## Lab5 同义词实验

### 1. 实验要求

本次实验中，你将完成一个寻找无权图中最短路径的ADT，最短路算法被广泛应用于很多地方，你将应用你的解决方案用于处理同义词问题。即给定任意两个单词，在给定的同义词库中找出它们之间的最短路，并且任意两个单词有边相连表示他们是同义词关系。

### 2. 回答问题

#### 2.1.0 Graph Construction定义graph的类型

Task 4.1 (2%). In MkAllShortestPaths.sml, deﬁne the type graph that would allow you to implement the following functions within the required cost bounds. **Leave a brief comment explaining why you chose the representation that you did.**

|  |
| --- |
| Graph类型定义代码如下：  type graph = (((Set.set) table)\*int\*int)  定义的graph类型为：一个表示图的table \* 图的边的数量 \* 图的节点的数量。  其中table的key为当前节点，value为它的outneighbors，即为一个子节点指向出度节点的table。然后我们要把这个table用一个set存（一个key有多个对应value），因为我们后续在计算节点数量时候需要用到set里面的union操作。 |

#### 2.1.1 简述makeGraph函数的思路

Task 4.2 (8%). Implement the function

makeGraph : edge seq -> graph

which generates a graph based on an input sequence E of directed edges. The number of vertices in the the resulting graph is equal to the number of vertex labels in the edge sequence. For full credit, makeGraph must have O(|E|log|E|) work and O(log2|E|) span.

|  |
| --- |
| makegraph求解思路如下：   1. 对于sequence E（边的集合）进行collect，得到我们想要的key指向outneighbors的一个table，然后用map操作把collect之后得到的seq变成set，方便后续union操作求得vertices的数量。 2. 将table中的key和E中的第二个元素（即每个key对应的outneighbor）进行union操作，得到的集合即为图中所有的点。 3. 函数返回值即为我们构造好的table\_graph，边的数量length E，以及节点的数量Set.size vertices。 |

#### 2.1.2 makeGraph函数的代码实现及复杂度分析

|  |
| --- |
| 代码实现如下：  fun makeGraph (E : edge seq) : graph =  let  val table\_graph = Table.map (Set.fromSeq) (Table.collect E)  val vertices = Set.union(domain table\_graph, Set.fromSeq (map #2 E))  in  (table\_graph, length E, Set.size vertices)  end  复杂度分析如下：  第一句构造table\_graph，collect复杂度为O(|E|log|E|) work 和 O(log2 |E|) span，然后再用map把seq变成set，复杂度为O(V)的work和O(1)的span，不会超过collect的复杂度。  第二句把key和value进行set.union, union复杂度为O(mlog(1+n/m)) work 和 O(log2 (1+n/m)),其中n为两个set中较大的，m为两个set中较小的。显然m肯定远小于|E|（因为节点数量远少于边的数量），1+n/m更是远小于|E|，这一句的复杂度远小于collect复杂度。  计算length E的work在O(1),span在O(1);计算Set.size vertices的work在O(|V|),span为O(1)  综上，这个函数主要复杂度在collect上，复杂度为O(|E|log|E|) work 和 O(log2 |E|) span。 |

#### 2.1.3 numEdges和numVertices函数的实现

Task 4.3 (6%). Implement the functions

numEdges : graph -> int numVertices : graph -> int

which return the number of directed edges and the number of unique vertices in the graph, respectively.

|  |
| --- |
| 思路如下：  我们在makegarph的时候已经计算了numEdges和numVertices，这里直接调用即可，代码实现如下：  (\* Task 2.2 \*)  (\*边的数量即为graph的第二个元素\*)  fun numEdges (G : graph) : int = #2 G  (\*节点的数量即为graph的第三个元素\*)  fun numVertices (G : graph) : int = #3 G |

