Nén dữ liệu

Bài tập môn: Cấu trúc Dữ liệu và Thuật toán

Trường Đại học Khoa học Tự nhiên, ĐHQG Hà Nội

Lê Hồng Phương, <phuonglh@gmail.com>, 2015.

Nén dữ liệu

Nén dữ liệu là một bài toán cơ bản của tin học. Nhiệm vụ chính của nén dữ liệu là biểu diễn dữ liệu ở định dạng gọn, giúp giảm bộ nhớ cần dùng. Ta nói dữ liệu được lưu ở *định dạng nén*.

Các định dạng nén có mặt ở khắp nơi. Mỗi đĩa CD nhạc chứa các bài hát ở định dạng nén. Mỗi DVD chứa dữ liệu video nén. Máy ảnh chứa các ảnh ở một định dạng nén. Điện thoại di động cũng nén các tín hiệu âm thanh trước khi chuyển đến cột thu phát sóng GSM. Các trang web cũng được truyền đi qua mạng dưới định dạng nén trước khi tới trình duyệt, trình duyệt sẽ giải nén trước khi hiển thị. Tất cả các ứng dụng này đều không thể thực hiện được nếu không dùng các thuật toán nén. Lí do đơn giản là nếu dữ liệu không được nén thì kích thước của chúng rất lớn, không thể chứa một bộ phim trong mỗi đĩa DVD, hoặc dữ liệu thoại không thể truyền đi đủ nhanh qua mạng điện thoại trong thời gian thực, nhất là với các mạng 3G.

Có hai loại thuật toán nén. Loại thứ nhất là nén mà không mất mát dữ liệu: điển hình là nếu dữ liệu là văn bản thì không được làm mất chữ trong quá trình nén. Chẳng hạn, các trang web thường được nén bằng một thuật toán nén không mất dữ liệu như GZIP hoặc Deflate. Tuy nhiên, với hình ảnh, nhạc và video, ta có thể chấp nhận chất lượng của dữ liệu nén bị giảm đi chút ít so với dữ liệu gốc. Như vậy ta có thể sử dụng loại thứ hai là các thuật toán nén không bảo toàn dữ liệu hoàn toàn, chỉ cung cấp tỉ lệ nén tốt nhất, dựa trên các tính chất của dữ liệu cần nén. Ví dụ, thuật toán MP3 loại bỏ bớt các tần số âm thanh mà tai người không nghe được, hay một thuật toán nén video dùng phương pháp phát hiện di chuyển để biểu diễn một cảnh theo cách gọn nhất. Các hình ảnh có thể được biểu diễn bằng các định dạng nén không mất dữ liệu như TIFF hoặc PNG, hoặc các định dạng nén có mất như JPEG, tùy thuộc vào ngữ cảnh sử dụng.

Một tiêu chí cần xem xét khi nén dữ liệu là tốc độ nén: trên các thiết bị có dung lượng hạn chế như điện thoại di động, ta không thể nén các ảnh dưới định dạng cần quá nhiều tính toán để giải nén, vì điều này làm cho thiết bị tiêu hao nhiều điện năng, tốn thời gian.

Các thuật toán nén dữ liệu hiện tại đang là một chủ đề nghiên cứu rất tích cực. Người ta đang tìm cách thay thế định dạng JPEG, vì mặc dù định dạng này đang được sử dụng nhiều trên mạng nhưng chất lượng không được hoàn hảo.

Cấu trúc của bài tập

Bài tập lập trình này có hai phần, tương ứng với hai thuật toán nén không mất dữ liệu. Trong phần 1, bạn cài đặt một thuật toán nén đơn giản, gọi là thuật toán RLE. Bài tập này rèn luyện kĩ năng làm việc

trên các mảng và viết các vòng lặp không tầm thường. Trong phần 2, bạn tìm hiểu thuật toán nén cổ điển có tên LZ77, tiền thân của thuật toán LZW. Ngày nay, thuật toán LZW vẫn được sử dụng để nén các ảnh dưới định dạng GIF hoặc TIFF và các tài liệu PDF. LZ77 được sử dụng trong hệ thống tệp tin NTFS của Microsoft, và trong thuật toán Deflate, kết hợp với mã Huffman.

Bài tập này tương đối dài. Phần 1 tương đối dễ, nhưng phần 2 khó hơn.

