Team Projects – Topics

Michal Tomasz Godziszewski

October 2025

1 Graph Pathfinding and Shortest Path Algorithms

Design and implement an optimal pathfinding system for graph-based environments. Develop a solution to find shortest paths in a graph (such as a maze, grid map, or network) between specified start and goal nodes. The problem requires exploring graph search algorithms that guarantee optimal solutions and analyzing their efficiency. You must implement classical algorithms (e.g., breadth-first search, Dijkstra's algorithm) as well as heuristic search (A^* algorithm) to handle weighted graphs and large search spaces. You will have to rigorously reason about graph properties, ensure algorithmic correctness, and compare the complexity and optimality conditions of each approach (for example, BFS always finds shortest paths in an unweighted graph, Dijkstra's finds shortest paths in weighted graphs, and A^* finds optimal paths given an admissible heuristic). The final system should optionally include a visualization or interactive component (e.g., displaying the explored nodes and the resulting shortest path on a grid), to demonstrate the algorithm's behavior.

1.1 Required Components and Algorithms:

1. Graph Representation: Design a suitable data structure to represent the graph (e.g. adjacency list for a general graph or a grid matrix for a maze). The representation should efficiently support the required operations (neighbor expansion, weight lookup, etc.).
2. Shortest Path Algorithms: Implement the following algorithms in C++:
   (a) Breadth-First Search (BFS) for unweighted graphs (ensuring shortest paths in terms of edge count).
   (b) Dijkstra's Algorithm for weighted graphs, computing minimum-distance paths
   (c) A^* Search Algorithm with an admissible heuristic (such as Euclidean or Manhattan distance in a grid) to accelerate search. The A^* implementation must be proven to yield optimal paths under the chosen heuristic.

3. Algorithmic Analysis: Include functionality or documentation to measure and compare the performance of these algorithms on various graph sizes and densities. Analyze time complexity and memory usage for each approach on sample inputs.

4. Optional Enhancements: Teams are encouraged to add features such as interactive visualization of the search process (e.g., a GUI showing the graph, visited nodes, and the final path), or support for dynamic obstacles (updating paths in real-time). Such additions may involve additional algorithms (e.g. incremental pathfinding or dynamic graph algorithms) and will earn extra credit if implemented correctly.

5. Testing Suite: Develop a set of automated tests (using sample graphs and mazes) to verify correctness of path results and robustness of the implementation (including edge cases like disconnected graphs or multiple goals).

1.2 Final Deliverables:

1. Complete Source Code: A well-structured C++ codebase in a repository (Git is encouraged - with commit history evidencing contributions from all team members). The code should be thoroughly documented and include instructions for compilation and execution.

2. Project Report: A formal report detailing the problem, the mathematical and algorithmic techniques used, design decisions, and an analysis of results. It should include explanations of how each algorithm works, proofs or arguments of correctness (e.g. why A^* is optimal with an admissible heuristic), complexity analysis, and a discussion of empirical performance results.

3. Demonstration and User Guide: If a visualization/interface is implemented, provide a short user manual or demo video illustrating how to use the program to load a graph or maze and visualize the pathfinding process. If console-based, provide example input files and corresponding outputs. The clarity and usability of the system will be part of the evaluation.

4. Git Repository (optional): The Git repository (if Git is used for the project) must be shared, containing all code, documentation, and a clear commit log. Proper use of version control (frequent commits with meaningful messages, use of branches for features, and merge commits) will be assessed.

1.3 Language and Tools Requirements:

- Programming Language: C++ is strongly recommended for all implementation work, due to its efficiency and fine-grained control over memory and

performance. The project should make use of standard libraries and efficient data structures for implementing algorithms like Dijkstra's (priority queue). While other compiled languages (such as Java) may be permitted with instructor's approval, Python is explicitly discouraged due to its performance limitations for heavy algorithmic computation in this project.

- Development Tools: A modern C++17 (or later) compiler should be used. The use of a debugger and memory-checking tools (e.g., gdb, Valgrind) is encouraged to ensure correctness. If a visualization component is included, libraries such as SDL or SFML (for graphics) or a web-based visualization can be used.

