CSE 6140/ CX 4140
Computational Science and Engineering ALGORITHMS

Coping with NP-completeness - 1

Instructor: Xiuwei Zhang

Assistant Professor

School of Computational Science and Engineering

Partially based on slides by Prof. Ümit V. Çatalyürek

Schedule for this week

Monday 10/19:
First lecture on the new chapter -- coping with NPC problems (backtracking)

Tuesday 10/20:
Review for Test2.
Introduction to the project.

Monday 10/26:
Second lecture on coping with NPC problems (branch-and-bound)

Wednesday 10/28:
Test2

Dealing with NP-complete problems

Branch & bound	Sacrifice running time: create an algorithm with running time exponential in the input size (but which might do well on the inputs you use)
Approximation	Sacrifice quality of the solution: quickly find a solution that is *provably not very bad*
Local search	Quickly find a solution for which you cannot give any quality guarantee (but which might often be good in practice on real problem instances)
Restriction	By restricting the structure of the input (e.g., to planar graphs, 2SAT), faster algorithms are usually possible.
Randomization	Use randomness to get a faster average running time, and allow the algorithm to fail to find optimum with some small probability.
Parameterized algorithms	Sacrifice running time: allow the running time to have an exponential factor, but ensure that the exponential dependence is not on the entire input size but just on some parameter that is hopefully small

Exact Solution Strategies

- ***exhaustive search*** (*brute force*)
  - useful only for small instances
- ***backtracking***
  - Construct the alternatives component by component
  - <u>eliminates some unnecessary cases </u>from consideration
  - yields solutions in better time for many instances but worst case is still exponential
- ***branch-and-bound***
  - further refines the backtracking idea (eliminating unnecessary cases) for optimization problems

Solution Space of a problem

- SAT: all possible assignments - 2^n
- Vertex Cover: all possible subsets of nodes of size k: n choose k
- Set Cover: all possible subsets of sets of size k
- Hamiltonian Cycle: all possible permutations of nodes – n!
- Knapsack: all possible subsets of items - 2^n

- Brute Force search will always explore all of these options
- Backtracking and branch-and-bound perform a systematic search
  - removing some options when possible
  - often taking much less time than taken by a brute force search.

Backtracking

- Backtracking is a systematic method to iterate through all the possible configurations of a search space.
  - It is a general algorithm/technique which must be customized for each individual application.

- In the general case, we will model our solution as a vector $a = (a_1; a_2; \ldots; a_n)$, where each element a_i is selected from a finite ordered set S_i.

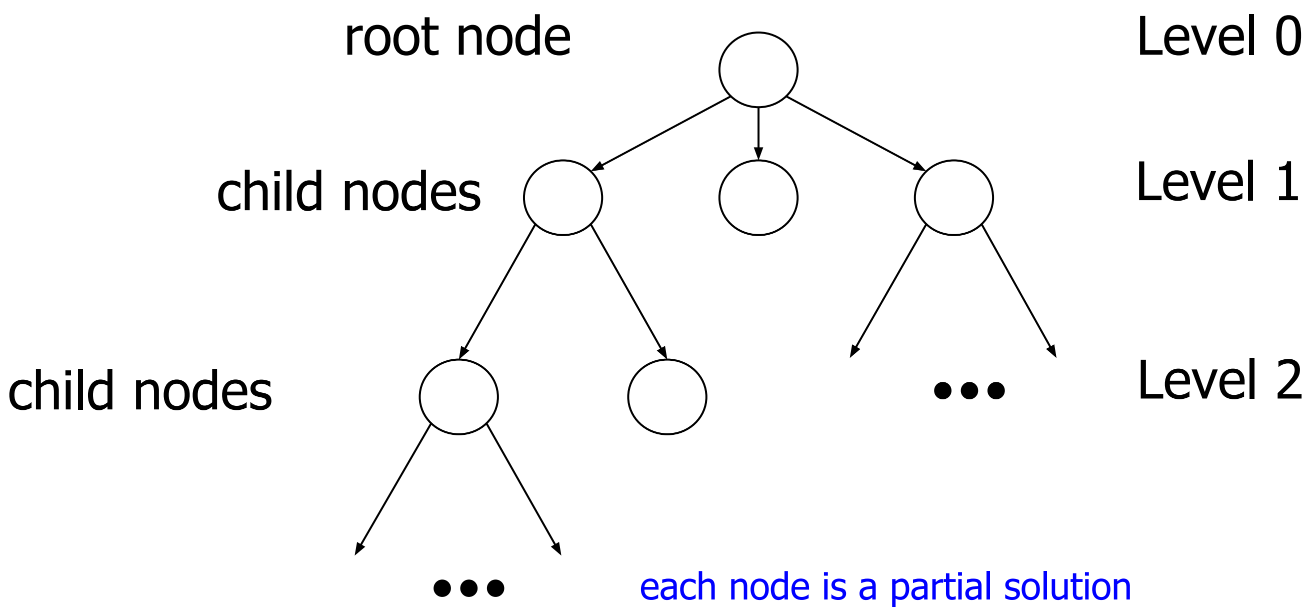
  - E.g. Such a vector might represent an arrangement where a_i contains the i-th element of the permutation.
  - E.g. Or the vector might represent a given subset S, where a_i is true if and only if the i-th element of the universe is in S.

