Chapter 22
Backtracking and Branch and Bound

22.1 Backtracking

※ 22.1.1 *Introduction to Backtracking*

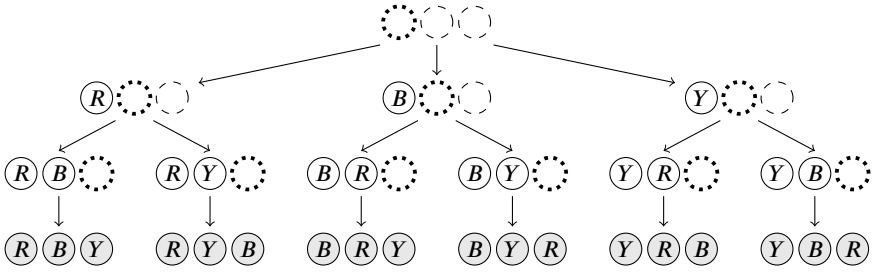
Backtracking is an algorithm family that can be used to solve *constraint satisfaction* problems. Unlike optimization problems, which were introduced in the previous chapter, **constraint satisfaction problems** are only concerned with whether there *exists* a solution that satisfies all constraints, and not what the best solution is. The backtracking approach can often be applied to solve problems that ask you to either find a single solution or enumerate over all solutions that satisfy a given set of constraints.

Similar to the brute force approach, backtracking algorithms are designed to explore all possible solutions to determine if any of them satisfy a given set of constraints. However, unlike brute force, backtracking stops checking a partial solution *as soon as it violates any constraint*. This process of removing partial solutions that cannot lead to a valid solution is known as **pruning**.

To demonstrate how backtracking works, let's start with a simple example with no constraints. Suppose you have three marbles that you have to place in a straight line: a red marble (R), and blue marble (B), and a yellow marble (Y). Your goal is to write a function that prints out all the different ways these marbles can be placed, assuming they can be placed in any order.

The simplest way to solve this problem is to consider all possible choices that can be made at each step. For the first marble, we can either choose R, B, or Y. If we choose R as our first marble, we can choose B or Y as our second marble. If we choose B as our first marble, we can choose R or Y as our second marble. If we choose Y as our first marble, we can choose R or B as our second marble (and so on). Since we are making a series of decisions that bring us toward a solution, we can represent the solution set to the problem in the form of a tree, where each branch represents a choice that was made to reach a solution (this type of tree is known as a *state-space tree*).

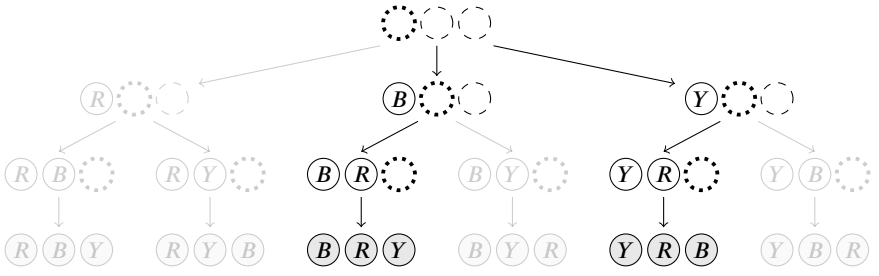
The state-space tree for the marble problem is shown below:



Since we have no constraints on how the marbles can be placed, we can obtain all orderings of these three marbles by performing a depth-first search on this tree. Every time we hit a leaf node, we know that we have reached a valid outcome of our problem.

Remark: Note that the state-space tree is not actually a physical tree that is stored in memory, but rather an abstract representation of the solution space of our problem. The backtracking algorithm does not necessarily involve physical tree objects; rather, the state-space tree is only used to demonstrate that the behavior of backtracking is similar to a depth-first search over a solution space, as if the solution space were represented as a tree where nodes represent partial input states and branches represent choices.

Now, let's add a constraint to this problem. Consider the same scenario as before, but this time, suppose the red marble must *always* be in the middle position. Notice how our tree changes when this constraint is added — certain branches no longer lead us to a valid solution! As a result, there is no need to explore these branches during our depth-first search. *This is the core difference between backtracking and brute force — brute force would check every branch, even ones that do not work, while backtracking would stop checking a branch as soon as it realizes the branch cannot lead to a viable solution.*

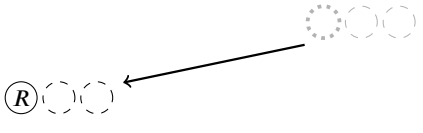


This idea forms the foundation of the backtracking approach, which performs a depth-first search on the solution space and *prunes* branches that cannot lead to a valid solution. Backtracking algorithms essentially boil down to the following four steps:

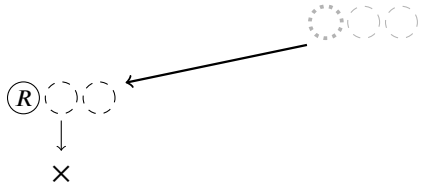
1. Make a choice that leads you toward a solution (which moves you down a level of the state-space tree).
2. Check if the choice is **promising** (that is, if it can lead you to a valid solution that satisfies the constraints). If it is, fix the choice you just made and recurse on the subproblem that remains — this allows you to explore further down the branch until you either obtain a valid solution or discover that the path no longer satisfies the given constraints.
3. Check if the current partial solution is a valid complete solution to the problem. If it is, you have a solution. If the problem only asks for one solution, you are done. If you are asked for all solutions, keep track of the solution you obtained.
4. Undo the choice you most recently made (i.e., backtrack) so that you can explore other branches.

To demonstrate this process, consider these steps when applied to the marble problem, under the constraint that the red marble must be in the middle. First, our algorithm will make a choice. It doesn't matter what choice we make as long as it brings us closer to a solution, so for the sake of simplicity, we will always choose a marble for the leftmost position available.

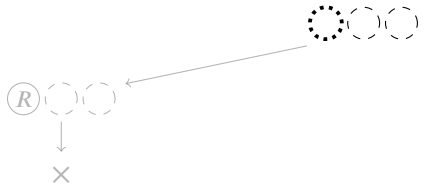
There are three choices we can make here: we can either place the red marble, the blue marble, or the yellow marble in the first position. Each of these choices represents a branch of our state-space tree. For our example, we will choose the red marble first (you can choose any marble here since all the branches will need to be explored eventually, but for consistency, we will go with the order R, then B, then Y):



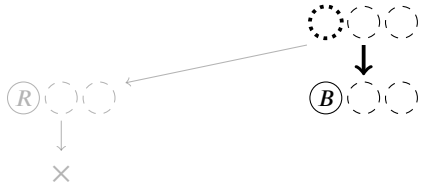
After making a choice, we then check if the choice is promising. In this case, the choice is *not* promising, as it violates the constraint that the red marble must be in the middle. Thus, we can *prune* all the solutions down this path, since we know that none of them will yield a valid solution.



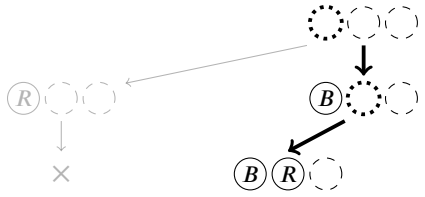
Now, we backtrack by undoing the choice we just made. This allows us to explore another branch by making a different choice.



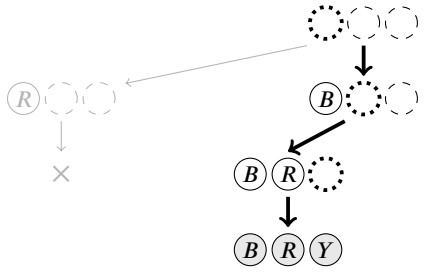
We then repeat this process until we have considered all viable branches of the tree. Since we have already considered the choice of the red marble, we will now choose the blue marble for the leftmost position.



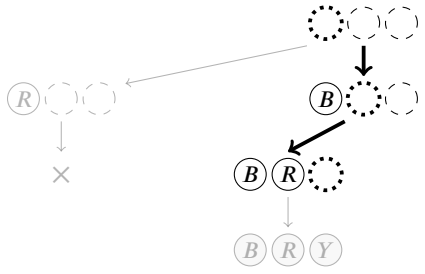
Again, we will check if the choice is promising. Here, the choice is indeed promising, as it does not violate any constraints. Thus, we will recurse down this branch by fixing the blue marble in place and making a choice for the marble in the middle. There are two choices we can make: the red marble and the yellow marble. Here, we will first consider the red marble.



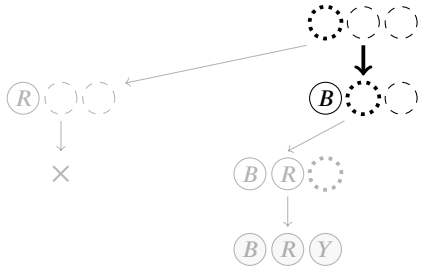
This choice is promising, so we recurse down the branch by making a choice for the final marble. Only the yellow marble is left, so we will choose the yellow marble for this position.



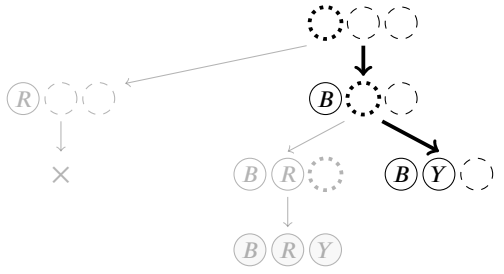
We have made a choice for all the positions and thus have a full solution. This solution does not violate any constraints, so it is a valid solution for our problem. If our goal was to simply discover if a solution exists, then we would be done! However, since we want to enumerate over all solutions, we will continue searching. To do so, we keep track of this solution (usually in a container of solutions) and then undo the choice of the final marble so that we can explore other branches.



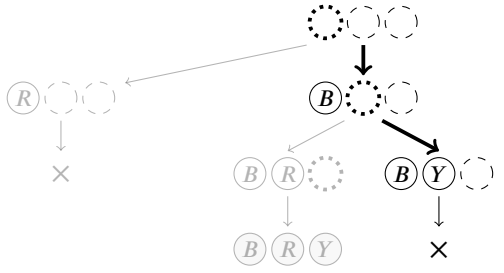
We have considered all possible choices for the third marble, so there are no more branches we have to consider for the case where B and R take up the first two positions. We then undo the choice of the second marble to move up a level of the tree.



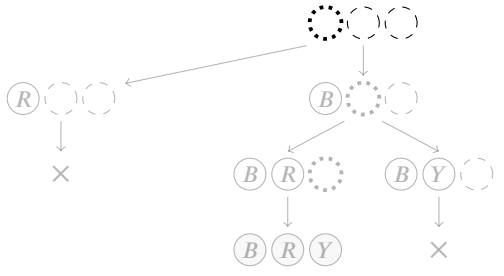
Since we have finished considering the red marble in the middle position, we will now consider the yellow marble.



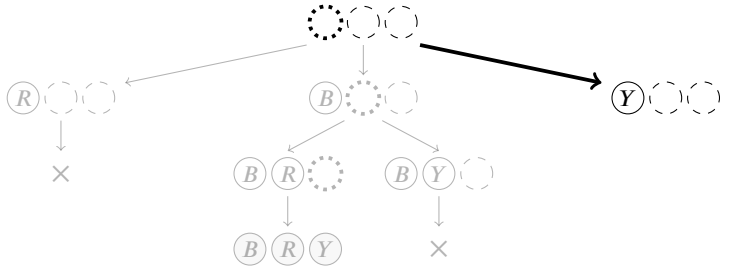
However, this choice is *not* promising, so we will prune off all solutions that begin with the blue and yellow marbles.



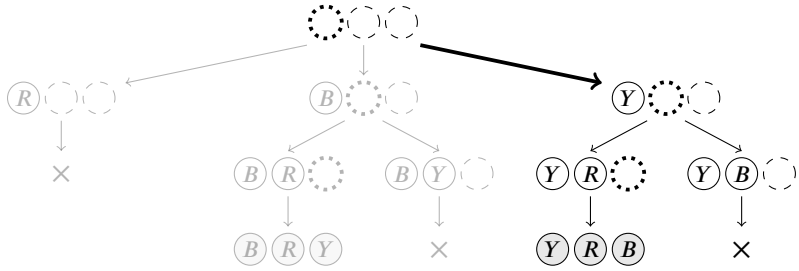
We then undo the choice and return back to the state where only the blue marble has been placed. At this point, we have considered all possible scenarios where the blue marble is in the first position, so we can undo that choice as well.



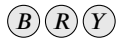
The last marble we still need to consider for the first position is the yellow marble, so we will select it to explore our final branch.

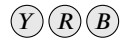


This choice does not break any constraints, so we will continue recursing down its branch. Using a similar approach as before, we will consider both *R* and *B* as the second marble. The branch with *R* as the second marble leads to a valid solution, while the branch with *B* as the second marble gets pruned since it breaks the given constraint.



Thus, there are two valid solutions to the marble problem that satisfy the constraint that the red marble must be in the middle:





※ 22.1.2 *Backtracking Structure*

The good news about backtracking solutions is that they often share a similar structure, so the logic that is used to solve one backtracking problem can often be applied to solve other backtracking problems. When writing a backtracking algorithm, you typically want to do the following four things:

1. Identify the choice you need to make at each step to bring you closer to a solution.

Identifying the *choice* allows you to conceptualize the state-space tree for a backtracking problem. This step is usually relatively straightforward and can be inferred from the type of problem you are given (such as adding a number to a running collection, or placing a piece down on a grid). A rule of thumb is to think about the choices you would need to make to solve the problem if you were brute forcing it by hand, and then apply that choice in a backtracking approach.

2. Devise a mechanism for checking whether a partial solution is a valid full solution.

You will need a way to determine whether a partial solution could potentially be a complete solution to the entire problem. For example, in the marble example from before, you would know that a solution is complete once all three marbles have been placed. If any position is empty, then the partial solution cannot be a full valid solution. This check is needed so that your algorithm knows when to store a solution or exit the search.

3. Devise a mechanism for checking whether a partial solution is promising.

You will need a way to determine whether a partial solution is promising, so that you can prune branches that cannot lead to a valid solution. This is often done by checking a partial solution against the given constraints. Implementing a promising check is not always trivial, and can even be the most complicated component of a backtracking algorithm. However, this process is also the most important, since a backtracking algorithm without any promising checks essentially degenerates to brute force.

4. Using the three features above, implement a function for exploring the solution space.

For backtracking problems, this is often done by writing a recursive function that performs the choices needed to extend a partial solution and then checks that partial solution against the given constraints. If you have all of the things above, you can typically implement this function using one of the following two templates. Here, `check_node()` checks the current partial solution (the function is named that way because each partial solution is a node of a backtracking problem's state-space tree).

Template 1 (Promising Check Before Recursive Call):

```
Algorithm check_node(partial_solution):
  if partial_solution is a valid full solution:
    store solution
  else:
    if (promising(partial_solution)):
      for each choice that can be made to extend partial_solution:
        partial_solution = partial_solution + choice
        check_node(partial_solution)
        partial_solution = partial_solution - choice
```

Template 2 (Promising Check After Recursive Call):

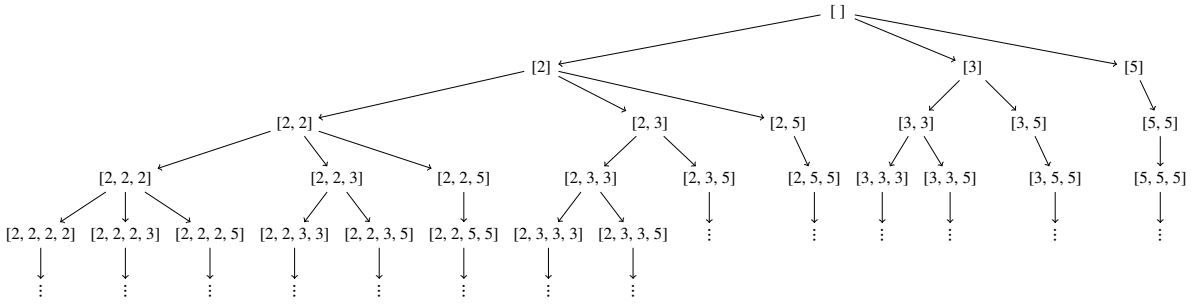
```
Algorithm check_node(partial_solution):
  if partial_solution is not promising:
    return
  if partial_solution is a valid full solution:
    store solution
  else:
    for each choice that can be made to extend partial_solution:
      partial_solution = partial_solution + choice
      check_node(partial_solution)
      partial_solution = partial_solution - choice
```

Both templates accomplish in the same thing; the only difference is that the first template checks if a partial solution is promising *before* a recursive call is made, while the second template checks if a partial solution is promising *after* a recursive call is made. Either version can be used, but depending on the problem you are trying to solve, one may be easier to implement than the other. To demonstrate this strategy in action, we will look at a few examples of backtracking problems over the next few pages.

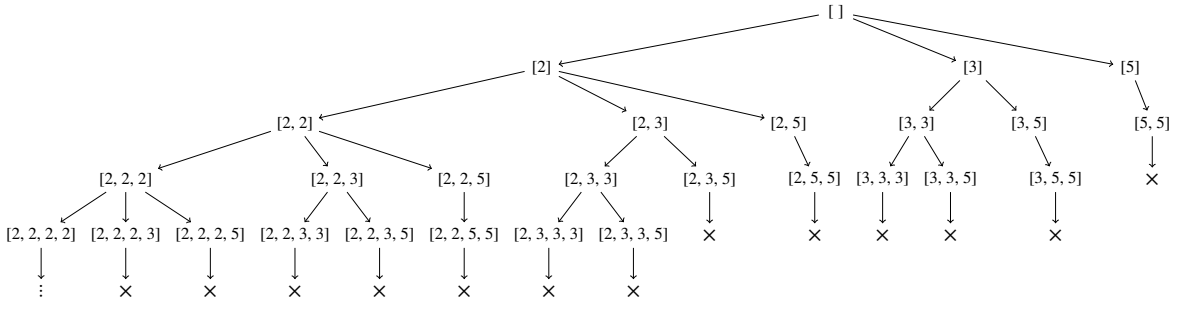
※ 22.1.3 *Solving Problems Using Backtracking: Combinations*

Example 22.1 You are given an vector of distinct positive integers, `nums`, as well as a target integer `target`. Your goal is to write a function that returns a list of all *unique* combinations of numbers in `nums` that sum to `target` (you can return the solutions in any order). For example, if you are given the array `nums = [2, 3, 5]` and `target = 8`, you would return `[2, 2, 2, 2]`, `[2, 3, 3]`, and `[3, 5]`, since these are the unique combinations of numbers in `nums` that sum up to 8.

Since this is a constraint satisfaction problem that asks you to enumerate over all solutions that satisfy a given set of constraints, backtracking can be used to implement a solution. To implement a backtracking solution, we first want to think about the *choice* we should make at each step to bring us closer to a solution. In this case, the choice is to add a number from `nums` to a running combination of numbers, and there are three possible choices that we can make at each step: we can either add a 2, a 3, or a 5 to our running total. To ensure that we aren't considering duplicates (i.e., both `[2, 3]` and `[3, 2]`), we will not add any values whose index position in `nums` is less than the index of any value already in our running collection. The state-space tree for this problem is shown below:



In addition to the choice, we also need a way to check if a partial solution is a valid complete solution, and if a partial solution is promising. To determine if a partial solution is complete, we just need to check if its sum is equal to our target value of 8. Similarly, because the values in `nums` are all positive, we can check if a choice is promising by looking at the sum of the new partial solution once the new value is added — a branch can be pruned as soon as its sum exceeds the target value of 8.



