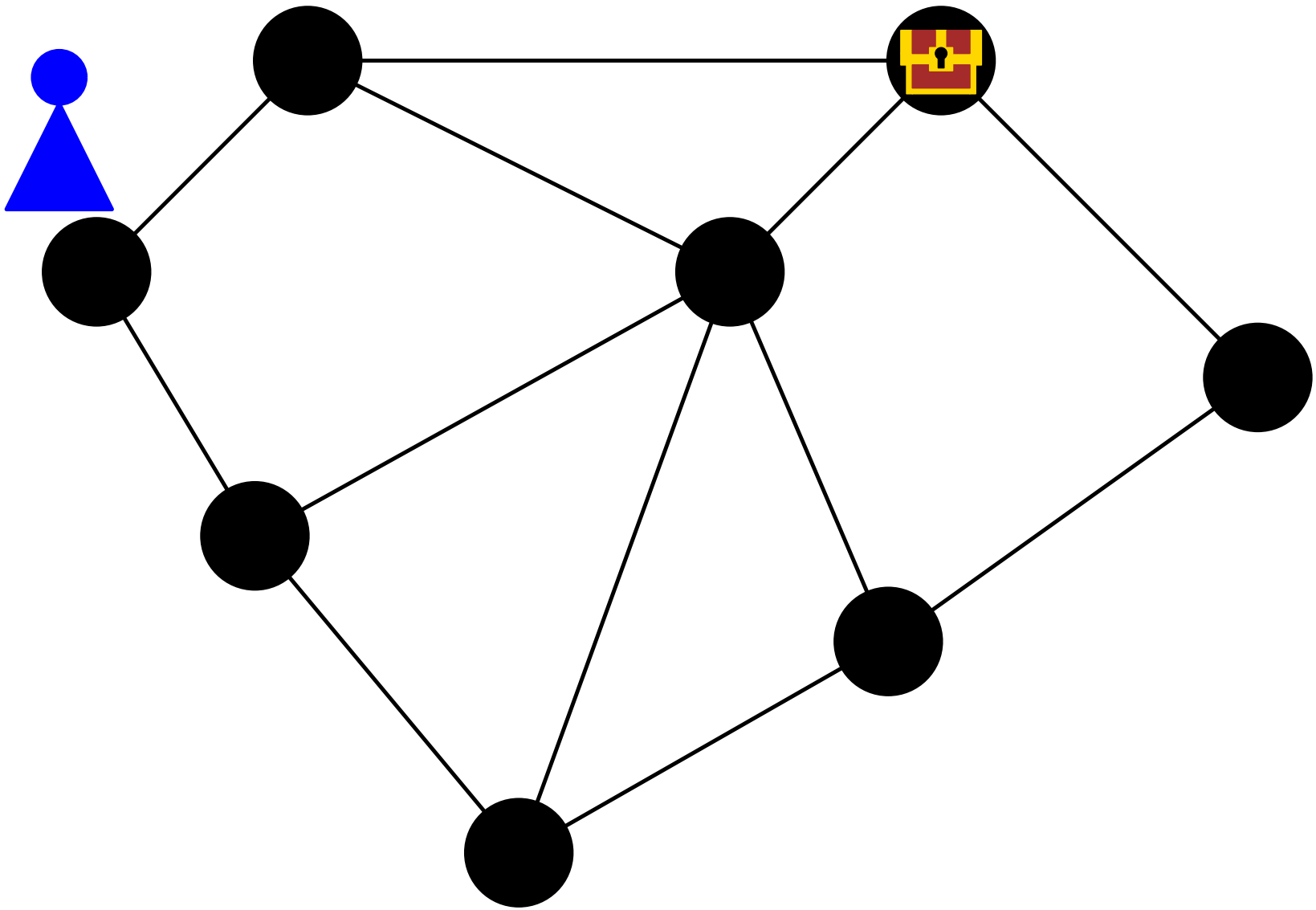
Almost-Optimal Deterministic Treasure Hunt in Arbitrary Graphs

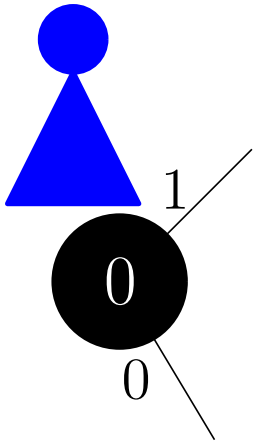
Sébastien Bouchard[1], Yoann Dieudonné[2], **Arnaud Labourel**[3] and Andrzej Pelc[4]

[1] LIP6, Paris, France
[2] MIS Lab., Université de Picardie Jules Verne, Amiens, France
[3] Aix Marseille Univ, CNRS, LIS, Marseille, France
[4] Département d'informatique, Université du Québec en Outaouais, Gatineau, Canada.

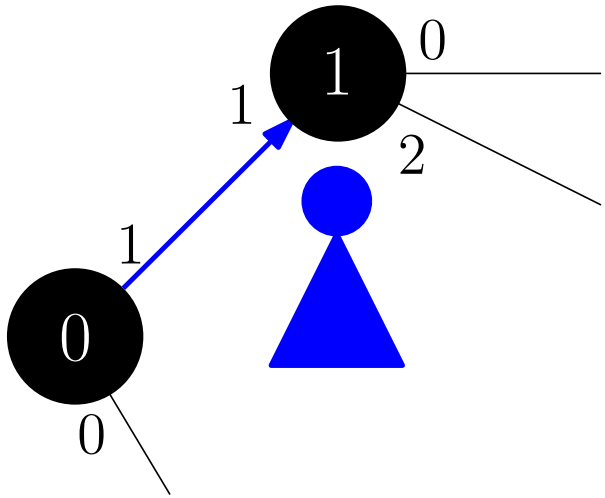
An agent searching a treasure in a graph



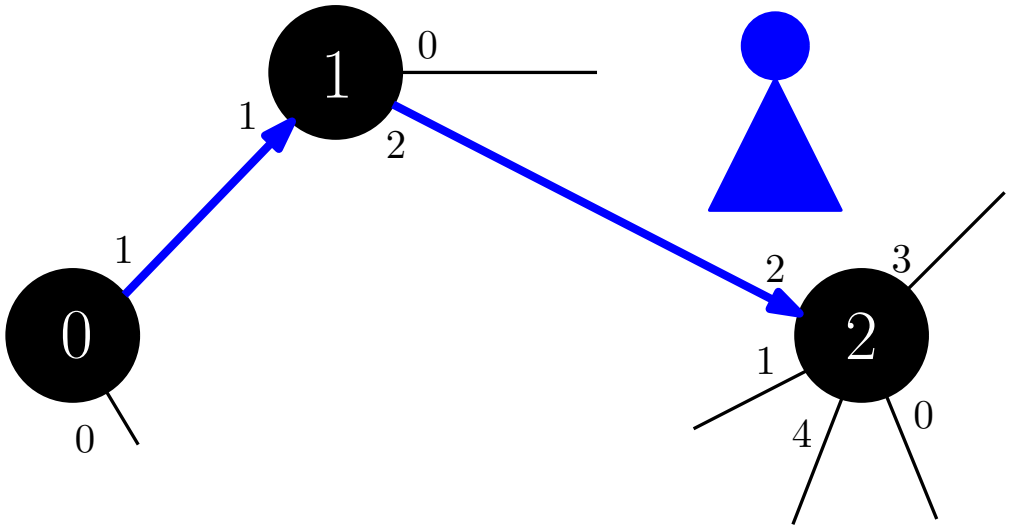
An agent searching a treasure in a graph



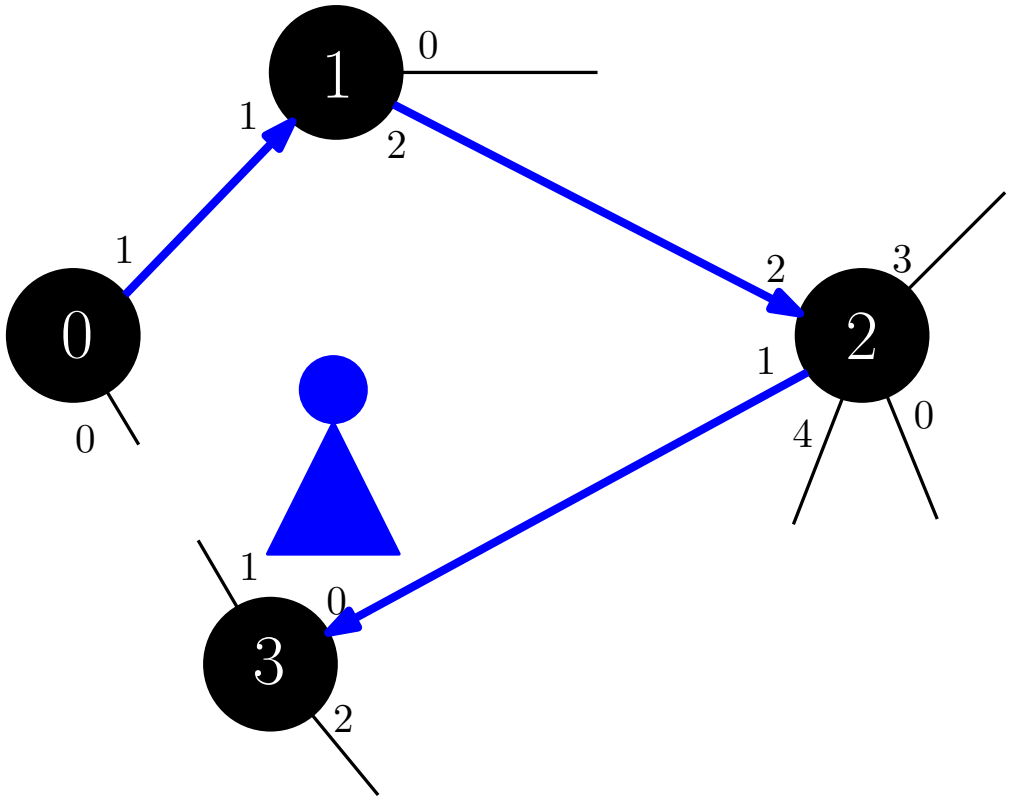
An agent searching a treasure in a graph



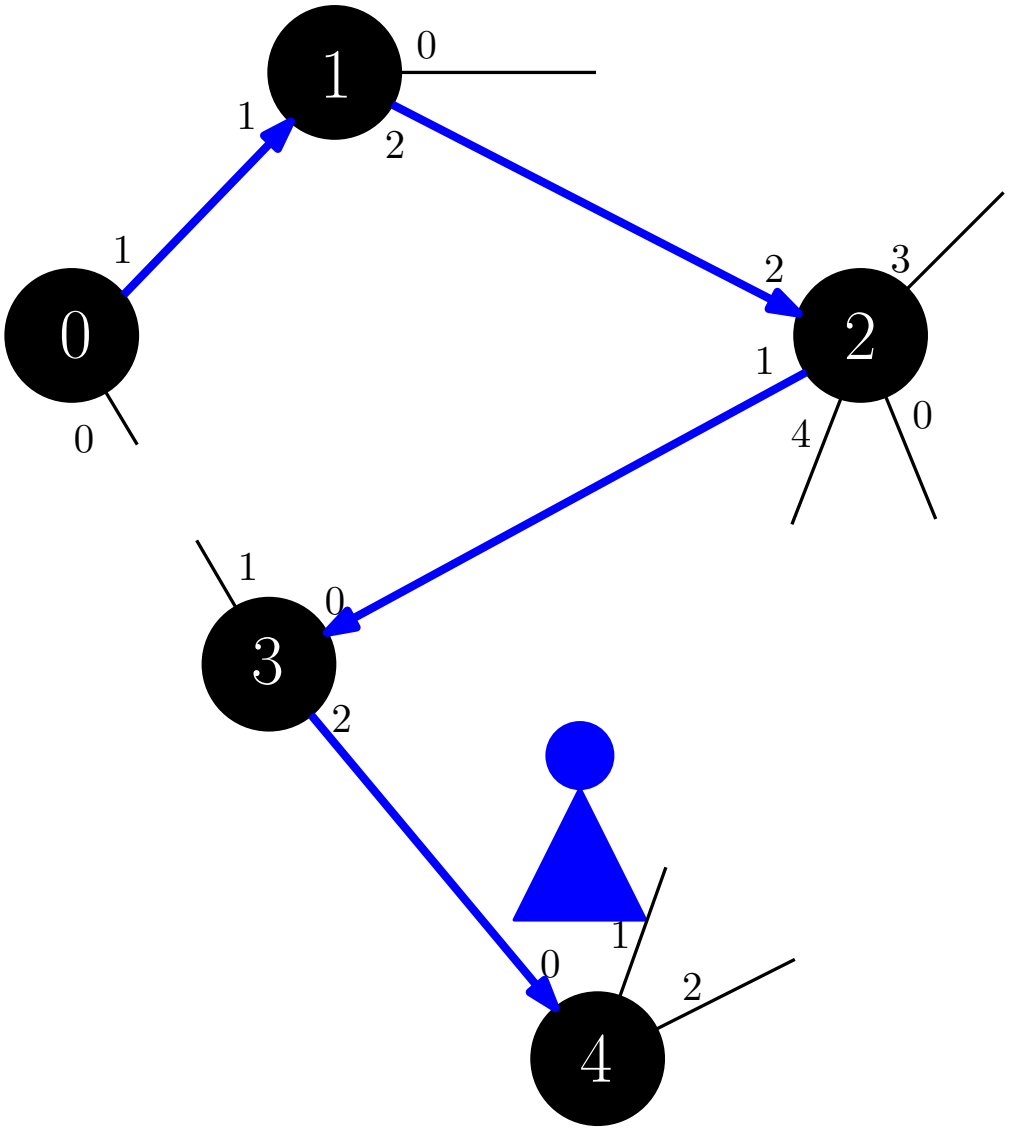
An agent searching a treasure in a graph



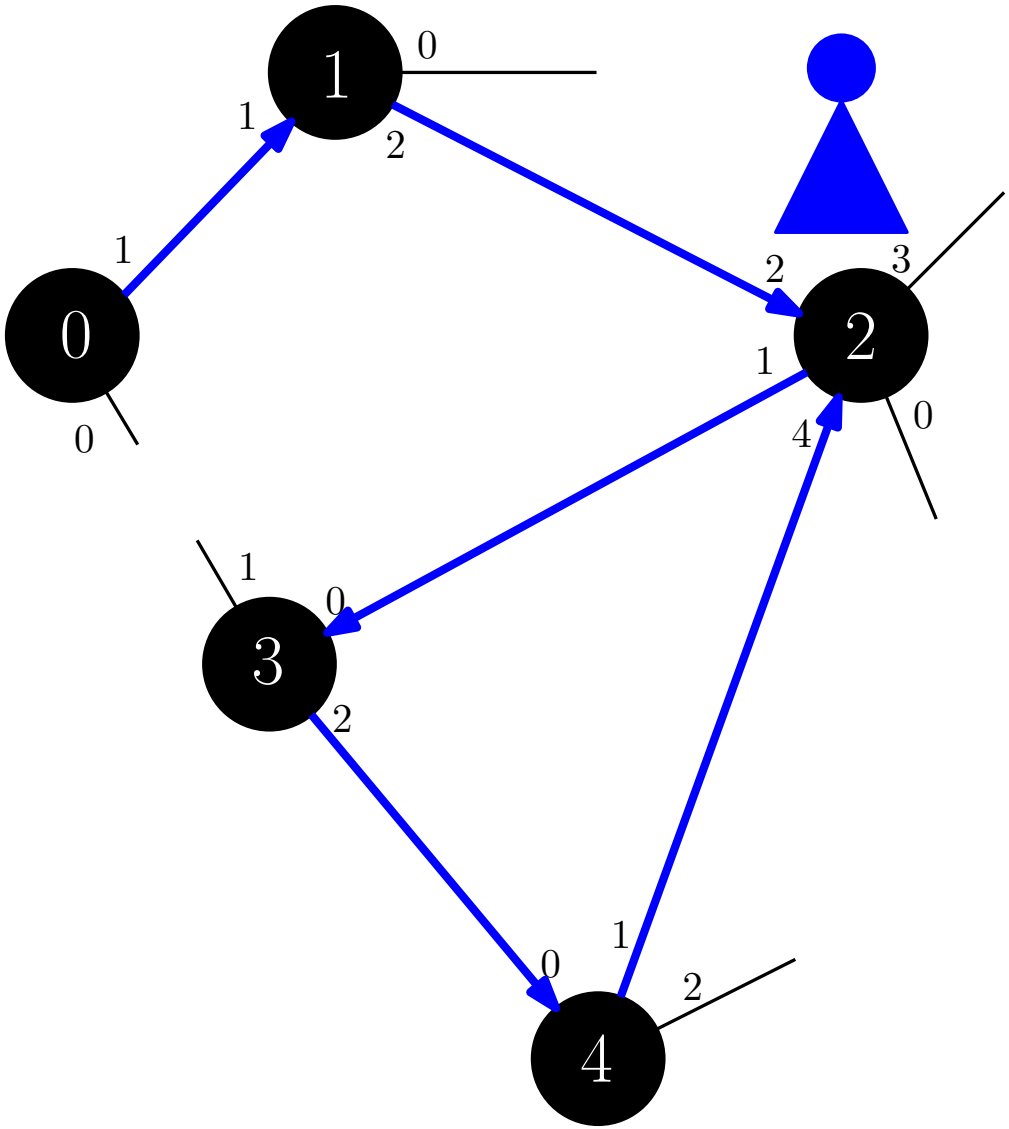
An agent searching a treasure in a graph



