École Centrale de Nantes

AlPro

Tutorial 2

2.1 Checkers

We consider a checkers game over a 10 by 10 checkerboard with 8 men and a king. The positions of men and the king are given in a file in the format you will define (it may depend on the data structure your choose). The king can move over diagonals on which he is, forward and backward. He chose a direction, go over the man he wants to capture and go to any free square behind him, then he can target a new man to do a multiple capture. This is called a flying king. The choices of direction and of positioning for the king allows a lot of possibilities. And the rules of checkers states that you have to play the move that allows you to take the most enemy men. What is the maximum number of men that we can capture with a king ?

- **Adapt the algorithm from the TP1 (for the queen and pawns problem) to the rules of checkers without the multiple capture.**
- **Modify the capture function to take into account a possible multiple capture. You may add other functions and you have to use recursivity. To see the details of the rules : https://en.wikipedia.org/wiki/Checkers**

```
int[10][10] checkerBoard; // this is a boolean matrix, with only 0s and 1s

int kx, ky; // the kings coordinates


/* main algorithm to construct the data structure and compute the maximal
   capture */

function main{
  Integer kx, ky←read_data(file);
  checkerBoard←read_data(file);
  Integer nbr←explore(checkerBoard, kx, ky);
  printf("The maximal capture is %d",nbr);
}



/* exploring algorithm to get the number of men in the capture from a point */
Integer explore(checkerBoard, kx, ky, di, dj){
  /* di, dj are the direction of the previous position, the one that we need
     should not to explore */

  Integer[4] nb_cap←{0,0,0,0,};
  Integer max←0;
  Integer count←0;
  Integer nbr← 0;
  Integer[10][10] B←copy(checkerBoard);

  for i from -1 to 1{ // i gives the direction on the x-axis
    for j from -1 to 1{ // j gives the direction on the y-axis
```

```
      if ( i!=0 AND j!=0 AND i!=di AND j!=dj){
        // to ignore all non-diagonal directions and the forbidden one.
        (mx,my)←manCaptureDir(checkerBoard,kx,ky,i,j);
        if(mx!=-1 AND my!=-1){
          // there is a capturable man with an empty square behind

          B←updateBoard(checkerBoard,mx,my); // put 2 for the capture man
          max←expLineDir(B,mx,my,i,j);
          nb_cap[count]←1+max;
        }else // no capturable man, or no empty square behind in this direction
          nb_cap[count]←0;
        count++;
      }
    }
  }return maximum(nb_cap);
}



/* function that given a position (mx,my) and a direction (i,j) visits all the
   possible moves from there to see how many men can be captured. */
Integer expLineDir(B, mx, my, i, j){
  Integer max←0;
  Integer k←1;
  Integer dx←mx*k, dy←my*k;
  Integer res←0;

  while(0≤mx+k*i≤9 AND 0≤my+k*j≤9 AND B[mx+k*i][my+k*j]==0){
    res←explore(B,mx+k*i,my+k*j,i*-1,j*-1);
    if (res>max)
      max←res;
    k++;
  }return max;
}



(Integer,Integer) manCaptureDir(B,kx,ky,i,j){
  Integer k=1;
  while(0≤kx+k*i≤9 AND 0≤ky+k*j≤9){
    if (B[kx+k*i][ky+k*j])
      return (kx+k*i,ky+k*j);
    k++;
  }return (-1,-1);
}



/* other version with only one recursive function */
int capture(chessBoard, kx, ky, num){
  int max=0;
  cBoard=copy(chessBoard);
  for(int i=-1;i<=1;i++){
    for(int j=-1;j<=1;j++){
      if(i!=0 && j!=0){
        (px,py)=manCaptureDir(cBoard,kx,ky,i,j);
        if(px!=-1 && py!=-1){
          if(cBoard[px+i][py+j]!=1 && cBoard[px][py]=2)
            cBoard[px][py]=2;
```

```
            int temp=capture(cBoard,px+i,py+j,num+1);
            if(max<temp) max=temp;
          }
        }
      }
    }
    return max;
}
```