#### 2.1.4 简述outNeighbors函数的思路

Task 4.4 (6%). Implement the function

outNeighbors : graph -> vertex -> vertex seq

which returns a sequence Vout containing all out neighbors of the input vertex. In other words, given a graph G =(V,E), outNeighbors G v containsall w s.t. (v,w)∈E. If the input vertex is not in the graph, outNeighbors returns an empty sequence. For full credit, outNeighbors must have O(|Vout|+log|V|) work and O(log|V|) span, where V is the set of vertices in the graph.

|  |
| --- |
| 实现思路如下：  我们查找v的value，如果v没有value，证明它没有outneighbors，返回空串；如果有value，返回这些value |

#### 2.1.5 outNeighbors函数的代码实现及复杂度分析

|  |
| --- |
| 代码实现如下：  fun outNeighbors (G : graph) (v : vertex) : vertex seq =  case find (#1 G) v of  NONE => Seq.empty()  |SOME s => Set.toSeq s  复杂度分析：  Find这一句的复杂度的work和span均为O(log|V|)，Set.toSeq s这一句work为O(|Vout|)，span为O(1)。综上这个函数对的work为O(|Vout|+log|V|)，span为 O(log|V|)。 |

#### 2.1.6 ASP类型的确定及简短说明

Task4.5(2%). InMkAllShortestPaths.sml,deﬁnethetypeaspthatwouldallowyoutoimplement the following functions within the required cost bounds. Leave a brief comment explaining why you chose the representation that you did.

|  |
| --- |
| ASP类型定义代码如下：  type asp = (vertex seq table)  我们定义的asp类型为一个由子节点指向父节点seq的table |

#### 2.1.7 简述makeASP函数的思路

Task 4.6 (23%). Implement the function

makeASP : graph -> vertex -> asp

to generate an asp which contains information about all of the shortest paths from the input vertex v to all other reachable vertices. If v is not in the graph, the resulting asp will be empty. Given a graph G = (V,E), makeASP G v must have O(|E|log|V|) work and O(Dlog2|V|) span, where D is the longest shortest path (i.e., the shortest distance to the vertex that is the farthest from v).

|  |
| --- |
| 解决思路如下：   1. 如果节点v没有outneighbors，即返回空的table。 2. 其他情况下，执行BFS操作，如果F(frontier)为空，则返回X(已访问过的节点)。 3. 如果F不为空，首先定义一个函数mkparentpair，对于节点s的outneighbors，定义由子节点反过来指向父节点的pair。 4. 对于当前的frontier的节点，全部调用mkparentpair函数，得到每个节点由子节点指向父节点的串，将这个串flatten之后collect，得到了每个子节点指向其所有父节点的table，这些父节点用seq表示。 5. 将F加入X形成新的X，将新指向的frontier减去已经访问过的X得到新的F，将新的X和F输出，继续以其为参数调用BFS,直到F的大小为0 |

