Algorithms

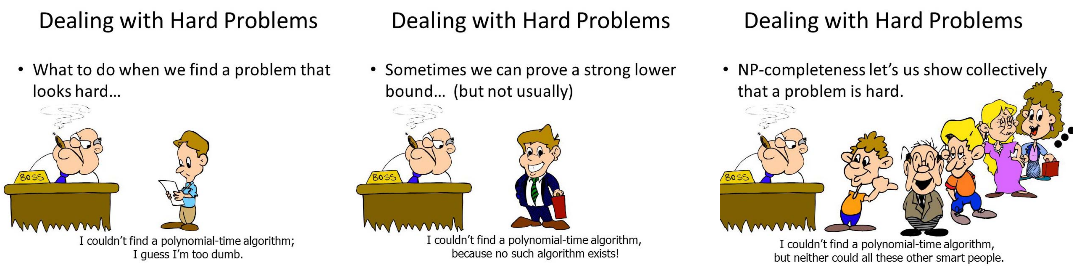
Lecture Notes — Coping with Hard Problems

Imdad ullah Khan

Contents

1 Strategies for Dealing with Hard Problems

Suppose you are tasked with solving some kind of problems in your company. If you are lucky, the problem is solvable using some design paradigm that we have studied so far in this course. In particular, dynamic programming and linear programming are able to take care of many problems. However, you may not be so lucky. Then, you would tell your boss one of the following three things:


Dealing with Hard Problems
• What to do when we find a problem that looks hard…
BOSS
I couldn't find a polynomial-time algorithm; I guess I'm too dumb.
Dealing with Hard Problems
• Sometimes we can prove a strong lower bound… (but not usually)
BOSS
I couldn't find a polynomial-time algorithm, because no such algorithm exists!
Dealing with Hard Problems
• NP-completeness let's us show collectively that a problem is hard.
BOSS
I couldn't find a polynomial-time algorithm, but neither could all these other smart people.

- "I cannot solve the problem, because I am too dumb"
- "The problem is not solvable (in poly-time)". However, you would need a proof, which will actually amount to $P = NP$ (if your problem is in NP). In this case, you would no longer need the job. Simply collect your million dollars from the Clay Institute and enjoy your life.
- "I can not solve the problem but neither can all these extremely smart people." if you can prove that your problem is NP-complete. If you really worked hard to find a solution but your attempts were fruitless, a little more work may lead you to this proof.

The trouble with the above scenarios is that the problem at hand remains and this theoretical exercise does not help practically. Hence, we will now study what to do in such a case.

"NP-Completeness is not a death certificate, it is the beginning of a fascinating adventure"

When you prove a problem to be NP-Complete (or NP-hard), then, as per popular belief that $P \neq NP$, it essentially means that

1. There is no polynomial time

2. deterministic algorithm
3. to exactly solve this problem
4. for all possible input instances

The four keywords impose very strict requirements. Unless our goal is to prove $P = NP$ or $P \neq NP$, regarding which we already assume the latter, then, in practice we may relax one of these requirements to sustain our job. It turns out that relaxing any of these requirements does indeed help a lot practically and opens up huge avenues of possibilities.

So what are our options to deal with NP-Hard problems? Let's consider the following questions:

- Do we need to solve the problem for all valid input instances?
  - Sometimes, we just need to solve a restricted version of the problem that includes realistic instances (special cases)
- Is exponential-time algorithms OK for our instances?
  - The problem with exponential-time algorithms is not primarily that they are "slow" but rather that they don't scale well. So if our relevant instances are small, then exponential-time may be acceptable. Moreover, we can reduce the base or exponent in many practical cases. For example from 2^n to $2^{\sqrt{n}}$ or 1.5^n.
- Is non-optimality acceptable?

  - In some cases, It is OK if our algorithm just outperforms other algorithms. Consider the following scenario: A fit person and a non-fit person are being chased by a bear. The fit person says: "Spending so much time in the gym is worth it." The non-fit person says: "Why? You still won't outrun the bear." The fit person replies: "I don't need to outrun the bear. I just need to outrun you."

Therefore, we may sacrifice one of three desired features i.e. solve any arbitrary instance of the problem, optimally and in polynomial time by designing algorithms that, respectively, solve special cases of the problem, or approximately solve problems, or may take exponential time. We summarize these strategies to cope with the hard requirements of NP-Complete problems by relaxing them in Table **??**.

Poly-time	Deterministic	Exact/Optimal solution	All cases/ Parameters	Algorithmic Paradigm
✓	✓	✓	✗	Special Cases Algorithms Fixed Parameter Tractability
✓	✓	✗	✓	Approximation Algorithms Heuristic Algorithms
✓	✗	$\mathbb{E}$(✓)	✓	Mote Carlo Randomized Algorithm
$\mathbb{E}$(✓)	✗	✓	✓	Las Vegas Randomized Algorithm
✗	✓	✓	✓	Intelligent Exhaustive Search

Table 1: Coping with NP-hard Problems

We briefly describe each of the above listed approaches to tackle hard problems before diving deeper into each one individually.

1. Special cases can be based on the structure of input instances or depend on particular range of one or more parameters. The problem is easier for such special cases, for which exact results are attainable in polynomial time.

2. Approximation algorithms and heuristic algorithms provide nearly exact solutions, i.e. the output is 'close' to the optimal solution. While approximation algorithms output solutions of guaranteed quality in poly-time, heuristics algorithms do not have any guarantees on the solution which is hopefully good in poly-time.

3. Randomized algorithms use coin flips for making decisions. In addition to being used for approximation of hard problems, they are also used for easy problems (in P). Monte Carlo algorithms output solutions which may not be exact but always take polynomial time whereas Las Vegas algorithms always output the optimal solution but not necessarily in polynomial time.

4. Intelligent exhaustive search takes exponential time in the worst case but could be very efficient on typical more realistic instances where the base and/or exponent are usually smaller. Techniques for this approach include Backtracking, Brand-and-Bound, and Local Search.

2 Algorithms for Special Cases

We first see some examples of general NP-complete problems for which there are polynomial time solutions.

- The VERTEX-COVER(G, k) problem is the dual of BIPARTITE-MATCHING when G is a bipartite graph.
- The $3d$ − MATCHING problem is simply the graph matching problem for the $2d$ case.
- The KNAPSACK can be solved in polynomial time if the numbers of items and maximum capacity are polynomial.
- The general SAT(f) is NP-Hard whereas 2 − SAT(f) is easy to solve.
- The WEIGHTED-INDEPENDENT-SET(G) problem can be solved in polynomial time using dynamic programming if G is a tree. As a result, the INDEPENDENT-SET(G) problem (i.e. G has no (or uniform) weights) is also an easy special case if G is a tree.

We discuss how the solve the last two of these special cases in detail.