Backtracking

- The principal idea is to construct solutions one component at a time and evaluate such partially constructed candidates as follows.
- At each step in the backtracking algorithm, we start from a given partial solution, say, a = (a_1; a_2; :::; a_k), and try to extend it by adding another element at the end.
- If a partially constructed solution can be extended further without violating the problem's constraints, it is done by taking the first remaining legitimate option for the next component.
- If there is no legitimate option for the next component, no alternatives for *any* remaining component need to be considered.
  - In this case, the algorithm backtracks to replace the last component of the partially constructed solution with its next option.

Backtracking

- This kind of processing is often implemented by constructing a tree of choices being made, called the ***state-space tree***.
- Its **root** represents an **initial state** before the search for a solution begins.
- The **nodes of the first level** in the tree represent the **choices made for the first component** of a solution,
- the nodes of the second level represent the choices for the second component, and so on.



Backtracking

- A node in a state-space tree is said to be ***promising*** if it corresponds to a partially constructed solution that may still lead to a complete solution; otherwise, it is called ***nonpromising***.

- **Leaves** represent either **nonpromising dead ends** or **complete solutions** found by the algorithm.

- If the current node turns out to be nonpromising, the algorithm backtracks to the node’s parent to consider the next possible option for its last component;

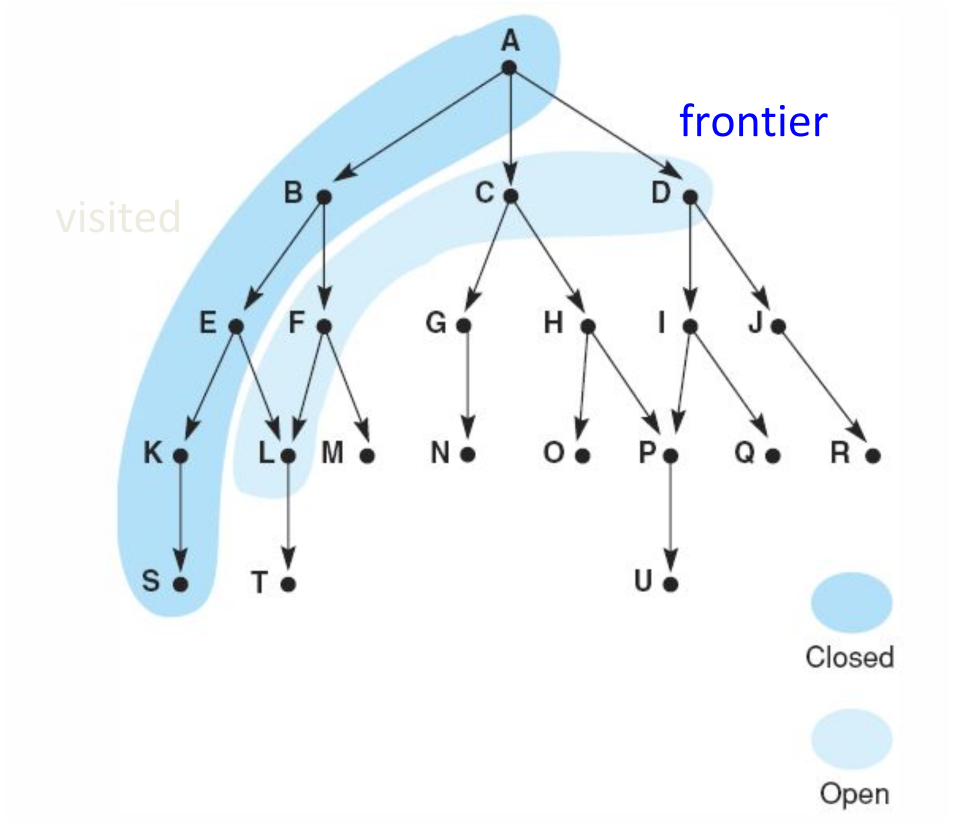
- if there is no such option, it backtracks one more level up the tree, and so on.