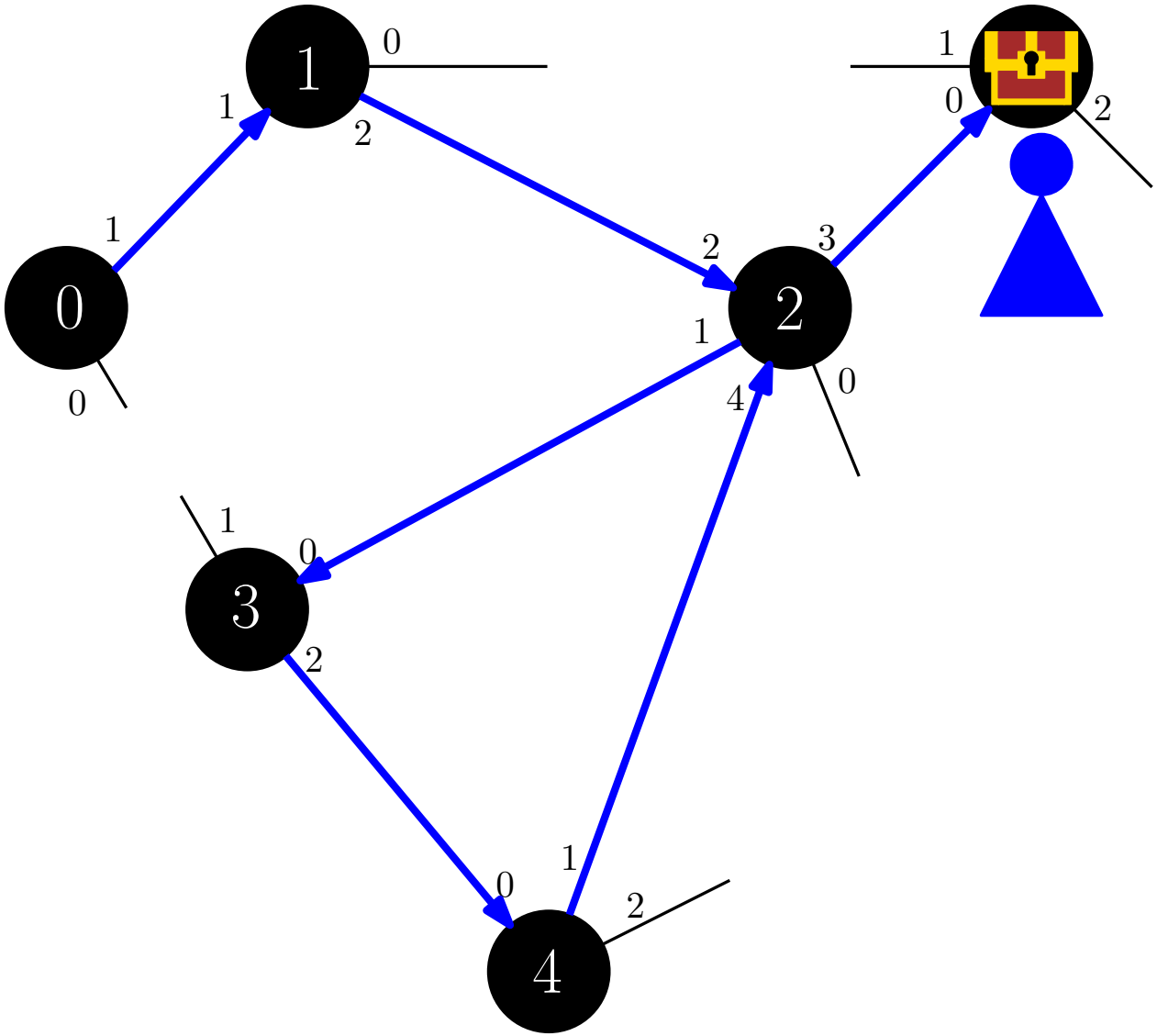
An agent searching a treasure in a graph



An agent searching a treasure in a graph



An agent searching a treasure in a graph



Treasure hunt in graphs

The problem

A mobile agent has to find a treasure inside a network.

- Ports corresponding to edges incident to a node of degree δ are numbered $0, 1, \ldots, \delta - 1$.
- The agent sees the degree and identifier of its current node.
- At each step the agent can choose a port number, move through the corresponding edge and see the incoming port number.

Benchmarking treasure hunt algorithms

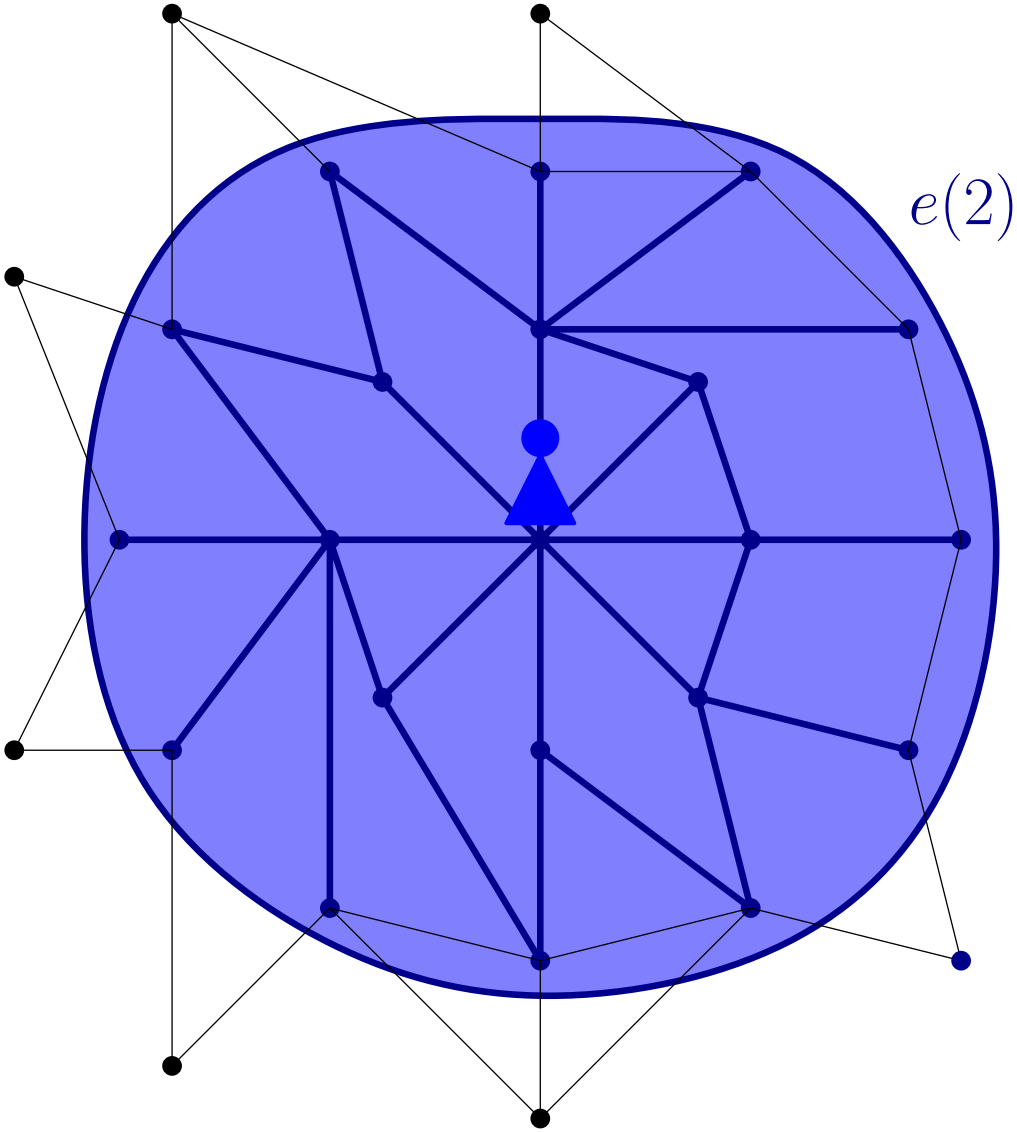
Cost of a treasure hunt algorithm : number of moves performed by the agent before finding the treasure in the worst case.

d : initial distance between the agent and the treasure
$e(r)$: number of edges whose at least one extremity is at distance $\leq r-1$ from the starting position of the agent

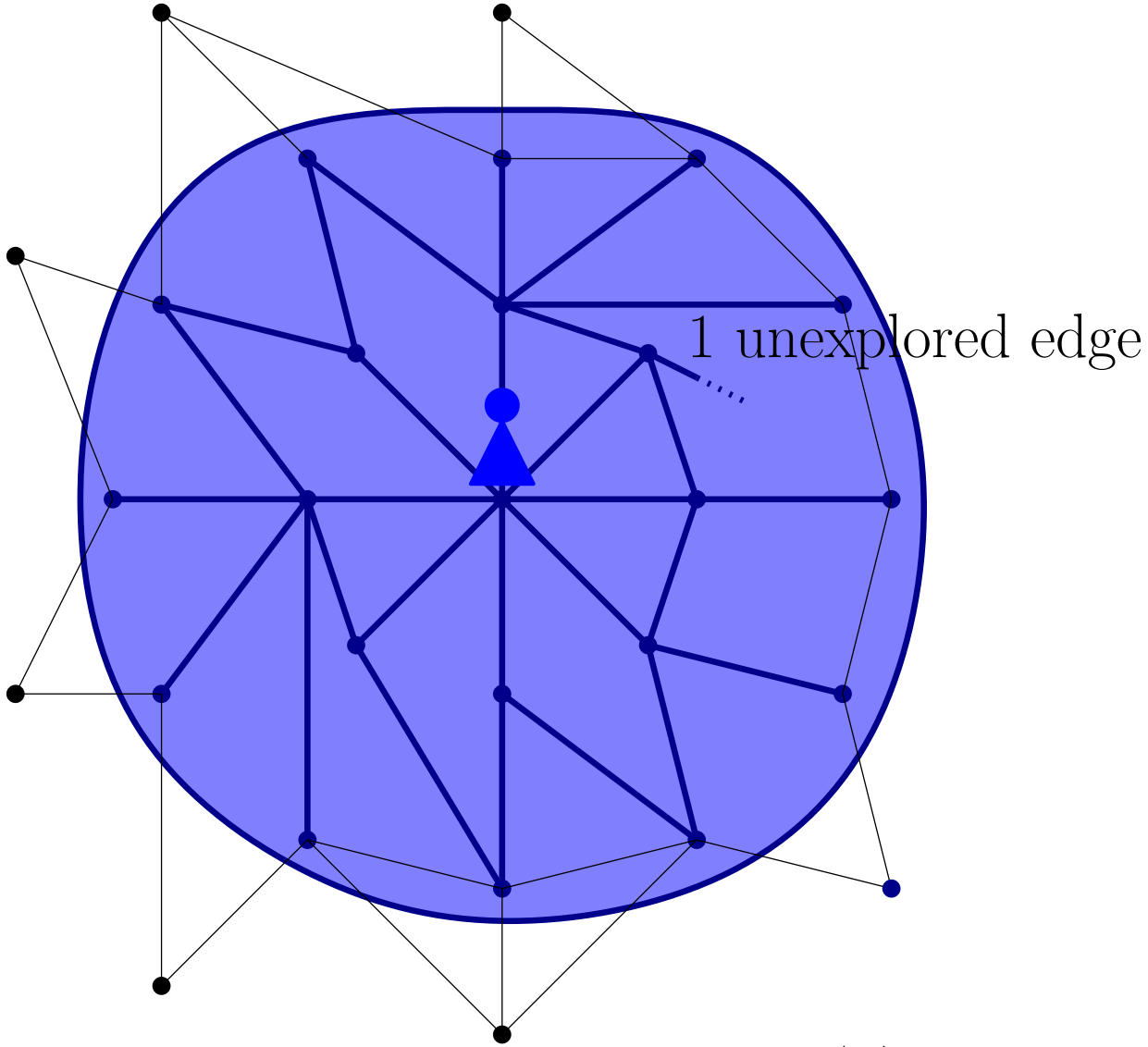
Lower bound for treasure hunt

The cost of any treasure hunt algorithm is at least $e(d)$.

Idea of the lower bound

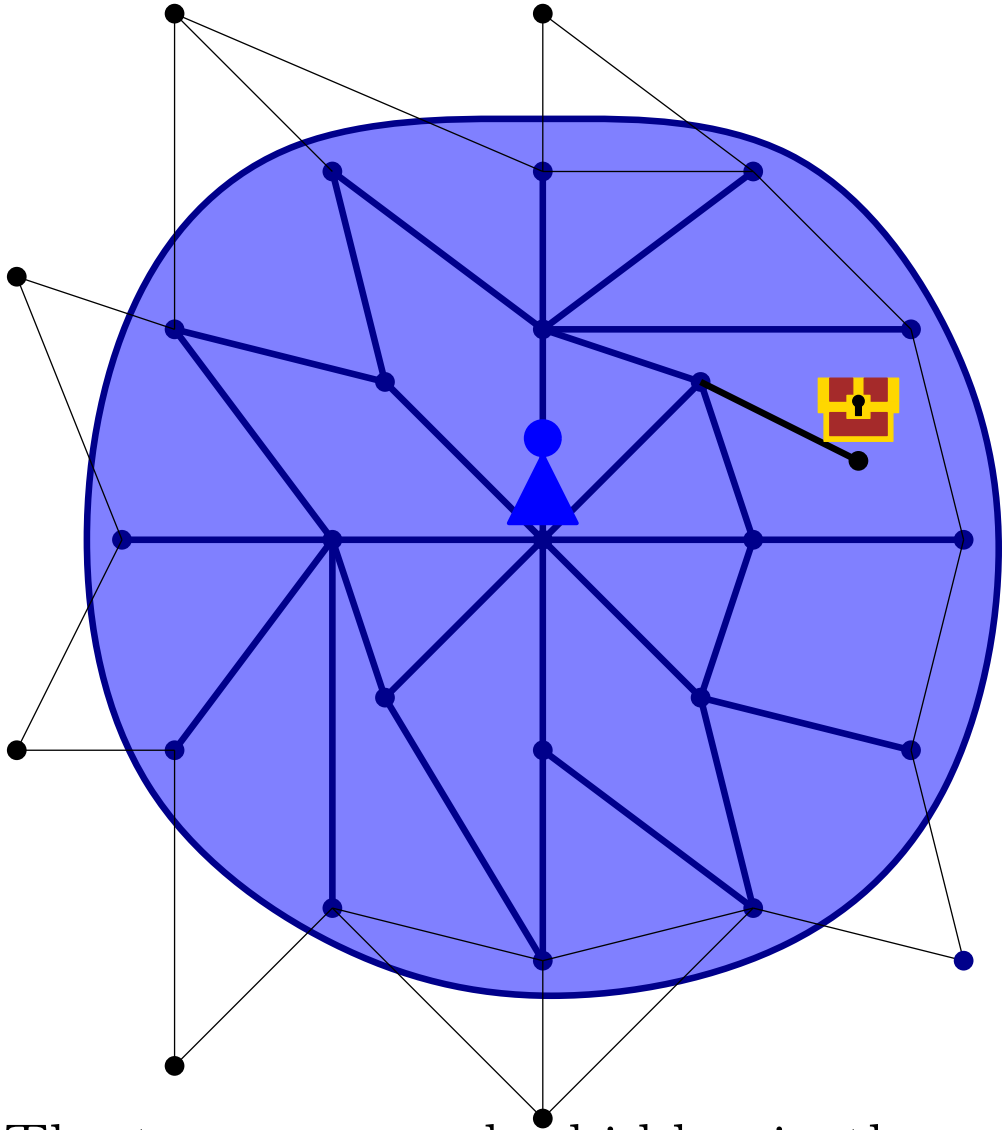


Idea of the lower bound



Assume that the agent has explored $\leq e(d) - 1$ edges

Idea of the lower bound



The treasure can be hidden in the node connected to the unexplored edge

Related works on general graphs

$e(r)$: number of edges whose at least one extremity is at distance $\leq r-1$
$n(r)$: number of nodes at distance $\leq r$

Awerbuch, Betke, Rivest and Singh 1999

Treasure hunt with cost $O(e(d+o(d))+n(d+o(d))^{1+o(1)})$

Duncan, Kobourov and Kuma 2006

Treasure hunt with cost
$O(e((1+\epsilon)d)+n((1+\epsilon)d)/\epsilon)$ for $0<\epsilon<1$.

Almost optimal treasure hunt

Our result (ICALP 2021)

Treasure hunt on general graphs with cost $O(e(d)\log d)$.

Since $d \leq e(d)$ the cost only differs from the best possible cost by a logarithmic factor.
Our result refutes the following conjecture :

Conjecture Awerbuch, Betke, Rivest, Singh 1999

Is it possible (we conjecture not) to find a treasure in time nearly linear in the number of those vertices and edges whose distance to the source is less than or equal to that of the treasure?

Restricted models

Rope-restricted model

- The agent is attached to its starting position by a rope of length L
- The rope unwinds by a length 1 with every forward edge traversal and rewinds by a length of 1 with every backward edge traversal.

Fuel-restricted model

- The agent has a fuel tank of capacity B.
- Each move consumes one unit of fuel.
- The agent can fully replenish its tank at its starting position.