2.1 2-SAT Search Problem

Given n Boolean variables $x_1, \dots, x_n$, where x_i can take a value of 0 or 1, a literal is a variable appearing in some formula as x_i or $\bar{x}_i$ and a clause of size 2 is an OR of two literals. A 2-CNF formula, which is AND of one or more clauses of size 2 or less, is satisfiable if there is an assignment of 0/1 values to the variables such that the formula evaluates to 1 (or True).

Problem 1 (2-SAT(f) search problem)**.** *: Find satisfying assignment for f if one exists*

For example, $(x \vee y) \wedge (\overline{x} \vee z) \wedge (\overline{z})$ is satisfied with $x = 0, y = 1, z = 0$ but $(x \vee y) \wedge (y) \wedge (\overline{x} \vee z) \wedge (\overline{y})$ is not satisfiable

How can be easily solve the 2-SAT problem? Let's start by analyzing what a clause in a 2-CNF formula actually means.

Evidently, a one literal clause means that the literal must evaluate to True for the clause to be true. For example, (ℓ_1) means ℓ_1 must be true (or it cannot be false). Similarly, for clause $(\ell_1 \vee \ell_2)$ to be true, at least one of ℓ_1 and ℓ_2 must be true or *both cannot be false.* i.e. if $\ell_1 = 0$, then $\ell_2 = 1$ and if $\ell_2 = 0$, then $\ell_1 = 1$. In other words, if $\overline{\ell_1} = 1$, then $\ell_2 = 1$ and if $\overline{\ell_2} = 1$, then $\ell_1 = 1$. A series of implications of this form constitute a 2-CNF formula.

Note that implications are transitive i.e. $[a \implies b \text{ and } b \implies c] \implies (a \implies c)$. For example, consider the formula $(\overline{x} \vee y) \wedge (\overline{y} \vee z)$. From this formula, we get the implications $x = 1 \implies y = 1$ and $y = 1 \implies z = 1$. Together, these implications mean that $x = 1 \implies z = 1$.

To model all the implications we get from the clauses in a 2-CNF formula, we build an **implication graph** which is a digraph $G = (V, E)$ where V are variables of f and their negations (i.e. all possible literals) and E corresponds to the two implications from each clause in the 2-CNF formula. See examples of 2-CNF formulas and the corresponding implication graphs in Figure **??**.

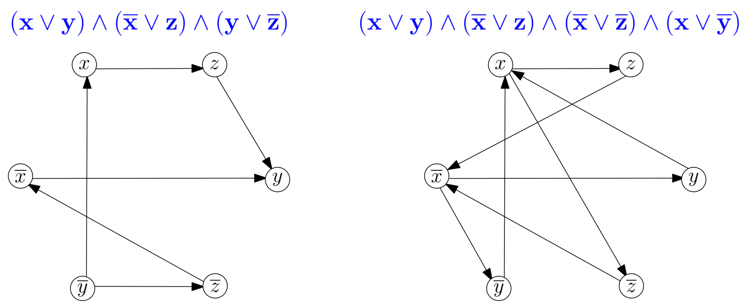

$(\mathbf{x} \vee \mathbf{y}) \wedge (\overline{\mathbf{x}} \vee \mathbf{z}) \wedge (\mathbf{y} \vee \overline{\mathbf{z}})$
$(\mathbf{x} \vee \mathbf{y}) \wedge (\overline{\mathbf{x}} \vee \mathbf{z}) \wedge (\overline{\mathbf{x}} \vee \overline{\mathbf{z}}) \wedge (\mathbf{x} \vee \overline{\mathbf{y}})$
x
z
$\overline{x}$
y
$\overline{y}$
$\overline{z}$

Figure 1: 2-CNF formulas and the corresponding implication graphs

We have already seen that transitive of implications and this property also translates to the implication graph. Another property of implication graphs is *skew symmetry*. Skew symmetry means that if there is an edge from u to v, then there is also an edge from $\overline{v}$ to $\overline{u}$. Skew symmetry follows from construction. For example, there is an edge l_1 to l_2 because of the clause $(\overline{l_1} \vee l_2)$, which also means there must be an edge from $\overline{l_2}$ to $\overline{l_1}$. This also generalizes to path, i.e. if there is a path from $l-1$ to l_2, then there is a path from $\overline{l_2}$ to $\overline{l_1}$.

What is the purpose of these implication graphs? Note that in Figure **??**, the left formula has a satisfying solution whereas the right formula has no satisfying solution. This fact is actually depicted in the implication graphs. We discuss how an implication graph can be used to find a satisfying assignment, if possible, in the corresponding 2-CNF formula.

Satisfying all clauses is equivalent to ensuring that all implications (now edges in G) are true. Since an implication $x \implies y$ is always except for the case when $x = 1$ and $y = 0$. Thus, if all edges are satisfied, this means there is no edge (x, y), with the vertex (literal) x has value 1 and the vertex (literal) y has value 0. See Figure **??** for an example.

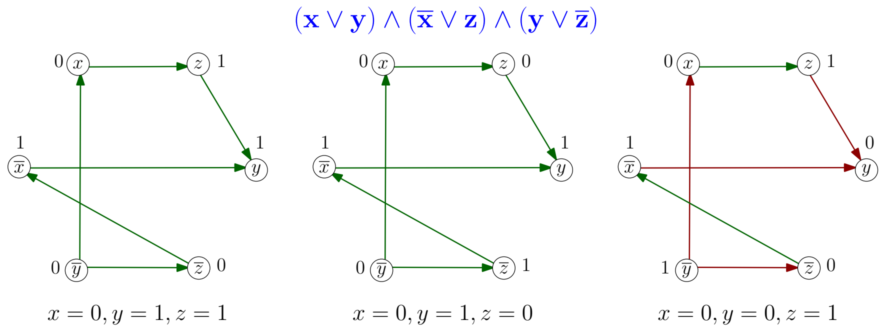
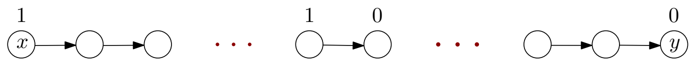

$(\mathbf{x} \vee \mathbf{y}) \wedge (\overline{\mathbf{x}} \vee \mathbf{z}) \wedge (\mathbf{y} \vee \overline{\mathbf{z}})$
$x = 0, y = 1, z = 1$
$x = 0, y = 1, z = 0$
$x = 0, y = 0, z = 1$

Figure 2: For this formula and implication graph, assignments $(x, y, z) = (0, 1, 1)$ and $(x, y, z) = (0, 1, 0)$ satisfy all edges but the assignment $(x, y, z) = (0, 0, 1)$ does not satisfy the red edges

We want an assignment to variables such that there is no edge from 1 to 0 in the corresponding implication graph, i.e. the assignment satisfies the formula. Therefore, we try satisfy all the edges of the implication graph. We can now reword the $2-\text{SAT}$ problem as: Make an implication graph from a formula and find an assignment to vertices that is not-conflicting ($\ell \neq \overline{\ell}$) and all edges are satisfied.



