Georgia Tech

CSE 6140/ CX 4140:

Computational Science and Engineering

ALGORITHMS

Instructor: Anne Benoit

Visiting Associate Professor, CSE

Based on slides by Bistra Dilkina, Jennifer Welch, George Bebis, and Kevin Wayne

CSE 6140 / CX 4140

PROJECT

VERTEX COVER optimization problem

Teams

- Randomly assigned
- Groups of 3 or 4
- Please fill the questionnaire for Programming Languages
- Team assignments (letter) will be released by the end of the week
- Project released next Tuesday (Oct 31)

- **No swaps allowed**

4 Algorithms

- Approximation Algorithm
- Local Search (2 variants)
  - Choose a method to select a starting solution
  - Choose a neighborhood
  - Choose a method to explore search space
    - Hill Climbing, Simulated Annealing, Iterated Local Search, Tabu Search, Genetic Algorithm
- Branch-and-Bound
  - How do you branch/expand on each node?
  - How do you bound on each node?

Executable

- Parameters
  - -alg [BnB | Approx | LS1 | LS2]
  - -inst filename.gr
  - -time cutoff (in minutes)
- Output:
  - SolFile <filename_method.sol> with 2 lines:
    - line1: quality of best solution found
    - line2: list of vertices in cover v1,v2,…,vn,v1
  - SolTrace <filename_method.trace> for BnB and LS
    - Each line is 2 numbers: timestamp quality
    - Record every time when a new better solution is found

Deliverables

- Partial report/check in on Thu. Nov 16
  - You should have been able to parse input and produce output
  - You should have at least 2 approaches working by then
  - Report best quality within 10 mins for each benchmark instance
- Report (the bulk of your grade) due on Tue. Dec 5
  - Written like a proper academic paper
  - Describes your approaches and choices, Data structures, Worst-case running time
  - For each instance, report results on all ALGs in terms of relative optimality gap and time with max X mins time cutoffs
  - Bnb: Does the empirical running time match the worst-case complexity?
  - Approx: How does the empirical relative error compare to the approximation guarantee?
  - How do the methods compare?