Our algorithm on restricted models

r: radius of the graph

For any constant ϵ, our treasure hunt algorithm can be transformed into:

- an algorithm for rope-restricted model with a rope of length $(1+\epsilon)r$
- an algorithm for fuel-restricted model with a fuel tank of capacity $2(1+\epsilon)r$

Both algorithms have cost $O(e(d)\log d)$.

Easy case: trees with knowledge

For now, we assume that:

- the graph is a tree,
- we know the values $e(i)$ for every i.

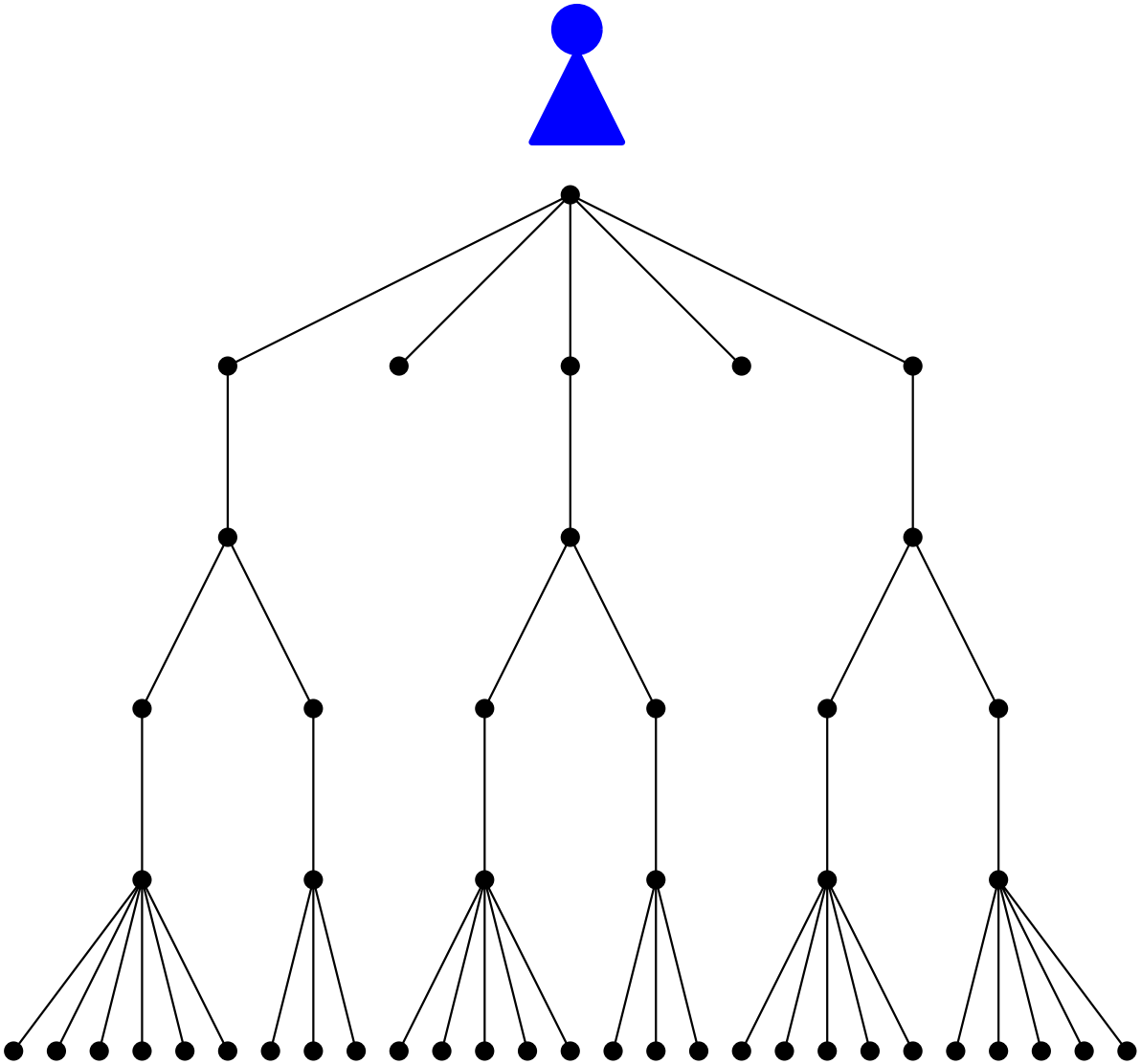
Idea of the algorithm

Iteratively explore balls that grows exponentially.

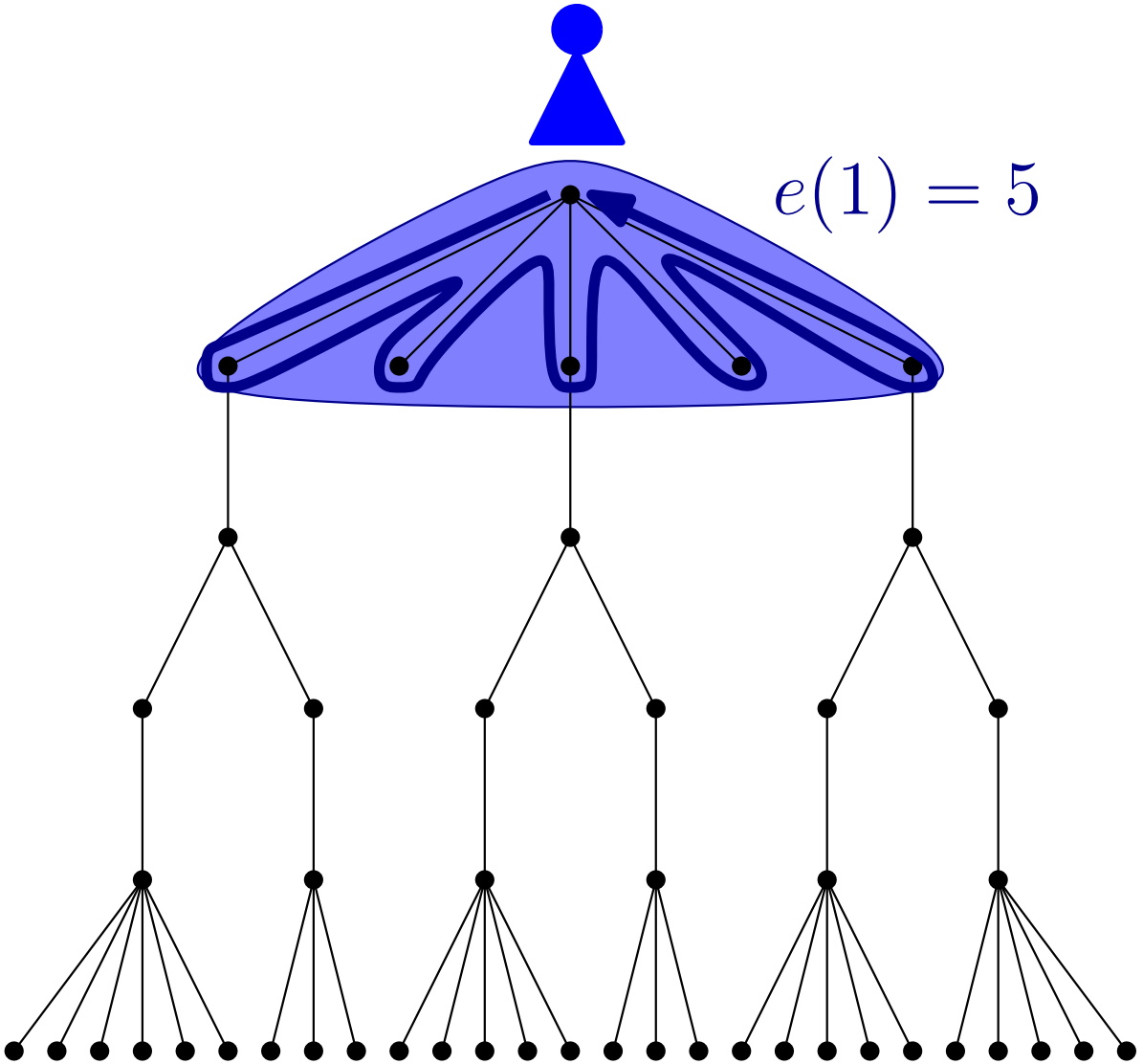
Algorithm

- Compute values h_i such that:
  - $h_0 = 1$
  - $h_{i+1} = \min\{h \mid e(h) \geq 2e(h_i)\}$ for all $i \geq 0$
- Iteratively explore balls of radius h_i for $i = 0, 1, 2, \ldots$

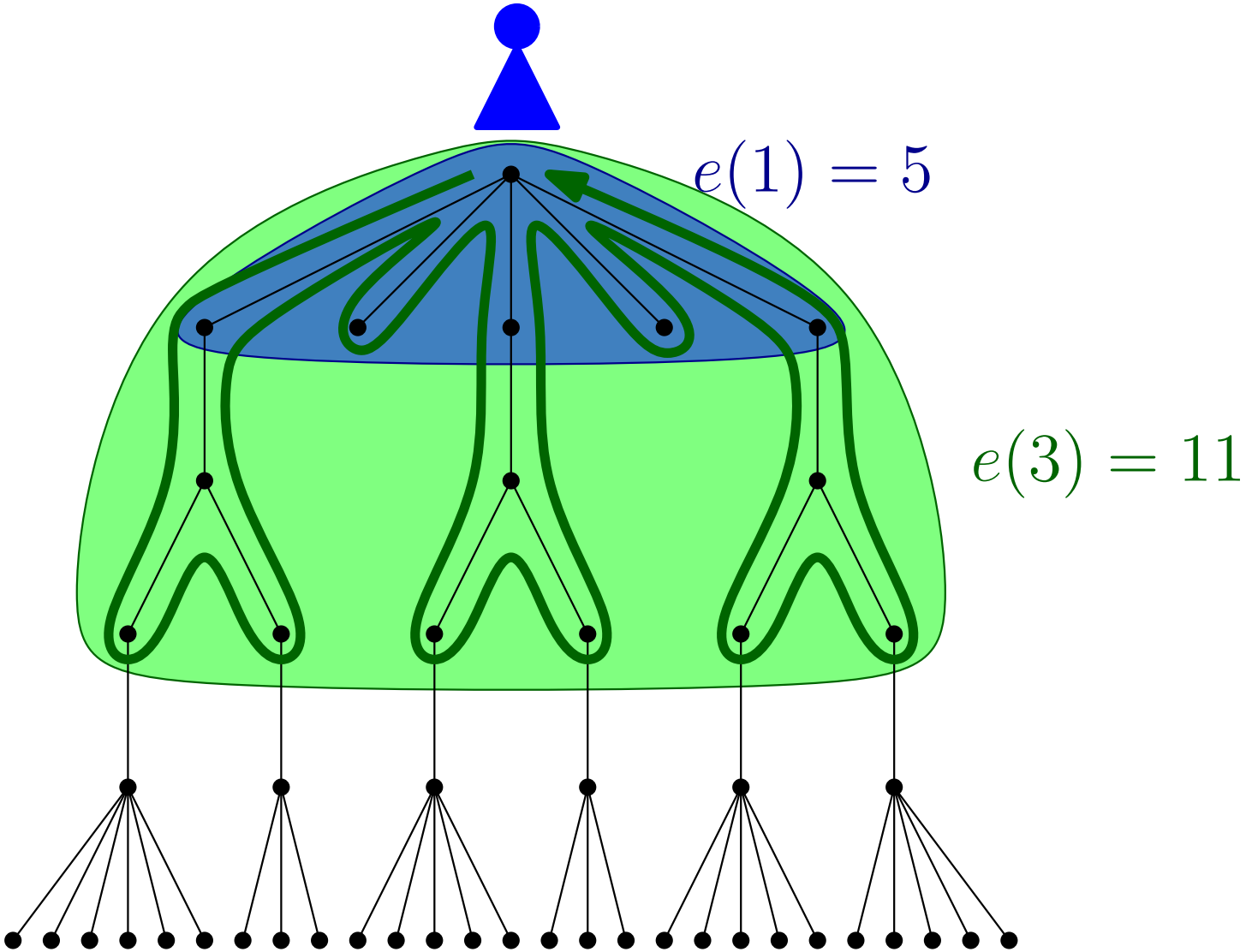
Trees with knowledge



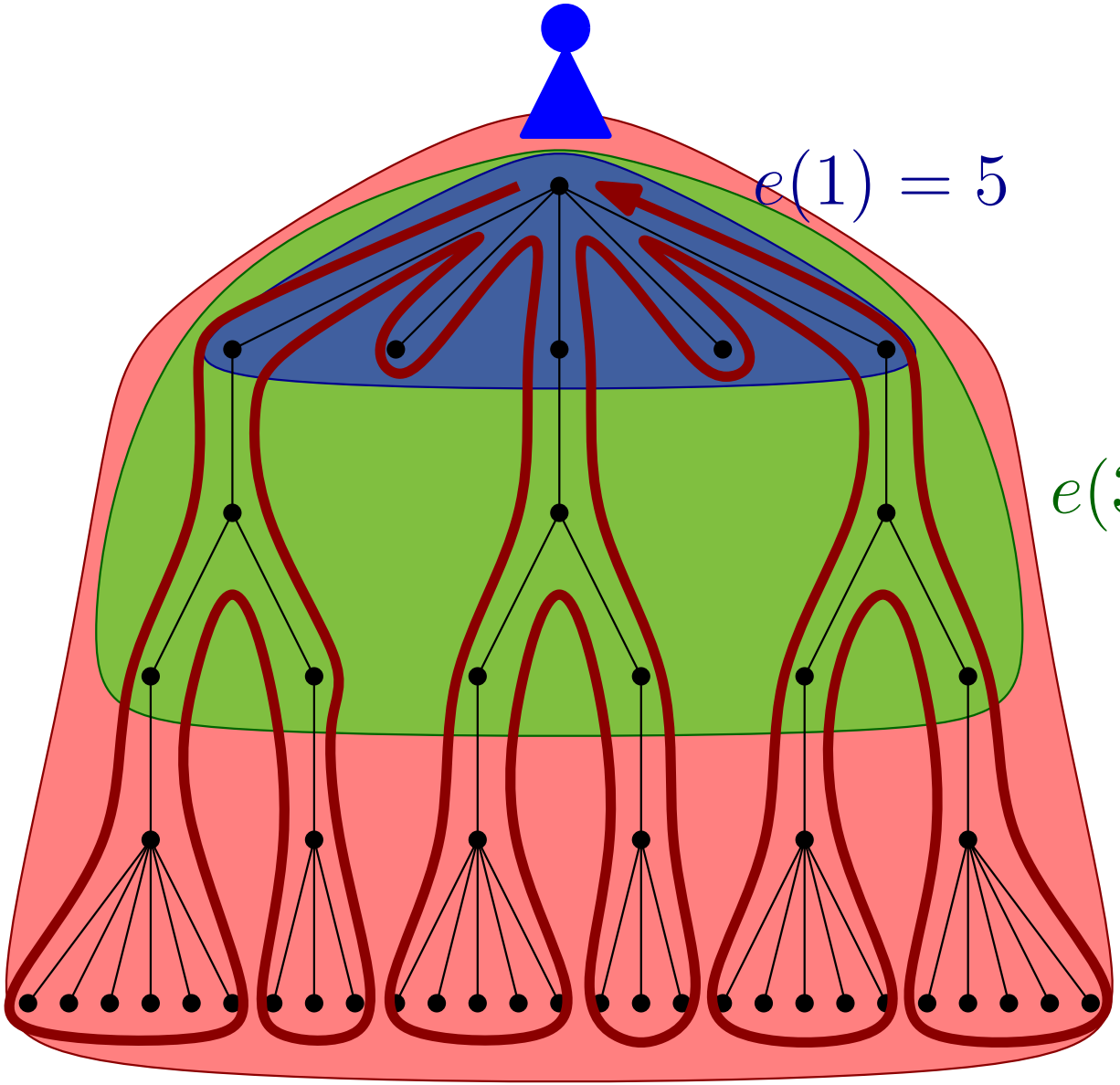
Trees with knowledge



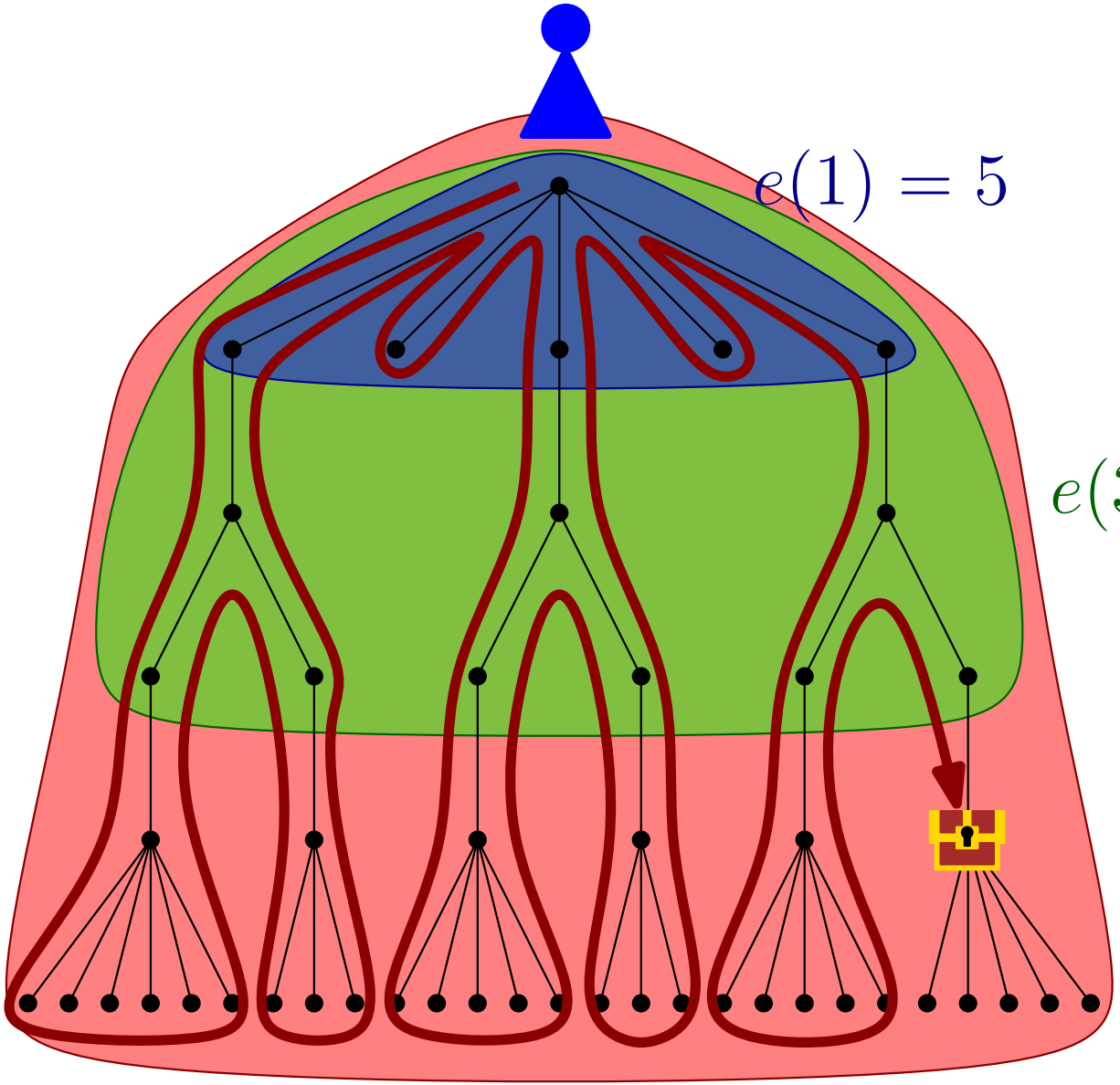
Trees with knowledge



Trees with knowledge



Trees with knowledge



Trees with knowledge (second attempt)