In any assignment that satisfies all edges, there cannot be a 1 to 0 edge or, due to the transitive property of implications, or 1 to 0 path. This means that if there is a path from u to v, we should not assign $u = 1$ and $v = 0$. This works for unidirectional paths, but what if there are bidirectional paths? In the case of bidirectional paths, whenever there is a path from u to v and a path from v to u, then u and v must be assigned the same value. This implies that all literals lying in the same strongly connected components, must be assigned the same value. We can further deduce that if a literal and its negation are lying in the same strongly connected components, then the formula is not satisfiable. In fact, the formula would not be satisfiable only in this case. This observation gives us an algorithm to find a satisfiable assignment, if possible, of a $2-$ CNF formula.

The algorithm for the $2-\text{SAT}$ problem is as follows. We first construct an implication graph from the given formula, find its strongly connected components (SCC) and give each component the same value. Here, the question arises that which component should get 1 which should get 0? Note that the component graph is a DAG. To ensure that there is no path from 1 to 0, we traverse vertices in reverse topological ordering of their SCC's and if literals in current SCC are not assigned, then set all of them to 1 and their negations to 0.

Theorem 1. *If no literal and its negation are in the same components, then the above algorithm produces a valid and satisfying assignment. If a literal and its negation are found in the same component, then the formula can not be satisfied.*

Proof. If a literal is set to 1, then all the literals reachable from it have already been set to 1 and if a literal is set to 0, then all the literals reachable from it have already been set to 0. This is because we are processing literals in reverse topological order which means we have made sure that all 0's are on the left and 1's are on the right. If u is reachable from v, then v's component has already been processed so it must be 1. Suppose for contradiction that $u = 0$, then $\overline{u} = 1$ must have been processed earlier, since this is the only way vertices is labelled 0. Since there is an edge from v to u, by skew symmetry there is an edge from $\overline{u}$ to $\overline{v}$. Because we are processing components in reverse topological order, $\overline{v} = 1$ since it's negation has not yet been processed. However, this results in a contradiction as the same label (1) is assigned to both v nd $\overline{v}$. □

2.2 Max-Independent-Set in Trees

The MAX-INDEPENDENT-SET(G) search problem is NP-HARD, as it can be easily reduced from decision version, but can be solved efficiently when G is acyclic, i..e G is a tree or forest. Figure **??** shows multiple independent sets and a maximum independent set in a tree.

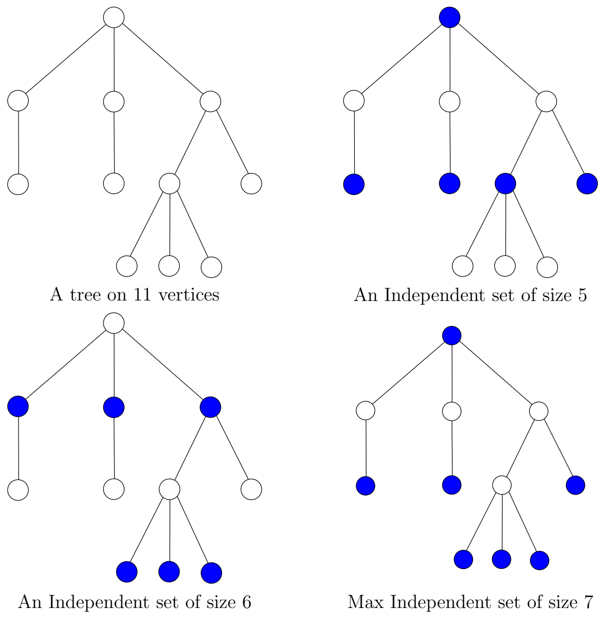

A tree on 11 vertices
An Independent set of size 5
An Independent set of size 6
Max Independent set of size 7

Figure 3: Examples of Independent Sets in a Tree

To solve the MAX-INDEPENDENT-SET(G) search problem easily for acyclic graphs, we use a divide and conquer approach in which subtrees rooted at node u interact with the rest of tree only through u so that there is limited dependence among them. In this way, the u's subtrees problem can be decoupled from the remaining tree.

It is obvious that any tree has at least one leaf (or actually two leaves, if there are only two vertices in the tree).

Theorem 2. *For any leaf u in tree T, there is a maximum independent set containing u*

Proof. Let S be a max independent set not containing u, i.e. $S \not\ni u$, and let v be the only neighbor of u (since u is a leaf and has degree 1. Now as evident in Figure **??**, $v \in S$, otherwise u can be in S contradicting the maximality of S. The validity and size of the max independent set S remains the same if u is exchanged for v to ensure that $u \in S$.

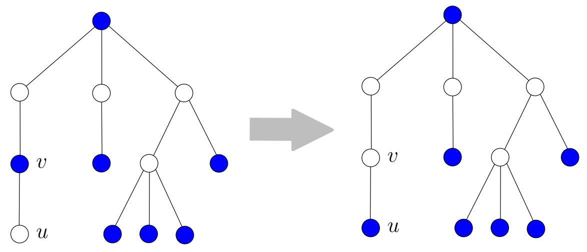


Figure 4: There is a maximum independent set containing all the leaves in T

□

The above theorem consequently means that for any leaf u in T, a max independent set is the union of $\{u\}$ and a max independent set in $T \setminus \{u\}$. We use this fact to design a polynomial time algorithm, outlined in Algorithm **??** to find a maximum independent set in a forest.

Algorithm 1 Max Independent set in *Forest F*

```
$S \leftarrow \emptyset$
while $E(F) \neq \emptyset$ do
    Let $u$ be a leaf and $v$ be its neighbor
    $S \leftarrow S \cup \{u\}$
    Remove $u, v$ from $V(F)$ and all edges incident to $u$ and $v$ from $E(F)$
```

The runtime of the above algorithm is clearly $O(n + m)$.

3 Fixed Parameter Tractability

Sometimes, even the guaranteed sub-optimal solutions such as those given by approximation algorithms may be too expensive. For such cases, we consider the parameterized complexity, which is a measure of complexity with more than 1 input parameters. We want exact algorithms but we allow the running time to be exponential in one parameter but polynomial in the size of the input which constitute other parameters of the problem. For example, we would like algorithms with time complexity of the order $2^k n^2$ or $k! n \log n$. Such problems are called Fixed-Parameter Tractable and the algorithms for such problems are called Fixed-Parameter Tractability (FPT) algorithms.

3.1 Vertex Cover

Recall that a vertex cover in a graph is subset C of vertices such that each edge has at least one endpoint in C.

