Interface for module Search

1. Search algorithms, presented both functionally and imperatively.

2. Functional search

module type *FunctionalProblem* = sig

states

type *state*
val *success* : *state* → *bool*

moves

type *move*
val *moves* : *state* → (*move* × *state*) *list*

visited states are stored in a table using the following type and functions; when it is not necessary/desired to mark visited states, simply provide a function *mem* always returning false. Note: *clear* is used for iterative deepening search only.

type *table*
val *create* : *unit* → *table*
val *add* : *table* → *state* → *unit*
val *mem* : *table* → *state* → *bool*
val *clear* : *table* → *unit*
end

search s searches for a successful state starting from state s; if there is such a state, it returns it, together with the path from s; otherwise, it raises *Not_found*.
Depth-first search

module *FunctionalDFS*(*P* : *FunctionalProblem*) : sig
val *search* : *P.state* → *P.state* × *P.move list*
end

Breadth-first search

module *FunctionalBFS*(*P* : *FunctionalProblem*) : sig
val *search* : *P.state* → *P.state* × *P.move list*
end

Iterative deepening search

module *FunctionalIDS*(*P* : *FunctionalProblem*) : sig
val *search* : *P.state* → *P.state* × *P.move list*
end

3. Imperative search

```
module type ImperativeProblem = sig
```

the state is global and imperative

```
  val success : unit → bool
```

moves

```
  type move
  val moves : unit → move list
  val do_move : move → unit
  val undo_move : move → unit
```

visited states are stored using the following functions: *add* stores the current state in the table and *mem* checks whether the current state is already in the table

```
  val add : unit → unit
  val mem : unit → bool
  val clear : unit → unit
end
```

search () searches for a successful state starting from the current state; if there is such a state, it returns the path from the initial state; otherwise, it raises *Not_found*.
Depth-first search

```
module ImperativeDFS(P : ImperativeProblem) : sig
  val iter : (P.move list → unit) → unit
  val search : unit → P.move list
end
```

Breadth-first search

```
module ImperativeBFS(P : ImperativeProblem) : sig
  val search : unit → P.move list
end
```

Iterative deepening search

```
module ImperativeIDS(P : ImperativeProblem) : sig
  val search : unit → P.move list
end
```

Module Search

4. Functional search

```
module type FunctionalProblem = sig
  type state
  type move
  val success : state → bool
  val moves : state → (move × state) list
  type table
  val create : unit → table
  val add : table → state → unit
  val mem : table → state → bool
  val clear : table → unit
end
```

Depth-first search

```
module FunctionalDFS(P : FunctionalProblem) = struct
  let search s0 =
    let visited = P.create () in
    let already s = (P.mem visited s) ∨ (P.add visited s; false) in
    let rec dfs path s =
      if already s then raise Not_found;
      if P.success s then s, List.rev path else first path (P.moves s)
    and first path = function
      | [] → raise Not_found
      | (m, s) :: r → try dfs (m :: path) s with Not_found → first path r
    in
    dfs [] s0

  let search s0 =
    let visited = P.create () in
    let already s = (P.mem visited s) ∨ (P.add visited s; false) in
    let rec dfs = function
      | [] →
          raise Not_found
      | (path, s) :: _ when P.success s →
          s, List.rev path
      | (path, s) :: stack →
          dfs
            (List.fold_left
               (fun stack (m, s′) →
                  if already s′ then stack else (m :: path, s′) :: stack)
               stack (P.moves s))
    in
```

```
    let _ = already s0 in
    dfs [[], s0]
end
```

Breadth-first search

```
module FunctionalBFS(P : FunctionalProblem) = struct
  let search s0 =
    let visited = P.create () in
    (* meaning is here "already queued" *)
    let already s = (P.mem visited s) ∨ (P.add visited s; false) in
    let _ = already s0 in
    let q = Queue.create () in
    Queue.add ([], s0) q;
    let rec bfs () =
      if Queue.length q = 0 then raise Not_found;
      let path, s = Queue.take q in
      if P.success s then
        s, List.rev path
      else begin
        List.iter
          (fun (m, s') → if ¬ (already s') then Queue.add (m :: path, s') q)
          (P.moves s);
        bfs ()
      end
    in
    bfs ()
```

variant without Queue: we use two lists instead and we don't even care about reversing the second one when the first becomes empty, since order within a given level is unimportant

```
end
```