#### 2.1.8 makeASP函数的代码实现及复杂度分析

|  |
| --- |
| 代码实现如下：  fun makeASP (G : graph) (v : vertex) : asp =  (\*构造一个由子节点指向父节点的table，即我们所求的asp，开始节点为v\*)  case find (#1 G) v of  (\*如果v节点没有邻接的边，即返回空的table\*)  NONE => Table.empty()  |SOME \_ =>  let  fun BFS (X,F) =  if size F = 0 then X else  let  (\*这个函数将s指向它的父节点(outNeighbors)\*)  fun mkparentpair s = Seq.map (fn v => (v,s)) (outNeighbors G s)  (\*这一步将所有F中的元素指向好的pairs进行flatten之后collect合并，最终 变为F中的元素每个都指向其父元素的table\*)  val table\_parent\_pair = Table.collect (flatten (map mkparentpair (Set.toSeq (domain F))))  (\*这一步将X和F合并，形成新的X，已访问过的节点\*)  val update\_X = Table.merge (fn(a,b) => a) (X,F)  (\*这一步将新的将新的Frontier，即从集合中去掉已经访问过的X\*)  val update\_F = Table.erase (table\_parent\_pair, domain update\_X)  in  (\*新的X和F，递归进行BFS操作\*)  BFS(update\_X, update\_F)  end  in  (\*初始状态，已访问过节点集合X为空，Frontier为v\*)  BFS(Table.empty(), Table.singleton(v,Seq.empty()))  end  复杂度分析如下：  回顾整个算法，首先看v节点在不在graph里面，如果不在，返回空的table，其余情况调用BFS,这一句work和span仅为O(log|V|)。调用BFS即为，对当前F（frontier）都执行当前节点指向其parents的操作然后再对key（即子节点）进行collect构造出子节点指向父节点seq的table，这样递归调用BFS，这一句work 为O(||F||log|V|)，span为O(log2 |V|)，然后每次更新已经访问过的节点X和F（frontier），更新X即把F并进原来的X，work不会超过O（|F|），span不会超过O（1）；更新F即为寻找当前更新过的X新的frontier，即为collect过后的table的元素，并去掉已访问过的元素，work不会超过O（log|V|），span不会超过O（1），结束条件是直到F里面元素个数为0，遍历完整个graph。又我们结束时遍历完了所有的边，边的总数为|E|，每一次调用BFS的work保持在O(||F||log|V|)，故递归完成时我们的遍历完所有的边，work为O（|E|log|V|），又因为深度为D，每一次span保持在O(log2 |V|)，故总的span保持在0(Dlog2 |V|)  综上，该算法的work为O（|E|log|V|），span为0(Dlog2 |V|)。 |

#### 2.1.9 简述report函数的思路

Task 4.7 (15%). Implement the function

report : asp -> vertex -> vertex seq seq

which, given an asp for a source vertex u, returns all shortest paths from u to the input vertex v as a sequence of paths (each path is a sequence of vertices). If no such path exists, report asp v evaluates to the empty sequence. For full credit, report must have O(|P||L|log|V|) work and span, where V is the set of vertices in the graph, P is the number of shortest paths from u to v, and L is the length of the shortest path from u to v.

|  |
| --- |
| 代码实现思路：   1. 首先寻找G中有无v，如果没有，返回空串 2. 如果有v但是v没有frontier，返回v 3. 其它情况下，从v开始调用DFS，每一次找其parent，遍历所有路径到各初始节点u 4. 最后将串翻转，得到从u到v的路径 |

#### 2.1.10 report函数的实现及复杂度分析

|  |
| --- |
| 代码实现如下：  fun report (A : asp) (v : vertex) : vertex seq seq =  let  fun DFS (v : vertex) : vertex list seq =  (\*寻找节点v是否在A中，如果不在，依照题目要求返回空串，如果为孤立点，它的path为它自己\*)  case Table.find A v of  NONE => Seq.empty ()  |SOME parents => if (Seq.length parents) = 0 then Seq.singleton [v]  else  (\*其他情况，从v开始，递归调用DFS，倒过来沿着每条路径去找初始节点\*)  let  fun path\_to\_parent u = DFS u  fun add\_v vpath = [v] @ vpath  val parent\_vpath = Seq.flatten (Seq.map path\_to\_parent parents)  val vpath = Seq.map add\_v parent\_vpath  in  vpath  end  in  (\*将串翻转，得到u到v的路径\*)  let  fun reverse\_seq\_vpath vpath = Seq.rev (Seq.fromList vpath)  val all\_vpath = DFS v  val afterRev\_vpath = Seq.map (reverse\_seq\_vpath) (all\_vpath)  in  afterRev\_vpath  end  end  复杂度分析：  回顾整个算法，首先我们用find查找v是否在A里面，如果有返回其parents(v对应的value)，如果没有返回空串，这一步复杂度为O(log|V|)。然后我们递归调用DFS函数，并每次利用map操作对每个parents里面的元素指向下一个元素，将map后的结果flatten，然后我们将构造好的路径加上v（因为我们从v的parents开始构造的，没有算v），由此构造出v到起始节点u的路径。由于我们是递归执行以上操作，所以这里的work和span就是每次遍历的节点的个数的和乘上O(log|V|)(因为我们每次都要调用find)。又因为一共有路径|P|条，路径深度为|L|，所以节点遍历个数和为|P|\*|L|,每一次都要调用find，所以构造从v到起点u的路径work和span为O(|P||L|log|V|)。最后我们还要将串翻转，得到从起点u到v的路径，由于一共有路径|P|条，路径深度为|L|，所以work不会超过O(|P|\*|L|),我们可以并行执行此操作，所以span不会超过O(|L|)。  综上，整个算法的work和span为O(|P||L|log|V|)。 |