2.2 Lists

Using a linked list of integer values, gives an algorithm to :

- **print elements of the lists**
- **add an element at the end of the list**
- **search a value and return the number of occurrences in the list**
- **remove an element whose value is given as a parameter**
- **add an element at the right place inside a sorted list**
- **search for an element in a sorted list**
- **remove an element in a sorted list**

```
typedef struct element{
  ind data;
  struct element *next;
}element;

/* 1 if e1<e2, 0 if e1==e2 and 1 if e1>e2 */
int cmp(element e1,element e2){
  if (e1.data>e2.data) return -1;
  if (e1.data==e2.data) return 0;
  return 1;
}

/* prints a list of integers */
int printList(element* list){
  element *current=list;
  printf("[ ");
  while(current!=NULL){
    printf("%d",current->data);
    if(current->next!=NULL)
      printf(", ");
    current=current->next;
  }
  printf(" ]\n");

  return 0;
}


/* recursively print a list */
void recPrintList(element* list){
  if (list!=NULL){
    printf("%d ",list->data);
    recPrintList(list->next);
  }
}


void printList1(element* list){
  printf("[ ");
```

```
  recPrintList(element* list);
  printf("]\n");
}



/* recursively add an element at the end */
void add(element* list,int nbr){
  if (list==NULL){
    element new;
    new.next=NULL;
    new.data=nbr;
    return new;
  }if (list->next==NULL){
    element new;
    new.next=NULL;
    new.data=nbr;
    list->next=new;
    return list;
  }return add(list->next,nbr);
}


/* recursively search for a value */
void search(element* list,element value){
  if (list==NULL) return 0;
  int found=0;
  if (cmp(*list,value)==0) found=1;
  return found+search(list,value);
}


/* recursively remove an element */
void remove(element* list,element value){
  if (list==NULL) return list;
  if (!cmp(*list,value))
    return list->next;
  list->next=remove(list->next,value);
}


/* add an element in a sorted list */
element* add_sort(element* list,element new){
  if(list==NULL) return new;
  if(cmp(*list,new)<0){
    new.next=list;
    return &new;
  }list->next=add_sort(list->next,new);
  return list;
}

/* search for an element in a sorted list */
int search_sort(element* list,element value){
  if(list==NULL) return NULL;
  if(cmp(*list,value)<0) return 0;
  int found=0;
  if(cmp(*list,value)==0) found=1;
  return found+search_sort(list->next,value);
}
```

```
/* recursively remove an element in a sorted list */
void remove(element* list,element value){
  if (list==NULL || cmp(*list,value)<0)
    return list;
  if (!cmp(*list,value))
    return list->next;
  list->next=remove(list->next,value);
}
```

2.3 Merge sort

The merge sort is a recursive sorting algorithm :

1. **If the array has only one element, it is already sorted.**
2. **Otherwise, separate the array into two approximately equal parts.**
3. **Recursively sort the two parts with the merge sort algorithm.**
4. **Merge the two sorted arrays into one sorted array.**

```
// merge sort
function merge_sort(array,b,e){
  // array of integer for b the beginning and e the end
  if(size(array)>1){
    (Integer e1,Integer b1)←split(array,b,e);
    merge_sort(array,b,e1);
    merge_sort(array,b1,e);
    merge(array,b,e1,b1,e);
  }
}

// split
(Integer,Integer) split(array,b,e){
  Integer e1←((e-b)/2)+b;
  Integer b1←e1+1;
  return (e1,b1);
}

// merge
void merge(array,b,e1,b1,e){
  Integer c1←b;
  Integer c2←b1;
  int e11=e1;
  while(c1<=e1 || c2<=e){
    if(array[c1]<array[c2]) )
      c1←c1+1;
    else{
      Integer tmp←array[c2];
      for(Integer i=c2-1;i>=c1;i--)
        array[i+1]←array[i];
      array[c1]←tmp;
      c2←c2+1;
      e11++;
    }
  }
}
```