1. Thuật toán RLE

Thuật toán RLE thích hợp để nén các ảnh đen trắng. Mục tiêu chính là rút gọn một mảng nhị phân thành một mảng ngắn hơn, như sau. Thay vì biểu diễn ảnh bởi mảng gồm 9 phần tử {0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0} ta dùng mảng gồm 5 phần từ {0, 5, 1, 1, 0, 5}, biểu diễn số lần lặp lại của mỗi phần tử: lặp lại 5 lần số 0, 1 lần số 1 và lại 5 lần số 0.

Trong bài tập này, tiêu chuẩn nén của ta là độ dài của mảng trong Java. Lưu ý rằng trong thực tế, ta sẽ cần dùng phương án biển diễn hiệu quả các bit để tránh lãng phí bộ nhớ. Ở đây, ta không quan tâm tới việc xử lí các bit mà chấp nhận là mỗi số 0 và 1 đều được biểu diễn bằng một số nguyên, chiếm 32 bits trong bộ nhớ.

1.1. Độ dài của mảng nén

Mở tệp RLE.java, viết hàm tĩnh public static int length(int[] t) để tính độ dài của mảng nén của một mảng t. Trong ví dụ trên, hàm này sẽ trả về 6.

Hàm bạn viết cần xử lí các trường hợp đặc biệt. Nếu đầu vào là mảng độ dài 0 thì mảng nén cũng có độ dài 0. Nếu mảng vào là null thì cần báo lỗi, trả về lỗi nghiêm trọng, như IllegalArgumentException. Những quy định như vậy cần được thực hiện trong suốt bài tập này.

Mở tệp Test.java, chạy hàm testRle1, kiểm tra kết quả.

1.2. Nén

Tiếp theo, viết hàm public static int[] compress(int[] t) để nén mảng t bằng thuật toán RLE, trả về mảng nén. Hàm này sử dụng hàm length(int[]) để biết độ dài của mảng kết quả cần tạo.

Mở tệp Test.java, chạy hàm testRle2, kiểm tra kết quả.

1.3. Giải nén

Đây là thao tác ngược với thao tác nén. Ta tiến hành theo hai bước tương tự như thao tác nén. Trước tiên viết hàm public static int lengthInverse(int[] t) để tìm độ dài của mảng giải nén khi biết mảng nén t. Sau khi viết xong hàm này, chạy hàm testRle3 để kiểm tra kết quả. Tiếp theo, viết hàm public static int[] decompress(int[] t) để giải nén một mảng t được nén bằng thuật toán RLE. Chạy hàm testRle4() để kiểm tra kết quả.

2. Thuật toán LZ77

Thuật toán LZ77 phức tạp hơn và khó cài đặt hơn thuật toán RLE. Tuy nhiên thuật toán này có tỉ lệ nén tốt hơn RLE. Khi cài đặt, bạn nên thử và kiểm tra bằng tay với những dữ liệu nhỏ, nên hiển thị

các mảng, bổ sung hàm toString() để hiển thị các lớp tương ứng.

2.1. Mô tả thuật toán LZ77

Ý tưởng của thuật toán LZ77 gói gọn trong cụm từ "*lịch sử lặp lại*". Nói cách khác, LZ77 sẽ tìm sự giống nhau giữa *tương lai* và *quá khứ*. Ý tưởng này được mô tả chi tiết như sau.

Ta lưu một *vị trí hiện tại* (currentPosition) trong mảng cần nén. Vị trí hiện tại này đánh dấu điểm khởi đầu của *các kí tự sắp tới* (phần mảng chưa được nén); vị trí hiện tại cũng đánh dấu điểm kết thúc của cửa sổ (windows), gồm N kí tự ở trước vị trí hiện tại.

```
{ 1, 0, 1, 0, 1, 1, 1 } ← mảng cần nén
            ↑
          vị trí
          hiện tại
```

Trong ví dụ trên, các kí tự sắp tới là {0, 1, 1, 1}, và cửa sổ, ta giả định có kích thước N = 100, là {1, 0, 1}; trong ví dụ này, cửa sổ chỉ gồm 3 kí tự chứ không phải 100 kí tự vì bị chặn từ đầu dãy.

LZ77 tìm *sự tương tự* giữa *các kí tự sắp tới* và *cửa sổ*. Ví dụ, chuỗi {0, 1} xuất hiện ở vị trí hiện tại và cũng xuất hiện ở cửa sổ, ở hai kí tự trước đó. Ta nói rằng ta tìm được một *sự xuất hiện* (occurrence). Sự xuất hiện này được đặc trưng bởi *kích thước* (size) của nó, ở đây là 2, và *vị trí trở về* (retour), là vị trí của nó tương ứng với vị trí hiện tại. Trong ví dụ trên, sự xuất hiện nằm trước vị trí hiện tại 2 kí tự, nên vị trí trở về của nó là 2.

