دانشگاه صنعتی امیرکبیر
(پلی تکنیک تهران)

گزارش پروژه پیاده‌سازی اول

درس معماری افزاره‌های شبکه

استاد درس : دکتر صبایی

پاییز ۱۴۰۲

مهلا شریفی

۹۸۳۱۰۳۵

فهرست

ساخت درخت

استفاده

می‌توانید prefixها را دستی وارد کنید و یا از دستور read file برای این منظور استفاده کنید.

کد

```
208         elif command == 'insert':
209             try:
210 
211                 input_data = input("Enter prefix, length, next_hop: ").split()
212                 prefix = input_data[0]
213                 length, next_hop = map(int, input_data[1:])
214 
215                 if(length == 0):
216                     root.next_hop = next_hop
217 
218                 else:
219                     insert(root, prefix, length, next_hop, stride, prefix_base)
```

- خط ۲۱۶ : اگر length برابر 0 باشد، به این معناست که گره مربوطه ریشه است.

```python
def insert(root, prefix, length, next_hop, stride, base):
    current_node = root
    integer_prefix = int(prefix[:length],base)
    binary_number = bin(integer_prefix)[2:]

    rjusted = binary_number.rjust(length, '0')
    binary_prefix = rjusted.ljust(32, '0')

    for i in range(0, length, stride):
        bit_pattern = binary_prefix[i:i+stride]

        if i + stride > length:
            curr_pattern = str(binary_prefix[i:length])
            ex = len(curr_pattern)
            remaining_bits = stride - (length - i)
            # Calculate the number of combinations for the remaining bits
            num_combinations = 2 ** (remaining_bits)
            # Generate all combinations for the remaining bits and create nodes
            for j in range(num_combinations):
                # Generate the binary representation for the current combination
                combination = bin(j)[2:].zfill(remaining_bits)

                pattern = curr_pattern + combination
                if pattern not in current_node.children:
                    current_node.children[pattern] = Node(next_hop=next_hop, length= length)
                else:
                    if current_node.children[pattern].length < length:
                        current_node.children[pattern].next_hop = next_hop
                        current_node.children[pattern].length = length

        else:

            # If there is no child for the bit pattern, create a new node
            if bit_pattern not in current_node.children:
                current_node.children[bit_pattern] = Node()
            current_node = current_node.children[bit_pattern]
            # If we have reached the end of the prefix, set the next hop

        if (i + stride == length ):
                current_node.next_hop = next_hop
                current_node.length = length
```

- خطوط ۵۳ تا ۵۷: عدد وارد شده را به یک رشته بیت باینری ۳۲ بیتی تبدیل می‌کند. یعنی 001 تبدیل می‌شود به 001000…
- خط ۶۰: در هر مرحله به اندازه stride جلو می‌رود.
- خط ۶۲: زمانی برقرار است که تعداد بیت‌ها prefix بر stride بخش‌پذیر نباشد.(مثلا stride ۲ باشد ولی length برابر ۳ باشد.) در این صورت در آخرین دور وارد این حلقه می‌شود.
- خطوط ۷۱ تا ۷۹ : فرضا اگر داشته باشیم prefix = 010001 و stride =4 در دومین iteration حلقه ۴ حالت مختلف که توسط این prefix ساپورت می‌شوند، (0100, 0101, 0110, 0111) تولید می شود و همگی next_hob مربوط به prefix فعلی را اتخاذ می‌کنند. (اگر گره وجود داشته باشد و

length آن بیشتر از length گره‌ی فعلی باشد باشد، به دلیل قاعده longest prefix matching،
next_hopاشان عوض نمی‌شود.)

- خطوط ۸۹ تا ۹۱: برای دور آخر گره‌هایی است که طول prefix بر stride بخش پذیر است و باقی مانده ندارد.

```
33  ##This function will sort inputs for more efficent creation of trie
34  def pre_process(inputs):
35      # ipv4 is 32 bit. hence length is between 0 to 32
36      length_dict = {length: [] for length in range(0, 33)}
37
38      for prefix, length, next_hop in inputs:
39          length_dict[length].append([prefix, length, next_hop])
40
41      for length in length_dict:
42          length_dict[length].sort(key=lambda x: x[0])
43
44      return length_dict
```

- برای خواندن از فایل یک مرحله پیش‌پردازش به منظور افزایش سرعت انجام می‌شود. به این شکل که ابتدا ورودی‌ها برحسب length مرتبط می‌شوند و سپس عملیات ساخت درخت آغاز می‌شود.

