Algorithms and Analysis

Lesson 28: *Know What's Important*

Optimising code, strategies

Outline

1. **Time Complexity**
2. Strategies
   - Brute Force Methods
   - Divide and Conquer
   - Greedy Algorithms
   - Dynamic Programming
   - Linear Programming
   - Backtracking
   - Heuristic Search

Designing Algorithms from Scratch

- In writing code you will come across new problems
- Your first task is to see if you can map this onto a problem with a well known solution
- If not you are going to have to come up with a solution
- Two questions you need to ask yourself are
  - ⋆ How efficient does my solution need to be?
  - ⋆ How do I go about solving the problem?

Designing Algorithms from Scratch

- In writing code you will come across new problems—at least if your lucky
- Your first task is to see if you can map this onto a problem with a well known solution
- If not you are going to have to come up with a solution
- Two questions you need to ask yourself are
  - ⋆ How efficient does my solution need to be?
  - ⋆ How do I go about solving the problem?

Designing Algorithms from Scratch

- In writing code you will come across new problems—at least if your lucky
- Your first task is to see if you can map this onto a problem with a well known solution
- If not you are going to have to come up with a solution
- Two questions you need to ask yourself are
  - ⋆ How efficient does my solution need to be?
  - ⋆ How do I go about solving the problem?

Designing Algorithms from Scratch

- In writing code you will come across new problems—at least if your lucky
- Your first task is to see if you can map this onto a problem with a well known solution
- If not you are going to have to come up with a solution
- Two questions you need to ask yourself are
  - ⋆ How efficient does my solution need to be?
  - ⋆ How do I go about solving the problem?

Designing Algorithms from Scratch

- In writing code you will come across new problems—at least if your lucky
- Your first task is to see if you can map this onto a problem with a well known solution
- If not you are going to have to come up with a solution
- Two questions you need to ask yourself are
  - ⋆ How efficient does my solution need to be?
  - ⋆ How do I go about solving the problem?

Designing Algorithms from Scratch

- In writing code you will come across new problems—at least if your lucky
- Your first task is to see if you can map this onto a problem with a well known solution
- If not you are going to have to come up with a solution
- Two questions you need to ask yourself are
  - ⋆ How efficient does my solution need to be?
  - ⋆ How do I go about solving the problem?

Is Solving the Problem Time Critical?

- The majority of code is not time critical
- Even when programs are slow there is usually only one part of the code which takes almost all the time
- However there are times when solving the problem naïvely is going to take too long
- Advice on improving the performance of your code is not hard to come by. . .

Is Solving the Problem Time Critical?

- The majority of code is not time critical
- Even when programs are slow there is usually only one part of the code which takes almost all the time
- However there are times when solving the problem naïvely is going to take too long
- Advice on improving the performance of your code is not hard to come by. . .

Is Solving the Problem Time Critical?

- The majority of code is not time critical
- Even when programs are slow there is usually only one part of the code which takes almost all the time
- However there are times when solving the problem naïvely is going to take too long
- Advice on improving the performance of your code is not hard to come by. . .

Is Solving the Problem Time Critical?

- The majority of code is not time critical
- Even when programs are slow there is usually only one part of the code which takes almost all the time
- However there are times when solving the problem naïvely is going to take too long
- Advice on improving the performance of your code is not hard to come by. . .

Advice on Optimising Code

- "More computing sins are committed in the name of efficiency (without necessarily achieving it) than any other single reason—including blind stupidity" W. Wulf
- "We *should* forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil." D. Knuth
- "We follow two rules in the matter of optimization:
  - ⋆ Rule 1. Don't do it
  - ⋆ Rule 2 (for experts only). Don't do it yet—that is, not until you have a perfectly clear and unoptimized solution"

  M. A. Jackson
- "Strive to write good programs rather than fast ones" J. Block

Advice on Optimising Code

- "More computing sins are committed in the name of efficiency (without necessarily achieving it) than any other single reason—including blind stupidity" W. Wulf
- "We *should* forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil." D. Knuth
- "We follow two rules in the matter of optimization:
  - ⋆ Rule 1. Don't do it
  - ⋆ Rule 2 (for experts only). Don't do it yet—that is, not until you have a perfectly clear and unoptimized solution"

  M. A. Jackson
- "Strive to write good programs rather than fast ones" J. Block

Advice on Optimising Code

- "More computing sins are committed in the name of efficiency (without necessarily achieving it) than any other single reason—including blind stupidity" W. Wulf
- "We *should* forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil." D. Knuth
- "We follow two rules in the matter of optimization:
  - ⋆ Rule 1. Don't do it
  - ⋆ Rule 2 (for experts only). Don't do it yet—that is, not until you have a perfectly clear and unoptimized solution"

  M. A. Jackson
- "Strive to write good programs rather than fast ones" J. Block

Advice on Optimising Code

- "More computing sins are committed in the name of efficiency (without necessarily achieving it) than any other single reason—including blind stupidity" W. Wulf
- "We *should* forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil." D. Knuth
- "We follow two rules in the matter of optimization:
  - ⋆ Rule 1. Don't do it
  - ⋆ Rule 2 (for experts only). Don't do it yet—that is, not until you have a perfectly clear and unoptimized solution"

  M. A. Jackson
- "Strive to write good programs rather than fast ones" J. Block

Time Complexity

- Ignoring this good advice, your next job is to decide what is the time complexity of your algorithm and whether you can tolerate that time complexity?

- Typically your algorithm will be
  - ⋆ constant or (log)-linear time ☺
  - ⋆ quadratic/cubic,. . .
  - ⋆ exponential time ☹

- Lets deal with this in inverse order

Time Complexity

- Ignoring this good advice, your next job is to decide what is the time complexity of your algorithm and whether you can tolerate that time complexity?

- Typically your algorithm will be
  - ⋆ constant or (log)-linear time ☺
  - ⋆ quadratic/cubic,. . .
  - ⋆ exponential time ☹

- Lets deal with this in inverse order

Time Complexity

- Ignoring this good advice, your next job is to decide what is the time complexity of your algorithm and whether you can tolerate that time complexity?

