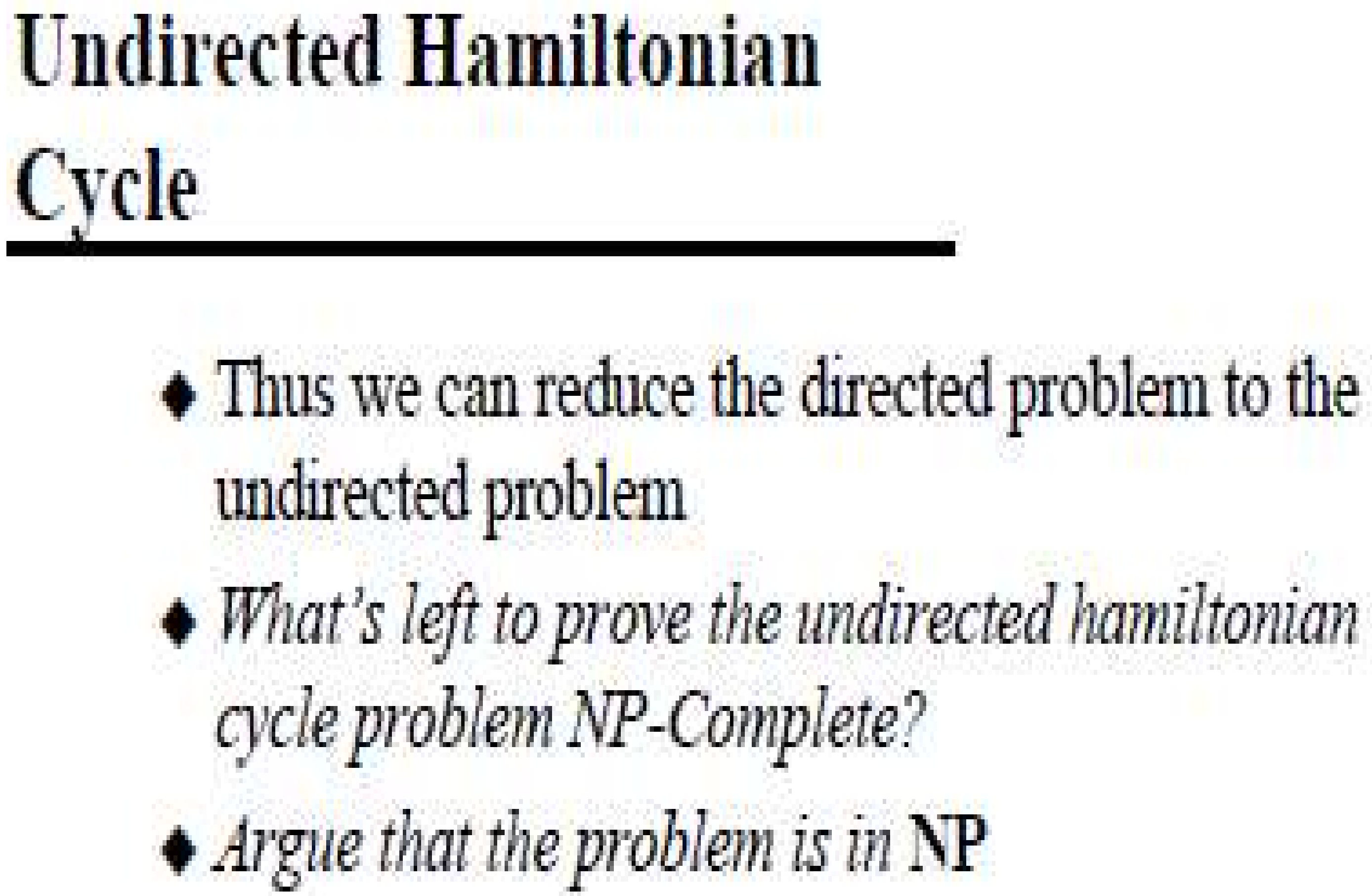
Directed Hamiltonian Cycle $\Rightarrow$ Undirected Hamiltonian Cycle





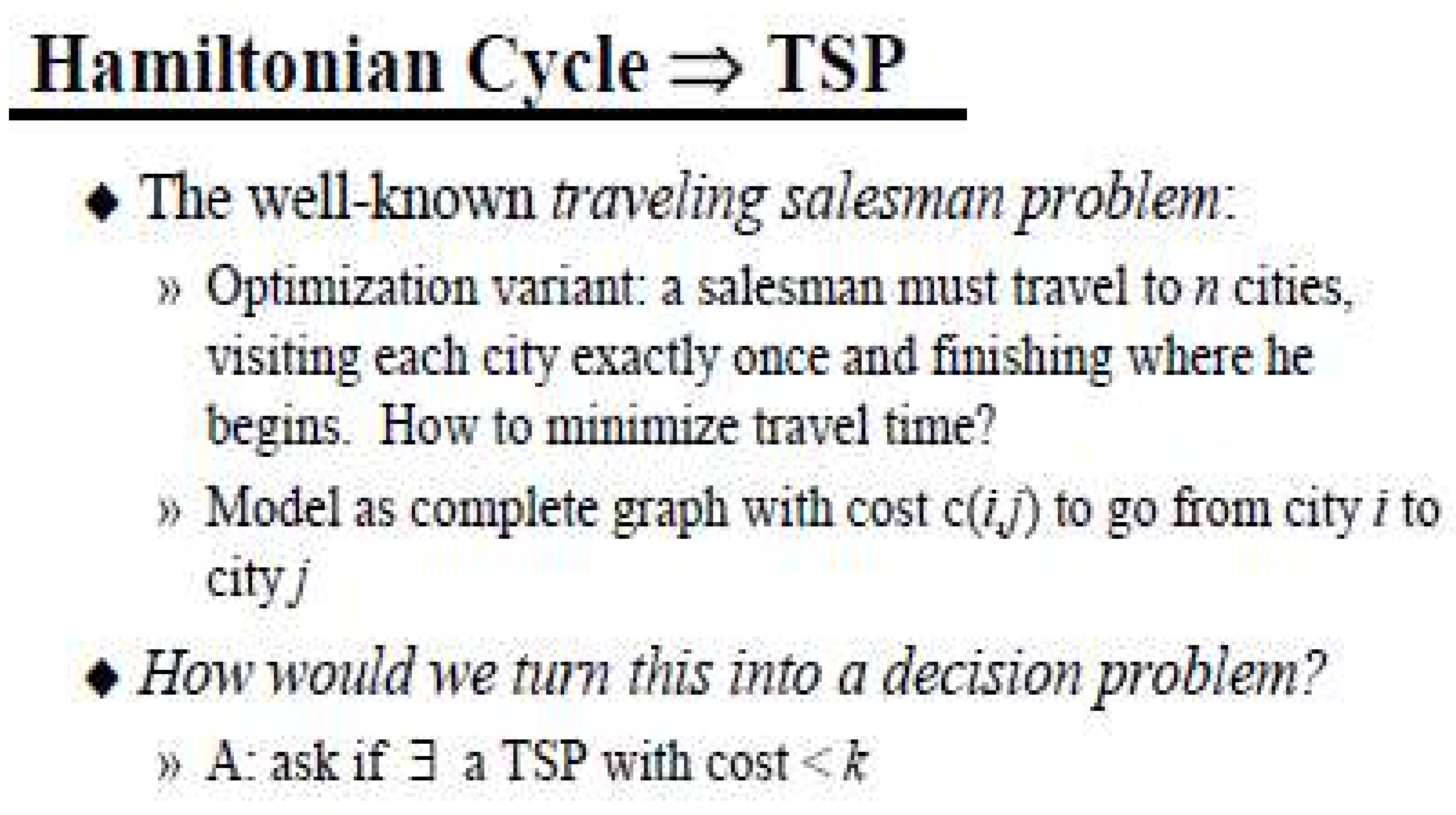
Transformation: Directed ⇒ Undirected Ham. Cycle

- Transform graph G = (V, E) into G' = (V', E'):
  - » Every vertex v in V transforms into 3 vertices v^1, v^2, v^3 in V' with edges (v^1,v^2) and (v^2,v^3) in E'
  - » Every directed edge (v, w) in E transforms into the undirected edge (v^3, w^1) in E' (draw it)
  - » *Can this be implemented in polynomial time?*
  - » *Argue that a directed hamiltonian cycle in G implies an undirected hamiltonian cycle in G'*
  - » *Argue that an undirected hamiltonian cycle in G' implies a directed hamiltonian cycle in G*



Undirected Hamiltonian Cycle

- Thus we can reduce the directed problem to the undirected problem
- *What's left to prove the undirected hamiltonian cycle problem NP-Complete?*
- *Argue that the problem is in* NP

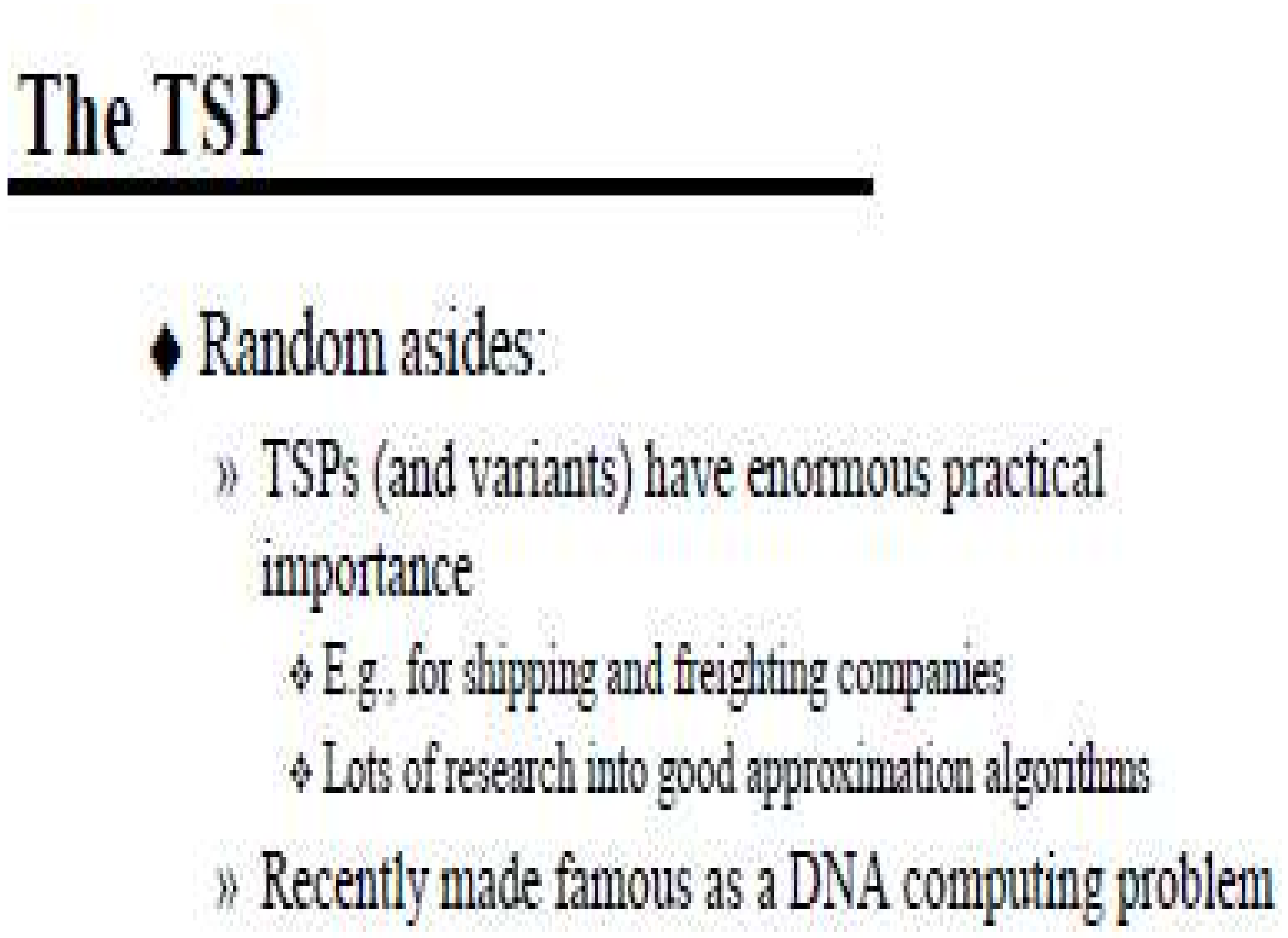


Hamiltonian Cycle ⇒ TSP

- The well-known *traveling salesman problem*:
  - » Optimization variant: a salesman must travel to *n* cities, visiting each city exactly once and finishing where he begins. How to minimize travel time?
  - » Model as complete graph with cost c(*i*,*j*) to go from city *i* to city *j*
- *How would we turn this into a decision problem?*
  - » A: ask if ∃ a TSP with cost < *k*