تنظیمات

استفاده

با استفاده از دستور set configuration، sride و مبنای ورودی را می‌توانید تعیین کنید. مثلا مبنای ورودی در مثال دستور کار ۲ و در فایل ضمیمه ۱۶ است.

```
Enter command (Read File, Insert, Print, visualize, Lookup, Set Configuration, Finish): set configuration
Enter new stride (1, 2, 4, 8, etc.): 8
Stride set to 8
Enter the base for prefix (binary, decimal, hexadecimal): binary
Base for prefix set to binary
```

کد

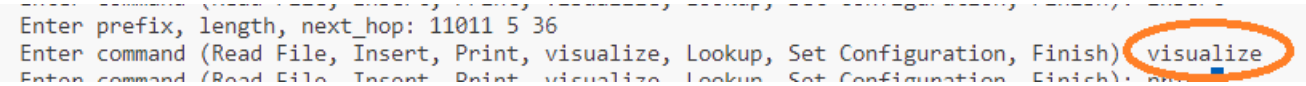
```
elif command == 'set configuration':
    try:
        new_stride = int(input("Enter new stride (1, 2, 4, 8, etc.): "))
        if new_stride > 0:
            stride = new_stride
            print(f"Stride set to {stride}")
        else:
            print("Invalid stride value. Please enter a positive integer.")

        # Additional code to set the base for prefix input
        new_base = input("Enter the base for prefix (binary, decimal, hexadecimal): ").strip().lower()
        if new_base in ["binary", "decimal", "hexadecimal"]:
            if new_base == "binary":
                base = 2
            elif new_base == "decimal":
                base = 10
            elif new_base == "hexadecimal":
                base = 16
            print(f"Base for prefix set to {new_base}")
        else:
            print("Invalid base. Please enter 'binary', 'decimal', or 'hexadecimal'.")
    except ValueError:
        print("Invalid input. Please enter an integer for stride.")
```

- دو متغییر base و stride به عنوان ورودی به تابع insert پاس داده می‌شوند.

مشاهده‌ی نتیجه (بصری)

استفاده

پس از ساخت درخت (با دستور insert و یا واردن کردن نام فایل) با دستور visualize می‌توانید نتیجه MultiTrie ساخته شده را در قالب یک فایل png مشاهده کنید.



کد

برای مصورسازی نتیجه از ابزار Graphviz استفاده شده است. از طریق این لینک می‌توانید ابزار را نصب کنید.

```python
def visualize_trie(node, graph=None, parent_name=None, edge_label=''):
    if graph is None:
        graph = Digraph(comment='Trie')

    if parent_name is None:
        parent_name = 'root'
        root_label = str(node.next_hop) if node.next_hop is not None else ''
        graph.node(parent_name, label=root_label)

    sorted_children_keys = sorted(node.children.keys())

    # Iterate over the children of the current node
    for bit_pattern in sorted_children_keys:
        child_node = node.children[bit_pattern]
        # The name of the node in the graph is a combination of its parent name and its bit pattern
        node_name = f'{parent_name}-{bit_pattern}'

        # If the node has a next_hop value, use it as the label, otherwise leave it blank
        if child_node.next_hop is not None:
            graph.node(node_name, label=str(child_node.next_hop))
        else:
            graph.node(node_name, label='')

        # The label for the edge is the bit pattern leading to the current node
        graph.edge(parent_name, node_name, label=bit_pattern)

        # Recursively call visualize_trie to add the children of the current node to the graph
        visualize_trie(child_node, graph, node_name, bit_pattern)
```

- برچسب هر گره در گراف next_hob آن است.
- برچسب یال بین یک پدر و فرزند، کلید آن فرزند در دیکشنری بچه‌های پدر است.
- مرتب‌سازی فرزندان هر گره (اینکه فرضا فرزند "0" سمت چپ فرزند "1" قرار بگیرد) در هنگام چاپ خروجی انجام می‌شود. به این شکل از پیچیدگی جابجایی در هنگام ساخت درخت کاسته می‌شود و نیاز نیست ورودی‌ها به ترتیب وارد شوند.)