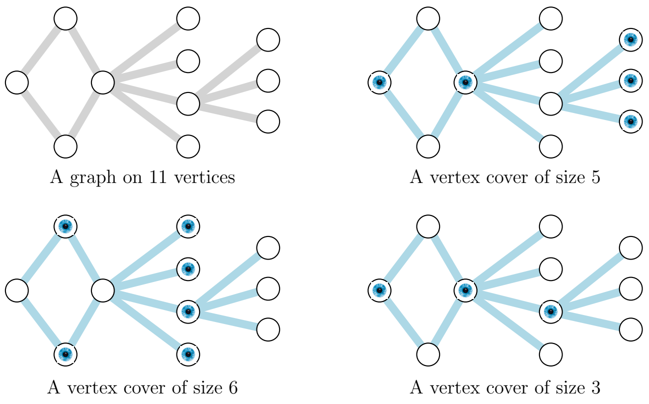


Figure 5: Examples of different vertex covers in a graph

Consider the search version of the following problem:

Problem 2. VERTEX-COVER(G, k) *search problem: Find a vertex cover of size k in G?*

A brute force algorithm for SEARCH-VERTEX-COVER is to check whether each possible k-subset S of V is a vertex cover. To check if S is a vertex cover of $G = (V, E)$, for each $v \in S$, traverse adjacency list of v and count how many edges are contained in S and how many are in $[S, \overline{S}]$. If this count is equal to $|E|$, then S is a vertex cover and otherwise not.

The total runtime of the brute force algorithm is $O(\binom{n}{k}kn) = O(kn^{k+1})$. For a small and fixed value of k, this algorithm is polynomial in n. However, for slightly larger k and large n, this algorithm is impractical. For example, if $n = 10000, k = 20$, the runtime is approximately of the order $\sim 10^{82}$.

We will design a FPT algorithm for SEARCH-VERTEX-COVER with runtime $2^k nk$. Again, for $n = 10000, k = 20$, the runtime of the FPT algorithm would be $2^{20} \times 10000 \times 20 \ll 10^{82}$, which is much more efficient than the brute force algorithm.

The basic idea is to take full advantage of k being small and enumerate all possibilities for some k edges. Note that if we pick an edge (u, v), then for any vertex cover S of size k (which we will call a k-cover), either $u \in S$ or $v \in S$.

For $x \in V$, let $G - \{x\} = (V \setminus \{x\}, E \setminus \{(a, b) \in E : a = x \vee b = x\})$, i.e. $G - \{x\}$ is the graph after removing the vertex x and all edges incident on x.

Theorem 3. *For any edge $(u, v) \in E$, G has a k-cover if and only if $G - \{u\}$ or $G - \{v\}$ has a $k - 1$-cover.*

Proof. If $u \in S$, then it is obvious that $S' = S \setminus \{u\}$ is a $(k - 1)$-cover in $G - \{u\}$. To see the other direction, note that a $(k - 1)$-cover S' in $G - \{u\}$ must cover all edges except those incident on u. Thus, $S' \cup \{u\}$ is a k-cover in G. □

We use this theorem in an FPT algorithm for SEARCH-VERTEX-COVER by recursively trying to find a $(k-1)$-cover in both $G-u$ and $G-v$. The algorithm is outlined in Algorithm **??**.

Algorithm 2 Algorithm to find vertex cover of size k

```
function VERTEX-COVER($G, k$)
    if $k = 0$ then
        if $E(G) = \emptyset$ then                ▷ $O(n)$ time to check if all adj. lists are empty
            return $\emptyset$
        else
            return NF
    else
        $e = (u, v) \in E(G)$                     ▷ Pick an arbitrary edge in $G$
        $S_u \leftarrow$ VERTEX-COVER($G - \{u\}, k-1$)
        $S_v \leftarrow$ VERTEX-COVER($G - \{v\}, k-1$)     ▷ $O(n)$ time to make $G - \{x\}$
        if $S_u \neq$ NF then
            return $S_u \cup \{u\}$
        else if $S_v \neq$ NF then
            return $S_v \cup \{v\}$
        else
            return NF
```

The runtime of this algorithm can be written as a recurrence relation. For an input $G = (V, E)$ with $|V| = n$ and k is the size of the vertex cover to be found, the runtime $T(n, k)$ is:

$$T(n,k) = \begin{cases} O(n) & \text{if } k = 0 \\ 2T(n-1, k-1) + O(n) & \text{if } k > 0 \end{cases}$$

The base case is $O(n)$ to check if all adjacency lists are empty. The recursive case takes $O(n)$ time in removing edges incident to u and v.

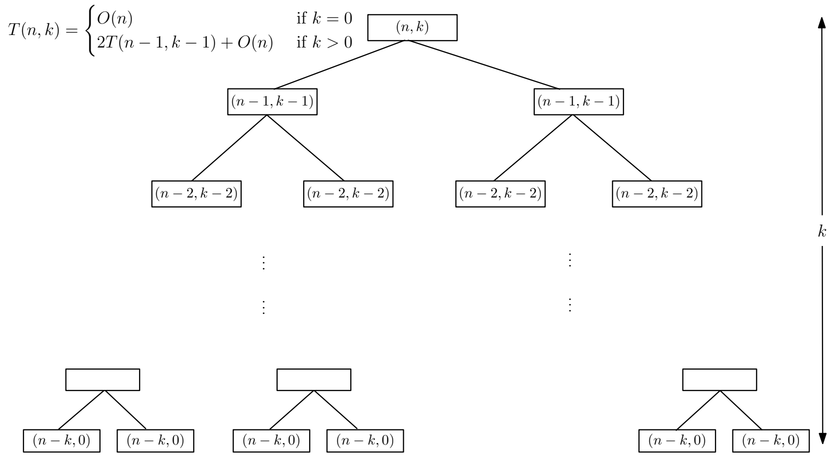

$T(n,k) = \begin{cases} O(n) & \text{if } k = 0 \\ 2T(n-1,k-1) + O(n) & \text{if } k > 0 \end{cases}$
(n,k)
$(n-1,k-1)$
$(n-1,k-1)$
$(n-2,k-2)$
$(n-2,k-2)$
$(n-2,k-2)$
$(n-2,k-2)$
k
$(n-k,0)$

Figure 6: Recursion tree of FPT algorithm to find k-cover in G

As shown in Figure **??**, the recursion tree of the algorithm is a complete binary tree with height k. Since there are 2^k leaves and 2^{k-1} internal nodes where each recursive invocation is at most $O(n)$, the total runtime of the algorithm $T(n,k)$ is $O(2^k n)$, which is polynomial in n but exponential in k.

Note that the optimization version of the MIN-VERTEX-COVER(G) can be solved by using the above algorithm for SEARCH-VERTEX-COVER(G,k) at most n times. This can be done by starting with $k = n$ and repeatedly calling the above algorithm, each time decrementing k by 1 until the algorithm can not find a vertex cover of size k. Then, the minimum vertex cover of G is of size $k + 1$.