2 Combinatorial Optimization – Traveling Salesman Problem Solver

Solve the following combinatorial optimization challenge: the Traveling Salesman Problem (TSP). The task is to develop a program that finds efficient routes for a "salesman" who must visit a set of cities and return to the start. Formally, given n cities and pairwise distances (or costs) between them, the goal is to determine the shortest possible tour that visits each city exactly once and returns to the origin. The TSP is a classic NP-hard problem, meaning no known polynomial-time algorithm can solve all instances optimally. Implement algorithms that find optimal solutions for small instances (using exhaustive search or dynamic programming) and heuristic algorithms that produce near-optimal solutions for larger instances. The mathematical component involves understanding the exponential growth of solution space and the trade-offs between accuracy and runtime. You must analyze and prove properties of their algorithms (correctness and complexity) and possibly explore optimization techniques like pruning and approximation. A successful project will result in a working TSP solver that can demonstrate the optimal tour on moderate-sized inputs and provide insight into the efficiency of different algorithms.

2.1 Required Components and Algorithms:

1. Problem Modeling: Define a clear representation for the TSP. Cities can be represented by indices or coordinates (for Euclidean instances), and distances can be given by a matrix or computed via a distance function. Ensure the data structures chosen (e.g., a distance matrix or adjacency list of weighted edges) allow constant-time distance queries.

2. Exact Solution Algorithm: Implement an exact TSP solver to guarantee optimality on smaller instances:

   (a) Brute Force Search: As a baseline, implement a brute force approach that checks all permutations of cities to find the shortest tour. This will have factorial complexity and is only feasible for very small n, but serves to validate other methods.

   (b) Dynamic Programming (Held-Karp): Implement the Held-Karp algorithm using bitmask DP. While exponential, it is a significant improvement over brute force and can optimally solve instances up to 10-20 cities. Use this to handle medium-sized cases and to verify heuristics on smaller inputs.

3. Optimization Techniques: If possible, enhance the exact solver with branch-and-bound or pruning strategies to cut off portions of the search space by using lower bound estimates on tour lengths.

4. Heuristic or Approximate Algorithms: Implement one or more heuristic algorithms to handle larger instances (where exact methods are infeasible):

(a) Greedy Heuristic (Nearest Neighbor): Construct a tour by repeatedly visiting the nearest unvisited city. This runs quickly (polynomial time) and often yields a decent tour, though not always optimal.

(b) Tour Improvement Heuristic (2-opt or 3-opt): Implement a local search that iteratively improves an existing tour by eliminating crossing paths (2-opt swap) or other re-arrangements, until no better tour is found. This can significantly improve upon the initial greedy solution.

(c) Alternate Heuristic (Optional): You may also implement more advanced heuristics such as Christofides' algorithm for metric TSP (which guarantees at most 50% above optimal) or metaheuristics like Simulated Annealing or Genetic Algorithms for extra exploration.

5. Result Visualization (Optional): If the input cities have coordinates (e.g., points on a plane), provide a visualization of the tour path. This could be done by outputting an image or using a simple GUI plotting the cities and drawing lines for the tour. Visualization helps in understanding the tour structure and verifying correctness (e.g., seeing that no obvious shorter connections are omitted).

6. Performance Analysis: Include functionality to measure the length of tours found and the runtime of algorithms. Students should conduct tests on various input sizes (e.g., 5, 10, 15, 20, ... cities) and possibly on random versus structured city layouts, to illustrate how the exact algorithm becomes infeasible as n grows (exponential time) and how the heuristics scale. Provide theoretical complexity discussion for each algorithm and empirical observations.

2.2 Final Deliverables:

1. Source Code Repository: A complete C++ project (possibly in a Git repository), including all source files, build scripts (Makefile or equivalent), and test data. The repository should reflect a collaborative effort (with multiple contributors and descriptive commit history). The code must be clear, modular, and adhere to good coding standards (meaningful variable names, appropriate use of classes or structures, and no memory leaks).

2. Executable and Usage Instructions: A compiled executable (or instructions to compile from source) along with a usage guide. The program should accept input (list of cities/distances) and output a tour and its total cost. Include example input files (and expected outputs for verification of correctness).