مشاهده‌ی نتیجه (چاپ)

استفاده

پس از وارد کردن دستور print نیاز است تا نام فایلی که قرار است درخت در آن چاپ شود را وارد کنید.

```
Enter prefix, length, next_hop: 11011 5 56
Enter command (Read File, Insert, Print, visualize, Lookup, Set Configuration, Finish): print
Enter the file name to print the trie: example-stride8.txt
Trie has been printed to example-stride8.txt
```

کد

```python
def print_trie(node, file, bit_pattern='', indent=0):
    # Base case: if the current node has a next_hop, write it to the file
    if node.next_hop is not None:
        file.write(' ' * indent + f"Bit pattern: {bit_pattern} -> Next hop: {node.next_hop}\n")

    sorted_children_keys = sorted(node.children.keys())

    # Increase the indentation for child nodes
    new_indent = indent + 4

    # Recursively call print_trie for each child
    for child_bit_pattern in sorted_children_keys:
        child_node = node.children[child_bit_pattern]

        full_bit_pattern = bit_pattern + child_bit_pattern
        file.write(' ' * indent + f"Child bit pattern: {child_bit_pattern}\n")
        print_trie(child_node, file, full_bit_pattern, new_indent)

```

- خط ۱۵۲ مشابه دلیل توضیح داده شده در نمایش نتیجه به شکل بصری است.

جستجو در درخت

استفاده

```
Current memory usage: 0.02 MB; Peak: 0.02 MB
Enter command (Read File, Insert, Print, visualize, Lookup, Set Configuration, Finish) lookup
Enter IP address to lookup: 01011111000011110000111100001111
Next hop for 01011111000011110000111100001111: 25
```

مجددا ip ورودی می‌تواند بر هر مبنایی وارد شود (ip باید یک عدد ۳۲ بیتی باشد.)

کد

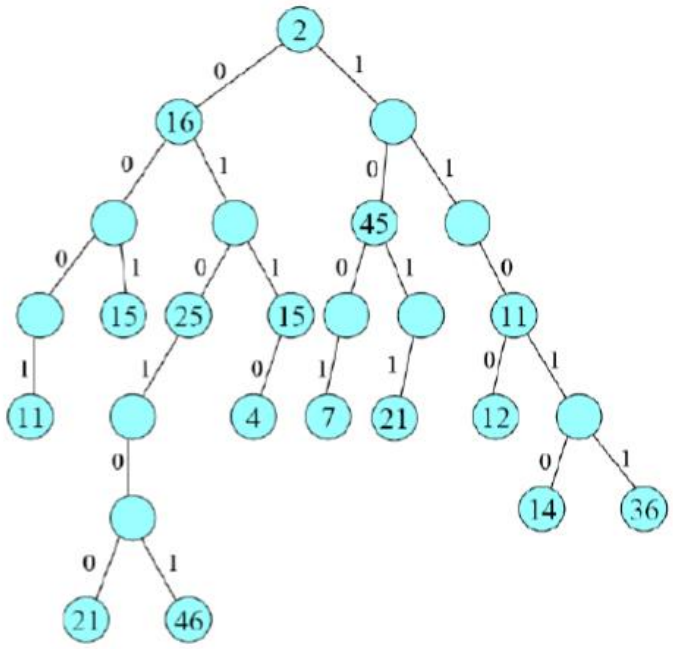
```python
def lookup(root, ip_address, stride, base):
    binary_ip = bin(int(ip_address, base))[2:].zfill(32)
    current_node = root
    best_match = root.next_hop

    for i in range(0, 32, stride):
        bit_pattern = binary_ip[i:i + stride]

        # Check if the bit pattern matches any child of the current node
        if bit_pattern in current_node.children:
            current_node = current_node.children[bit_pattern]

            if current_node.next_hop is not None:
                best_match = current_node.next_hop
        else:
            break

    return best_match
```

