Backtracking, Branch and Bound

Week 11

COMP 1201 (Algorithmics)

ECS, University of Southampton

20 May 2020

Previously...

Approximation Algorithms

- Tractable (i.e. polynomial time) algorithms.
- Always output a solution with value within an α factor of an optimal solution.
- Next Fit: a 2-approximation algorithm for Bin Packing, which runs in $O(n)$ time.
- First Fit Decreasing (FFD): a $\frac{3}{2}$-approximation algorithm for Bin Packing, which runs in $O(n^2)$ time.

State Space Representation

- Many real world problems involve taking a series of actions to manipulate the state of the system.
- One of the central ideas in the area of planning and search within artificial intelligence.
- It helps to think of states as nodes of a graph which are linked if there exists an action taking us from one state to another.
- This provides a **state space representation** of the problem (we saw something similar when we derived a low bound on sorting).

State Space Representation (8-Puzzle)

- Consider the 8-Puzzle problem.
- We can place one tile into the empty slot in a single move.
- The objective is to rearrange the tiles from their initial state into their goal state.
- How can we **search** for a solution?

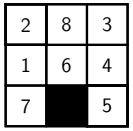
2	8	3
1	6	4
7		5

Initial state

8	5	
6	4	1
2	7	3

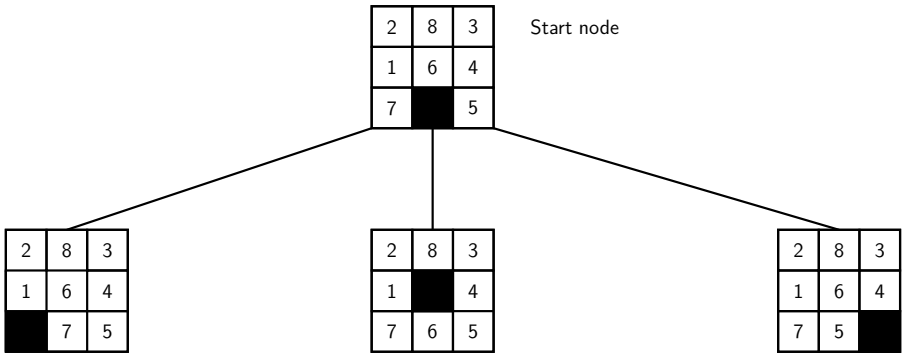
Goal state

State Space Representation (8-Puzzle)

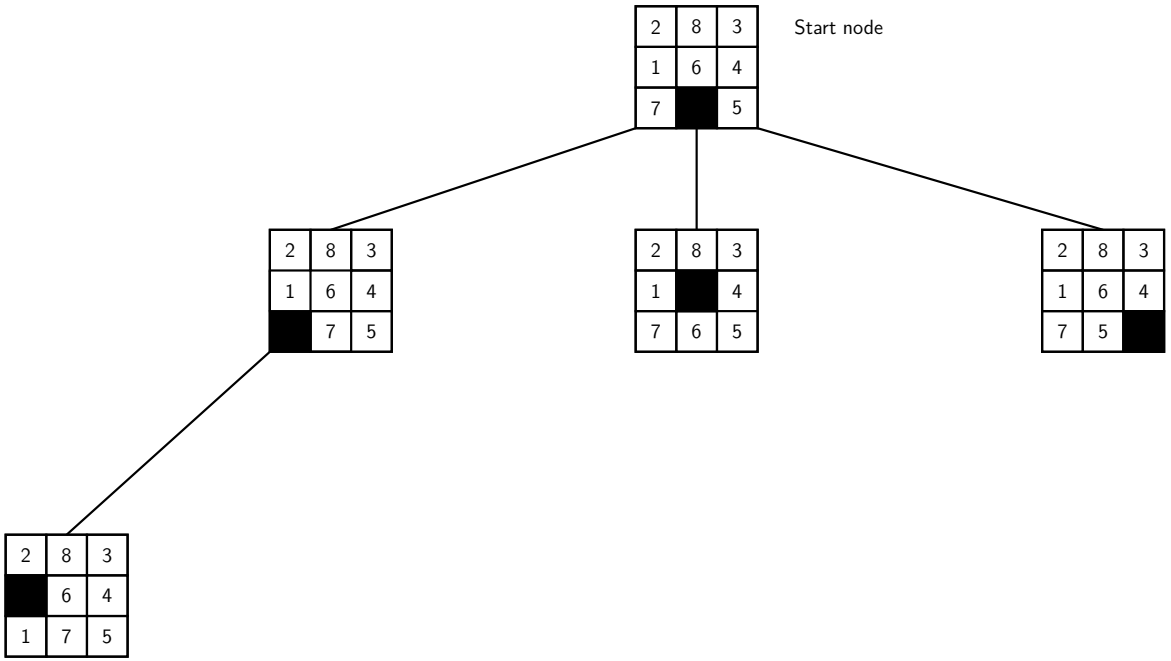


Start node

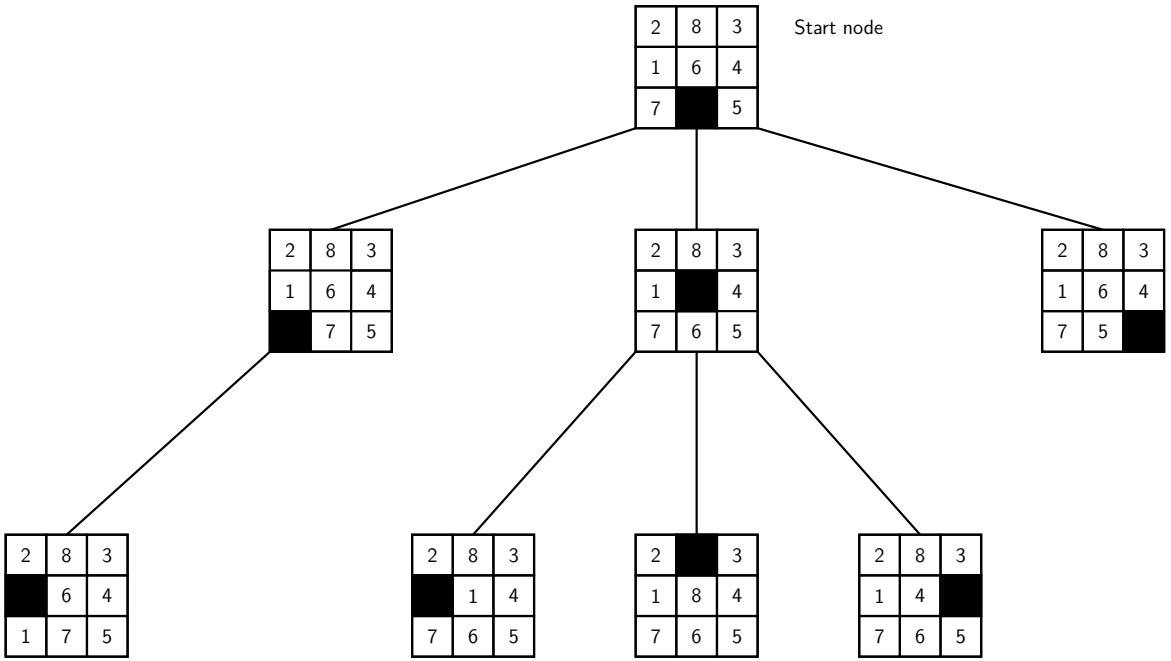
State Space Representation (8-Puzzle)



State Space Representation (8-Puzzle)



State Space Representation (8-Puzzle)



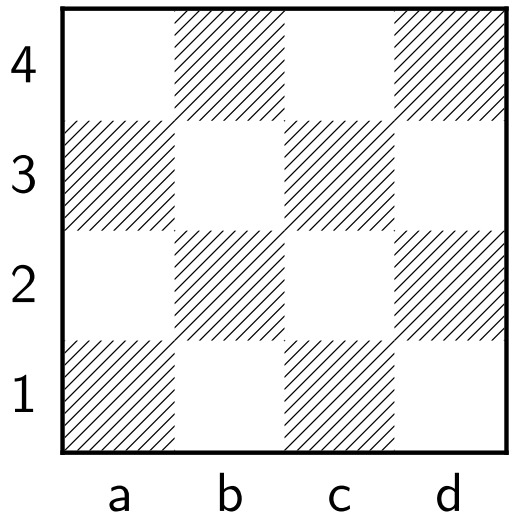
Large State Spaces

- The search space typically increases exponentially with problem size.
- We can find the quickest solution to the 8-puzzle using **breadth-first search**, but larger puzzles soon become intractable.
- Nevertheless, a lot of important problems involve very large state spaces and we have to find algorithms to explore them.

Backtracking

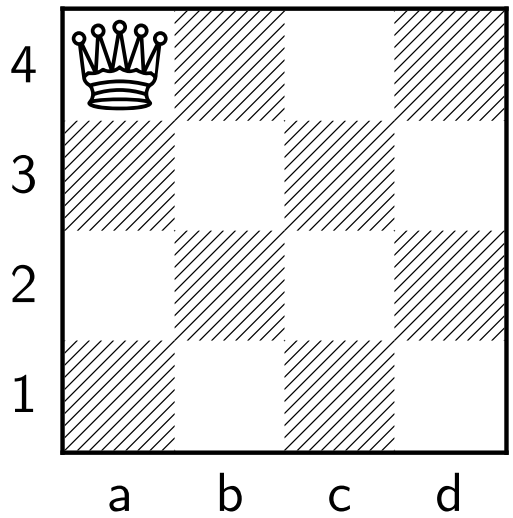
- **Backtracking** is used to find feasible solutions in large state spaces.
- E.g. solving Sudoku.
- It works until either:
  1. a feasible solution is found (and we can finish), or
  2. no feasible solution is possible (and we **backtrack**).
- We often search the state space using depth first search.

4-Queens Problem



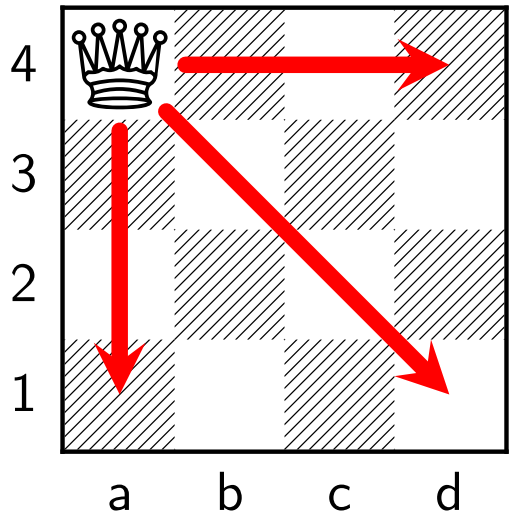
Let us consider a 4×4 chessboard.

4-Queens Problem



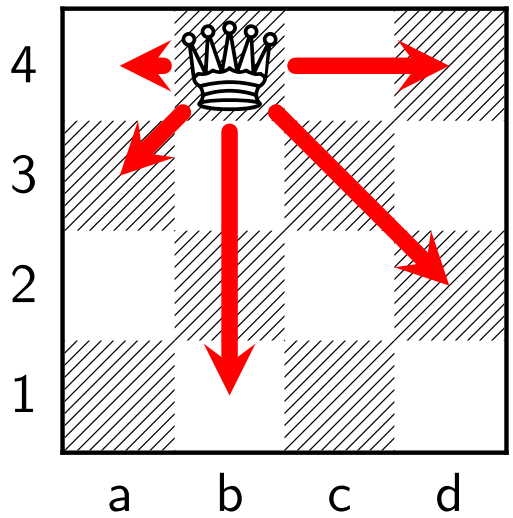
To solve the **4-queens problem** we are required to place 4 queen pieces into the board in a way that none are under attack.

4-Queens Problem



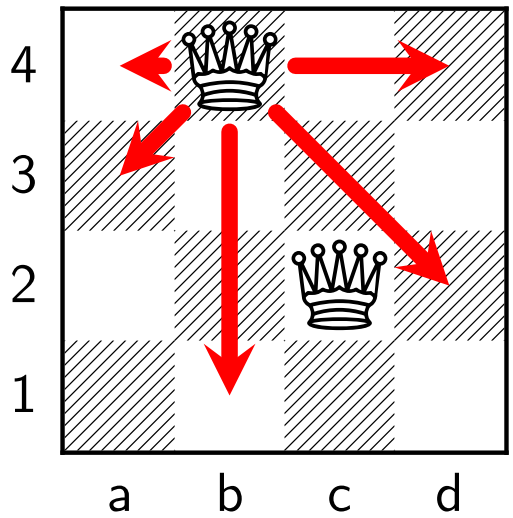
A queen may move along straight lines and diagonals.

4-Queens Problem



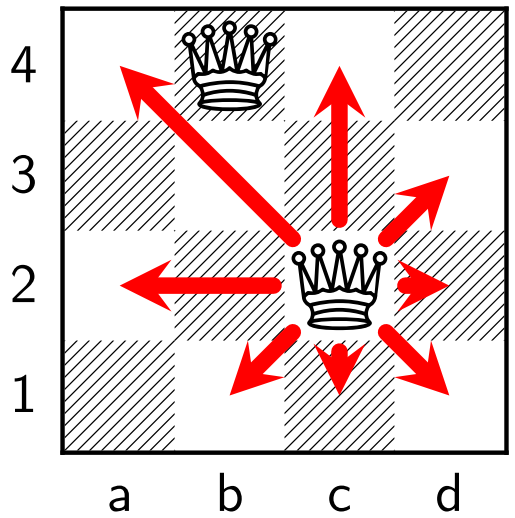
A queen may move along straight lines and diagonals.

4-Queens Problem



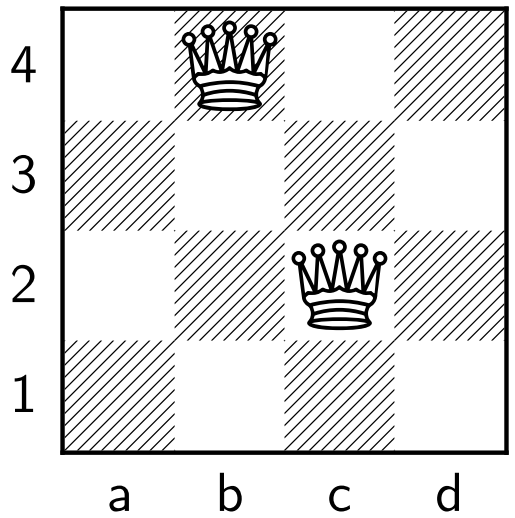
Under this placement of 2 queen pieces, neither piece is under attack by the other one.

4-Queens Problem



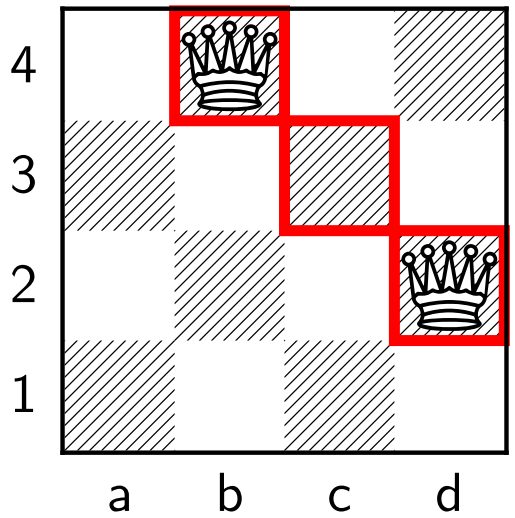
Under this placement of 2 queen pieces, neither piece is under attack by the other one.

4-Queens Problem



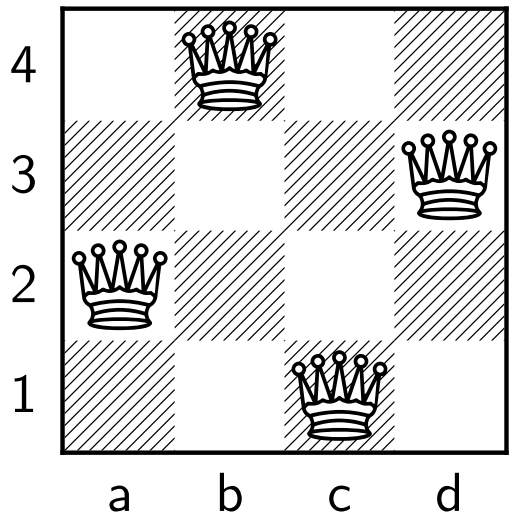
Under this placement of 2 queen pieces, neither piece is under attack by the other one.

4-Queens Problem



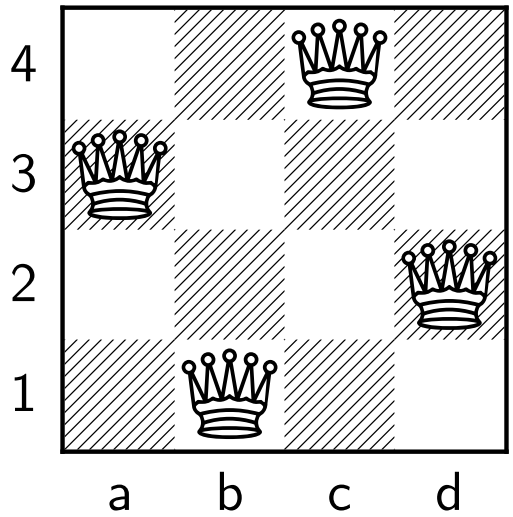
Here we clearly have a **conflict**! This cannot part of our solution.

4-Queens Problem



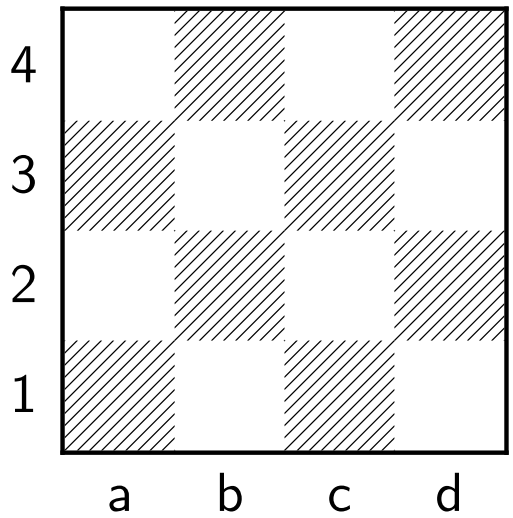
A possible **solution** to the 4-queens problem.

4-Queens Problem



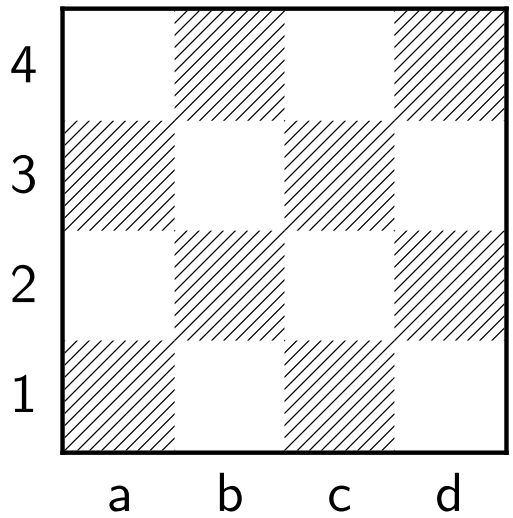
Another possible **solution** to the 4-queens problem.