Problem with the first attempt

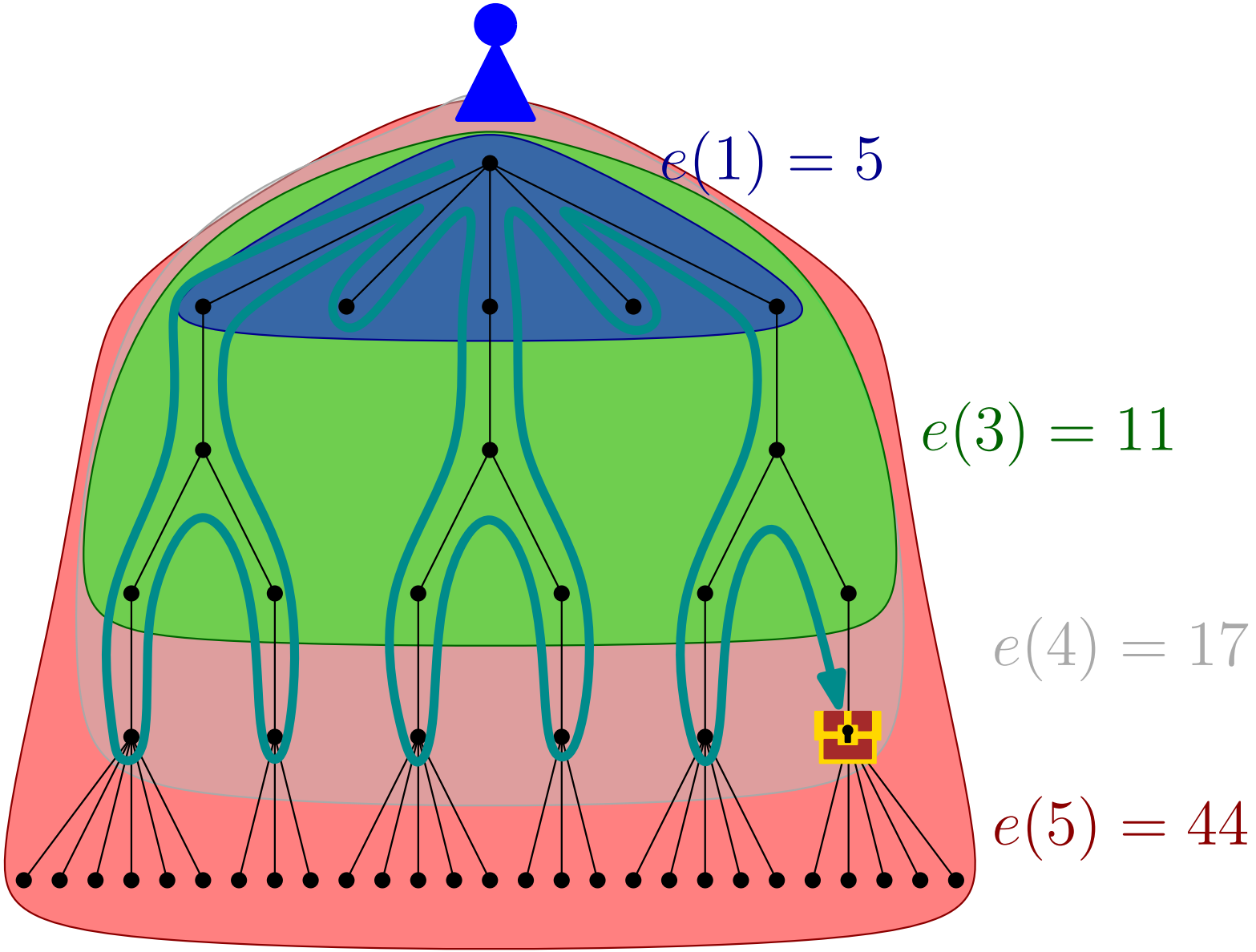
If the balls grow too fast $e(h_{i+1}) >> e(h_i)$ and the difference of their radius is ≥ 2 then the treasure can be hidden at distance $h_{i+1} - 1$ and the cost is $\Omega(e(h_{i+1})) >> e(h_{i+1} - 1)$.

$\Rightarrow$ Add an exploration at depth $h_{i+1} - 1$ in this case.

New algorithm: pick in order for h_{i+1}

1. Pick $h_i + 1$ if $e(h_i + 1) \geq 2e(h_i)$
2. Pick (if it exists) the minimum radius h such that $2e(h_i) \leq e(h) \leq 4e(h_i)$
3. Pick the maximum radius h such that $e(h) < 2e(h_i)$

Trees with knowledge (second attempt)



Trees with knowledge (second attempt)

Cost of the new algorithm

The treasure is found at cost $16e(d)$.

Key ideas of the proof :

- The cost of exploring the last ball (the one in which the treasure is found) is at most four times $e(d)$.
- The size of the explored balls grows exponentially (either $\times 2$ in one iteration of the loop or $\times 4$ in two iterations), hence the total cost is proportional to the cost of the exploration of the last ball.

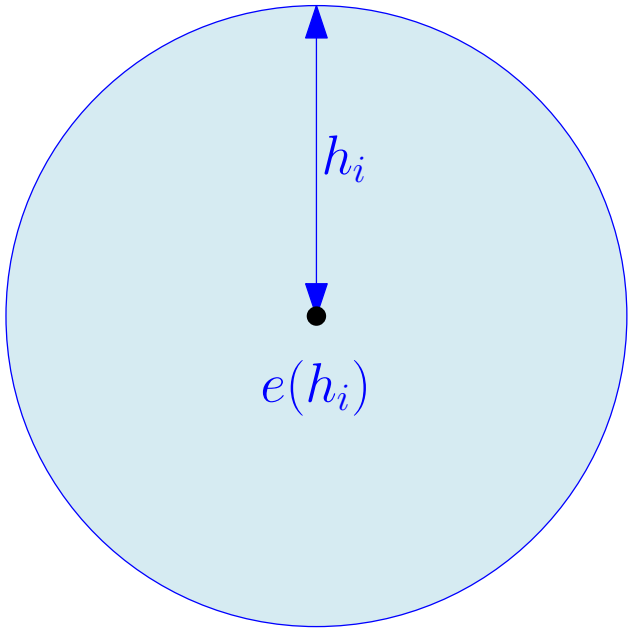
Trees without knowledge

Idea of the algorithm

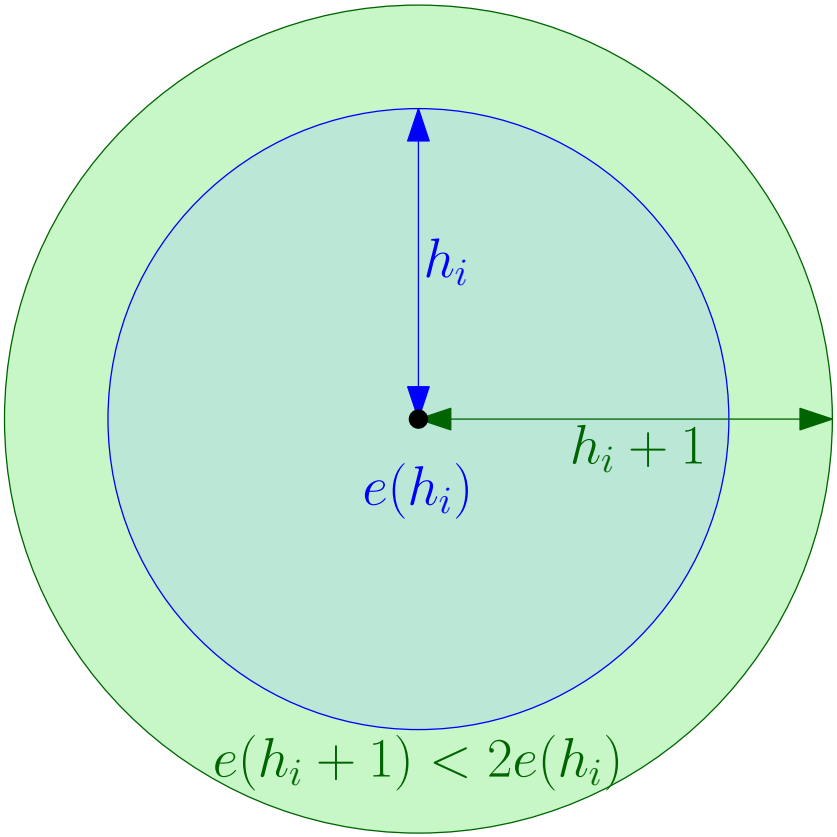
Assuming that the ball of radius h_i has been explored. Search the next radius h_{i+1} by trying to explore balls of radius between $h_i + 1$ and $2h_i$ with a binary search.

Exploration will be done with a Budgeted DFS that tries to explore the ball of radius h but will abort and backtrack if the number of explored edges is $> 4e(h_i)$ and $h > h_i + 1$.