4 Intelligent Exhaustive Search

So far, we have seen that, sometimes, specific structures in instances and parameters are helpful to solve the problem in polynomial time. For example, IND-SET for the case of trees, and $2-$ SAT. However, sometimes, even a well-characterized special structure does not help and the problem still may not be solvable in polynomial time. For example, IND-SET is NP-HARD even for planar graphs (graphs that can be drawn in the plane with no edges crossing) and although 3-SAT is a special class of the SAT problem, but is still NP-HARD. Furthermore, in many cases, we may not be able to characterize the particular cases neatly. Does this mean we have no option but to solve the problem in exponential time? Definitely not. We can still, sometimes, avoid exhaustively searching for a solution in exponential time with clever methods. However, often these algorithms are still exponential time in the worst case, but with the right ideas they are very efficient on typical (likely) instances. We study two such methods: Backtracking, and Branch and Bound in the context of $3-$ SAT and TSP respectively.

4.1 Backtracking

Often, a solution to a problem can be made with a series of choices, with each choice representing a partial solution. These partial solutions form a tree (or DAG). **Backtracking** refers to the brute force solution where only feasible partial solutions are considered. Feasibility and in-feasibility of partial solutions are determined given the specific problem at hand. The idea in backtracking is that many partial solutions can be rejected very quickly without completing them (or reading them in full), thus reducing the time taken to find the solution.

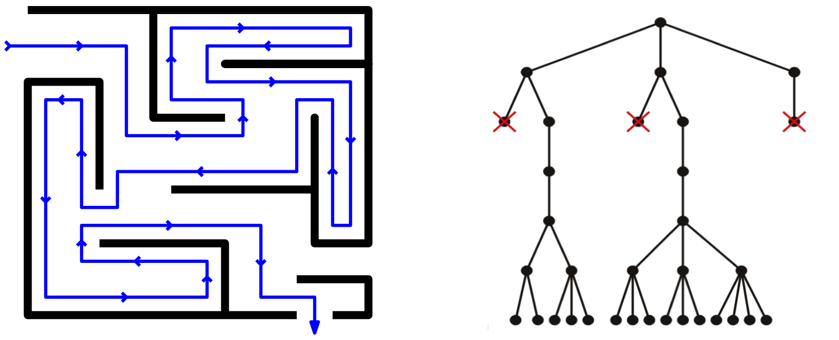


Figure 7: Finding a path in a maze: backtrack when you reach a dead-end

We now see how exhaustive search for SAT can be done.

Given a CNF formula f on n variables and m clauses , the brute force algorithm to find a satisfying assignment for f checks all possible assignment to n variable in $O(m+n)$ time. Since there are 2^n possible unique assignments, the total running time is $O(2^n(n+m))$. This brute force algorithm can be visualized as a complete full binary tree where the root of the tree corresponds to variable x_1 and the left and right branches correspond to values of 1 and 0 for x_1 respectively. The left and right sub-trees contain all possibilities for variables $x_2, \ldots, x_n$.

We can also do the same exhaustive search a bit more intelligently by reducing our search space. Instead of considering all 2^n branches of the binary tree, we carefully track each branch and stop when we reach a dead branch, i.e. a branch that can not be extended. For example, consider a formula of the form $f = (\ldots) \wedge \ldots \wedge (x_6) \wedge \ldots (\ldots)$. We can immediately reject all solutions $(x_1, \ldots, x_n) \in \{0,1\}^n$ with $x_6 = 0$ because they will not satisfy the formula irrespective of the values assigned to all other variables except x_6, since clause (x_6) would not be satisfied. This saves us a lot of time, since out of the 2^n sized search space, we have eliminated 2^{n-1} solutions.

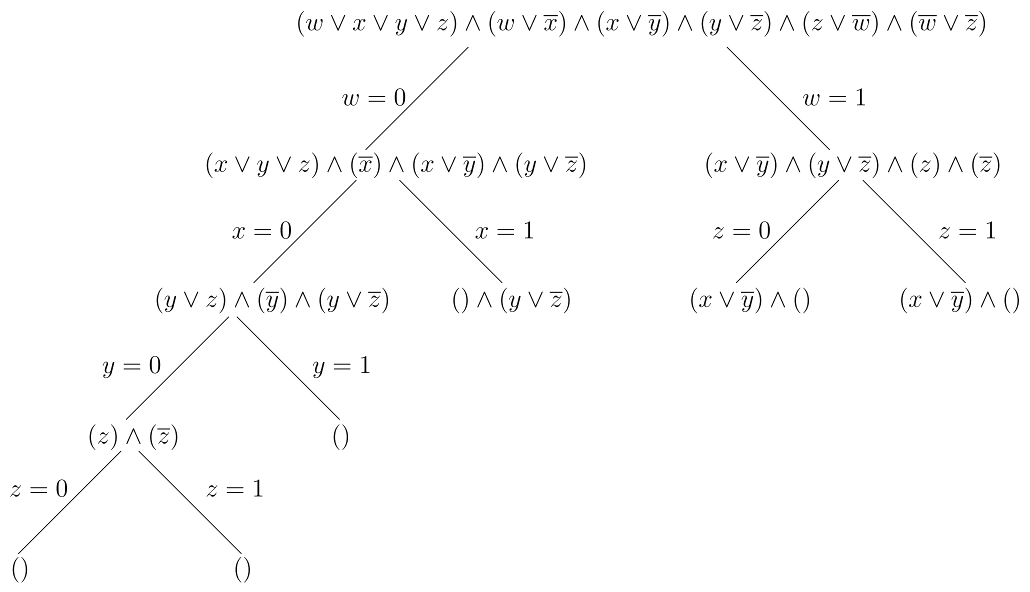

$(w \vee x \vee y \vee z) \wedge (w \vee \overline{x}) \wedge (x \vee \overline{y}) \wedge (y \vee \overline{z}) \wedge (z \vee \overline{w}) \wedge (\overline{w} \vee \overline{z})$
$w = 0$
$w = 1$
$(x \vee y \vee z) \wedge (\overline{x}) \wedge (x \vee \overline{y}) \wedge (y \vee \overline{z})$
$(x \vee \overline{y}) \wedge (y \vee \overline{z}) \wedge (z) \wedge (\overline{z})$
$x = 0$
$x = 1$
$z = 0$
$z = 1$
$(y \vee z) \wedge (\overline{y}) \wedge (y \vee \overline{z})$
$() \wedge (y \vee \overline{z})$
$(x \vee \overline{y}) \wedge ()$
$(x \vee \overline{y}) \wedge ()$
$y = 0$
$y = 1$
$(z) \wedge (\overline{z})$
$()$
$z = 0$
$z = 1$
$()$
$()$

Figure 8: An example of intelligent exhaustive search for SAT

As shown in the more elaborate example above in Figure **??**, we can infer some general rules for intelligent exhaustive search. Firstly, when a literal in a clause is 1, we remove the clause since it is satisfied. Secondly, when a literal in a clause is 0, we remove the variable from the clause since it depends on other literals in it. This means that if a clause becomes empty, none of its literals satisfied it, at any node, then the formula is not satisfiable along this branch of assignment. A partial assignment cannot satisfy the formula if there is an empty clause (no literal is 1). Thus, we do not need to expand this branch any further, and we backtrack out of the branch and prune out the search space along the branch. If (and when) the formula becomes empty at any node in the tree, we do not need to explore further since all clauses have been satisfied. Values of the variables corresponding to branches above are fixed, and the remaining variables take any arbitrary values since the formula is already satisfied.