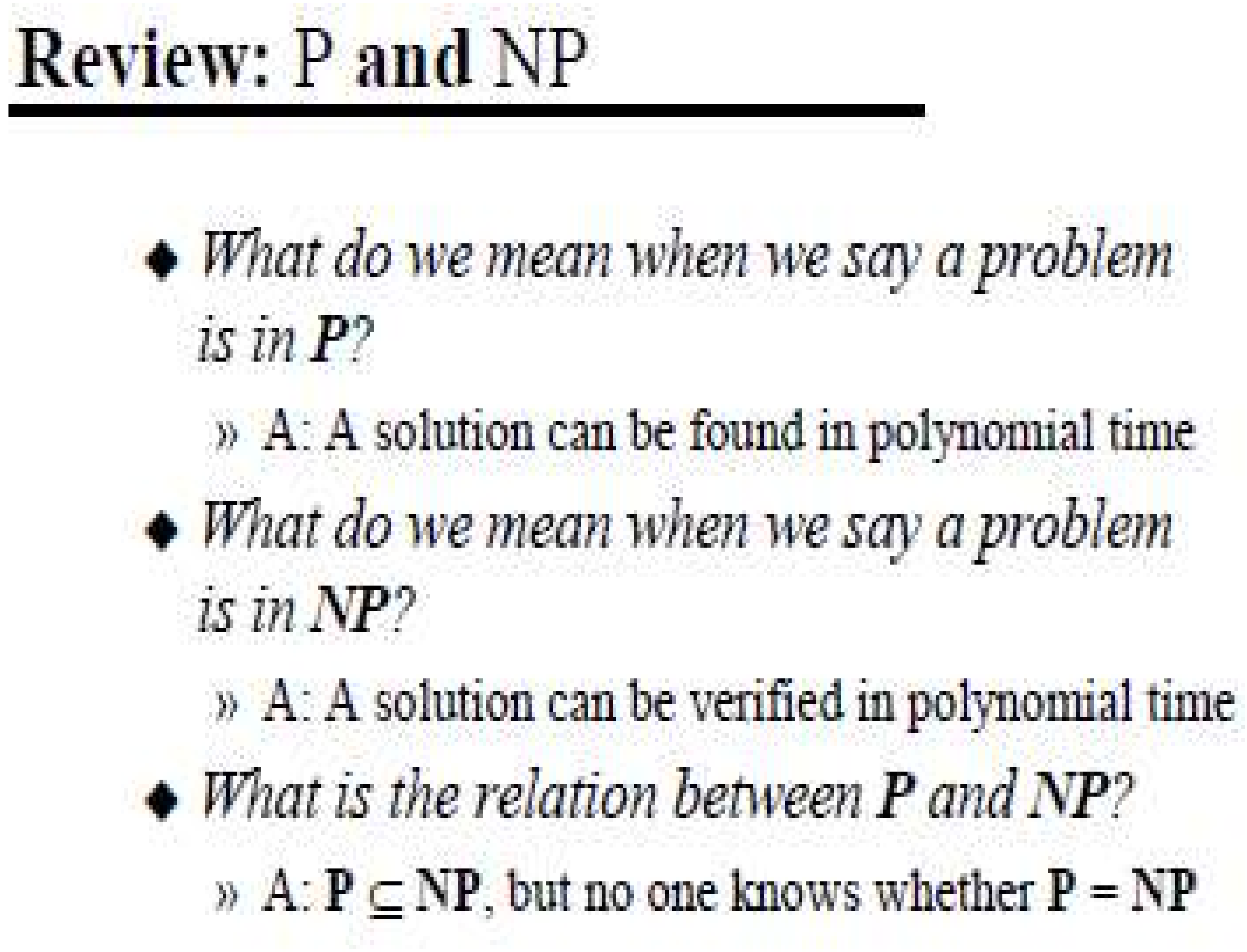
Hamiltonian Cycle ⇒ TSP

- The steps to prove TSP is NP-Complete:
  - » Prove that TSP ∈ **NP** (*Argue this*)
  - » Reduce the undirected hamiltonian cycle problem to the TSP
    - So if we had a TSP-solver, we could use it to solve the hamilitonian cycle problem in polynomial time
    - *How can we transform an instance of the hamiltonian cycle problem to an instance of the TSP?*
    - *Can we do this in polynomial time?*



The TSP

- Random asides:
  - » TSPs (and variants) have enormous practical importance
    - E.g., for shipping and freighting companies
    - Lots of research into good approximation algorithms
  - » Recently made famous as a DNA computing problem



Review: P and NP

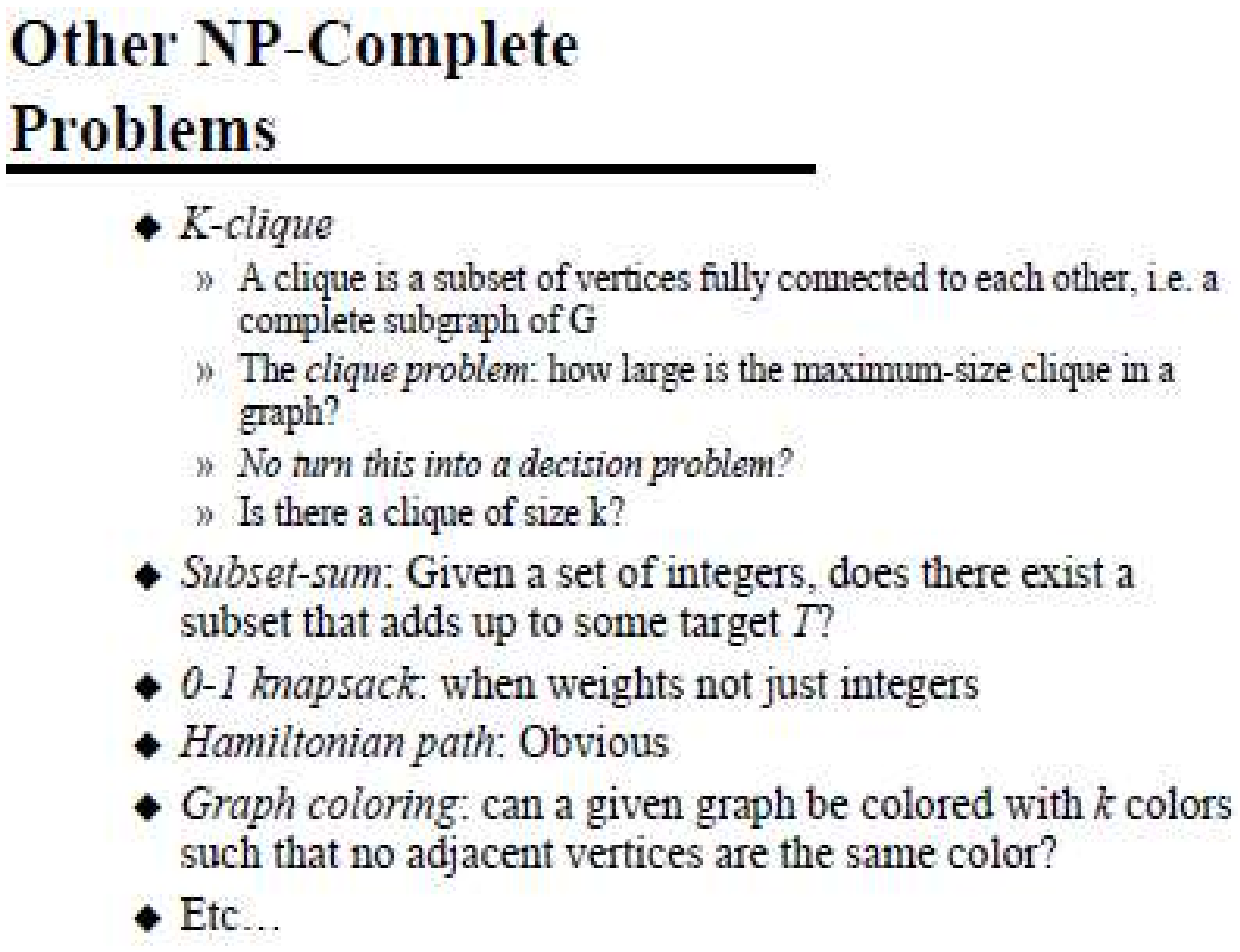
- *What do we mean when we say a problem is in **P**?*
  - » A: A solution can be found in polynomial time
- *What do we mean when we say a problem is in **NP**?*
  - » A: A solution can be verified in polynomial time
- *What is the relation between **P** and **NP**?*
  - » A: **P** $\subseteq$ **NP**, but no one knows whether **P** = **NP**

Review: NP-Complete

- *What, intuitively, does it mean if we can reduce problem P to problem Q?*
  - » P is "no harder than" Q
- *How do we reduce P to Q?*
  - » Transform instances of P to instances of Q in polynomial time s.t. Q: "yes" iff P: "yes"
- *What does it mean if Q is NP-Hard?*
  - » Every problem $P \in \mathbf{NP} \leq_p Q$
- *What does it mean if Q is NP-Complete?*
  - » Q is NP-Hard and $Q \in$ **NP**

Review: Proving Problems NP-Complete

- *How do we usually prove that a problem R is NP-Complete?*
  - » A: Show $R \in$ **NP**, and reduce a known NP-Complete problem Q to R



Other NP-Complete Problems

- *K-clique*
  - » A clique is a subset of vertices fully connected to each other, i.e. a complete subgraph of G
  - » The *clique problem*: how large is the maximum-size clique in a graph?
  - » *No turn this into a decision problem?*
  - » Is there a clique of size k?
- *Subset-sum*: Given a set of integers, does there exist a subset that adds up to some target *T*?
- *0-1 knapsack*: when weights not just integers
- *Hamiltonian path*: Obvious
- *Graph coloring*: can a given graph be colored with *k* colors such that no adjacent vertices are the same color?
- Etc…

General Comments

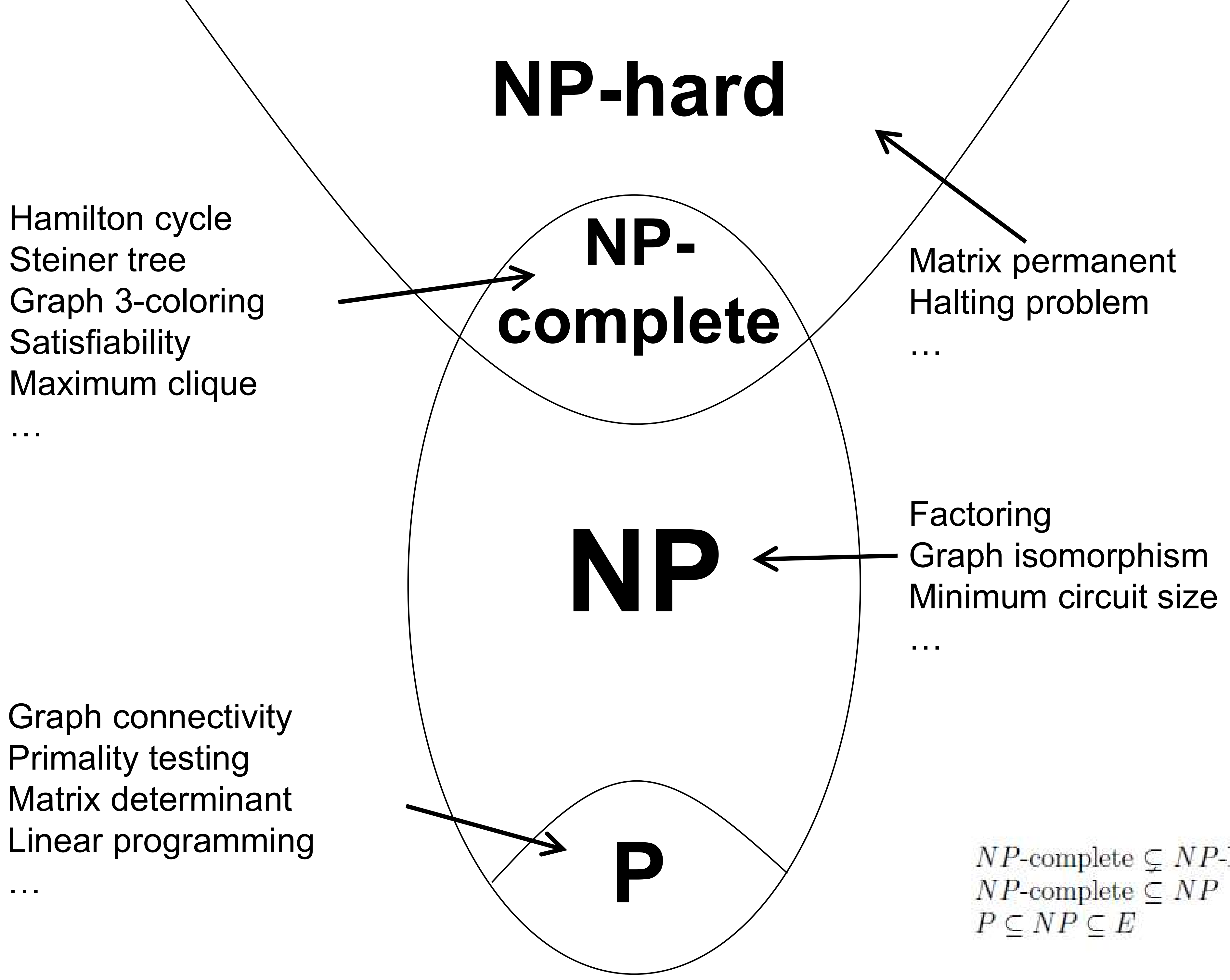
- Literally hundreds of problems have been shown to be NP-Complete
- Some reductions are profound, some are comparatively easy, many are easy once the key insight is given

NP-Hard and NP-Complete

- NP-hard. A problem Y is NP-hard if, for every problem X in NP, $X \leq_p Y$.
- NP-complete. A problem Y is NP-complete, if it is NP-hard and in NP.