Deliverables

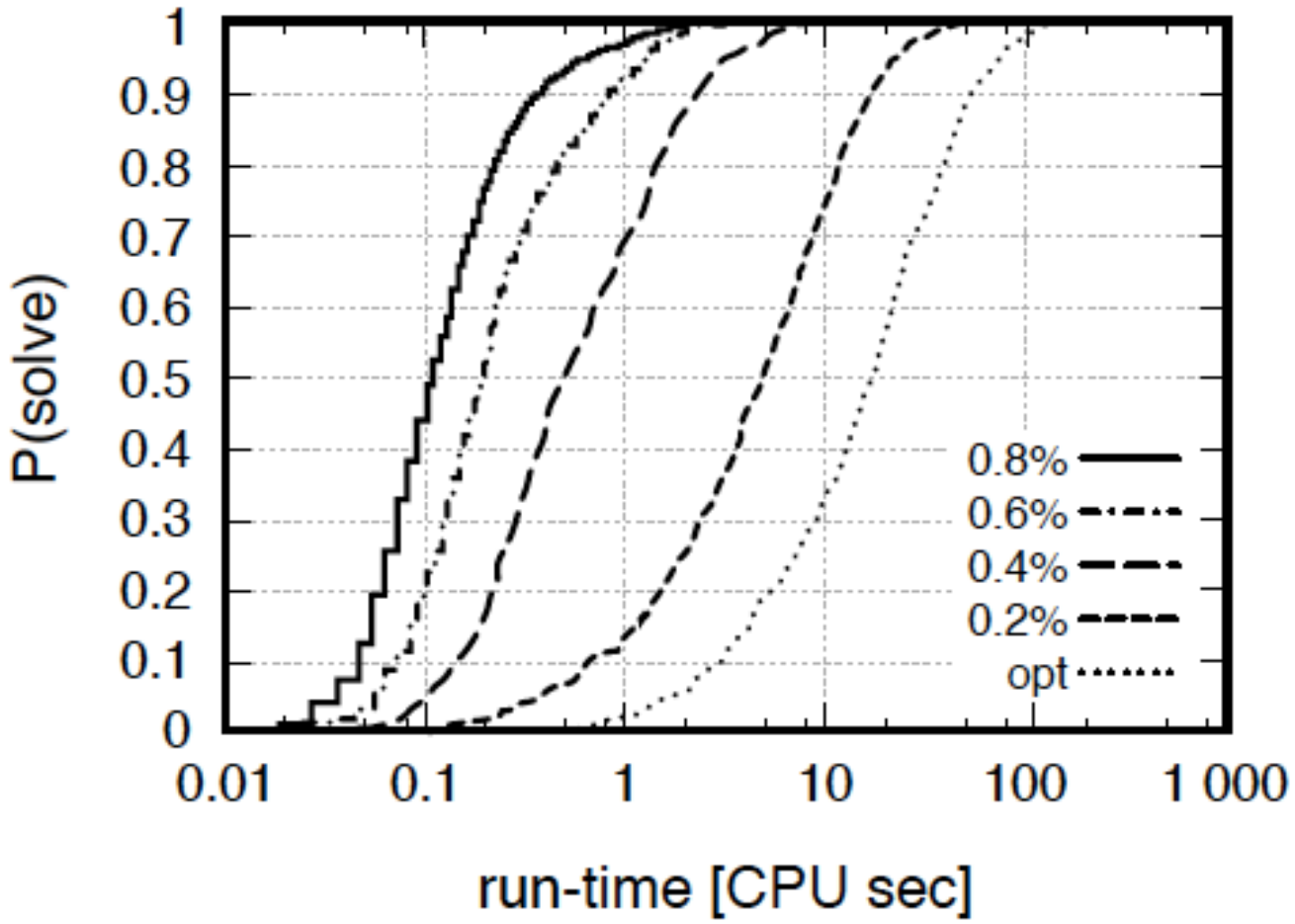
- Local Search Approaches
  - Why did you make the choices that you did?
  - Report any sources of ideas you have used
  - Try to be creative – make your own variant of a known LS approach, don't just implement exactly the same
  - Select 3 instances:
    - Generate QRTDs and SQDs
    - Boxplot for each instance with Mean, Median, Q1, Q3, etc
- Enter your LS approaches in our Competition!!! – submit entries of your LS approaches each week to compete against other teams (starting Nov. 16)

Team Participation

- Each individual will tell us by Dec. 6 a thoughtful and honest evaluation of the contributions of your group members, including yourself (on t-square)
- For each individual, include a score from 1 to 10 indicating your evaluation of their work (10 meaning they were a valuable member of the team that made significant contributions to the project and were good to work with, 1 meaning they contributed none or very little and were difficult to work with).
- You may also include any clarifying comments, etc. (especially for low scores).
- If a person receives consistently low evaluations from peers, then their score will be proportionally decreased.