4-Queens Problem



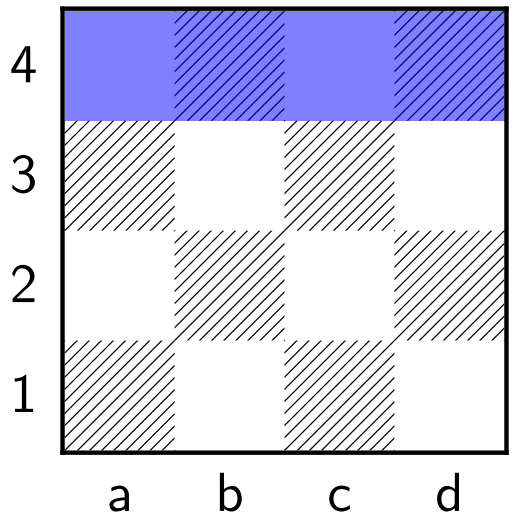
How do we solve the 4-queens problem using backtracking?

4-Queens Problem



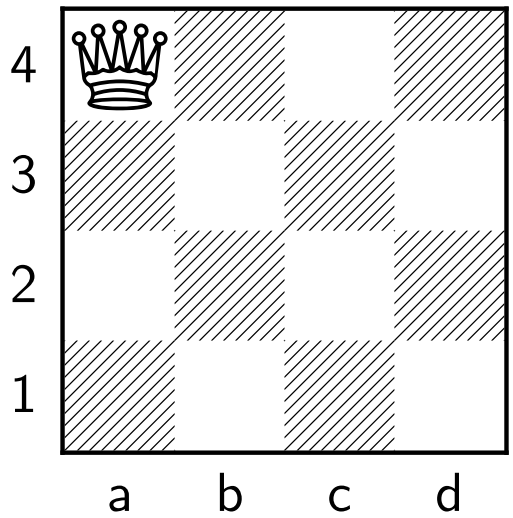
Let us represent our solutions by vectors of length 4 where each non-zero entry j in position i denotes a queen placed in row i and column j. The initial state of an empty board is $(0, 0, 0, 0)$.

4-Queens Problem



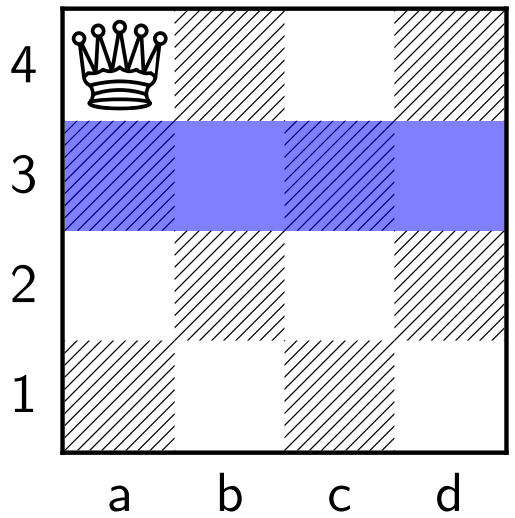
Consider the top row (rank) and place a queen in a position where no piece is under attack.

4-Queens Problem



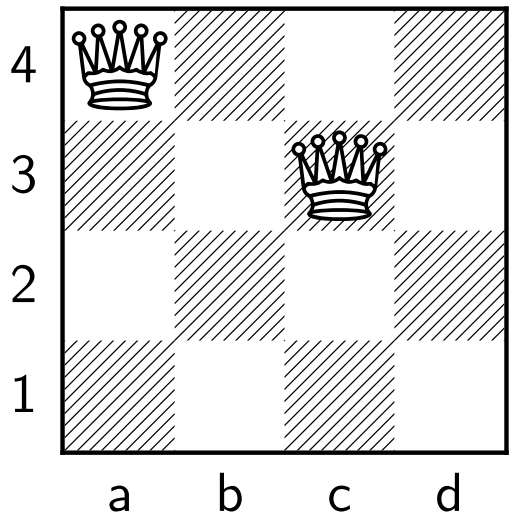
Consider the top row (rank) and place a queen in a position where no piece is under attack. State: $(0, 0, 0, a)$.

4-Queens Problem



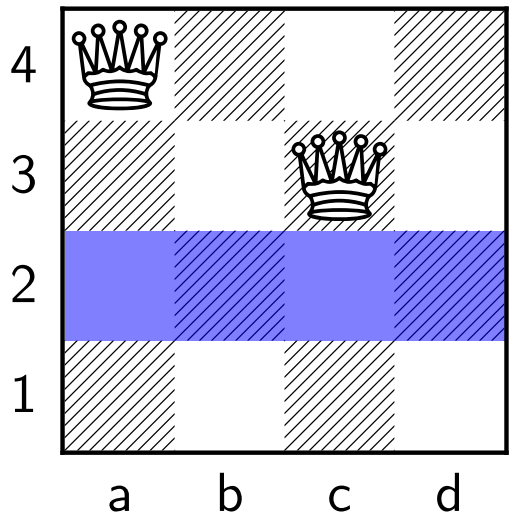
Proceed to the next row (rank) down and place a queen in a position where no piece is under attack.

4-Queens Problem



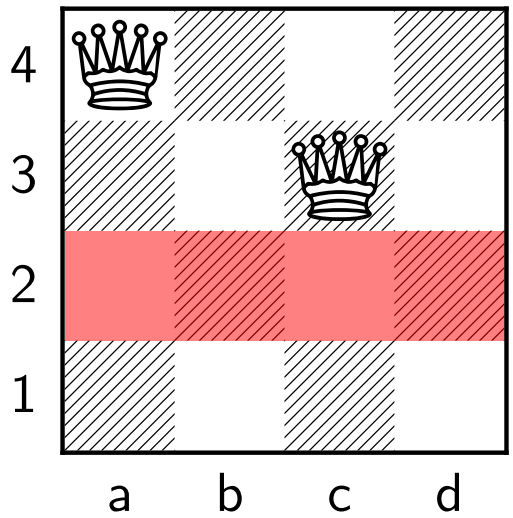
Proceed to the next row (rank) down and place a queen in a position where no piece is under attack. State: $(0, 0, c, a)$.

4-Queens Problem



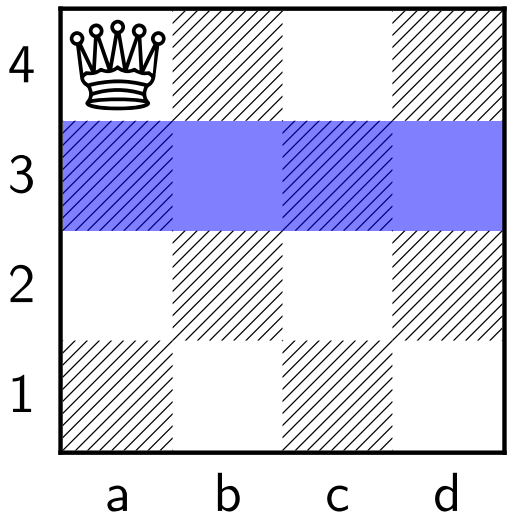
Proceed to the next row down (row 2) and place a queen in a position where no piece is under attack.

4-Queens Problem



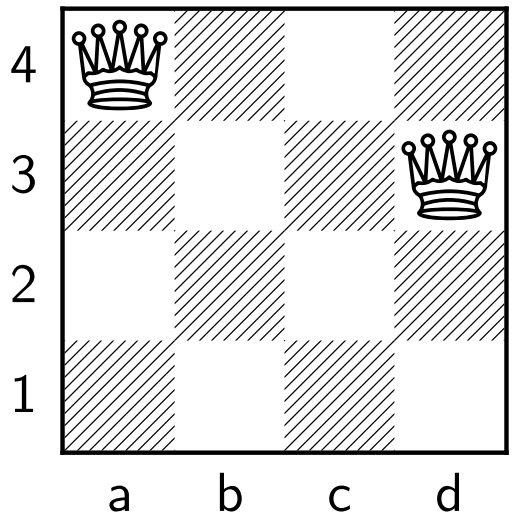
Proceed to the next row down (row 2) and place a queen in a position where no piece is under attack. That's impossible! **Backtrack!**

4-Queens Problem



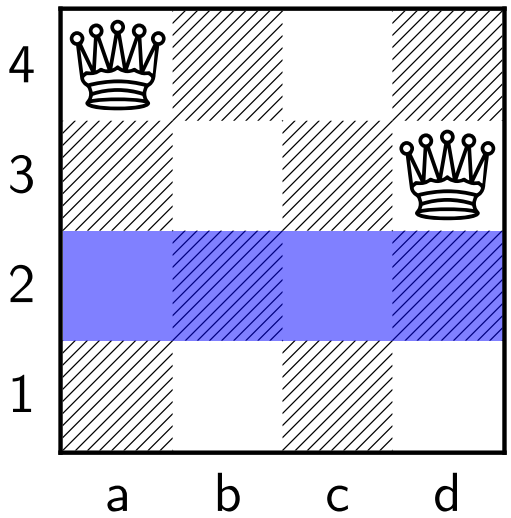
Go back to row 3 and find an alternative safe position for the queen piece.

4-Queens Problem



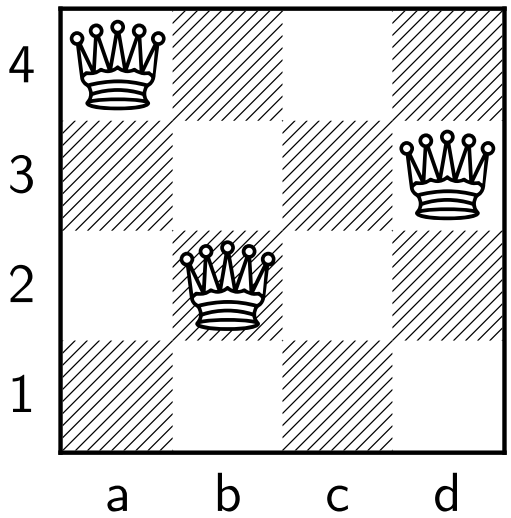
Go back to row 3 and find an alternative safe position for the queen piece. State: $(0, 0, d, a)$.

4-Queens Problem



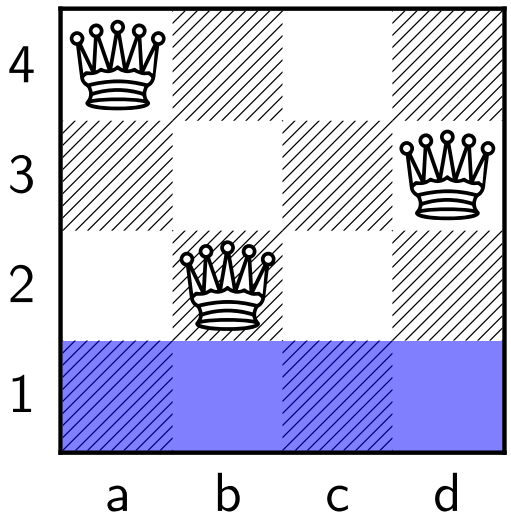
Go to the next row and find a safe position for the queen piece.

4-Queens Problem



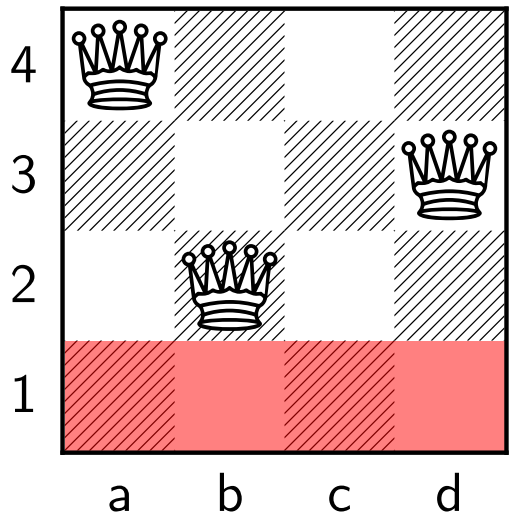
Go to the next row and find a safe position for the queen piece.
State: $(0, b, d, a)$.

4-Queens Problem



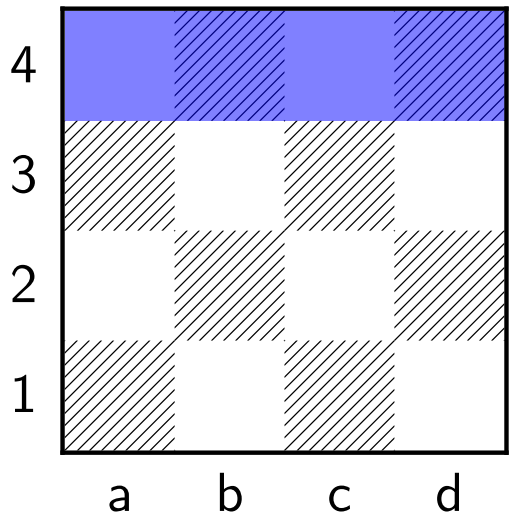
Go to the bottom row and find a safe position for the queen piece.

4-Queens Problem



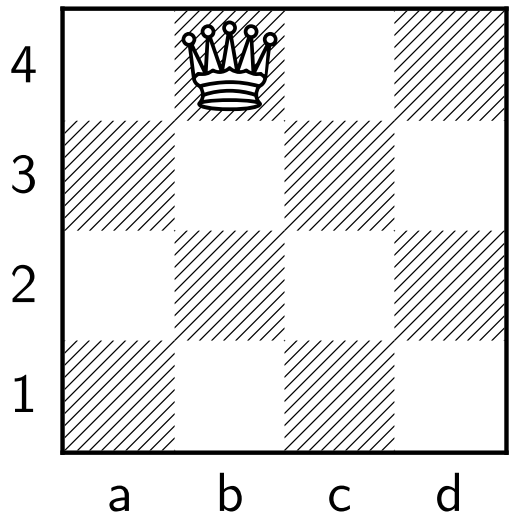
Go to the bottom row and find a safe position for the queen piece. That's impossible! **Backtrack!**

4-Queens Problem



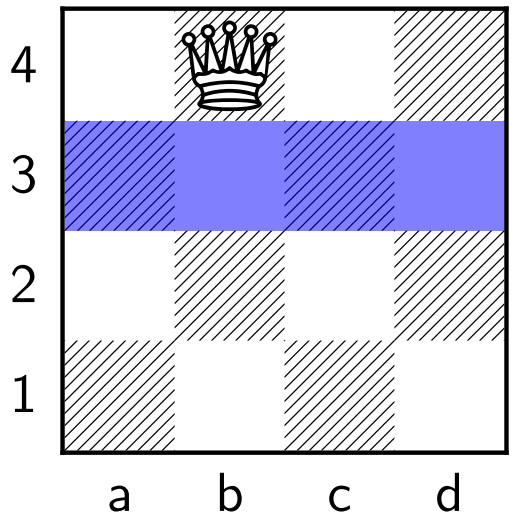
Go back the top row (rank) and place a queen in a different position where no piece is under attack.

4-Queens Problem



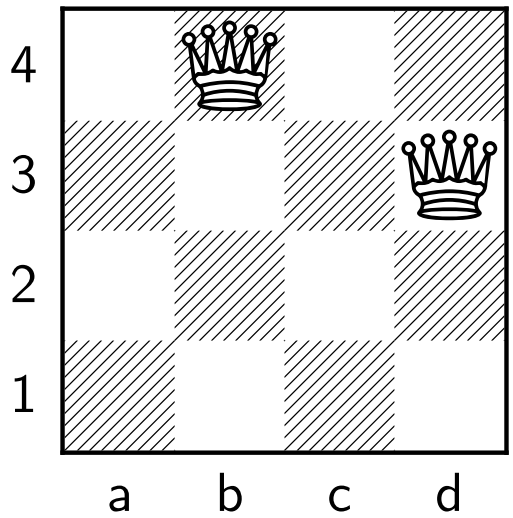
Go back the top row (rank) and place a queen in a different position where no piece is under attack. State: $(0, 0, 0, b)$.

4-Queens Problem



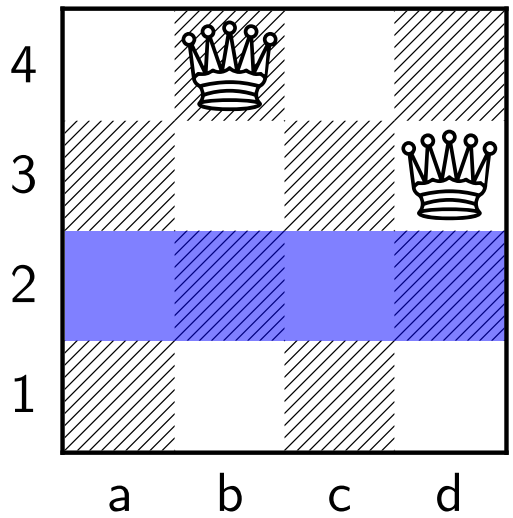
Go down a row and place a queen piece in a safe position.

4-Queens Problem



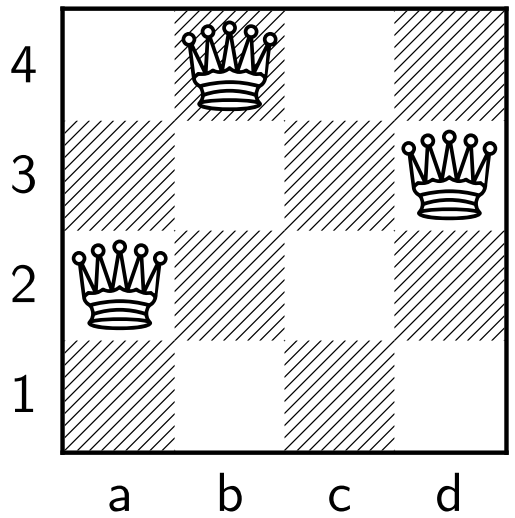
Go down a row and place a queen piece in a safe position. State: $(0, 0, d, b)$.

4-Queens Problem



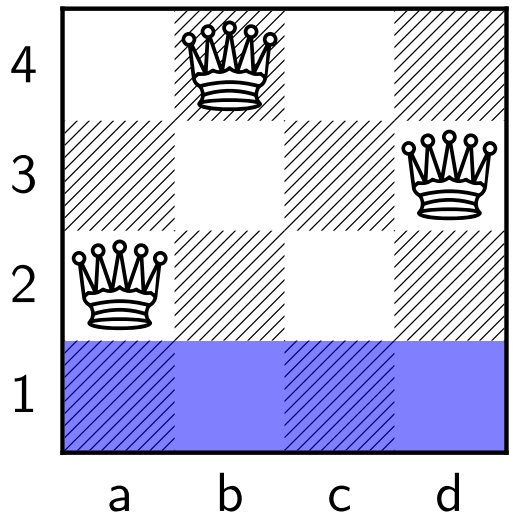
Go down a row and place a queen piece in a safe position.

4-Queens Problem



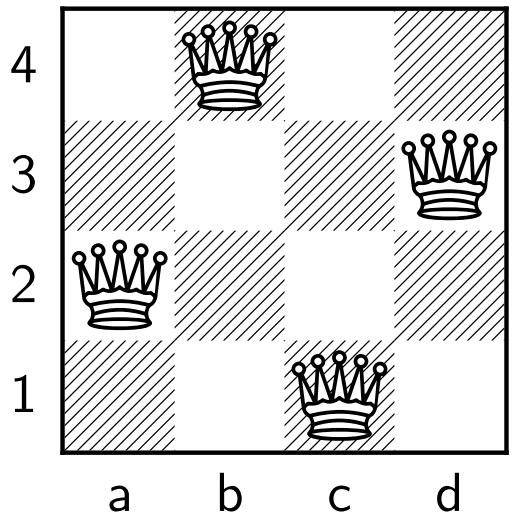
Go down a row and place a queen piece in a safe position.
State: $(0, a, d, b)$.