Khi ta nói rằng ta tìm một sự xuất hiện, thì ý nói ta tìm sự xuất hiện luôn bắt đầu ở vị trí hiện tại.

2.2. Cấu trúc dữ liệu

Ta tạo lớp Occurrence mô tả sự xuất hiện như sau:

```
class Occurrence {
	int retour;
	int size;

	Occurrence (int retour, int size) {
		this.retour = retour;
		this.size = size;
	}

	@Override
	public String toString() {
		return "(" + retour + "," + size + ")";
	}
}
```

2.3. Tìm sự xuất hiện

Viết hàm public static Occurrence longestOccurrence(int[] t, int currentPosition, int windowSize) của lớp LZ77. Hàm này có tham số là một mảng vào cần nén, vị trí hiện tại và kích thước cửa sổ, trả về

sự xuất hiện dài nhất tìm được **trong cửa sổ**. Chú ý rằng kí tự hiện tại không nằm trong cửa sổ.

Nếu không tìm được sự xuất hiện nào thì hàm này trả về một Occurrence với kích thước 0 và vị trí trở về 0. Trong trường hợp có nhiều lần xuất hiện có cùng kích thước, thì hàm này sẽ trả về sự xuất hiện nằm bên trái nhất, tức là ở vị trí xa nhất so với vị trí hiện tại.

Rõ ràng, nếu cửa sổ càng lớn thì hàm này càng phải tính toán nhiều. Trong bài tập này, ta cố định cửa sổ có kích thước 100.

Hàm này thực hiện khá nhiều tính toán trên mảng, dễ có lỗi nếu không cẩn thận. Bạn nên dùng bút và giấy để suy nghĩ và tính toán trước khi lập trình. Cần lưu ý thêm một số điểm sau:

- Giả thiết và kết luận của hàm này là gì?
- Trong các trường hợp đặc biệt thì hàm thực hiện như thế nào? (Ở vị trí đầu của mảng, ở vị trí cuối của mảng)
- Cấu trúc tổng quát của hàm này là gì? Có bao nhiêu vòng lặp lồng nhau được thực hiện?

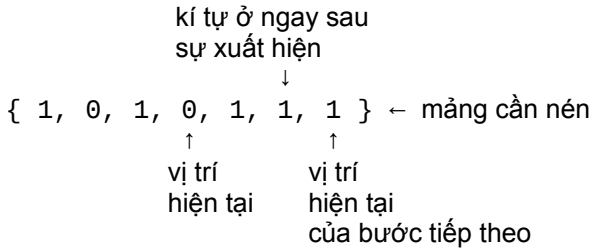
Mở tệp Test.java, chạy hàm testLz1(), kiểm tra kết quả.

2.4. Mô tả thuật toán LZ77 (tiếp theo)

Thuật toán LZ77 thực hiện như sau:

1. Ta bắt đầu từ đầu mảng cần nén;
2. Ta tìm sự xuất hiện dài nhất o, có kích thước k;
3. Ta lưu lại o cùng với *kí tự sắp tới*, ở vị trí ngay sau o;
4. Ta tiến k+1 kí tự và quay lại bước 2.

Ta lấy lại ví dụ ở trên:



Tại thời điểm này, thuật toán lưu sự xuất hiện (2, 2) và kí tự 1, chuyển tới bước tiếp theo. Việc lưu lại kí tự sắp tới là quan trọng. Không phải lúc nào ta cũng tìm được sự xuất hiện. Nhưng lúc nào ta cũng phải lưu lại kí tự nằm ở vị trí hiện tại.

2.5. Cấu trúc dữ liệu (tiếp theo)

Ta sử dụng lớp Element để lưu các *phần tử* của thuật toán LZ77. Mỗi Element chứa hai trường dữ liệu: một trường lưu sự xuất hiện và một trường lưu kí tự sắp tới:

```
class Element {
	Occurrence o;
	int s;

	Element(Occurrence o, int s) {
		this.o = o;
		this.s = s;
	}

	@Override
	public String toString() {
		return o.toString() + s;
	}
}
```

Như vậy, kết quả của phép nén sẽ là một chuỗi phần tử, chứa trong một mảng Element[].