- Typically your algorithm will be
  - ⋆ constant or (log)-linear time ☺
  - ⋆ quadratic/cubic,. . .
  - ⋆ exponential time ☹

- Lets deal with this in inverse order

Exponential Time

- You can often tolerate this if your problem is small
- E.g. you're trying to solve a small puzzle
- or planning a short sequence of actions
- However, in many cases exponential is just too long
  - ⋆ Look for an efficient solution (e.g. dynamics programming)
  - ⋆ Redefine the problem (e.g. use a linear approximation)
  - ⋆ Settle for a sub-optimal solution (e.g. using heuristic search)

Exponential Time

- You can often tolerate this if your problem is small
- E.g. you're trying to solve a small puzzle
- or planning a short sequence of actions
- However, in many cases exponential is just too long
  - ⋆ Look for an efficient solution (e.g. dynamics programming)
  - ⋆ Redefine the problem (e.g. use a linear approximation)
  - ⋆ Settle for a sub-optimal solution (e.g. using heuristic search)

Exponential Time

- You can often tolerate this if your problem is small
- E.g. you're trying to solve a small puzzle
- or planning a short sequence of actions
- However, in many cases exponential is just too long
  - ⋆ Look for an efficient solution (e.g. dynamics programming)
  - ⋆ Redefine the problem (e.g. use a linear approximation)
  - ⋆ Settle for a sub-optimal solution (e.g. using heuristic search)

Exponential Time

- You can often tolerate this if your problem is small
- E.g. you're trying to solve a small puzzle
- or planning a short sequence of actions
- However, in many cases exponential is just too long
  - ⋆ Look for an efficient solution (e.g. dynamics programming)
  - ⋆ Redefine the problem (e.g. use a linear approximation)
  - ⋆ Settle for a sub-optimal solution (e.g. using heuristic search)

Exponential Time

- You can often tolerate this if your problem is small
- E.g. you're trying to solve a small puzzle
- or planning a short sequence of actions
- However, in many cases exponential is just too long
  - ⋆ Look for an efficient solution (e.g. dynamics programming)
  - ⋆ Redefine the problem (e.g. use a linear approximation)
  - ⋆ Settle for a sub-optimal solution (e.g. using heuristic search)

Exponential Time

- You can often tolerate this if your problem is small
- E.g. you're trying to solve a small puzzle
- or planning a short sequence of actions
- However, in many cases exponential is just too long
  - ⋆ Look for an efficient solution (e.g. dynamics programming)
  - ⋆ Redefine the problem (e.g. use a linear approximation)
  - ⋆ Settle for a sub-optimal solution (e.g. using heuristic search)

Quadratic and Cubic Time

- There are algorithms which are cubic (e.g. LP, inverting a matrix)
- For many applications this is acceptable because there is no hurry for the solution or the problem isn't that big
- For large data sets these algorithms might just be impractical
- Often taking advantage of the structure of the problem (e.g. sparsity) can speed things up (e.g. a good LP package)
- Sometimes quadratic algorithms can be made log-linear using a divide and conquer or a greedy strategy

Quadratic and Cubic Time

- There are algorithms which are cubic (e.g. LP, inverting a matrix)
- For many applications this is acceptable because there is no hurry for the solution or the problem isn't that big
- For large data sets these algorithms might just be impractical
- Often taking advantage of the structure of the problem (e.g. sparsity) can speed things up (e.g. a good LP package)
- Sometimes quadratic algorithms can be made log-linear using a divide and conquer or a greedy strategy

Quadratic and Cubic Time

- There are algorithms which are cubic (e.g. LP, inverting a matrix)
- For many applications this is acceptable because there is no hurry for the solution or the problem isn't that big
- For large data sets these algorithms might just be impractical
- Often taking advantage of the structure of the problem (e.g. sparsity) can speed things up (e.g. a good LP package)
- Sometimes quadratic algorithms can be made log-linear using a divide and conquer or a greedy strategy

Quadratic and Cubic Time

- There are algorithms which are cubic (e.g. LP, inverting a matrix)
- For many applications this is acceptable because there is no hurry for the solution or the problem isn't that big
- For large data sets these algorithms might just be impractical
- Often taking advantage of the structure of the problem (e.g. sparsity) can speed things up (e.g. a good LP package)
- Sometimes quadratic algorithms can be made log-linear using a divide and conquer or a greedy strategy

Quadratic and Cubic Time

- There are algorithms which are cubic (e.g. LP, inverting a matrix)
- For many applications this is acceptable because there is no hurry for the solution or the problem isn't that big
- For large data sets these algorithms might just be impractical
- Often taking advantage of the structure of the problem (e.g. sparsity) can speed things up (e.g. a good LP package)
- Sometimes quadratic algorithms can be made log-linear using a divide and conquer or a greedy strategy

Sub-Quadratic Algorithm

- If you have to run an algorithm on any data of arbitrary size you want a sub-quadratic algorithm
- We have seen this in practice with sorting
- The fast Fourier transform revolutionised digital signal processing when it was introduced (reducing a quadratic algorithm to a log-linear algorithm)
- An active area of research is **big data** where the only algorithms that can be used are sub-quadratic algorithms

Sub-Quadratic Algorithm

- If you have to run an algorithm on any data of arbitrary size you want a sub-quadratic algorithm
- We have seen this in practice with sorting
- The fast Fourier transform revolutionised digital signal processing when it was introduced (reducing a quadratic algorithm to a log-linear algorithm)
- An active area of research is **big data** where the only algorithms that can be used are sub-quadratic algorithms

Sub-Quadratic Algorithm

- If you have to run an algorithm on any data of arbitrary size you want a sub-quadratic algorithm
- We have seen this in practice with sorting
- The fast Fourier transform revolutionised digital signal processing when it was introduced (reducing a quadratic algorithm to a log-linear algorithm)
- An active area of research is **big data** where the only algorithms that can be used are sub-quadratic algorithms

Sub-Quadratic Algorithm

- If you have to run an algorithm on any data of arbitrary size you want a sub-quadratic algorithm
- We have seen this in practice with sorting
- The fast Fourier transform revolutionised digital signal processing when it was introduced (reducing a quadratic algorithm to a log-linear algorithm)
- An active area of research is **big data** where the only algorithms that can be used are sub-quadratic algorithms

Good Code is Fast Code

- Using appropriate data structures and algorithms is by far the best way of speeding up code
- Many coders wedded to arrays try to simulate sets and maps very inefficiently often using code that does not scale
- Changing the structure of a program can lead to huge speed ups
- If you require more speed then concentrate on the inner loop where almost all the work is done (usually gives less than a factor of two)—optimising code in outer loops is pointless
- Using the right strategy will give the biggest speed-up

Good Code is Fast Code

- Using appropriate data structures and algorithms is by far the best way of speeding up code
- Many coders wedded to arrays try to simulate sets and maps very inefficiently often using code that does not scale
- Changing the structure of a program can lead to huge speed ups
- If you require more speed then concentrate on the inner loop where almost all the work is done (usually gives less than a factor of two)—optimising code in outer loops is pointless
- Using the right strategy will give the biggest speed-up

Good Code is Fast Code

- Using appropriate data structures and algorithms is by far the best way of speeding up code
- Many coders wedded to arrays try to simulate sets and maps very inefficiently often using code that does not scale
- Changing the structure of a program can lead to huge speed ups
- If you require more speed then concentrate on the inner loop where almost all the work is done (usually gives less than a factor of two)—optimising code in outer loops is pointless
- Using the right strategy will give the biggest speed-up

