ALG2 Assignment 4

Answer the questions in the document!
Don't just give the final answer but show how you got to that answer. If you are solving problems on paper, you may include (readable) pictures of this as well.

1. Imagine we have an algorithm with $2^n * n^2$ time complexity and a machine that can handle inputs of size 30 but not larger. Now we have found a 10 times faster algorithm for the same problem, thus with complexity $2^n n^2 /10$. What is the maximal input size that we can handle with this new algorithm? And with a 100 times faster algorithm?

   The maximal input size for the 10 times faster algorithm is approximately n=33
   The maximal input size for the 100 times faster algorithm is approximately n=36

2. Under which condition a "brute force" algorithm is still acceptable:
   a. Not much time to design and program another algorithm (Yes/Maybe/No)
   b. Instances are small (e.g. <20) (Yes/Maybe/No)
   c. Need to solve the problem only once (Yes/Maybe/No)

3. For n=50 what is the speedup of an algorithm with complexity $2^{\sqrt{n}}$ vs 2^n
   a. more than million
   b. more than a billion
   c. more than a trillion

4. For n=50 what is the speedup of an algorithm with complexity 1.1^n vs 2^n
   a. more than million
   b. more than a billion
   c. more than a trillion

5. You have discussed a "bounded search tree" method for finding a vertex cover of a graph by which in each iteration an edge e with vertices A and B is selected and the tree then "branches" in three further cases: 1. A is in the vertex cover and B is not, 2. B is in the vertex cover but A is not, 3. Both A and B are in the vertex cover. Reason about the following statements:
   a. the search tree based algorithm always finds the smallest vertex cover True
      The search tree method explores all possible ways to include the vertices in the vertex cover by branching on each edge and considering all combinations of including its endpoints in the cover. This exhaustive search guarantees that the smallest vertex cover will be found because it considers all possible vertex cover sets.

   b. in some special cases the search tree will not give the correct answer False
      Given the exhaustive nature of the search tree method, it will always find the correct vertex cover, including the smallest one. There are no special cases where it fails to find the correct answer because it systematically explores all possible vertex covers.

c. at each level of the search tree it determines the assignment of at least one vertex
   True
   Each branching step in the search tree corresponds to a decision about the inclusion or exclusion of one of the endpoints of an edge in the vertex cover. Therefore, at each level of the tree, at least one vertex's status is determined (either included or excluded from the vertex cover).

d. it is possible to construct a search tree that determines the assignments of at least two vertices at each level True
   It is possible to enhance the search tree method to determine the assignments of two vertices at each level by considering pairs of vertices or edges. For example, instead of branching on a single edge, the algorithm could branch on two edges simultaneously, thus determining the status of both pairs of vertices at each level. This approach could potentially reduce the depth of the search tree, although it might increase the branching factor.

6. In the same search tree method as in question 6, assume that the assignment of at least 2 vertices is determined in each level of the tree, starting with 0 assignment at the root of the tree. The level of the root is 0 and the level of the leaves of the tree is n/2. In every leaf of the tree all vertices have been assigned (in the vertex cover or not), thus each leaf is a possible vertex cover for the given graph. How many leaves, and thus different assignments for the vertices of the graph, are at the last level of the search tree?
   a. 2^n
   b. $3^{n/2}$
   c. 3^n
   d. 2^{3n}
   e. $2^{n/2} \cdot 3$

7. You are thinking about developing a "bounded search tree" algorithm for independent set of a graph. As usual every vertex is either in the set (is assigned "1") or is not in the set (is assigned "0"). For an edge e with vertices A and B you are reasoning what a valid assignments of A and B could be with respect to an independent set. Which of the following is true:
   a. At most one of A and B can be assigned "1" True
   b. Both A and B can be assigned "1"
   c. At least one of A and B must be assigned "1"
   d. Both A and B could be assigned "0" True
   e. Both A and B must be assigned "0"

8. What is the complexity of the best known (exact) algorithms for maximum independent set in a (general) graph? Are there polynomial algorithms for some special cases? Please add reference to the sources you use for this question. chatGPT

The complexity of the best-known exact algorithms for the maximum independent set (MIS) problem in general graphs is quite high due to its NP-hard nature. For general graphs, the problem remains challenging and is typically approached with exponential-time algorithms. Specifically, there are algorithms that can solve the problem in time $O(1.1996^n)$ using polynomial space, which represents one of the more efficient known approaches as of the latest updates.

For special cases, the complexity can drastically reduce. For example:

- In graphs with a maximum degree of 3, the problem can be solved in $O(1.0836^n)$, indicating a slightly better time complexity for these constrained graph types.
- Polynomial-time algorithms are available for particular graph classes such as bipartite graphs, where the problem can be solved using bipartite matching algorithms. This is feasible due to properties that simplify the identification of independent sets in these structures.

Moreover, for other special classes like claw-free graphs, P_5-free graphs, perfect graphs, and chordal graphs, polynomial-time solutions exist, some of which can be implemented even in linear time.

Regarding approximation, while exact solutions for general graphs are costly, there are efficient approximation algorithms for restricted classes of graphs. For example, planar graphs can be approximated within any ratio less than 1 in polynomial time, and similar polynomial-time approximation schemes exist for any family of graphs closed under taking minors.

For a detailed understanding of these complexities and algorithm specifics, you might find comprehensive discussions and formal proofs in dedicated graph theory texts or algorithm research papers.

These complexities highlight the challenges and nuances of solving the maximum independent set problem across different types of graphs and demonstrate the ongoing advancements in algorithm research aimed at tackling this NP-hard problem.

9. Imagine you have a computer that can handle at most 1 billion different solutions (different assignments of the vertices in a graph). Now consider your answers to question 10. What is the maximum size (n, where n is the number of vertices) your computer can handle with this algorithm? Imagine you have an algorithm for an independent set with a time complexity of 1.1^n. What is the maximum size you can handle with this algorithm? 217

10. Indicate which vertices can be removed in the following graph with pre-processing. is the graph missing?

11. Based on the SAT pruning rules discussed in class, show how we can pre-process the following SAT formulas (was not present due to illness):

    a. $C = \left(x_1 \vee \overline{x_2} \vee x_3\right) \wedge \left(\overline{x_1} \vee x_1\right) \wedge \left(\overline{x_2} \vee x_3\right) \wedge \left(x_4 \vee \overline{x_5} \vee x_3\right) \wedge (x_1 \vee x_2 \vee \overline{x_5} \vee x_3)$

    b. $C = (x_1 \vee x_2) \wedge (x_4) \wedge (\overline{x_5} \vee x_2 \vee \overline{x_1} \vee x_6) \wedge (\vee \overline{x_1} \vee \overline{x_3}) \wedge (x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (\overline{x_6} \vee x_7)$

    c. $C = (\overline{x_3} \vee x_4 \vee x_2) \wedge (x_5 \vee \overline{x_2} \vee x_6) \wedge (\overline{x_4} \vee x_1 \vee \overline{x_6}) \wedge (x_2 \vee \overline{x_2}) \wedge (x_1) \wedge (x_1 \vee \overline{x_4} \vee \overline{x_2} \vee x_2 \vee$