مثال دستورکار

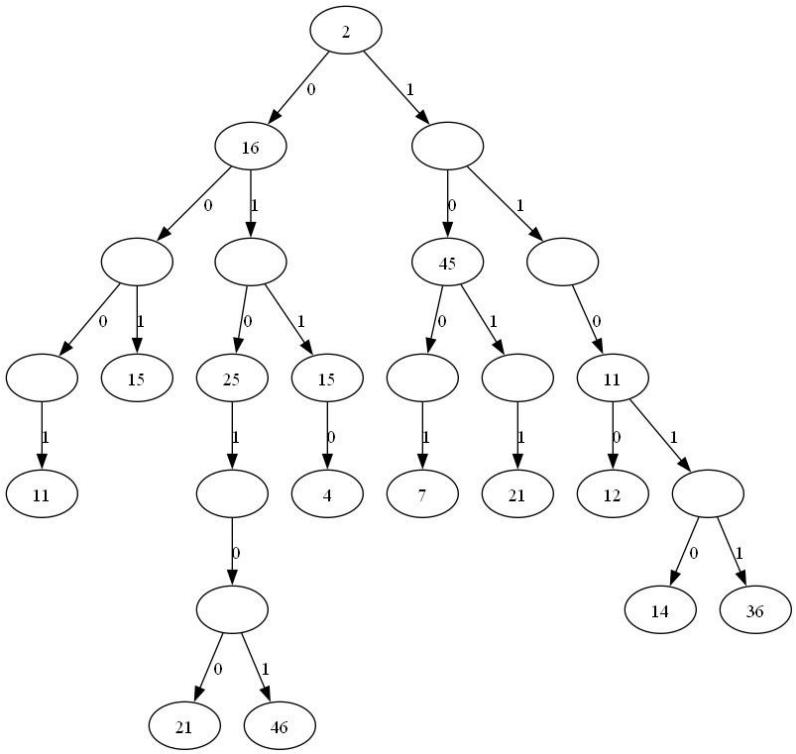
برای سادگی دستورات مورد نیاز برای ساخت درخت در فایل example.txt قرار گرفته است. (برای تست صحت کد می‌توانید از آن استفاده کنید یا دستورات را دستی وارد کنید.)

در هر مرحله می‌توان درخت را با دستورات print و visualize نمایش داد.

طبق خواسته‌ی دستورکار درخت زیر را با strideهای مختلف می‌سازیم.