- Theorem. Suppose Y is an NP-complete problem. Then Y is solvable in poly-time iff P = NP.
- Pf. $\Leftarrow$ If P = NP then Y can be solved in poly-time since Y is in NP.
- Pf. $\Rightarrow$ Suppose Y can be solved in poly-time.
  - Let X be any problem in NP. Since $X \leq_p Y$, we can solve X in poly-time. This implies NP $\subseteq$ P.
  - We already know P $\subseteq$ NP. Thus P = NP. ▪

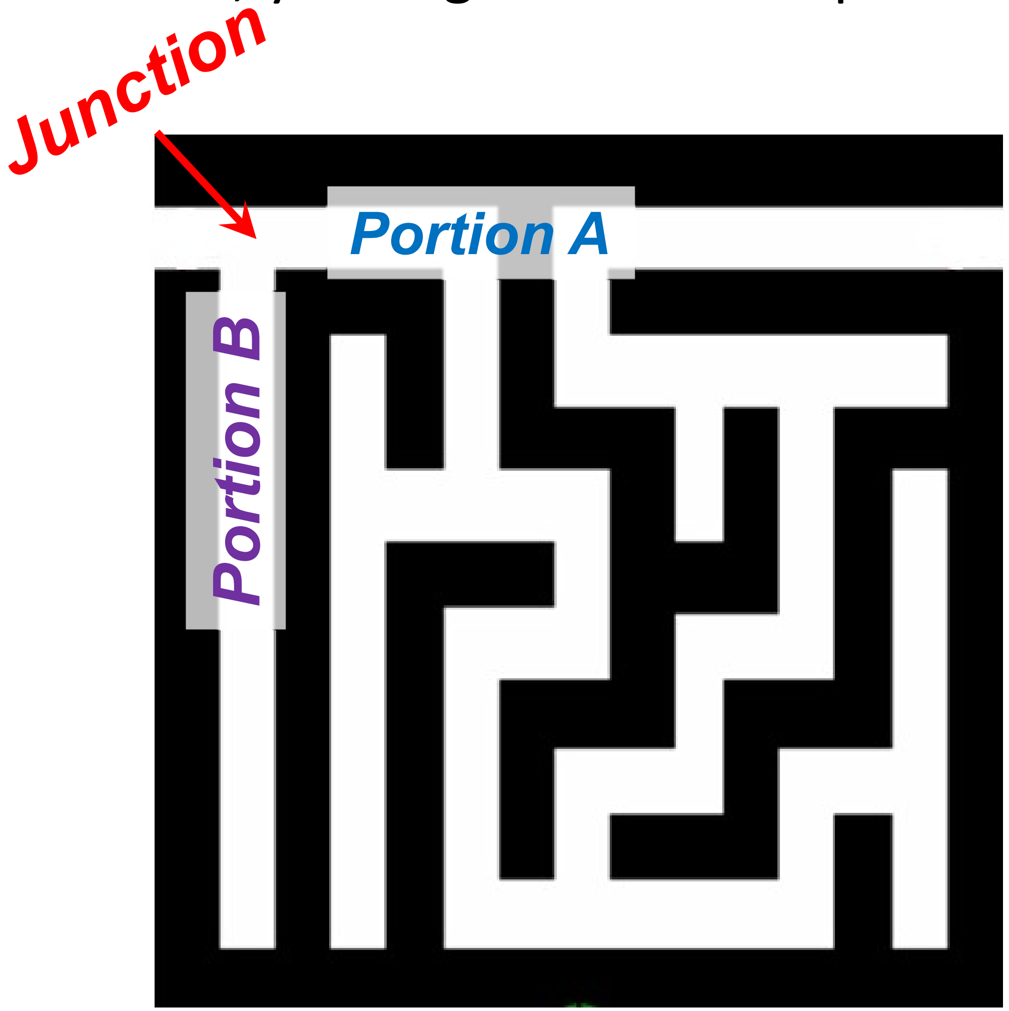
- Fundamental question. Do there exist "natural" NP-complete problems?



NP-complete $\subsetneq$ NP-hard
NP-complete $\subseteq NP$
$P \subseteq NP \subseteq E$

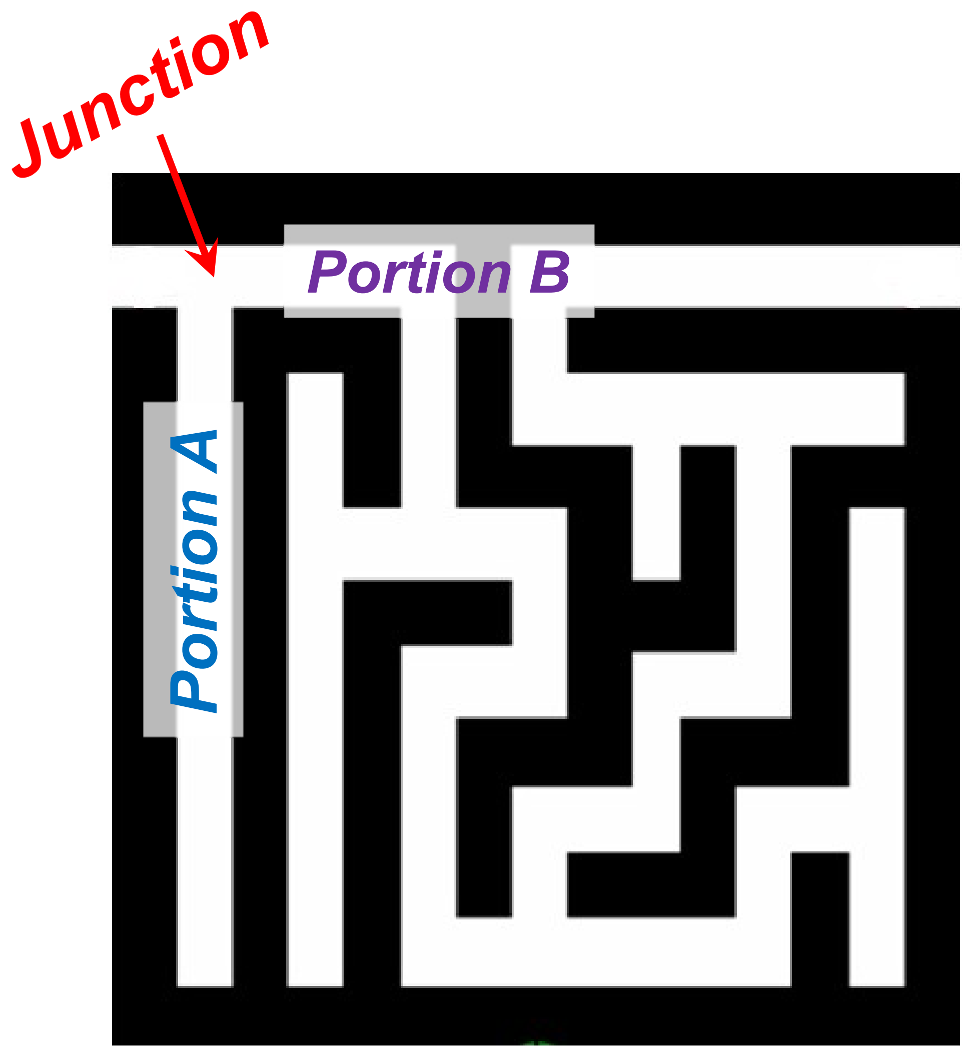
Backtracking

- Backtracking is a technique used to solve problems with a large search space, by systematically trying and eliminating possibilities.
- A standard example of backtracking would be going through a maze.
  - At some point in a maze, you might have two options of which direction to go:



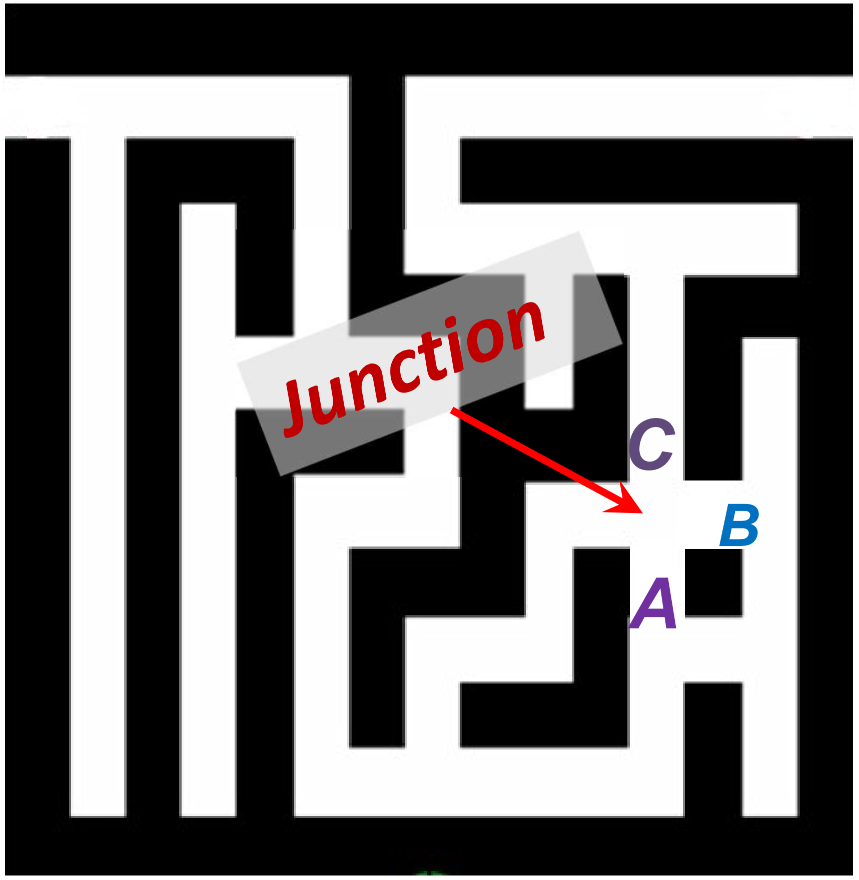
Backtracking

- One strategy would be to try going through **Portion A** of the maze.
  - If you get stuck before you find your way out, then you ***"backtrack"*** to the junction.
- At this point in time you know that **Portion A** will ***NOT*** lead you out of the maze,
  - so you then start searching in **Portion B**



Backtracking

- Clearly, at a single junction you could have even more than 2 choices.
- The backtracking strategy says to try each choice, one after the other,
  - if you ever get stuck, ***"backtrack"*** to the junction and try the next choice.
- If you try all choices and never found a way out, then there IS no solution to the maze.



Backtracking

❑ ***Backtracking*** is the problem-solving method in which the solution space is scanned, but with the additional condition that only the *possible* candidate solutions are considered.

❑ **Backtracking** is a general algorithm for finding all (or some)solutions to some computational problem that incrementally builds candidates to the solutions, and abandons each partial candidate *c* ("backtracks") as soon as it determines that *c* cannot possibly be completed to a valid solution.

❑ Suppose you have to make a series of *decisions,* among various *choices,* where
  - You don’t have enough information to know what to choose
  - Each decision leads to a new set of choices
  - Some sequence of choices (possibly more than one) may be a solution to your problem

-Backtracking is a methodical way of trying out various sequences of decisions, until you find one that “works”

The n-Queens Problem

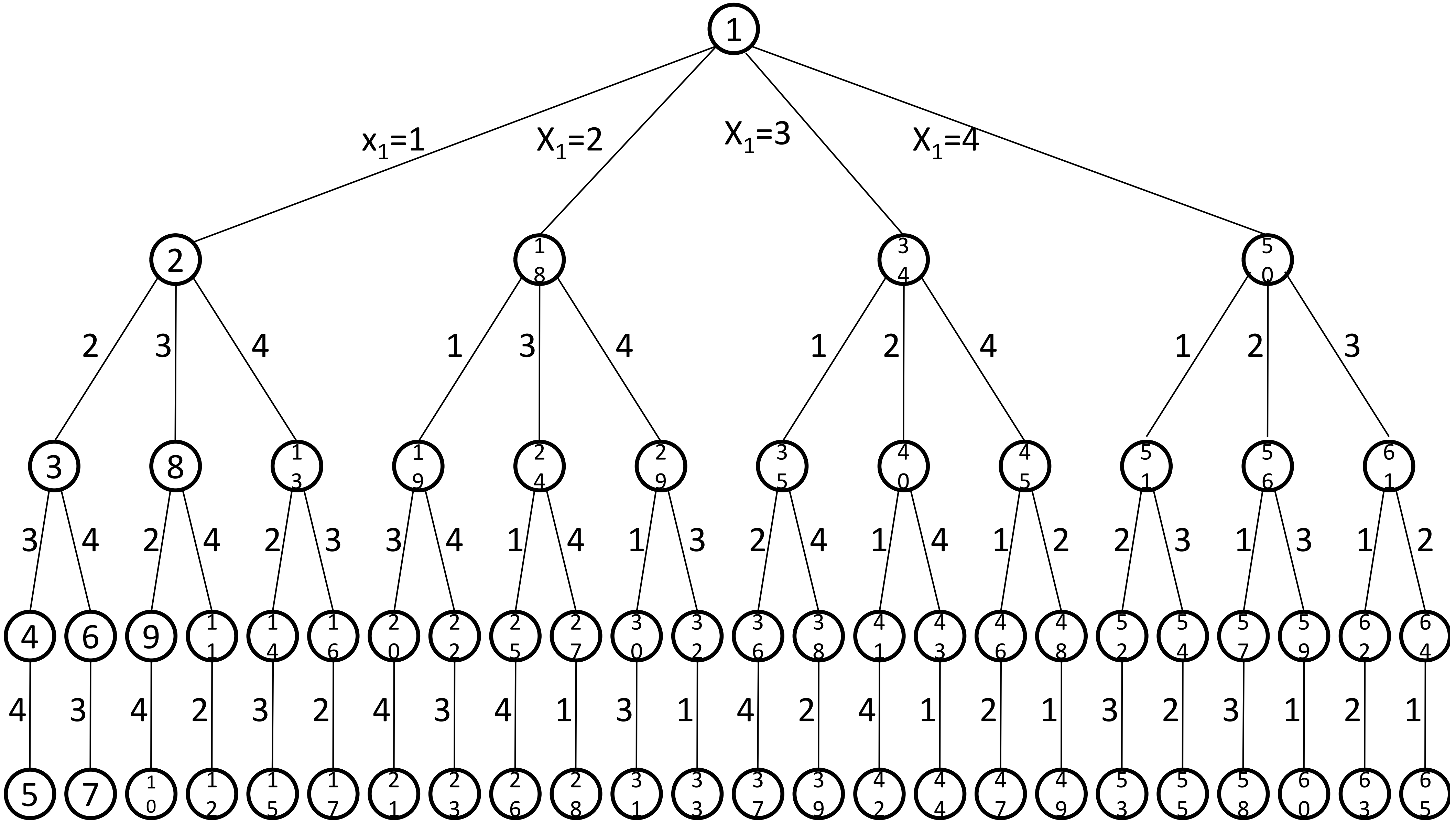
- The problem is to place n queens (chess pieces) on an n by n board so that no two queens are in the same row, column or diagonal

x			x			x	
	x		x		x		
		x	x	x			
x	x	x	Q	x	x	x	x
		x	x	x			
	x		x		x		
x			x			x	
			x				x

		Q	
Q			
			Q
	Q		

- The 4 by 4 board above shows a solution for n = 4
- The text gives an explicit construction for a solution for each n $\geq$ 4
- But first we will introduce an algorithm strategy called ***backtracking***, which can be used to construct ***all*** solutions for a given n.