Now that we know how to check for whether a partial solution is complete or promising, we also need a way to implement these checks efficiently. A naïve approach to implement our promising check would be to individually sum up every partial solution we encounter. However, this method is quite inefficient, since each summation takes linear time on the size of the partial solution, and we end up doing a summation for every partial solution we encounter in the tree. To prevent our algorithm from performing a summation at every step, we can keep track of a separate integer `remain` that stores how much we can add to a partial solution before it becomes too large. For example, if we are currently considering the partial solution `[2, 3]`, then `remain` would have a value of 3, since this is the most we can add to this partial solution without going over our target of 8. This allows us to complete a promising check in constant time, since we only need to check if `remain` is non-negative. Similarly, this additional variable allows us to easily check if a solution is complete, since a solution is complete only if `remain` is exactly 0.

Now that we have identified the choice we need to make and the procedure for checking whether the current solution is promising or complete, we can use the backtracking template to write a solution for the combination sum problem.

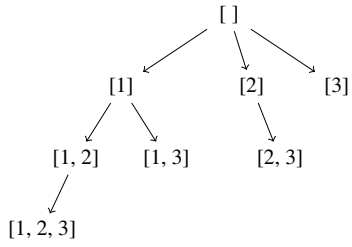
```
std::vector<std::vector<int32_t>> combination_sum(const std::vector<int>& nums, int32_t target) {
  std::vector<std::vector<int32_t>> solutions;
  std::vector<int32_t> current_partial_solution;
  check_sum(nums, target, solutions, current_partial_solution, 0);
  return solutions;
} // combination_sum()

void check_sum(const std::vector<int32_t>& nums, int32_t remain,
               std::vector<std::vector<int32_t>>& solutions,
               std::vector<int32_t>& current_partial_solution, size_t start_idx) {
  if (remain == 0) {
    solutions.push_back(current_partial_solution);
  } // if
  else {
    for (size_t i = start_idx; i < nums.size(); ++i) {
      current_partial_solution.push_back(nums[i]);
      if (nums[i] <= remain) {
        check_sum(nums, remain - nums[i], solutions,  current_partial_solution, i);
      } // if
      current_partial_solution.pop_back();
    } // for i
  } // else
} // check_sum()
```

Example 22.2 Given a vector of unique integers `nums`, implement a function to return all possible subsets that can be constructed using this vector of integers (i.e., the power set). For example, given the vector `[1, 2, 3]`, you would return

```
[ [], [1], [2], [1, 2], [3], [1, 3], [2, 3], [1, 2, 3] ]
```

This problem is similar to the combination sum problem, where we add values to a running collection (using the same rules to ensure we do not check the same subset twice). However, in this case, we do not want to add values that have already been added, since a valid subset cannot contain duplicate elements. The state-space tree for this problem is shown below:



Notice that every node of the state-space tree is a subset of the original input array — thus, we will need to keep track of every partial solution in our final output container. Furthermore, there are no constraints that allow us to prune off any branches in the tree, so we will need to treat every branch as promising. The solution is shown below:

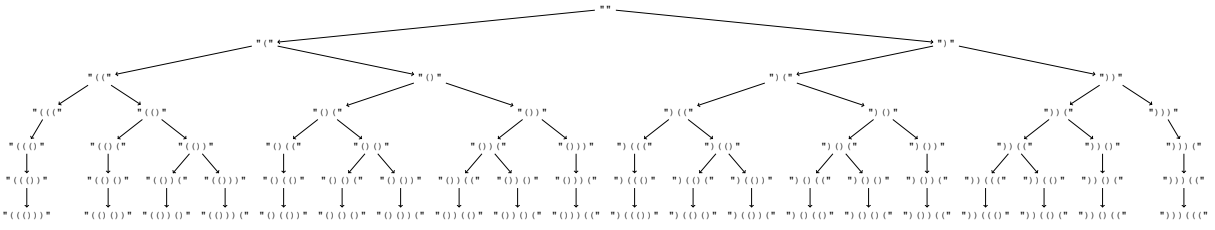
```
std::vector<std::vector<int32_t>> power_set(const std::vector<int32_t>& nums) {
  std::vector<std::vector<int32_t>> solutions;
  std::vector<int32_t> current_partial_solution;
  check_node(nums, solutions, current_partial_solution, 0);
  return solutions;
} // power_set()

void check_node(const std::vector<int32_t>& nums, std::vector<std::vector<int32_t>> &solutions,
                std::vector<int32_t>& current_partial_solution, size_t start_idx) {
  solutions.push_back(current_partial_solution);
  for (size_t i = start_idx; i < nums.size(); ++i) {
    current_partial_solution.push_back(nums[i]);
    check_node(nums, solutions, current_partial_solution, i + 1);
    current_partial_solution.pop_back();
  } // for i
} // check_node()
```

Example 22.3 Given a positive integer n, implement a function that returns all combinations of well-formed sequences that can be constructed using n pairs of parentheses. For example, given $n = 3$, you would return

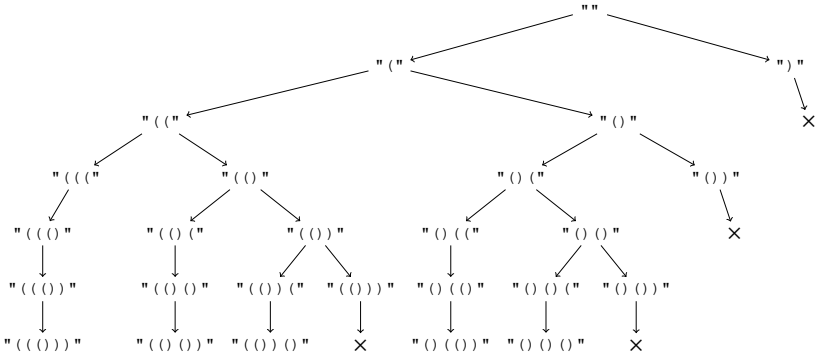
```
["((()))", "(()())", "(())()", "()(())", "()()()"]
```

Since this is a constraint satisfaction problem, it can be solved using a backtracking approach. Here, each choice would be to add a parenthesis (either left or right, as long as we have one available) to a running sequence of parentheses. The state-space tree for $n = 3$ is shown below:



Just by looking at the tree, it is easy to determine which branches are promising and which are not. However, when we actually implement this problem, we do not want to individually calculate whether a sequence of parentheses is balanced for every partial solution we encounter! Is there a faster way to determine whether a partial solution is promising, preferably in constant time?

To address this, notice that a sequence of parentheses is promising only if the number of left parentheses is greater than or equal to the number of right parentheses. In addition, the number of left parentheses in the sequence cannot exceed n. Thus, to efficiently determine if a partial solution is promising, we just need to keep track of the number of left and right parentheses present in any partial solution. If the number of right parentheses exceeds the number of left parentheses, or if the number of left parentheses exceeds n, we can prune the branch of the tree.

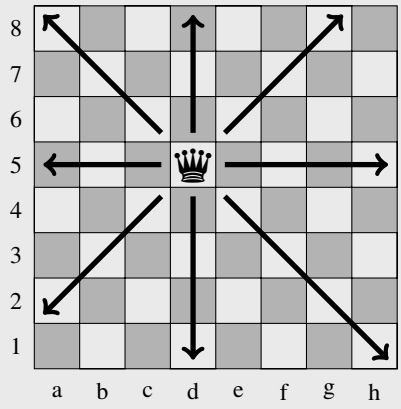


We can also easily determine whether we have a potential full solution by looking at the number of parentheses in our current sequence. Since a valid solution involves n pairs of parentheses, the length of a valid sequence must have length $2n$. Using this information, we can construct the following backtracking solution for this problem:

```
1  std::vector<std::string> generate_parentheses(int32_t n) {
2    std::vector<std::string> solutions;
3    std::string current_partial_solution;
4    check_parentheses(n, solutions, current_partial_solution, 0, 0);
5    return solutions;
6  } // generate_parentheses()
7
8  void check_parentheses(int32_t n, std::vector<std::string>& solutions,
9                         std::string current_partial_solution, int32_t left, int32_t right) {
10   if (current_partial_solution.length() == 2 * n) {
11     solutions.push_back(current_partial_solution);
12   } // if
13   else {
14     if (left < n) {
15       check_parentheses(n, solutions, current_partial_solution + "(", left + 1, right);
16     } // if
17     if (right < left) {
18       check_pare ntheses(n, solutions, current_partial_solution + ")", left, right + 1);
19     } // if
20   } // else
21 } // check_parentheses()
```

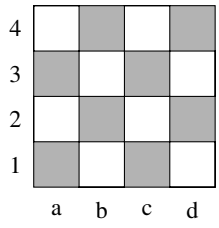
※ 22.1.4 *Solving Problems Using Backtracking: N-Queens*

Example 22.4 In the game of chess, the *queen* is a powerful piece that can move any number of squares vertically, horizontally, or diagonally. The movement capability of a queen is shown on the chessboard below:

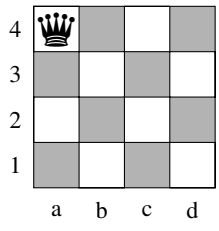


The premise of the N-Queens problem is relatively straightforward: given a chessboard of size $N \times N$, is it possible to place N queens on the chessboard without any of the queens directly threatening each other? In other words, is it possible to place N queens on a $N \times N$ chessboard such that no queen is directly in the path of another queen? Implement a function that takes in a dimension n of a $n \times n$ board and returns all possible placements of n queens such that no queen threatens any other queen on the board.

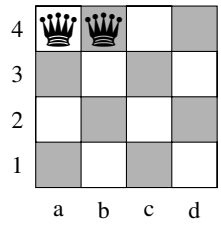
Since this is a constraint satisfaction problem, we can solve it using backtracking. In this problem, we want to place queens on the chessboard under the constraint that any queen we place cannot threaten any other queen already on the chessboard. Let's look at the backtracking algorithm in action on a 4×4 board:

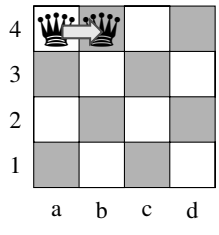


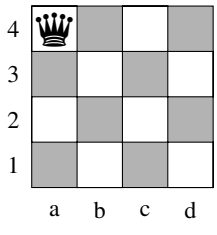
A sensible first choice is to place a queen in the first position and start exploring partial solutions from there. This choice does not break any constraints, since there are no other queens currently on the chessboard.



What should our second choice be? We could consider every empty cell from left to right, then top to bottom. If we did this, we would put a queen in the second cell of the chessboard, see that it is not promising, and undo the choice.

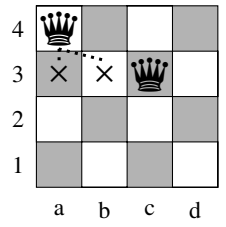




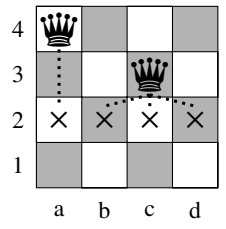


However, this is not an efficient way to explore the search space, since we know that a queen placed in the same row as another queen will always violate our constraint. To reduce our search space and make the backtracking process more efficient, we can instead make choices one *row* at a time. That is, after we place a queen in one row, the next choice will place a queen in the *next* row, which eliminates the need to check the horizontal constraint on a regular basis. Note that queens can be placed one column at a time as well — the idea here is to not waste time making choices that you already know will not be promising without having to make the choice!

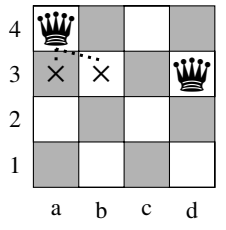
In our example using the 4×4 chessboard, after we place a queen in the top row (row 4 using offical chessboard numbering), we will then attempt to put our next queen in row 3. We consider columns *a* and *b*, but see that they are not promising. The first position in row 3 that does not violate any constraints is column *c*.



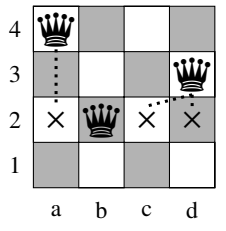
Next, we will attempt to place a third queen in row 2. However, we have a problem: the presence of queens in squares $a4$ and $c3$ makes it impossible to place a queen in row 2 this is not threatened by another queen!



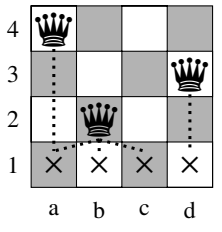
Therefore, our current partial solution cannot lead to a valid solution, so we undo the choice we most recently made (the queen in square $c3$). We then consider the next open position in row 3 that does not violate any constraints, square $d3$.

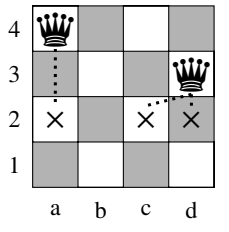


We will now attempt to place a third queen again. With queens in squares $a4$ and $d3$, there is one square in row 2 that is not threatened by any other queens: square $b2$. Thus, we can safely place a queen in this position.

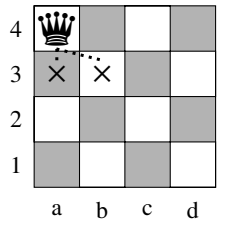


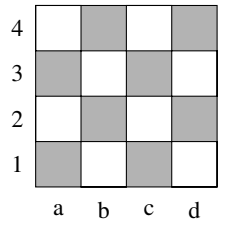
However, the placement of a queen in square $b2$ prevents us from placing a queen in row 1. This partial solution cannot lead to a valid solution, so we undo the choice of placing a queen in square $b2$.



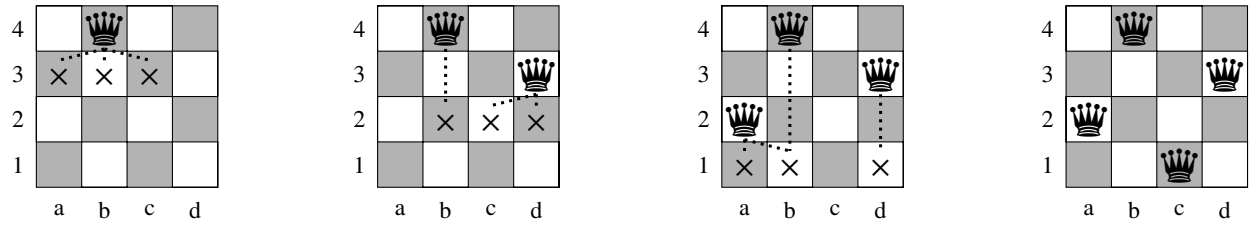


We have considered all the valid placements of queens in row 2, so we backtrack again by undoing the placement of the queen in square $d3$. We also considered all valid placements of queens in row 3, so we backtrack by undoing the placement of our initial queen in square $a4$.





Since placing the queen at position *a*4 did not lead us to a solution, we will try placing the queen at position *b*4, the next position in the first row. The next steps of the backtracking process are shown below:



We were able to place a queen in all four rows, so we have a solution to the problem that satisfies all constraints! To find additional solutions, we would undo the queen placement in square *c*1 and continue searching using the same approach as before.

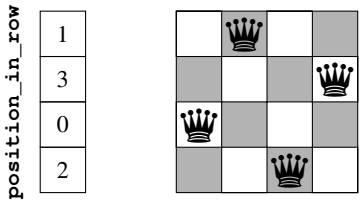
To summarize, the components of the N-Queens problem are as follows:

- The *choice* we need to make is to place a queen in the next unconsidered square (from left to right) in the first row of the chessboard that does not yet have a queen.
- To determine if a partial solution is a *valid complete solution*, we check if we have successfully placed a queen in the final row of the chessboard.
- To determine if a partial solution is *promising*, we check if the queen we place is threatened by any existing queens already on the chessboard.

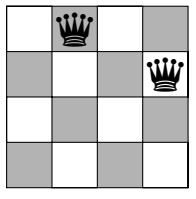
Using this information, we can implement the following solution to the N-Queens problem:

```
1  using Board = std::vector<std::vector<bool>>;
2
3  std::vector<Board> n_queens(size_t n) {
4    std::vector<Board> solutions;
5    Board current_partial_solution(n, std::vector<bool>(n, false));
6    place_queen(solutions, current_partial_solution, 0, n);
7    return solutions;
8  } // n_queens()
9
10 bool promising(Board& current_partial_solution, size_t row, size_t col, size_t n) {
11   // check for queens in same column
12   for (size_t r = 0; r < row; ++r) {
13     if (current_partial_solution[r][col] == true) {
14       return false;
15     } // if
16   } // for r
17   // check for queens in same 45 degree diagonal
18   for (size_t r = row, c = col; r-- > 0 && c-- > 0; ) {
19     if (current_partial_solution[r][c] == true) {
20       return false;
21     } // if
22   } // for r
23   // check for queens in same 135 degree diagonal
24   for (size_t r = row, c = col; r-- > 0 && c++ < n; ) {
25     if (current_partial_solution[r][c] == true) {
26       return false;
27     } // if
28   } // for r
29   return true;
30 } // promising()
31
32 void place_queen(std::vector<Board>& solutions, Board& current_partial_solution,
33                  size_t row, size_t n) {
34   if (row == n) {
35     solutions.push_back(current_partial_solution);
36   } // if
37   else {
38     for (size_t col = 0; col < n; ++col) {
39       if (promising(current_partial_solution, row, col, n)) {
40         current_partial_solution[row][col] = true;
41         place_queen(solutions, current_partial_solution, row + 1, n);
42         current_partial_solution[row][col] = false;
43       } // if
44     } // for col
45   } // else
46 } // place_queen()
```

Instead of representing the board as a 2-D array, it is more efficient to implement the chessboard using several 1-D arrays that store row positions, column availability, and diagonal availabilities. In this implementation, we will first define an array that stores the position of queens that have been placed so far for each row of the chessboard (e.g., as shown below):



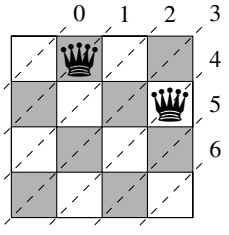
We will also need an array of Booleans that stores the availability of queens in each column. If a queen already exists in a column, the array stores false; otherwise, it stores true.



T	F	T	F

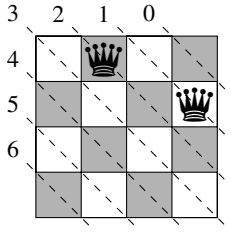
column

The same is done for the left and right diagonals. There are a total of seven left and right diagonals, as shown below. The availability of these diagonals are tracked using two arrays of size seven, starting from top corner diagonal as index zero (in the figures below, the number next to each diagonal represents its index of the array).



T	F	T	T	F	T	T

left_diagonal



T	F	F	T	T	T	T

right_diagonal

To determine which diagonal a square of the board belongs to, we just need to do some simple math: given an $n \times n$ board, the square at row r and column c belongs to left diagonal $r+c$ and right diagonal $(r-c)+(n-1)$. In other words, to check if a queen placed in row r, column c breaks any of the diagonal constraints, we can simply check the value at index $r+c$ of the `left_diagonal` array and the value at index $(r-c)+(n-1)$ of the `right_diagonal` array.