Treasure hunt for trees

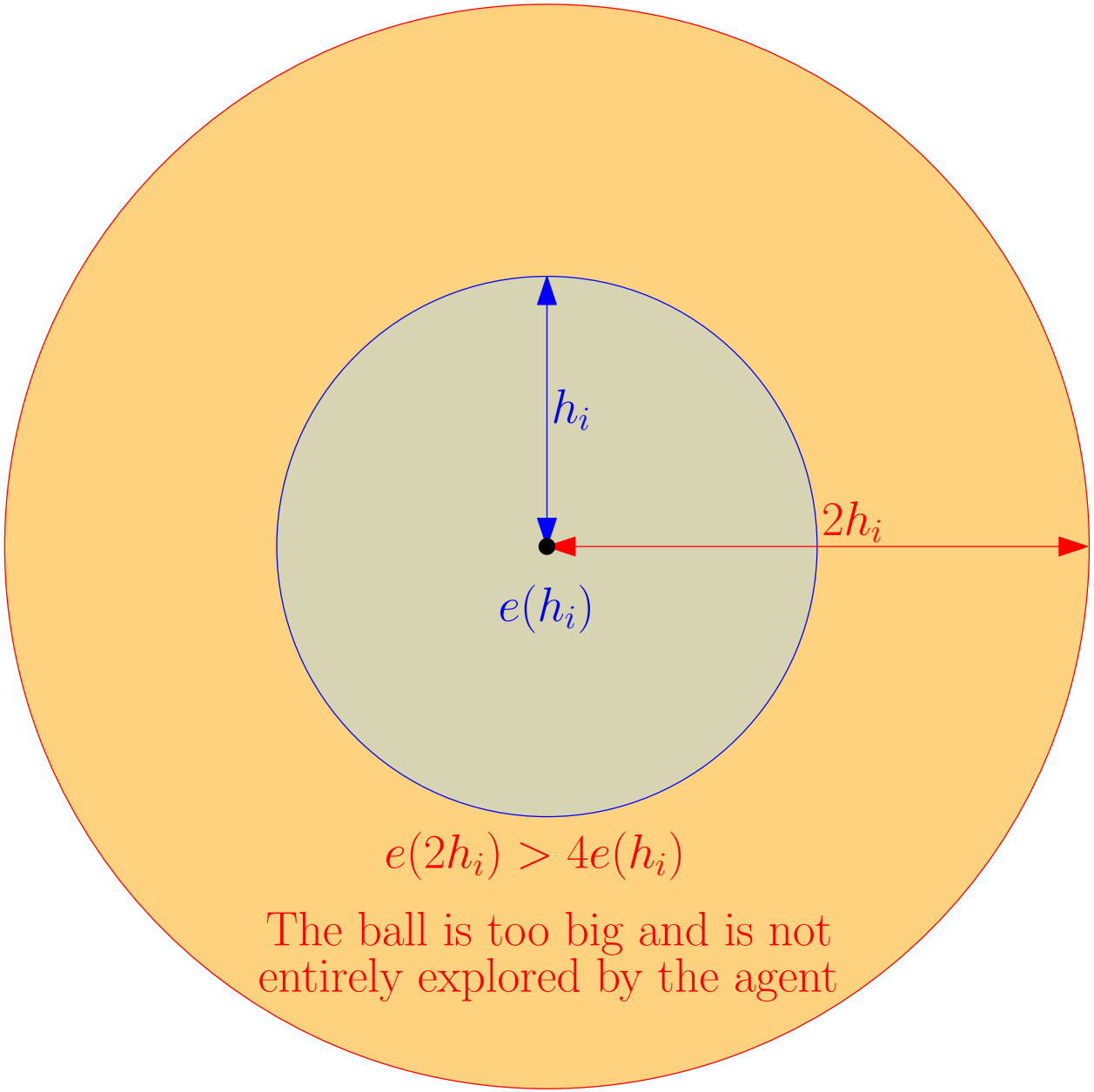


Treasure hunt for trees

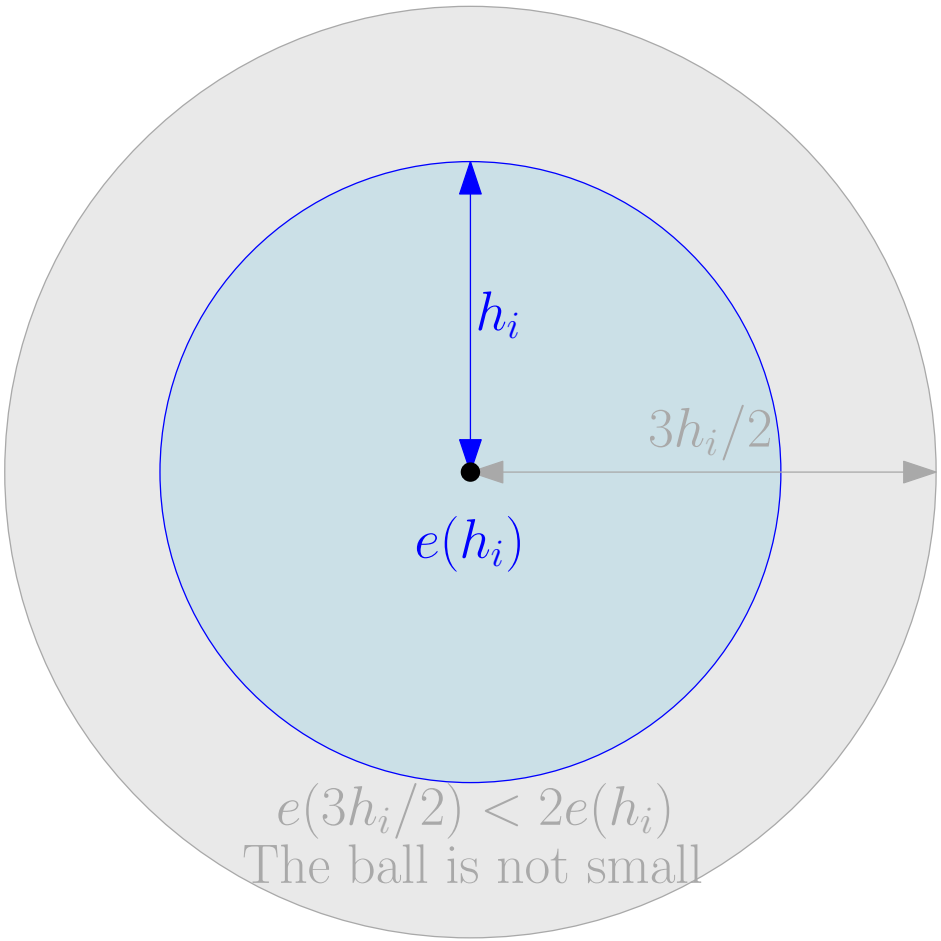


The ball is too small

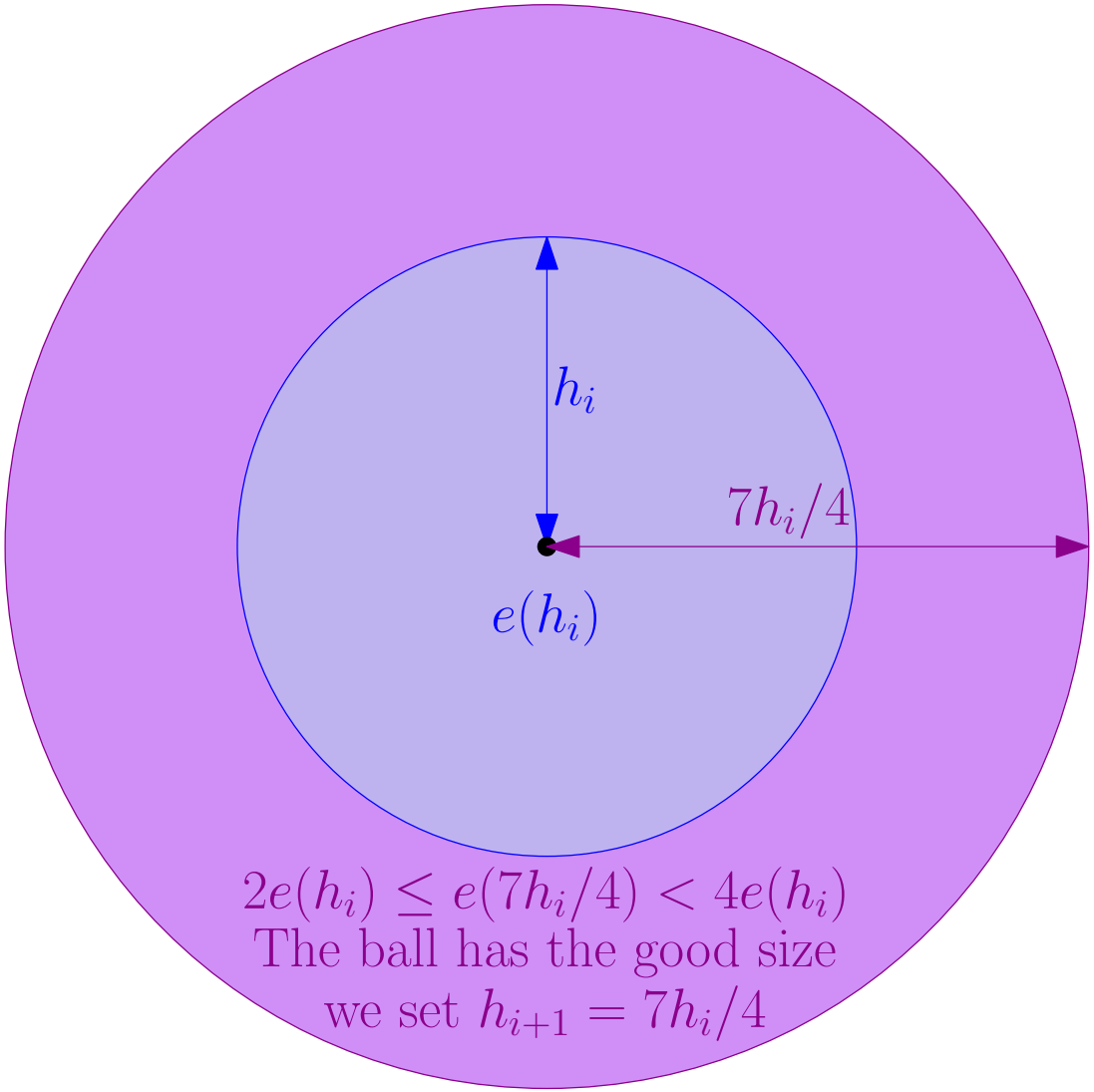
Treasure hunt for trees



Treasure hunt for trees



Treasure hunt for trees



Cost of treasure hunt for trees

Cost of the new algorithm

The treasure is found at cost $O(\log(d).e(d))$.

- We count the cost of explorations with doubling radius separately: there are at most $\log d$ such explorations and the cost of each one is $O(e(d))$ and so their total cost is $O(\log(d).e(d))$
- Other $e(h_i)$ grows exponentially, hence the total cost is proportional to the cost of the last iteration.
- The cost of the last iteration is $O(\log(d).e(d))$ because there are at most $\log(d)$ tries of explorations at each iteration and each one has a cost in $O(e(d))$.

General case: arbitrary graph

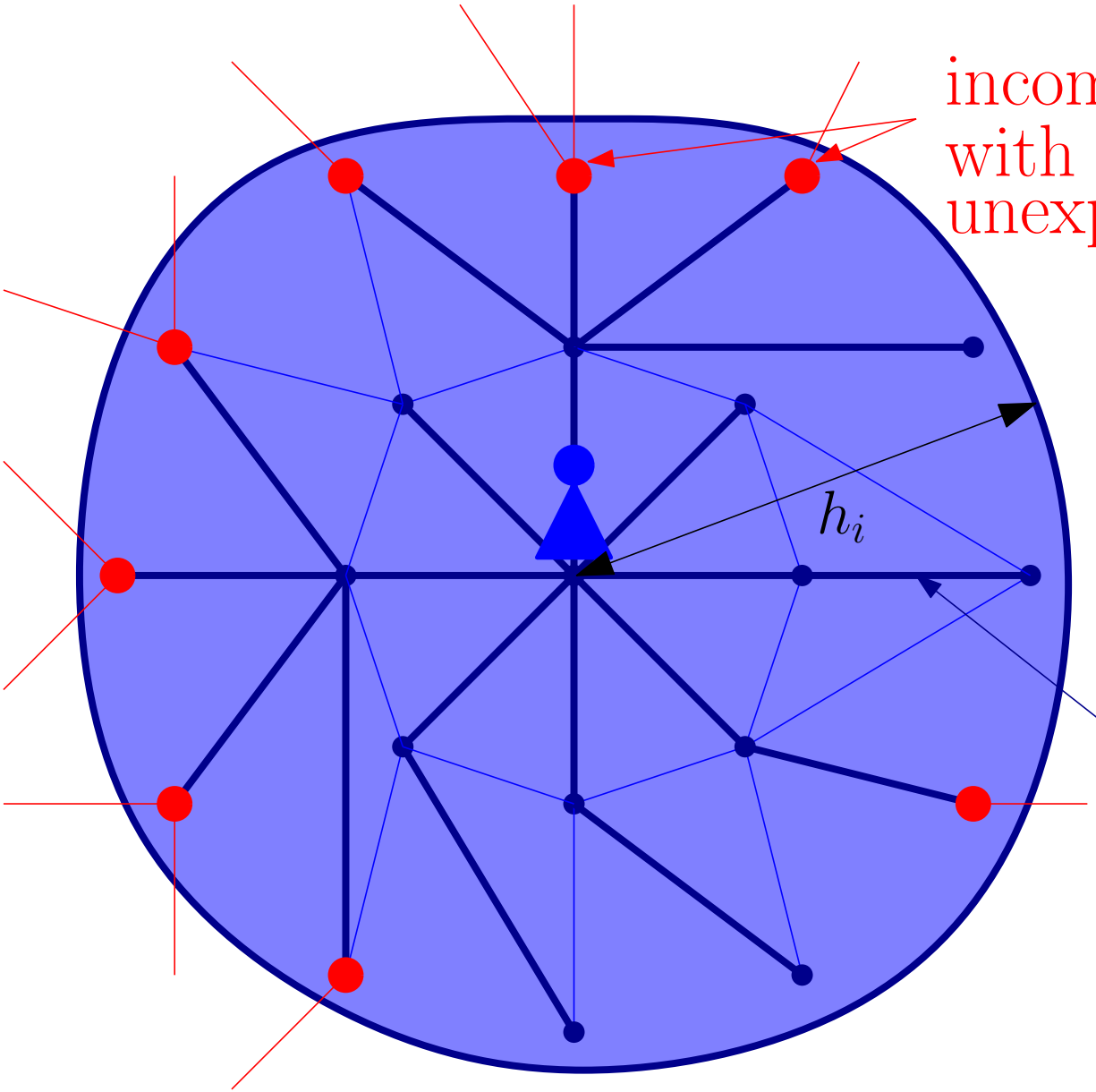
We assume by induction that we already have explored the ball of radius h_i and we want to explore all nodes at distance $\leq h_i + l$.

We maintain:

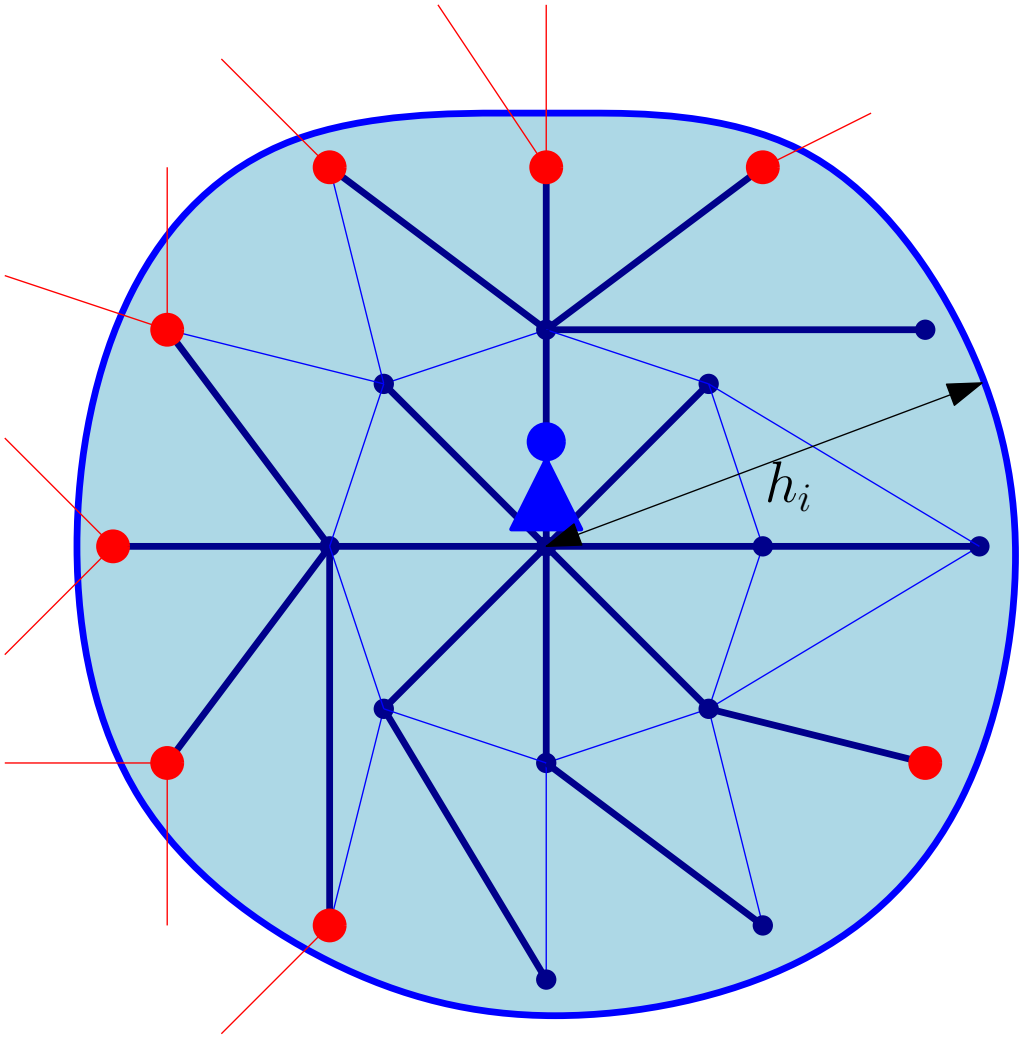
- A list of incomplete nodes, i.e., node at distance $< h_i + l$ with incident unexplored edges
- A tree containing all the incomplete nodes

The agent will visit each incomplete node u in some DFS order of the tree and explore all unexplored nodes at distance $\leq l$ of u.
Like with the treasure-hunt on trees, the exploration will have a budget and abort if there are too many edges.