3. Project Report: A formal document (ca. 15 pages) describing the problem and its significance, the algorithms implemented, and the results. This report should include:

(a) Problem Formalization: Definition of TSP with mathematical notation.

(b) Algorithm Descriptions: Explanation of each algorithm (brute force, DP, heuristics) with pseudocode and complexity analysis. Cite any known theoretical results (e.g., NP-hardness, approximation ratios) that support your design.

(c) Results: Tables or graphs comparing performance (runtime vs. n) and solution quality. Discuss any interesting observations, such as how quickly the exact method's time blows up as n increases, or how close the heuristic solutions are to optimal.

(d) Discussion: Challenges encountered, how they were overcome, and what potential improvements or alternative approaches could be explored (for example, could a different heuristic perform better, or could parallel processing speed up the search?).

(e) Team Contribution Log: A brief section or separate document summarizing each team member's contributions, ensuring accountability in the collaborative process. This can often be inferred from Git commits as well.

(f) Demonstration Materials: If a visual demo was implemented, include screenshots or a short video/gif of the tour visualization for a sample instance. If not, ensure the report includes a sample output tour for a non-trivial instance to illustrate the program's functionality.

2.3 Language and Tools Requirements

- Primary Language: C++ is strongly encouraged for implementing the TSP solver. The computational intensity of TSP (especially the exponential-time algorithms) makes performance critical. Using C++ allows low-level optimizations (e.g., bit operations for the DP state, efficient memory management for recursion, etc.). If the team has a compelling reason, another high-performance language like Java or Rust may be considered, but interpretive languages (e.g., Python) should be avoided for the core solving logic due to their slow execution on heavy computations.

- Libraries: The use of standard C++ STL libraries (such as ¡algorithm¿, ¡bitset¿ for DP bit masks, ¡vector¿, and ¡cmath¿) is permitted and encouraged. If visualization is implemented, a graphics library or plotting toolkit can be used (provided it is cross-platform or easy to set up). For any third-party library usage, approval may be required to ensure it does not trivialize the problem (e.g., you should not use an existing TSP solver library).

- Git and Collaboration: If you use Git, use branching for developing major features (e.g., a branch for implementing the DP solver, another for the heuristic, etc.) and merge them after thorough testing. Regular code

reviews within the team are recommended. Make sure to use meaningful commit messages and document any significant merges or conflict resolutions.

- Development Environment: It is the team's responsibility to ensure the code runson my windows computer. Testing should be done with optimization flags enabled (e.g., using -O2 or -O3 for the C++ compiler) to handle larger instances. Memory and performance profiling tools can be used to optimize the solution.

3 Applied Cryptography – RSA Encryption and Secure Communication

Delve into applied cryptography by implementing a secure communication system based on RSA public-key encryption. The goal is to create a working software prototype that can generate cryptographic keys, encrypt and decrypt messages, and possibly integrate symmetric encryption for efficient data transfer. The core of the project is the RSA cryptosystem, which relies on fundamental number-theoretic algorithms. RSA is a widely used public-key scheme for secure data transmission, based on the mathematical fact that while it is easy to multiply two large primes, it is extremely difficult to factor their product. Implement RSA key generation (including prime number generation and computing modular inverses), encryption and decryption functions, and then use these to secure a simple message exchange or data storage system. This project is mathematically rigorous: it will require understanding modular arithmetic, prime factorization, and the properties of Euler's totient function. Teams must ensure the correctness of their implementation (for example, verifying that encryption and decryption are inverses under the generated keys) and consider efficiency (large integer arithmetic and optimization for large primes). Optionally, the project can incorporate a symmetric cipher (such as AES) to encrypt bulk data, with RSA used to exchange the symmetric key - mimicking real-world secure communication protocols. The final system should demonstrate the entire pipeline of secure communication: key generation, encryption, transmission (simulated), and decryption, all implemented by the team (preferably in C++).