Another interesting question that arises is, which node should be expanded first? Since we want to prune out as much of the search space as possible, we would like to explore nodes such that the corresponding formula in the sub-problem is likely to result in empty clauses. Therefore, we should choose a sub-problem with the smallest clause and branch on a variable in that clause. This makes sense because if there is a singleton clause then definitely at least one branch would be pruned. However, if there are sub-problems of the same sizes then we should choose the one which is lowest in the tree, hoping that is the closest to a satisfying assignment so chances of getting a partial assignment which successfully satisfies the formula sooner is higher. Several intelligent ways to choose which node to expand further are used practically in many SAT-Solvers.

Backtracking is a technique which systematically exploits the kind of leads we discussed above.

Generally, a backtracking procedure requires a test that looks at a sub-problem and quickly declares one of three outcomes: failure, success, and uncertainty. Failure implies that the sub-problem has no solution whereas success implies that a solution to the sub-problem is found. If the outcome is uncertain, further exploration is needed as it is not yet clear if the sub-problem is a failure or a success. A generic backtracking algorithm for any given instance of a problem is shown in Algorithm **??**.

Algorithm 3 Backtracking procedure for Problem P, Instance I_0

```
$\mathcal{S} \leftarrow \{\mathrm{I}_0\}$
while $\mathcal{S} \neq \emptyset$ do
    Choose a subproblem instance $\mathrm{I} \in \mathcal{S}$
    $\mathcal{S} \leftarrow \mathcal{S} \setminus \{\mathrm{I}\}$
    EXPAND I into $\{\mathrm{I}_1, \mathrm{I}_2, \ldots, \mathrm{I}_k\}$
    for each $\mathrm{I}_j$ do
        if TEST($\mathrm{P}_j$) = SUCCESS then
            return the current solution
        else if TEST($\mathrm{P}_j$) = FAILURE then
            return NF
        else
            $\mathcal{S} \leftarrow \mathcal{S} \cup \{\mathrm{I}_j\}$
return NF
```

We have discussed a brute-force search algorithm for SAT that takes $O(2^n(n+m))$ time and also seen a variable centric approach to intelligent exhaustive search for SAT problem. Now, we consider a clause centric approach to see how backtracking can be used for the $3-\mathrm{SAT}$ problem.

A 3-CNF formula $f = l_1 \wedge l_2 \wedge l_3$ can be thought of as $(l_1 \vee l_2 \vee l_3) \wedge (f')$ unless f is empty. Note that f' too is a (possibly empty) 3-CNF formula. Applying the distributive law on this representation of f, we get that $f = (l_1 \wedge f') \vee (l_2 \wedge f') \vee (l_3 \wedge f')$. Note that these three disjunctions are not $3-$ CNF formulae. This gives us the backtracking algorithm shown in Algorithm **??**. Let $f[x = \textbf{true}]$ denote f with the value of x plugged in as **true**.

Algorithm 4 Backtracking for 3-SAT

```
function CHECK-SAT($f$)
    if $f$ is empty then
        return true
    else
        Let $f = (\ell_1 \vee \ell_2 \vee \ell_3) \wedge (f')$
        if CHECK-SAT($f'[\ell_1 = \textbf{true}]$) then        ▷ implies $l_1 \wedge f' = \textbf{true}$
            return true
        if CHECK-SAT($f'[\ell_2 = \textbf{true}]$) then
            return true
        if CHECK-SAT($f'[\ell_3 = \textbf{true}]$) then
            return true
        return false
```

Let $T(n)$ be the runtime of this algorithm for a formula on n variables. Since, in each step, the number of variables are reduced by at least one as one literal is fixed. Thus,

$$T(n) = \begin{cases} 3T(n-1) + O(poly(n,m)) & \text{if } n \geq 1 \\ 1 & \text{otherwise} \end{cases}$$

A simple recursion tree expansion or substitution reveals that $T(n) = O(3^n \cdot poly(n,m))$. This is even worse that the variable centric brute-force search. However, we observe that, similar to dynamic programming, sub-problems are overlapping and result in unnecessary repetitions. Therefore, we need to make the sub-problems mutually exclusive. Since every satisfying assignment found by the above algorithm satisfies the clause $(\ell_1 \vee \ell_2 \vee \ell_3)$, the assignment must be exactly one of the following types: $l_1 = \textbf{true}$ or $l_1 = \textbf{false} \wedge l_2 = \textbf{true}$ or $l_1 = \textbf{false} \wedge l_2 = \textbf{false} \wedge l_3 = \textbf{true}$. We can, thus, pin point any of these three types of satisfying type assignments to three literals in exactly one of the recursive calls. The modified clause centric algorithm based on this idea is shown in Algorithm **??**

Algorithm 5 Backtracking for 3-SAT

```
function CHECK-SAT($f$)
    if $f$ is empty then
        return true
    else
        Let $f = (\ell_1 \vee \ell_2 \vee \ell_3) \wedge (f')$
        if CHECK-SAT($f'[\ell_1 = \textbf{true}]$) then
            return true
        if CHECK-SAT($f'[\ell_1 = \textbf{false} \wedge \ell_2 = \textbf{true}]$) then
            return true
        if CHECK-SAT($f'[\ell_1 = \textbf{false} \wedge \ell_2 = \textbf{false} \wedge \ell_2 = \textbf{true}]$) then
            return true
        return false
```

The runtime of the modified algorithm is reduced by k when values of k literals are fixed in the formula. Thus, $T(n)$ now is

$$T(n) = \begin{cases} T(n-1) + T(n-2) + T(n-3) + O(poly(n,m)) & n \geq 1 \\ 1 & \text{else} \end{cases}$$

The closed form of this recurrence is $T(n) = O(1.84^n)$, which is substantially faster than the $O(2^n)$ algorithm. For example, even for $n \sim 100$, the modified algorithm is 4180 times faster.

4.2 Branch and Bound

Branch and Bound is a technique for optimization problems which is analogous to backtracking for search problems. In fact, branch-and-bound can be viewed as a generalization of backtrakcing for optimization problems. For our discussion, we consider a minimization problem. Maximization problems would follow the same pattern.

In this method, we find the best way to build the solution incrementally from partial solutions of sub-problems, rejecting a partial solution that will not lead to an optimal solution. As in backtracking, we need a basis for eliminating partial solutions. For minimization problems, in order to reject a partial solution, we must be certain that its cost exceeds that of some known solution. Generally, the optimal cost is not known or cannot be computed efficiently because otherwise the problem would already be solved. Instead, we use a quick lower bound on this cost.