#### 2.1.11 关于测试

Task 4.8 (6%). Test your ALL\_SHORTEST\_PATHS implementation in the ﬁle Tests.sml by adding test cases to the appropriate lists (see the ﬁle for reference – there are existing test cases to guide you). The following functions are deﬁned to you’re your implementation of functions of ALL\_SHORTEST\_PATHS against your test cases in Tests.sml.

|  |
| --- |
| 原本测试样例和自己添加的测试如下：  (\*下面为原本的测试样例\*)  val edgeseq = [(1,2)]  val edgeseq2 = [(1,2),(2,3),(3,4),(2,4),(1,5),(5,4),(5,6),(6,7)]  val test3 = [(2,1),(4,2),(4,3),(4,5),(1,3),(3,5),(3,2),(1,4),(2,5),(5,1)]  (\*下面为自己添加的测试数据\*)  val myedgeseq1 = [(1,4),(2,6),(3,4),(5,1),(4,2),(5,3),(1,2)]  val myedgeseq2 = [(1,2),(1,3),(1,4),(1,5),(2,3),(2,4),(2,5),(3,4),(3,5),(4,5)]  (\*NumEdges和NumVertices测试\*)  val testsNum = [(\*以下为本身提供的样例\*)  edgeseq, edgeseq2,test3,  (\*以下为自己添加的测试\*)  myedgeseq1,myedgeseq2  ];  (\*OutNeighbors测试\*)  val testsOutNeighbors = [(\*以下为本身提供的样例\*)  (edgeseq, 1), (edgeseq, 2),(test3,1),(edgeseq2,5),(edgeseq2,7),(test3,9),  (\*以下为自己添加的测试\*)  (edgeseq2,3),(test3,3),(test3,5),  (myedgeseq1,1),(myedgeseq1,3),(myedgeseq1,5),  (myedgeseq2,1),(myedgeseq2,3),(myedgeseq2,5)]  (\*Report测试\*)  val testsReport = [(\*以下为本身提供的样例\*)  ((edgeseq, 1), 2), ((edgeseq2, 1), 4), ((edgeseq2, 1), 7),((test3,4),2),  ((test3,1),3),((test3,6),2),((test3,1),6),  (\*以下为自己添加的测试\*)  ((edgeseq2,3),4),((test3,3),7),((test3,5),7),  ((myedgeseq1,2),6),((myedgeseq1,3),7),((myedgeseq1,4),8),  ((myedgeseq2,1),4),((myedgeseq2,4),1),((myedgeseq2,5),2)]  NumEdges测试截图如下：    NumVertices测试截图如下：    OutNeighbors测试截图如下：    Report测试截图如下： |

#### 2.1.12 定义thesaurus数据类型并简述理由

|  |
| --- |
| thesaurus类型定义代码如下：  type thesaurus = graph  把thesaurus定义成graph，对于近义词，即为key指向其近义词的一个pair，这样形成一个graph。 |

#### 2.1.13 实现下列函数并简述算法核心

Task 5.1 (10%). Implement the function

make : (string \* string seq) seq -> thesaurus which generates a thesaurus given an input sequence of pairs (w,S) such that each word w is paired with its sequence of synonyms S. You must deﬁne the type thesaurus yourself.