Putting this all together, we can implement the following solution to the N-Queens problem. Unlike before, we can now check if a column or diagonal is available in *constant* time (by checking a Boolean stored in an array) without needing to conduct a linear traversal every time. Our `promising()` function still gets the job done, but much more efficiently!

```
using Board = std::vector<std::vector<bool>>;

class NQueensSolver {
  size_t size;
  std::vector<size_t> position_in_row;
  std::vector<bool> column, left_diagonal, right_diagonal;
  std::vector<Board> solutions;
  const bool AVAILABLE = true;

public:
  NQueensSolver(size_t n)
    : size{n}, position_in_row(n, -1), column(n, true),
      left_diagonal(2 * n - 1, true), right_diagonal(2 * n - 1, true) {}

  void place_queen(size_t row) {
    if (row == size) {
      store_solution_board();
      return;
    } // place_queen()

    for (size_t col = 0; col < size; ++col) {
      if (promising(row, col)) {
        position_in_row[row] = col;
        column[col] = !AVAILABLE;
        left_diagonal[row + col] = !AVAILABLE;
        right_diagonal[row - col + (size - 1)] = !AVAILABLE;
        place_queen(row + 1);

        // undo and backtrack
        column[col] = AVAILABLE;
        left_diagonal[row + col] = AVAILABLE;
        right_diagonal[row - col + (size - 1)] = AVAILABLE;
      } // if
    } // for col
  } // place_queen()

  bool promising(size_t row, size_t col) {
    return column[col] == AVAILABLE &&
           left_diagonal[row + col] = AVAILABLE &&
           right_diagonal[row - col + (size - 1)] = AVAILABLE;
  } // promising()

  void store_solution_board() {
    Board solution(size, std::vector<bool>(size, false));
    for (size_t row; row < size; ++row) {
      for (size_t col; col < size; ++col) {
        if (position_in_row[row] == col) {
          solution[row][col] = true;
        } // if
      } // for col
    } // for row
    solutions.push_back(solution);
  } // store_solution_board()

  void solve() {
    place_queen(0);
  } // solve()

  std::vector<Board> get_solutions() {
    return solutions;
  } // get_solutions()
};

std::vector<Board> n_queens(size_t n) {
  NQueensSolver nqs(n);
  nqs.solve();
  return nqs.get_solutions();
} // n_queens()
```

※ 22.1.5 *Solving Problems Using Backtracking: Sudoku Solver* (✻)

Example 22.5 Sudoku, originating from the Japanese term 数独 meaning "single number", is a combinatorial number puzzle involving a 9×9 grid of digits that is further subdivided into nine 3×3 subgrids. An example of a Sudoku board is shown below:

	2			**6**				**5**
	7	**3**		**9**				
8					**2**			**4**
7	**9**	**5**						
4								**3**
						5	**7**	**9**
2			**3**					**8**
				8		**2**	**6**	
5				**7**			**3**	

The goal of Sudoku is to fill out every cell of this grid with the digits 1 to 9 in a way such that every row, column, and 3×3 subgrid contains all the digits from 1 to 9. For example, the following is a solution to the above Sudoku puzzle:

9	**2**	4	8	**6**	3	7	1	**5**
1	**7**	**3**	4	**9**	5	8	2	6
8	5	6	7	1	**2**	3	9	**4**
7	**9**	**5**	6	3	8	1	4	2
4	1	2	9	5	7	6	8	**3**
6	3	8	1	2	4	**5**	**7**	**9**
2	6	7	**3**	4	1	9	5	**8**
3	4	1	5	**8**	9	**2**	**6**	7
5	8	9	2	**7**	6	4	**3**	1

Given a Sudoku board of characters representing the digits 1-9, implement a function that solves the Sudoku puzzle.

There are several ways we could approach this problem. We could use brute force to generate all possible boards (by considering 1-9 in every open cell) and then validate every single one of those boards, but that would be inefficient. Instead, since we know that the board is governed by a set of constraints, we can use the backtracking approach to reduce the number of possible board configurations we end up checking.

To devise a backtracking solution, we will first identify the choice we need to make to move toward a solution — in this case, the choice is to place a digit from 1 to 9 in an empty cell. Second, we want to devise a method to determine whether a partial solution is a completed solution; this can be done by checking if the entire board is filled. Lastly, we want to devise a method to determine whether a partial solution is promising; this can be done by checking that the digit we want to place does not already exist in the same row, column, or subgrid of the Sudoku board. Using these three components, we can write a straightforward backtracking to the Sudoku problem:

- Traverse the cells of the given Sudoku grid and place a digit in the empty cells one by one (all digits from 1-9 should be considered for each empty cell).
- Check if the grid is still valid by ensuring that the newly placed digit does not already exist in the same row, column, or subgrid. If the grid is still valid after our choice, fix the newly placed digit in place and recurse on the remaining empty cells. Otherwise, prune the branch by removing the digit that was placed.
- Once all the cells have been filled with a valid digit, the algorithm completes and we have a solution.

A simple backtracking solution for the Sudoku solver problem is shown on the next page. The `solve_sudoku()` function takes in a 9×9 grid where empty cells are represented as the character `'.'` and modifies the grid to store the solution to the Sudoku problem. For example, calling `solve_sudoku()` on the grid on the left would turn it into the grid on the right:

```
[ [".","2",".",".","6",".",".",".","5"],
  ["."."7","3",".","9",".",".",".",".","."],
  ["8",".",".",".",".",".","2",".",".","4"],
  ["7","9","5",".",".",".",".",".",".","."],
  ["4",".",".",".",".",".",".",".",".","3"],
  [".",".",".",".",".",".",".","5","7","9"],
  ["2",".",".","3",".",".",".",".",".","8"],
  [".",".",".",".",".","8",".","2","6","."],
  ["5",".",".",".","7",".",".",".","3","."] ]
```

```
[ ["9","2","4","8","6","3","7","1","5"],
  ["1","7","3","4","9","5","8","2","6"],
  ["8","5","6","7","1","2","3","9","4"],
  ["7","9","5","6","3","8","1","4","2"],
  ["4","1","2","9","5","7","6","8","3"],
  ["6","3","8","1","2","4","5","7","9"],
  ["2","6","7","3","4","1","9","5","8"],
  ["3","4","1","5","8","9","2","6","7"],
  ["5","8","9","2","7","6","4","3","1"] ]
```

An implementation of this solution is shown below:

```
void solve_sudoku(std::vector<std::vector<char>> &board) {
  if (!place_digit(board, 0, 0)) {
    std::cout << "No solution" << std::endl;
  } // if
} // solve_sudoku()

bool promising(std::vector<std::vector<char>>& board, size_t row, size_t col, char digit) {
  for (size_t i = 0; i < 9; ++i) {
    // check same row
    if (board[row][i] == digit) {
      return false;
    } // if
    // check same column
    if (board[i][col] == digit) {
      return false;
    } // if
    // check same subgrid
    if (board[row - row % 3 + i / 3][col - col % 3 + i % 3] == digit) {
      return false;
    } // if
  } // for i
  return true;
} // promising()

bool place_digit(std::vector<std::vector<char>>& board, size_t row, size_t col) {
  if (row == 9) {
    return true;
  } // if
  if (col == 9) {
    return place_digit(board, row + 1, 0);  // check first empty cell in next row
  } // if
  if (board[row][col] != '.') {
    return place_digit(board, row, col + 1);  // ignore if cell already filled
  } // if
  for (char c = '1'; c <= '9'; ++c) {
    if (promising(board, row, col, c)) {
      board[row][col] = c;
      if (place_digit(board, row, col + 1)) {
        return true;
      } // if
      board[row][col] = '.';
    } // if
  } // for c
  return false;
} // place_digit()
```

※ 22.1.6 *Backtracking Time Complexity*

What is the time complexity of a backtracking algorithm? Since backtracking is simply an extension of brute force that relies on pruning to reduce the solution space, the worst-case time complexity of backtracking is actually bounded by the time complexity of a brute force approach! Why is this so? Consider the scenario where you are extremely unlucky and are unable to prune anything — if this happens, you are essentially checking every possible solution. As a result, the worst-case time complexity of a backtracking algorithm is based on the total number of solutions you would have to check if nothing is pruned. That being said, even though brute force serves as an upper bound on asymptotic runtime, the actual performance of backtracking is typically much faster (since the worst-case scenario rarely happens).

Example 22.6 You are given a $n \times n$ Sudoku board, where n is a perfect square number. Assuming that the number of possibilities for each cell is n (all numbers from 1 to n), and the number of empty cells at the beginning is m, what is the worst-case time complexity of a backtracking solution for a Sudoku solver of this $n \times n$ board?

There are a total of m open cells, and n possibilities for each cell. Thus, by the fundamental counting principle, there are a total of n^m possible boards that can be created. Thus, the worst-case time complexity of a backtracking solution to the Sudoku solver problem is bounded by $O(n^m)$.

22.2 Branch and Bound

※ 22.2.1 *Branch and Bound: Minimization Problems*

Previously, we discussed the algorithm family of backtracking, which solves constraint satisfaction problems by performing a search over the solution space and pruning partial solutions that cannot lead to a valid solution. This technique of pruning partial solutions can also be used to solve *optimization* problems. If we apply the concepts of backtracking to solve an optimization problem, we end up with a similar algorithm family known as **branch and bound**. In branch and bound, the `promising()` function now has two responsibilities:

1. It must determine if a partial solution can ever lead to a complete solution that *satisfies all constraints* (similar to backtracking).
2. It must also determine if the partial solution can ever lead to an *optimal* solution.

This second task can be completed by estimating a **bound** on the the best solution obtainable from continuing a partial solution. The bound gives you a reasonable estimate for the remaining work needed to complete a solution, without having to do the work itself. This allows the algorithm to quickly identify whether it should continue looking down the branch it is currently exploring: if the bound of a partial solution is not better than the best complete solution known so far, then there is no reason to continue exploring that partial solution (this process of systematically eliminating partial solutions from consideration is known as **pruning**).

It is important to note that bounds are simply *estimates*, and a bound does *not* have to match the actual best solution of extending a partial solution (which can often be expensive to compute). However, you still want your bounds to be as close to the actual solution as possible, since a more accurate bound allows you to prune more efficiently.

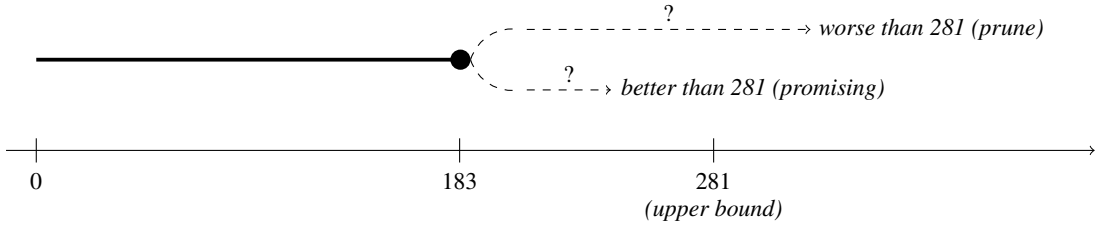
There are also limits on what a valid bound can be: an estimated bound must be *at least as optimistic* as the actual best solution attainable from continuing a partial solution. Otherwise, your algorithm could potentially prune the optimal solution by mistakenly believing that its branch is not promising (an issue known as *overpruning*)!

The precise calculation of bounds depends on whether you are trying to solve a minimization or a maximization problem. If you are working with a *minimization* problem, you would need to identify the following two bounds at every partial solution:

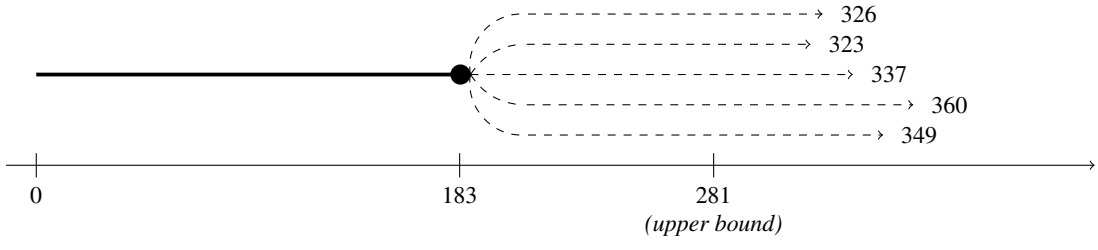
- A **lower bound**, which is an *optimistic estimate* of the best solution you could get from extending the current partial solution. The lower bound is calculated by taking the cost of the partial solution so far and adding it to an *underestimate* of the remaining cost required to complete the partial solution (we will discuss why an underestimate is needed for minimization problems later in the section).
- An **upper bound**, which is the cost of the best complete solution encountered so far. The upper bound, as its name suggests, serves as an upper limit for the lower bound estimate. If the calculated lower bound of a partial solution is greater than or equal to the current upper bound, then the partial solution can be pruned.

The lower and upper bounds are compared to determine if a branch is promising. If the lower bound is less than the current upper bound, then it may be possible to discover a better solution along that branch, and you should continue extending the current partial solution. Otherwise, there is no way to discover a solution that is better by extending the partial solution, and the branch should be pruned.

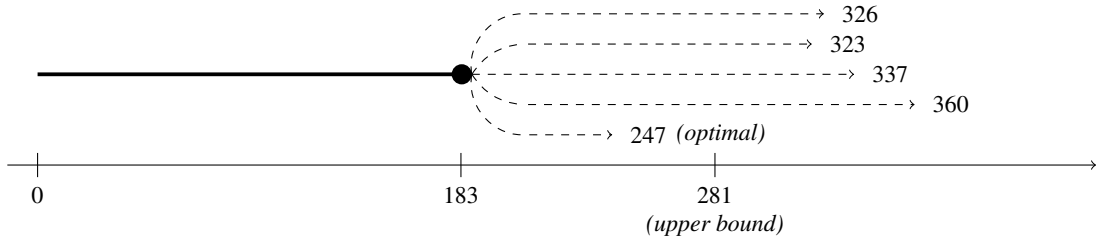
Let's look at this process using an example. Suppose we are trying to solve a minimization problem, where the best complete solution we know so far has a total cost of 281, and the current partial solution we are considering has a cost of 183. Our goal with branch and bound is to discover the optimal solution, so our algorithm needs to know whether the current partial solution can ever lead to a outcome that is better than our best-known solution of 281. If it cannot, then the partial solution cannot be optimal, and we can stop exploring it immediately.



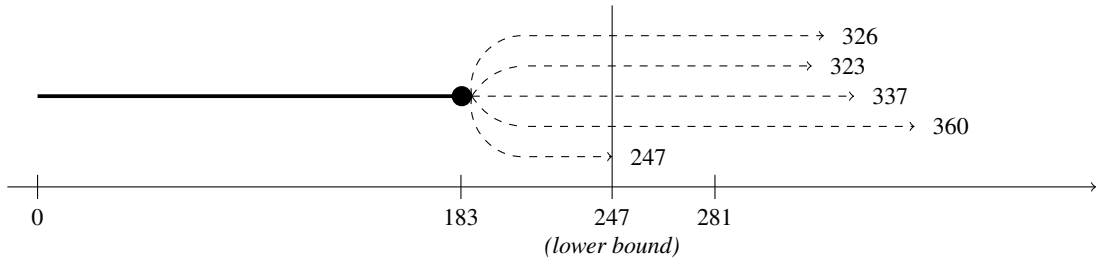
However, there isn't much we can infer about the branch just by looking at the current cost of the partial solution (which is 183). For instance, it is entirely possible that every solution down this branch leads to a solution that is worse than 281 (which indicates the branch should be pruned).



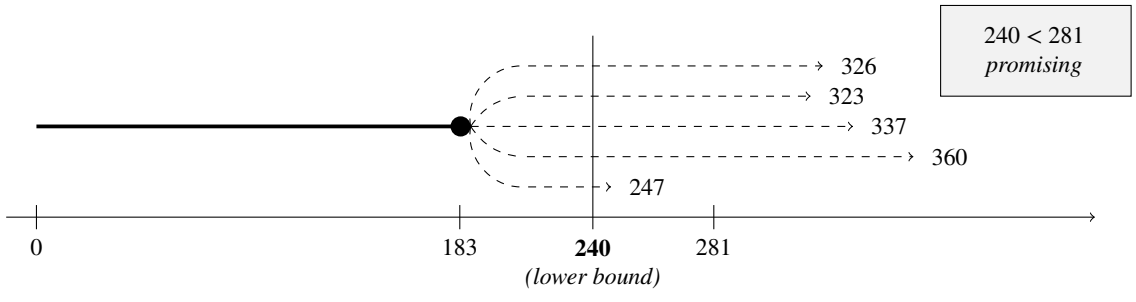
However, it is also possible that this branch eventually leads us down to an optimal solution.



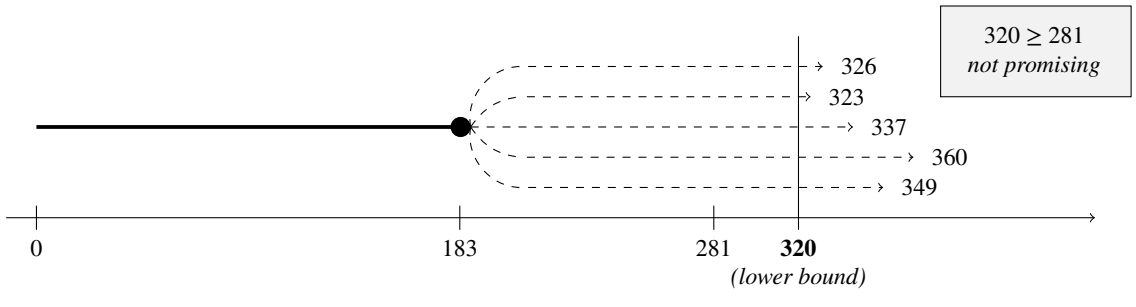
The cost of the current partial solution typically gives us no information on whether the current branch is promising. To determine if we should continue looking down a branch, we must measure the cost of choices that have not yet been included in the partial solution! This is where the lower bound comes in: the lower bound estimates the best possible solution attainable from continuing the current partial solution. In the previous example, the best solution possible from extending the current branch is 247, so 247 would be our ideal lower bound.



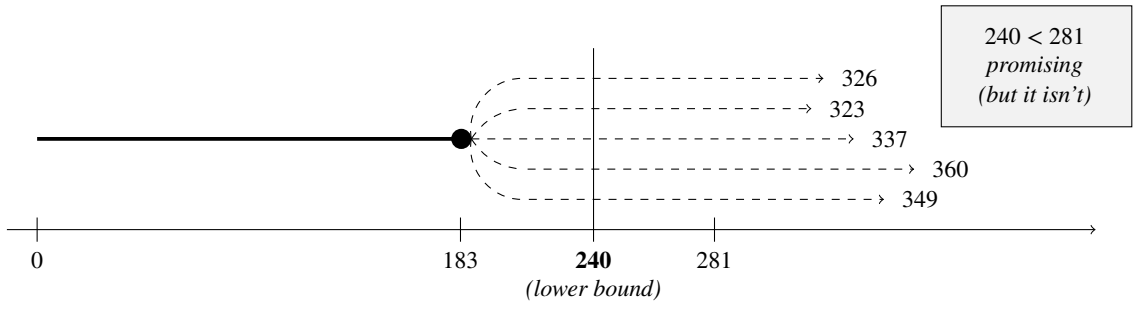
However, just because 247 is the ideal lower bound for this partial solution does not mean your lower bound calculation must be 247. In fact, finding the actual best solution is not always recommended, especially if doing so is computationally expensive. For minimization problems, lower bounds are underestimates, so any lower bound value under 247 would work (although the closer to 247 you are, the better). For instance, 240 would also be a valid lower bound estimation, as shown below. This indicates that it is possible to obtain a solution that is *potentially as good as 240* if the current partial solution is extended. Since the best-known solution is 281, this indicates and that you should continue exploring this branch, as it could lead you to a better solution.