Backtracking

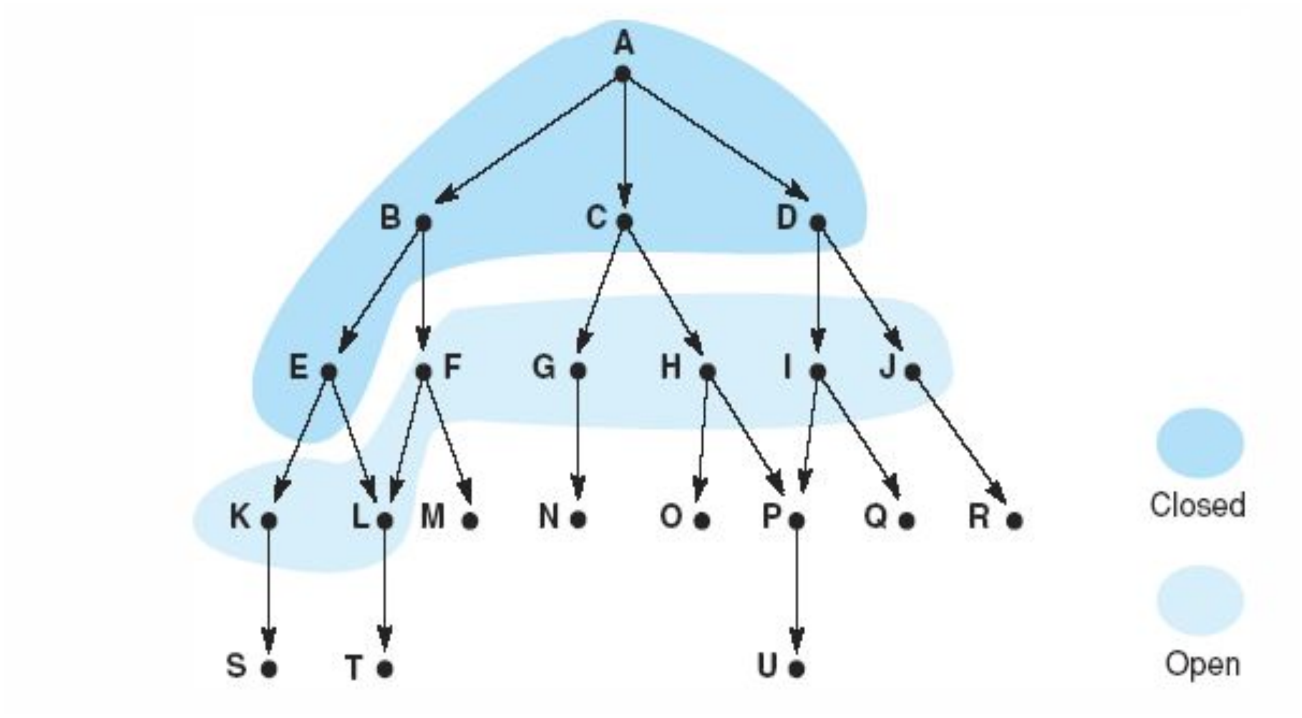
```
Backtracking(P) // Input: problem P
F ← {(∅,P)}     // Frontier set of configurations
while F ≠ ∅ do
  Choose (X,Y) ∈ F – the most "promising" configuration
  Expand (X,Y), to candidate extensions (new choices)
  Let $(X_1,Y_1)$, $(X_2,Y_2)$, ..., $(X_k,Y_k)$ be extended candidates
  for each new configuration $(X_i,Y_i)$ do
    "Check" $(X_i,Y_i)$
    if "solution found" then
      return the solution derived from $(X_i,Y_i)$
    if not "dead end" then
      F ← F ∪ {$(X_i,Y_i)$}
  // else nothing to expand from
return "no solution"
```

(X,Y) associated with each node, where X is a partial solution, and Y is the remaining subproblem

Depth-first search (LIFO - last in first out)



Breadth-first search (FIFO first in first out)



Best-first Search

- Best-first search expands the most promising node/partial solution in the set F