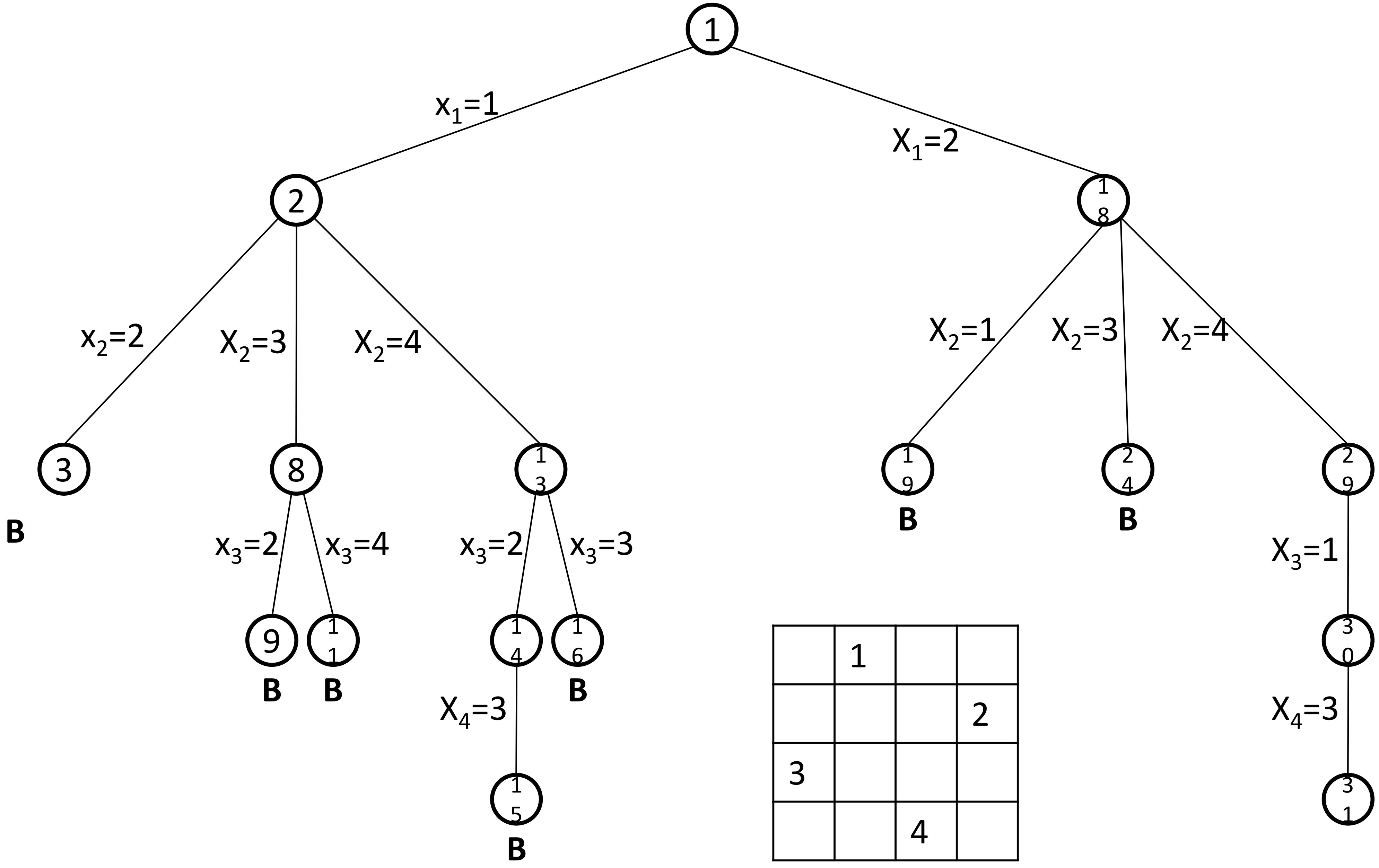
4-queen Solution Space



Backtracking Solution to n-Queens Problem

- We can solve this problem by generating a dynamic tree in a depth-first manner
- We will try to place a queen in each column from the first to the last
- For each column, we must select a row
- We start by placing a queen in row 1 of the first column
- Then we check the rows in the second column for positions that do not conflict with our choice for the first column
- After placing a queen in the second column, we continue to the third, etc.
- If at any point we find we cannot place a queen in the current column, we back up to the previous column and try a different row for that column
- We then continue trying to place more queens
- This process can be visualized by a tree of configurations
- The nodes are partial solutions of the problem
- The root is an empty board
- The children of the root will be boards with placements in the first column
- We will generate the nodes of the tree dynamically in a depth-first way

4-queen Solution during backtracking



	1		
			2
3			
		4	

Backtracking Solution to n-Queens Problem

- We could continue with the previous example to obtain all possible solutions for n = 4
- Our goal now is to convert this approach into an explicit algorithm with n queens.
- Consider an nxn chessboard and try to find all ways to place n nonattacking queens.
- We track the position of the queens by using an array row
- We observed from the 4-queens problem that we can let $(x_1, . . . , x_n)$ represent a solution in which x_i is the column of the ith row where the ith queen is placed.
- The x_i's will all be distinct since no two queens can be placed in the same column.
- Now how do we test whether two queens are on the same diagonal?

Backtracking Solution to n-Queens Problem

- **Test whether two queens are on the same diagonal?**
- Imagine the chessboard squares being numbered as the indices of the two-dimensional array a[1:n, 1:n].
- Observe that every element on the same diagonal that runs from the upper left to the lower right has the same (*row-column)* value.
- Every element on the same diagonal that goes from the upper right to the lower left has the same (*row+column)* value.
- Suppose two queens are placed at position (i,j) and (k,l). Then they are on the the same diagonal only if

  i-j=k-l or i+j=k+l that is j-l=i-k or j-l=k-l
- Therefore, two queens lie on the same diagonal in and only if

  |j-l|=|i-k|
- The algorithm is invocked by Nqueen(1,n).

N-Queen Algorithm

```
Algorithm Place(k,i)
/*Returns true if a queen can be placed in kth row and  ith column. Otherwise it returns false.*/
{
    for j:=1 to k-1 do
        if((x[j]==i) or (abs(x[j]-i)=abs(j-k)) //Two same column or Same diagonal
            then return false
    return true.
}
```

```
Algorithm NQueens(k,n)
//Using backtracking, this procedure prints all possible placements of n queens
{
    for i:=1 to n do
        {
            if(place(k,i))then
            {
            x[k]:=i;
            if(k=n) then write(x[1:n]);
            else NQueens(k+1,n);
            }
        }
}
```

Running Time

- Improvements can be made so that **Place(k,i)** runs in *O*(1) time rather than ***O***(k) time.
- We will will obtain an upper bound for the running time of our algorithm by bounding the number of times NQueens(k,n) is called for each k<n
  - **k = 1**: **1** time, by n_queens
  - **1 < k < n**: at most **n(n-1)⋯(n-k+2)**, since there are n(n-1)⋯(n-k+2) ways to place queens in the first k-1 columns in distinct rows
- Ignoring recursive calls, NQueens(k,n) executes in Θ(n) time for k < n.
- This gives a worst-case bound for NQueens(k,n) of

  n [n(n-1)⋯(n-k+2)] for 1 < k < n
- For k = n, there is at most one placement possible for the queen. Also, the loop in NQueens executes in Θ(n) time.
- There are n(n-1)⋯2 ways for the queens to have been placed in the first n-1 columns, so the worst case time for NQueens(n,n) is

  n [n(n-1)⋯2]

Running Time

- Combining the previous bounds, we get the bound

  n [1 + n + n(n-1) + ⋯ + n(n-1) ⋯2]
  = n·n! [1/n! + 1/(n-1)! + ⋯ + 1/1!]

- A result from calculus: $$e = \sum_{i=0}^{\infty} \frac{1}{i!} = 1 + \sum_{i=1}^{\infty} \frac{1}{i!}$$

- This means that **n·n! [1/n! + 1/(n-1)! + ⋯ + 1/1!] ≤ n·n! (e-1)**

- We thus have that our algorithm runs in ***O*(n·n!) time**

Branch-and-Bound

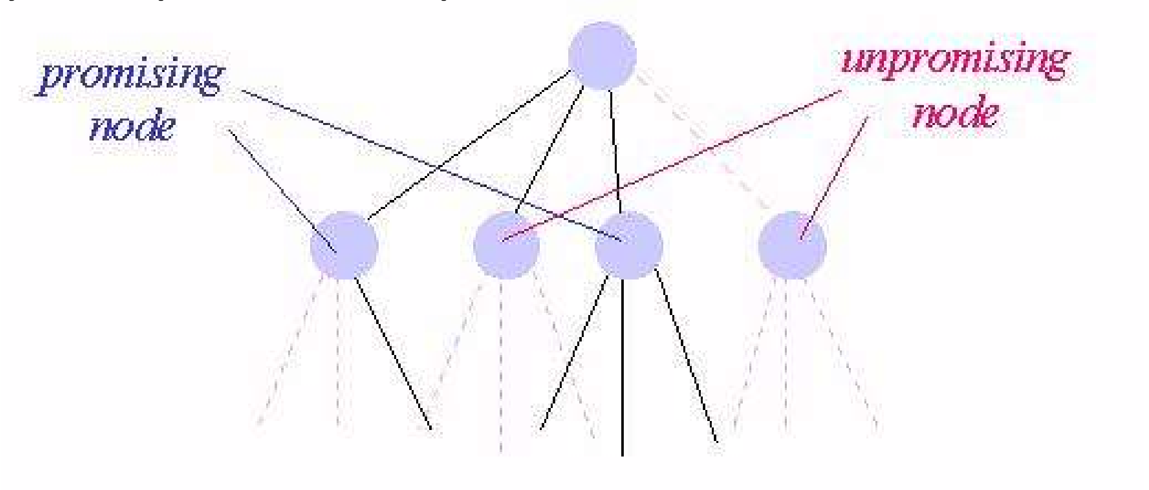
- Effective for optimization problems
- Extended Backtracking Algorithm
- Instead of stopping once a single solution is found, continue searching until the best solution is found
- Has a scoring mechanism to choose most promising configuration in each iteration

Branch-and-Bound

The branch-and-bound problem solving method is very similar to backtracking in that a state space tree is used to solve a problem. The differences are that B&B

(1) does not limit us to any particular way of traversing the tree and

(2) is used only for optimization problems.



A B&B algorithm computes a number (bound) at a node to determine whether the node is promising. The number is a bound on the value of the solution that could be obtained by expanding the state space tree beyond the current node.

Branch-and-Bound

Types of Traversal

When implementing the branch-and-bound approach there is no restriction on the type of state-space tree traversal used. Backtracking, for example, is a simple type of B&B that uses depth-first search.

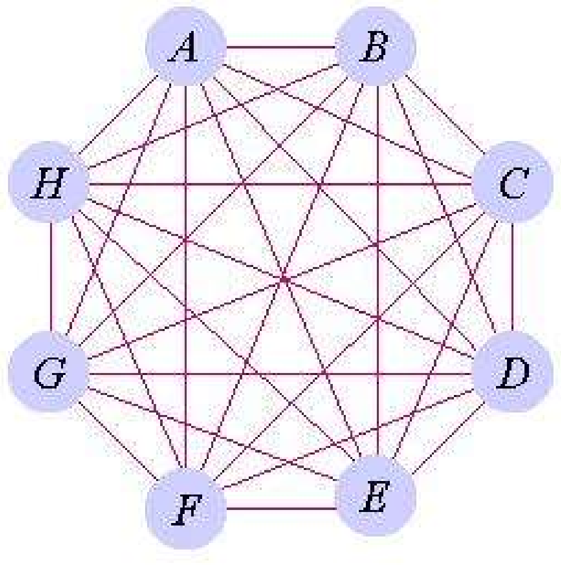
A better approach is to check all the nodes reachable from the currently active node (breadth-first) and then to choose the most promising node (best-first) to expand next.

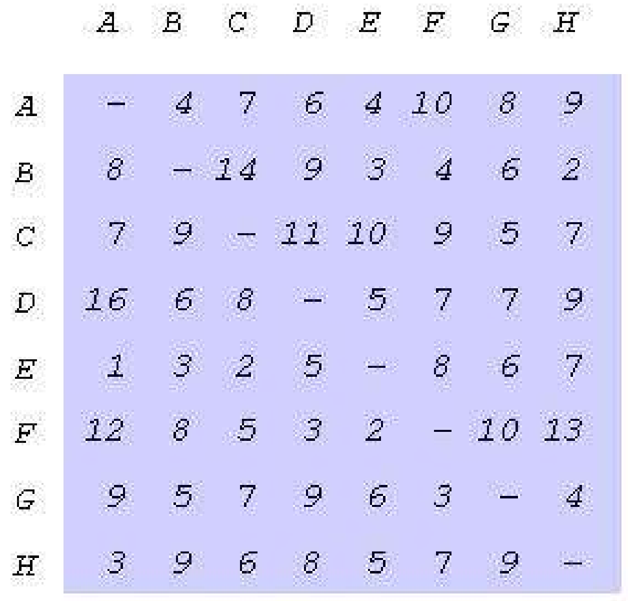
An essential element of B&B is a greedy way to estimate the value of choosing one node over another. This is called the bound or bounding heuristic. It is an underestimate for a minimal search and an overestimate for a maximal search.