Good Code is Fast Code

- Using appropriate data structures and algorithms is by far the best way of speeding up code

- Many coders wedded to arrays try to simulate sets and maps very inefficiently often using code that does not scale

- Changing the structure of a program can lead to huge speed ups

- If you require more speed then concentrate on the inner loop where almost all the work is done (usually gives less than a factor of two)—optimising code in outer loops is pointless

- Using the right strategy will give the biggest speed-up

Good Code is Fast Code

- Using appropriate data structures and algorithms is by far the best way of speeding up code
- Many coders wedded to arrays try to simulate sets and maps very inefficiently often using code that does not scale
- Changing the structure of a program can lead to huge speed ups
- If you require more speed then concentrate on the inner loop where almost all the work is done (usually gives less than a factor of two)—optimising code in outer loops is pointless
- Using the right strategy will give the biggest speed-up

Outline

1. Time Complexity
2. **Strategies**
    - Brute Force Methods
    - Divide and Conquer
    - Greedy Algorithms
    - Dynamic Programming
    - Linear Programming
    - Backtracking
    - Heuristic Search

Algorithmic Strategies

- Good algorithms are difficult to invent
- However, many algorithms follow particular patterns or strategies
- Understanding these strategies is important for deriving new algorithms
- We have seen the classic strategies throughout the course

Algorithmic Strategies

- Good algorithms are difficult to invent
- However, many algorithms follow particular patterns or strategies
- Understanding these strategies is important for deriving new algorithms
- We have seen the classic strategies throughout the course

Algorithmic Strategies

- Good algorithms are difficult to invent
- However, many algorithms follow particular patterns or strategies
- Understanding these strategies is important for deriving new algorithms
- We have seen the classic strategies throughout the course

Algorithmic Strategies

- Good algorithms are difficult to invent
- However, many algorithms follow particular patterns or strategies
- Understanding these strategies is important for deriving new algorithms
- We have seen the classic strategies throughout the course

Brute Force

- Many problems have obvious **brute force** solutions
- E.g. in sorting; selection sort, insertion sort and bubble sort are fairly simple algorithm that do the obvious
- Similarly searching an array using sequential search provides an obvious solution
- Sometimes, brute force methods are the best you can do—e.g. sequential search on an unordered array

Brute Force

- Many problems have obvious **brute force** solutions
- E.g. in sorting; selection sort, insertion sort and bubble sort are fairly simple algorithm that do the obvious
- Similarly searching an array using sequential search provides an obvious solution
- Sometimes, brute force methods are the best you can do—e.g. sequential search on an unordered array

Brute Force

- Many problems have obvious **brute force** solutions
- E.g. in sorting; selection sort, insertion sort and bubble sort are fairly simple algorithm that do the obvious
- Similarly searching an array using sequential search provides an obvious solution
- Sometimes, brute force methods are the best you can do—e.g. sequential search on an unordered array

Brute Force

- Many problems have obvious **brute force** solutions
- E.g. in sorting; selection sort, insertion sort and bubble sort are fairly simple algorithm that do the obvious
- Similarly searching an array using sequential search provides an obvious solution
- Sometimes, brute force methods are the best you can do—e.g. sequential search on an unordered array

Brute Force

- Many problems have obvious **brute force** solutions
- E.g. in sorting; selection sort, insertion sort and bubble sort are fairly simple algorithm that do the obvious
- Similarly searching an array using sequential search provides an obvious solution
- Sometimes, brute force methods are the best you can do—e.g. sequential search on an unordered array

Exhaustive Search

- For optimisation problems such as the travelling salesperson problem, the brute force method is to try all possible solutions
- This **exhaustive search** starts to hurt very quickly and becomes intractable for moderate size problems (e.g. tours of length 20)
- Even for sorting, brute force methods become unattractive when the inputs are long
- We really want to do better

Exhaustive Search

- For optimisation problems such as the travelling salesperson problem, the brute force method is to try all possible solutions
- This **exhaustive search** starts to hurt very quickly and becomes intractable for moderate size problems (e.g. tours of length 20)
- Even for sorting, brute force methods become unattractive when the inputs are long
- We really want to do better

Exhaustive Search

- For optimisation problems such as the travelling salesperson problem, the brute force method is to try all possible solutions
- This **exhaustive search** starts to hurt very quickly and becomes intractable for moderate size problems (e.g. tours of length 20)
- Even for sorting, brute force methods become unattractive when the inputs are long
- We really want to do better

Exhaustive Search

- For optimisation problems such as the travelling salesperson problem, the brute force method is to try all possible solutions
- This **exhaustive search** starts to hurt very quickly and becomes intractable for moderate size problems (e.g. tours of length 20)
- Even for sorting, brute force methods become unattractive when the inputs are long
- We really want to do better

Divide and Conquer

- Another strategy we have met is **divide and conquer**
  - ⋆ Divide the problem into two or more parts
  - ⋆ Solve the parts
  - ⋆ Combine the parts to obtain a full solution
- This needs to be quicker than solving the original problem by brute force
- We can do this division recursively until the problems are trivial to solve

Divide and Conquer

- Another strategy we have met is **divide and conquer**
  - ⋆ Divide the problem into two or more parts
  - ⋆ Solve the parts
  - ⋆ Combine the parts to obtain a full solution
- This needs to be quicker than solving the original problem by brute force
- We can do this division recursively until the problems are trivial to solve

Divide and Conquer

- Another strategy we have met is **divide and conquer**
  - ⋆ Divide the problem into two or more parts
  - ⋆ Solve the parts
  - ⋆ Combine the parts to obtain a full solution
- This needs to be quicker than solving the original problem by brute force
- We can do this division recursively until the problems are trivial to solve

Divide and Conquer

- Another strategy we have met is **divide and conquer**
  - ⋆ Divide the problem into two or more parts
  - ⋆ Solve the parts
  - ⋆ Combine the parts to obtain a full solution
- This needs to be quicker than solving the original problem by brute force
- We can do this division recursively until the problems are trivial to solve

Divide and Conquer Problems

- Algorithms based on this idea include
  - ⋆ Computing the integer power of a number
  - ⋆ Binary search
  - ⋆ Merge sort
  - ⋆ Quick sort
  - ⋆ Fast Fourier transform
- Often implemented using recursion
- It's nice when it works—but not all problems allow this

Divide and Conquer Problems

- Algorithms based on this idea include
  - ⋆ Computing the integer power of a number
  - ⋆ Binary search
  - ⋆ Merge sort
  - ⋆ Quick sort
  - ⋆ Fast Fourier transform
- Often implemented using recursion
- It's nice when it works—but not all problems allow this

Divide and Conquer Problems

- Algorithms based on this idea include
  - ⋆ Computing the integer power of a number
  - ⋆ Binary search
  - ⋆ Merge sort
  - ⋆ Quick sort
  - ⋆ Fast Fourier transform
- Often implemented using recursion
- It's nice when it works—but not all problems allow this

