KMP multiple pattern matching

Authored by omri nahor

211359054

==Description==

A pattern matching algorithem which builds a data structure that supports the KMP algorithem search with a modified pi function(will be named fail function in this article).

The data structure is an ‘ascii’ tree in which each node represents one and only one string that can be lead to it. If two strings share a prefix, they will also share the same nodes for that prefix.

Main interface is done using the pm\_pattern\_matching functions that uses a modified linked list labeled as slist.

==structures==

1. pm\_labeled\_edge(pm\_labeled\_edge\_t) – a structure that holds a match between a label(char) and a state. In the context of this program, that match is a one to one match between 2 states connected by a the label. Variables:

* label – holds the label
* state – pointer to the end-edge state

1. pm\_state(pm\_state\_t) – a node in the data structure tree. Variables:

* id – distinct ID for each node to help monitor the data structure
* Depth – depth of node in data structure to help monitoring.
* Transitions – a linked list of edges to advance in the tree after “reading” a label(char from the string)
* Fail – a pointer to a state in case the goto function doesn’t have a next state to go to with the read label
* Output – a linked list of strings that holds all the output that should be displayed if this node was reached

1. pm\_match(pm\_match\_t) – holds the data of a match. Variables:

* pattern – the pattern that was found
* start\_pos – the starting position(index) in the searched string the the pattern was found
* end\_pos – the starting position(index) in the searched string the the pattern was found
* fstate – the state that the algorithem was in when the match was made

1. pm(pm\_t) – the “head” of the data structure. Variables:

* new\_state – a counter of the number of states in the data structure. Used to assign id to new states
* zero\_state – a pointer to the initial state of the data structure(represent an empty string)

==functions==

* pm\_init(pm\_t\*) – initializes the fields of pm\_t. doesn’t allocate space for pm\_t!
* pm\_addstring(pm\_t \*,const unsigned char \*, \_size\_t n) – adds a string of length n as a pattern in the data structure. Return value: 0 – success, -1 – error.
* pm\_makeFSM(pm\_t \*) – calculates the fail function and applies it to all nodes currently in the data structure. Return value: 0 – success, -1 – error.
* pm\_goto\_set(pm\_state\_t \*from\_state,unsigned char symbol,pm\_state\_t \*to\_state) – creates an edge with values symbol, to\_state and adds it to from state’s transitions list. Return value: 0 – success, -1 – error.
* pm\_goto\_get(pm\_state\_t \*state,unsigned char symbol) – search’s for an edge with label the same as symbol in transitions list of state. Return value: NULL – no such edge found, else – state pointer value from the found edge.
* pm\_fsm\_search(pm\_state\_t \*,unsigned char \*,\_size\_t) – search’s for all matches of any pattern logged into the data structure in string size n, starting from given state. Return value: a list of pm\_match\_t containing all match’s found.

NOTE: the list is dynamically allocated and must be freed first with dbllist\_destroy(\*list pointer\*, DBLLIST\_FREE\_DATA) and then using free(\*list pointer\*) to avoid data loss.

==Program Files==  
pm\_patterm\_matching – all pm structures and functions

Slist – all list structures and functions

==how to compile==