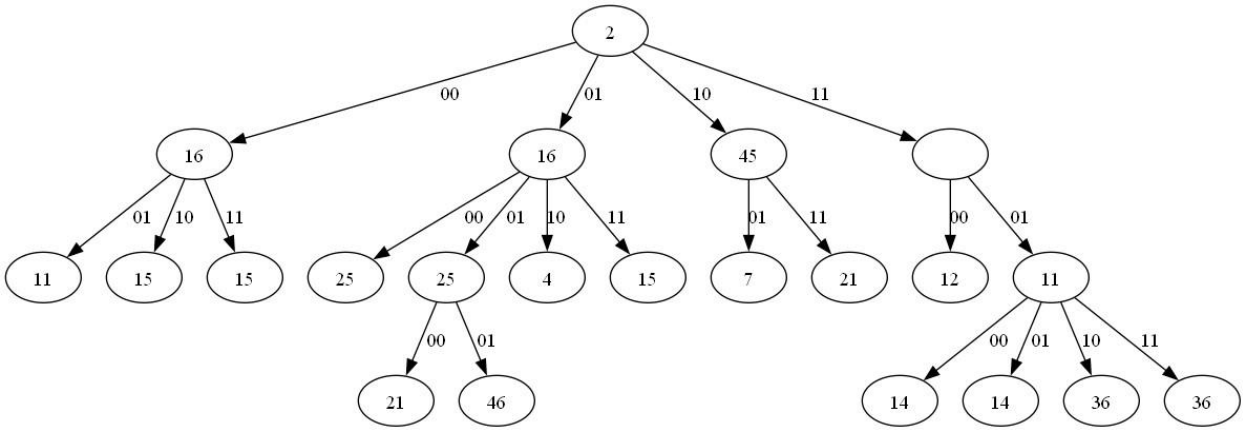

2
16
45
15
25
15
11
11
4
7
21
12
14
36
21
46

Stride = 1


2
16
45
15
25
15
11
11
4
7
21
12
14
36
21
46

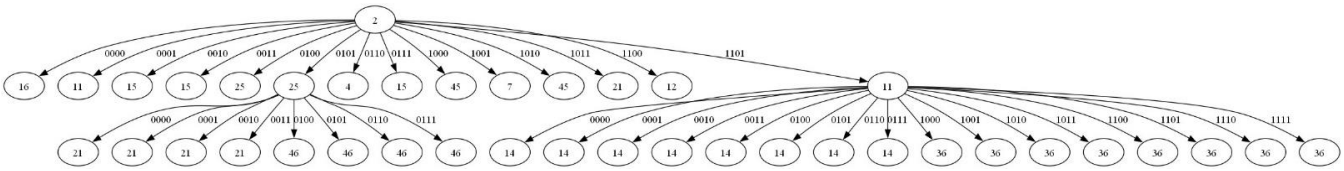
```
Bit pattern:  -> Next hop: 2
Child bit pattern: 0
    Bit pattern: 0 -> Next hop: 16
    Child bit pattern: 0
        Child bit pattern: 0
            Child bit pattern: 1
                Bit pattern: 0001 -> Next hop: 11
        Child bit pattern: 1
            Bit pattern: 001 -> Next hop: 15
    Child bit pattern: 1
        Child bit pattern: 0
            Bit pattern: 010 -> Next hop: 25
            Child bit pattern: 1
                Child bit pattern: 0
                    Child bit pattern: 0
                        Bit pattern: 010100 -> Next hop: 21
                    Child bit pattern: 1
                        Bit pattern: 010101 -> Next hop: 46
        Child bit pattern: 1
            Bit pattern: 011 -> Next hop: 15
            Child bit pattern: 0
                Bit pattern: 0110 -> Next hop: 4
Child bit pattern: 1
    Child bit pattern: 0
        Bit pattern: 10 -> Next hop: 45
        Child bit pattern: 0
            Child bit pattern: 1
                Bit pattern: 1001 -> Next hop: 7
        Child bit pattern: 1
            Child bit pattern: 1
                Bit pattern: 1011 -> Next hop: 21
    Child bit pattern: 1
        Child bit pattern: 0
            Bit pattern: 110 -> Next hop: 11
            Child bit pattern: 0
                Bit pattern: 1100 -> Next hop: 12
            Child bit pattern: 1
                Child bit pattern: 0
                    Bit pattern: 11010 -> Next hop: 14
                Child bit pattern: 1
                    Bit pattern: 11011 -> Next hop: 36
```

Stride = 2

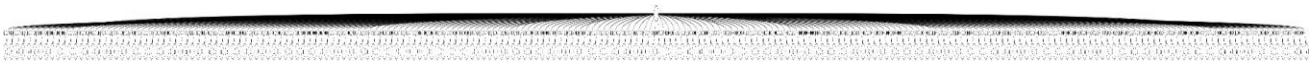



```
Bit pattern:  -> Next hop: 2
Child bit pattern: 00
    Bit pattern: 00 -> Next hop: 16
    Child bit pattern: 01
        Bit pattern: 0001 -> Next hop: 11
    Child bit pattern: 10
        Bit pattern: 0010 -> Next hop: 15
    Child bit pattern: 11
        Bit pattern: 0011 -> Next hop: 15
Child bit pattern: 01
    Bit pattern: 01 -> Next hop: 16
    Child bit pattern: 00
        Bit pattern: 0100 -> Next hop: 25
    Child bit pattern: 01
        Bit pattern: 0101 -> Next hop: 25
        Child bit pattern: 00
            Bit pattern: 010100 -> Next hop: 21
        Child bit pattern: 01
            Bit pattern: 010101 -> Next hop: 46
    Child bit pattern: 10
        Bit pattern: 0110 -> Next hop: 4
    Child bit pattern: 11
        Bit pattern: 0111 -> Next hop: 15
Child bit pattern: 10
    Bit pattern: 10 -> Next hop: 45
    Child bit pattern: 01
        Bit pattern: 1001 -> Next hop: 7
    Child bit pattern: 11
        Bit pattern: 1011 -> Next hop: 21
Child bit pattern: 11
    Child bit pattern: 00
        Bit pattern: 1100 -> Next hop: 12
    Child bit pattern: 01
        Bit pattern: 1101 -> Next hop: 11
        Child bit pattern: 00
            Bit pattern: 110100 -> Next hop: 14
        Child bit pattern: 01
            Bit pattern: 110101 -> Next hop: 14
        Child bit pattern: 10
            Bit pattern: 110110 -> Next hop: 36
        Child bit pattern: 11
            Bit pattern: 110111 -> Next hop: 36
```