4-Queens Problem



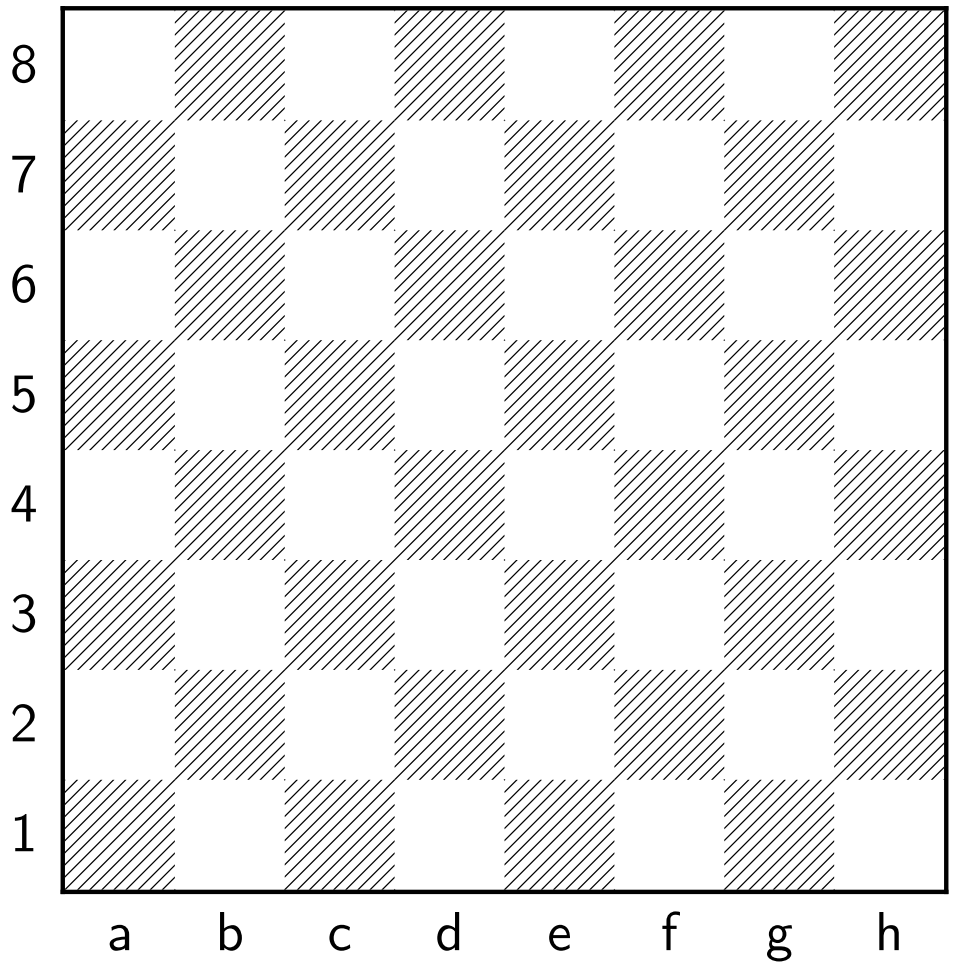
Go down a row and place a queen piece in a safe position.

4-Queens Problem



We have a solution! State: (c,a,d,b).

8-Queens Problem



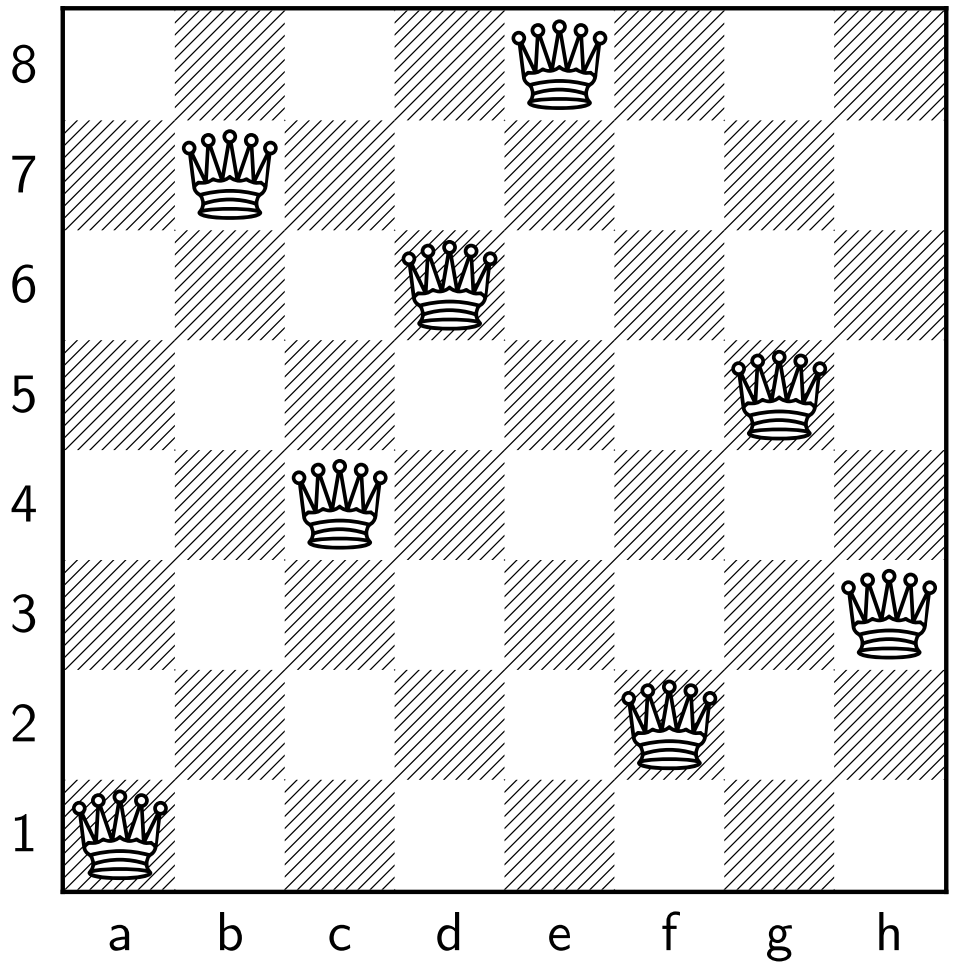
What about placing 8 queen pieces on a regular board?

8-Queens Problem

What about placing 8 queen pieces on a regular board?

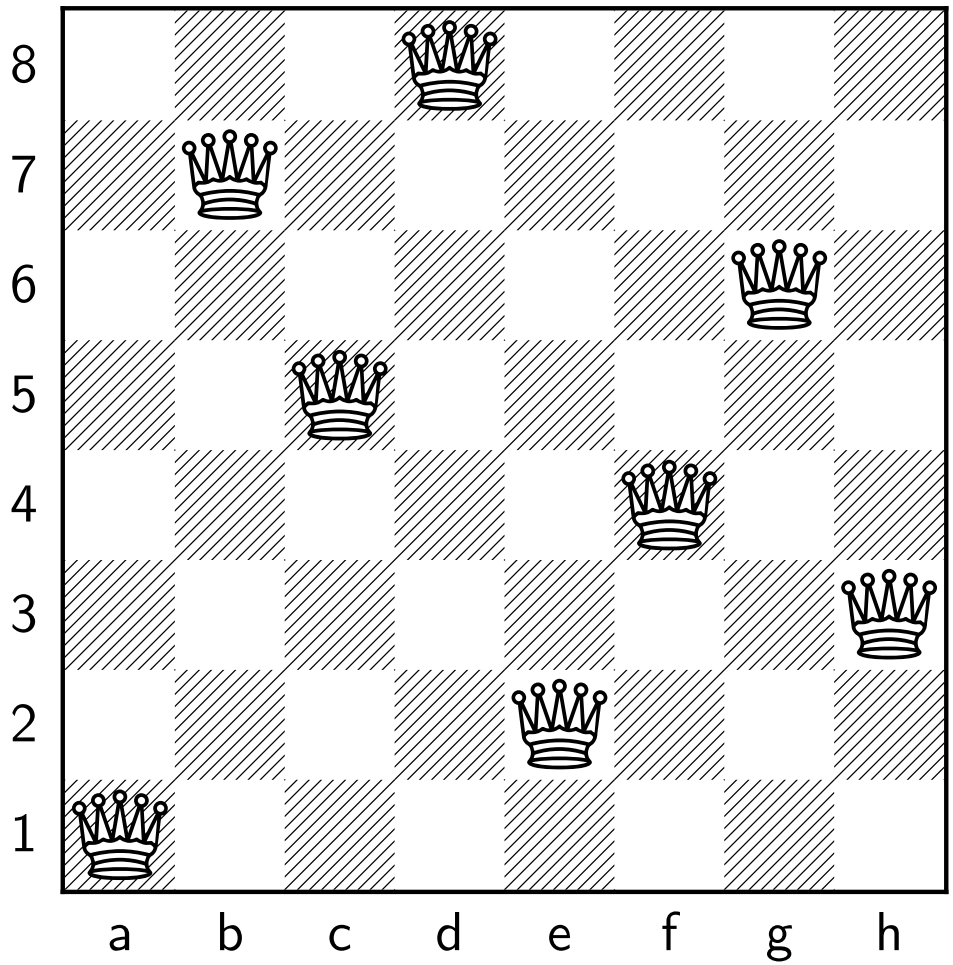
- Classic problem (first posed in 1848).
- Solved by various mathematicians in the 19th century.
- A backtracking solution given by Edsger Dijkstra in 1972.
- Over 100 backtracking steps are needed to find the first solution.
- A backtracking algorithm can be used to find all the solutions to the 8-queen problem.

8-Queens Problem



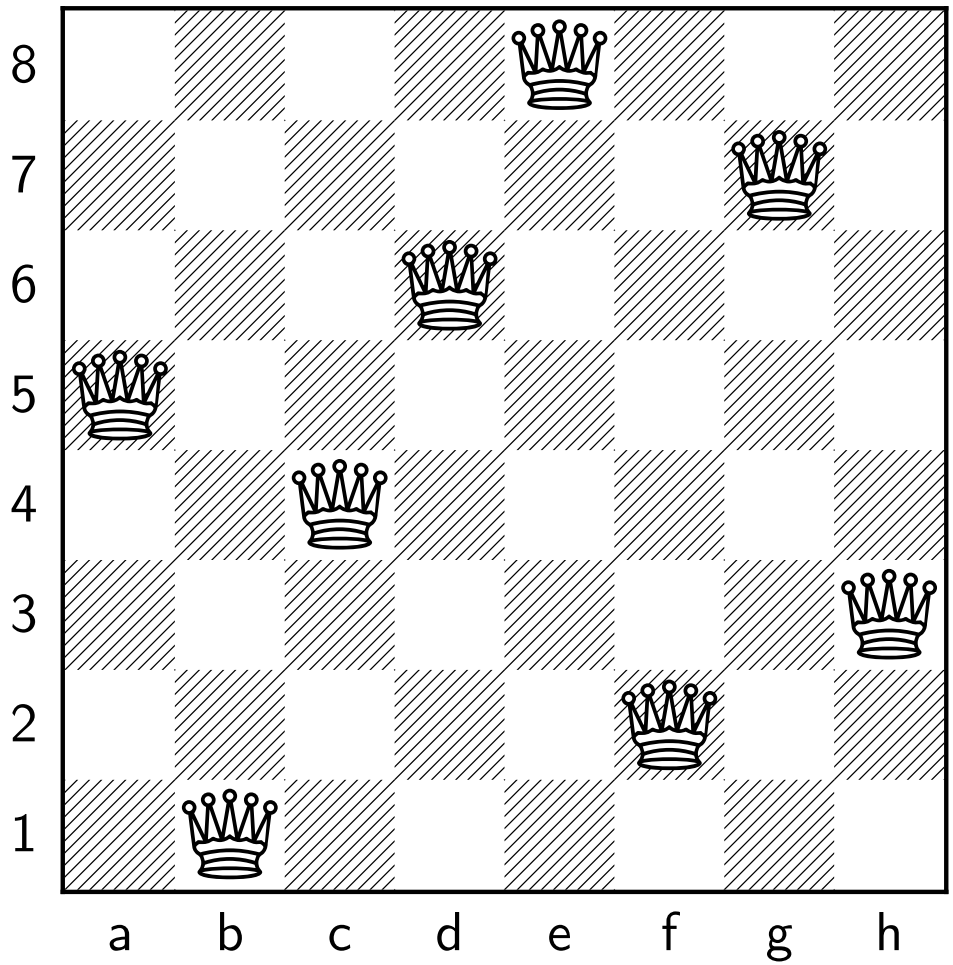
There are 92 distinct solutions to the 8-queens problem.

8-Queens Problem



There are 92 distinct solutions to the 8-queens problem.

8-Queens Problem



There are 92 distinct solutions to the 8-queens problem.

n-Queens Problem

- The n-queens problem is a generalisation of the 8-queens problem to an $n \times n$ chessboard.
- All solutions have only been discovered for $n = 1, \ldots, 27$.
- The case $n = 27$ was only solved in 2016.
- It has $234,907,967,154,122,528$ solutions.

Thomas Preußner, Matthias Engelhardt
"Putting Queens in Carry Chains, No 27", *Journal of Signal Processing Systems* (88).
https://link.springer.com/article/10.1007/s11265-016-1176-8

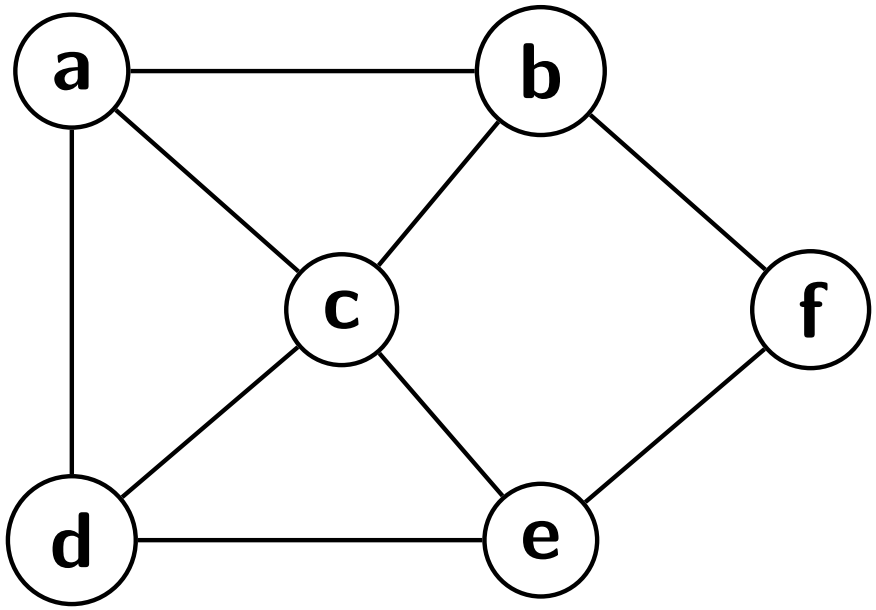
n-Queens Problem

- An implementation of backtracking is easily done using *recursion*.
- Requires one to implement a function that *efficiently* checks for conflicts in the current state representing a partial solution.
- Very similar to depth first search.
- A good implementation of backtracking prunes away large chunks of the search space and is more efficient than a naïve search for solutions.

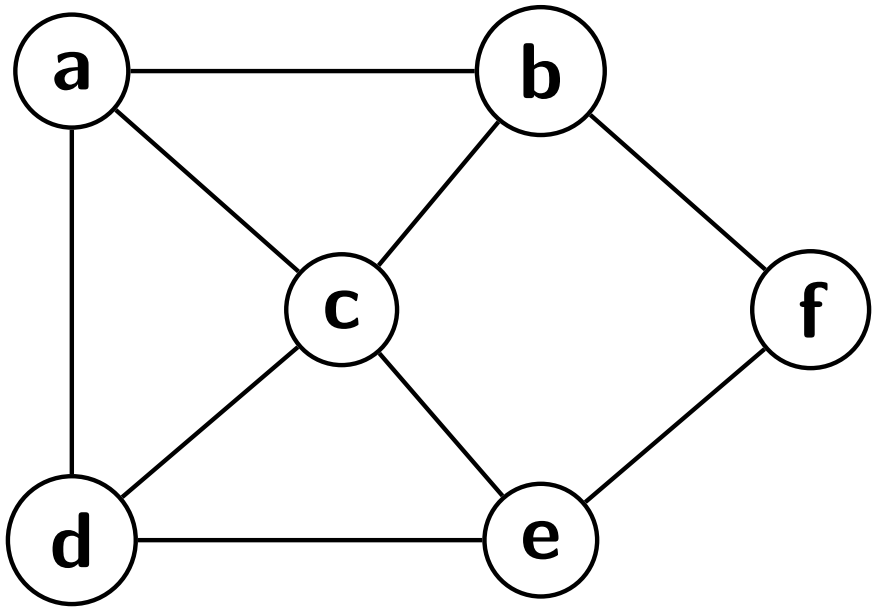
Hamiltonian Cycles

- A **Hamiltonian cycle** is a tour through a graph which visits every vertex once only and returns to the start.
- The general problem of deciding whether a Hamiltonian cycle exists in a graph is **NP-Complete**.
- There are no currently known algorithms that can solve this problem in polynomial time.
- For many graphs this is not too hard.

Hamiltonian Cycle Example (Backtracking)



Hamiltonian Cycle Example (Backtracking)

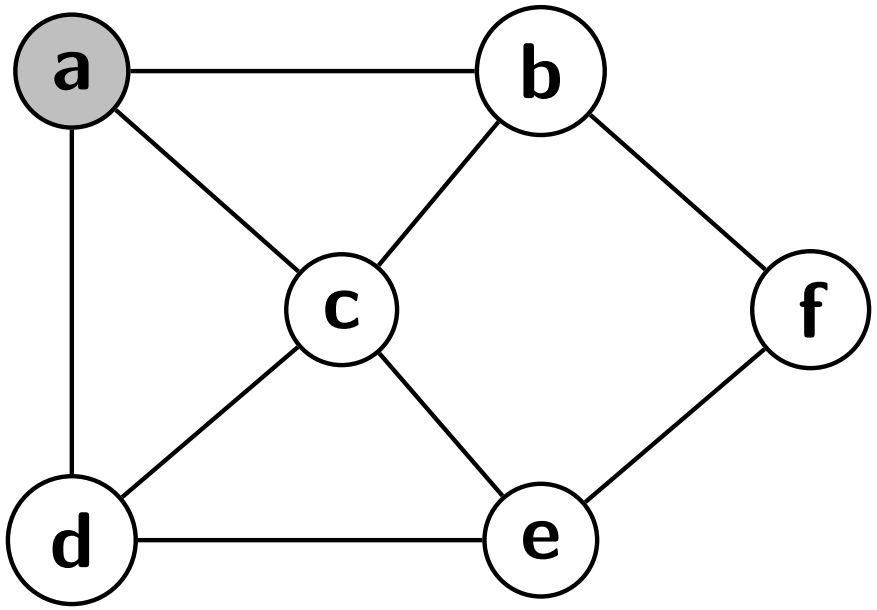


Let us represent partial solutions by a vector of length 7

$$(S, 0, 0, 0, 0, 0, S)$$

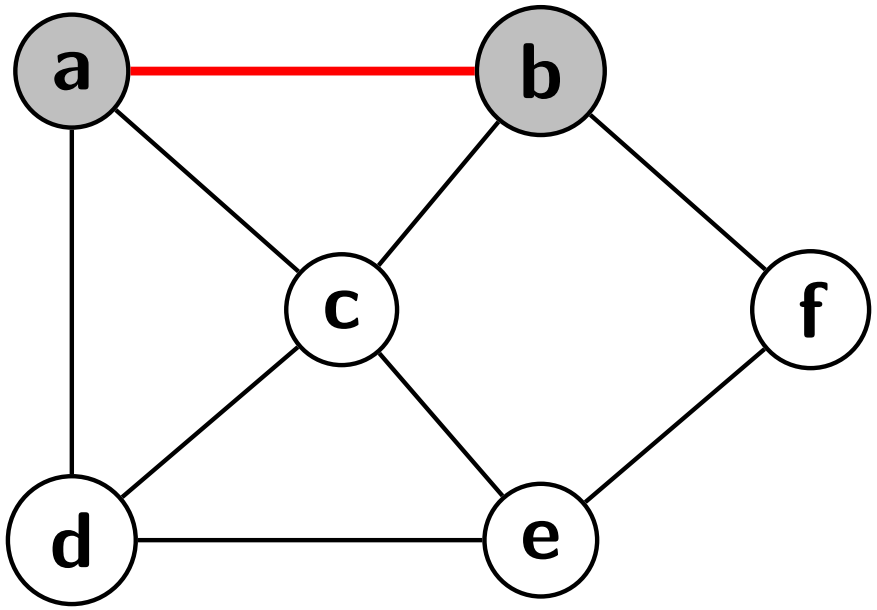
S is the beginning/ending node of our tour and 0 is unknown.