Fast Fourier Transform

- The Fourier Transform provides a different "view" of a sequence such as a signals, images, etc.

$$\tilde{f}(\boldsymbol{k}) = \sum_{x_1=0}^{n-1} \cdots \sum_{x_d=0}^{n-1} f(\boldsymbol{x})\, \mathrm{e}^{2\pi \mathrm{i} \boldsymbol{k}\cdot\boldsymbol{x}/n}$$

- The **Fast Fourier Transform** was an algorithm divised by John Tukey and James Cooley in 1965 to compute the Fourier Transform quickly

- Gauss had used exactly this idea in a paper in 1805 to save himself work computing a Fourier transform by hand

- It is based on a divide-and-conquer strategy and takes $O(n \log(n))$ operation compared to the $O(n^2)$ brute force method

Fast Fourier Transform

- The Fourier Transform provides a different "view" of a sequence such as a signals, images, etc.

$$\tilde{f}(\boldsymbol{k}) = \sum_{x_1=0}^{n-1} \cdots \sum_{x_d=0}^{n-1} f(\boldsymbol{x})\, \mathrm{e}^{2\pi \mathrm{i} \boldsymbol{k}\cdot\boldsymbol{x}/n}$$

- The **Fast Fourier Transform** was an algorithm divised by John Tukey and James Cooley in 1965 to compute the Fourier Transform quickly

- Gauss had used exactly this idea in a paper in 1805 to save himself work computing a Fourier transform by hand

- It is based on a divide-and-conquer strategy and takes $O(n\log(n))$ operation compared to the $O(n^2)$ brute force method

Fast Fourier Transform

- The Fourier Transform provides a different "view" of a sequence such as a signals, images, etc.

$$\tilde{f}(\boldsymbol{k}) = \sum_{x_1=0}^{n-1} \cdots \sum_{x_d=0}^{n-1} f(\boldsymbol{x})\, \mathrm{e}^{2\pi \mathrm{i} \boldsymbol{k}\cdot\boldsymbol{x}/n}$$

- The **Fast Fourier Transform** was an algorithm divised by John Tukey and James Cooley in 1965 to compute the Fourier Transform quickly

- Gauss had used exactly this idea in a paper in 1805 to save himself work computing a Fourier transform by hand

- It is based on a divide-and-conquer strategy and takes $O(n\log(n))$ operation compared to the $O(n^2)$ brute force method

Fast Fourier Transform

- The Fourier Transform provides a different "view" of a sequence such as a signals, images, etc.

$$\tilde{f}(\boldsymbol{k}) = \sum_{x_1=0}^{n-1} \cdots \sum_{x_d=0}^{n-1} f(\boldsymbol{x})\, \mathrm{e}^{2\pi \mathrm{i} \boldsymbol{k}\cdot\boldsymbol{x}/n}$$

- The **Fast Fourier Transform** was an algorithm divised by John Tukey and James Cooley in 1965 to compute the Fourier Transform quickly

- Gauss had used exactly this idea in a paper in 1805 to save himself work computing a Fourier transform by hand

- It is based on a divide-and-conquer strategy and takes $O(n\log(n))$ operation compared to the $O(n^2)$ brute force method

Applications of FFT

- The application of FFT are enormous
- It lies at the heart of digital signal processing
- It is frequently used in image analysis
- It is a type of wavelet transform used in JPEG
- It is even used in fast multiplication of very large integers—with important applications in cryptography

Applications of FFT

- The application of FFT are enormous
- It lies at the heart of digital signal processing
- It is frequently used in image analysis
- It is a type of wavelet transform used in JPEG
- It is even used in fast multiplication of very large integers—with important applications in cryptography

Applications of FFT

- The application of FFT are enormous
- It lies at the heart of digital signal processing
- It is frequently used in image analysis
- It is a type of wavelet transform used in JPEG
- It is even used in fast multiplication of very large integers—with important applications in cryptography

Applications of FFT

- The application of FFT are enormous
- It lies at the heart of digital signal processing
- It is frequently used in image analysis
- It is a type of wavelet transform used in JPEG
- It is even used in fast multiplication of very large integers—with important applications in cryptography

Applications of FFT

- The application of FFT are enormous
- It lies at the heart of digital signal processing
- It is frequently used in image analysis
- It is a type of wavelet transform used in JPEG
- It is even used in fast multiplication of very large integers—with important applications in cryptography

Greedy Algorithms

- The **greedy strategy** is to build a solution to a problem by choosing the best available option
- If you are lucky this will give an optimal solution
- Example of optimal algorithms based on the greedy strategy include
  - ⋆ Constructing Huffman trees
  - ⋆ Prim's algorithms
  - ⋆ Kruskal's algorithm
  - ⋆ Dijkstra's algorithm
- Often uses priority queues

Greedy Algorithms

- The **greedy strategy** is to build a solution to a problem by choosing the best available option
- If you are lucky this will give an optimal solution
- Example of optimal algorithms based on the greedy strategy include
  - ⋆ Constructing Huffman trees
  - ⋆ Prim's algorithms
  - ⋆ Kruskal's algorithm
  - ⋆ Dijkstra's algorithm
- Often uses priority queues

Greedy Algorithms

- The **greedy strategy** is to build a solution to a problem by choosing the best available option
- If you are lucky this will give an optimal solution
- Example of optimal algorithms based on the greedy strategy include
  - ⋆ Constructing Huffman trees
  - ⋆ Prim's algorithms
  - ⋆ Kruskal's algorithm
  - ⋆ Dijkstra's algorithm
- Often uses priority queues

Greedy Algorithms

- The **greedy strategy** is to build a solution to a problem by choosing the best available option
- If you are lucky this will give an optimal solution
- Example of optimal algorithms based on the greedy strategy include
  - ⋆ Constructing Huffman trees
  - ⋆ Prim's algorithms
  - ⋆ Kruskal's algorithm
  - ⋆ Dijkstra's algorithm
- Often uses priority queues

Non-optimal Greedy Algorithms

- Greedy algorithms can also be used to solve optimisation problems such as the travelling salesperson problem
- In the TSP we can start at some city and move to the nearest as-yet-unvisited city
- The algorithm is guaranteed to find a solution no longer than $0.5(\lfloor \log_2(n) \rfloor + 1)$ times the optimal tour length
- It usually does substantially better, but it is very unlikely to find the optimal for very long tours
- It is more the exception rather than the rule that Greedy algorithms find an optimal solutions

Non-optimal Greedy Algorithms

- Greedy algorithms can also be used to solve optimisation problems such as the travelling salesperson problem
- In the TSP we can start at some city and move to the nearest as-yet-unvisited city
- The algorithm is guaranteed to find a solution no longer than $0.5(\lfloor \log_2(n) \rfloor + 1)$ times the optimal tour length
- It usually does substantially better, but it is very unlikely to find the optimal for very long tours
- It is more the exception rather than the rule that Greedy algorithms find an optimal solutions