Stride = 4



Stride = 8



نکته : پرینت نتایج مثال در پوشه‌ی Example => Resluts آورده شده است.

فایل ضمیمه

نکته : نتیجه ساخت درخت برای strideهای مختلف فایل ضمیمه در پوشه‌ی Appendix => Results آورده شده است.

الف) زمان جستجو در این چهار حالت را محاسبه و با هم مقایسه کنید. آیا جواب بدست آمده مورد انتظار است یا خیر؟

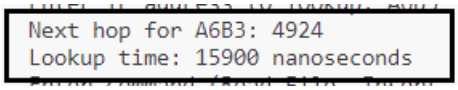
Stride = 1

```
Next hop for A6B3: 4924
Lookup time: 40300 nanoseconds
```

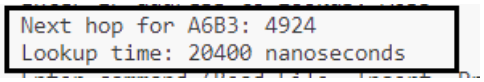
Stride = 2

```
Next hop for A6B3: 4924
Lookup time: 21800 nanoseconds
```

Stride = 4



Stride = 8



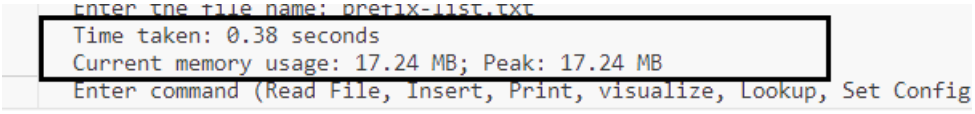
عملا یکسان شده اند. یعنی همگی 0 ثانیه بوده‌اند. به همین خاطر از نانوثانیه استفاده شده است. تفاوت به قدری معنا دار نیست که بتوان نتیجه‌گیری انجام داد. (با آزمایش مجدد مقادیر دیگری به دست می‌آیند.)

به طور کلی این که با افزایش sride، زمان lookup کمتر شود یا بیشتر وابسته به ساختار درخت است و حکم کلی جود ندارد.

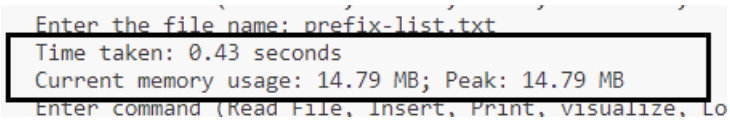
ج) برای هر چهار حالت فضای حافظه مصرف شده را بدست آوردید؟ آیا جواب بدست آمده مورد انتظار است یا خیر؟

در زیر زمان ساخت درخت و حافظه مصرفی به ازای strideهای مختلف آورده شده است.

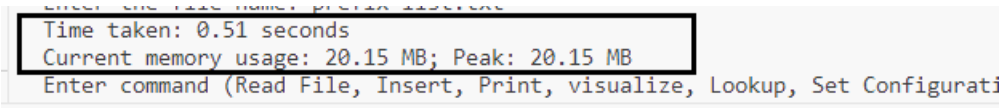
Stride = 1



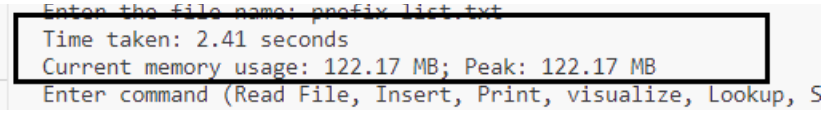
Stride = 2



Stride = 4



Stride = 8



با افزایش stride حافظه مصرف می‌تواند کاهش یا افزایش پیدا کند و این وابسته به میزان بالانس بودن بودن درخت است. چرا که با افزایش stride عمق درخت کاهش می‌یابد اما تعداد فرزندان هر گره افزایش می‌یابد.