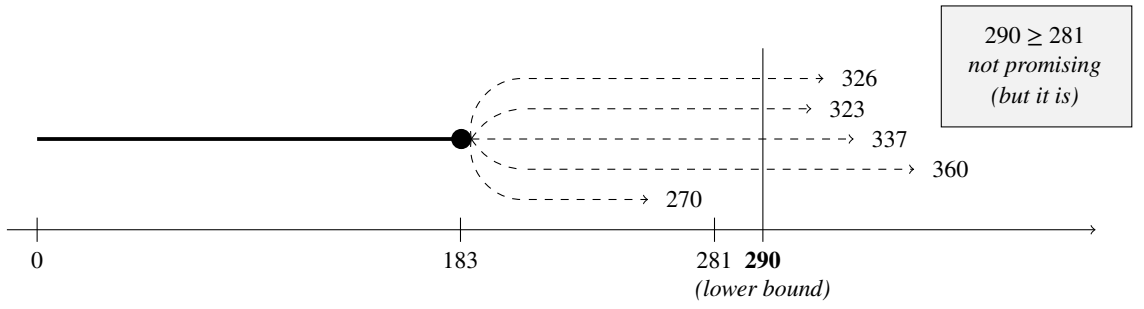
On the other hand, if the lower bound is larger than or equal to the upper bound, then the best solution from continuing a partial solution can never be better than the best solution known so far. In the example below, the lower bound is 320, which indicates that it is impossible to obtain a solution better than 320 by extending the current partial solution. Since 320 is worse than 281, it is safe to prune the branch.



In general, you want the lower bound estimate to be *as close as possible* to the actual best outcome of extending the branch. A lower bound that is too optimistic could cause you to waste time exploring solutions that aren't promising. For example, a lower bound of 240 for the following branch is valid, but you would mistakenly believe that there is a better solution when there actually isn't. This would worsen the performance of your branch and bound algorithm.



However, the lower bound estimate cannot ever be *worse* than the actual best outcome of extending a branch! If your lower bound is too pessimistic, you could mistakenly believe a branch is not promising even if it includes the actual optimal solution. For example, consider the following partial solution, where the best possible outcome of extending the partial solution is 270. However, if your lower bound is too high (e.g., 290), your algorithm would conclude that it is impossible to obtain a solution better than 290 by continuing the branch, and the partial solution would be incorrectly pruned.



To summarize, the steps of a branch and bound algorithm for a minimization problem are as follows:

1. Start with an initial upper bound of infinity.
2. Find the first complete solution and use its cost as an upper bound to prune branches during the rest of the search.
3. Explore the solution space (similar to backtracking). For each partial solution encountered, calculate a lower bound estimate for the total cost required to complete the solution.
4. Prune partial solutions whose lower bound is greater than or equal to the current upper bound. This is because the lower bound is an optimistic estimate of the best solution you could get from continuing the partial solution, and the upper bound is the best-known solution; if the optimistic estimate of continuing a branch is worse than the best-known solution, there is no reason to continue down that branch.
5. If a branch is fully explored without being pruned, and the solution of this completed branch is *better* than the existing upper bound, update the upper bound to this new value (since the upper bound represents the best solution encountered so far).
6. After the entire solution space is explored, the current upper bound is the optimal solution.

The pseudocode for exploring the search space is shown below. At the end of the search, the value of `current_best` (i.e., the upper bound) stores the optimal solution to the minimization problem.

```
1   Algorithm check_node(partial_solution, current_best):
2     if partial_solution is a valid full solution:
3       if partial_solution < current_best:
4         current_best = partial_solution
5     else:
6       if (promising(partial_solution, current_best)):
7         for each choice that can be made to extend partial_solution:
8           partial_solution = partial_solution + choice
9           check_node(partial_solution, current_best)
10          partial_solution = partial_solution - choice
11
12  bool promising(partial_solution, current_best):
13    lower_bound = calculate_lower_bound()
14    if lower_bound < current_best:
15      return true
16    return false
```

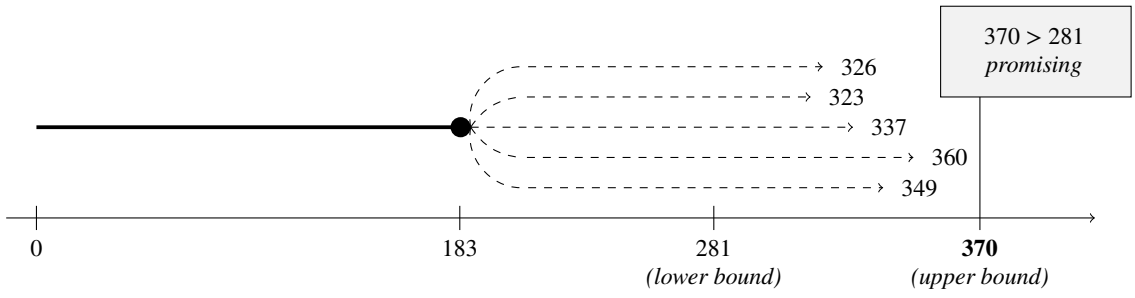
※ 22.2.2 *Branch and Bound: Maximization Problems*

Maximization problems also work in a similar fashion, just with the bounds reversed: the upper bound now becomes the estimate for the best solution attainable from extending a partial solution, and the lower bound now becomes the best complete solution encountered so far. The two bounds for maximization problems are summarized below:

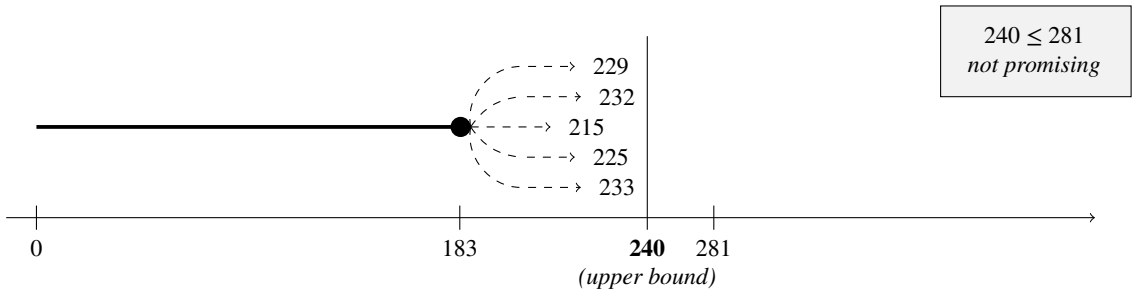
- The **upper bound** of a maximization problem is an *optimistic estimate* for the best solution you could get from extending the current partial solution. The upper bound is calculated by taking the value of the partial solution and adding it to an *overestimate* of the remaining value obtainable from completing the partial solution.
- The **lower bound** of a maximization problem is the value of the best complete solution encountered so far. The lower bound serves as the lower limit for the upper bound estimation. If the calculated upper bound of a partial solution is not greater than the current lower bound, then the partial solution can be pruned.

If the upper bound is greater than the current lower bound, then it is possible to discover a solution that is better than the current best solution, and you should continue extending the current partial solution. Otherwise, there is no way to discover a solution that is better from extending the partial solution, and the branch should be pruned.

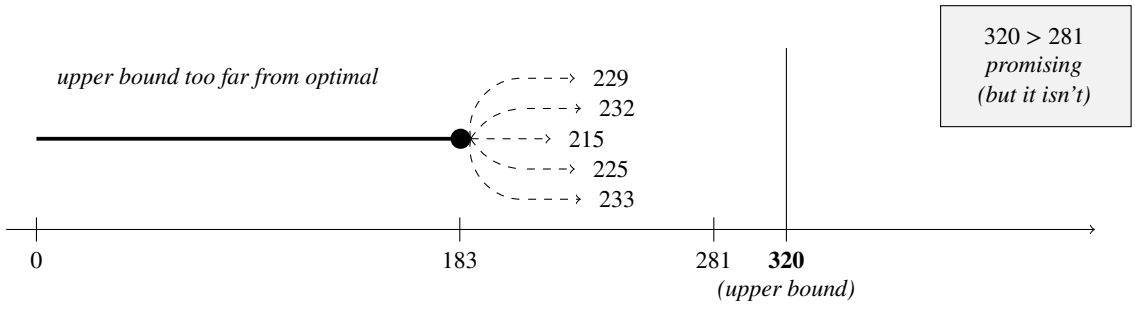
Consider the following example, where the current partial solution has a value of 183 and the best known complete solution has a value of 281. Since this is a maximization problem, you want to find an upper bound on the best solution you could get from extending the current branch. The upper bound must be an overestimate, so anything above 360 would work (since 360 is the best solution obtainable from continuing the partial solution in this example). In this case, let's assume our upper bound is 370. Since this upper bound is better than the lower bound of 281, it is possible to obtain a better solution from extending the branch, so the partial solution is promising.

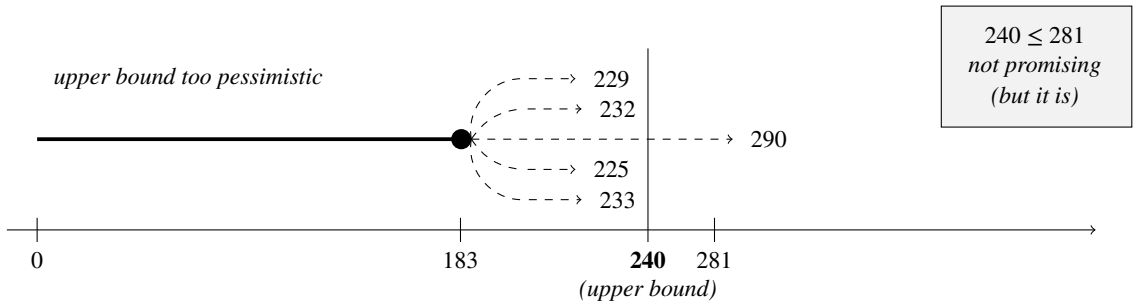


On the other hand, if the estimated upper bound were less than or equal to the lower bound, then there would be no way to obtain a better solution from extending the current branch. In such a case, the branch should be pruned. In the example below, the upper bound is 240, which indicates that 240 is the best we can do from continuing this partial solition. However, since 281 is the best-known solution for this maximization problem, this branch is not promising and is thereby pruned from our search.



Similar to before, the upper bound you estimate for maximization problems should be as close as possible to the actual best solution from extending a partial solution (to allow you to prune efficiently). However, it must also be *more optimistic*, or you could end up pruning the optimal solution. These two error cases are shown below — the first (underpruning) would make the algorithm inefficient, while the second (overpruning) could cause you to prune the optimal solution and end up with the wrong answer:





To summarize, the steps of a branch and bound algorithm for a maximization problem are as follows:

1. Start with an initial lower bound of zero (or negative infinity, if negatives are allowed).
2. Find the first complete solution and use its cost as a lower bound to prune branches during the rest of the search.
3. Explore the solution space. For each partial solution encountered, calculate an upper bound estimate for the total value attainable from completing the solution.
4. Prune partial solutions whose upper bound is less than or equal to the current lower bound. This is because the upper bound is an optimistic estimate of the best solution you could get from continuing the partial solution, and the lower bound is the best-known solution; if the optimistic estimate of continuing a branch is worse than the best-known solution, there is no reason to continue down that branch.
5. If a branch is fully explored without being pruned, and the solution of this completed branch is *better* than the existing lower bound, update the lower bound to this new value (since the lower bound represents the best solution encountered so far).
6. After the entire solution space is explored, the current lower bound is the optimal solution.

The pseudocode for this approach is shown below. At the end of the search, the value of `current_best` (i.e., the lower bound) stores the optimal solution to the maximization problem:

```
Algorithm check_node(partial_solution, current_best):
  if partial_solution is a valid full solution:
    if partial_solution > current_best:
      current_best = partial_solution
  else:
    if (promising(partial_solution, current_best)):
      for each choice that can be made to extend partial_solution:
        partial_solution = partial_solution + choice
        check_node(partial_solution, current_best)
        partial_solution = partial_solution - choice

bool promising(partial_solution, current_best):
  upper_bound = calculate_upper_bound()
  if upper_bound > current_best:
    return true
  return false
```

Branch and bound algorithms are not always easy to implement, even if they follow a common pattern! The most challenging component of any branch and bound algorithm is the process of estimating the bounds, since this step can make or break the algorithm. There are also performance tradeoffs you must consider when identifying your bounds — better bounds can save you time by allowing you to prune more efficiently, but they can also require a non-trivial amount of time to compute! In some cases, a quick bound estimation can get the job done. In other cases, spending more time perfecting your bounds can be worth the runtime improvements you get from pruning. For instance, an $\Theta(n^2)$ algorithm to estimate bounds may seem inefficient, but it is often preferable if the alternative involves exploring $\Theta(n!)$ additional branches.

Similar to backtracking, the worst-case time complexity of a branch and bound algorithm is bounded by the complexity of a brute force approach, which occurs in the case where nothing gets pruned. However, like before, this worst-case scenario rarely ever happens, so the performance of branch and bound is typically superior to that of brute force.

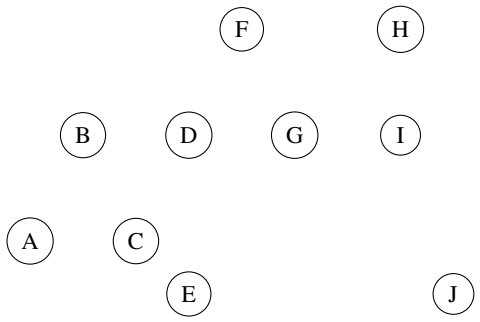
In this class, we will focus on two important problems that can be solved using branch and bound. First, we will look at the *traveling salesperson problem*, a minimization problem that can be solved using a branch and bound approach. Then, in chapter 24, we will look at the *knapsack problem*, a maximization problem that can also be solved using this algorithm family.

22.3 The Traveling Salesperson Problem (TSP)

※ 22.3.1 *Estimating Bounds for TSP*

In this section, we will discuss the **traveling salesperson problem (TSP)**, one of the most extensively studied optimization problems in computer science. In the traveling salesperson problem, you are given a collection of cities and distances between each pair of cities, and you want to find the shortest route that visits every city once and returns back to the starting city (such a cycle that traverses every node of a graph once is known as a **Hamiltonian cycle**).

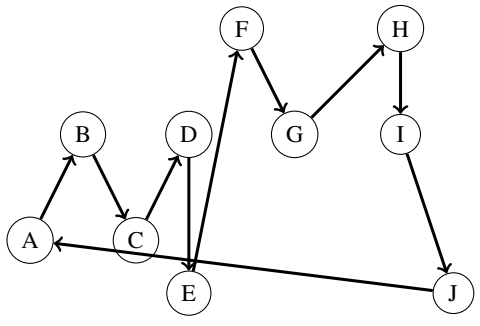
Suppose you are given the following ten cities, and you want to find the shortest Hamiltonian cycle that visits every city exactly once:



City	Coordinate
A	$(2,3)$
B	$(3,5)$
C	$(4,3)$
D	$(5,5)$
E	$(5,2)$
F	$(6,7)$
G	$(7,5)$
H	$(9,7)$
I	$(9,5)$
J	$(10,2)$

If you were to use brute force to find a solution, you would have to check every Hamiltonian cycle possible to identify an optimal path. There are a total of $\frac{(n-1)!}{2}$ such cycles, since from any starting vertex there are $n-1$ edges to choose from for the first vertex, $n-2$ edges to choose from for the second vertex, $n-3$ edges to choose from for the third vertex, and so on (we divide the final result by two since half of these permutations are simply mirror images of each other, i.e., $A \to B \to C \to A$ and $A \to C \to B \to A$). Because there exist $O(n!)$ possible Hamiltonian cycles given n cities, a brute force approach to TSP would also take $O(n!)$ time, which is computationally infeasible for large input sizes.

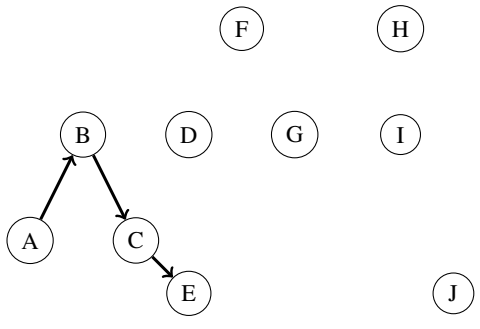
To reduce the runtime of TSP, we can use a branch and bound approach to prune off paths we know cannot be optimal, therefore reducing the size of our search space. To start off the branch and bound process, we will find the first complete solution and use its cost as an upper bound to prune branches during the rest of the search.[1] For our example, suppose the first complete solution we find is $A \to B \to C \to D \to E \to F \to G \to H \to I \to J \to A$, which has a total cost of 33.095:[2]



Connection	Distance
$A \to B$	2.236
$B \to C$	2.236
$C \to D$	2.236
$D \to E$	3.000
$E \to F$	5.099
$F \to G$	2.236
$G \to H$	2.828
$H \to I$	2.000
$I \to J$	3.162
$J \to A$	8.062
Total	**33.095**

Now we know that, whatever the optimal path is, it must have a weight less than or equal to 33.095. We will therefore set 33.095 as our initial upper bound. This allows us to prune future partial solutions that we know cannot lead us to a solution that is better than 33.095, the best solution we have encountered so far.

After identifying this upper bound, we will now need to implement a method to determine whether a partial solution is promising. Recall from the previous section that this step is done by estimating a *lower bound* for each partial solution, which serves as an optimistic estimate for the best solution attainable from continuing the partial solution. For example, consider the following partial solution $A \to B \to C \to E$:

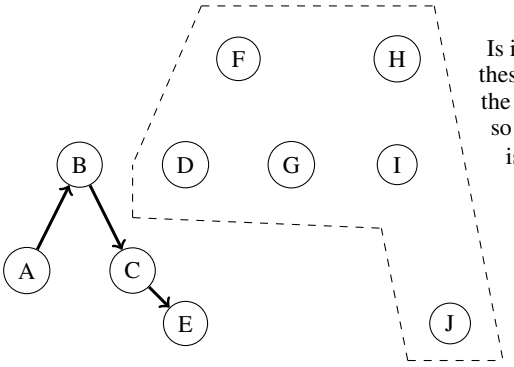


Connection	Distance
$A \to B$	2.236
$B \to C$	2.236
$C \to E$	1.414
–	–
–	–
–	–
–	–
–	–
–	–
–	–
Total	**5.886**

[1] This is actually not the best way to set an initial upper bound, since the first solution you find may not be an ideal bound. Instead, you should set the initial bound to the result of a TSP heuristic, which we will discuss in a later section.

[2] The direction of the arrows here are arbitrary chosen, and can be flipped around to produce the exact same path. The arrows are included here (and in future examples) to ease understanding of certain TSP concepts, so don't worry too much about why the direction of the path is the way it is. Two solutions with the same edges are considered to be identical, even if the direction of the arrows are flipped.

To determine whether or not we should extend this branch, we will need to estimate a lower bound on the the best solution we could get from extending this partial solution. If the lower bound is better than 33.095, we continue looking; otherwise, we should prune the branch. However, this requires us to estimate the cost of connecting points currently not in our path!



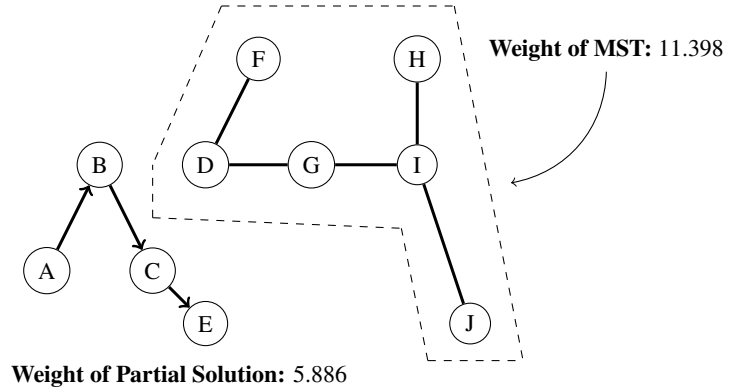
Is it possible to connect these remaining points to the current path in a way so that the total weight is less than 33.095?