Non-optimal Greedy Algorithms

- Greedy algorithms can also be used to solve optimisation problems such as the travelling salesperson problem
- In the TSP we can start at some city and move to the nearest as-yet-unvisited city
- The algorithm is guaranteed to find a solution no longer than $0.5(\lfloor \log_2(n) \rfloor + 1)$ times the optimal tour length
- It usually does substantially better, but it is very unlikely to find the optimal for very long tours
- It is more the exception rather than the rule that Greedy algorithms find an optimal solutions

Non-optimal Greedy Algorithms

- Greedy algorithms can also be used to solve optimisation problems such as the travelling salesperson problem
- In the TSP we can start at some city and move to the nearest as-yet-unvisited city
- The algorithm is guaranteed to find a solution no longer than $0.5(\lfloor \log_2(n) \rfloor + 1)$ times the optimal tour length
- It usually does substantially better, but it is very unlikely to find the optimal for very long tours
- It is more the exception rather than the rule that Greedy algorithms find an optimal solutions

Non-optimal Greedy Algorithms

- Greedy algorithms can also be used to solve optimisation problems such as the travelling salesperson problem
- In the TSP we can start at some city and move to the nearest as-yet-unvisited city
- The algorithm is guaranteed to find a solution no longer than $0.5(\lfloor \log_2(n) \rfloor + 1)$ times the optimal tour length
- It usually does substantially better, but it is very unlikely to find the optimal for very long tours
- It is more the exception rather than the rule that Greedy algorithms find an optimal solutions

Dynamic Programming

- Dynamic programming can be used for solving many problems
- It requires some (partial) ordering so that you can assign costs to partial solutions from previous solutions
- Requires imagination to think how to do this
- Used in inexact matching, shortest paths, line breaks, etc.

Dynamic Programming

- Dynamic programming can be used for solving many problems
- It requires some (partial) ordering so that you can assign costs to partial solutions from previous solutions
- Requires imagination to think how to do this
- Used in inexact matching, shortest paths, line breaks, etc.

Dynamic Programming

- Dynamic programming can be used for solving many problems
- It requires some (partial) ordering so that you can assign costs to partial solutions from previous solutions
- Requires imagination to think how to do this
- Used in inexact matching, shortest paths, line breaks, etc.

Dynamic Programming

- Dynamic programming can be used for solving many problems
- It requires some (partial) ordering so that you can assign costs to partial solutions from previous solutions
- Requires imagination to think how to do this
- Used in inexact matching, shortest paths, line breaks, etc.

Uses of Dynamic Programming

- Recurrent solutions to lattice models for protein-DNA binding
- Backward induction as a solution method for finite-horizon discrete-time dynamic optimization problems
- Method of undetermined coefficients can be used to solve the Bellman equation in infinite-horizon, discrete-time, discounted, time-invariant dynamic optimization problems
- Many string algorithms including longest common subsequence, longest increasing subsequence, longest common substring, Levenshtein distance (edit distance)
- Many algorithmic problems on graphs can be solved efficiently for graphs of bounded treewidth or bounded clique-width by using dynamic programming on a tree decomposition of the graph.
- The Cocke–Younger–Kasami (CYK) algorithm which determines whether and how a given string can be generated by a given context-free grammar
- Knuth's word wrapping algorithm that minimizes raggedness when word wrapping text

- The use of transposition tables and refutation tables in computer chess
- The Viterbi algorithm (used for hidden Markov models)
- The Earley algorithm (a type of chart parser)
- The Needleman–Wunsch and other algorithms used in bioinformatics, including sequence alignment, structural alignment, RNA structure prediction
- Floyd's all-pairs shortest path algorithm
- Optimizing the order for chain matrix multiplication
- Pseudo-polynomial time algorithms for the subset sum and knapsack and partition problems
- The dynamic time warping algorithm for computing the global distance between two time series
- The Selinger (a.k.a. System R) algorithm for relational database query optimization
- De Boor algorithm for evaluating B-spline curves
- Duckworth–Lewis method for resolving the problem when games of cricket are interrupted

- The value iteration method for solving Markov decision processes
- Some graphic image edge following selection methods such as the "magnet" selection tool in Photoshop
- Some methods for solving interval scheduling problems
- Some methods for solving word wrap problems
- Some methods for solving the travelling salesman problem, either exactly (in exponential time) or approximately (e.g. via the bitonic tour)
- Recursive least squares method
- Beat tracking in music information retrieval
- Adaptive-critic training strategy for artificial neural networks
- Stereo algorithms for solving the correspondence problem used in stereo vision
- Seam carving (content aware image resizing)
- The Bellman–Ford algorithm for finding the shortest distance in a graph
- Some approximate solution methods for the linear search problem
- Kadane's algorithm for the maximum subarray problem

Linear Programming

- Look out for problems with linear objectives or where a linearisation approximation is acceptable
- These can often be turned in linear programs which can be solved efficiently
- The constraints have to be linear and the variables take on continuous values
- Sometimes by being careful we can force integer solutions (see linear assignment in last lecture)
- Applications in planning, but also in many areas of optimisation

Linear Programming

- Look out for problems with linear objectives or where a linearisation approximation is acceptable
- These can often be turned in linear programs which can be solved efficiently
- The constraints have to be linear and the variables take on continuous values
- Sometimes by being careful we can force integer solutions (see linear assignment in last lecture)
- Applications in planning, but also in many areas of optimisation

Linear Programming

- Look out for problems with linear objectives or where a linearisation approximation is acceptable
- These can often be turned in linear programs which can be solved efficiently
- The constraints have to be linear and the variables take on continuous values
- Sometimes by being careful we can force integer solutions (see linear assignment in last lecture)
- Applications in planning, but also in many areas of optimisation