When performing a breadth-first traversal, the bound is used only to prune the unpromising nodes from the state-space tree. In a best-first traversal the bound cab also used to order the promising nodes on the live-node list.

Traveling Salesperson & B&B

In the most general case the distances between each pair of cities is a positive value with dist(A,B) /= dist(B,A). In the matrix representation, the main diagonal values are omitted (i.e. dist(A,A) = 0).

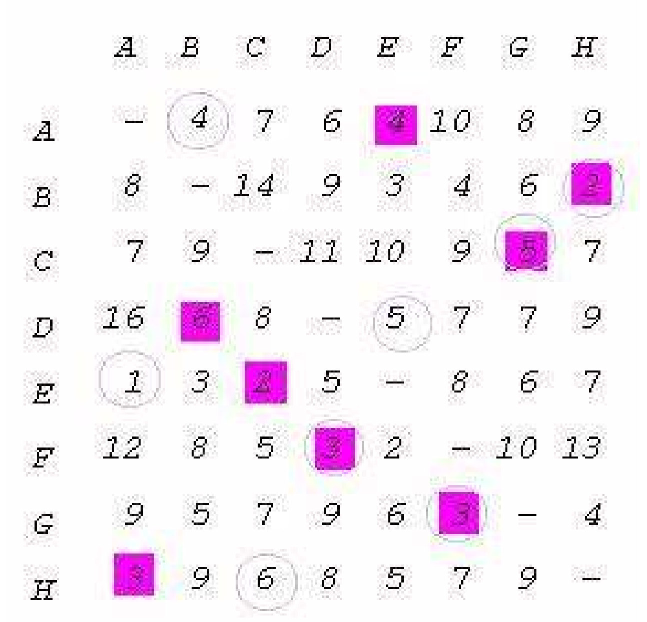




	A	B	C	D	E	F	G	H
A	–	4	7	6	4	10	8	9
B	8	–	14	9	3	4	6	2
C	7	9	–	11	10	9	5	7
D	16	6	8	–	5	7	7	9
E	1	3	2	5	–	8	6	7
F	12	8	5	3	2	–	10	13
G	9	5	7	9	6	3	–	4
H	3	9	6	8	5	7	9	–

Traveling Salesperson & B&B

Obtaining an Initial Tour Use the greedy method to find an initial candidate tour. We start with city A (arbitrary) and choose the closest city (in this case E). Moving to the newly chosen city, we always choose the closest city that has not yet been chosen until we return to A.



	A	B	C	D	E	F	G	H
A	–	4	7	6	4	10	8	9
B	8	–	14	9	3	4	6	2
C	7	9	–	11	10	9	5	7
D	16	6	8	–	5	7	7	9
E	1	3	2	5	–	8	6	7
F	12	8	5	3	2	–	10	13
G	9	5	7	9	6	3	–	4
H	3	9	6	8	5	7	9	–

Our candidate tour is

A-E-C-G-F-D-B-H-A

with a tour length of 28. We know that a minimal tour will be no greater than this.

It is important to understand that the initial candidate tour is not necessarily a minimal tour nor is it unique.

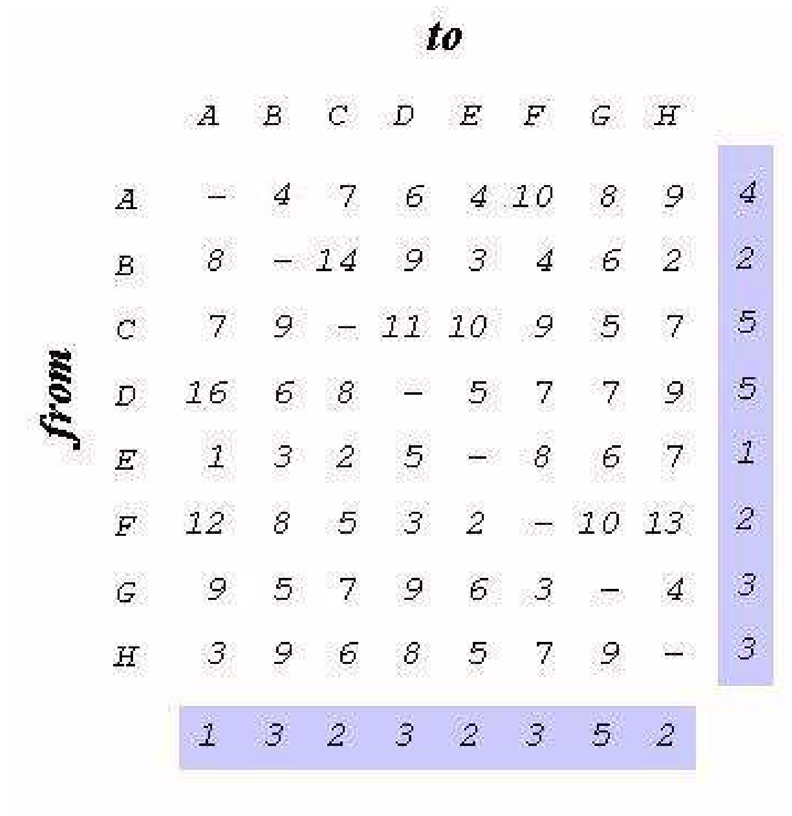
If we start with city E for example, we have,

E-A-B-H-C-G-F-D-E

with a length of 30

Traveling Salesperson & B&B

Defining a Bounding Heuristic



to

from	A	B	C	D	E	F	G	H	
A	–	4	7	6	4	10	8	9	4
B	8	–	14	9	3	4	6	2	2
C	7	9	–	11	10	9	5	7	5
D	16	6	8	–	5	7	7	9	5
E	1	3	2	5	–	8	6	7	1
F	12	8	5	3	2	–	10	13	2
G	9	5	7	9	6	3	–	4	3
H	3	9	6	8	5	7	9	–	3
	1	3	2	3	2	3	5	2	

Now we must define a bounding heuristic that provides an underestimate of the cost to complete a tour from any node using local information. In this example we choose to use the actual cost to reach a node plus the minimum cost from every remaining node as our bounding heuristic.

$$h(x) = a(x) + g(x)$$

a(x) - actual cost to node x
g(x) - minimum cost to complete

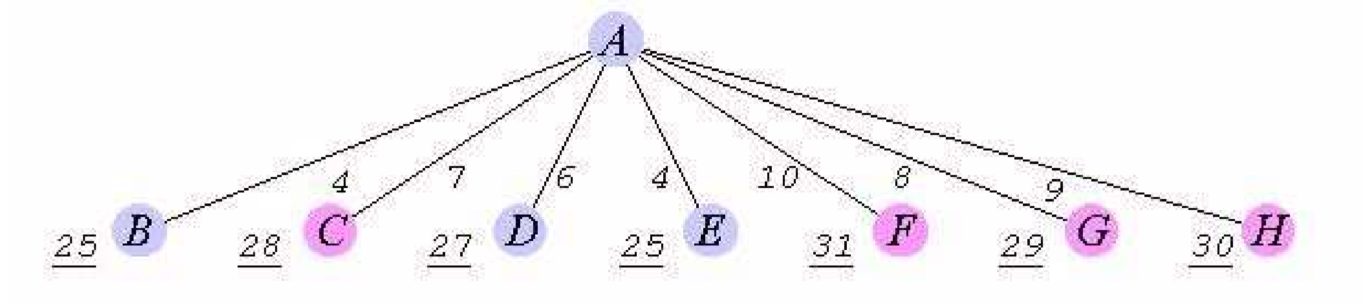
Since we know the minimum cost from each node to anywhere we know that the minimal tour cannot be less than 25.

Finding the Minimal Tour

1. Starting with node A determine the actual cost to each of the other nodes.

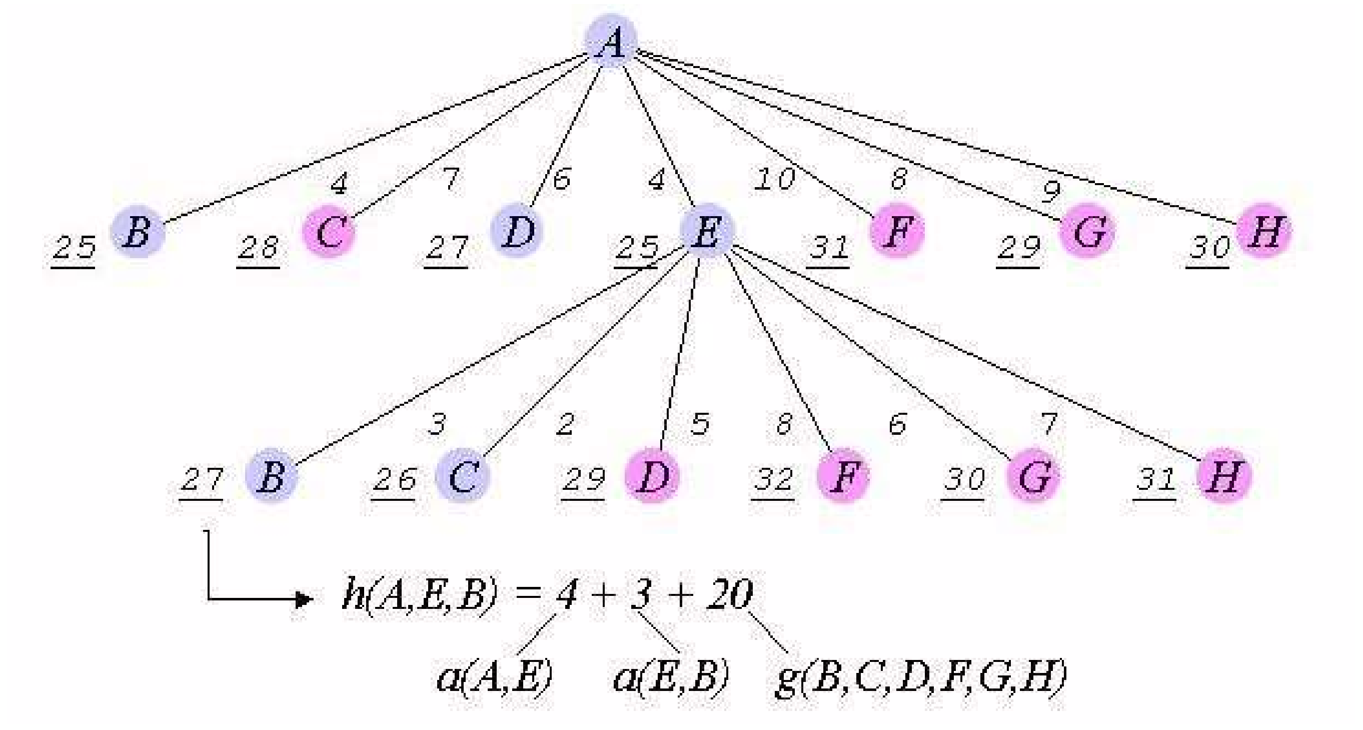
2. Now compute the minimum cost from every other node 25-4=21.

3. Add the actual cost to a node a(x) to the minimum cost from every other node to determine a lower bound for tours from A through B,C,. . ., H.

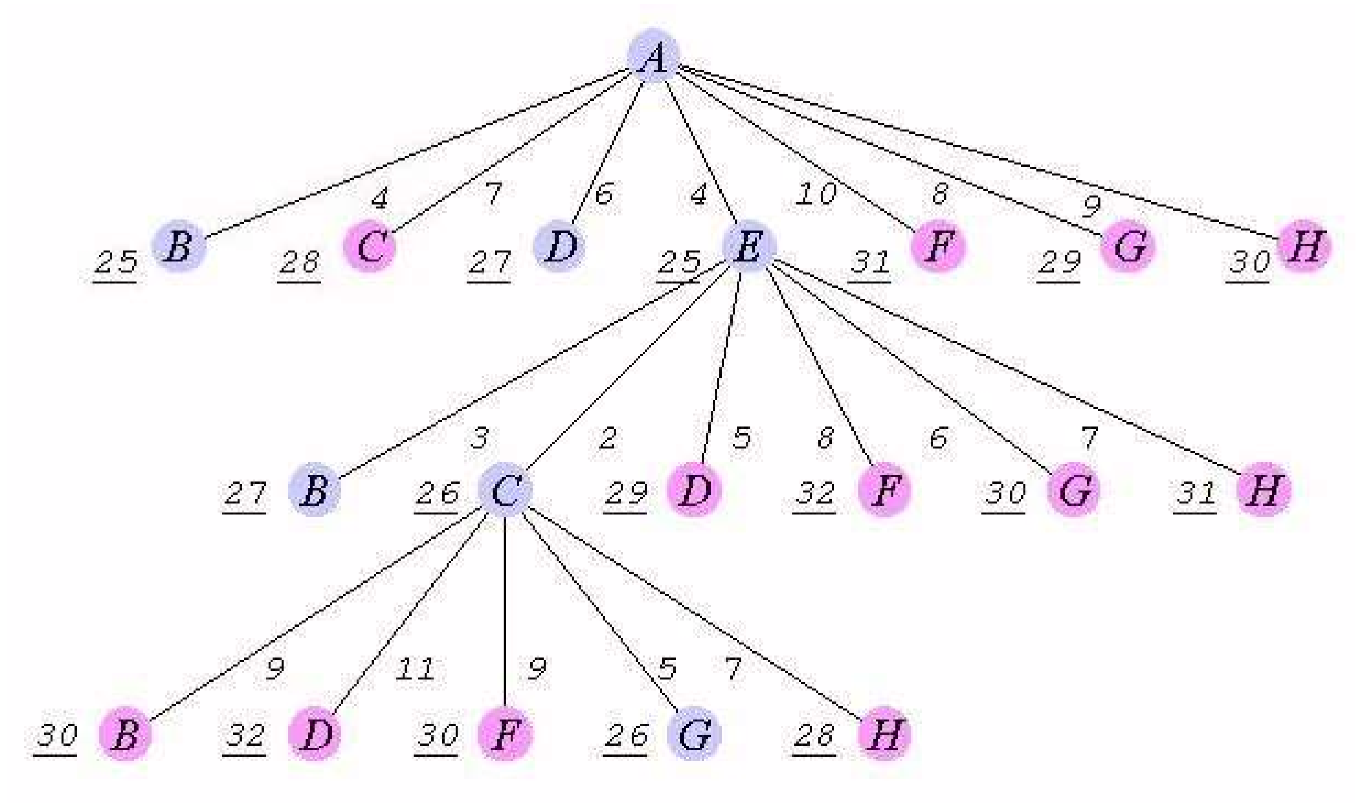


4. Since we already have a candidate tour of 28 we can prune all branches with a lower-bound that is ? 28. This leaves B, D and E as the only promising nodes.
5. We continue to expand the promising nodes in a best-first order (E1,B1,,D1).