Connection	Distance
$A \rightarrow B$	2.236
$B \rightarrow C$	2.236
$C \rightarrow F$	1.414
–	–
–	–
–	–
–	–
–	–
–	–
–	–
Total	**5.886**

This step is rather tricky — we obviously cannot solve for the actual best solution from extending the branch, since that is the problem we are trying to answer in the first place! Instead, we have to rely on an estimate of what the best possible solution could be. As noted previously, this estimate should be as close to the actual best solution as possible, but it can never be worse (which could cause you to prune off the optimal solution). How can we devise an efficient mechanism that always gives us an *underestimate* that is *as close as possible* to the actual best outcome of extending the current branch?

Let's look at the previous figure again. We know the total cost of our current partial solution $A \rightarrow B \rightarrow C \rightarrow E$ is 5.886. However, we do not know the cost of connecting the remaining points that are currently not in our path. If we could quickly solve for this, we would be able to produce a good lower bound estimation for our branch and bound algorithm! What is the best way to estimate the lowest possible cost of connecting these external points?

It turns out we can use a *minimum spanning tree* to estimate this cost. Since a minimum spanning tree gives us the lowest possible weight of connecting a set of points, an MST estimate is *guaranteed to be no worse* than the actual best cost of connecting the remaining points to our path. In our example, the MST of the remaining six points has a total weight of 11.398:



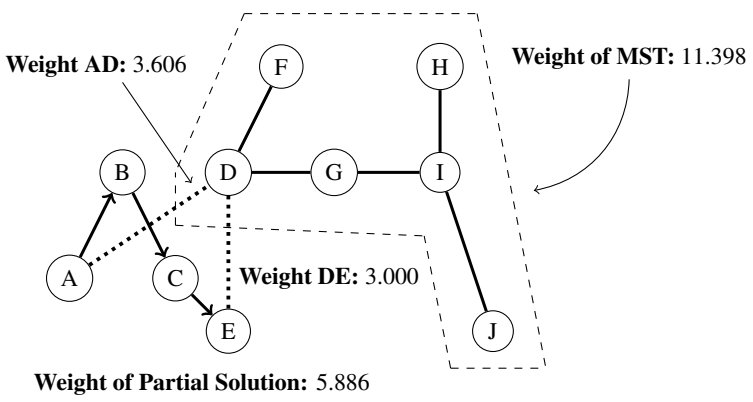
Weight of MST: 11.398

Weight of Partial Solution: 5.886

Connection	Distance
$A \rightarrow B$	2.236
$B \rightarrow C$	2.236
$C \rightarrow E$	1.414
$D \rightarrow F$	2.236
$D \rightarrow G$	2.000
$G \rightarrow I$	2.000
$H \rightarrow I$	2.000
$I \rightarrow J$	3.162
–	–
–	–
Total	**17.284**

Since our current partial solution has a weight of 5.886, and the MST connecting the remaining points has a weight of 11.398, the best solution attainable from continuing the partial solution cannot be better than $5.886 + 11.398 = 17.284$. This will always be an underestimate of the actual best solution, so we can use it as a valid lower bound for our branch and bound algorithm.

However, we can do even better — even though a lower bound of 17.284 works, it is not the most efficient for pruning. We can get a closer estimate of the ideal best solution by also connecting the endpoints of our current partial solution with the MST of the remaining points. To ensure that our lower bound remains optimistic, we will connect each endpoint of our current partial solution with the *nearest* vertex that is not in our path. In our example, the endpoints of our current partial solution are vertices A and E. Of the remaining vertices not in our path, vertex A is closest to vertex D, and vertex E is closest to vertex D. Thus, we would connect both A and E with D, as shown. (Since this is used to produce a lower bound estimate and not a valid path, it is okay to connect the two endpoints to the same point in the MST.)



Weight of Partial Solution: 5.886

Connection	Distance
$A \to B$	2.236
$B \to C$	2.236
$C \to F$	3.000
$D \to E$	2.236
$E \to G$	2.828
$G \to I$	1.000
$H \to I$	2.000
$I \to J$	3.162
$A \to D$	3.606
$E \to D$	3.000
Total	**23.890**

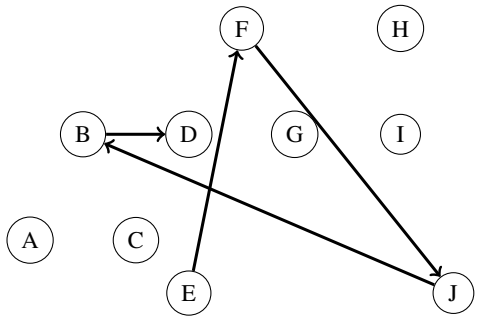
If we sum these four components together, we get a total weight of 23.89, which we can use as our lower bound. This indicates that we cannot get a solution better than 23.89 if we were to continue exploring the partial solution $A \to B \to C \to E$. We can therefore compare this value with our current best solution to determine whether it is worthwhile to continue extending the partial solution. If 23.89 is better than the best-known solution, we would continue looking; otherwise, we would prune the branch and consider another path.

In summary, to obtain a lower bound for the TSP problem, you can sum of the following four weights:

1. The cost of the current partial solution.
2. The cost of the MST connecting the remaining points not in the partial solution.
3. The cost of connecting one end of the partial solution with its closest point in the MST.
4. The cost of connecting the other end of the partial solution with its closest point in the MST.

This will always produce an optimistic estimate on the remaining cost, so it can be used as a valid lower bound. Notice that the estimate will likely not be a valid solution, since an MST is not guaranteed to produce a Hamiltonian cycle on the remaining points! However, this is okay, since this estimate is only used to determine whether a partial solution *could* lead to an optimal solution, and not what the actual optimal solution is. This method of estimating a lower bound is useful because it not only finds a lower bound close to the ideal, but it also can be done rather efficiently: the cost of finding an MST is significantly cheaper than exploring all remaining permutations of a partial solution!

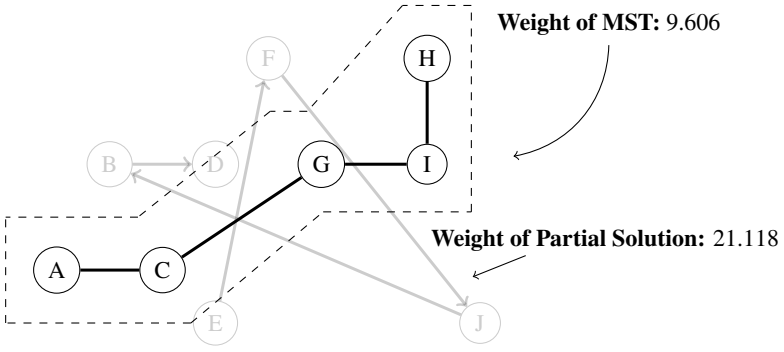
Here is another example of this lower bound method in action, but with a partial solution that is not promising. Suppose we are currently exploring the partial solution $E \to F \to J \to B \to D$, as shown:



Weight of Partial Solution: 21.118

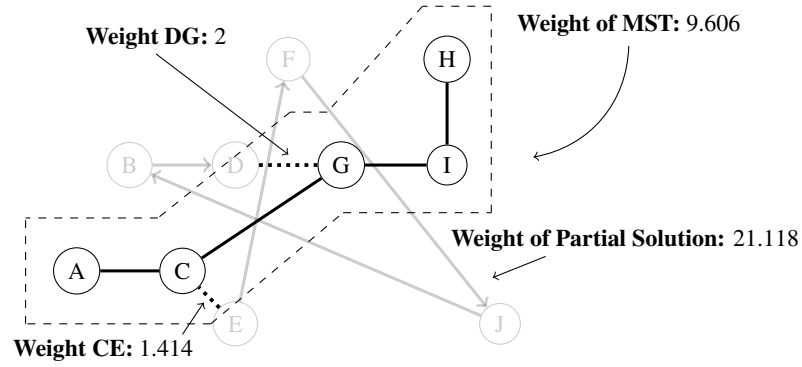
Connection	Distance
$E \to F$	5.099
$F \to J$	6.403
$J \to B$	7.616
$B \to D$	2.000
–	–
–	–
–	–
–	–
–	–
–	–
Total	**21.118**

To estimate a lower bound for this partial solution, we first find the weight of the MST that connects the remaining points, which is 9.606:



Connection	Distance
$E \to F$	5.099
$F \to J$	6.403
$J \to B$	7.616
$B \to D$	2.000
$A \to C$	2.000
$C \to G$	3.606
$G \to I$	2.000
$H \to I$	2.000
–	–
–	–
Total	**30.724**

Then, we calculate the cost of connecting the endpoints of our current partial solution (E and D) with their closest points in the MST. The closest point to E is C (for a cost of 1.414), and the closest point to D is G (for a cost of 2).



Connection	Distance
$E \to F$	5.099
$F \to J$	6.403
$J \to B$	7.616
$B \to D$	2.000
$A \to C$	2.000
$C \to G$	3.606
$G \to I$	2.000
$H \to I$	2.000
$E \to C$	1.414
$D \to G$	2.000
Total	**34.138**

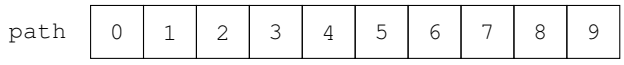
Adding everything up, we get a lower bound of $21.118 + 9.606 + 2 + 1.414 = 34.138$. This means that the best solution we could get from continuing the partial solution $E \to F \to J \to B \to D$ cannot be better than 34.138. Since we already know that the cost of the complete solution $A \to B \to C \to D \to E \to F \to G \to H \to I \to J \to A$ has a cost of 33.095, we know that our current partial solution is not promising. Thus, we know that an optimal solution cannot possibly include $E \to F \to J \to B \to D$, allowing us to prune this branch from our search space.

※ 22.3.2 *Generating Permutations*

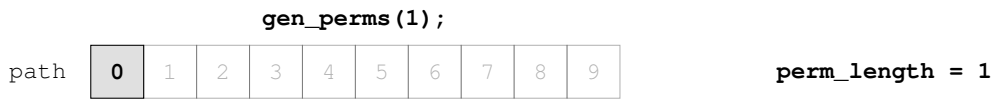
We now have a mechanism for determining whether a partial solution is promising. However, we still need a way to generate these partial solutions when exploring the solution space. In this class, we will use the following `gen_perms()` function to generate all partial solutions by permuting over all possible paths where the first `perm_length` items of a path vector are fixed.

```
1  template <typename T>
2  void gen_perms(std::vector<T>& path, size_t perm_length) {
3    if (perm_length == path.size()) {
4      // Base case, this gets hit if you fully explore a path without ever pruning it.
5      // This means the path currently stored in the vector may be an optimal solution,
6      // so compare it with the best solution known so far and update the upper bound if needed.
7      return;
8    } // if
9    if (!promising(path, perm_length)) {
10     return;
11   } // if
12   for (size_t i = perm_length; i < path.size(); ++i) {
13     std::swap(path[perm_length], path[i]);
14     gen_perms(path, perm_length + 1);
15     std::swap(path[perm_length], path[i]);
16   } // for i
17 } // gen_perms()
```

Let's take a closer look at how `gen_perms()` works. Consider the traveling salesperson problem for 10 vertices, numbered from 0 to 9. To use `gen_perms()`, we will store all of these vertices in a vector that stores the current path:

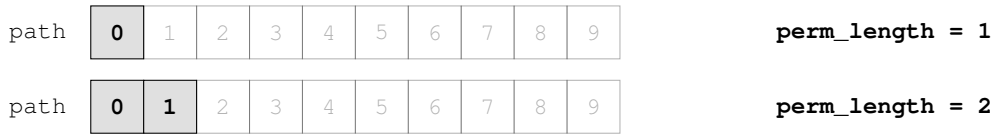


`gen_perms()` uses this `path` vector to identify the current partial solution at *any* point along our search. This is done using the `perm_length` variable, which represents the current length of our partial solution. To start the search, `gen_perms()` is called with a `perm_length` of 1:



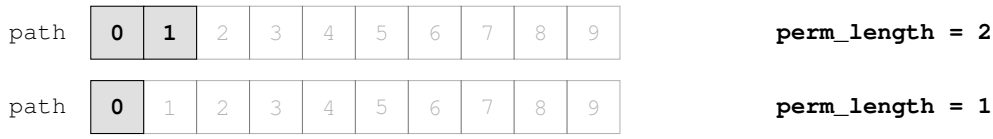
This indicates that the current partial solution has a length of 1 and only includes the vertex at the first position of the vector. We then check if this partial solution is promising using the lower bound strategy explained previously — if it is, we extend the partial solution by adding another vertex to the back of our path. This is done by moving the vertex we want to add to the back of the partial solution, and then recursing on the vector with a `perm_length` value that is larger by one. This process is implemented using the loop on line 10 of the provided code.

Let's look at this loop in action using the `path` vector above. In the first iteration of the loop, `i` is equal to 1, so index 1 is swapped with itself. We then make a recursive call with a `perm_length` of 2, which adds vertex 1 to our partial solution.

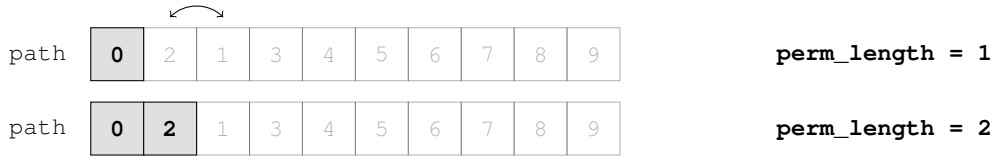


We then check if the partial solution $0 \rightarrow 1$ is promising. If it is, we make a recursive call with a `perm_length` of 3, which adds vertex 2 to the partial solution.

However, let's suppose that our `promising()` function determines that the partial solution $0 \rightarrow 1$ isn't promising. In this case, we would be able to prune off this branch. The pruning step happens when the function returns on line 8, and we return to the stack frame where `perm_length` is 1. Notice that this tells our algorithm to ignore any permutation that starts with $0 \rightarrow 1$, since we will no longer make any additional recursive calls that begin with these two items!



After the recursive call unrolls, we undo the swap we completed on line 11 (to ensure our loop explores the remaining branches correctly) and place the next element we want to consider at the end of the partial solution. A new element is considered with each iteration of the loop — here, `i` is incremented to 2 on the next iteration, so we swap the vertex at index 2 to the back of the partial solution and recurse with a `perm_length` of 2. This allows us to explore all branches that start with $0 \rightarrow 2$.



This process continues for the remainder of the search (we explore all paths that start with $0 \rightarrow 3$, then $0 \rightarrow 4$, and so on). At every step, we add a vertex to our path by swapping it into the next position of our partial solution and fixing it in place, and then recursively generate permutations on the remaining vertices not in our path. We then check if each partial solution is promising by comparing its lower bound with the best complete solution known so far (i.e., the upper bound). If it is promising, we continue extending the branch. Otherwise, we swap out the element we most recently added to our partial solution and try something else in its place.

Eventually, if every partial solution along a branch is promising, you will end up in a position where `perm_length` is equal to the size of the `path` vector. When this happens, the entire path is fixed, and you have a complete solution that could potentially be better than the best known solution so far. In this case, you should check to see if your current path is indeed better, and update your current best solution if it is.

There are two important details to note here. First, a path is always promising does *not* guarantee an optimal solution. This is because we use an *estimate* to determine if a branch is promising, and it is entirely possible for that estimate to be too optimstic. Because of this, you cannot blindly update your best solution if you reach the base case — instead, you will need to compare the current path with the best solution you've encountered to determine if it is actually better. Second, you cannot ensure that a solution is optimal until the initial `gen_perms()` call runs to completion. This is because you need to consider all possible solutions before you can safely conclude that one is better than the rest!

Remark: The `promising()` function estimates the cost of continuing a partial solution, which allows you to prune branches that cannot lead to an optimal solution. However, there comes a point where the work required to estimate a lower bound is actually *greater* than the work required to explore all the remaining branches. Suppose there are k vertices that remain unvisited in our current partial solution, where $k =$ `path.size() - perm_length`. The cost of checking all remaining branches along this partial solution is $k!$, since we have to permute over the k vertices that have not yet been added to the path. On the other hand, the cost of estimating a lower bound is $k^2 + 2k$, where the k^2 work comes from building the MST, and the $2k$ work comes from connecting the two endpoints of the current tour with their closest points in the MST. If we look at the total work required for different values for k, we would get the following:

k	TSP ($k!$)	Estimate ($k^2 + 2k$)
1	1	3
2	2	8
3	6	15
4	**24**	**24**
5	120	35
6	720	48
7	5040	63
8	40320	80
9	362880	99
10	3628800	120

It's pretty clear that you would prefer to estimate if k is large. However, if there are fewer than five points remaining, then you aren't getting much benefit from estimating a lower bound over simply brute forcing the remaining solutions. Thus, you can speed up your branch-and-bound algorithm by having `promising()` always return `true` if the number of remaining points is less than $k = 5$.

```
1  bool promising(std::vector<T>& path, size_t perm_length) {
2    if (path.size() - perm_length < 5) {
3      return true;
4    } // if
5    ... // estimate lower bound and compare with upper bound
6  } // promising()
```

22.4 TSP Heuristics

In the previous section, we devised a reasonably efficient branch and bound algorithm for solving the traveling salesperson problem. However, if we were to run our branch and bound algorithm on an input file containing several hundred points, we would see that it would take an incredibly long time to run. Even though branch and bound is an improvement over brute force, there are still a lot of branches that we have to explore, and the cost of estimating a lower bound is not negligible, especially as the number of vertices grows.

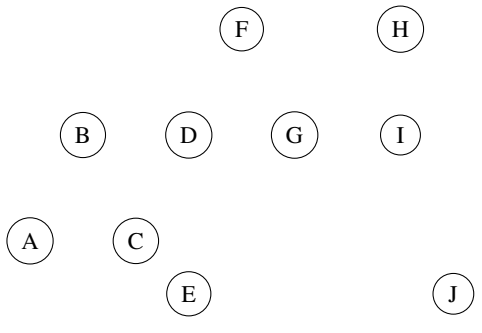
At the end of the day, TSP is still a computationally expensive problem, and an efficient branch and bound implementation can still struggle on large input sizes. Because of this, we often rely on **heuristics** to approximate solutions for TSP problems where the input size is prohibitively large. Unlike brute force or branch and bound, heuristics do *not* guarantee an optimal solution, but they have better time complexities and often produce approximations that are close to optimal. Heuristics can be classified into separate categories based on their behavior: *constructive heuristics* approximate an optimal solution by extending a partial solution step-by-step using a predefined set of rules, while *local search heuristics* start with a complete solution and attempt to improve it by applying a series of small changes. In this section, we will discuss a few heuristics that can be used to efficiently approximate the optional solution of a traveling salesperson problem.

※ 22.4.1 *Nearest Neighbor and 2-Opt*

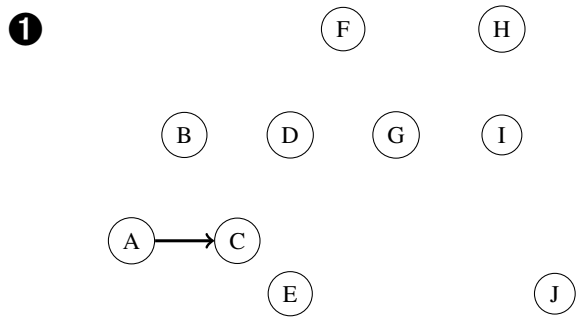
Nearest neighbor and 2-opt are actually two different heuristics that can be combined to produce a close-to-optimal TSP solution in polynomial time. **Nearest neighbor** is a relatively straightforward heuristic that relies on the greedy approach to identify an optimal path. The steps of the nearest neighbor heuristic are as follows:

1. Start with an initial partial solution containing a single arbitrarily selected city.
2. Identify the unvisited neighbor closest to the vertex most recently added to the partial solution, and add it to the path.
3. After all points are added to the partial solution, connect the first and last vertices in the path to complete the solution.