Hamiltonian Cycle Example (Backtracking)



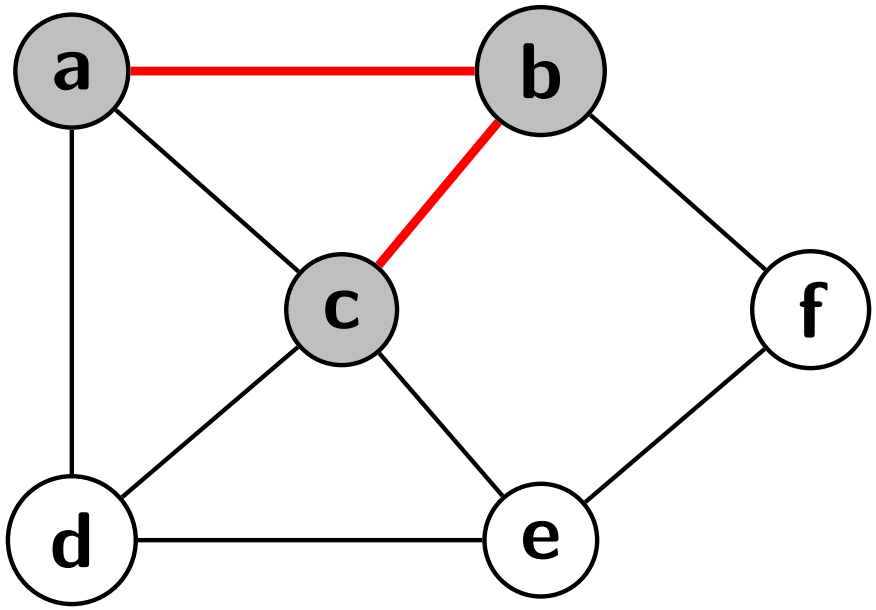
$$(\mathbf{a}, 0, 0, 0, 0, 0, \mathbf{a})$$

Hamiltonian Cycle Example (Backtracking)



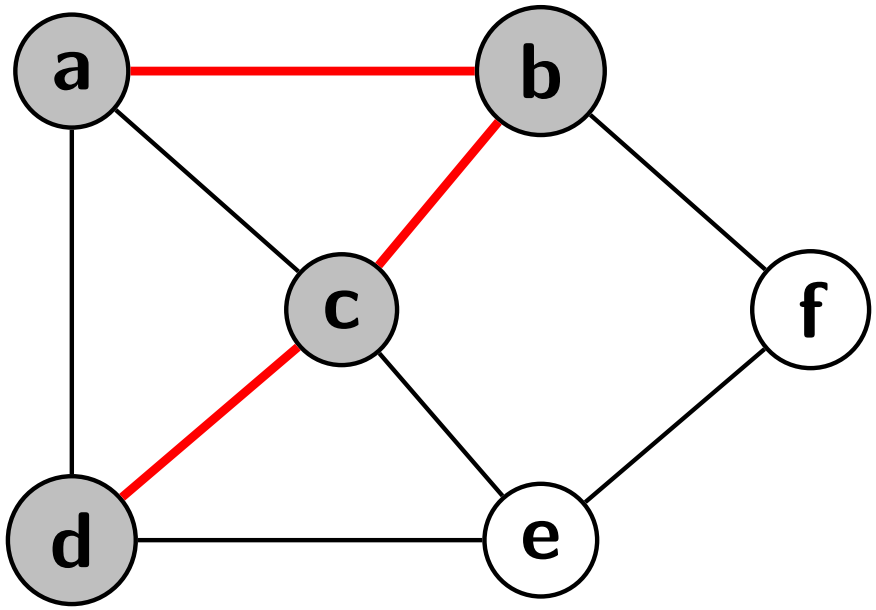
$$(\mathbf{a}, \mathbf{b}, 0, 0, 0, 0, \mathbf{a})$$

Hamiltonian Cycle Example (Backtracking)



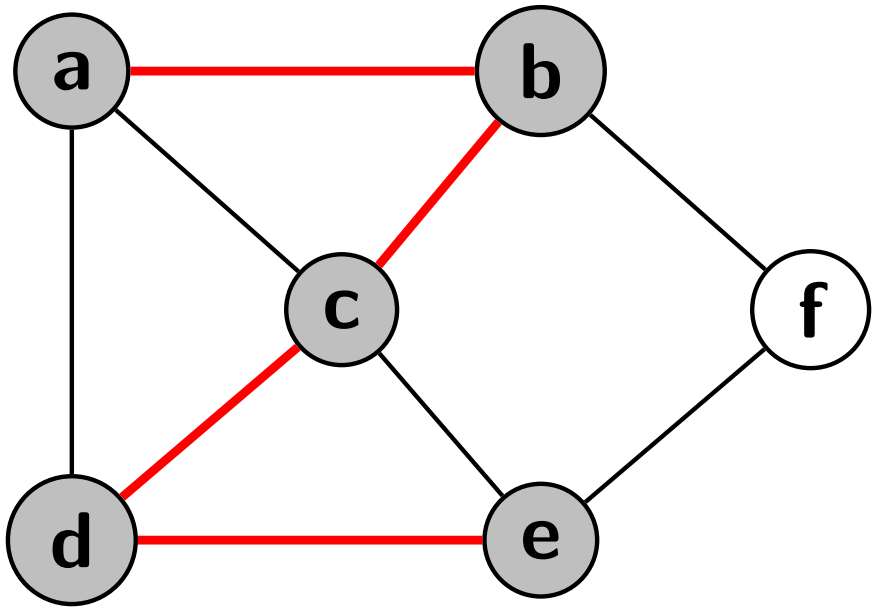
$$(\mathbf{a}, \mathbf{b}, \mathbf{c}, 0, 0, 0, \mathbf{a})$$

Hamiltonian Cycle Example (Backtracking)



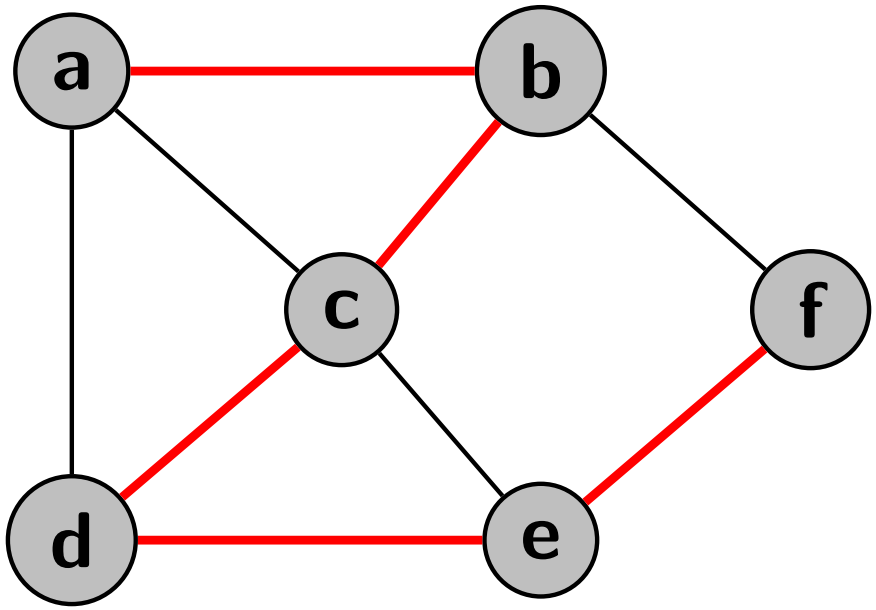
$$(\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}, 0, 0, \mathbf{a})$$

Hamiltonian Cycle Example (Backtracking)



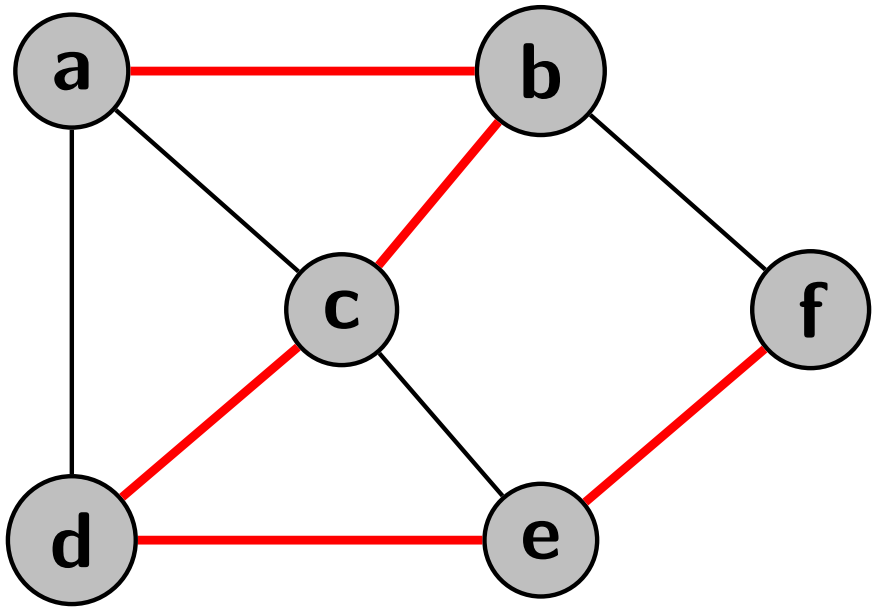
$$(\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}, \mathbf{e}, 0, \mathbf{a})$$

Hamiltonian Cycle Example (Backtracking)



$(\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}, \mathbf{e}, \mathbf{f}, \mathbf{a})$

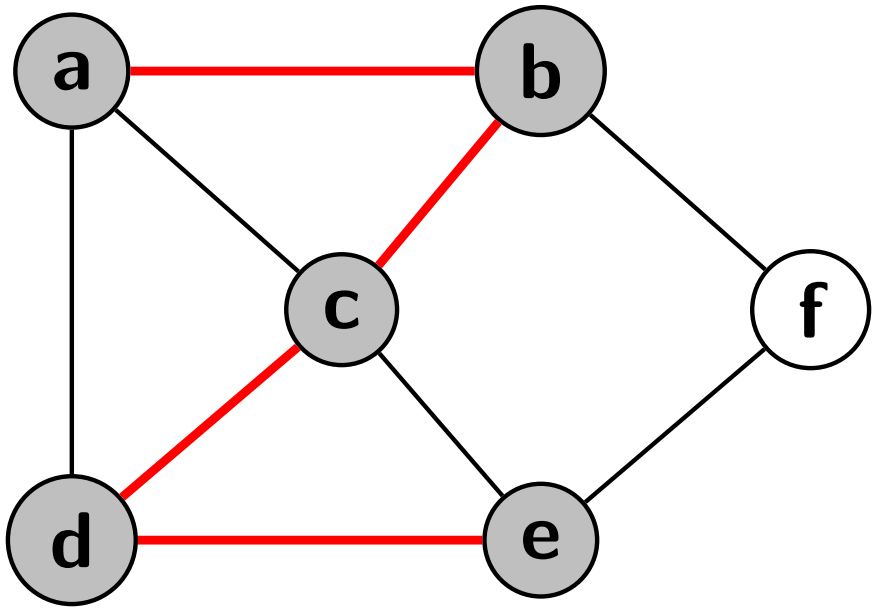
Hamiltonian Cycle Example (Backtracking)



$(\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}, \mathbf{e}, \mathbf{f}, \mathbf{a})$

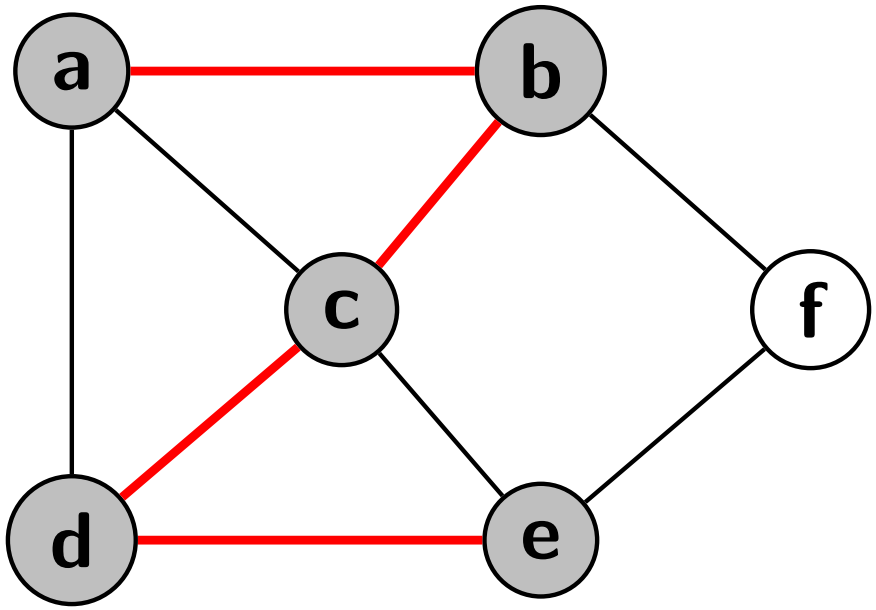
Backtrack!

Hamiltonian Cycle Example (Backtracking)



$$(\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}, \mathbf{e}, 0, \mathbf{a})$$

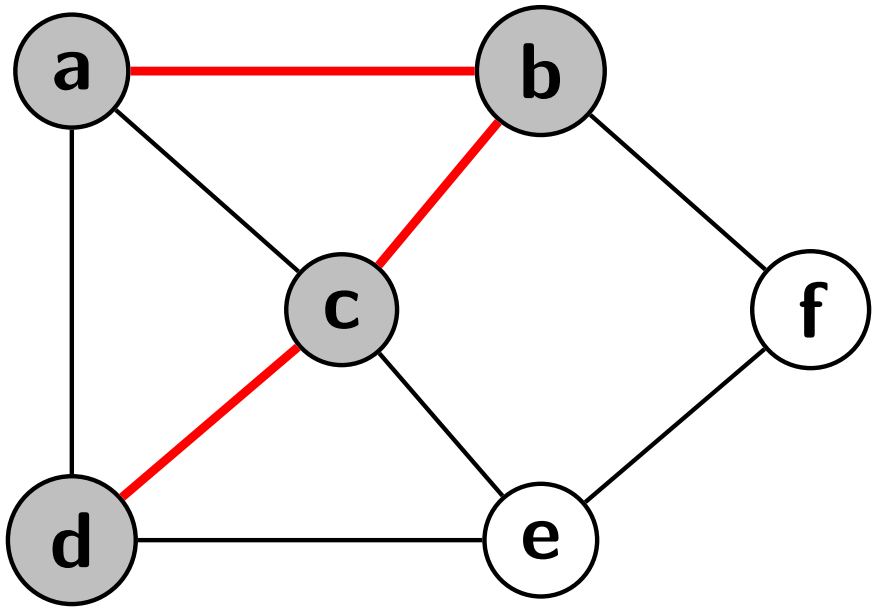
Hamiltonian Cycle Example (Backtracking)



$(\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}, \mathbf{e}, 0, \mathbf{a})$

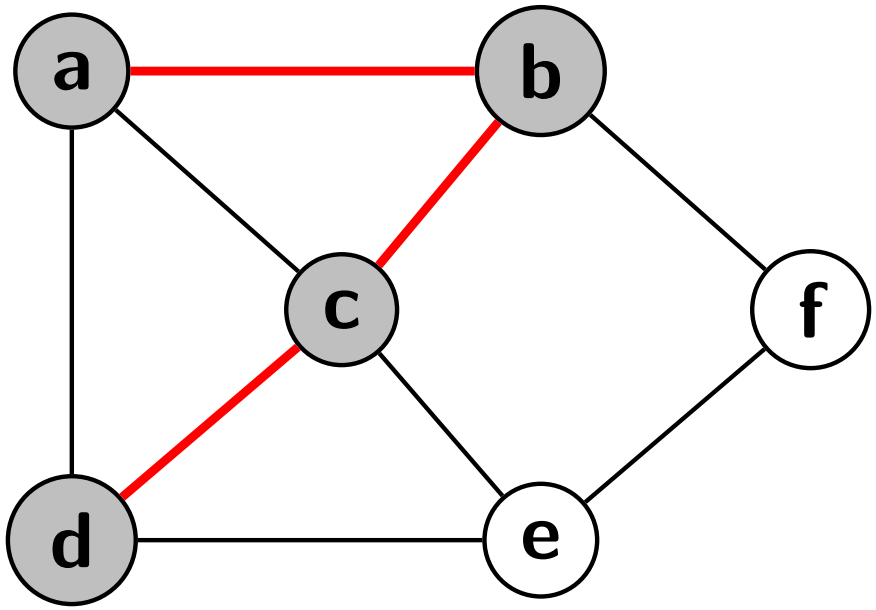
Backtrack!

Hamiltonian Cycle Example (Backtracking)



$$(\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}, 0, 0, \mathbf{a})$$

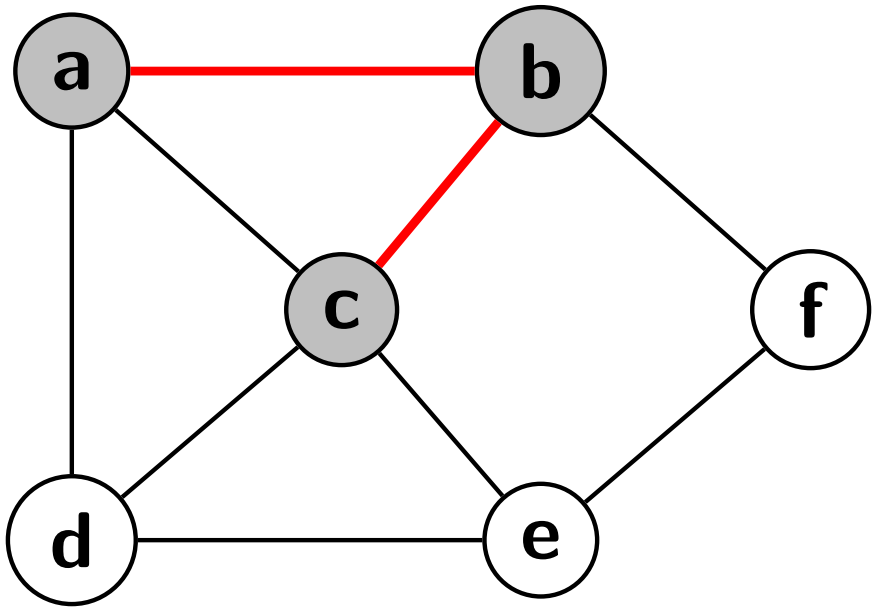
Hamiltonian Cycle Example (Backtracking)



$$(\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}, 0, 0, \mathbf{a})$$

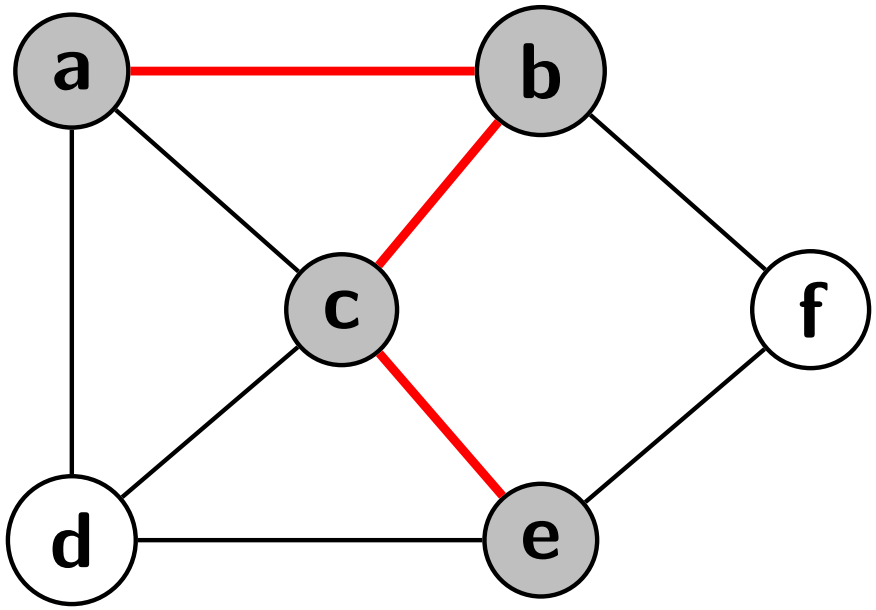
Backtrack!