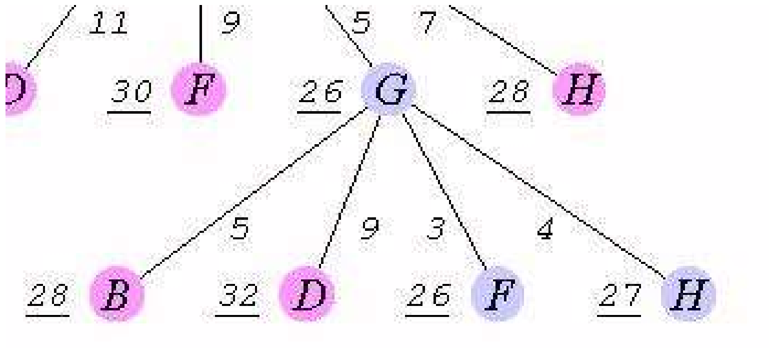
5. We continue to expand the promising nodes in a best-first order (E1,B1,,D1).



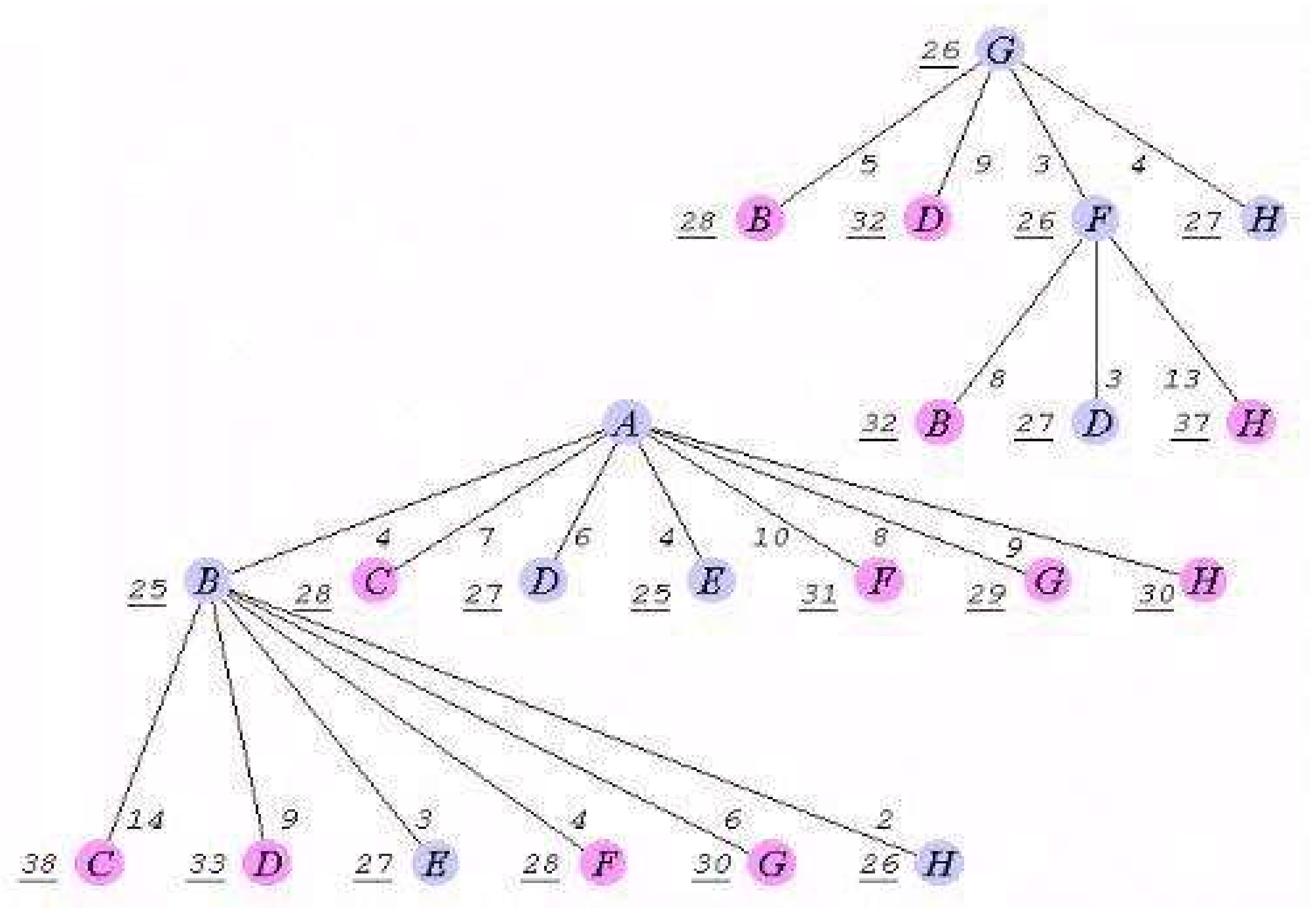
6. We have fully expanded node E so it is removed from our live-node list and we have two new nodes to add to the list. When two nodes have the same bounding values we will choose the one that is closer to the solution. So the order of nodes in our live-node list is (C2,B1,B2,D1).



7. We remove node C2 and add node G3 to the live-node list. (G3,B1,B2,D1).

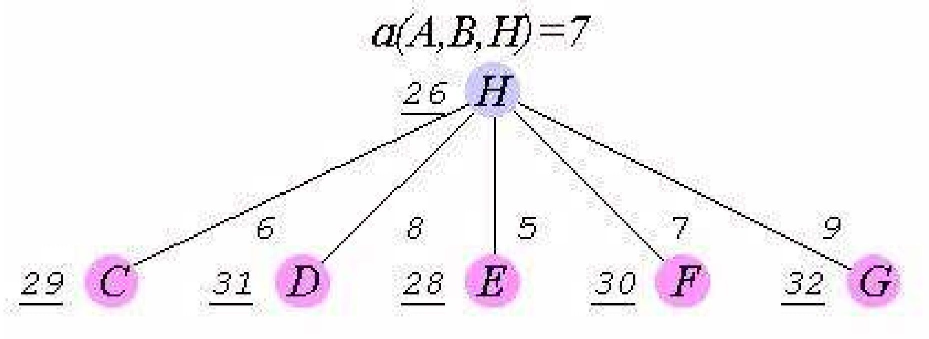


8. Expanding node G gives us two more promising nodes F4 and H4. Our live-node list is now (F4,B1,H4,B2,D1).



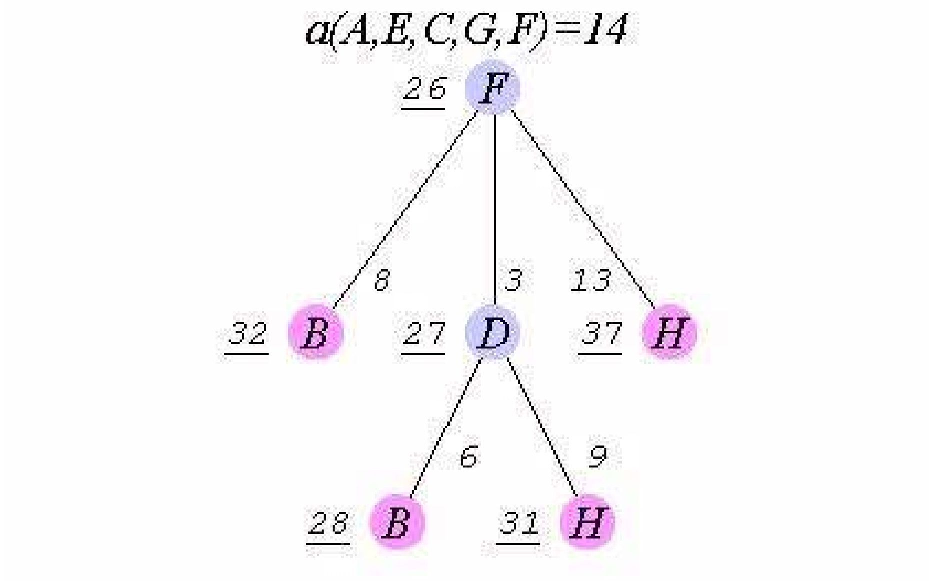
9. A 1st level node has moved to the front of the live-node list. (B1,D5,H4,B2,D1)

10. (H2,D5,H4,B2,D1)



11. (D5,H4,B2,D1)

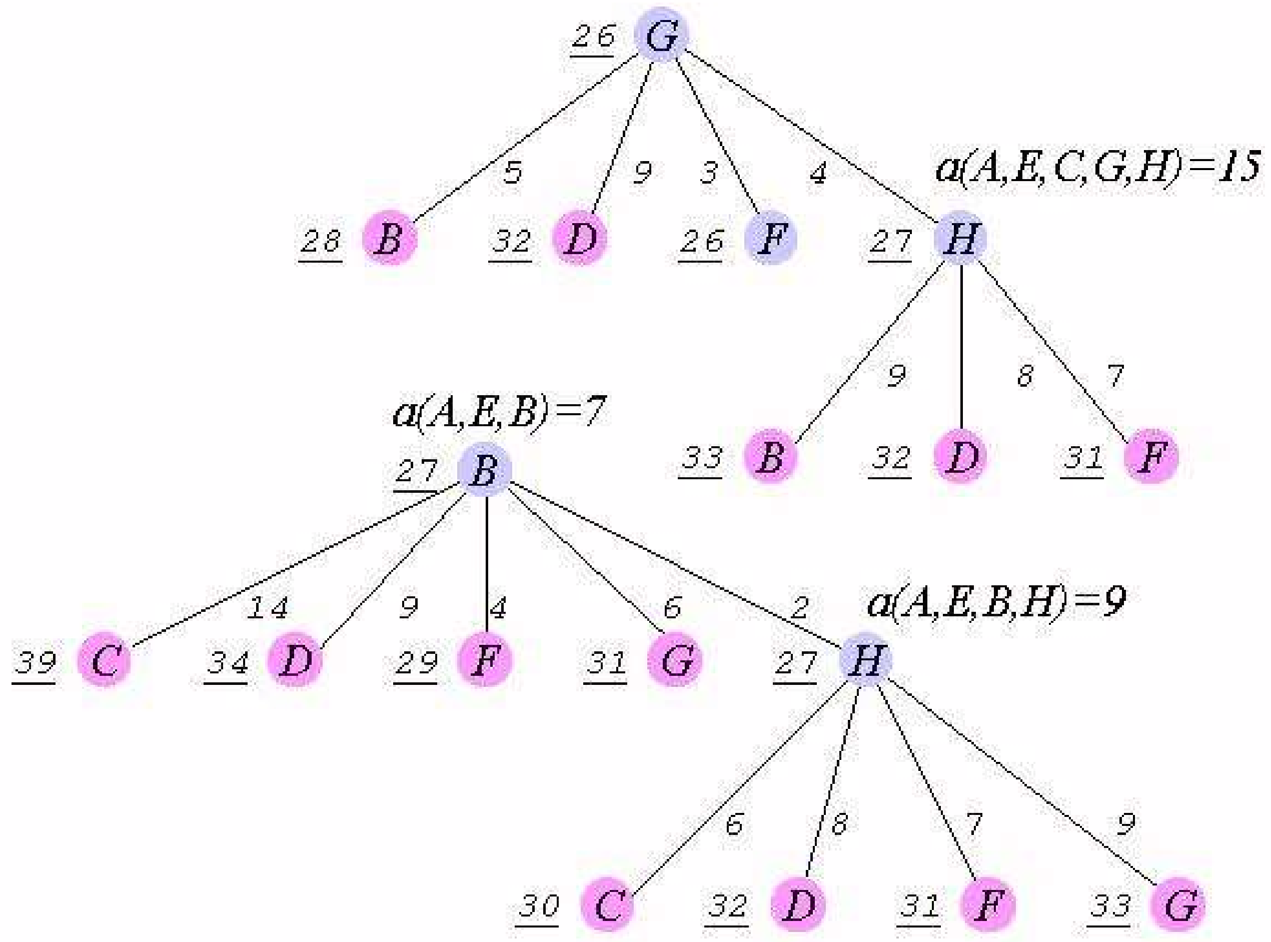
12. (H4,B2,D1)