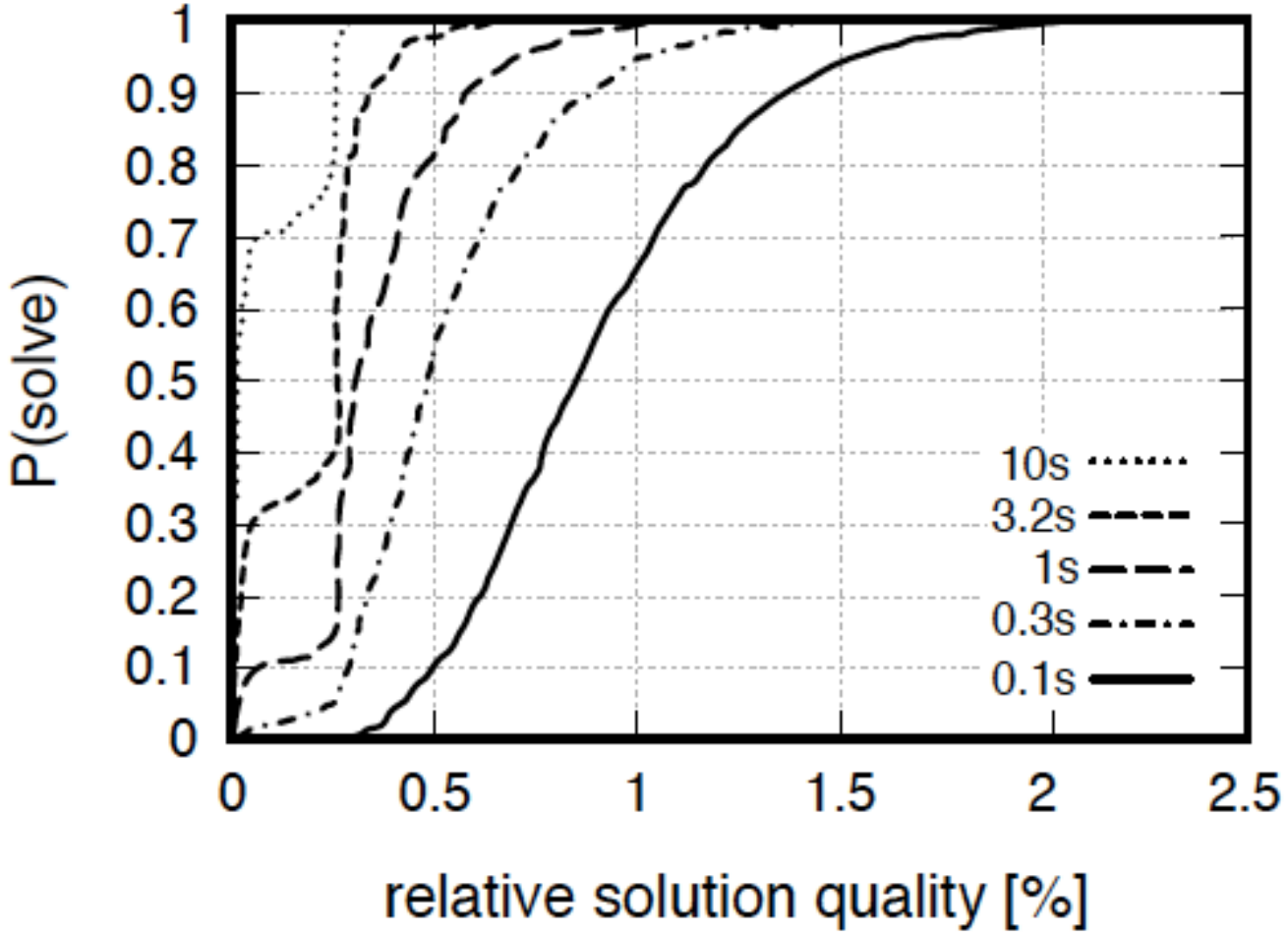
Georgia Tech

Qualified RTDs for various solution qualities:



Georgia Tech

Solution quality distributions for various run-times:



Georgia Tech

CSE 6140 / CX 4140

Dealing with NP-complete problems

Backtracking

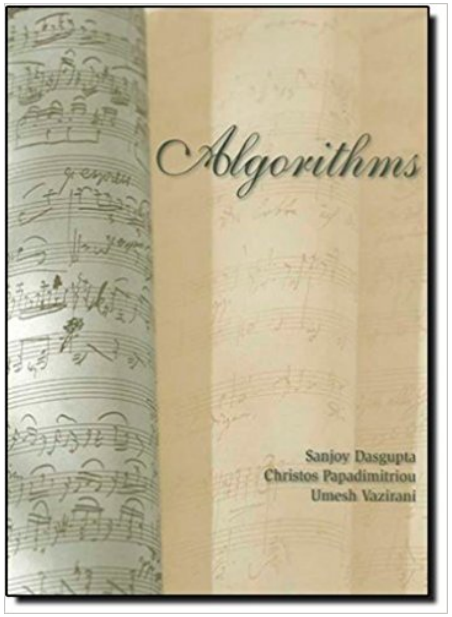
Dealing with NP-complete problems

Georgia Tech

Branch & bound, Backtracking	Sacrifice running time: create an algorithm with running time exponential in the input size (but which might do well on the inputs you use)
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
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.