Hamiltonian Cycle Example (Backtracking)



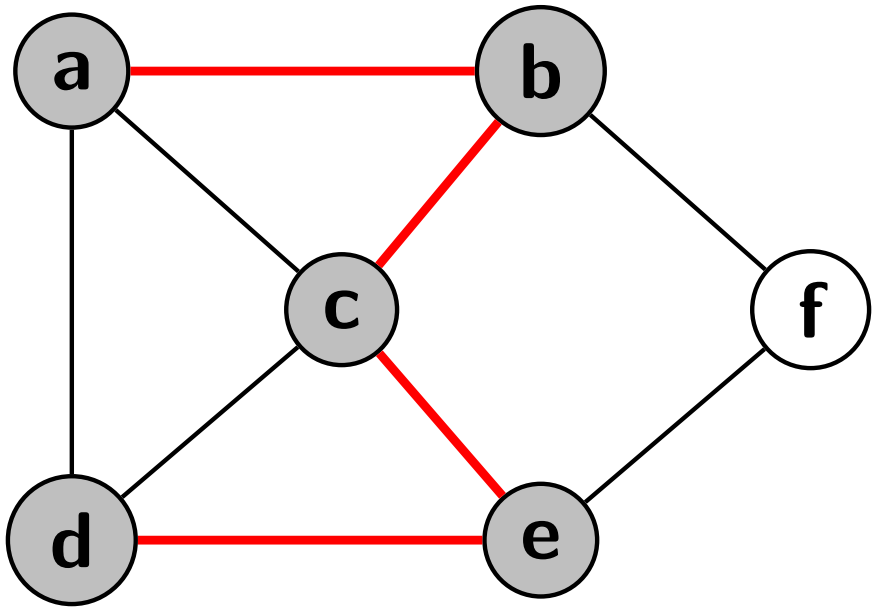
$$(\mathbf{a}, \mathbf{b}, \mathbf{c}, 0, 0, 0, \mathbf{a})$$

Hamiltonian Cycle Example (Backtracking)



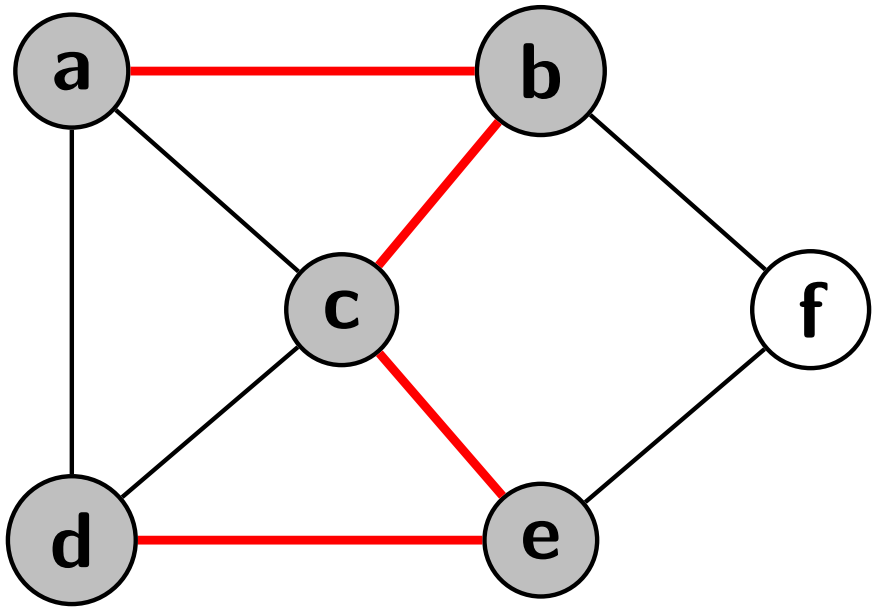
$$(\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{e}, 0, 0, \mathbf{a})$$

Hamiltonian Cycle Example (Backtracking)



$$(\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{e}, \mathbf{d}, 0, \mathbf{a})$$

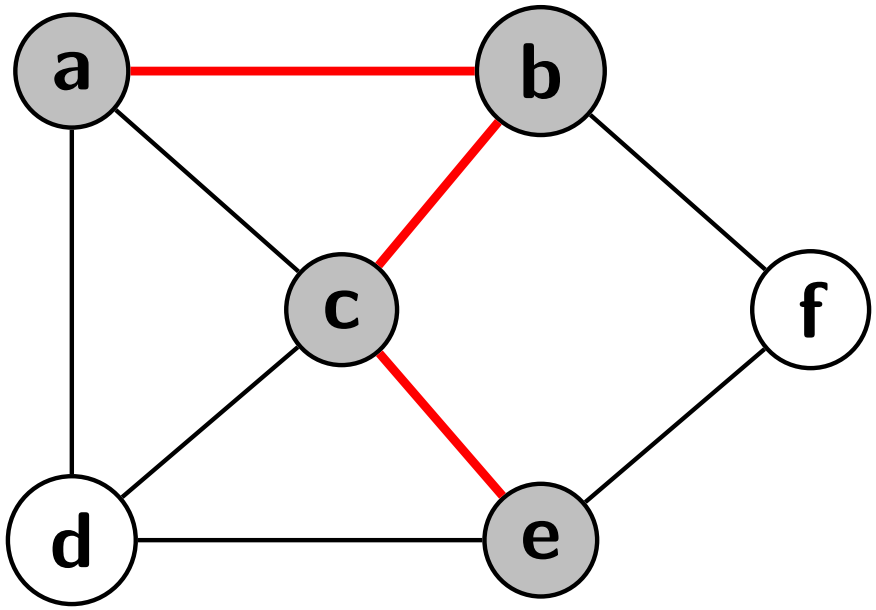
Hamiltonian Cycle Example (Backtracking)



$(\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{e}, \mathbf{d}, 0, \mathbf{a})$

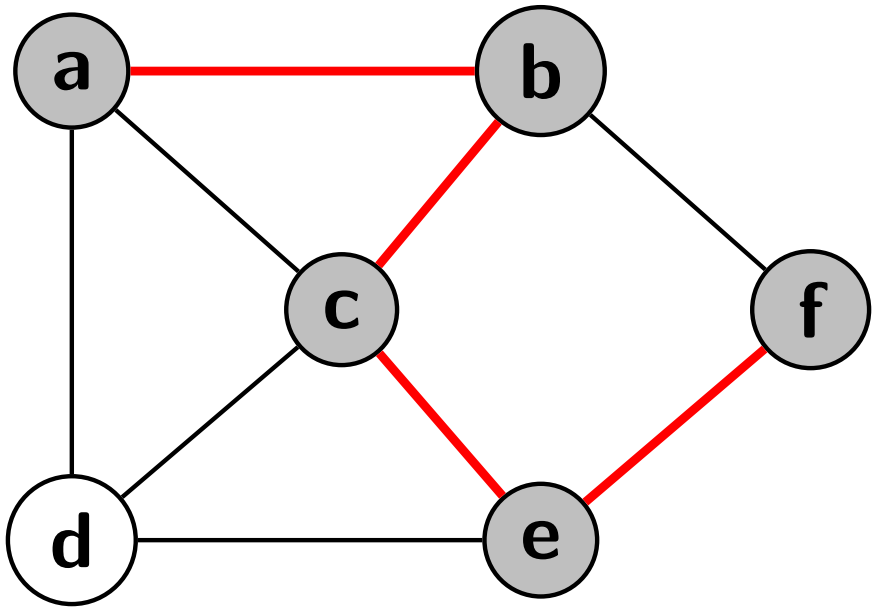
Backtrack!

Hamiltonian Cycle Example (Backtracking)



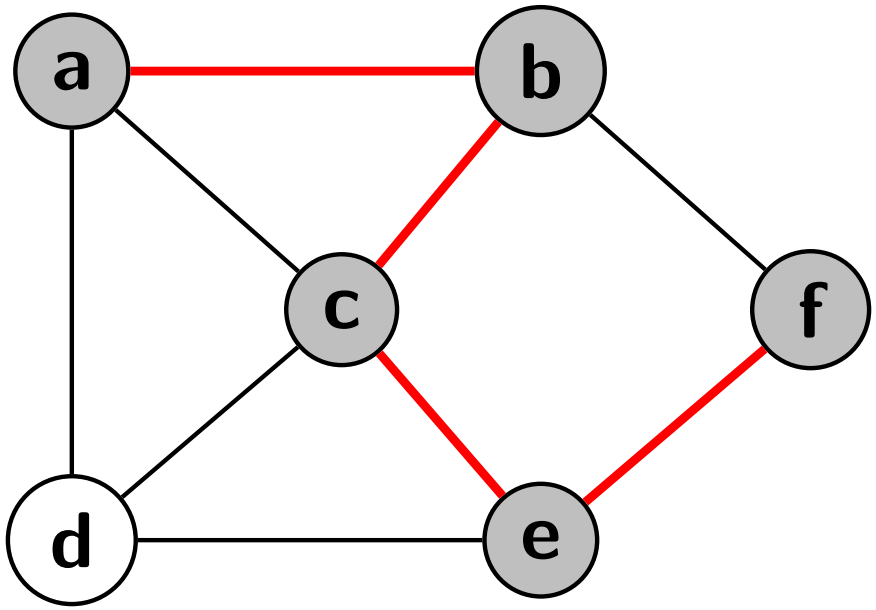
$$(\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{e}, 0, 0, \mathbf{a})$$

Hamiltonian Cycle Example (Backtracking)



$$(\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{e}, \mathbf{f}, 0, \mathbf{a})$$

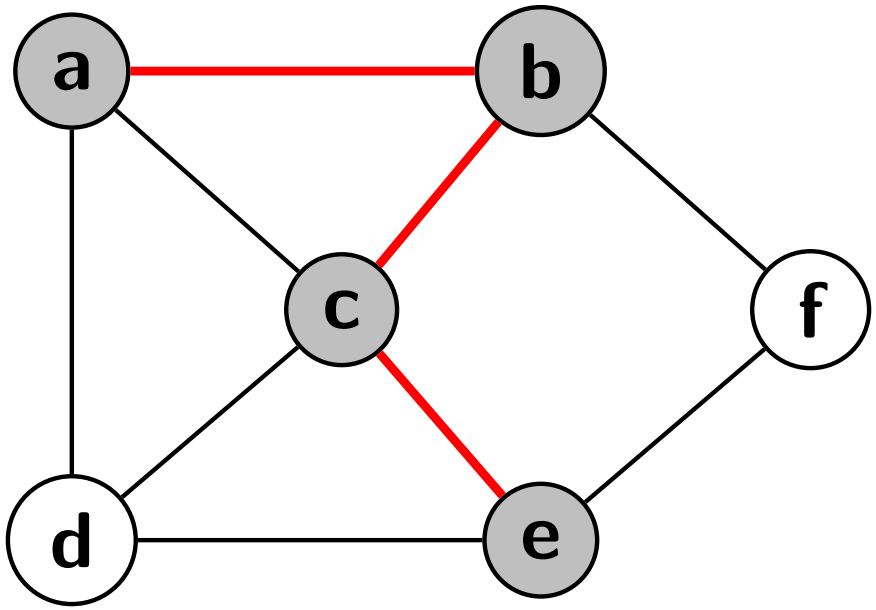
Hamiltonian Cycle Example (Backtracking)



$(\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{e}, \mathbf{f}, 0, \mathbf{a})$

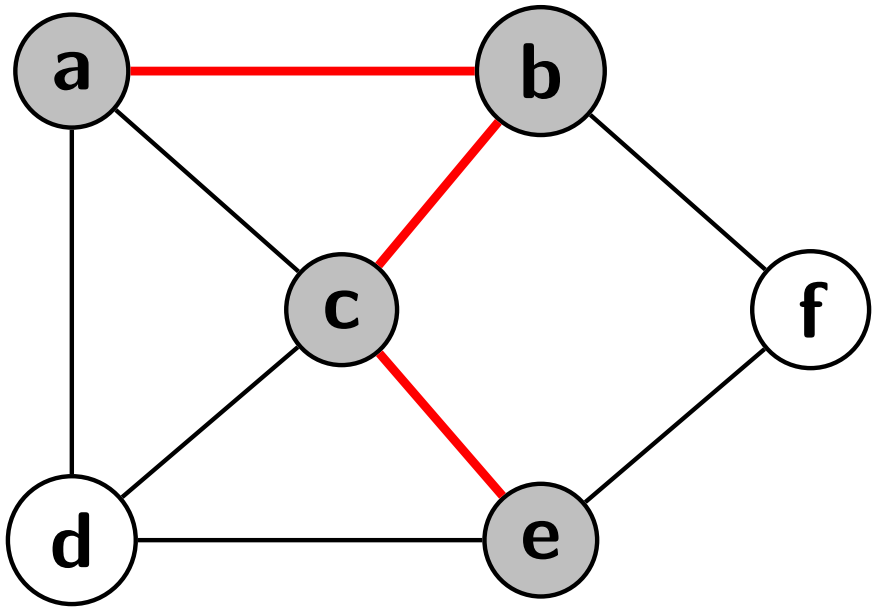
Backtrack!

Hamiltonian Cycle Example (Backtracking)



$$(\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{e}, 0, 0, \mathbf{a})$$

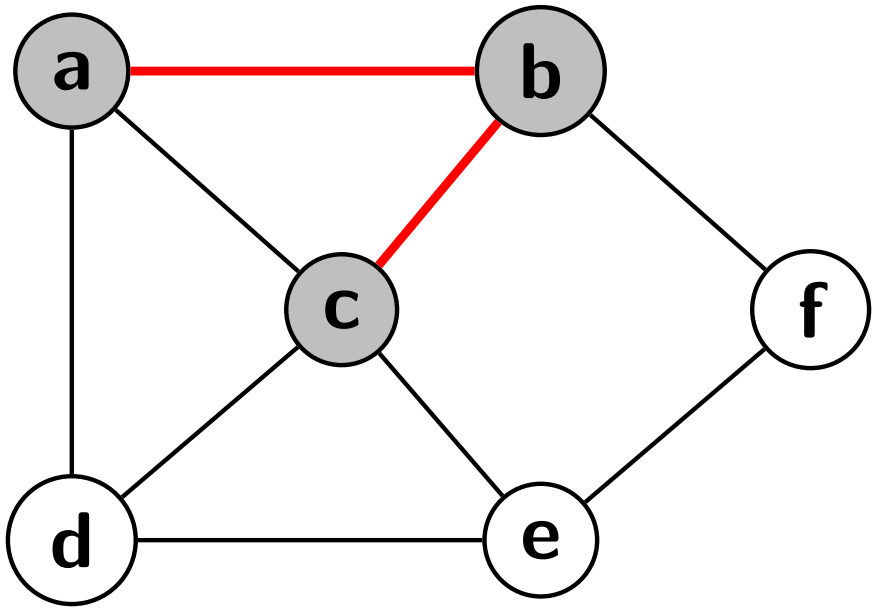
Hamiltonian Cycle Example (Backtracking)



$$(\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{e}, 0, 0, \mathbf{a})$$

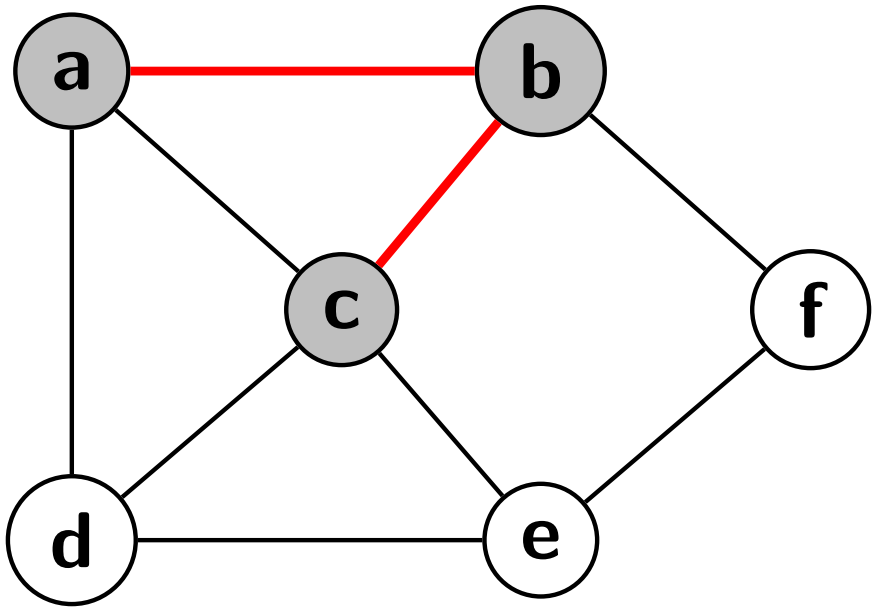
Backtrack!