|  |
| --- |
| 算法思路如下：   1. 定义一个single\_mkedges函数，作出word指向每个synonyms的pair。 2. 对于S中每一个w调用上述函数，构造所有的pairs。 3. 用makegraph将其变成图。   实现代码如下：  fun make (S : (string \* string seq) seq) : thesaurus =  let  (\*这里对于一个w，作出由word指向synonyms的pair\*)  fun single\_mkedges (w, syn) = Seq.map (fn s => (w, s)) syn  (\*对于S中的每一个w，都作single\_mkedges操作，构造所有的pairs\*)  val mkedges = flatten (map single\_mkedges S)  in  (\*将edges数对变成图\*)  makeGraph mkedges  end |

Task 5.2 (6%). Implement the functions

numWords : thesaurus -> int

synonyms : thesaurus -> string -> string seq

where numWords counts the number of distinct words in the thesaurus while synonyms returns a sequence containing the synonyms of the input word in the thesaurus. synonyms returns an empty sequence if the input word is not in the thesaurus.

|  |
| --- |
| 算法实现思路：  直接对已经变成graph类型的T，调用numVertices以及outNeighbors即可。  算法实现如下：  (\* Task 3.2 \*)  (\*word的数量就是graph中节点的数量\*)  fun numWords (T : thesaurus) : int = numVertices T  (\*每个w的近义词就是它的outneighbors\*)  fun synonyms (T : thesaurus) (w : string) : string seq = outNeighbors T w |

Task 5.3 (10%). Implement the function

query : thesaurus -> string -> string -> string seq seq such that query th w1 w2 returns all shortest path from w1 to w2 as a sequence of strings with w1 ﬁrst and w2 last. If no such path exists, query returns the empty sequence. For full credit, your function query must be staged.

|  |
| --- |
| 算法实现思路：  要返回w1到w2的all shortest path，只需要对其进行report操作，这里我们以w1为原点，构造一个asp，report w1到w2的all shortest path  实现代码如下：  fun query (T : thesaurus) (w1 : string) (w2 : string) : string seq seq =  report (makeASP T w1) w2 |

#### 2.1.14 关于测试

Task 5.4 (6%). Test your THESAURUS implementation in the ﬁle Tests.sml as before. The following functions are deﬁned to run your implementation of functions of ALL\_SHORTEST\_PATHS against your test cases in Tests.sml.

|  |
| --- |
| 测试样例以及自己添加的样例如下：  val testfile = "input/thesaurus.txt"  val testfile2 = "input/simpletest.txt"  (\*NumWords测试\*)  val testsNumWords = [testfile, testfile2]  (\*Synonyms测试\*)  val testsSynonyms =  [(\*以下为原始测试样例\*)  (testfile2, "HANDSOME"),  (testfile2, "VINCENT"),  (testfile2, "PRETTY"),  (testfile2,"BADASS"),  (testfile, "GOOD"),  (\*以下为自己添加的测试样例\*)  (testfile,"EDIT"),  (testfile,"PRICE"),  (testfile,"MAY"),  (testfile2,"YOLO"),  (testfile2,"BILL")]    (\*Query测试\*)  val testsQuery =  [ (\*以下为原始测试样例\*)  (testfile2, ("HANDSOME", "YOLO")),(testfile2,("BADASS","STUPID")),(testfile2,("PRETTY","CHRIS")),  (testfile, ("GOOD", "BAD")),(testfile, ("CLEAR", "VAGUE")),(testfile, ("LOGICAL", "ILLOGICAL")),  (testfile,("HAPPY","SAD")),(testfile,("LIBERAL","CONSERVATION")),(testfile, ("EARTHLY", "POISON")),  (\*以下为自己添加的测试样例\*)  (testfile, ("EVENT","DOZE")),(testfile,("EQUIP","LAMENTATION")),  (testfile,("A","ONE")),(testfile,("ARC","ARCH")),(testfile,("HOME","WAX")),(testfile,("AMPLE","LESS"))]  NumWords测试如下：    Synonyms测试截图如下：    Query测试截图如下： |