2.6. Quy tắc kết thúc

Ta giả định rằng mảng cần nén chỉ chứa các kí tự nhị phân 0, 1 và luôn kết thúc bằng một kí tự đặc biệt là 2. Quy tắc này là quan trọng để thuật toán chạy đúng trong mọi trường hợp. Bạn cần bổ sung một số 2 vào cuối mỗi mảng trước khi chạy thuật toán.

Xét ví dụ sau, giả sử mảng cần nén là {1, 1}. Tại vị trí 1, ta muốn tạo sự xuất hiện (1, 1). Khi đó ta cần truy cập vào kí tự tiếp theo, không tồn tại vì ta đã ở cuối mảng. Việc thêm số 2 vào cuối mảng đảm bảo rằng số 2 không bao giờ tham gia vào bất kì sự xuất hiện nào, và sẽ luôn là kí tự sắp tới hợp lệ.

Như vậy, trong thuật toán này, ta luôn coi số 2 là số cuối cùng của mảng, từ đó số này được tính vào độ dài của mảng, được nén như những kí tự khác, và khi giải nén thì cũng là kí tự cuối cùng của kết quả.

2.7. Độ dài của mảng nén

Viết hàm public static int length(int[] t, int windowSize) tính số phần tử cần dùng để biểu diễn một mảng t khi được nén bằng thuật toán LZ77. Hàm này sẽ gọi hàm longestOccurrence đã viết ở trên. Chú ý rằng mỗi khi tìm được một sự xuất hiện, bạn cần tiến tiếp tới các vị trí tiếp theo của mảng.

Chạy hàm testLz2() và kiểm tra kết quả.

2.8. Nén

Hàm nén dữ liệu tương tự như hàm tính độ dài của mảng nén đã viết ở trên. Tuy nhiên, thay vì trả về độ dài của mảng, hàm này trả về mảng Element[]. Viết hàm public static Element[] compress(int[] t, int windowSize) để nén mảng t và trả về mảng nén.

Để hiển thị và kiểm tra kết quả nén, bạn dùng hàm printCompression(Element[]) đã viết sẵn. Hàm này in mảng nén gồm các Element ra và kết thúc bằng dấu xuống dòng. Mỗi Element được in theo định dạng (r, t)s trong đó (r, t) biểu diễn sự xuất hiện và s là kí tự sắp tới. Mảng kết quả nén gồm các

Element viết liên tiếp nhau, không có kí tự trắng. Ví dụ, nếu bạn lập trình đúng, khi chạy hàm testLz3() thì kết quả sẽ bắt đầu bằng chuỗi sau:

```
(0,0)0(1,1)0(0,0)1(4,4)1(2,2)0(10,3)1
```

Chạy testLz3() và kiểm tra kết quả.

2.9. Độ dài của mảng giải nén

Viết hàm public static int lengthInverse(Element[] t) lấy đầu vào là một mảng nén và trả về độ dài của mảng giải nén. Hàm này tương đối đơn giản, vì mỗi Occurrence đã chứa thông tin về độ dài.

Chạy testLz4() và kiểm tra kết quả.

2.10. Giải nén

Viết hàm public static int[] decompress(Element[] t) để giải nén một mảng đã nén. Cho t là một mảng nén dữ liệu, hàm này trả về mảng dữ liệu ban đầu. Khi cài đặt hàm này, bạn sẽ cần tới hàm phụ trợ static void blit(int[] t1, int[] t2, int start1, int len, int start2) để chép các kí tự từ mảng t1, bắt đầu từ vị trí start1 tới vị trí start1 + len – 1, sang mảng t2, bắt đầu từ vị trí start2. Bạn nên thử hàm này với các mảng có kích thước nhỏ để đảm bảo hàm phụ trợ này chạy đúng, trước khi sử dụng nó trong hàm giải nén.

Để hiển thị và kiểm tra kết quả giải nén, bạn dùng hàm printDecompression(int[] t) đã viết sẵn. Hàm này in mảng t ra màn hình, kết thúc bằng dấu xuống dòng. Mỗi kí tự của mảng cách nhau bằng một kí tự trắng. Ví dụ, nếu bạn lập trình đúng, khi chạy hàm testLz5() thì kết quả sẽ bắt đầu bằng chuỗi sau

```
1 0 1 0 1 0 0 0 0 1 0 1 1 1 0 0 1 1 1 1
```

Chạy testLz5() và kiểm tra kết quả.