Hamiltonian Cycle Example (Backtracking)



$$(\mathbf{a}, \mathbf{b}, \mathbf{c}, 0, 0, 0, \mathbf{a})$$

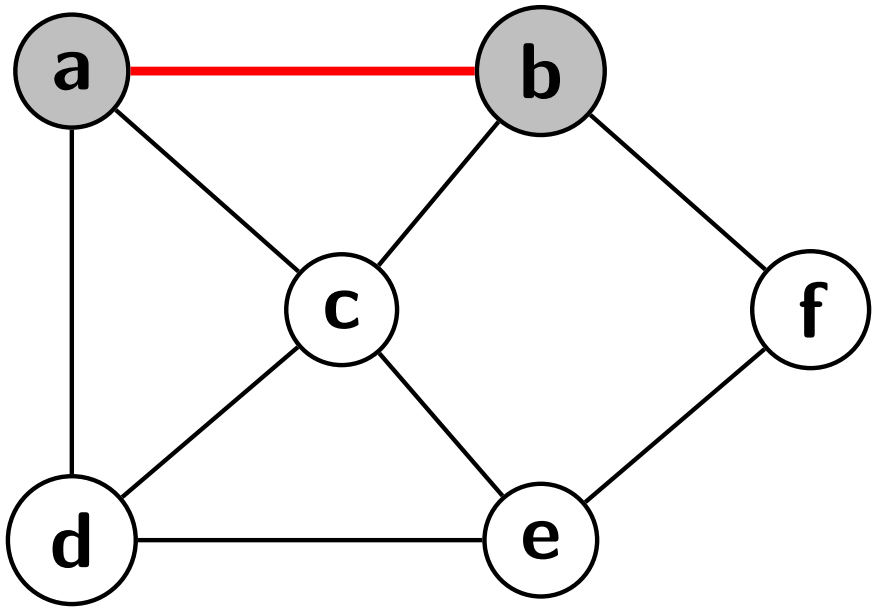
Hamiltonian Cycle Example (Backtracking)



$$(\mathbf{a}, \mathbf{b}, \mathbf{c}, 0, 0, 0, \mathbf{a})$$

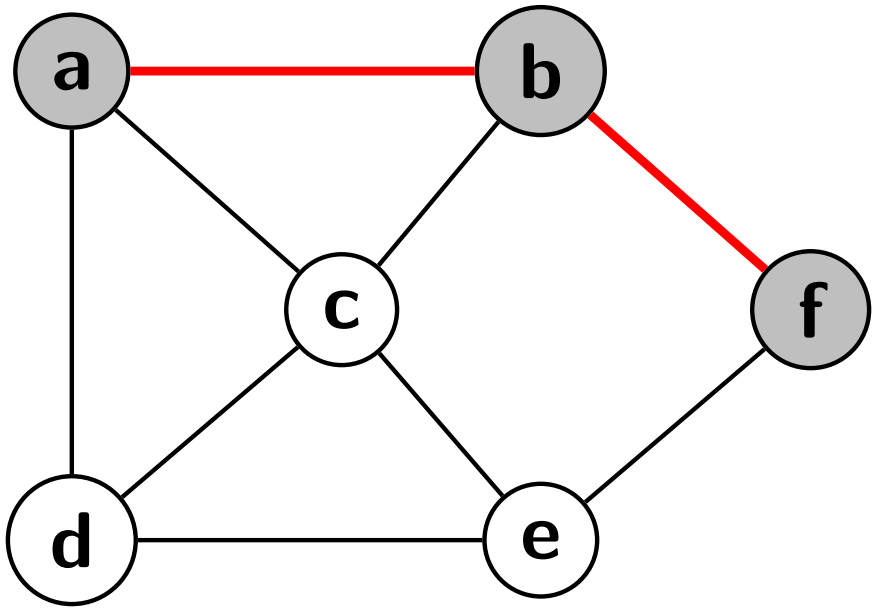
Backtrack!

Hamiltonian Cycle Example (Backtracking)



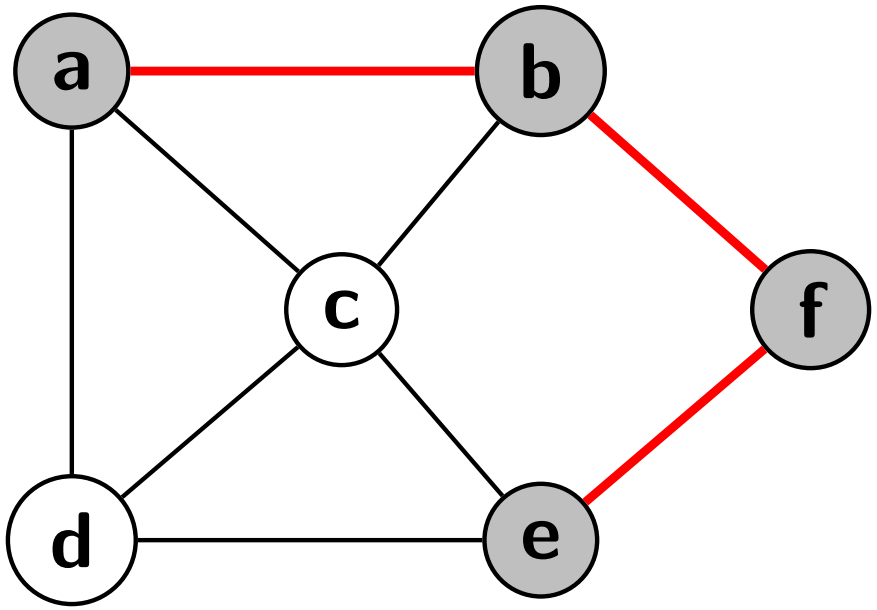
$$(\mathbf{a}, \mathbf{b}, 0, 0, 0, 0, \mathbf{a})$$

Hamiltonian Cycle Example (Backtracking)



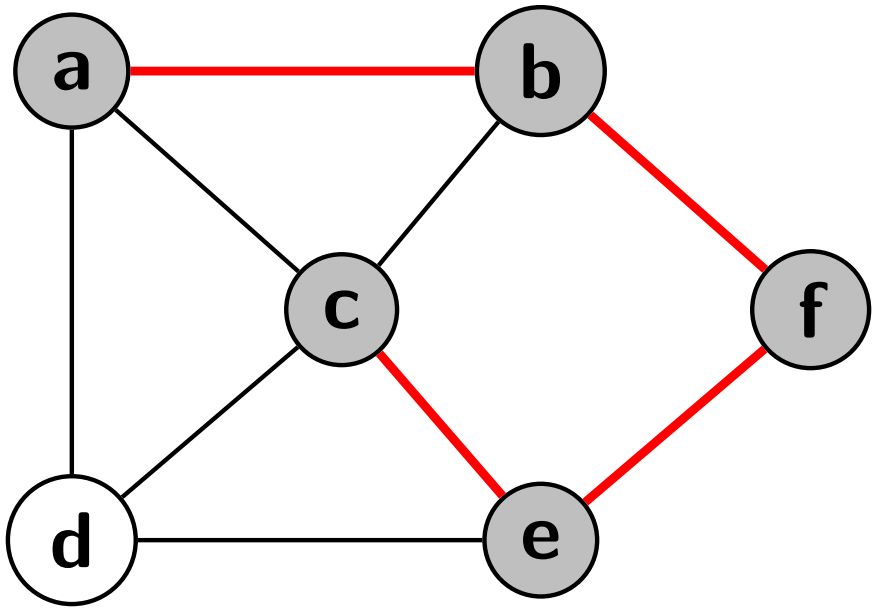
$$(\mathbf{a}, \mathbf{b}, \mathbf{f}, 0, 0, 0, \mathbf{a})$$

Hamiltonian Cycle Example (Backtracking)



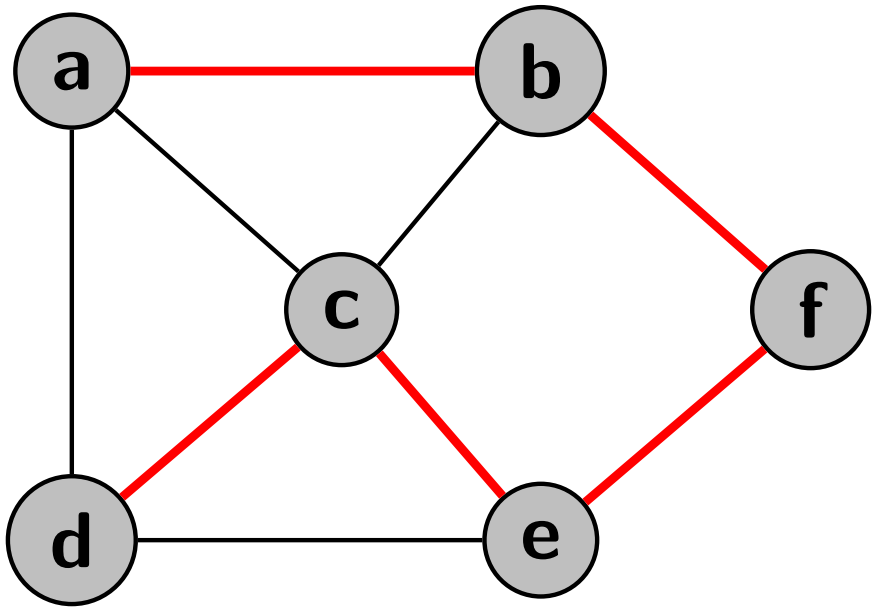
$$(\mathbf{a}, \mathbf{b}, \mathbf{f}, \mathbf{e}, 0, 0, \mathbf{a})$$

Hamiltonian Cycle Example (Backtracking)



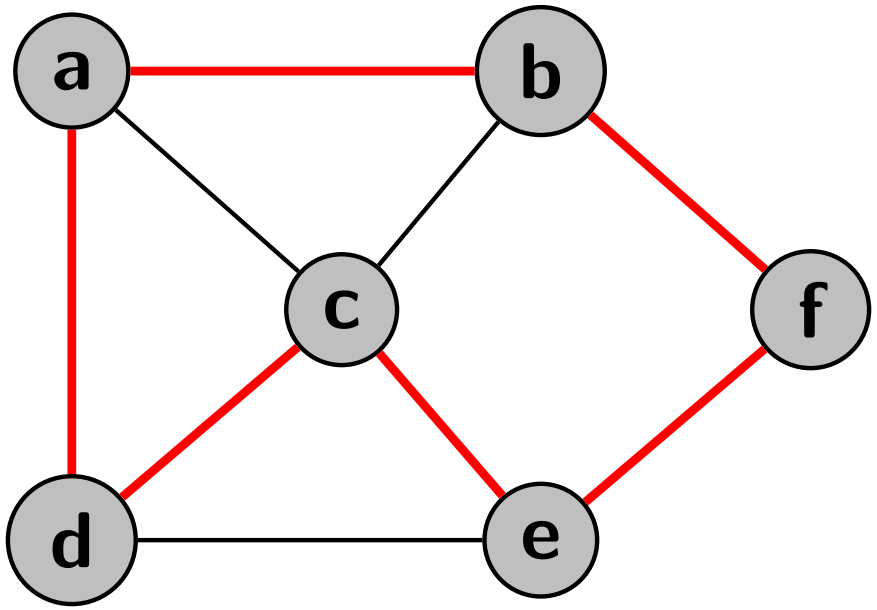
$$(\mathbf{a}, \mathbf{b}, \mathbf{f}, \mathbf{e}, \mathbf{c}, 0, \mathbf{a})$$

Hamiltonian Cycle Example (Backtracking)



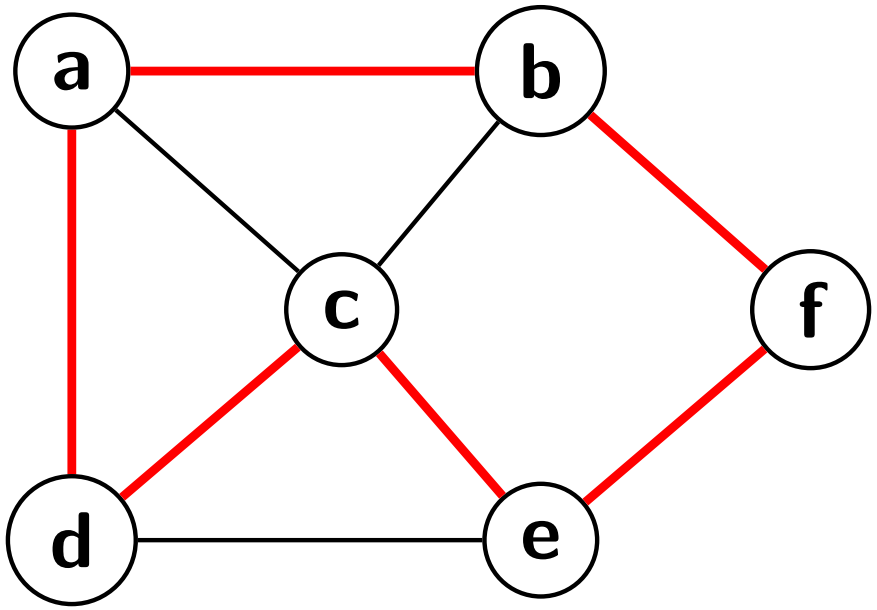
$(\mathbf{a}, \mathbf{b}, \mathbf{f}, \mathbf{e}, \mathbf{c}, \mathbf{d}, \mathbf{a})$

Hamiltonian Cycle Example (Backtracking)



$(\mathbf{a}, \mathbf{b}, \mathbf{f}, \mathbf{e}, \mathbf{c}, \mathbf{d}, \mathbf{a})$

Hamiltonian Cycle Example (Backtracking)



Hamiltonian cycle : $(\mathbf{a}, \mathbf{b}, \mathbf{f}, \mathbf{e}, \mathbf{c}, \mathbf{d}, \mathbf{a})$

Boolean Satisfiability (SAT) Problem

The Boolean satisfiability problem (also **SAT problem**) asks whether a given Boolean formula is satisfiable.

Boolean Satisfiability (SAT) Problem

The Boolean satisfiability problem (also **SAT problem**) asks whether a given Boolean formula is satisfiable.

E.g. is the following Boolean formula satisfiable?

$$(x_1 \wedge x_2 \vee x_3) \wedge (x_4 \wedge x_1 \vee x_3) \wedge \neg(x_4 \vee x_1)$$

That is, does there exist an assignment of truth values to the Boolean variables x_1, x_2, x_3, x_4 that makes the formula True?

Boolean Satisfiability (SAT) Problem

The Boolean satisfiability problem (also **SAT problem**) asks whether a given Boolean formula is satisfiable.

E.g. is the following Boolean formula satisfiable?

$$(x_1 \wedge x_2 \vee x_3) \wedge (x_4 \wedge x_1 \vee x_3) \wedge \neg(x_4 \vee x_1)$$

That is, does there exist an assignment of truth values to the Boolean variables x_1, x_2, x_3, x_4 that makes the formula True?

Yes! $x_1 =$ False, $x_2 =$ True, $x_3 =$ True, $x_4 =$ False.

Boolean Satisfiability (SAT) Problem

The Boolean satisfiability problem (also **SAT problem**) asks whether a given Boolean formula is satisfiable.

E.g. is the following Boolean formula satisfiable?

$$(x_1 \wedge x_2 \vee x_3) \wedge (x_4 \wedge x_1 \vee x_3) \wedge \neg(x_4 \vee x_1)$$

That is, does there exist an assignment of truth values to the Boolean variables x_1, x_2, x_3, x_4 that makes the formula True?

Yes! $x_1 =$ False, $x_2 =$ True, $x_3 =$ True, $x_4 =$ False.

The SAT problem was the first example of an **NP-Complete** problem (*Cook-Levin theorem*, 1971).

Boolean Satisfiability (SAT) Problem

Just as with Linear Programming, we can encode **many** problems in terms of Boolean satisfiability (SAT).

Problems in circuit design can often be cast as SAT problems.

An output from a wire in a digital circuit corresponds to a Boolean formula whose variables correspond to the input wires (along the wires $1 =$ True, $0 =$ False).

One may e.g. model a *multiplier circuit* as a Boolean formula, which permits one to ask if there are inputs to the circuit that result in the number 21 (10101 in binary) in the output.

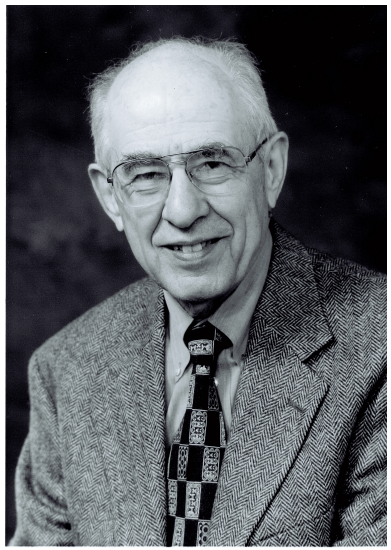
This is equivalent to asking "is 21 a prime number?".

DPLL Algorithm

Davis–Putnam–Logemann–Loveland (DPLL) Algorithm

- Introduced in 1962 as a refinement of an earlier 1960 algorithm by Davis and Putnam.
- A backtracking-based algorithm for solving the **SAT problem**.
- Worst case running time is exponential.
- Serves as the backbone of modern **SAT solvers** (e.g. MiniSAT, Z3).
- Major practical applications in *hardware verification*.

Martin Davis



Hilary Putnam

Backtracking (Summary)

- Backtracking is a standard algorithm for solving constraint problems with large search spaces.
- It can take an **exponential** amount of time, however with many constraints it will often find solutions relatively quickly.
- A backtracking algorithm does not solve, for example, Sudoku in the same way as a human would.
- We can often speed up backtracking by adding *more constraints* (although, this can make writing the program more involved).

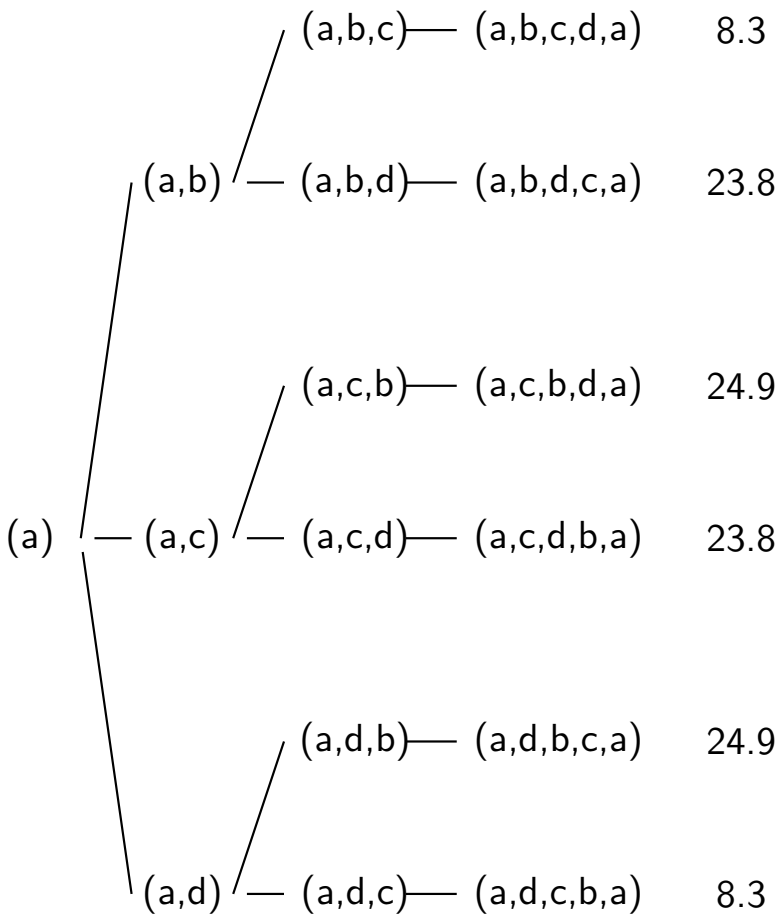
Optimisation Problems

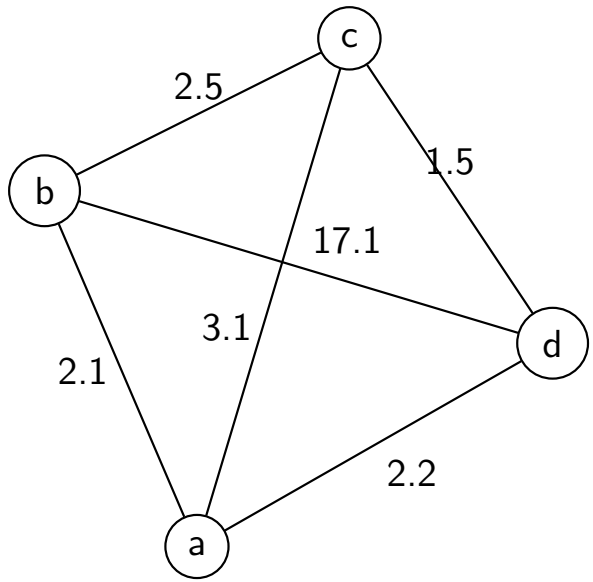
- In many **optimisation problems** (TSP, graph-colouring, etc.) we have a huge search space ($n!$, k^n).
- However, in optimisation problems we're less interested in constraints.
- If we are interested in finding an optimum, we can use the cost as a constraint!
- Any partial solution has to have a lower cost than the best solution we have found so far.
- This allows us to develop a backtracking strategy known as **branch and bound**.