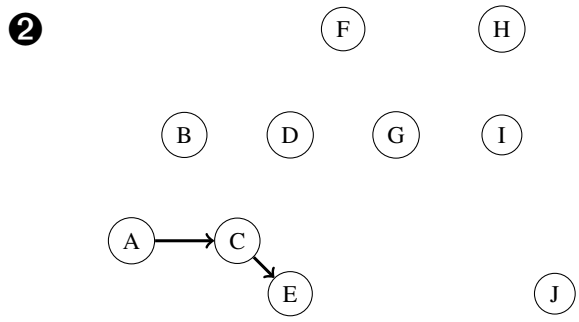
For example, let's run the nearest neighbor heuristic on the following points, with A as our initial starting vertex.



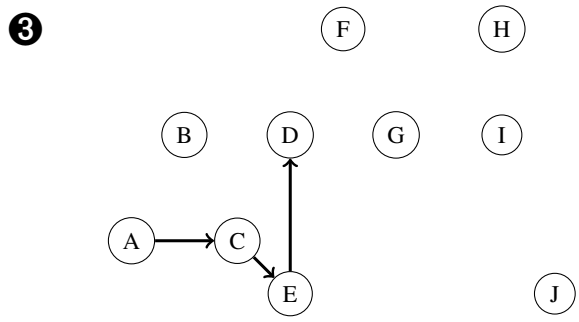
The closest unvisited vertex to A is vertex C, so we add C to our partial solution next:



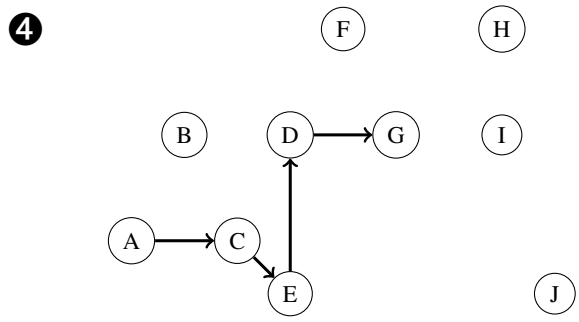
The closest unvisited vertex to C is E, so we add E to our partial solution next:



The closest unvisited vertex to E is D, so we add D to our partial solution next:

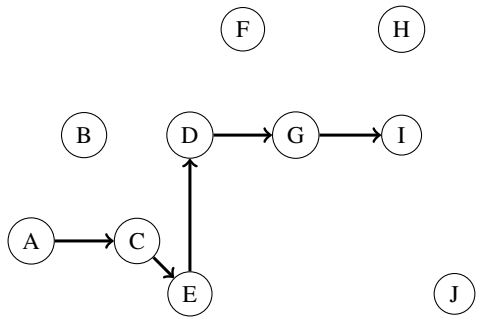


The closest unvisited vertex to D is G, so we add G to our partial solution next:



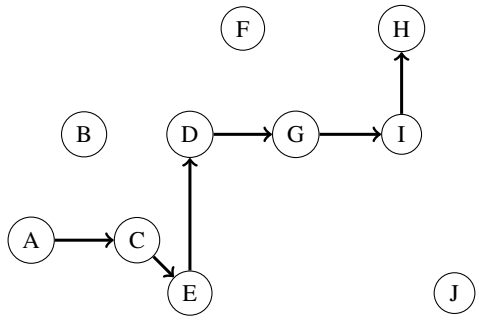
Continuing this process, we would get the following (where ties are broken arbitrarily):

❺



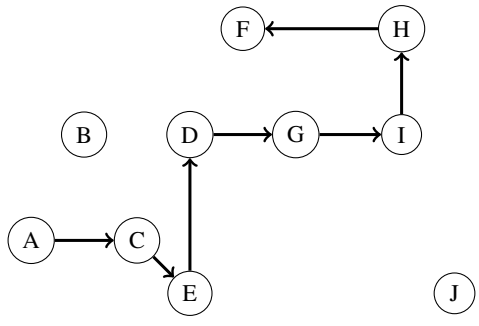
---

❻



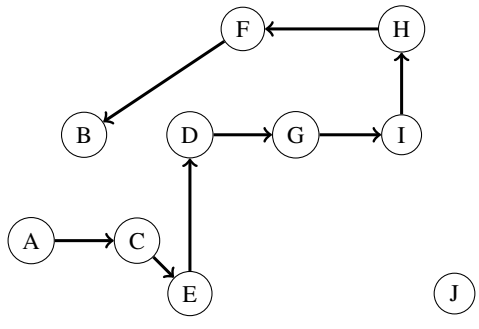
---

❼



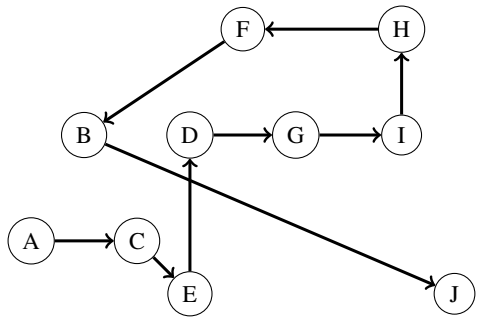
---

❽

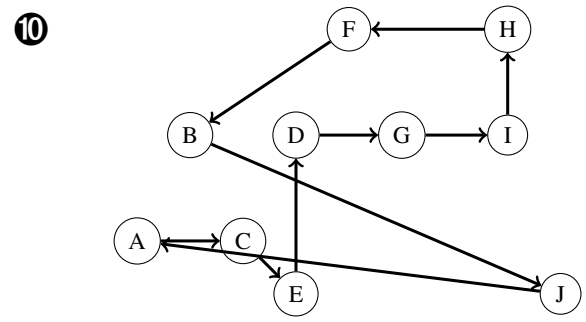


---

❾



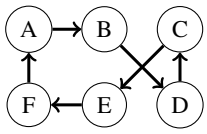
At the end, we connect the final point with our initial vertex. This is the result we get from the nearest neighbor heuristic:



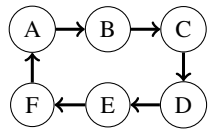
Connection	Distance
$A \to C$	2.000
$C \to E$	1.414
$E \to D$	3.000
$D \to G$	2.000
$G \to I$	2.000
$I \to H$	2.000
$H \to F$	3.000
$F \to B$	3.606
$B \to J$	7.616
$J \to A$	8.062
Total	**34.698**

The nearest neighbor heuristic can be implemented very efficiently for TSP, with a worst-case time complexity of $\Theta(n^2)$ for n points. However, as you can see, it does not always produce the best result. To improve this greedy solution, we can use a strategy known as **2-opt**, which can be used to improve an existing, non-optimal tour. It turns out that, if you are given any two *non-adjacent* edges in a tour, you can safely connect the first endpoints of the edges together and the second endpoints of the edges together without disconnecting the overall tour. That is, given two non-adjacent edges $\overline{AB}$ and $\overline{CD}$, you can replace them with $\overline{AC}$ and $\overline{BD}$ without invalidating your current path. This idea forms the basis of 2-opt, which *iterates over all pairs of non-adjacent edges in an existing tour and performs a swap if doing so reduces the total cost.*

For example, consider the following tour, where there exists a crossing between $\overline{BD}$ and $\overline{EC}$. If we perform 2-opt on this path, we would swap the endpoints of these two edges since it reduces the total weight of our path.



Weight of Path: 6.828



Weight of Path: 6

To swap two edges $(i, i+1)$ and $(j, j+1)$, where i and j are the indices of two vertices in our path, we would replace them with the new edges (i, j) and $(i+1, j+1)$. For instance, swapping vertices B and C on the path on the left turns it into the path on the right, where $\overline{BD}$ and $\overline{CE}$ are replaced with $\overline{BC}$ and $\overline{CD}$. To do this efficiently, we can simply *reverse* the order of all vertices from $i+1$ to j (the STL's `std::reverse()` can be helpful here). For example, consider the path above:

$$A \to B \to D \to C \to E \to F \to A$$

If we wanted to swap the edges $\overline{BD}$ and $\overline{CE}$ ($i = 1$, $j = 3$), we can reverse the order of all vertices between D and C.

$$A \to B \to \underline{\boldsymbol{C \to D}} \to E \to F \to A$$

Similarly, if we wanted to swap $\overline{AB}$ and $\overline{EF}$ ($i = 0$, $j = 4$), we would reverse the order of all vertices between B and E.

$$A \to \underline{\boldsymbol{E \to D \to C \to B}} \to F \to A$$

However, a swap is only completed if it improves the solution. To determine whether a swap is a good idea, we will measure the cost differential of replacing the old edges with the new edges:

$$\textit{cost differential} = \textit{new cost - old cost}$$

In the example above, swapping $\overline{BD}$ and $\overline{CE}$ is a beneficial decision, since it reduces the cost of the overall tour:

$$\textit{cost differential} = \underbrace{dist(\overline{BC}) + dist(\overline{DE})}_{\text{new edges}} - \underbrace{dist(\overline{BD}) - dist(\overline{CE})}_{\text{old edges}} = \textbf{-0.828}$$

On the other hand, swapping $\overline{AB}$ and $\overline{EF}$ would make things worse, so a swap is not made:

$$\textit{cost differential} = \underbrace{dist(\overline{AE}) + dist(\overline{BF})}_{\text{new edges}} - \underbrace{dist(\overline{AB}) - dist(\overline{EF})}_{\text{old edges}} = \textbf{+0.828}$$

One last detail to note is that after a swap is made, more beneficial swaps could arise that did not exist before. Because of this, we will need to *restart* the 2-opt algorithm whenever a swap is made, so that we can discover these new beneficial changes. Failure to do so could cause us to overlook beneficial swaps that were unearthed by previous changes we made to the tour.[3]

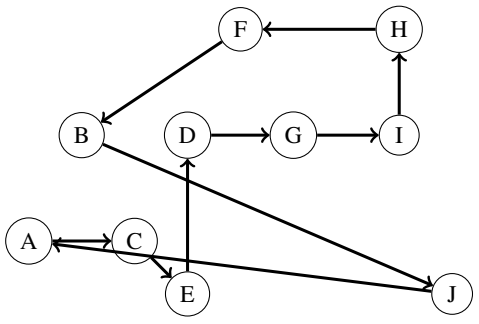
[3] Even though 2-opt may need to be run multiple times, there is a tradeoff between performance and accuracy here. The more times you run 2-opt on your path, the better your path distance will be, but the longer your algorithm will take. We will discuss this a bit more about this over the next few pages.

The pseudocode for the 2-opt heuristic is shown below. We iterate over all pairs of non-adjacent edges in our tour and determine if swapping the two edges improves the solution. If so, we reverse the order of all vertices from index `i + 1` to `j` in our path vector. As long as an iteration of the nested `for` loop discovers a path improvement and performs a swap, we restart another iteration of the outer loop from `i = 0`.[4] This process continues until no improvements can be made by swapping any two non-adjacent edges, giving us our final path.

```
best_distance = calc_distance(current_path)  // path after running nearest neighbor
has_improvement = true  // did you find a beneficial swap during the most recent iteration?
while has_improvement == true:
  has_improvement = false
  for i in range [0, num_cities - 2):
    for j in range [i + 2, num_cities):
      change = dist(i, j) + dist(i + 1, j + 1) - dist(i, i + 1) - dist(j, j + 1)
      if change < 0:
        reverse all vertices from index i + 1 to index j
        best_distance = best_distance + change
        has_improvement = true
```

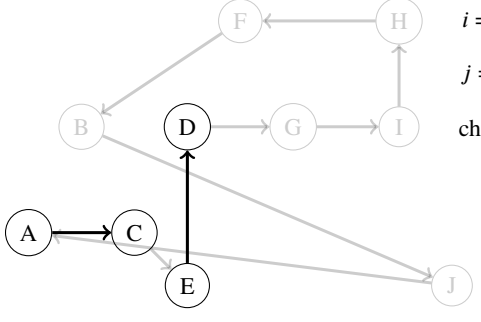
Remark: We will look at several different ways to implement the 2-opt heuristic in this section, so this is not the only way to do things! The implementation we are currently discussing is the complete, standard implementation of 2-opt, but this may not meet the time constraints required for course assignments. Later in this section, we will look at ways to reduce the complexity of the standard 2-opt implementation to meet these requirements (although at the cost of the heuristic's accuracy).

Let's run this 2-opt heuristic on the path we obtained from using nearest neighbor.



Connection	Distance
$A \to C$	2.000
$C \to E$	1.414
$E \to D$	3.000
$D \to G$	2.000
$G \to I$	2.000
$I \to H$	2.000
$H \to F$	3.000
$F \to B$	3.606
$B \to J$	7.616
$J \to A$	8.062
Total	**34.698**

The first pair of edges we consider are $\overline{AC}$ and $\overline{ED}$. However, swapping these two edges does not improve our solution, so a swap is not made.

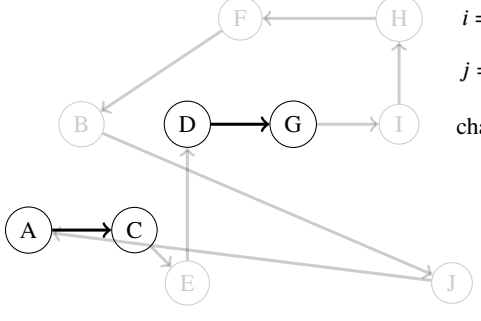


$i = 0$ (A)

$j = 2$ (E)

change = dist($\overline{AE}$) + dist($\overline{CD}$) - dist($\overline{AC}$) - dist($\overline{ED}$)
= 3.162 + 2.236 - 2 - 3 = **+0.398** (↑ cost, no swap)

Next, we consider the edges $\overline{AC}$ and $\overline{DG}$. Swapping these two edges also fails to improve the solution.



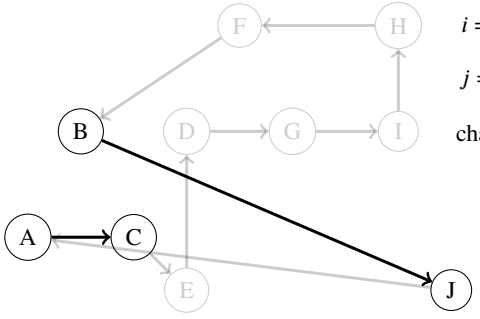
$i = 0$ (A)

$j = 3$ (D)

change = dist($\overline{AD}$) + dist($\overline{CG}$) - dist($\overline{AC}$) - dist($\overline{DG}$)
= 3.606 + 3.606 - 2 - 2 = **+3.212** (↑ cost, no swap)

[4] The path vector should have the starting vertex at both the beginning and the end, e.g., $A \to B \to C \to A$.

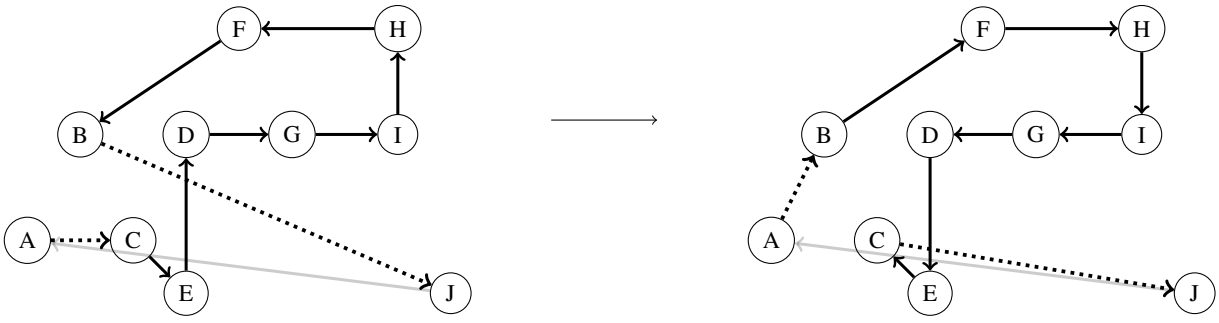
The first edges we encounter that are beneficial to swap are edges $\overline{AC}$ and $\overline{BJ}$. Swapping these two edges gives us an improvement of 1.297, so we perform the swap as shown (the dotted edges represent the edges that are swapped). Notice that this reverses the direction of all edges situated between the two new connections that were added.



$i = 0$ (A)

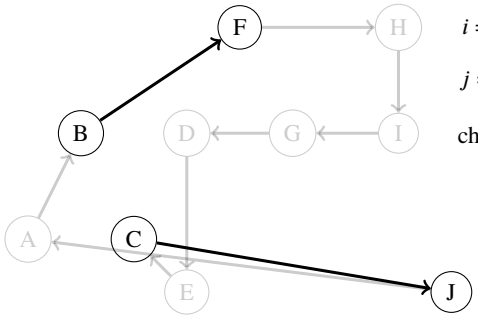
$j = 8$ (B)

change = dist($\overline{AB}$) + dist($\overline{CJ}$) - dist($\overline{AC}$) - dist($\overline{BJ}$)
= 2.236 + 6.083 - 2 - 7.616 = **-1.297** (↓ cost, swap $\overline{AC}$/$\overline{BJ}$ with $\overline{AB}$/$\overline{CJ}$)



Weight of New Solution: 34.698 - 1.297 = **33.401**

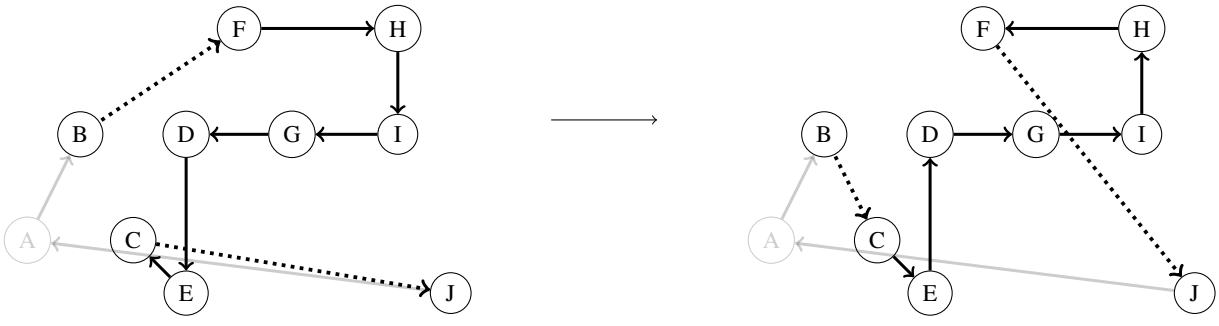
The next beneficial swap we encounter involve the edges $\overline{BF}$ and $\overline{CJ}$.



$i = 1$ (B)

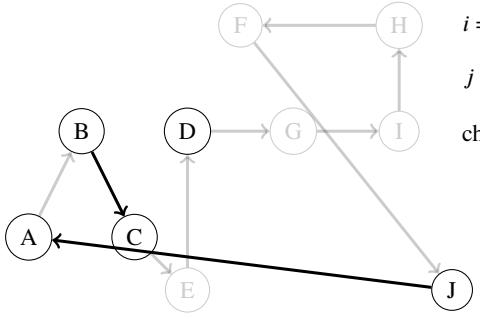
$j = 8$ (C)

change = dist($\overline{BC}$) + dist($\overline{FJ}$) - dist($\overline{BF}$) - dist($\overline{CJ}$)
= 2.236 + 6.403 - 3.606 - 6.083 = **-1.049** (↓ cost, swap $\overline{BF}$/$\overline{CJ}$ with $\overline{BC}$/$\overline{FJ}$)



Weight of New Solution: 33.401 - 1.049 = **32.352**

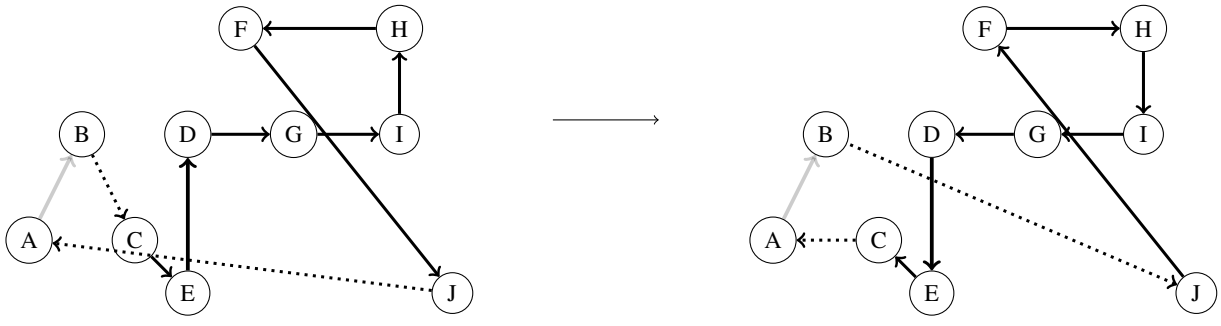
The remaining swaps performed during the 2-opt process are shown below.