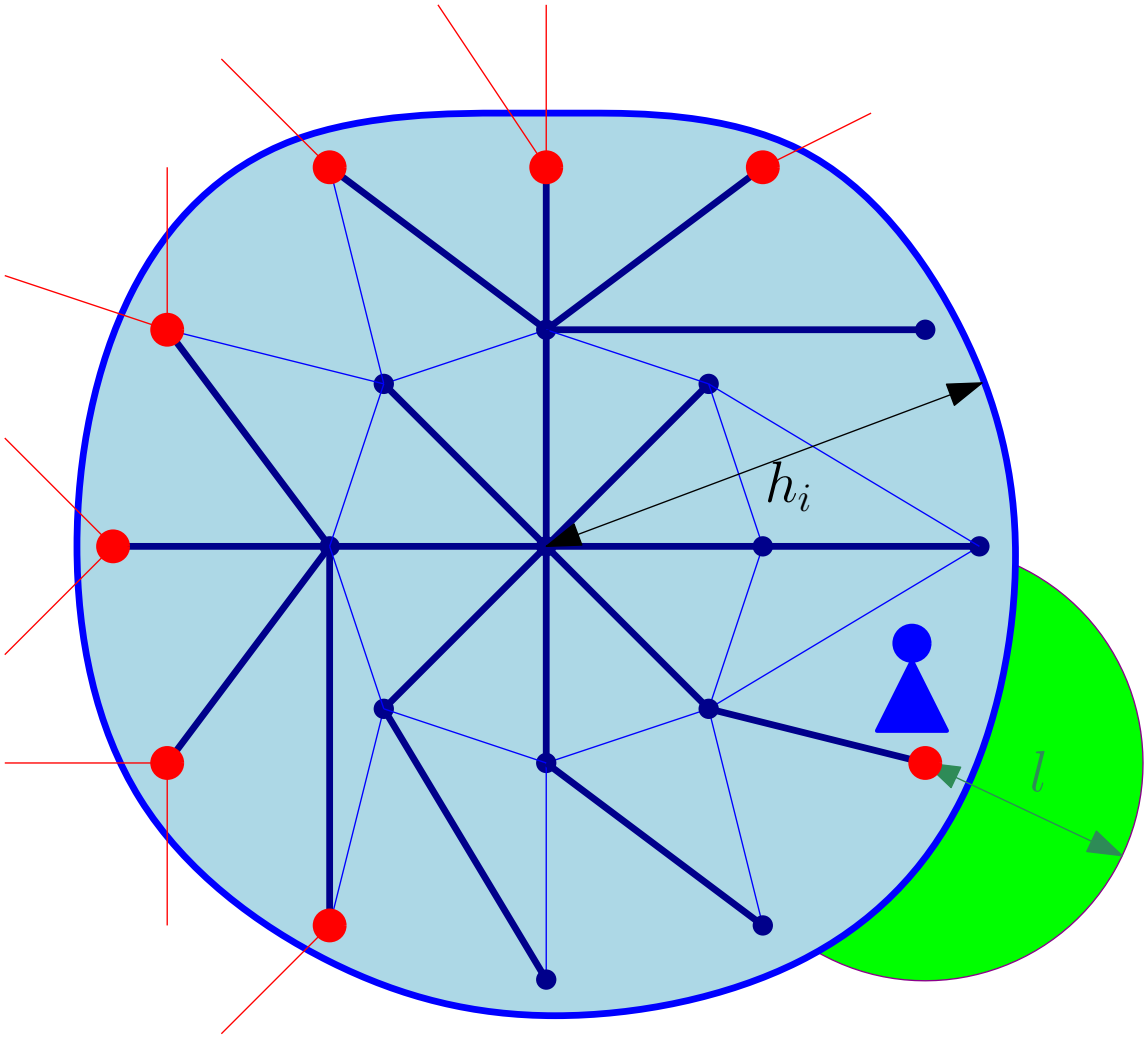
Spanning tree and incomplete nodes



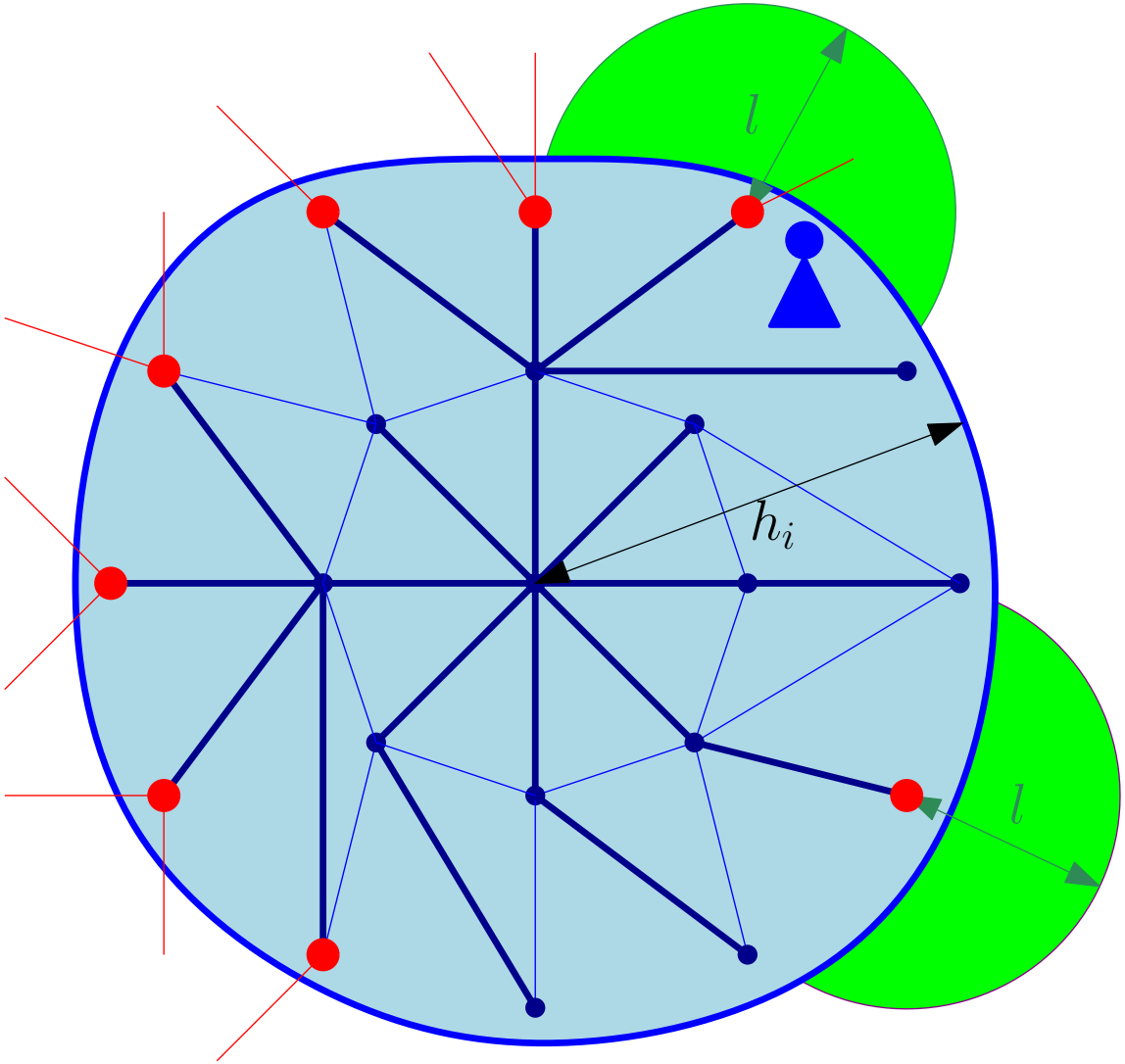
Exploring the ball of radius $h_i + l$



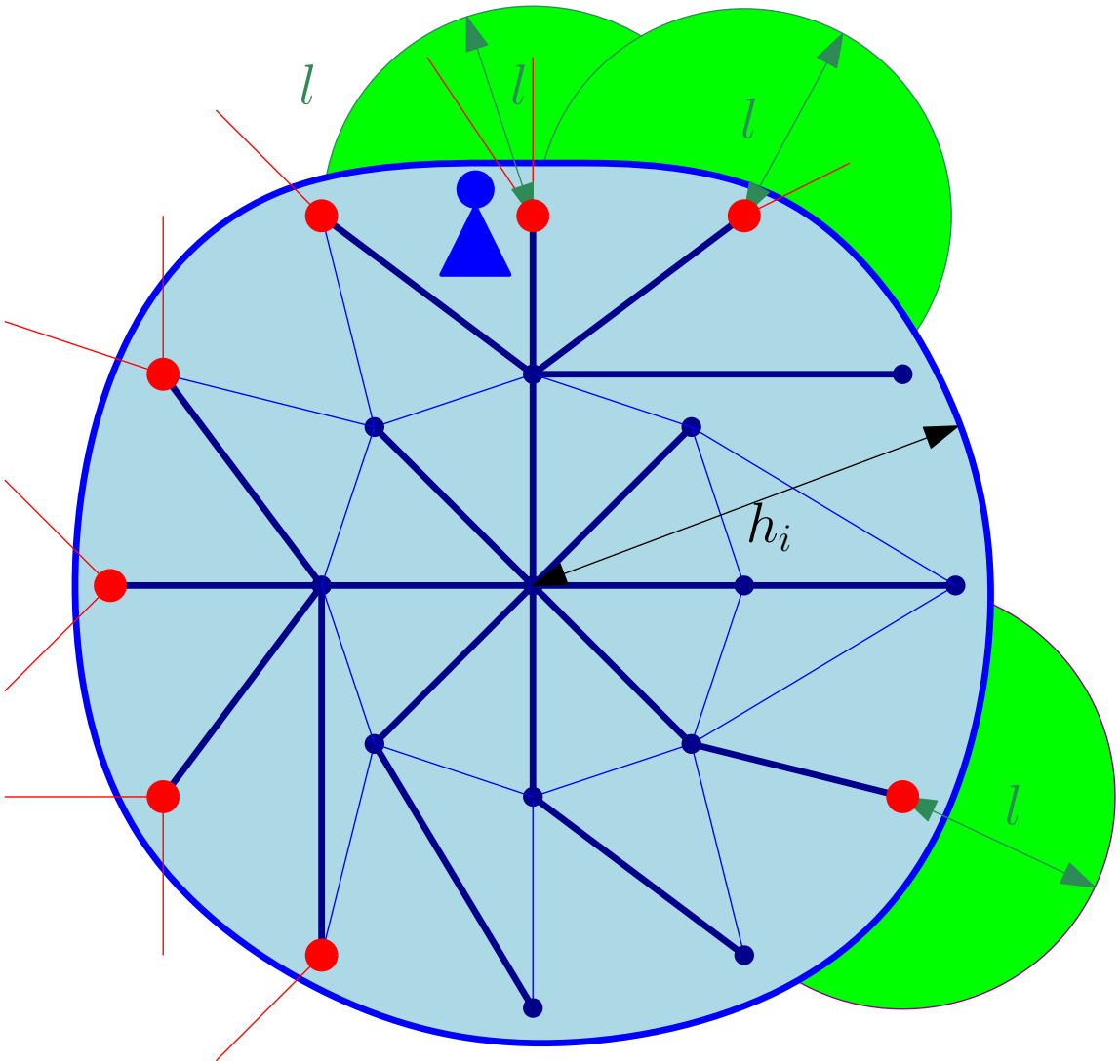
Exploring the ball of radius $h_i + l$



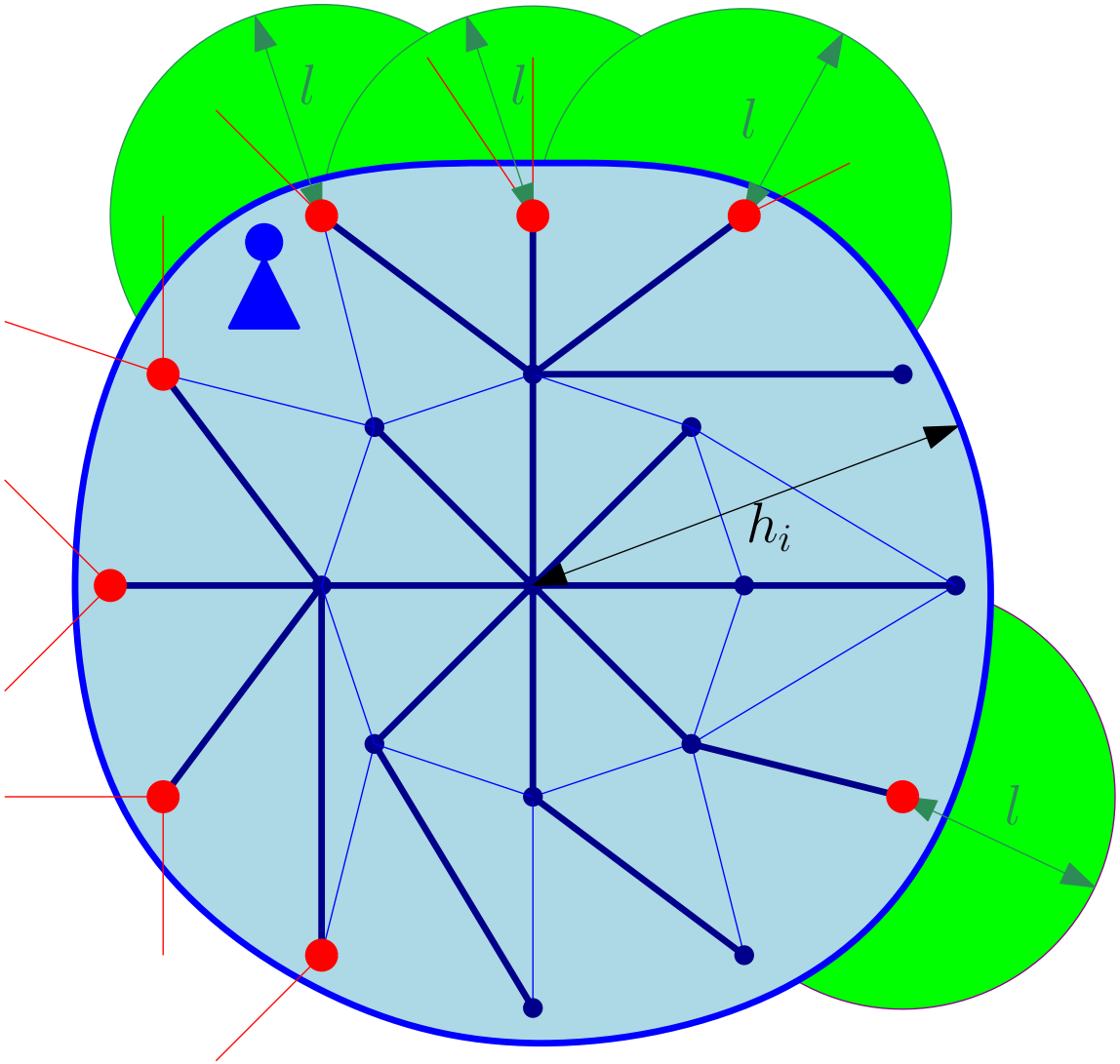
Exploring the ball of radius $h_i + l$



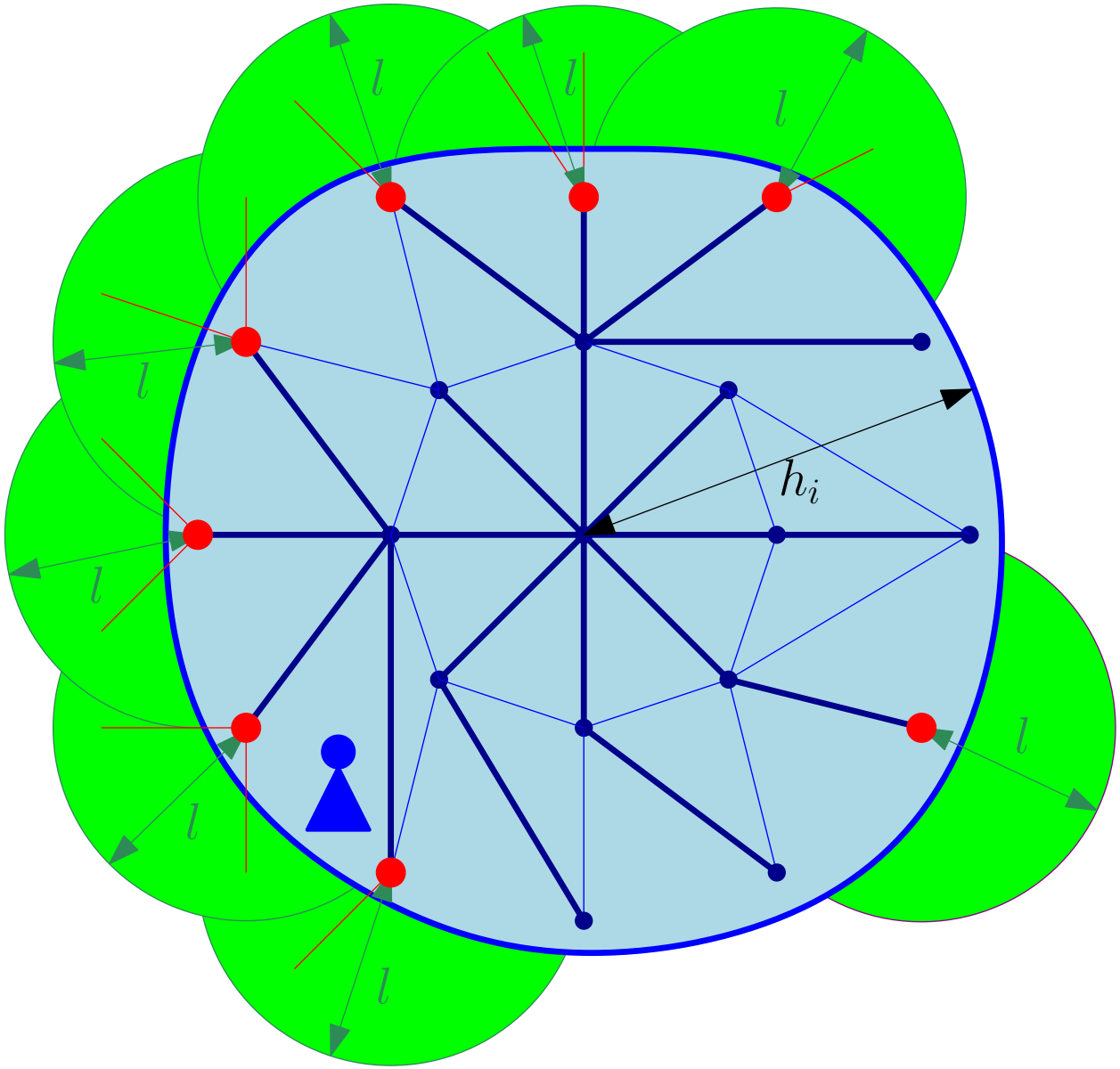
Exploring the ball of radius $h_i + l$



Exploring the ball of radius $h_i + l$

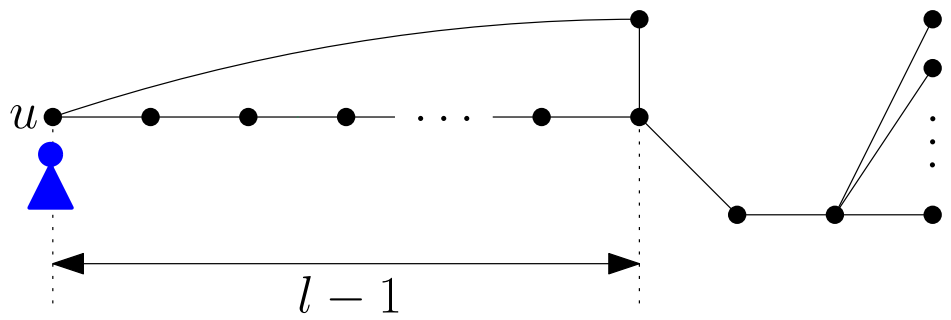


Exploring the ball of radius $h_i + l$



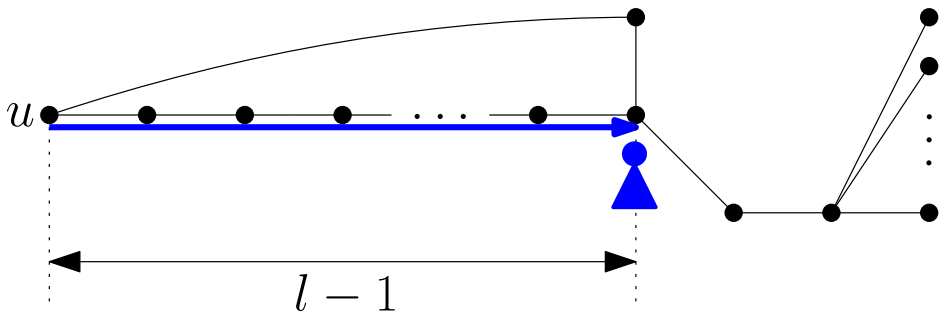
Exploring in general graphs

Unlike the case of trees, it is difficult to explore nodes at distance d since the exact distance is not known by the agent.
Example of exploration in which the agent does not explore everything at distance l if it is not allowed to traverse the same edge twice:



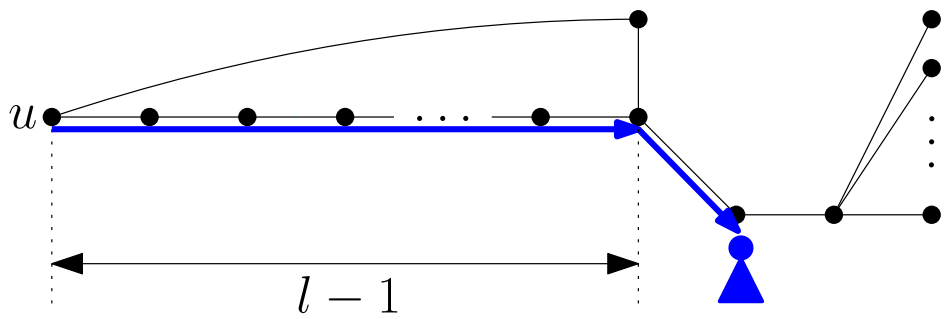
Exploring in general graphs

Unlike the case of trees, it is difficult to explore nodes at distance d since the exact distance is not known by the agent.
Example of exploration in which the agent does not explore everything at distance l if it is not allowed to traverse the same edge twice:



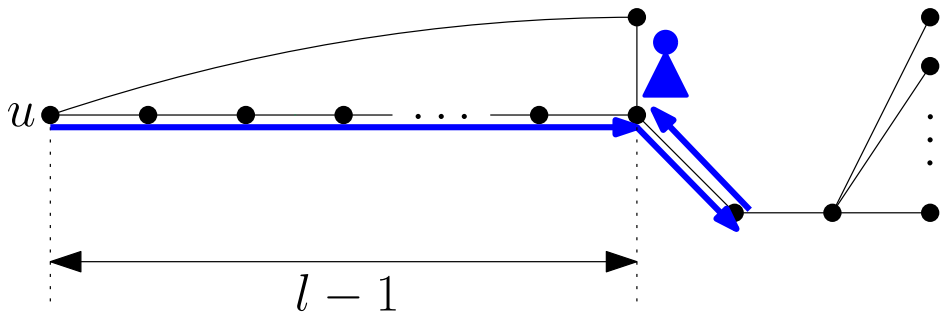
Exploring in general graphs

Unlike the case of trees, it is difficult to explore nodes at distance d since the exact distance is not known by the agent.
Example of exploration in which the agent does not explore everything at distance l if it is not allowed to traverse the same edge twice:



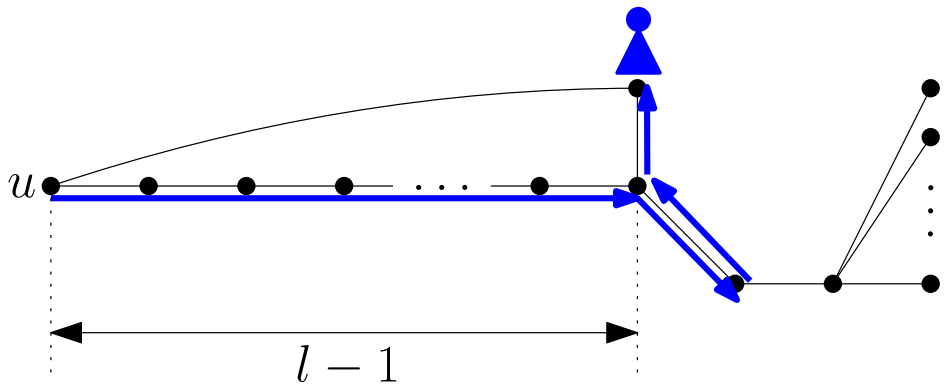
Exploring in general graphs

Unlike the case of trees, it is difficult to explore nodes at distance d since the exact distance is not known by the agent.
Example of exploration in which the agent does not explore everything at distance l if it is not allowed to traverse the same edge twice:



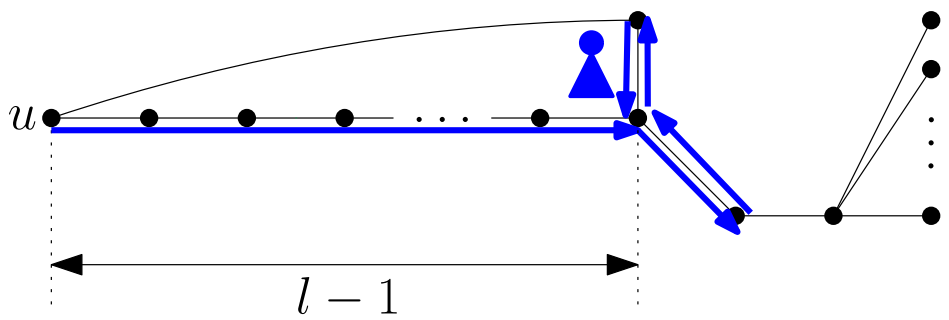
Exploring in general graphs

Unlike the case of trees, it is difficult to explore nodes at distance d since the exact distance is not known by the agent.
Example of exploration in which the agent does not explore everything at distance l if it is not allowed to traverse the same edge twice:



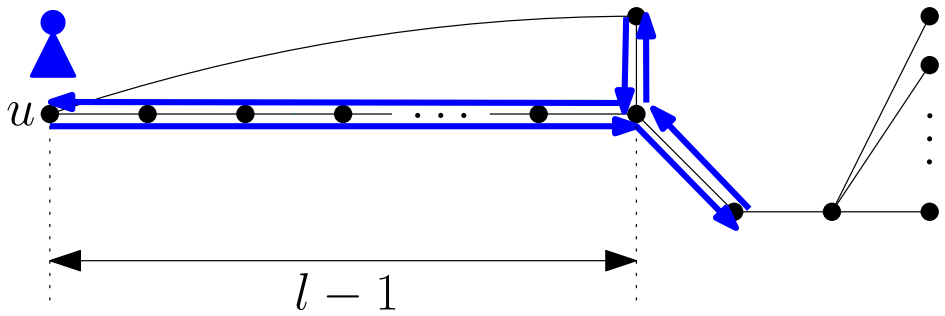
Exploring in general graphs

Unlike the case of trees, it is difficult to explore nodes at distance d since the exact distance is not known by the agent.
Example of exploration in which the agent does not explore everything at distance l if it is not allowed to traverse the same edge twice:



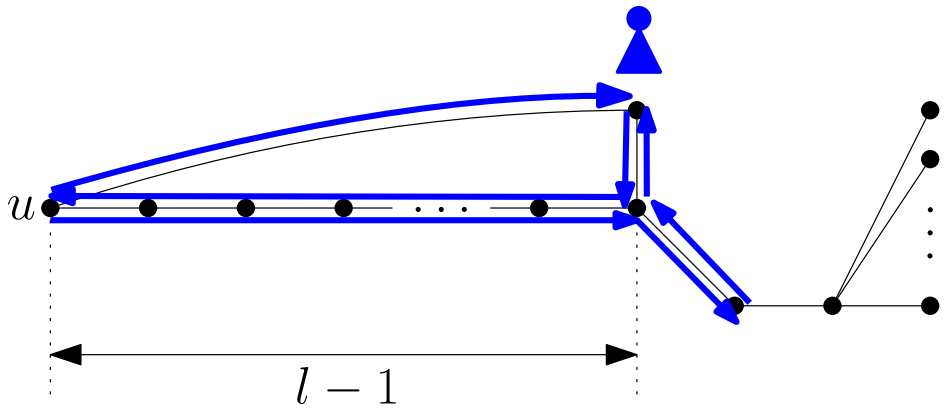
Exploring in general graphs

Unlike the case of trees, it is difficult to explore nodes at distance d since the exact distance is not known by the agent.
Example of exploration in which the agent does not explore everything at distance l if it is not allowed to traverse the same edge twice:



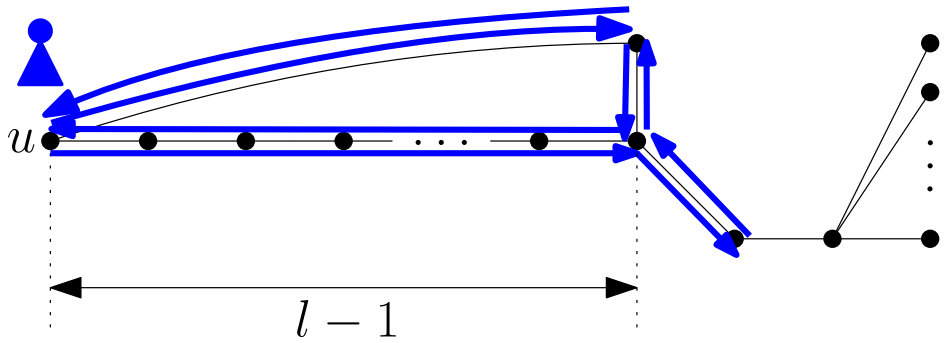
Exploring in general graphs

Unlike the case of trees, it is difficult to explore nodes at distance d since the exact distance is not known by the agent.
Example of exploration in which the agent does not explore everything at distance l if it is not allowed to traverse the same edge twice:



Exploring in general graphs

Unlike the case of trees, it is difficult to explore nodes at distance d since the exact distance is not known by the agent.
Example of exploration in which the agent does not explore everything at distance l if it is not allowed to traverse the same edge twice:



Solution: pruning algorithm

We use the pruning technique of Duncan et al. 2006.

Rough idea of the algorithm

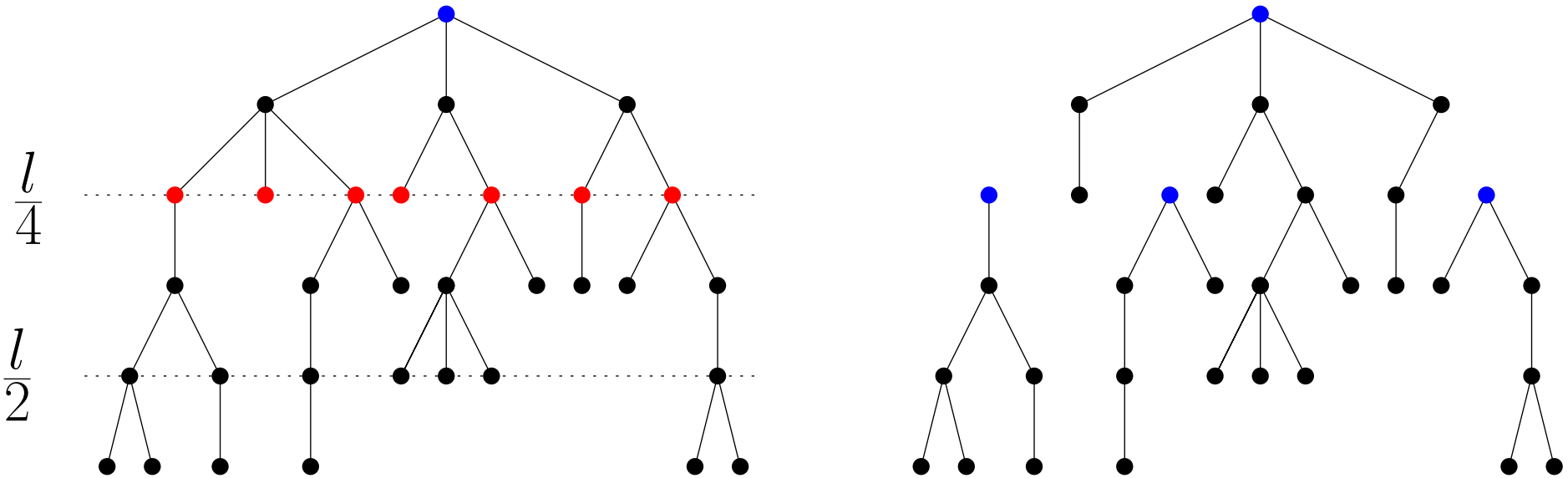
Repeat the following steps until there are no more incomplete node:

1. Move to an incomplete node u
2. Prune the subtree rooted in u by creating subtrees
3. Explore at depth $l/2$ from every node of the subtree containing u
4. Update the list of incomplete nodes
5. Update the subtrees by removing completed ones and merging those sharing a node.
6. backtrack to u

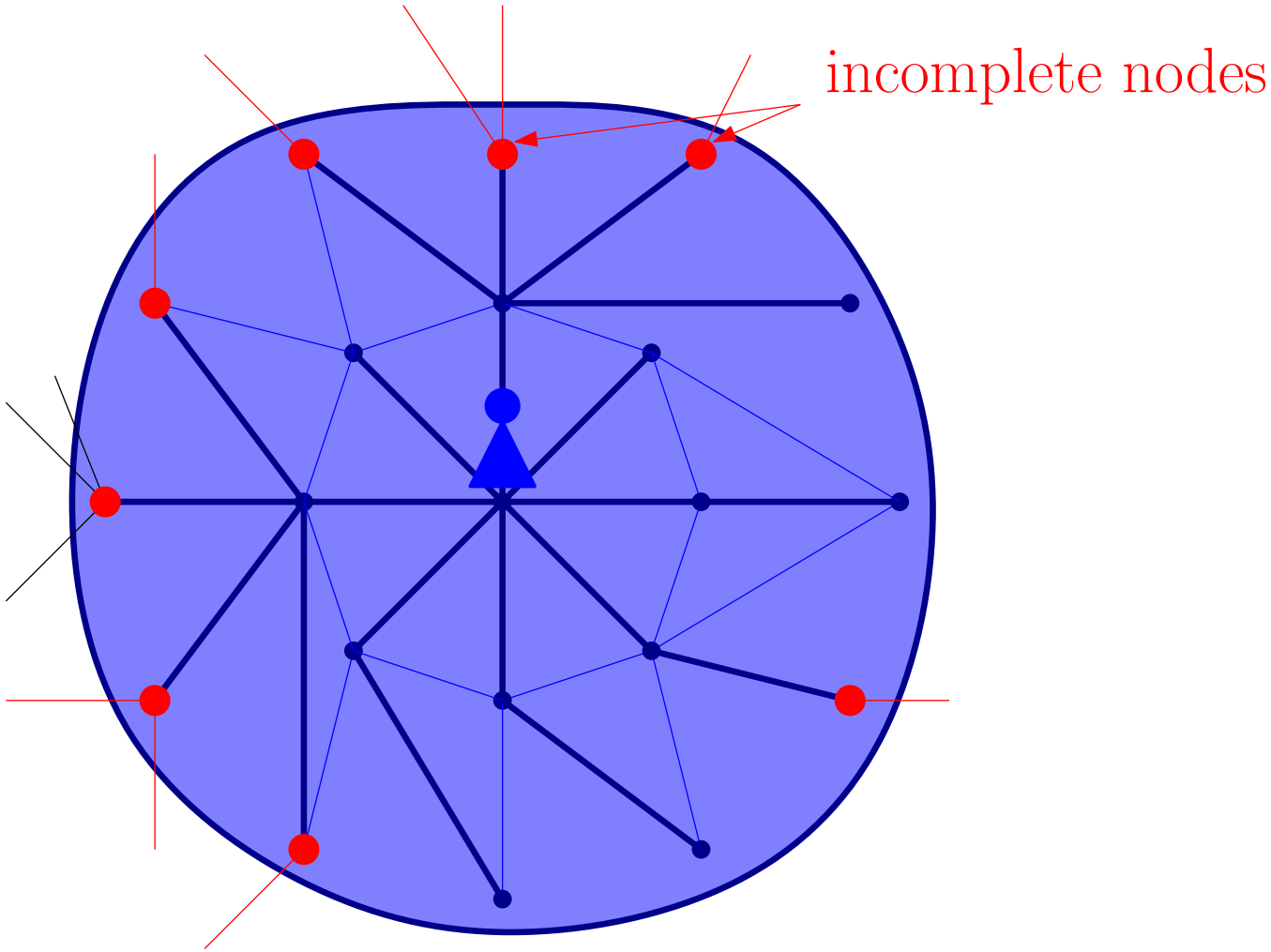
Pruning algorithm

Properties of the pruning

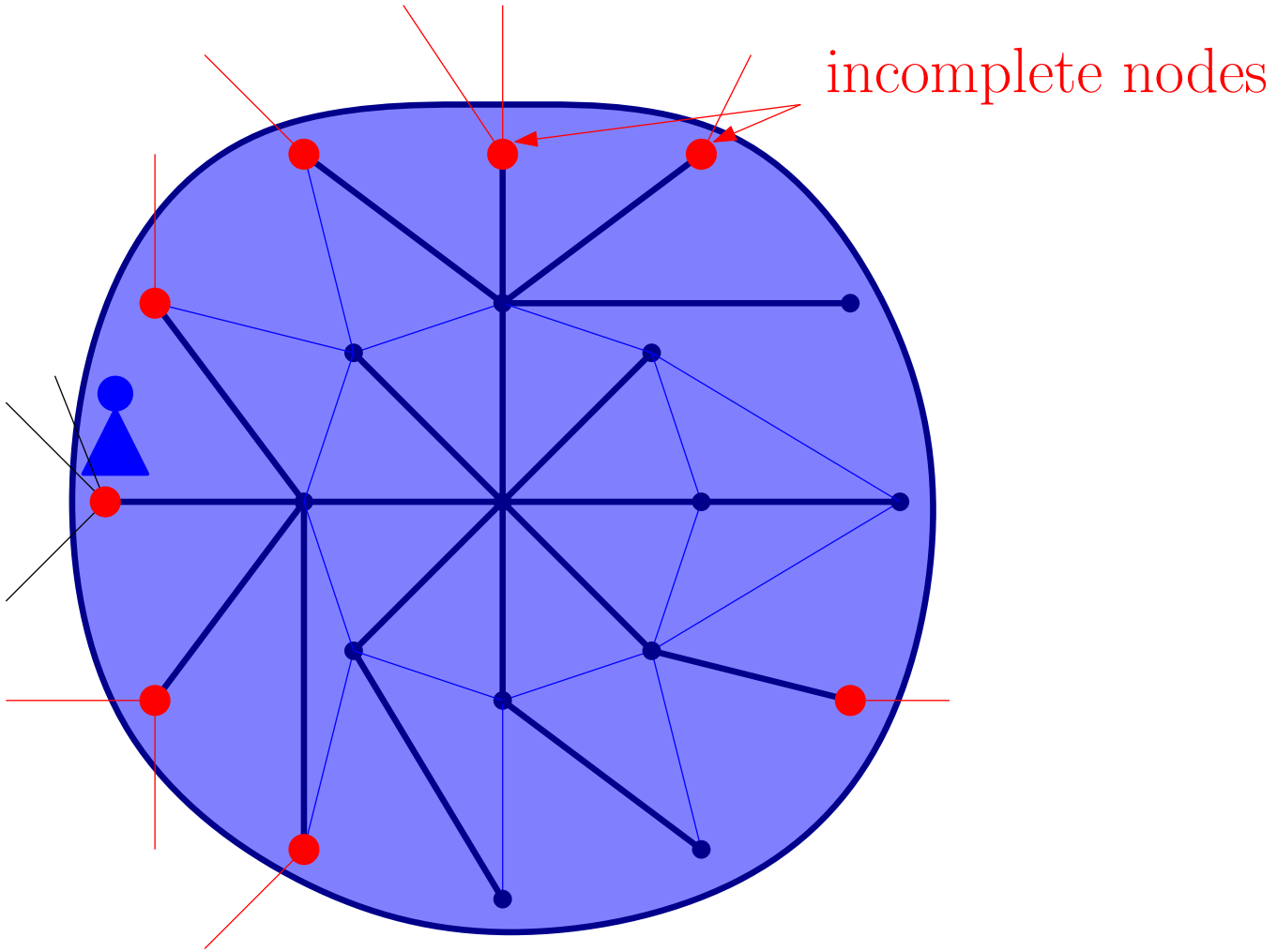
- the maximum depth on the original pruned is $l/2$
- no pruned subtree has depth less than $l/4$