Branch and Bound

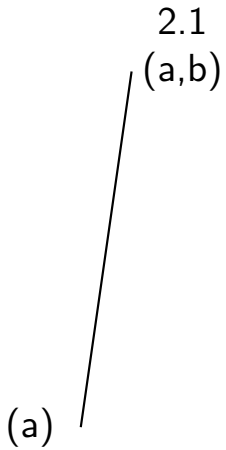
- **Branch and bound** is used for optimisation problems where efficient strategies just won't work.
- It beats exhaustive enumeration by eliminating many possible solutions without having to enumerate them all.
- Branch and bound can be *slow* as the constraints aren't necessarily very strong.
- We can sometimes strengthen the constraints, thus eliminating more of the search space.
- This strategy usually works well on fairly small problems, but often fails on very large problems.

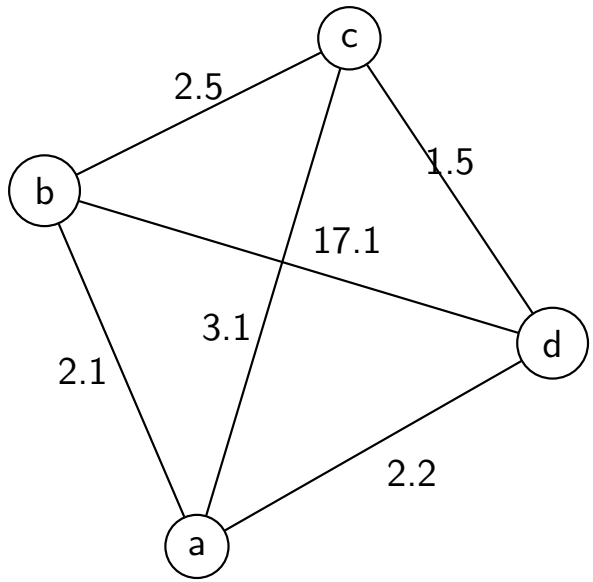
Travelling Salesman Problem (Brute Force Search)



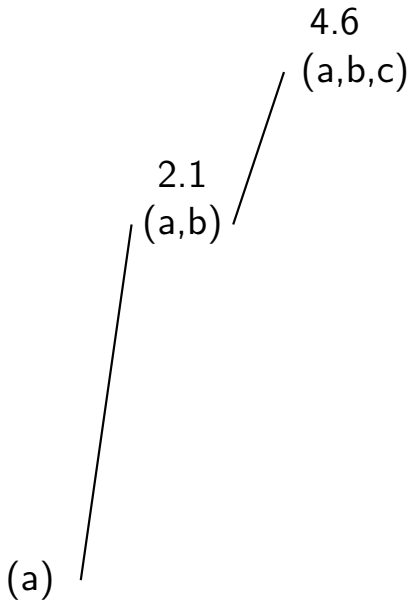


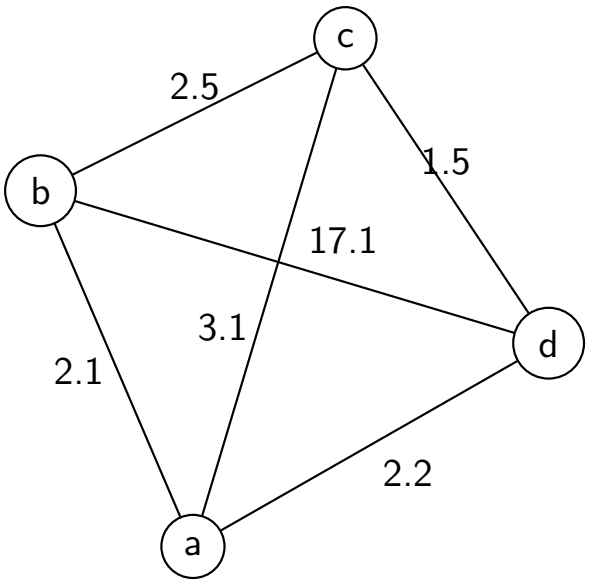
Travelling Salesman Problem (Branch and Bound)



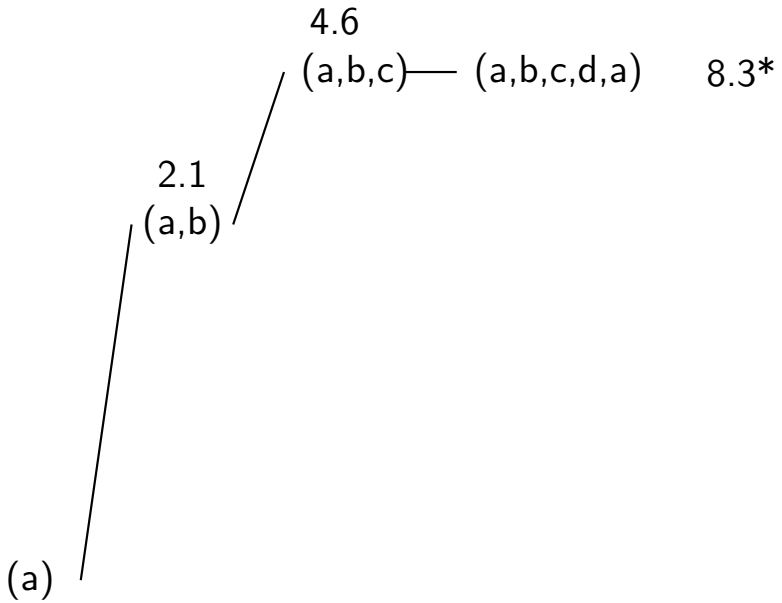


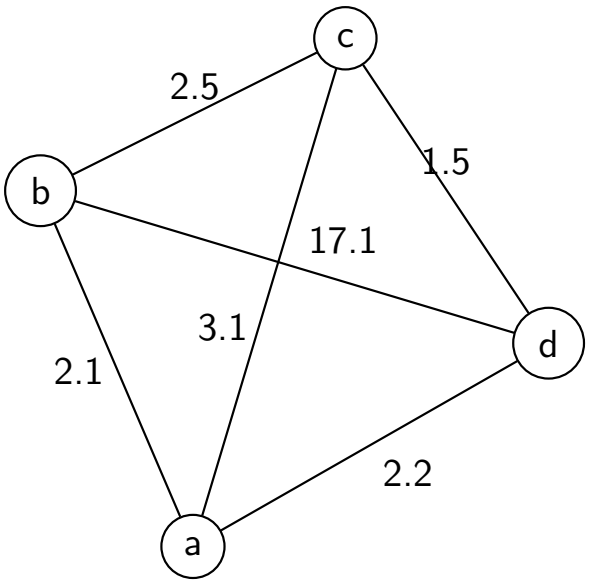
Travelling Salesman Problem (Branch and Bound)



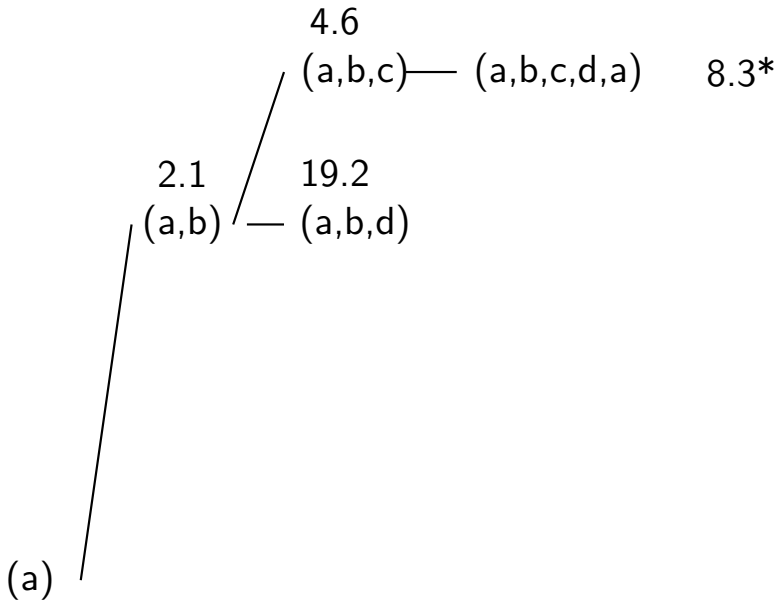


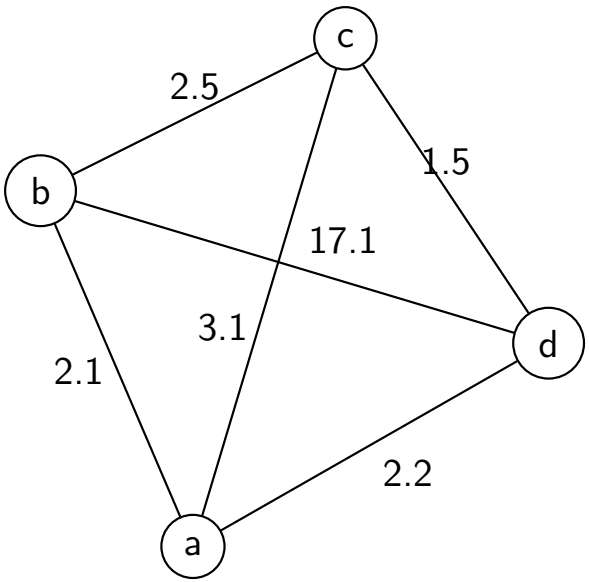
Travelling Salesman Problem (Branch and Bound)



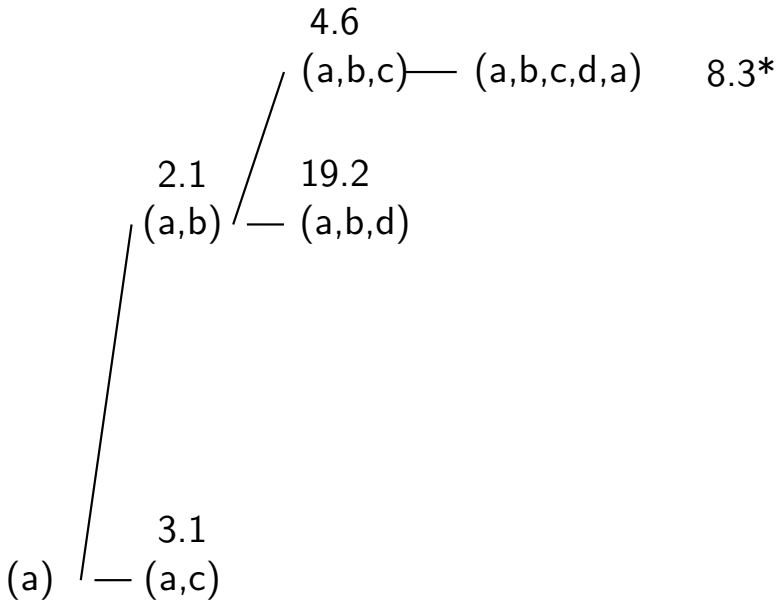


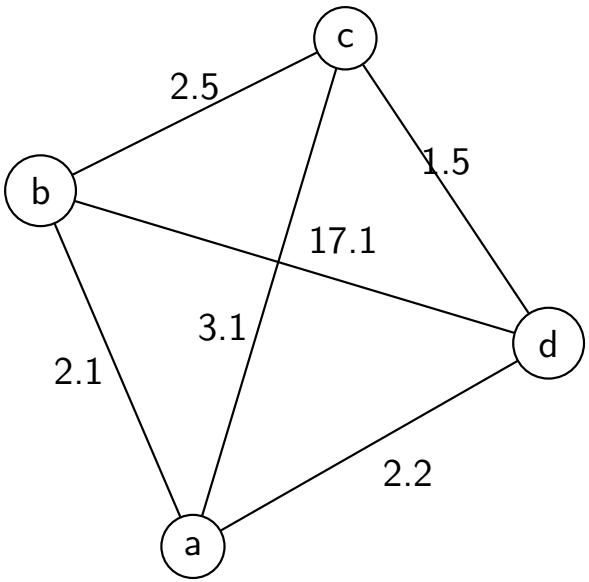
Travelling Salesman Problem (Branch and Bound)



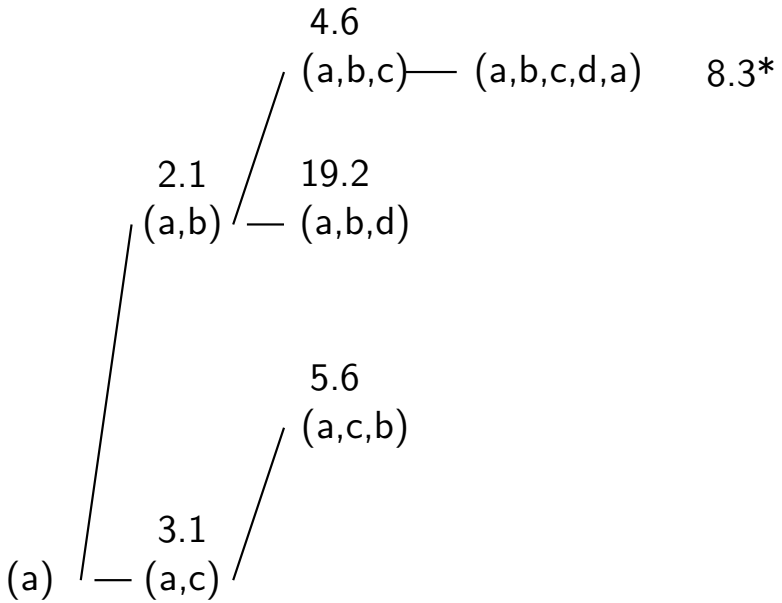


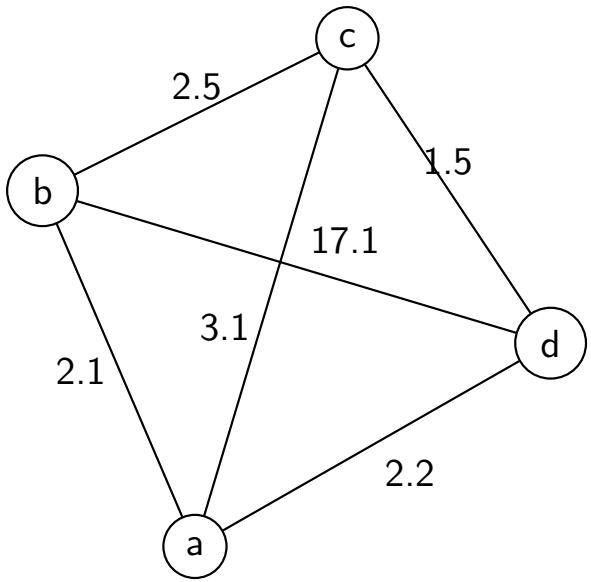
Travelling Salesman Problem (Branch and Bound)



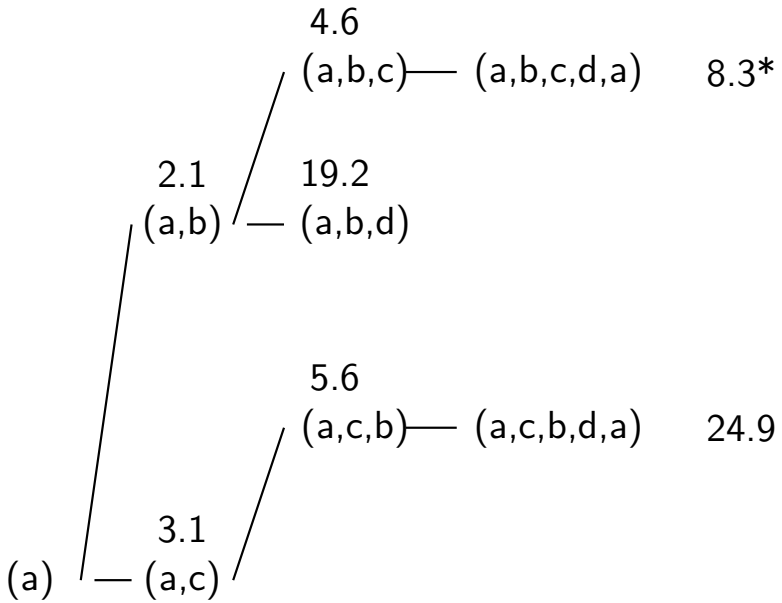


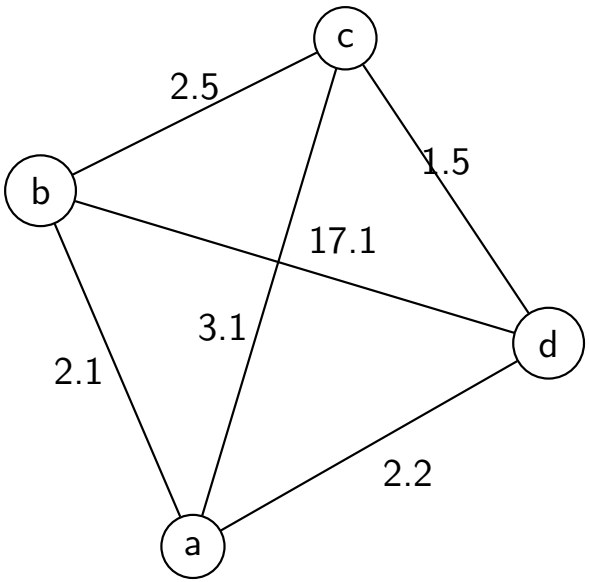
Travelling Salesman Problem (Branch and Bound)



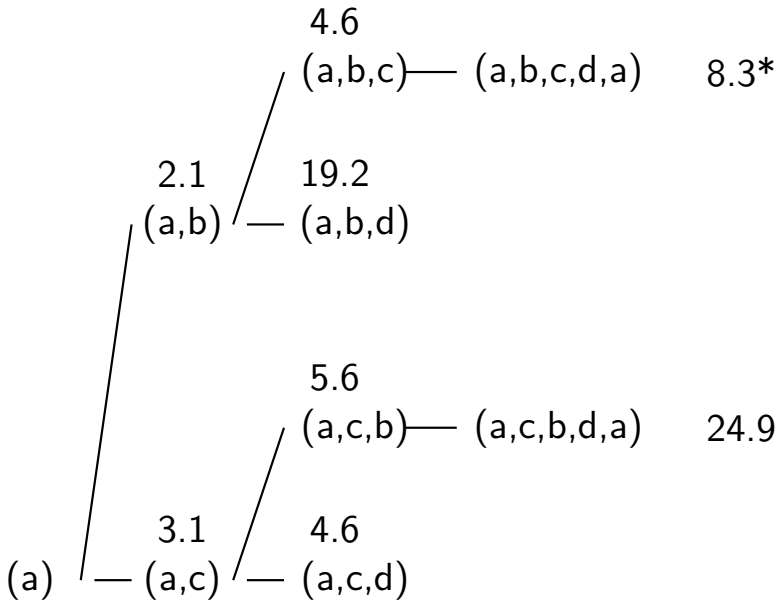


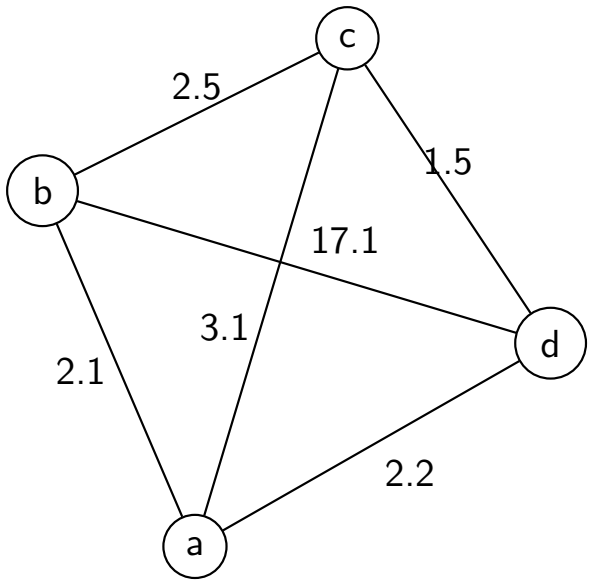
Travelling Salesman Problem (Branch and Bound)