Reading

- Dasgupta/Papadimitriou/Vazirani: Algorithms, ch. 9 (on t-square)
- BRV8 (exact solutions & approx, on t-square)
- KT12 (local search), KT11 (approx)
- CLRS35 (approx)



Exact Solution Strategies

Georgia Tech

- *Exhaustive search* (brute force)
  - Useful only for small instances
- ***Backtracking (decision problems)***
  - Construct the alternatives component by component
  - Eliminates some unnecessary cases from consideration
  - Yields solutions in better time for many instances but worst case is still exponential
- ***Branch-and-bound (optimization problems)***
  - 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

A first example: the n queens

- Chess board with n rows and n columns
- Place n queens on this chess so that none of them can attack any other in one move (attack can be done horizontally, vertically, or diagonally, and a queen can move as far as she wants on either of these directions)

- Search space size?
  - n^n (exactly one queen per row)
  - Many constraints -> many solutions can be discarded

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 ith 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 ith 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.

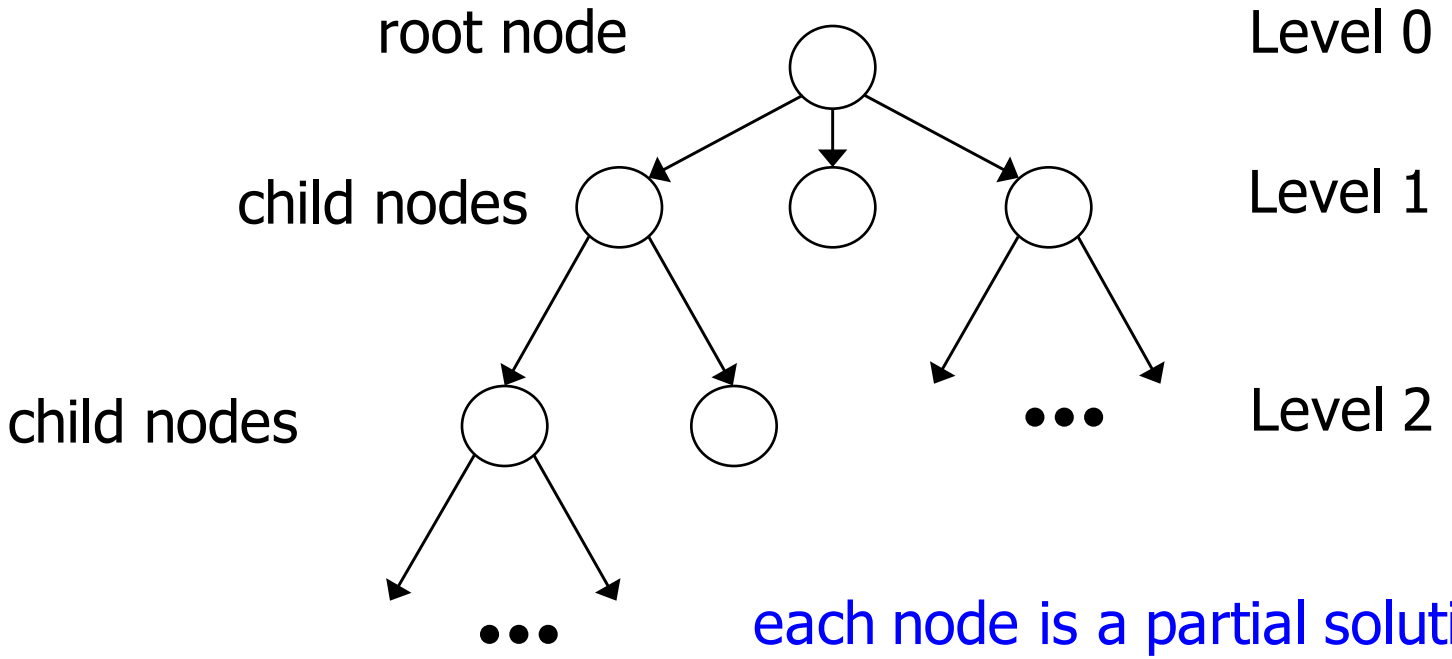
- At each step in the backtracking algorithm, we start from a given partial solution, say, $a = (a_1; a_2; \ldots ; a_k)$, and try to extend it by adding another element at the end.

- 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.