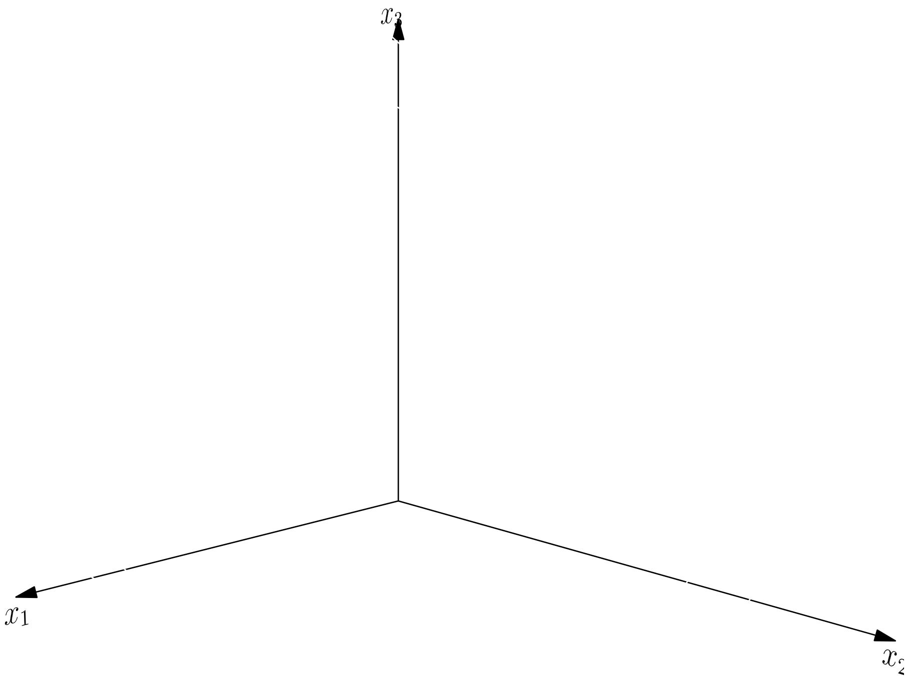
Linear Programming

- Look out for problems with linear objectives or where a linearisation approximation is acceptable
- These can often be turned in linear programs which can be solved efficiently
- The constraints have to be linear and the variables take on continuous values
- Sometimes by being careful we can force integer solutions (see linear assignment in last lecture)
- Applications in planning, but also in many areas of optimisation

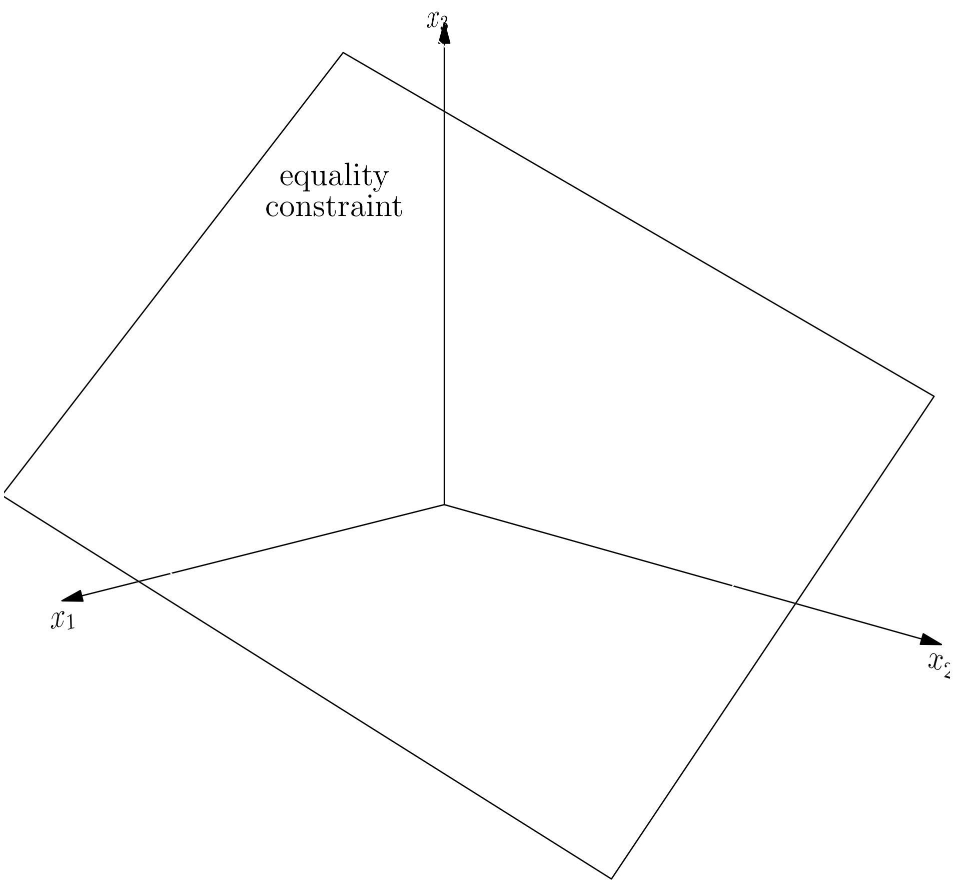
Linear Programming

- Look out for problems with linear objectives or where a linearisation approximation is acceptable
- These can often be turned in linear programs which can be solved efficiently
- The constraints have to be linear and the variables take on continuous values
- Sometimes by being careful we can force integer solutions (see linear assignment in last lecture)
- Applications in planning, but also in many areas of optimisation