Recall that, given a complete graph G on n vertices with edge weights, a TSP tour is a Hamiltonian cycle in G, as shown in Figure **??**.

Problem 3 (TSP(G) optimization problem:)**.** *Given a complete graph G on n vertices with edge weights, find the minimum cost Hamiltonian cycle in G .*

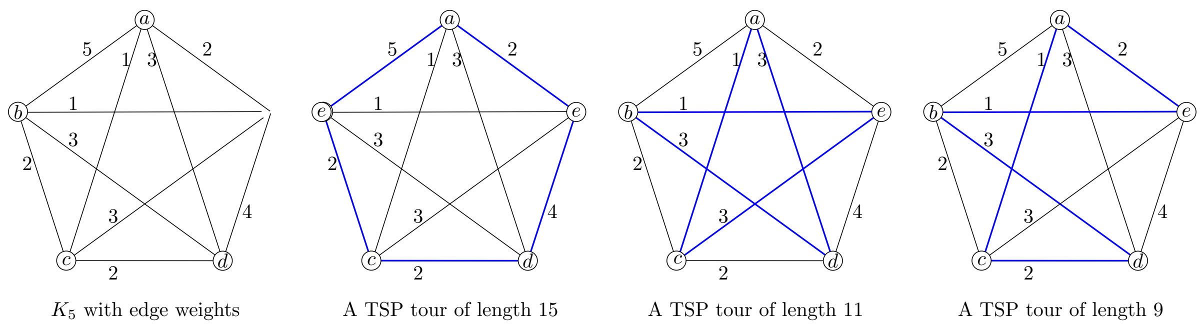


Figure 9: TSP tours of different lengths (costs)

he naive brute-force algorithm checks all possible $(n-1)!$ cycles in G, shown as a tree in Figure **??**

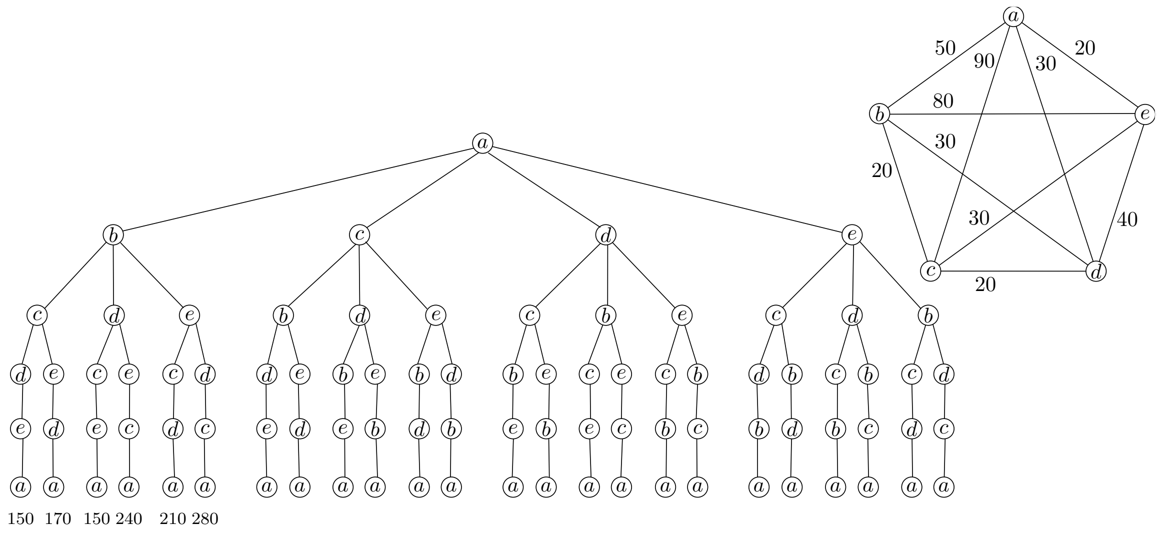


Figure 10: All possible cycles in a complete graph. The cost of the cycle is mentioned for the first six cycles on the left.

Rather than enumerating all possible cycles, we use the branch-and-bound technique which is to grow a tree of partial solutions, i.e. pieces of Hamiltonian paths, and at each tree node, we check if the extension of the current partial solution could possibly be better than the best known solution so far. If not, we do not continue exploring the branch further because we already have a better solution (with lower cost for minimization problem). Figure **??** shows how branch and bound reduces the search space on the concrete example in Figure **??**.

For bounding a branch, values other than the minimum cost solution so far can also be used. For example, another lower bound (that is tighter than ≥ 0) on the cost of the optimal TSP

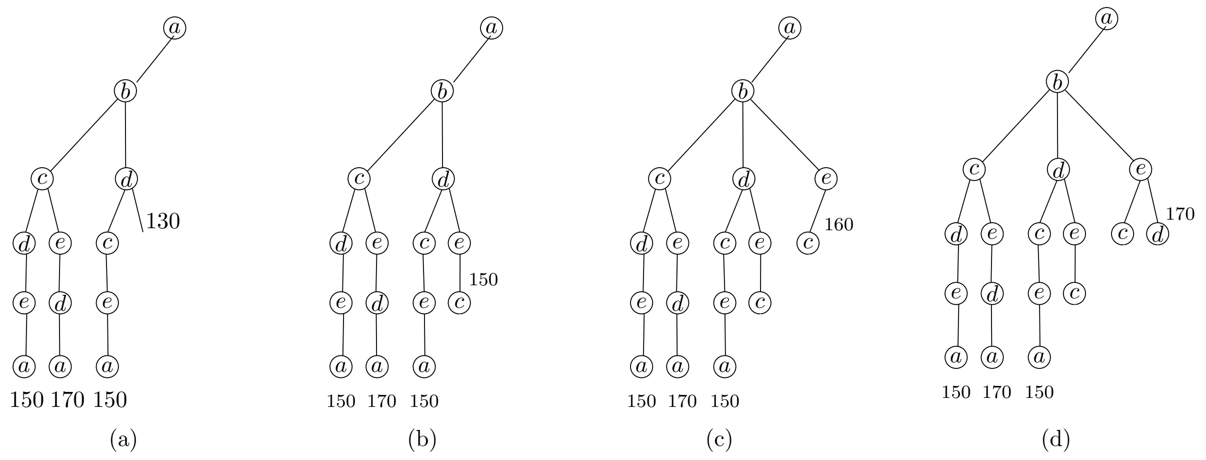


Figure 11: (a): Since the cost of the current partial solution is currently lower than the minimum cost solution so far, we continue to explore the branch. (b), (c) & (d): Since the cost of the current partial solution is now already higher than the minimum cost solution so far, we do not further explore the branch

tour using a subset of vertices $S \subset V$ using is the cost of the Minimum Spanning Tree (MST) in $G[S]$, i.e. $\frac{1}{2}\sum_{v \in S}$(two minimum length edges incidents on v).