Iterative deepening search

```
module FunctionalIDS(P : FunctionalProblem) = struct
  let search s0 =
    let visited = P.create () in
    let already s = (P.mem visited s) ∨ (P.add visited s; false) in
    let depth max =
      let rec dfs n path s =
        if n > max ∨ already s then raise Not_found;
        if P.success s then s, List.rev path else first n path (P.moves s)
```

```
      and first n path = function
        | [] ->
            raise Not_found
        | (m, s) :: r ->
            try dfs (succ n) (m :: path) s with Not_found -> first n path r
      in
      dfs 0 [] s0
    in
    let rec try_depth d =
      try depth d with Not_found -> P.clear visited; try_depth (succ d)
    in
    try_depth 0
end
```

5. Imperative search

```
module type ImperativeProblem = sig
  type move
  val success : unit -> bool
  val moves : unit -> move list
  val do_move : move -> unit
  val undo_move : move -> unit
  val add : unit -> unit
  val mem : unit -> bool
  val clear : unit -> unit
end
```

Depth-first search

```
module ImperativeDFS(P : ImperativeProblem) = struct
  let iter f =
    let already () = (P.mem ()) ∨ (P.add (); false) in
    let rec dfs path =
      if ¬ (already ()) then
        if P.success () then f (List.rev path) else first path (P.moves ())
    and first path = function
      | [] ->
          ()
      | m :: r ->
          P.do_move m; dfs (m :: path); P.undo_move m; first path r
    in
    dfs []
```

```
  let search () =
    let already () = (P.mem ()) ∨ (P.add (); false) in
    let rec dfs path =
      if already () then raise Not_found;
      if P.success () then List.rev path else first path (P.moves ())
    and first path = function
      | [] →
          raise Not_found
      | m :: r →
          try P.do_move m; dfs (m :: path)
          with Not_found → P.undo_move m; first path r
    in
    dfs []
end
```

Breadth-first search

```
module ImperativeBFS(P : ImperativeProblem) = struct
```

cut n elements at head of list l

```
  let rec cut_head n l = if n ≡ 0 then l else cut_head (pred n) (List.tl l)
```

find the common physical suffix of $l1$ and $l2$

```
  let common_psuffix (n1, l1) (n2, l2) =
    (* suffix applies when the two lists have same length *)
    let rec suffix l1 l2 =
      if l1 ≡ l2 then l1 else suffix (List.tl l1) (List.tl l2)
    in
    if n1 < n2 then suffix l1 (cut_head (n2 − n1) l2)
    else if n2 < n1 then suffix (cut_head (n1 − n2) l1) l2
    else suffix l1 l2

  let search () =
    let already () = (P.mem ()) ∨ (P.add (); false) in
    let q = Queue.create () in
    Queue.add (0, []) q;
    let cpath = ref (0, []) in
    let rec restore_state path =
      let suf = common_psuffix path !cpath in
      let rec backward = function
        | (m :: r) as p when p ≢ suf → P.undo_move m; backward r
        | _ → ()
```

```
      in
      let rec forward = function
        | (m :: r) as p when p ≢ suf → forward r; P.do_move m
        | _ → ()
      in
      backward (snd !cpath);
      forward (snd path);
      cpath := path
    in
    let rec bfs () =
      if Queue.length q = 0 then raise Not_found;
      let (n, path) as s = Queue.take q in
      restore_state s;
      if P.success () then
        List.rev path
      else if ¬ (already ()) then begin
        List.iter (fun m → Queue.add (succ n, m :: path) q) (P.moves ());
        bfs ()
      end else
        bfs ()
    in
    bfs ()
end
```

Iterative deepening search

```
module ImperativeIDS(P : ImperativeProblem) = struct
  let search () =
    let already () = (P.mem ()) ∨ (P.add (); false) in
    let depth max =
      let rec dfs n path =
        if n > max ∨ already () then raise Not_found;
        if P.success () then List.rev path else first n path (P.moves ())
      and first n path = function
        | [] →
            raise Not_found
        | m :: r →
            try P.do_move m; dfs (succ n) (m :: path)
            with Not_found → P.undo_move m; first n path r
      in
      dfs 0 []
```

```
    in
    let rec try_depth d =
      Printf.eprintf "trying depth %d...\n" d; flush stderr;
      try depth d with Not_found → P.clear (); try_depth (succ d)
    in
    try_depth 0
end
```