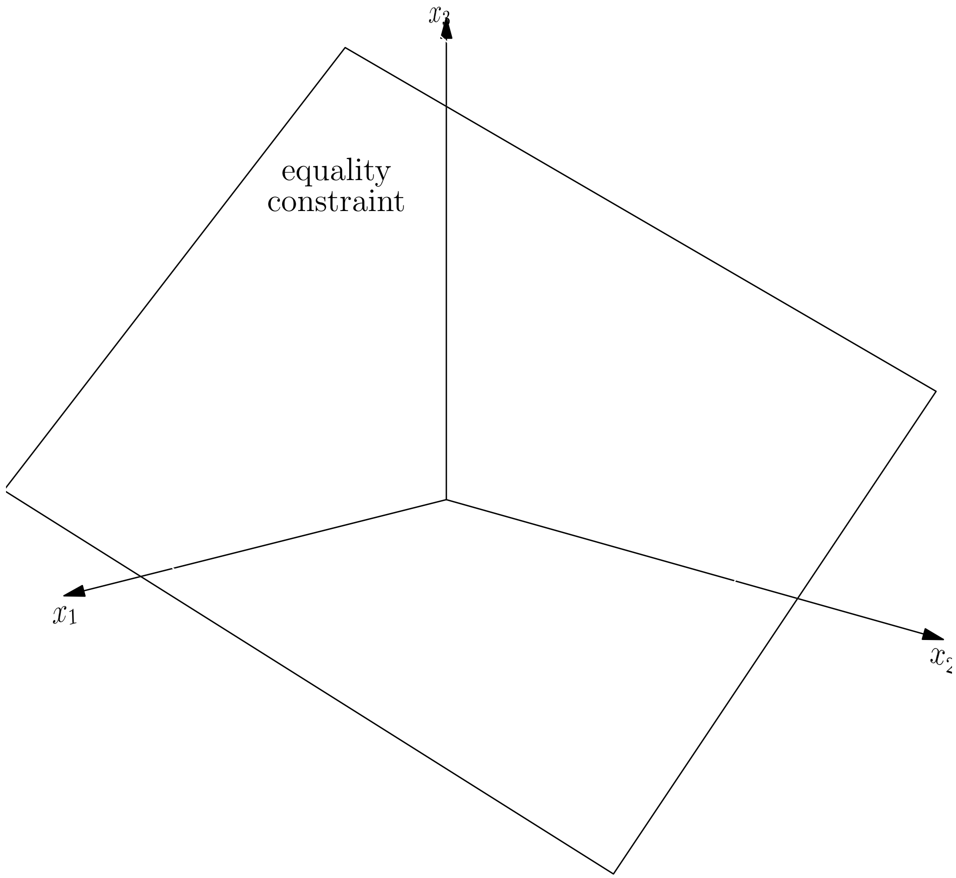
Solving Linear Programming



Solving Linear Programming

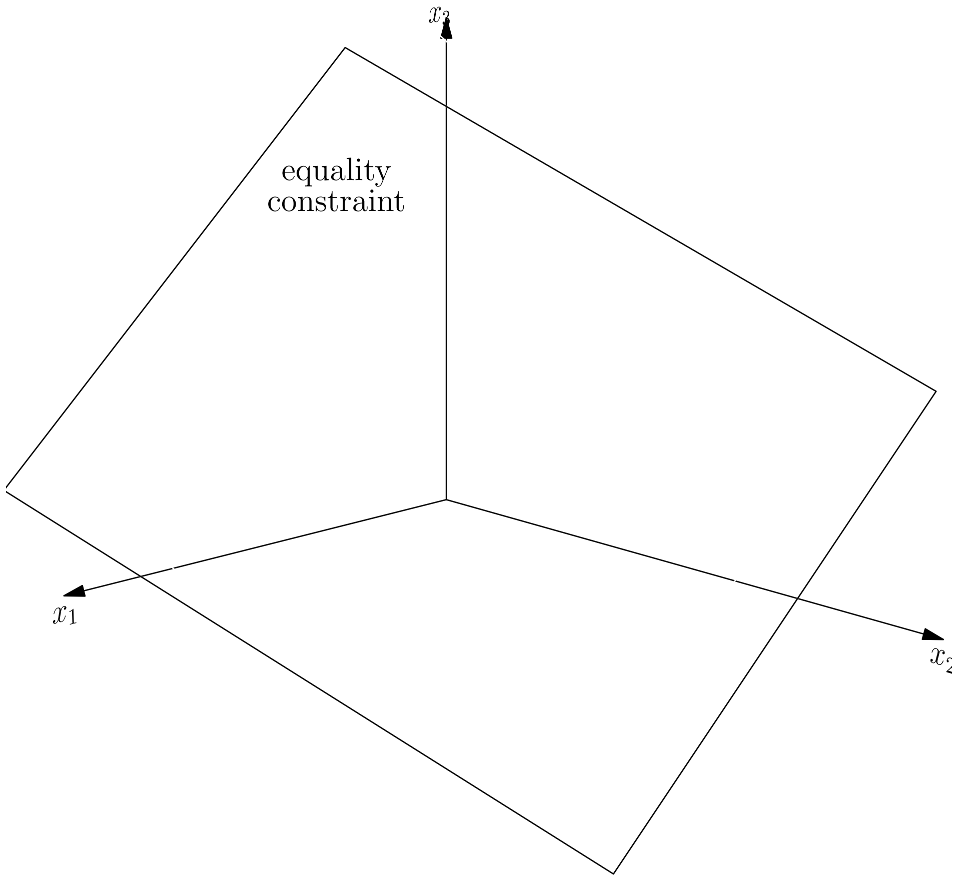


Solving Linear Programming



- The basic feasible points for LP problems with n variables and m constraints have at least $n - m$ zero variables

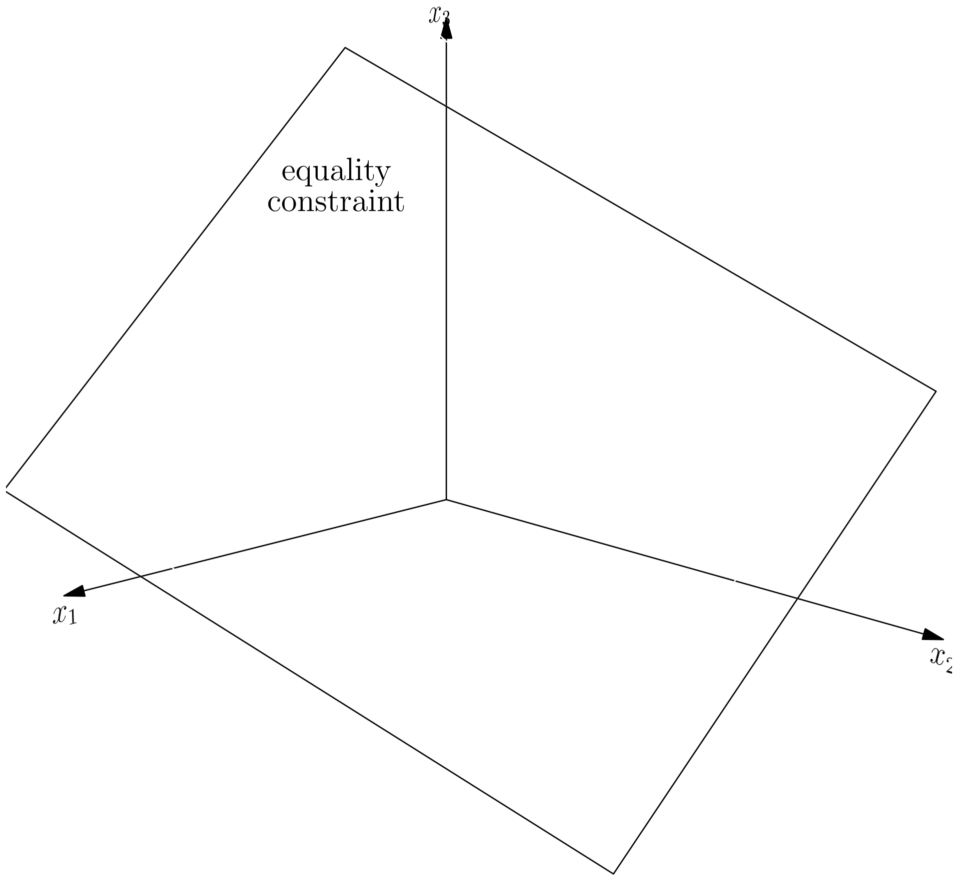
Solving Linear Programming



- The basic feasible points for LP problems with n variables and m constraints have at least $n - m$ zero variables

- Typical number of basic feasible solutions is $\binom{n}{m} \geq \left(\frac{n}{m}\right)^m$

Solving Linear Programming



- The basic feasible points for LP problems with n variables and m constraints have at least $n-m$ zero variables
- Typical number of basic feasible solutions is $\binom{n}{m} \geq \left(\frac{n}{m}\right)^m$
- Simplex algorithm organises iterative search for global solutions