3.1 Required Components and Algorithms:

1. RSA Key Generation: Implement the RSA key generation algorithm, ensuring the following criteria are met:

    - Prime numbers are generated on the order of at least 16 bits for demonstration, and preferably 1024 bits or higher for actual security
    - A probabilistic primality test (such as Miller-Rabin) or a library routine is used to test primality.
    - The output keys should be a public key and private key
    - Ensure to handle big integers for all these computations (use C++ multiple-precision libraries such as boost::multiprecision::cppint if needed, or implement basic big integer support).

2. RSA Encryption/Decryption: Implement the RSA cryptographic functions (this will require efficient modular exponentiation – use fast exponentiation by squaring to handle large exponents).

3. Include proper padding or format for messages if dealing with text strings (for simplicity, you may choose a straightforward encoding like converting

bytes to integers without complex padding schemes, but should be aware of security implications).

4. Symmetric Encryption Integration (Optional): To simulate a full secure communication session, implement a symmetric-key cipher for the message payload:

    - Use a well-known algorithm like AES (Advanced Encryption Standard) for encrypting the actual message data under a random symmetric key. AES is a standard symmetric cipher widely used to secure data. You can implement a simplified version of AES or use a provided library for AES if low-level implementation is too time-consuming, but you must understand and explain its working.
    - Encrypt the symmetric key itself with RSA (this is how hybrid encryption is done: RSA secures the small key, AES secures the bulk data). The receiver will use RSA to decrypt the symmetric key, then use that key to decrypt the message.

5. Secure Communication Protocol: Develop a simple protocol or application scenario to demonstrate the system:

    - For example, a command-line program where one user can generate a key pair, publish the public key, and another user can use it to encrypt a message. The encrypted message is then transmitted (simulated via a file or console output) and the recipient uses their private key to decrypt it.
    - Alternatively, implement a client-server simulation where the server holds a public/private key pair and clients send encrypted messages to it.

6. Ensure the protocol includes steps for key exchange (or public key distribution), encryption of data, and decryption. Although network programming is not required (the exchange can be simulated by function calls or file writing), the roles of sender and receiver should be clearly defined in the code structure.

7. Validation and Testing: Provide extensive testing for each component:

    - Test the correctness of the extended Euclidean algorithm.
    - Test RSA encryption/decryption on known small examples (where p, q, e, d are manually computed) to validate that decrypting an encrypted message returns the original.
    - If possible, test the system with various message sizes and contents (alphabetic, binary data) to ensure reliability. Also test the scenario with the optional symmetric cipher to verify that the hybrid encryption works (the final plaintext after full decryption matches the original).

8. Analyze the performance: for example, measure how encryption/decryption time grows with key size or message size, and ensure it is manageable for demonstration purposes.

9. Security Considerations: Though this is primarily an implementation project, students should be aware of basic security considerations. Include in the documentation a brief discussion on:

   - Why RSA is secure (related to difficulty of prime factorization) and any assumptions (e.g., the keys generated are large enough to prevent brute force).
   - The importance of using padding schemes in real applications (though not mandatory to implement PKCS1 padding here, acknowledging it is good practice).
   - The difference between public-key and symmetric-key encryption and why both are used (highlight the efficiency of symmetric ciphers like AES for bulk data.
   - Any limitations of your implementation (for instance, if the system is not resistant to certain attacks due to simplified padding or smaller key sizes used in testing).

3.2 Final Deliverables:

1. Implemented Software: A fully working cryptographic program (preferably in C++). This should be provided as source code in the Git repository, and optionally as a compiled binary for convenience. The program's functionality must include:

   (a) RSA key generation (outputting public/private key pairs).

   (b) RSA encryption and decryption (with appropriate input/output format for messages).

   (c) If included, symmetric encryption/decryption and the combined usage with RSA for key exchange.

   (d) The software should be usable via a simple interface (command-line interface is acceptable), where a user can follow steps to generate keys, encrypt a message, and decrypt a message.

2. Git Repository (if Git is used): The repository containing all source code, with a clear structure (separating modules for RSA, symmetric cipher, etc.). The commit history should indicate the development timeline and contributions. Include a README.md with instructions on how to build and run the program, as well as sample usage. Any external libraries or resources should be acknowledged and included (or installation instructions given).