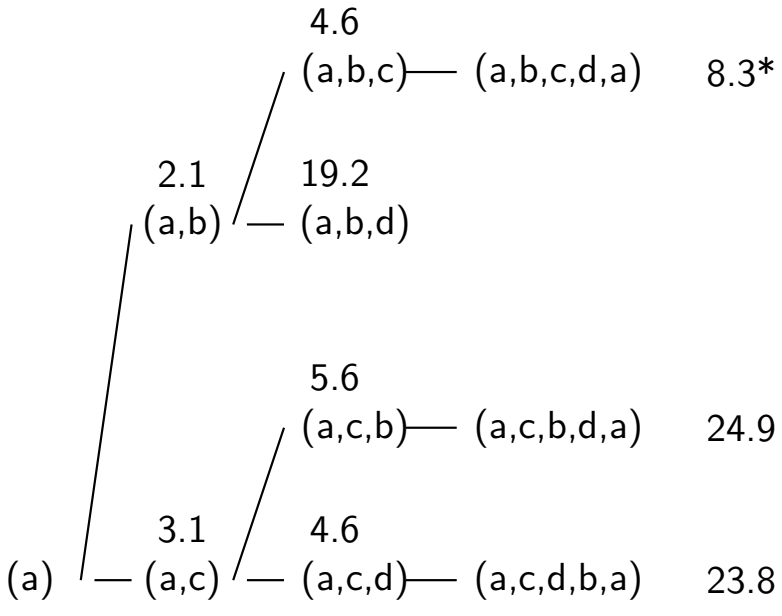


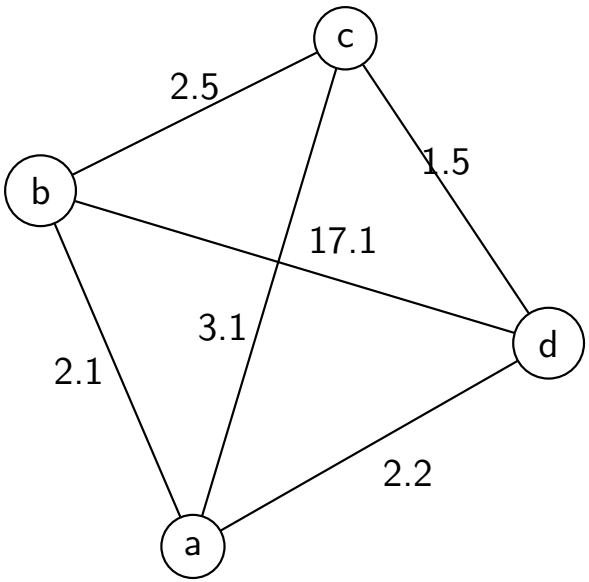
Travelling Salesman Problem (Branch and Bound)



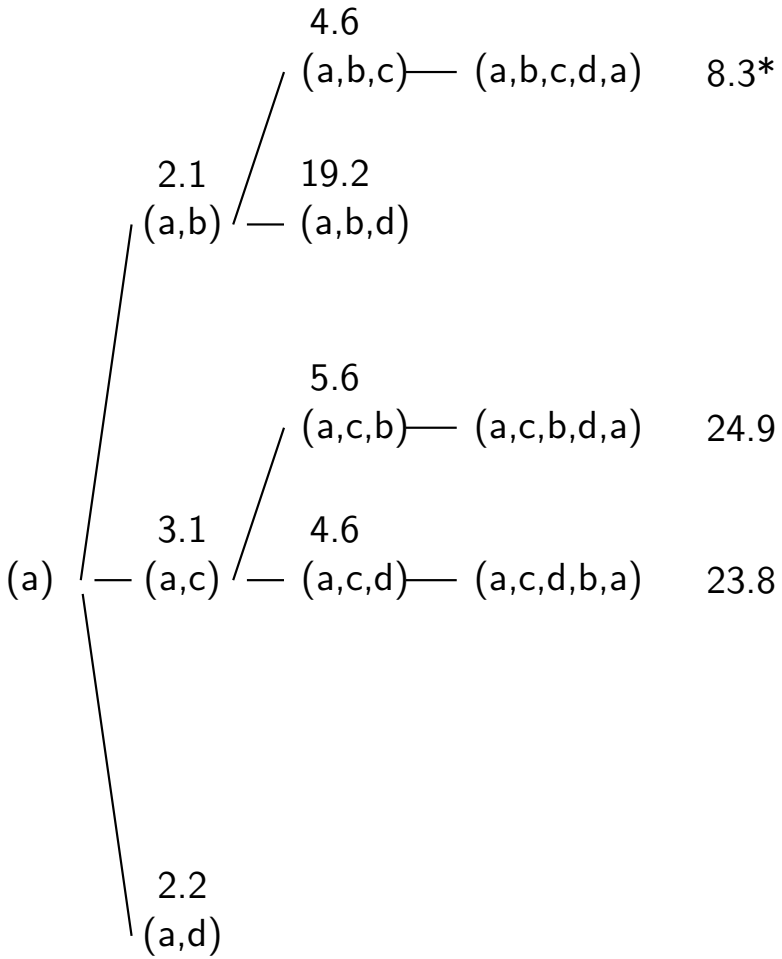


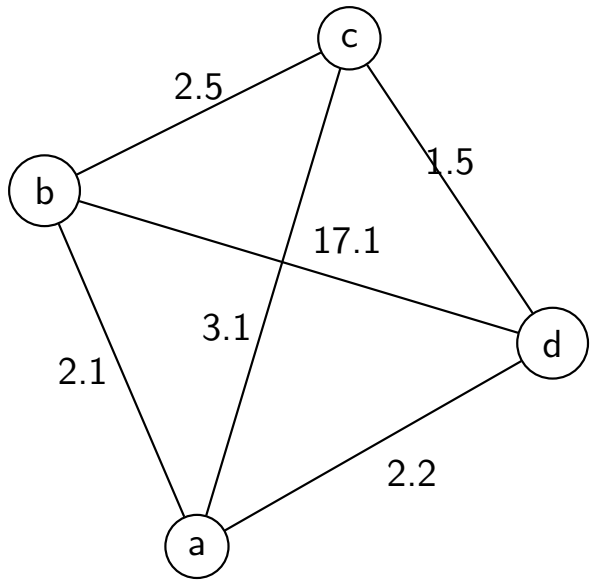
Travelling Salesman Problem (Branch and Bound)



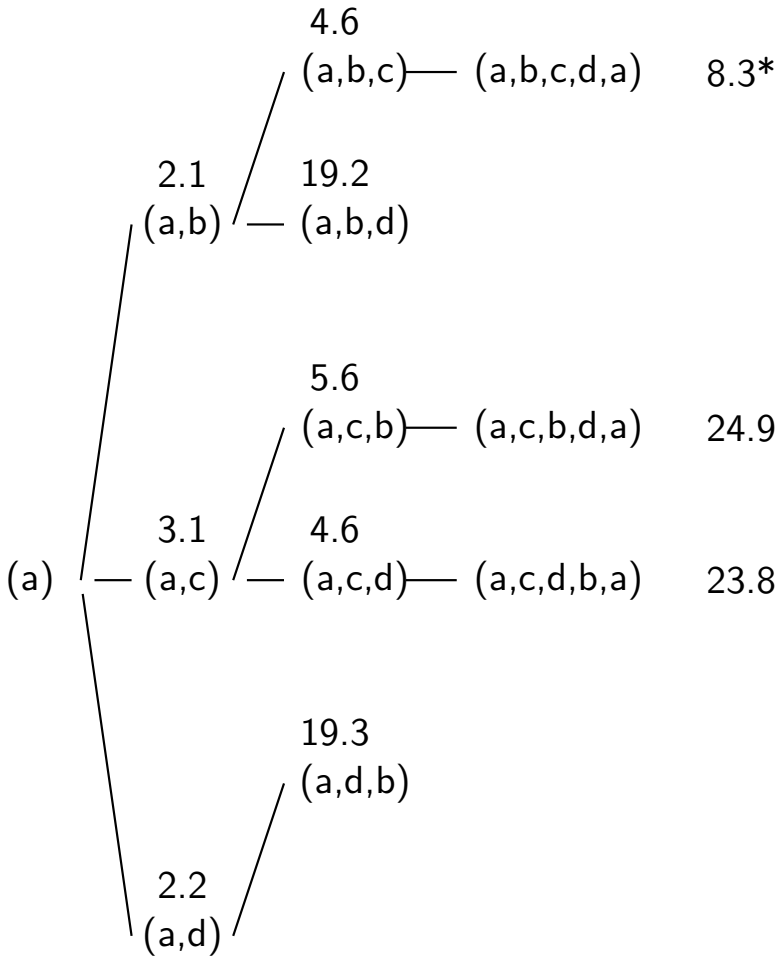


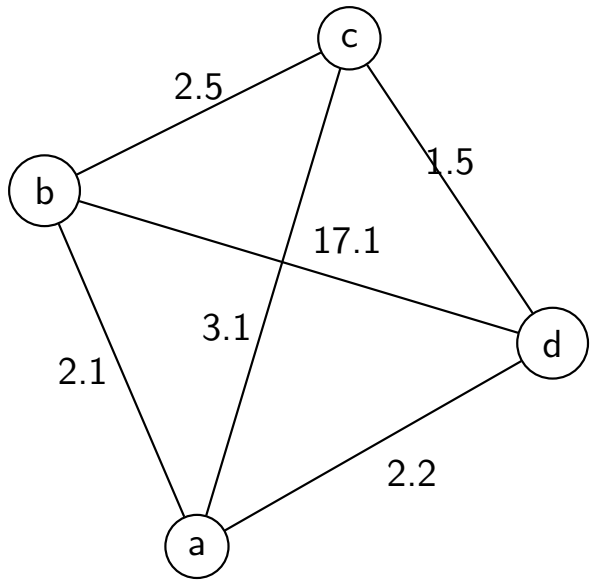
Travelling Salesman Problem (Branch and Bound)



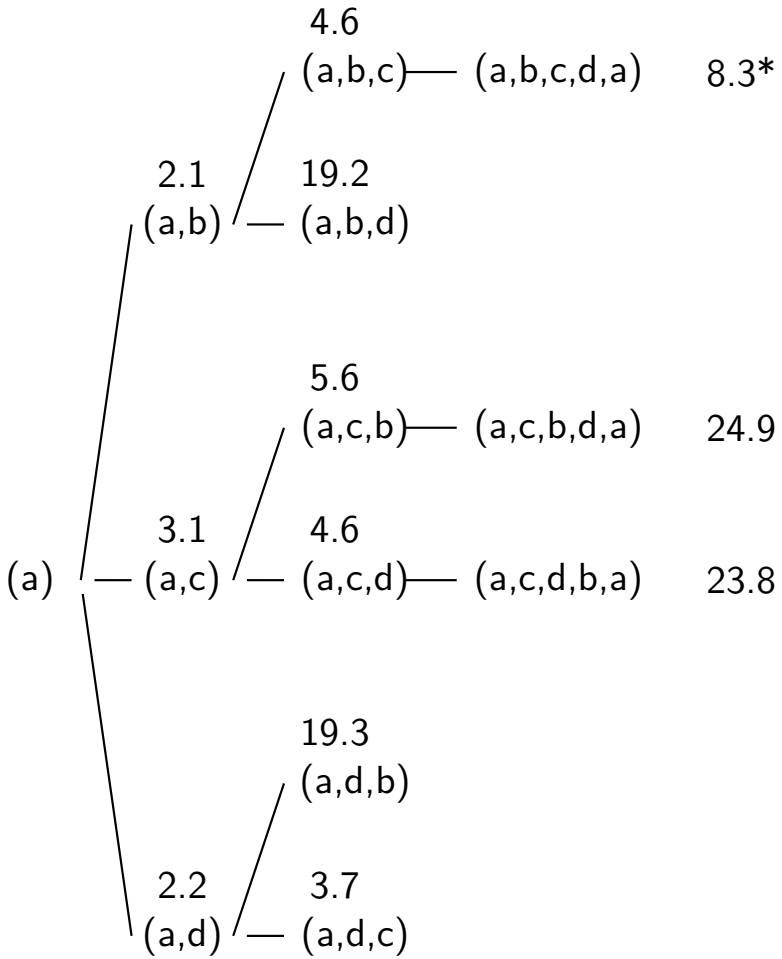


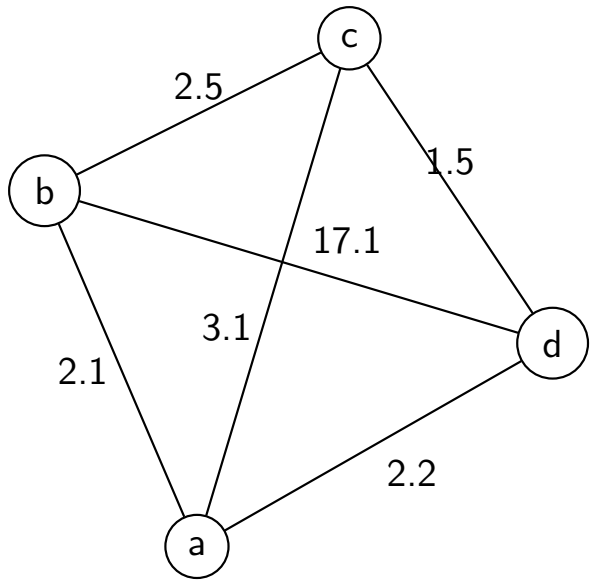
Travelling Salesman Problem (Branch and Bound)



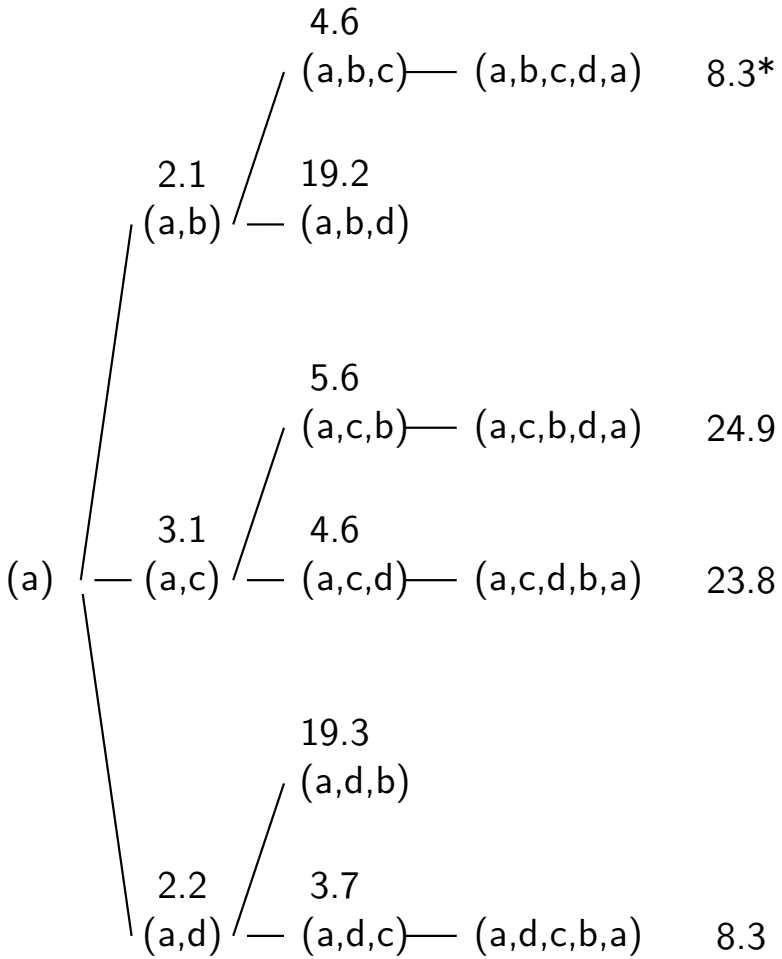


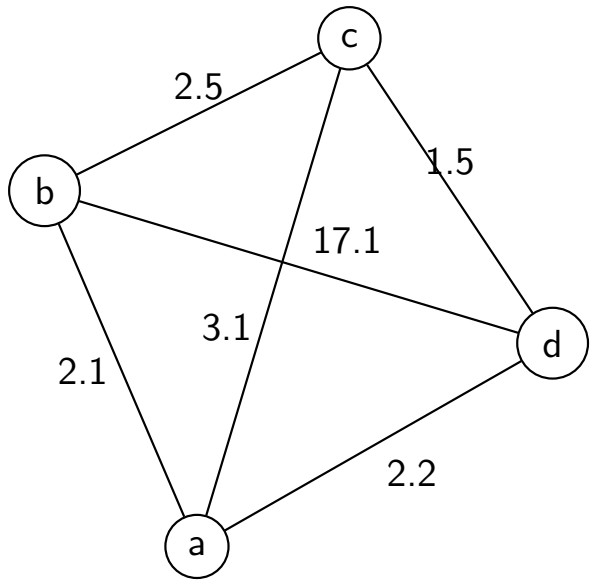
Travelling Salesman Problem (Branch and Bound)





Travelling Salesman Problem (Branch and Bound)





Travelling Salesman Problem (Branch and Bound)

- In practice, branch and bound can be improved by using information specific to the problem.
- For example, we can halve the search space for the 2-d version of TSP by considering only one direction: e.g. by insisting that we visit city a before city b.
- We can employ useful heuristics to get a good starting bound (e.g. by using a greedy algorithm).
- We can also obtain lower bounds on partial solutions to the TSP by observing that all unvisited cities will need to be visited and use their *minimum spanning tree* to obtain a lower bound.

Applications of Branch and Bound

- Branch and bound works for **many optimisation problems**.
- Its drawback is that you often end up still searching an exponentially large search space, even though it might be massively faster than a brute force enumeration method.
- To make it work well requires considerable work.
- For really large problems, branch and bound may be too slow.

Other Search Strategies

- Search is a big topic in AI.
- The algorithms used depend on the information available.
- A classic search scenario is when there is heuristic information about where the optimum lies.
- Algorithms such as $\mathbf{A}^*$ will find the best route given an admissible heuristic as efficiently as possible.
- You should learn about this next year in AI.

Planning and Game Playing

- Search is also used to find the best action to take in planning problems and game playing (e.g. computer chess)
- Again, it is useful to think in terms of search trees.
- Searching all paths on the search tree is usually infeasible.
- We look for ways of pruning the search tree to focus on good moves.
- Strategies include minimax and alpha-beta pruning.

Lessons

- Search has **many useful applications**.
- It is helpful to consider the search space as a **tree** whose branches correspond to possible actions.
- Backtracking is useful in search trees with constraints.
- For optimisation problems, **branch and bound** uses and costs of partial solutions as constraints.
- Widely applicable technique, but may take too long for large problems.

Further Reading:

Optional :

- **Rina Dechter and Daniel Frost** "Backtracking algorithms for constraint satisfaction problems", 1999 https://pdfs.semanticscholar.org/ad74e

Acknowledgements: Partly based on earlier COMP 1201 slides by Dr Adam Prügel-Bennett, University of Southampton.