Backtracking

- Backtracking is in many ways a brute force method
- It is just a way of exploring a search space
- However, when we solving problems with constraints then we can vastly reduce the number of solutions that we visit
- Used in many game playing, planning, verification, puzzle solving situations
- For optimisation we can use branch and bound which is backtracking using the best solution as a constraint

Backtracking

- Backtracking is in many ways a brute force method
- It is just a way of exploring a search space
- However, when we solving problems with constraints then we can vastly reduce the number of solutions that we visit
- Used in many game playing, planning, verification, puzzle solving situations
- For optimisation we can use branch and bound which is backtracking using the best solution as a constraint

Backtracking

- Backtracking is in many ways a brute force method
- It is just a way of exploring a search space
- However, when we solving problems with constraints then we can vastly reduce the number of solutions that we visit
- Used in many game playing, planning, verification, puzzle solving situations
- For optimisation we can use branch and bound which is backtracking using the best solution as a constraint

Backtracking

- Backtracking is in many ways a brute force method
- It is just a way of exploring a search space
- However, when we solving problems with constraints then we can vastly reduce the number of solutions that we visit
- Used in many game playing, planning, verification, puzzle solving situations
- For optimisation we can use branch and bound which is backtracking using the best solution as a constraint

Backtracking

- Backtracking is in many ways a brute force method
- It is just a way of exploring a search space
- However, when we solving problems with constraints then we can vastly reduce the number of solutions that we visit
- Used in many game playing, planning, verification, puzzle solving situations
- For optimisation we can use branch and bound which is backtracking using the best solution as a constraint

Computer Chess

- Computer chess algorithms explore the search tree of possible moves using backtracking with pruning
- Although they cannot look at all possible moves, they can look deep enough to play good chess
- In 1968 International Master David Levy bet that he would not be beaten by a computer in the next decade, a bet he won
- He was beaten in 1989 by Deep Thought
- In 1997 Deep Blue beat Gary Kasparov the reigning world champion
- Modern chess engines such as Deep Rybka and Houdini have a chess rating of around 3200 (c.f. Magnus Carsen's 2872—the top human player)

Computer Chess

- Computer chess algorithms explore the search tree of possible moves using backtracking with pruning
- Although they cannot look at all possible moves, they can look deep enough to play good chess
- In 1968 International Master David Levy bet that he would not be beaten by a computer in the next decade, a bet he won
- He was beaten in 1989 by Deep Thought
- In 1997 Deep Blue beat Gary Kasparov the reigning world champion
- Modern chess engines such as Deep Rybka and Houdini have a chess rating of around 3200 (c.f. Magnus Carsen's 2872—the top human player)

Computer Chess

- Computer chess algorithms explore the search tree of possible moves using backtracking with pruning
- Although they cannot look at all possible moves, they can look deep enough to play good chess
- In 1968 International Master David Levy bet that he would not be beaten by a computer in the next decade, a bet he won
- He was beaten in 1989 by Deep Thought
- In 1997 Deep Blue beat Gary Kasparov the reigning world champion
- Modern chess engines such as Deep Rybka and Houdini have a chess rating of around 3200 (c.f. Magnus Carsen's 2872—the top human player)

Computer Chess

- Computer chess algorithms explore the search tree of possible moves using backtracking with pruning
- Although they cannot look at all possible moves, they can look deep enough to play good chess
- In 1968 International Master David Levy bet that he would not be beaten by a computer in the next decade, a bet he won
- He was beaten in 1989 by Deep Thought
- In 1997 Deep Blue beat Gary Kasparov the reigning world champion
- Modern chess engines such as Deep Rybka and Houdini have a chess rating of around 3200 (c.f. Magnus Carsen's 2872—the top human player)

Computer Chess

- Computer chess algorithms explore the search tree of possible moves using backtracking with pruning
- Although they cannot look at all possible moves, they can look deep enough to play good chess
- In 1968 International Master David Levy bet that he would not be beaten by a computer in the next decade, a bet he won
- He was beaten in 1989 by Deep Thought
- In 1997 Deep Blue beat Gary Kasparov the reigning world champion
- Modern chess engines such as Deep Rybka and Houdini have a chess rating of around 3200 (c.f. Magnus Carsen's 2872—the top human player)

Computer Chess

- Computer chess algorithms explore the search tree of possible moves using backtracking with pruning
- Although they cannot look at all possible moves, they can look deep enough to play good chess
- In 1968 International Master David Levy bet that he would not be beaten by a computer in the next decade, a bet he won
- He was beaten in 1989 by Deep Thought
- In 1997 Deep Blue beat Gary Kasparov the reigning world champion
- Modern chess engines such as Deep Rybka and Houdini have a chess rating of around 3200 (c.f. Magnus Carsen's 2872—the top human player)

Heuristic Search

- When all else fails we have to settle for finding a good solution, not the best
- In fact, because so many problems that we are interested in turn out to be NP-hard this is quite common
- There are many strategies
  - ⋆ Neighbourhood search (hill-climbing, descent)
  - ⋆ Simulated annealing
  - ⋆ Evolutionary algorithms, etc.

Heuristic Search

- When all else fails we have to settle for finding a good solution, not the best
- In fact, because so many problems that we are interested in turn out to be NP-hard this is quite common
- There are many strategies
  - ⋆ Neighbourhood search (hill-climbing, descent)
  - ⋆ Simulated annealing
  - ⋆ Evolutionary algorithms, etc.

Heuristic Search

- When all else fails we have to settle for finding a good solution, not the best
- In fact, because so many problems that we are interested in turn out to be NP-hard this is quite common
- There are many strategies
  - ⋆ Neighbourhood search (hill-climbing, descent)
  - ⋆ Simulated annealing
  - ⋆ Evolutionary algorithms, etc.

Lessons

- Many applications bring up interesting programming challenges
- Some are dealt with by using sensible data structures and common algorithms
- However, often you face a new challenge requiring thought
- Thinking in terms of strategies and having a feel for time complexity is part of armoury of a professional programmer

Lessons

- Many applications bring up interesting programming challenges
- Some are dealt with by using sensible data structures and common algorithms
- However, often you face a new challenge requiring thought
- Thinking in terms of strategies and having a feel for time complexity is part of armoury of a professional programmer

Lessons

- Many applications bring up interesting programming challenges
- Some are dealt with by using sensible data structures and common algorithms
- However, often you face a new challenge requiring thought
- Thinking in terms of strategies and having a feel for time complexity is part of armoury of a professional programmer

Lessons

- Many applications bring up interesting programming challenges
- Some are dealt with by using sensible data structures and common algorithms
- However, often you face a new challenge requiring thought
- Thinking in terms of strategies and having a feel for time complexity is part of armoury of a professional programmer