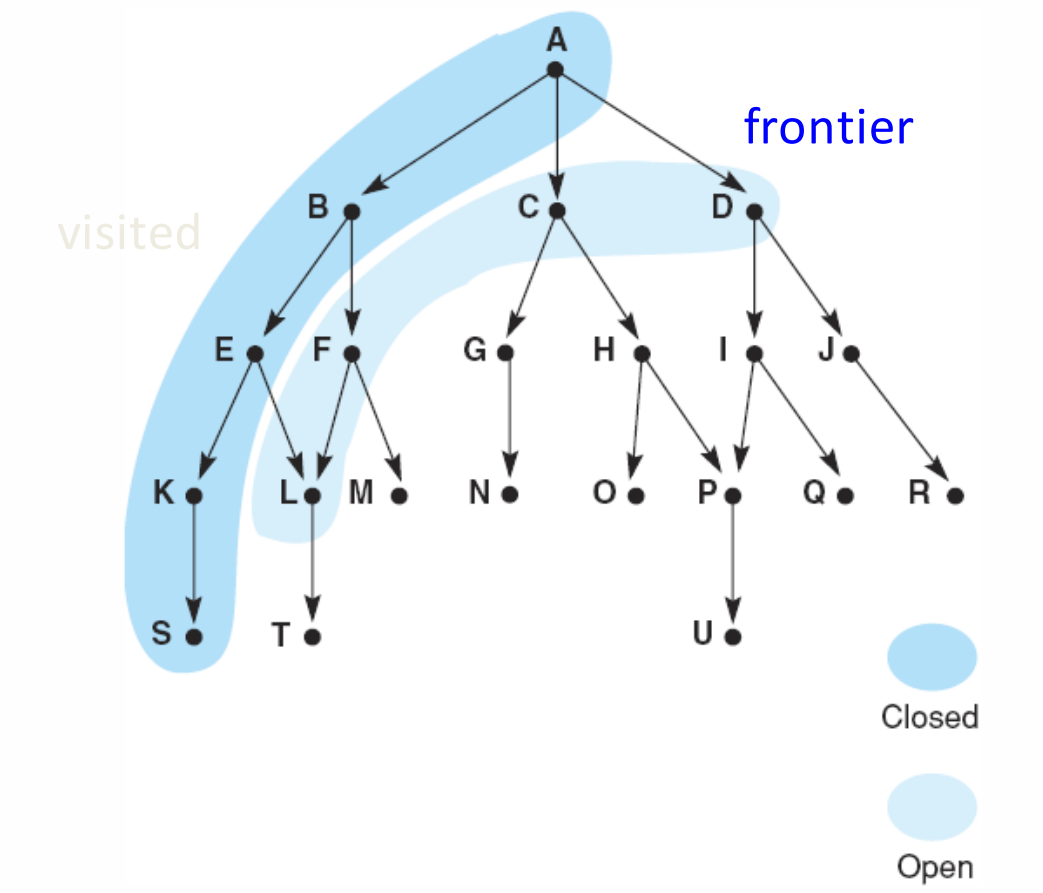
- If there is no such option, it backtracks one more level up the tree, and so on.