3. Technical Documentation: A comprehensive report (ca. 10 pages) describing the project. Key sections should include:

   (a) Introduction: Overview of RSA and cryptographic objectives of the project, including the problem statement in context (e.g., "securely transmitting a message over an insecure channel using our implemented cryptosystem").

   (b) Mathematical Background: Explanation of the mathematical foundations of RSA (prime numbers, modular arithmetic, one-way functions) in a way suitable for a second-year CS student. Cite the fundamental fact that RSA's security comes from factoring difficulty. If AES or another cipher was used, briefly explain symmetric vs. asymmetric encryption

   (c) Design and Implementation: Description of how the key generation was implemented (how primes are chosen/tested), how encryption and decryption are implemented (including any optimizations like fast modular exponentiation). Include pseudocode or flowcharts for the main algorithms. If a symmetric cipher is implemented, outline its operation (for AES, discuss the round structure at a high level).

   (d) Results: Present the results of testing the system. This includes examples of encryption/decryption (show a plaintext, the ciphertext produced, and the plaintext recovered). Also discuss performance results: for instance, how long key generation took for certain key sizes, how encryption time grows with message size, etc. If any limitations were encountered (e.g., inability to use extremely large keys due to time constraints), note them here.

   (e) Security Analysis: A short discussion on the security of the implemented system. Emphasize that while the project's RSA keys might not be large enough for real security (if that's the case), the implementation is conceptually correct. Mention any potential vulnerabilities (e.g., lack of proper random padding, which in real systems could lead to attacks). This section shows awareness of real-world cryptographic practices and where the project stands in relation.

   (f) Conclusion: Summarize what was learned and any future work (for example, how one could extend the project to a fuller system, use larger keys, or implement additional features like digital signatures).

4. User Guide: Include a brief user manual, either as a section in the report or a separate document, that explains how to run the program. This should have step-by-step instructions for generating keys and performing encryption/decryption, so the graders or instructors can easily verify the functionality. Include examples with actual command inputs and outputs for clarity.

5. Presentation/Demo (optional): Prepare the necessary materials (slides, and a live demo script). The presentation should focus on the problem

addressed, the approach taken, and a highlight of results (e.g., showing that your system successfully encrypted and decrypted a sample message).

3.3 Language and Tools Requirements:

- Programming Language: Implementation should be in C++ (preferred) or another low-level language like C. High-level languages like Python are not encouraged for the cryptographic computations, because managing big integers and performance-intensive tasks in Python would be too slow and abstracted. In C++, you may use libraries like boost::multiprecision for big integers or GMP (GNU Multiple Precision Arithmetic Library) if needed, but the core logic of RSA should be written and understood by the team (do not use any library's built-in RSA functions).

- Precision and Arithmetic: Since RSA deals with large integers, ensure your environment can handle integers larger than the standard 64-bit types. The use of cppint from Boost or Bigint libraries is acceptable. All random number generation for primes should use a cryptographically secure random source if possible (e.g., std::randomdevice or relevant library functions).

- Development Practices: You are encouraged to utilize Git for version control. Given the security-critical nature of cryptographic code, use Git to track all changes carefully—this also aids in code reviews. Teams should conduct mutual code reviews to catch logical errors (which can be common in complex math code). Unit tests for mathematical functions (gcd, mod inverse, primality test) should be written and run regularly.

- Testing Tools: It may be useful to incorporate existing test vectors for RSA (to ensure your implementation matches known outputs for small primes). Use debugging tools to step through key generation and encryption steps if results are not as expected. Memory checking tools (like Valgrind) are important to ensure no leaks or buffer overruns, especially when dealing with manual arrays or byte buffers for encryption.

- Documentation and Formatting: All code should be written in a clear, maintainable style. Given the formal nature of this project, proper inline documentation (comments explaining non-obvious sections of code, especially mathematical computations) is required. The final write-up should be prepared in a professional format, suitable for inclusion in a course syllabus or report compendium. Use of LaTeX for typesetting mathematical formulas in the report is encouraged (but not required) to maintain precision and clarity in notation.