$i = 1$ (B)

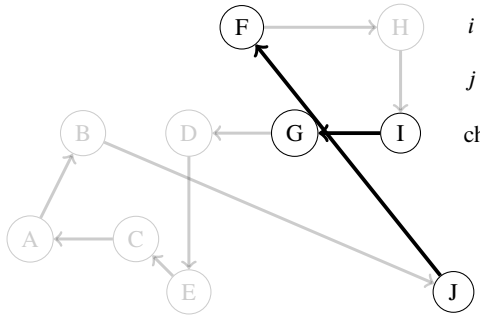
$j = 9$ (J)

change = dist($\overline{BJ}$) + dist($\overline{CA}$) - dist($\overline{BC}$) - dist($\overline{JA}$)
= 7.616 + 2 - 2.236 - 8.062 = **-0.683** (↓ cost, swap $\overline{B}\overline{C}/\overline{J}\overline{A}$ with $\overline{B}\overline{J}/\overline{C}\overline{A}$)



Weight of New Solution: 32.352 - 0.683 = **31.669**

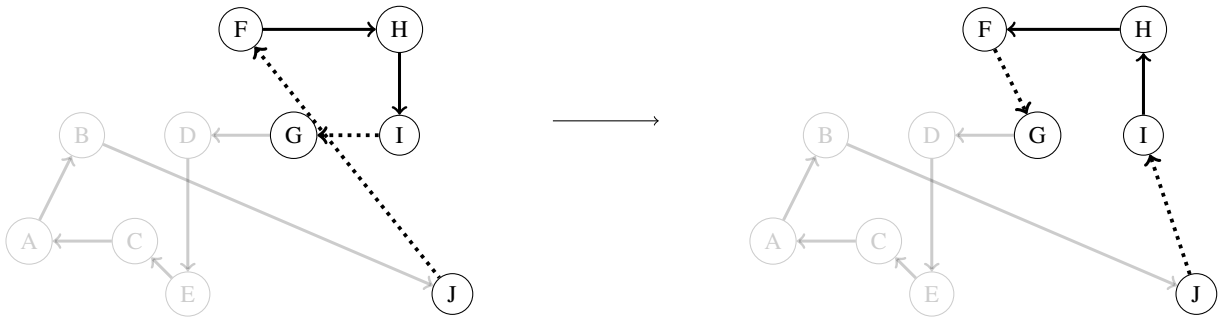
(... a few iterations omitted ...)



$i = 2$ (J)

$j = 5$ (I)

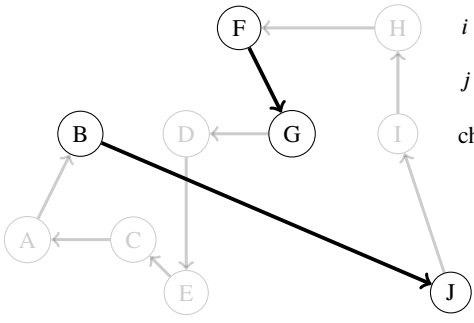
change = dist($\overline{JI}$) + dist($\overline{FG}$) - dist($\overline{JF}$) - dist($\overline{IG}$)
= 3.162 + 2.236 - 6.403 - 2 = **-3.005** (↓ cost, swap $\overline{J}\overline{F}/\overline{I}\overline{G}$ with $\overline{J}\overline{I}/\overline{F}\overline{G}$)



Weight of New Solution: 31.669 - 3.005 = **28.664**

(... a few iterations omitted ...)

The first iteration of the nested `for` loop completes here, since there are no more beneficial swaps among adjacent pairs for $i > 2$ (i.e., pairs situated after vertex J, which is at index 2 of our current path). Since we were able to improve the path in this first iteration (i.e., `has_improvement` is `true`), we begin a second iteration again from `i = 0`. This allows us to discover new swaps that can be made after previous changes we made to the tour. During this second iteration, the first beneficial swap we encounter is between $\overline{BJ}$ and $\overline{FG}$:

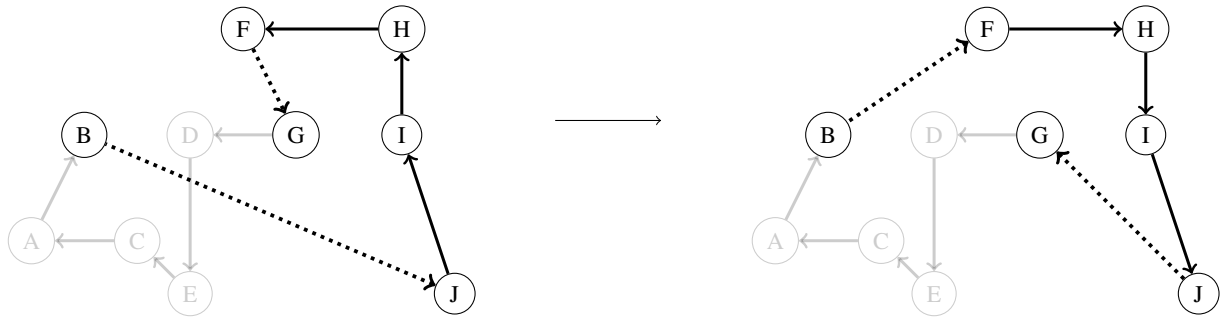


$i = 1$ (B)

$j = 5$ (F)

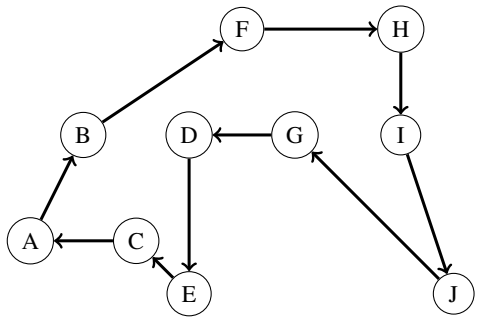
change = dist($\overline{BF}$) + dist($\overline{JG}$) - dist($\overline{BJ}$) - dist($\overline{FG}$)

= 3.606 + 4.243 - 7.616 - 2.236 = **-2.003** (↓ cost, swap $\overline{BJ}/\overline{FG}$ with $\overline{BF}/\overline{JG}$)



Weight of New Solution: 28.664 - 2.003 = **26.661**

No other beneficial swaps are found during this second iteration. Since we were still able to improve the path during our previous iteration, we will begin a third iteration of the nested `for` loop using this updated path. However, this third iteration fails to find any beneficial swaps, so `has_improvement` remains `false`. This causes the `while` loop to terminate, completing our algorithm. The final tour is shown below:



Current Weight of Path: 26.661

Note that 2-opt is a heuristic, so this solution is *not guaranteed* to be the actual optimal solution! In fact, you can see that this path is actually not optimal: during the second iteration, we could have gotten a better solution had we swapped $\overline{BJ}$ and $\overline{DE}$ ($i = 1$, $j = 7$) instead of $\overline{BJ}$ and $\overline{FG}$ ($i = 1$, $j = 5$). However, our algorithm does not know this, so we perform a swap as soon as we encounter one that is beneficial — this causes the less optimal swap to be performed (since $j = 5$ is considered before $j = 7$). Since swapping one of these pairs precludes us from swapping the other, the algorithm as currently implemented does not find the optimal solution. However, this is okay, since the goal of a heuristic is to find a solution that is close to optimal in a reasonable amount of time (which 2-opt successfully achieves).

What is the time complexity of this 2-opt implementation (reproduced below)? Let's first look at the nested `for` loop spanning from lines 5-11. Here, the body of the inner `for` loop can take up to $\Theta(n)$ time, since we need to perform a reversal whenever a swap is made (and reversing takes linear time). Since this work is the most expensive step and may be run up to $\Theta(n^2)$ times when looping over all pairs of non-adjacent edges (lines 5 and 6), the overall worst-case time complexity of the nested `for` loop is $\Theta(n^2) \times \Theta(n) = \Theta(n^3)$.

```
1   best_distance = calc_distance(current_path)  // path after running nearest neighbor
2   has_improvement = true  // did you find a beneficial swap during the most recent iteration?
3   while has_improvement == true:
4     has_improvement = false
5     for i in range [0, num_cities - 2):
6       for j in range [i + 2, num_cities):
7         change = dist(i, j) + dist(i + 1, j + 1) - dist(i, i + 1) - dist(j, j + 1)
8         if change < 0:
9           reverse all vertices from index i + 1 to index j
10          best_distance = best_distance + change
11          has_improvement = true
```

However, what about the outer `while` loop that repeats as long as an improvement has been made? The nested `for` loop on lines 5 and 6 may take polynomial time, but the overall performance of 2-opt also depends on how many times the outer `while` loop on line 3 executes.

This outer `while` loop runs as long as there are pairs in our path that can be swapped to further improve the path's total distance. Just how many times can this happen? The answer to this problem is not trivial. van Leeuwen and Schoone (1980) proved that an algorithm which removes intersections from a Hamiltonian path of n vertices on an Euclidean plane will ultimately produce an intersection-free tour after $O(n^3)$ removals, regardless of the order in which these intersections are removed.[5] However, this $O(n^3)$ bound assumes the restriction that 2-opt can only perform swaps to remove edges that intersect in the path. If we are allowed to perform a swap between any pair of non-adjacent edges that reduces total cost (regardless of whether the pair forms an intersection in the path), then the total number of swaps performed by 2-opt can become exponential in the worst case, as proven by Englert, Röglin, and Vöcking (2014).[6]

To avoid this, one optimization is to disregard the outer `while` loop entirely. Instead, the nested `for` loop is run a constant number of times, regardless of the number of improvements that are made. By doing so, we sacrifice the accuracy of the algorithm for improved efficiency. The following pseudocode is similar to the one above, except that it only runs the nested `for` loop once.

```
1   best_distance = calc_distance(current_path)  // path after running nearest neighbor
2   for i in range [0, num_cities - 2):
3     for j in range [i + 2, num_cities):
4       change = dist(i, j) + dist(i + 1, j + 1) - dist(i, i + 1) - dist(j, j + 1)
5       if change < 0:
6         reverse all vertices from index i + 1 to index j
7         best_distance = best_distance + change
```

A slightly more accurate (but less performant) implementation would still run the nested `for` loop a constant number of times, but would instead *restart* the inner `for` loop whenever an improvement is made (note that the *inner* loop is restarted and not the outer loop). This ensures that you can still perform beneficial swaps that are unearthed by previous swaps within the same iteration of the inner loop without having to restart the entire algorithm. This is shown below (again, this only runs the nested loops once, but you can tweak this based on your needs for performance vs. accurary — if you run the algorithm more times, you will get a better approximation, but your performance will also be slower).

```
1   best_distance = calc_distance(current_path)  // path after running nearest neighbor
2   for i in range [0, num_cities - 2):
3     for j in range [i + 2, num_cities):
4       change = dist(i, j) + dist(i + 1, j + 1) - dist(i, i + 1) - dist(j, j + 1)
5       if change < 0:
6         reverse all vertices from index i + 1 to index j
7         best_distance = best_distance + change
8         restart inner for loop from j = i + 2
```

Remark: The 2-opt heuristic improves an existing solution by reconnecting *pairs* of non-adjacent edges until no more pairs can be swapped to further improve the solution. However, 2-opt is not the only heuristic that uses this strategy to optimize a tour. Another heuristic, *3-opt*, does the same thing, but it instead reconnects edges in groups of three rather than two.

Heuristics such as 2-opt and 3-opt are often defined as *k-opt* heuristics, which improve existing solutions by repeatedly reconnecting k edges of a tour until no more changes can be made to improve the solution. A 3-opt approach often yields a better approximation of the optimal solution than 2-opt, but it is also more computationally expensive (and therefore slower to run). For this class, you will not need to worry about 3-opt (or any value of k above that) — 2-opt should be more than enough for any TSP problem you are required to solve.

[5] "Untangling a Traveling Salesman Tour in the Plane" by van Leeuwen and Schoone (1980).
[6] "Worst Case and Probabilistic Analysis of the 2-Opt Algorithm for the TSP" by Englert, Röglin, and Vöcking (2014).

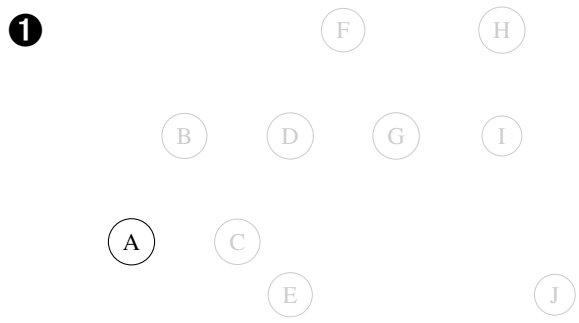
※ 22.4.2 *Nearest Insertion*

Another group of heuristics that can be used to approximate a solution for TSP are *insertion heuristics*. Insertion heuristics work by judicially adding vertices to a partial path in a way that minimizes the additional cost incurred at each step. In this section, we will introduce three insertion heuristics that can produce a TSP estimate in $\Theta(n^2)$ time: nearest, farthest, and arbitrary insertion.

Nearest Insertion

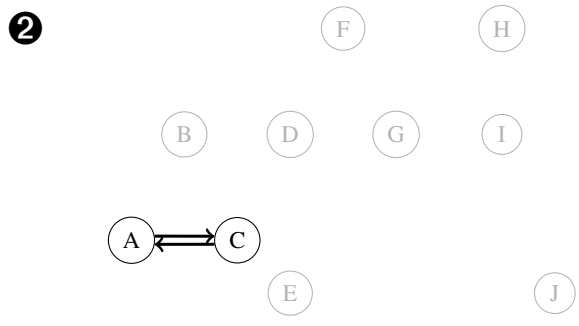
1. Initialize a partial tour with a vertex i, chosen arbitrarily (you can just start with the first vertex available).
2. Identify the vertex j such that the distance between i and j is *minimum*, and set the partial tour to $i \to j \to i$.
3. Identify the vertex k *not* in the partial tour that is *closest* to any node that is already in the partial tour.
4. Find the best place to insert vertex k into the partial tour to minimize cost. To do this, first identify the edge (i, j) in the partial path such that $c_{ik} + c_{kj} - c_{ij}$ is minimal (where c_{ij} denotes the distance between i and j), and then insert k between i and j. Notice that $c_{ik} + c_{kj} - c_{ij}$ represents the change in overall cost after vertex k is added in between vertices i and j, since you are removing the cost between i and j and adding the costs incurred by connecting k with i and j.
5. Once all the vertices have been added to the path, the algorithm completes. Otherwise, repeat steps 3-5 until all vertices have been added.

For example, let's run the nearest insertion heuristic on our previous points, using vertex A as our starting vertex:



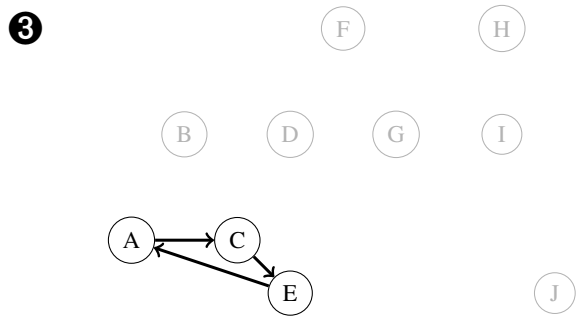
Current Weight of Path: 0.000

First, we find the closest vertex to A, which in this case is vertex C. Thus, we will start our initial path as $A \to C \to A$:



Current Weight of Path: 4.000

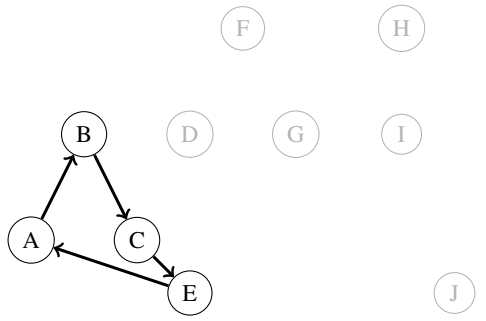
Next, we want to find the vertex *not* in our current path that is closest to any of the existing vertices in our path. Here, the closest vertex is vertex E, as it is closest to vertex C. We then iterate over the edges of our current path to find the position to insert vertex E. In this case, there are only two vertices connected, so the only option is to place E in between A and C.



Current Weight of Path: 6.576

The next vertex to add is vertex B, since it is closest to the vertices in our current partial path. The optimal position to insert B is between A and C, since it minimizes the cost of adding B to the path.

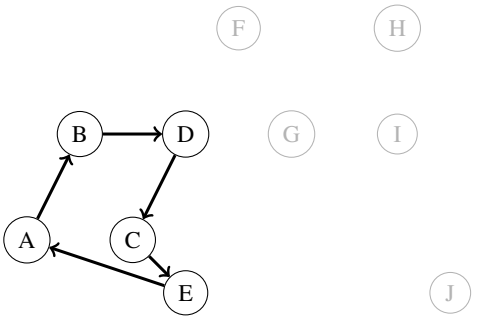
❹



Current Weight of Path: 9.048

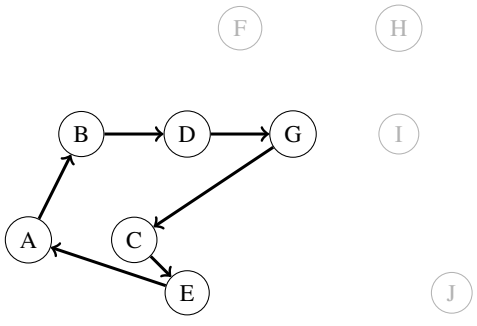
Continuing this process, we would build our tour in the following manner:

❺



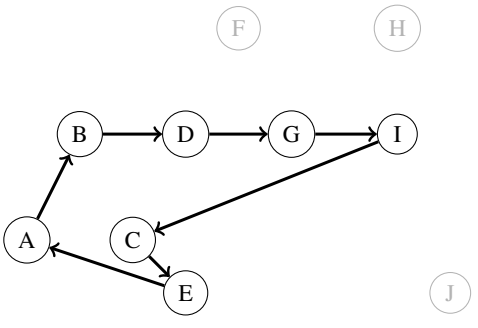
Current Weight of Path: 11.048

❻



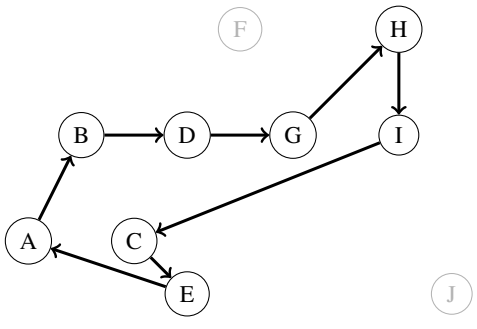
Current Weight of Path: 14.418

❼



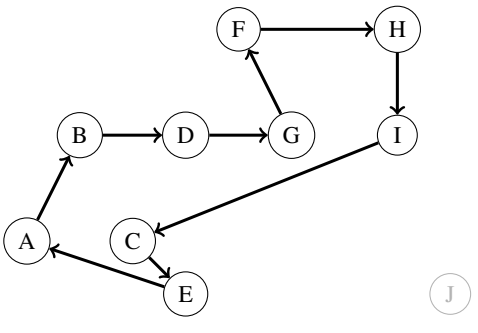
Current Weight of Path: 18.197

❽



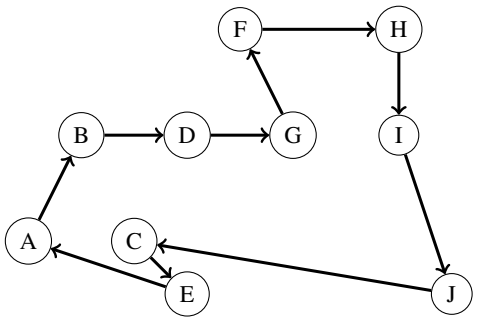
Current Weight of Path: 21.025

❾



Current Weight of Path: 23.433

❿



Current Weight of Path: 27.293

This the final result of the nearest insertion heuristic, with total weight 27.293.