- How would you implement a best-first search?
  - Depth-first is a stack
  - Breadth-first is a queue
  - Best-first is a priority queue

Backtracking

```
Backtracking(P) // Input: problem P
F ← {(∅,P)}     // Frontier set of configurations
while F ≠ ∅ do
  Choose (X,Y) ∈ F – the most "promising" configuration
  Expand (X,Y), to candidate extensions (new choices)
  Let $(X_1,Y_1)$, $(X_2,Y_2)$, ..., $(X_k,Y_k)$ be extended candidates
  for each new configuration $(X_i,Y_i)$ do
    "Check" $(X_i,Y_i)$
    if "solution found" then
      return the solution derived from $(X_i,Y_i)$
    if not "dead end" then
      F ← F ∪ {$(X_i,Y_i)$}
  // else nothing to expand from
return "no solution"
```

Satisfiability

- We will branch on a variable x, assigning True or False
- Assigning x=T
  - means all the clauses that contain literal x are satisfied
  - We can remove ¬x from all clauses where it appears (we need to find another literal to satisfy the clause)
- Similarly for x=F
- Each node of the backtracking search tree is associated with **X_i**, a partial assignment of variables to True or False
- Alternatively, think of each node as the **subproblem Y_i** remaining after applying the partial assignment, i.e. the remaining SAT subformula
  - we consider having **(X_i, Y_i) where Y_i is the remaining subproblem after making assignment X_i**

Satisfiability

- $\Phi = (w \lor x \lor y \lor z) \land (w \lor \neg x) \land (x \lor \neg y) \land (y \lor \neg z) \land (z \lor \neg w) \land (\neg w \lor \neg z)$

- X = {w=True} (partial assignment)
  - $Y = (x \lor \neg y) \land (y \lor \neg z) \land (z) \land (\neg z)$ (corresponding subproblem)

- <u>*Check(X,Y)*</u>:
  - If the subproblem has no clauses (all have been satisfied) -> we have solution
  - If the subproblems has an empty clause (all literals have been assigned false) -> we have no solution
  - Else, we have a new subproblem

Satisfiability

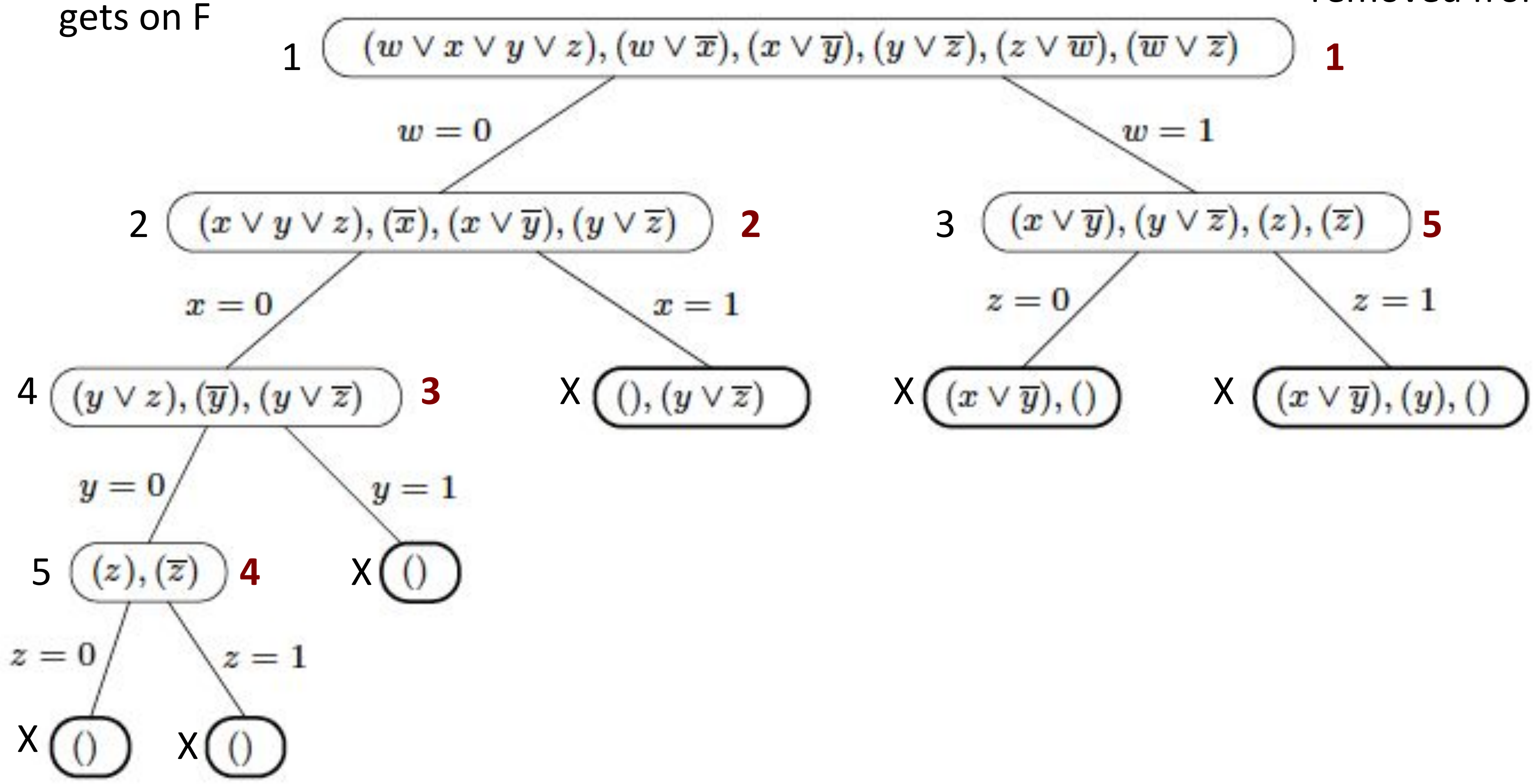
- $\Phi = (w \vee x \vee y \vee z) \wedge (w \vee \neg x) \wedge (x \vee \neg y) \wedge (y \vee \neg z) \wedge (z \vee \neg w) \wedge (\neg w \vee \neg z)$
- *Choose*: which subproblem to expand next
- *Expand*: which branching variable to use