Backtracking

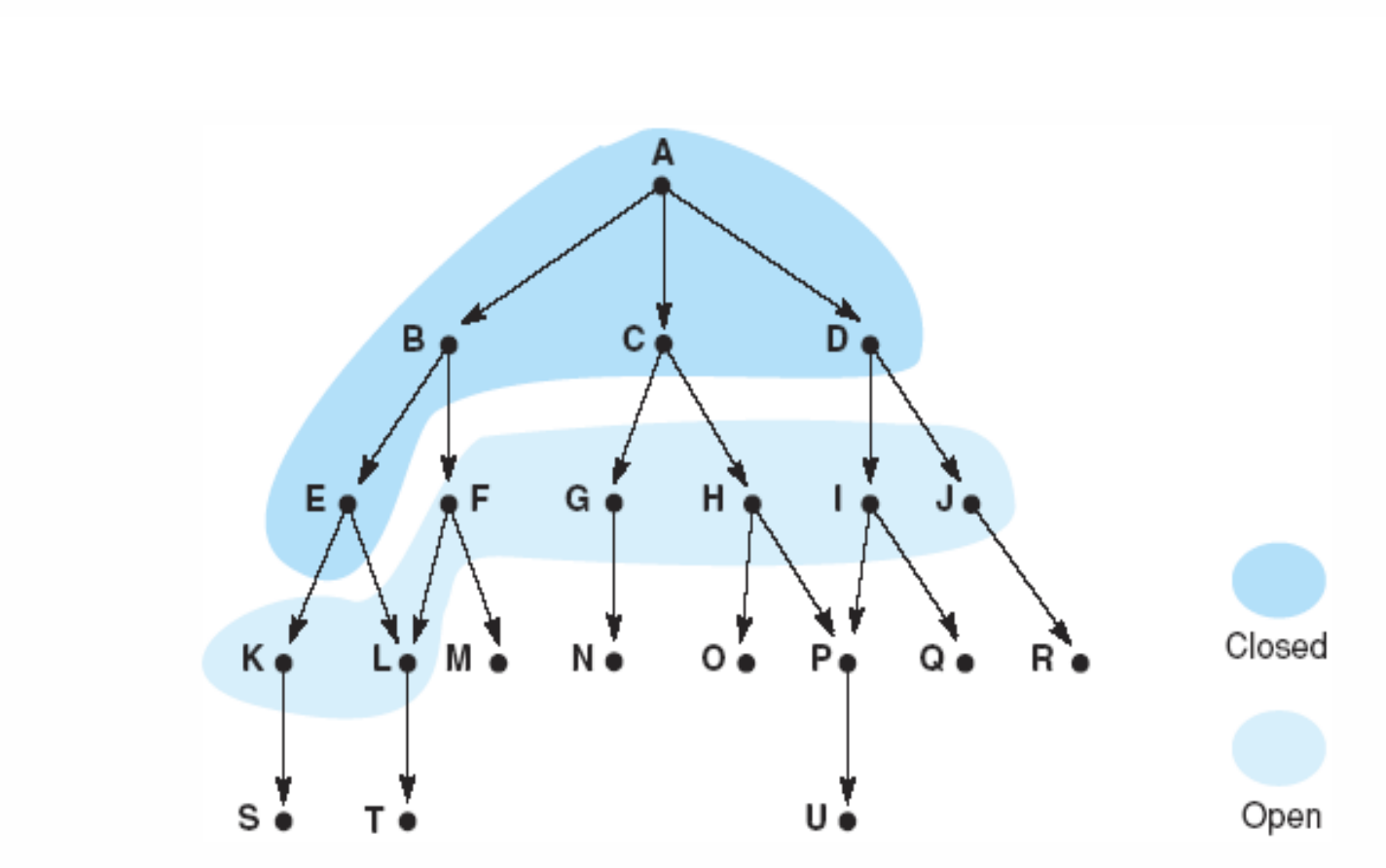
```
Backtracking(P) // Input: problem P
01 F ← {(∅,P)}     // Frontier set of configurations
02 while F ≠ ∅ do
03    Choose (X,Y)∈F – the most "promising" configuration
04    Expand (X,Y), to candidate extensions (new choices)
05    Let (X1,Y1), (X2,Y2), ..., (Xk,Yk) be extended candidates
06    for each new configuration (Xi,Yi) do
07       "Check" (Xi,Yi)
08       if "solution found" then
09          return the solution derived from (Xi,Yi)
10       if not "dead end" then
11          F ← F ∪ {(Xi,Yi)}
         // else nothing to expand from
12 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
01 F ← {(∅,P)}     // Frontier set of configurations
02 while F ≠ ∅ do
03   Choose (X,Y)∈F - the most "promising" configuration
04   Expand (X,Y), to candidate extentsions (new choices)
05   Let (X1,Y1), (X2,Y2), ..., (Xk,Yk) be extended candidates
06   for each new configuration (Xi,Yi) do
07      "Check" (Xi,Yi)
08      if "solution found" then
09         return the solution derived from (Xi,Yi)
10      if not "dead end" then