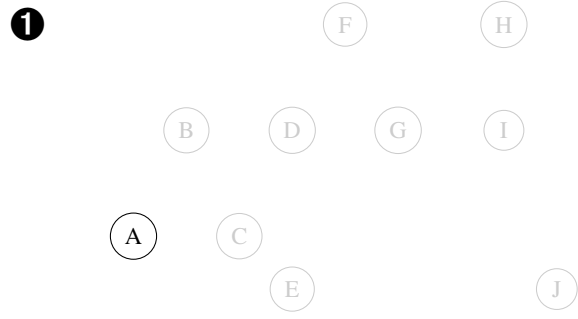
※ 22.4.3 *Farthest Insertion*

Farthest insertion is a similar heuristic, but as its name implies, it inserts vertices that are *farthest* from the current partial path rather than *nearest*. The steps for this heuristic are as shown:

Farthest Insertion

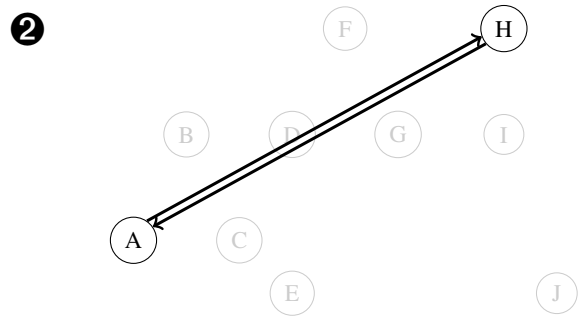
1. Initialize a partial tour with a vertex i, chosen arbitrarily (you can just start with the first vertex available).
2. Identify the vertex j such that the distance between i and j is *maximum*, and set the partial tour to $i \to j \to i$.
3. Identify the vertex k *not* in the partial tour that is *farthest* from any node that is already in the partial tour.
4. Find the best place to insert vertex k into the partial tour to minimize cost. To do this, first identify the edge (i, j) in the partial path such that $c_{ik} + c_{kj} - c_{ij}$ is minimal (where c_{ij} denotes the distance between i and j), and then insert k between i and j. Notice that $c_{ik} + c_{kj} - c_{ij}$ represents the change in overall cost after vertex k is added in between vertices i and j, since you are removing the cost between i and j and adding the costs incurred by connecting k with i and j.
5. Once all the vertices have been added to the path, the algorithm completes. Otherwise, repeat steps 3-5 until all vertices have been added.

Let's run the farthest insertion heuristic on our previous points, again with vertex A as our starting vertex:



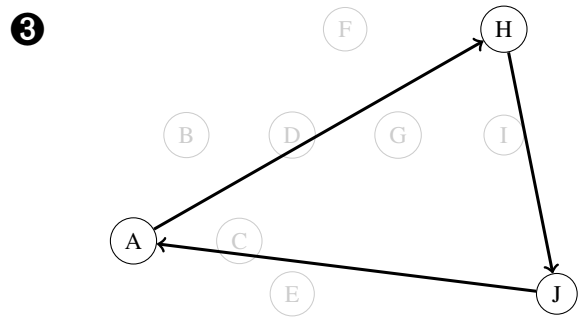
Current Weight of Path: 0.000

First, we find the farthest vertex from A, which in this case is vertex H. Thus, we will start our initial path as $A \to H \to A$:



Current Weight of Path: 16.124

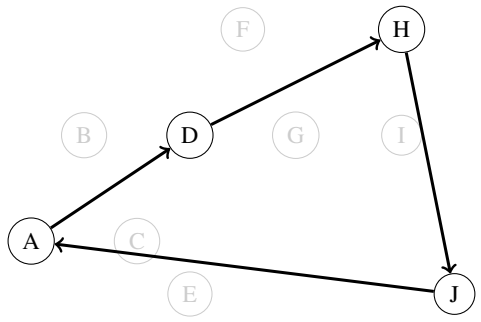
Next, we want to find the vertex *not* in our current path that is farthest from any of the existing vertices in our path, which in this case is vertex J. We then iterate over all the edges of our existing path to determine the best position to insert J. Here, there are only two vertices connected, so the only option is to place J in between A and H.



Current Weight of Path: 21.223

The next vertex to add is vertex D, since it is farthest from any of the vertices in our current partial path. The optimal position to insert D is between A and H, since it minimizes the cost of adding D to the path.

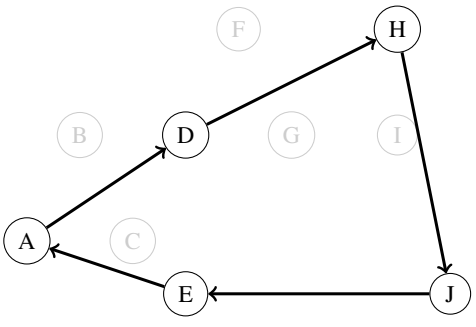
❹



Current Weight of Path: 21.239

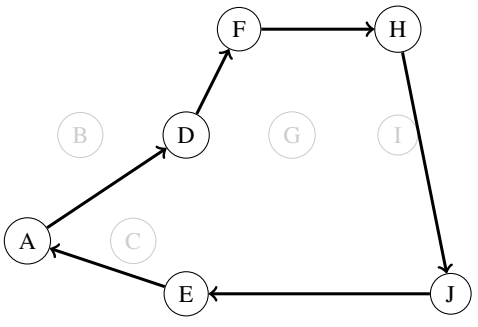
Continuing this process, we would build our tour in the following manner:

❺



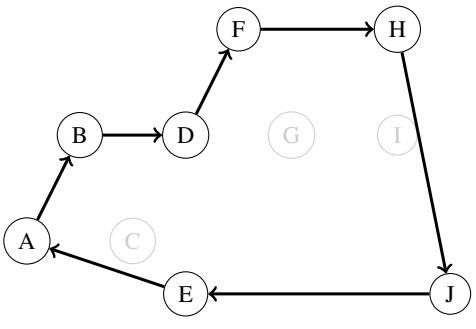
Current Weight of Path: 21.339

❻



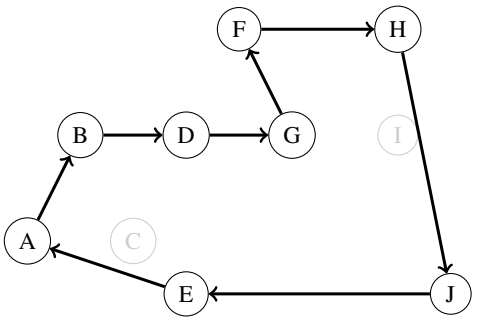
Current Weight of Path: 22.103

❼



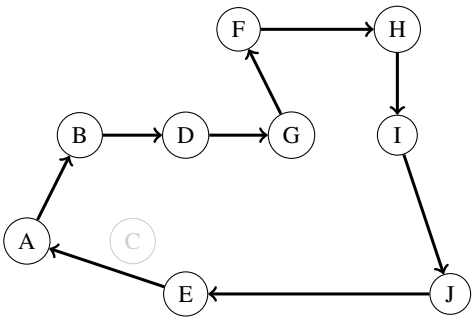
Current Weight of Path: 22.733

❽



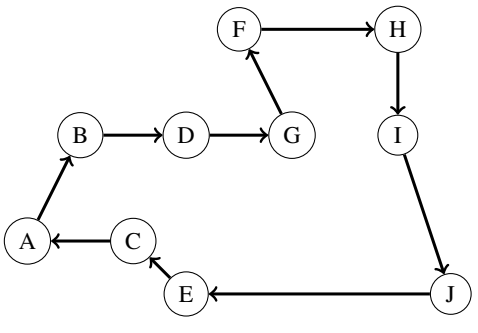
Current Weight of Path: 24.733

❾



Current Weight of Path: 24.796

❿



Current Weight of Path: 25.049

This is the final result of the farthest insertion heuristic, with total weight 25.049.

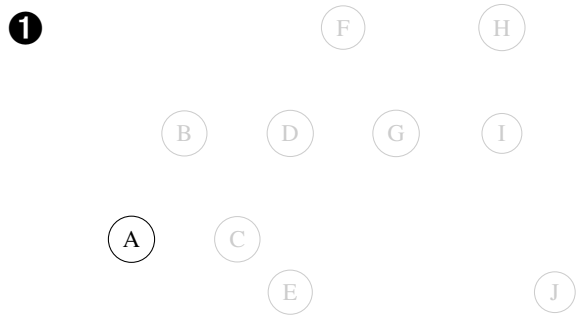
※ 22.4.4 *Arbitrary Insertion*

Lastly, we have arbitrary insertion, which arbitrarily selects vertices to add to the partial path (instead of finding the vertex that is closest or farthest away). The steps of arbitrary insertion are essentially identical to those of nearest insertion, without the step of identifying the vertex that is closest to any node already in the partial tour.

Arbitrary Insertion

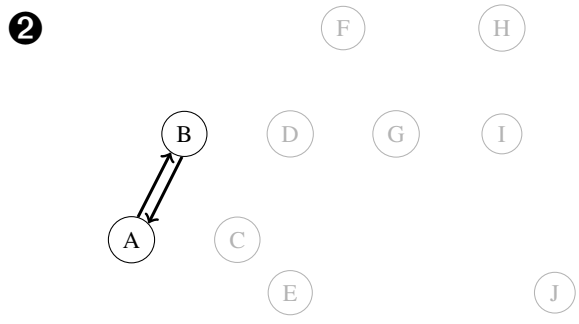
1. Initialize a partial tour with a vertex i, chosen arbitrarily (you can just start with the first vertex available).
2. Choose another arbitrary vertex j and set the initial partial tour to $i \to j \to i$.
3. Arbitrarily select a vertex k that is currently not in the partial tour.
4. Find the best place to insert vertex k into the partial tour to minimize cost. To do this, first identify the edge (i, j) in the partial path such that $c_{ik} + c_{kj} - c_{ij}$ is minimal (where c_{ij} denotes the distance between i and j), and then insert k between i and j. Notice that $c_{ik} + c_{kj} - c_{ij}$ represents the change in overall cost after vertex k is added in between vertices i and j, since you are removing the cost between i and j and adding the costs incurred by connecting k with i and j.
5. Once all the vertices have been added to the path, the algorithm completes. Otherwise, repeat steps 3-5 until all vertices have been added.

For example, let's run arbitrary insertion on the previous set of points. Since vertices can be chosen arbitrarily, we will add vertices to the path in alphabetical order. We begin with A as our starting vertex:



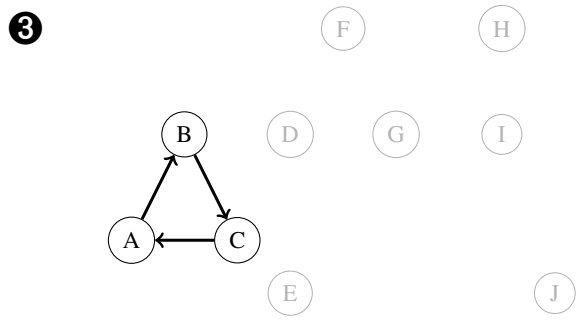
Current Weight of Path: 0.000

Next, we choose an arbitrary vertex and add it to our path. Since we are adding vertices in alphabetical order, we will add vertex B:



Current Weight of Path: 4.472

Then, we add the unvisited vertex that is next alphabetically within the collection of unvisited vertices, or vertex C:

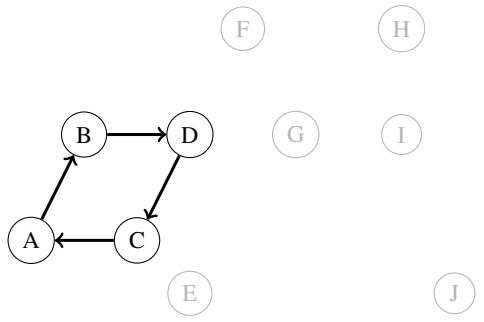


Current Weight of Path: 6.472

Remark: Since you are selecting points arbitrarily when deciding how to build your path, it does not matter what your starting points are (unlike nearest or farthest insertion, which require you to start with the unvisited vertex that is nearest or farthest from the starting vertex). As a result, with arbitrary insertion, you can skip the first two iterations above and just start your partial tour as $A \to B \to C \to A$.

The next vertex to add is vertex *D*, which is inserted into the path as shown:

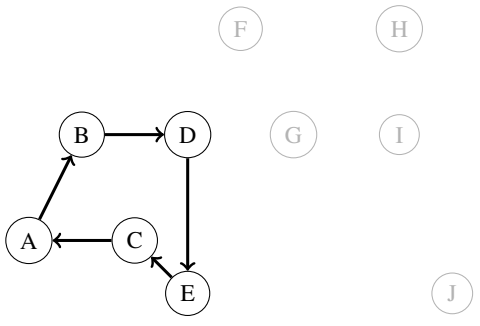
❹



Current Weight of Path: 8.472

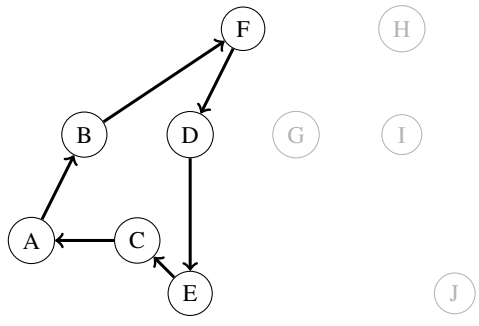
Continuing this process, we would build our tour in the following manner:

❺



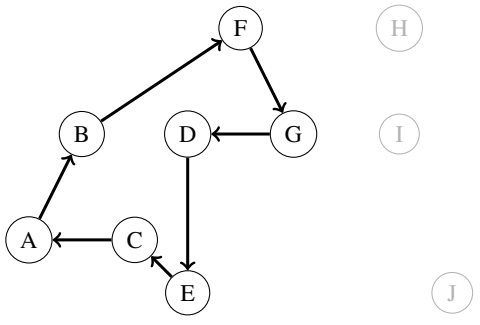
Current Weight of Path: 10.650

❻



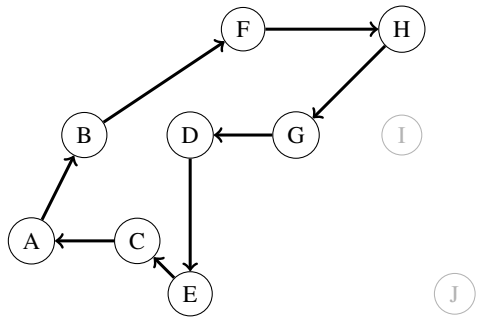
Current Weight of Path: 14.492

❼



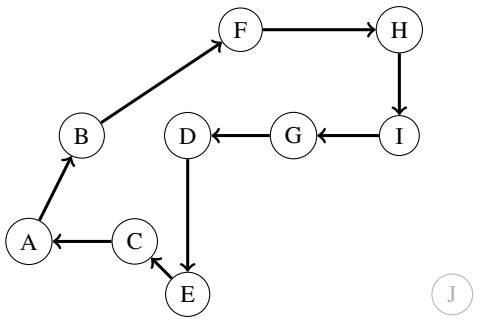
Current Weight of Path: 16.492

❽



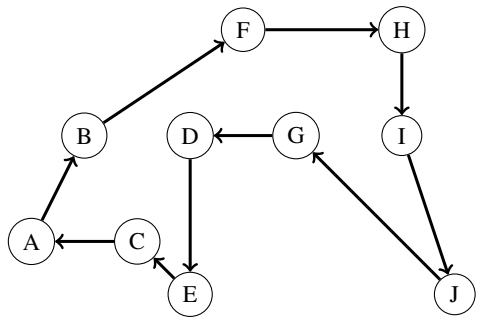
Current Weight of Path: 20.084

❾



Current Weight of Path: 21.256

❿



Current Weight of Path: 26.661

This is the final result of the arbitrary insertion heuristic, with total weight 26.661. As you can see, all three of these insertion heurstics were able to produce solutions that were close to optimal (25.049) in $\Theta(n^2)$ time.[7] If you aren't concerned with finding the exact optimal solution for a problem, a heuristic can provide a significant improvement over a branch-and-bound approach!

Remark: The heuristics we have covered so far are not exhaustive, and many other heuristics exist for the traveling salesperson problem. For instance, *cheapest insertion* is another insertion heuristic that behaves similarly to nearest insertion, but instead of choosing to insert the vertex that is closest to any vertex the current partial tour, it chooses the vertex that, when added, results in the shortest possible tour. *Christofides algorithm* is a heuristic that uses advanced graph concepts to approximate a solution that is always within a factor of 3/2 of the optimal solution. The *Lin-Kernighan heuristic* is an adaptive local-search heuristic that dynamically adjusts between different k-opt heuristics to produce an estimate that is closer to optimal.

Many of these more advanced heuristics may deliver better approximations, but similar the 3-opt approach discussed earlier, they are also a lot more expensive to compute. For the class, a $\Theta(n^2)$ heuristic such as nearest, farthest, or arbitrary insertion is enough for generating a reasonably close approximation for TSP in a short amount of time, which is why these additional heuristics are not covered in more detail.

So far, we have talked about how heuristics can be useful for approximating TSP in polynomial time. However, what if you absolutely needed to know the exact optimal solution? Would heuristics be useless in such a scenario?

Although heuristics cannot guarantee optimality, they can still be useful in helping you find the optimal solution during the branch-and-bound process. Recall that the efficiency of branch-and-bound for TSP relies on an accurate *upper bound*, which is used to prune branches we can guarantee are sub-optimal. Currently, we initially set the upper bound to the cost of the first complete solution we encounter (which we arbitrarily decided as the path $A \to B \to C \to D \to E \to F \to G \to H \to I \to J \to A$). Every branch that we explore later would then be compared to this initial bound until a better solution is found.

Since the upper bound is used as a threshold to decide whether a partial branch is promising, we want our upper bound to be as close as possible to the actual optimal solution to support efficient pruning. However, there is no guarantee that the first complete solution we find is close to optimal! If our first solution is too costly, we would waste time exploring sub-optimal branches at the beginning of our algorithm.

To address this issue, we can instead set our initial upper bound to *the solution of a heuristic* rather than the first solution we find during our search. This ensures that our initial upper bound is close-to-optimal, allowing us to prune branches efficiently from the get-go. Even though a heuristic may take time to run, the cost savings of setting a tighter upper bound is often well worth this extra effort!

[7] Don't judge the performance of each heuristic based on this example alone! Even though farthest insertion ended up performing the best, this depends on the input. Given another set of cities, it is entirely possible for a different heuristic to perform better.