- The strength of backtracking is that it eliminates unnecessary subproblems
  - we want *Check* to fail, i.e., when there is empty clause
- *Choose* the subproblem with the smallest clause
- *Expand* a variable that appears in the smallest clause of the chosen subproblem

Satisfiability

Time at which subproblem gets on F

Time at which subproblem is removed from F


1
$(w \vee x \vee y \vee z), (w \vee \overline{x}), (x \vee \overline{y}), (y \vee \overline{z}), (z \vee \overline{w}), (\overline{w} \vee \overline{z})$
1
$w = 0$
$w = 1$
2
$(x \vee y \vee z), (\overline{x}), (x \vee \overline{y}), (y \vee \overline{z})$
2
3
$(x \vee \overline{y}), (y \vee \overline{z}), (z), (\overline{z})$
5
$x = 0$
$x = 1$
$z = 0$
$z = 1$
4
$(y \vee z), (\overline{y}), (y \vee \overline{z})$
3
X $(), (y \vee \overline{z})$
X $(x \vee \overline{y}), ()$
X $(x \vee \overline{y}), (y), ()$
$y = 0$
$y = 1$
5
$(z), (\overline{z})$
4
X $()$
$z = 0$
$z = 1$
X $()$
X $()$

Hamiltonian Cycle

- HAM-CYCLE: given an undirected graph G = (V, E), does there exist a simple cycle Γ that contains every node in V.
- Partial Solution: (a, …, b) = (a,T,b)
  - a path from a vertex **a** to **b** going through vertices $T \subseteq V - \{a, b\}$
  - Decide the next vertex in the permutation
  - Start with partial solution (v_1, ∅, v_1)
- Subproblem: find a Hamiltonian Path from b to a in the **subgraph obtained by deleting nodes T and all their edges**
- *Expand*: choose an edge from b towards a node in V-T-{a,b}
- *Check*:
  - If G-T-{a,b} is disconnected or a node in V-T-{a,b} has degree 1, then "dead end"
  - If G-T is a path from b to a, then "Solution"
  - Else, new subproblem

Branch-and-Bound

- An enhancement of backtracking
- Applicable to optimization problems (assume we are minimizing)

- Keep track of BEST solution found so far (upper bound on optimal)
- For each node (partial solution), computes a lower bound LB on the value of the objective function for all descendants of the node (extensions of the partial solution)
  - any extension of this partial solution will have quality at least LB
- Uses the bound for:
  - ruling out certain nodes as “nonpromising” to prune the tree – if a node’s bound is not better than the best solution seen so far
  - guiding the search through state-space as a measure of “promise”