4.3 Pseudo Polynomial Algorithm for TSP

Recall that dynamic programming, a more general and powerful technique than divide-and-conquer, involves breaking up a problem into sub-problems, which may be overlapping or independent. Solutions using dynamic programming require us to identify the optimal sub-structure, which is when optimal solution to a problem is made of optimal solution to smaller sub-problems. We build solution to larger sub-problems, identify and avoid redundancy and repetitions using memorization.

Dynamic programming can be used to build pseudo polynomial algorithm for the TSP problem. It was devised by Bellman,and Held and Karp in 1962. So how is the sub-problem defined?

Note that if we can find the Hamiltonian path, we can easily also obtain the Hamiltonian cycle, since we get the path by breaking the cycle at some vertex. A good starting point is to construct some sub-path of a cycle. Since every vertex is included in the Hamiltonian cycle, we can assume without loss of generality that the Hamiltonian path starts at a fixed vertex v_0. The partial solution is to find the initial part of the path/tour, for which we need to know the last vertex of the partial solution and all vertices in between which are part of the partial solution. As shown in Figure **??**, for $S \subset V$, let $C(v_i, S)$ be the min cost path from $v_0 \in S$ to $v_i \in S$ that visits all and only vertices in S once.

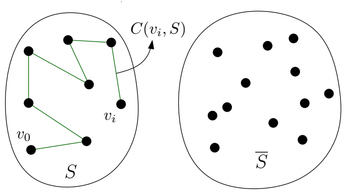


Figure 12: A partial solution (sub-path) of a Hamiltonian Cycle

For some $v_i \neq v_0$, $C(v_i, V)$ is an optimal Hamiltonian path of G. As discussed above, adding the edge (v_i, v_0) to this path gives a Ham cycle in G. Initially, $S = \{v_0\}$ and $C(v_0, S)$ is the empty path with cost 0. We gradually increase S to get optimal Hamiltonian path in G. Note that for $S = \{v_0\}$ and $i > 0$, $C(v_i, S)$ is not defined. We analyze the structure of the path $C(v_i, S)$, without actually knowing it.

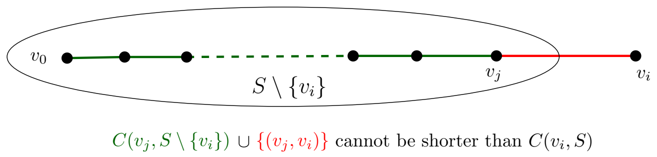

v_0
v_j
v_i
$S \setminus \{v_i\}$
$C(v_j, S \setminus \{v_i\}) \cup \{(v_j, v_i)\}$ cannot be shorter than $C(v_i, S)$

Figure 13: The structure of an optimal sub-path $C(v_i, S)$ in the Hamiltonian path

As shown in Figure **??**, let $v_j \in S$ be the second to last vertex in $C(v_i, S)$. Then, $C(v_j, S \setminus \{v_i\})$ is the min cost path from v_0 to v_j, because otherwise $C(v_i, S)$ would not be optimal. Figure **??** shows all possible Hamiltonian cycles for an example graph, and highlights the optimal cycles found using this algorithm.

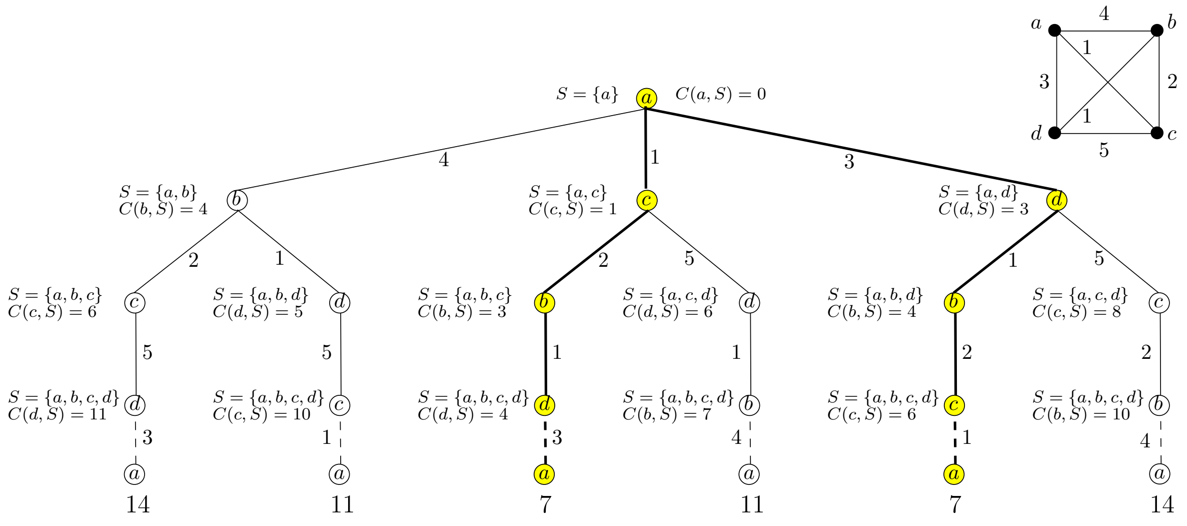


Figure 14: All possible Hamiltonian cycles for a sample graph using a dynamic programming formulation. The optimal cycles are highlighted.

We can formulate the following recurrence relation for $C(v_i, S)$:

$$C(v_i, S) = \begin{cases} 0 & \text{if } S = \{v_0\} \\ +\infty & \text{else if } v_i \notin S \vee i = 0 \\ \min\limits_{v_{j \neq i} \in S} \{C(v_j, S \setminus \{v_i\}) + w(v_j, v_i)\} & \text{else} \end{cases}$$

Then, $\textsc{tsp}(G) \min_{v_i \in V} \{C(v_i, V) + w(v_i, v_0)\}$.

To analyze the runtime, note that there are $2^n - 1$ possible $S \subset V$ and up to $n - 1$ options for the end-vertex v_i. Each of the $n \times 2^n$ sub-problems can be solved in $\mathcal{O}(n)$. Therefore, the total runtime of the dynamic programming solution for TSP is $\mathcal{O}(n^2 2^n)$, which is less than the $\mathcal{O}(n!)$ runtime of brute force solution. However, the space complexity of the dynamic programming solution is higher than that of the brute force solution, i.e. $\mathcal{O}(n2^n)$ and $\mathcal{O}(n^2)$, respectively.

Note that the formulation above only gives us the cost of the optimal Hamiltonian cycle. The actual Hamiltonian cycle can be found by backtracking in $\mathcal{O}(n^2)$. If previous vertex in a sub-path (selected v_j with min cost) is stored for each step during memoization, then backtracking can be done in $\mathcal{O}(n)$.

So far, we have discussed coping with NP-HARD problems by solving for special cases of the problems, dealing with fixed parameter tractable problems and using techniques for intelligent exhaustive search. Next, we will study the approximation and randomized algorithms designed for popular NP-HARD problems.