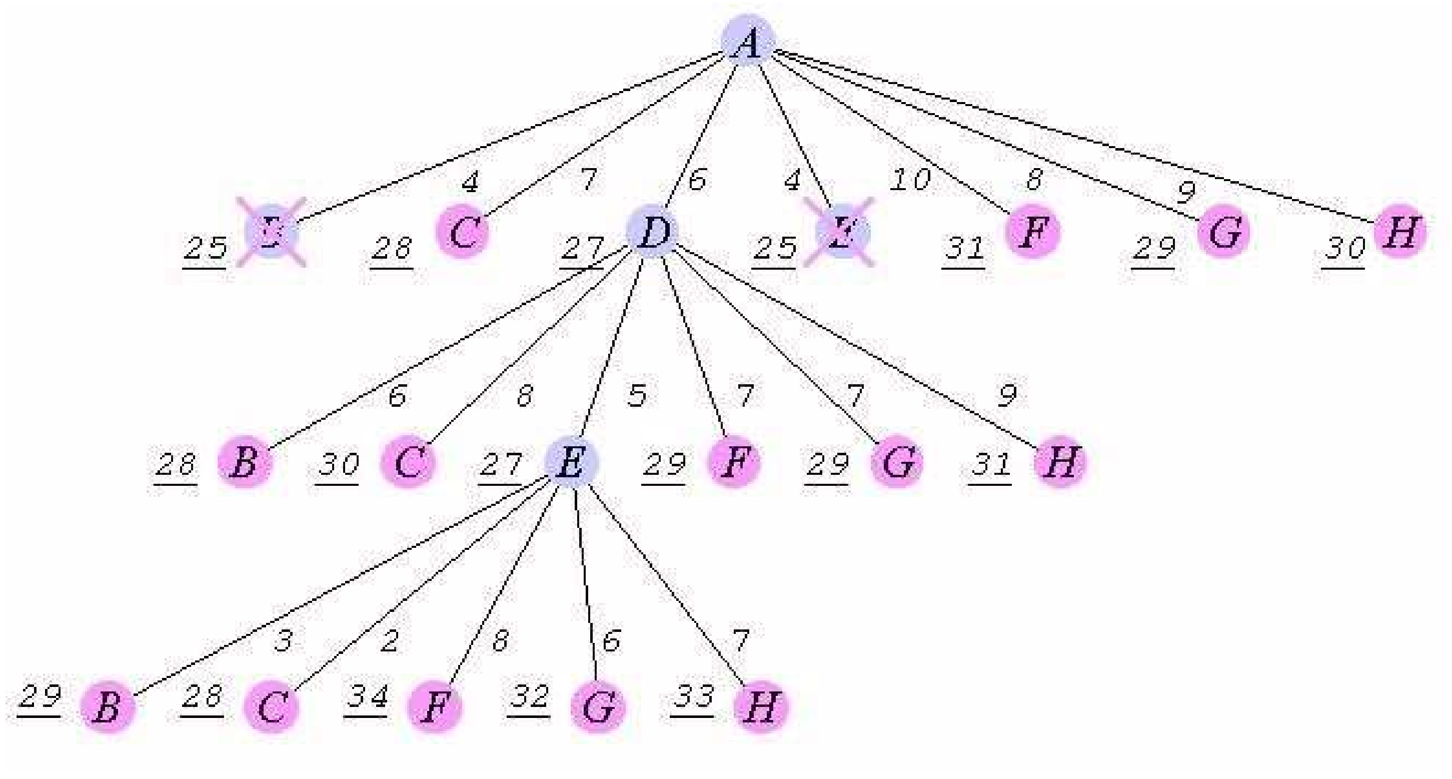
13. (B2,D1)

14. (H3,D1)

15. (D1)



26 G
5
9
3
4
a(A,E,C,G,H)=15
28 B
32 D
26 F
27 H
9
8
7
a(A,E,B)=7
27 B
33 B
32 D
31 F
14
9
4
6
2
a(A,E,B,H)=9
39 C
34 D
29 F
31 G
27 H
6
8
7
9
30 C
32 D
31 F
33 G

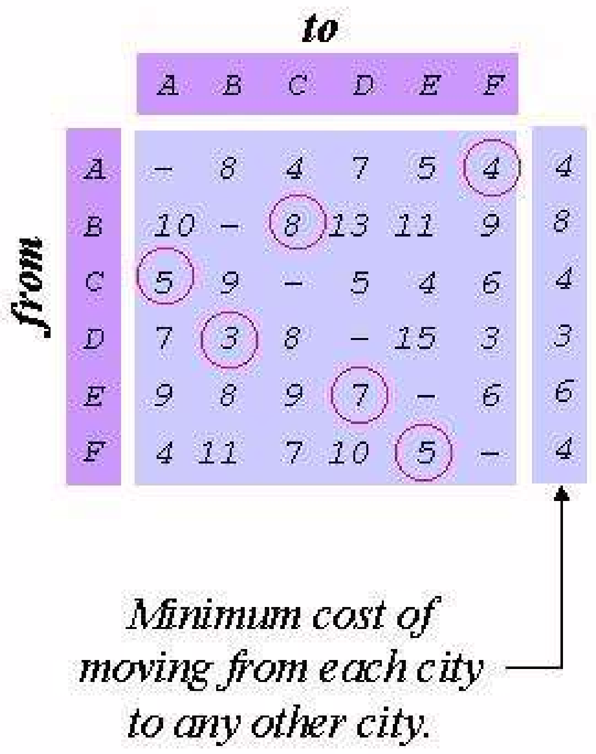


16.We have verified that a minimal tour has length >= 28. Since our candidate tour has a length of 28 we now know that it is a minimal tour, we can use it without further searching of the state-space tree.

Another Example

In the previous example the initial candidate tour turns out to be a minimal tour. This is usually not the case. We will now look at problem for which the initial candidate tour is not a minimal tour.

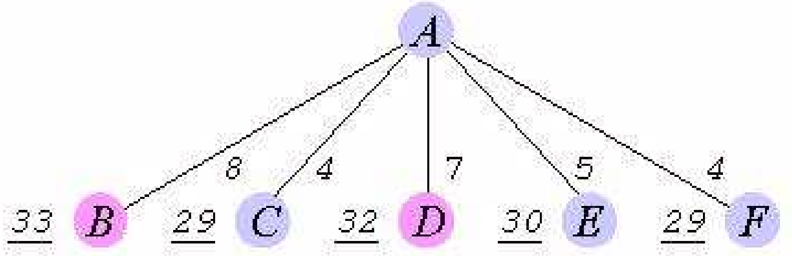
The initial tour is A-F-E-D-B-C-A with a total length of 32. The sum of the minimum costs of leaving every node is 29 so we know that a minimal tour cannot have length less than 29.



to

from	A	B	C	D	E	F	
A	–	8	4	7	5	4	4
B	10	–	8	13	11	9	8
C	5	9	–	5	4	6	4
D	7	3	8	–	15	3	3
E	9	8	9	7	–	6	6
F	4	11	7	10	5	–	4

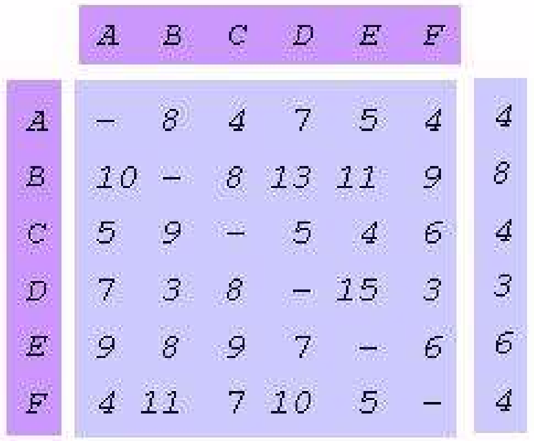
Minimum cost of moving from each city to any other city.



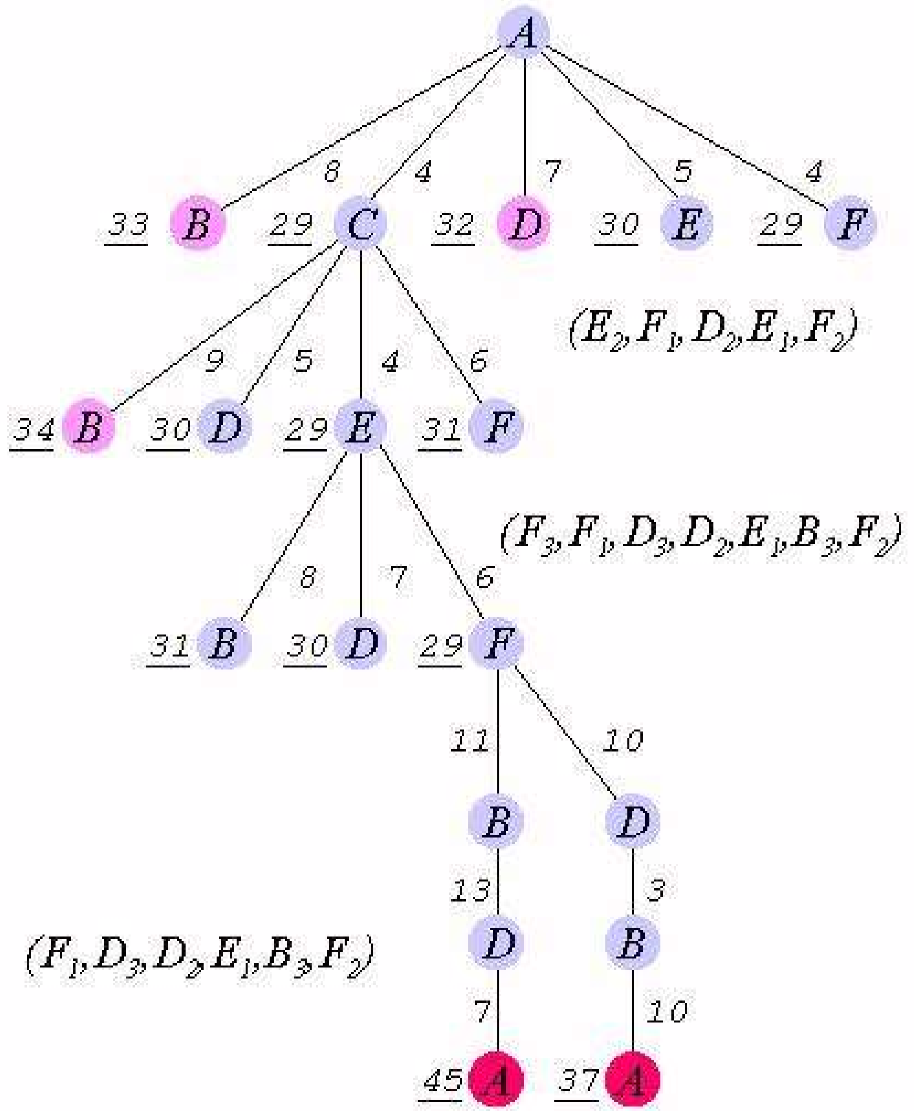
$$h(A,B) = a(A,B) + g(B,C,D,E,F)$$
$$32 \quad = \quad 8 \quad + \quad 25$$

live-node list = (C_1, F_1, E_1)

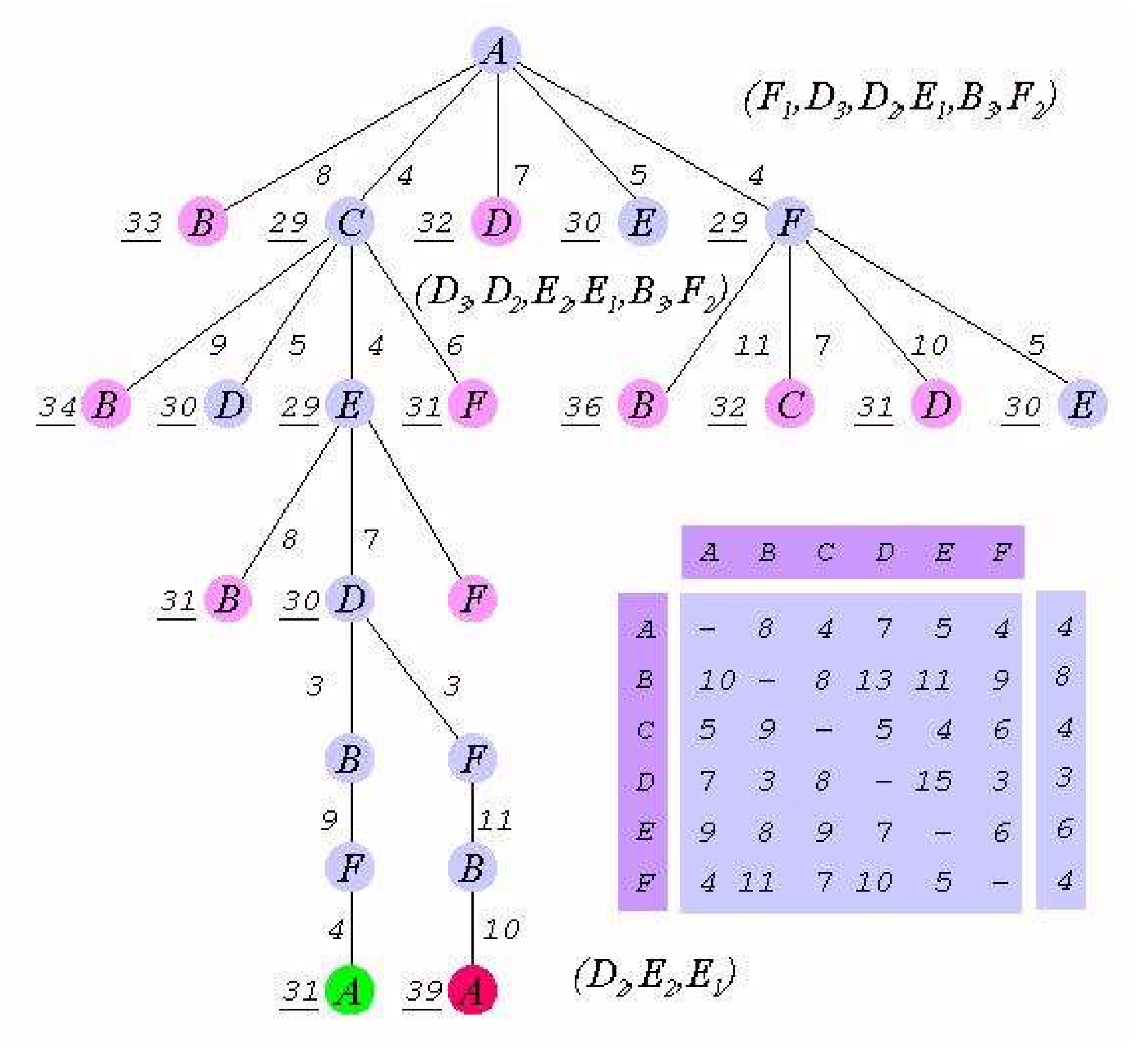
The live-node list is ordered by bounding value and then by level of completion. That is, if two nodes have the same bounding value we place the one that is closer to completion ahead of the other node. This is because the bounding value of the of the node closer to completion is more likely to be closer to the actual tour length.