11         F ← F ∪ {(Xi,Yi)}
        // else nothing to expand from
12 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 $\mathbf{X_i}$, a partial assignment of variables to True or False
- Alternatively, think of each node as the **subproblem $\mathbf{Y_i}$** remaining after applying the partial assignment, i.e. the remaining SAT subformula
  - we consider having **$(\mathbf{X_i}, \mathbf{Y_i})$ where $\mathbf{Y_i}$ is the remaining subproblem after making assignment $\mathbf{X_i}$**

Satisfiability

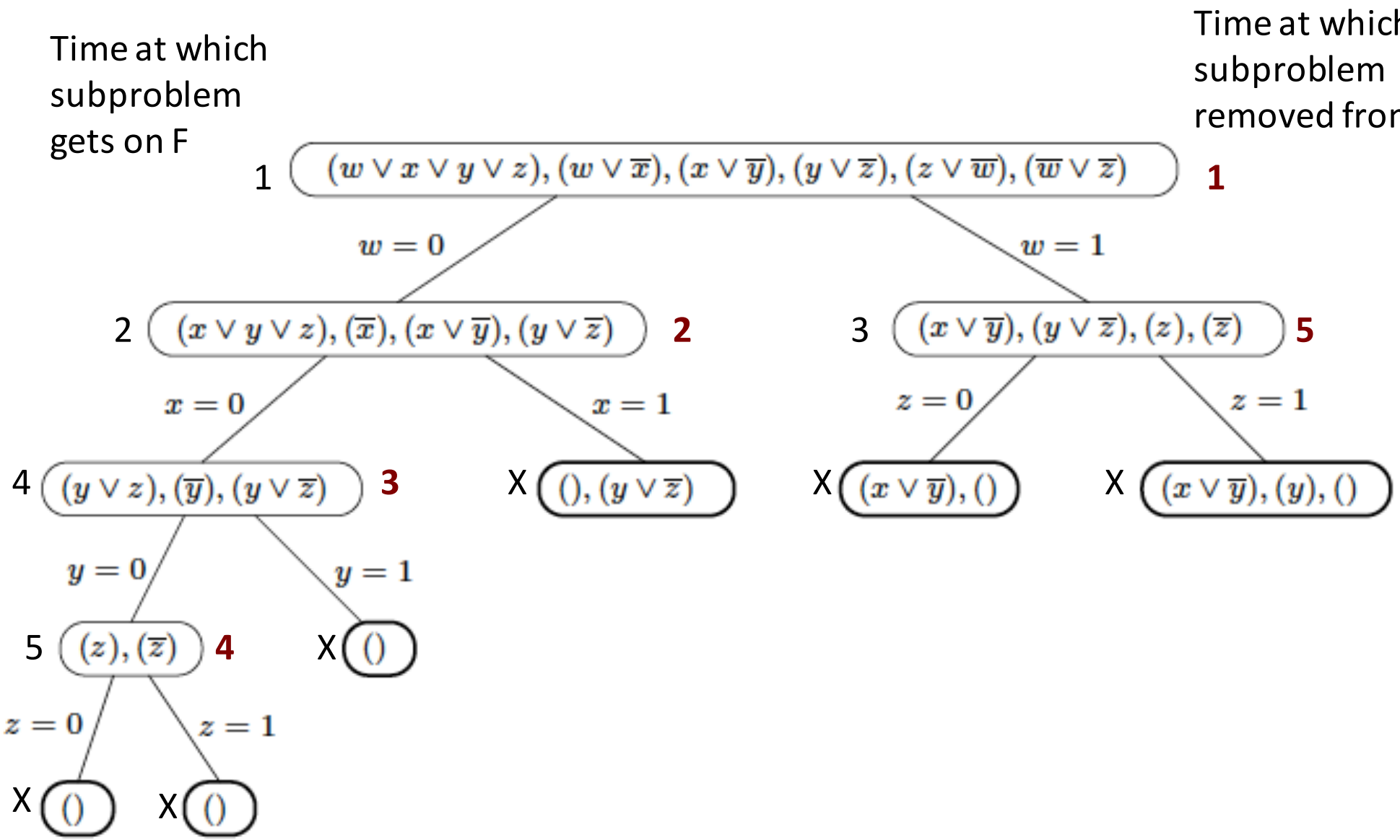
- $\Phi = (w \vee x \vee y \vee z) \wedge (w \vee \neg x) \wedge (x \vee \neg y) \wedge (y \vee \neg z)$
  $\wedge (z \vee \neg w) \wedge (\neg w \vee \neg z)$
- X = {w=True} (partial assignment)
  - *Y* = $(x \vee \neg y) \wedge (y \vee \neg z) \wedge (z) \wedge (\neg z)$ (corresponding subproblem)
- *Check(X,Y)*:
  - 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



Unsatisfiable formula

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 (v1, $\varnothing$, v1)
- 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