	A	B	C	D	E	F	
A	–	8	4	7	5	4	4
B	10	–	8	13	11	9	8
C	5	9	–	5	4	6	4
D	7	3	8	–	15	3	3
E	9	8	9	7	–	6	6
F	4	11	7	10	5	–	4

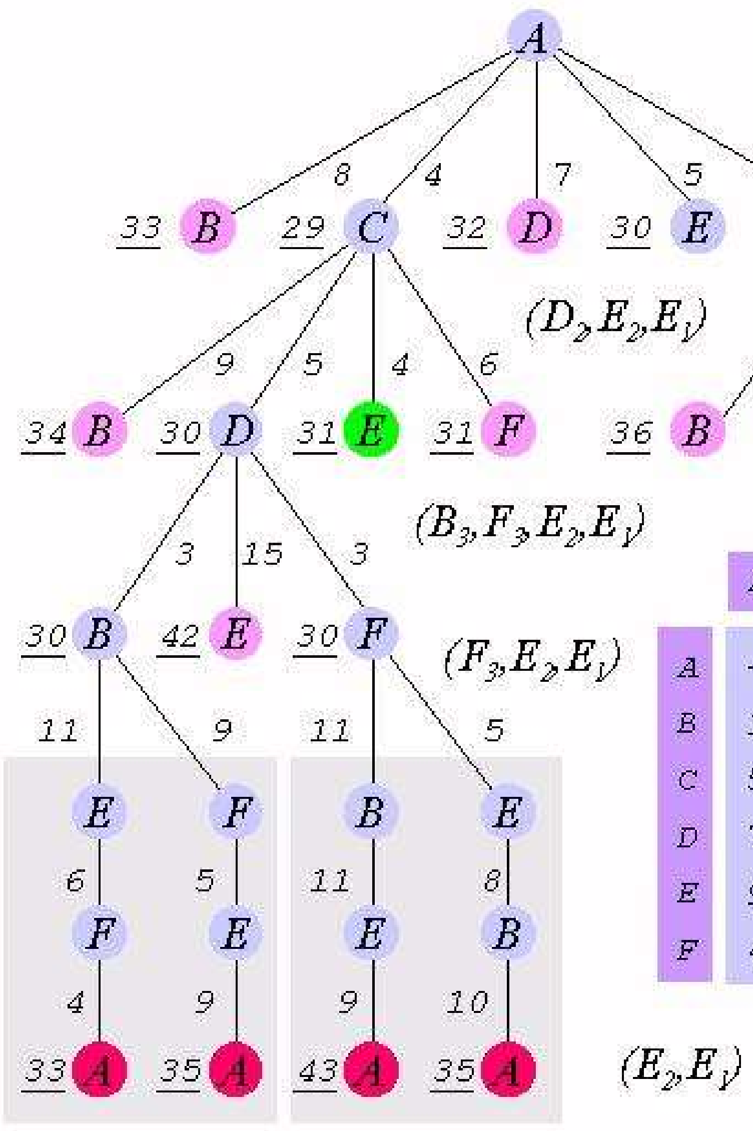


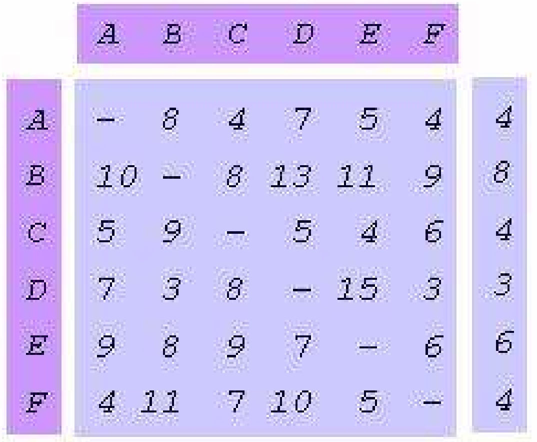
The node shown in red were expanded from an active node (F_3) with a possible tour length of 29. However the completed tours turned out to be significantly larger.



	A	B	C	D	E	F	
A	–	8	4	7	5	4	4
B	10	–	8	13	11	9	8
C	5	9	–	5	4	6	4
D	7	3	8	–	15	3	3
E	9	8	9	7	–	6	6
F	4	11	7	10	5	–	4

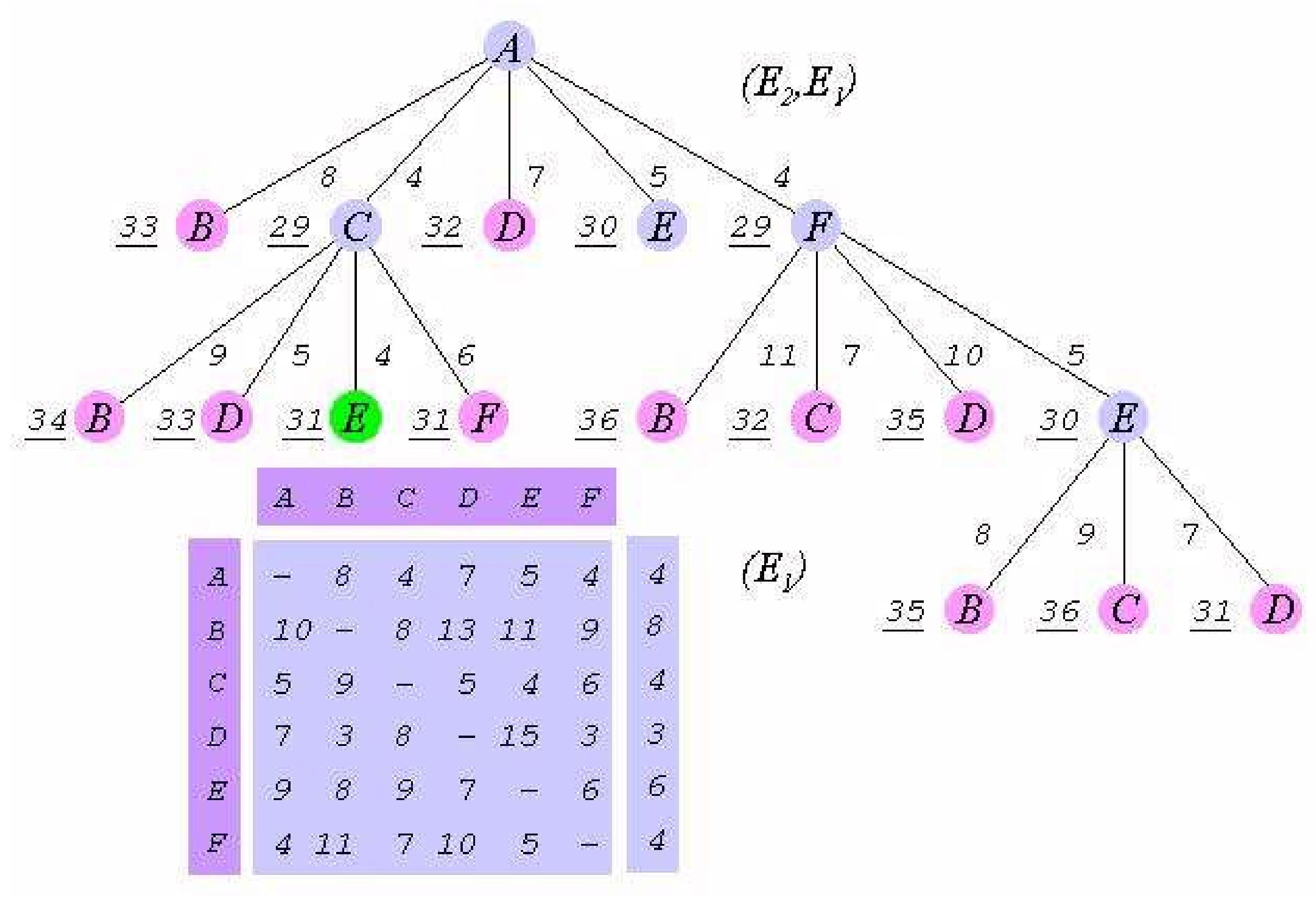
Expanding D_3 we find an actual tour that is shorter than our initial tour. We do not yet know if this is a minimal tour but we can remove nodes from our live-node list that have a bounding value of >= 31. Now we expand D_2 to find that it leads to tours of length >= 33.





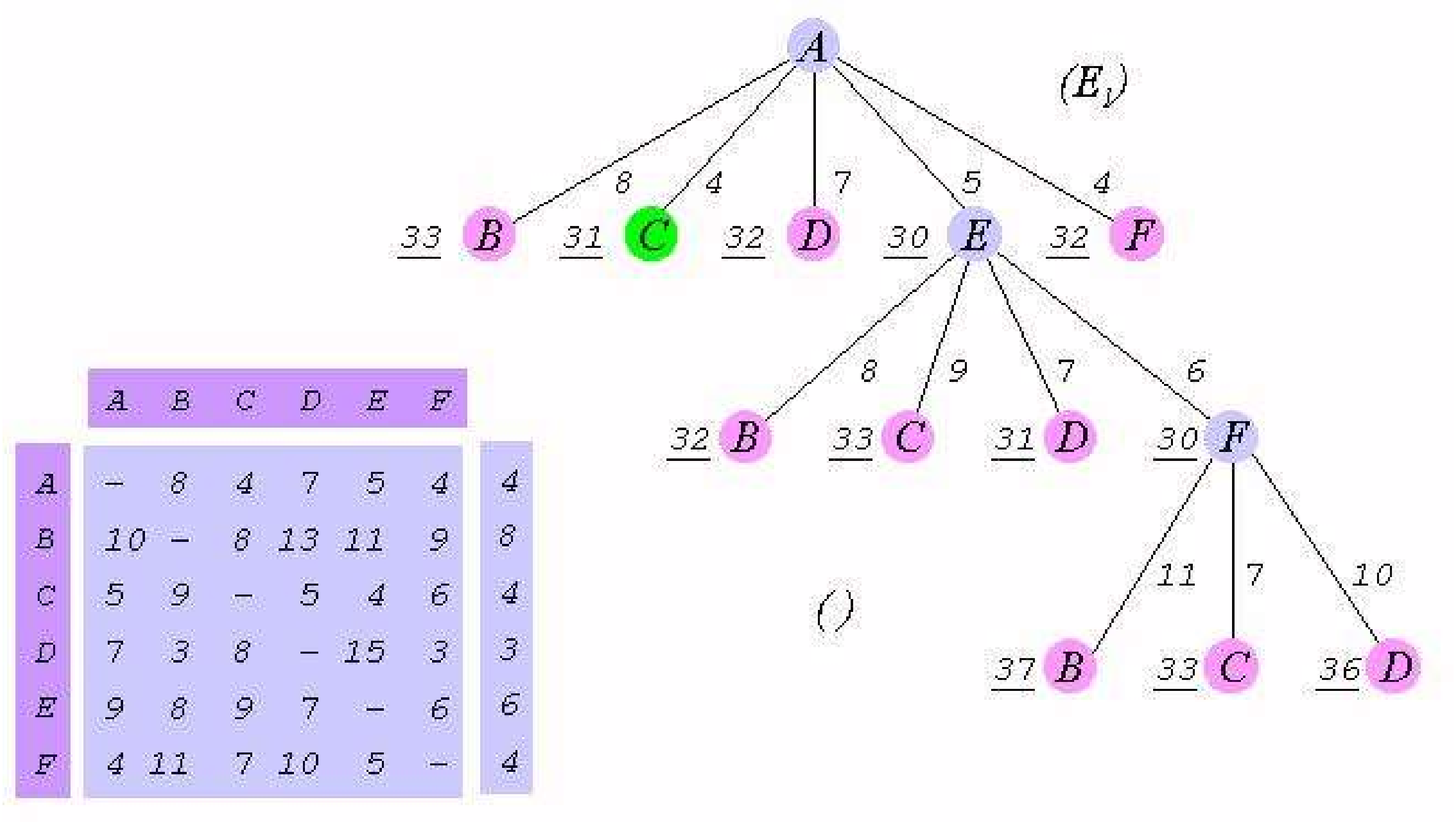
	A	B	C	D	E	F	
A	–	8	4	7	5	4	4
B	10	–	8	13	11	9	8
C	5	9	–	5	4	6	4
D	7	3	8	–	15	3	3
E	9	8	9	7	–	6	6
F	4	11	7	10	5	–	4

Expansion of E_2 eliminates this path as a candidate for a minimal tour.



	A	B	C	D	E	F	
A	–	8	4	7	5	4	4
B	10	–	8	13	11	9	8
C	5	9	–	5	4	6	4
D	7	3	8	–	15	3	3
E	9	8	9	7	–	6	6
F	4	11	7	10	5	–	4

When the live-node list is empty we are finished. In this example a minmal tour is A-C-E-D-B-F-A with a length of 31 we have found a tour that is shorter than the original candidate tour.



	A	B	C	D	E	F	
A	–	8	4	7	5	4	4
B	10	–	8	13	11	9	8
C	5	9	–	5	4	6	4
D	7	3	8	–	15	3	3
E	9	8	9	7	–	6	6
F	4	11	7	10	5	–	4

Branch and bound Applications

This approach is used for a number of NP-hard problems, such as

- Knapsack problem
- Integer programming
- Nonlinear programming
- Traveling salesman problem (TSP)
- Quadratic assignment problem (QAP)
- Maximum satisfiability problem (MAX-SAT)
- Nearest neighbor search (NNS)
- Cutting stock problem
- False noise analysis (FNA)
- Computational phylogenetics