Execution of exploration with pruning

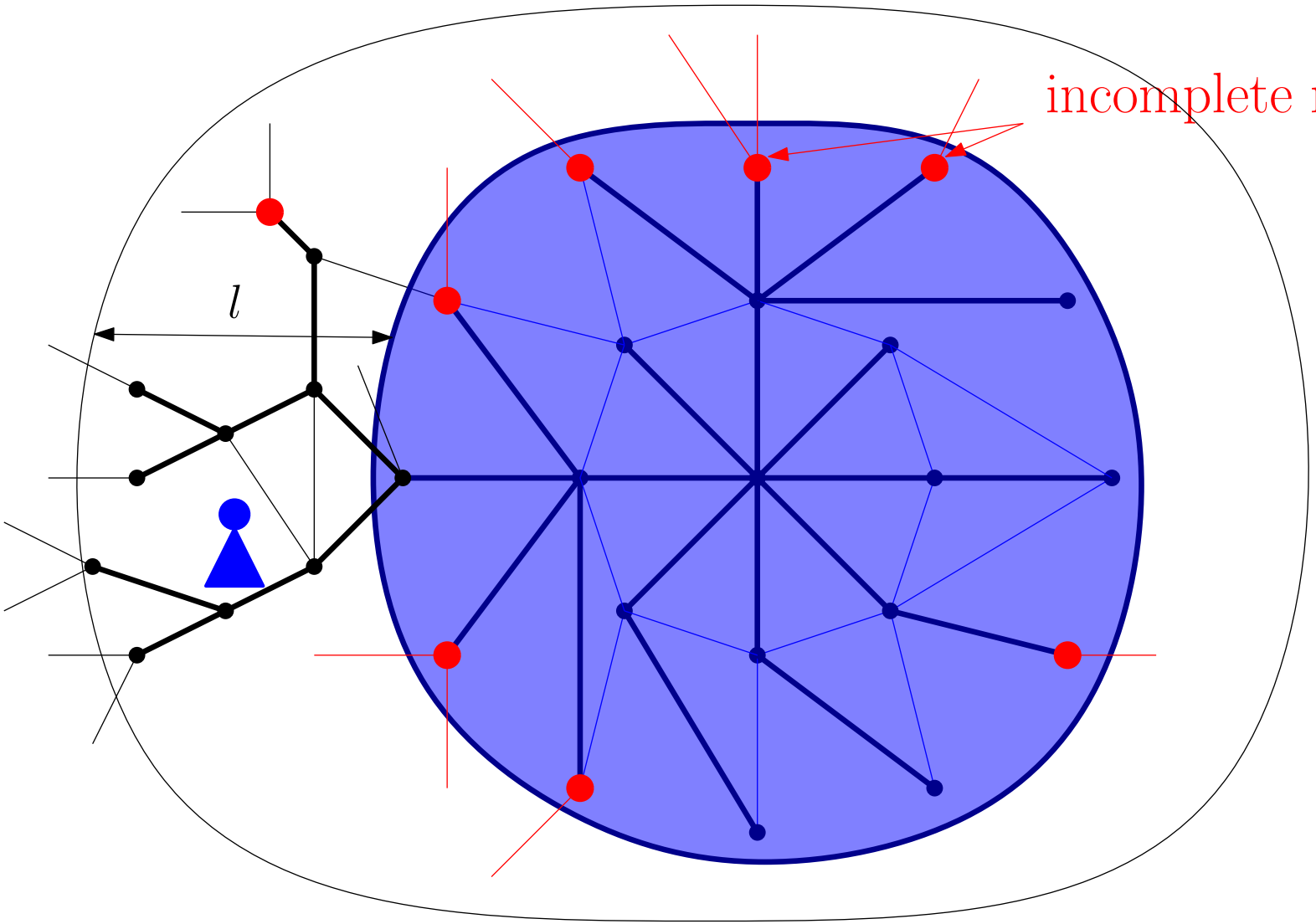


Execution of exploration with pruning



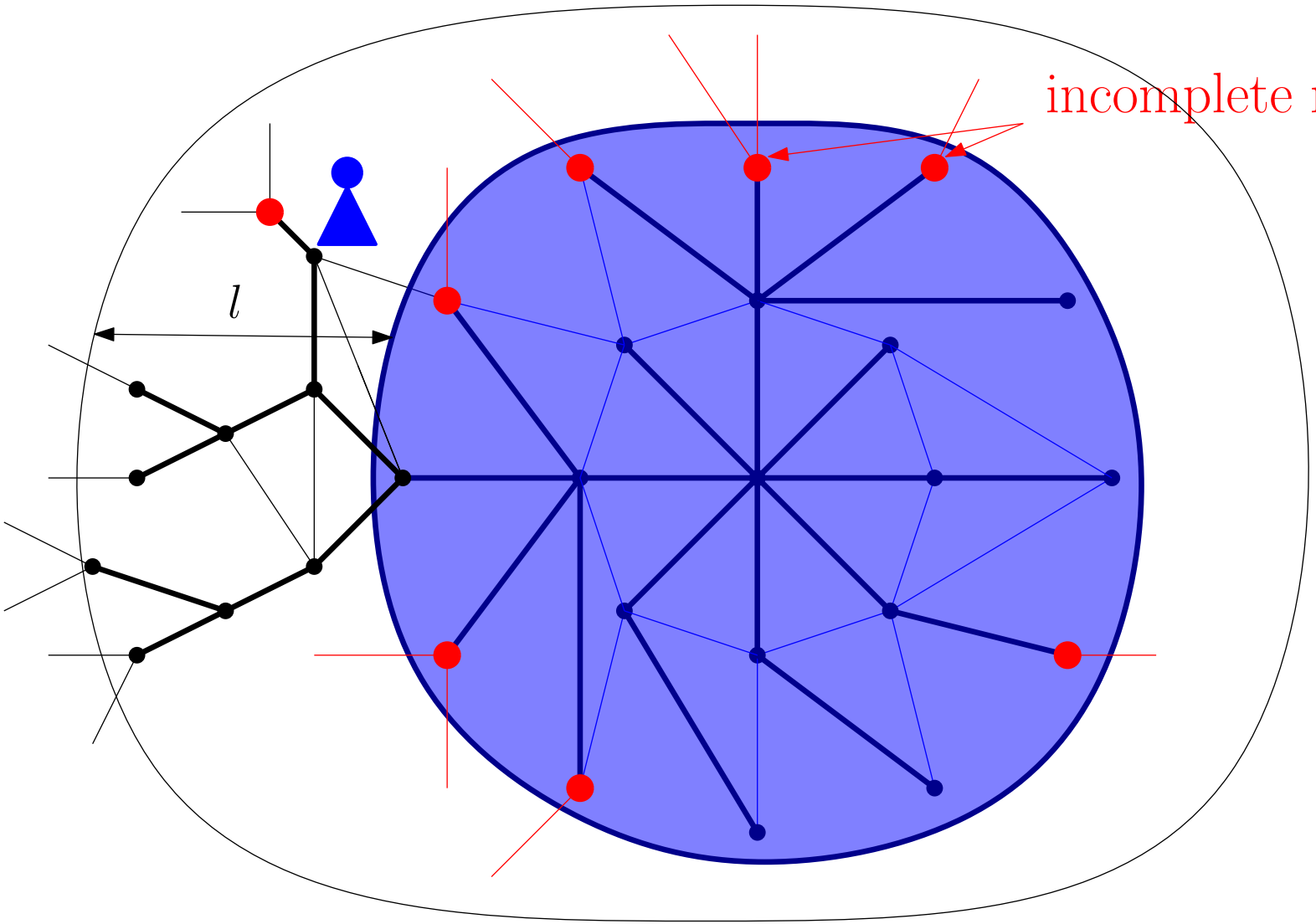
visit every incomplete node in a DFS order of the tree

Execution of exploration with pruning



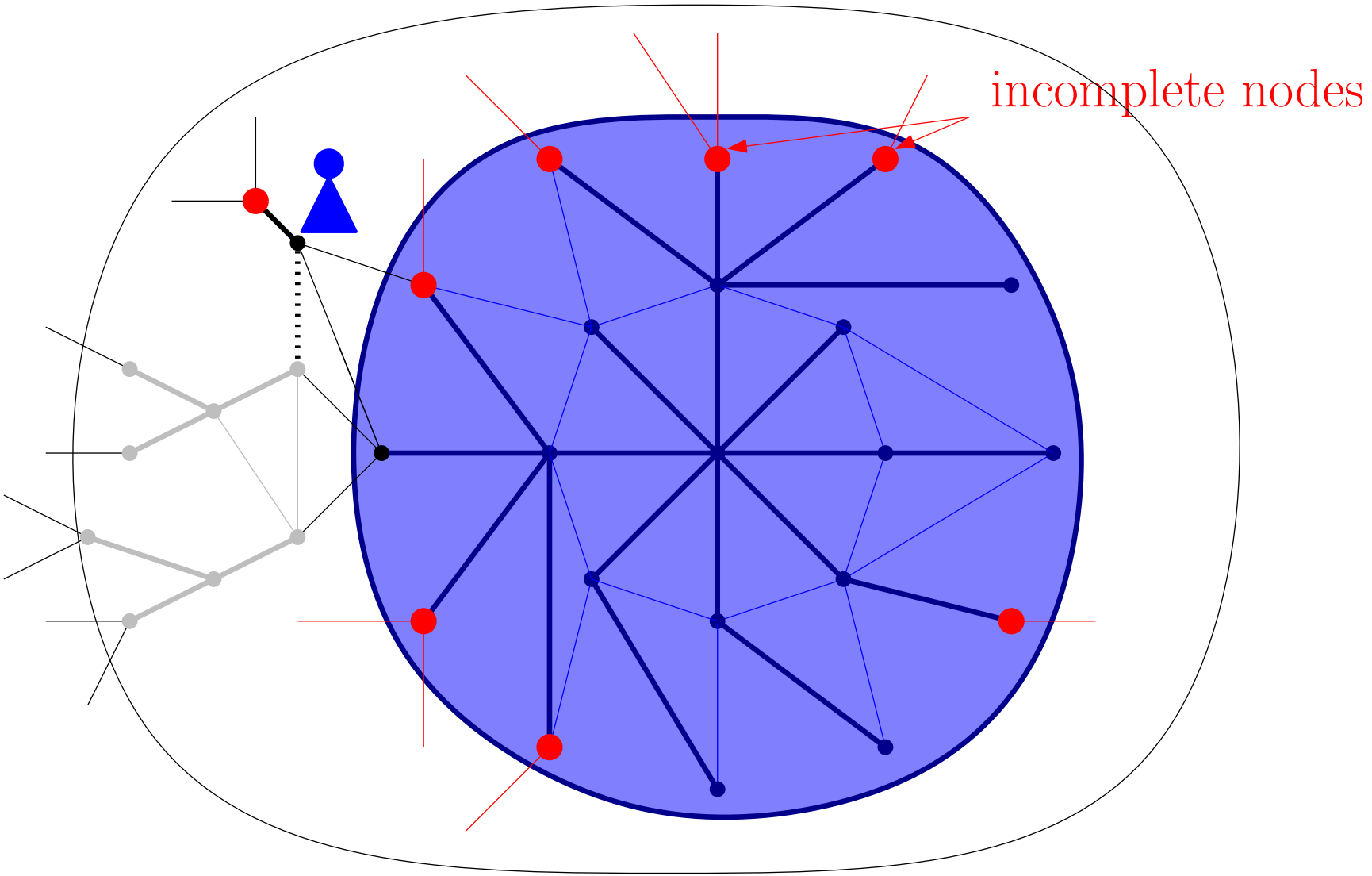
locally explore from the first incomplete node all nodes at distance l

Execution of exploration with pruning



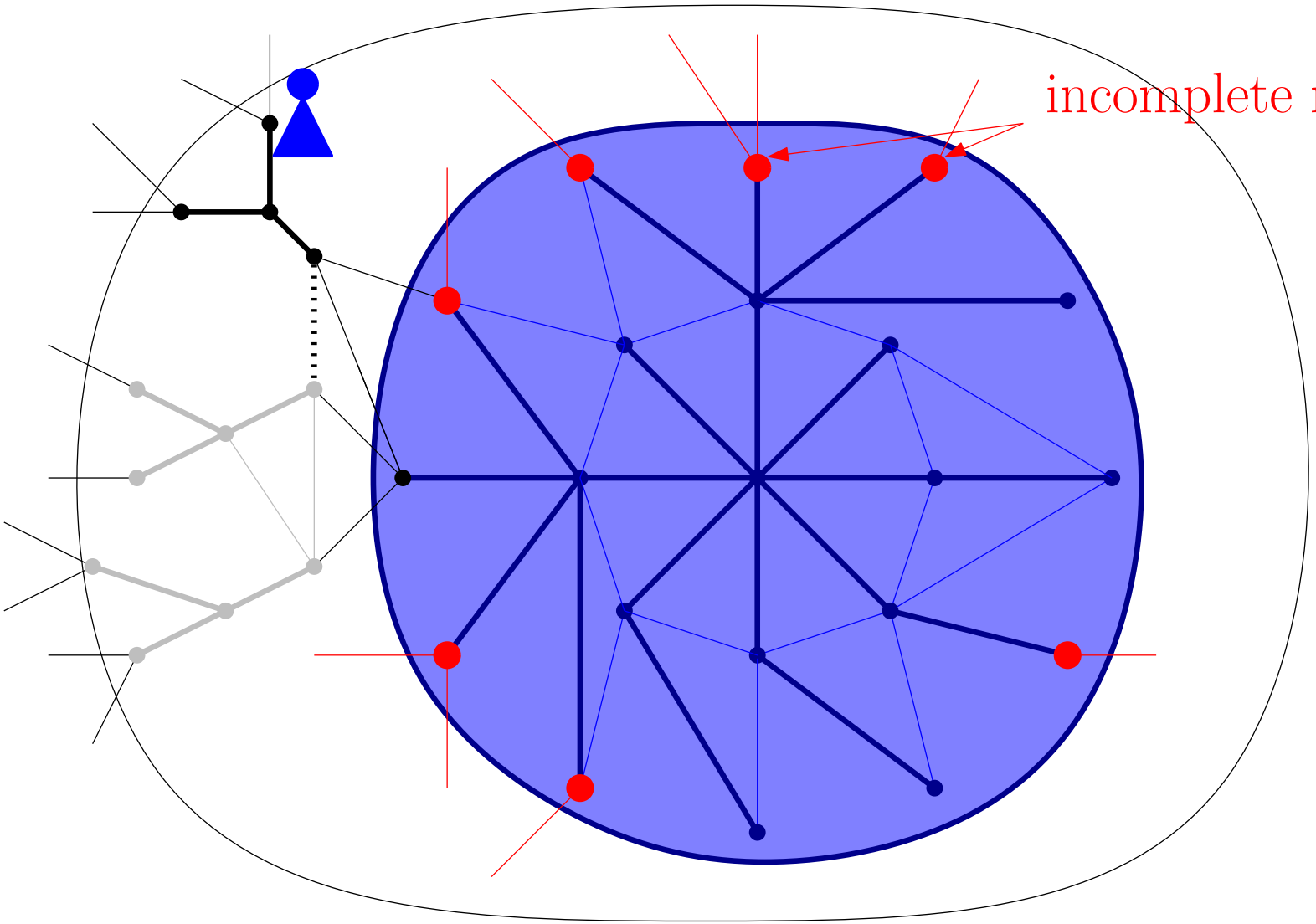
Since we do not allow to traverse twice an edge,
there may remain some incomplete nodes

Execution of exploration with pruning



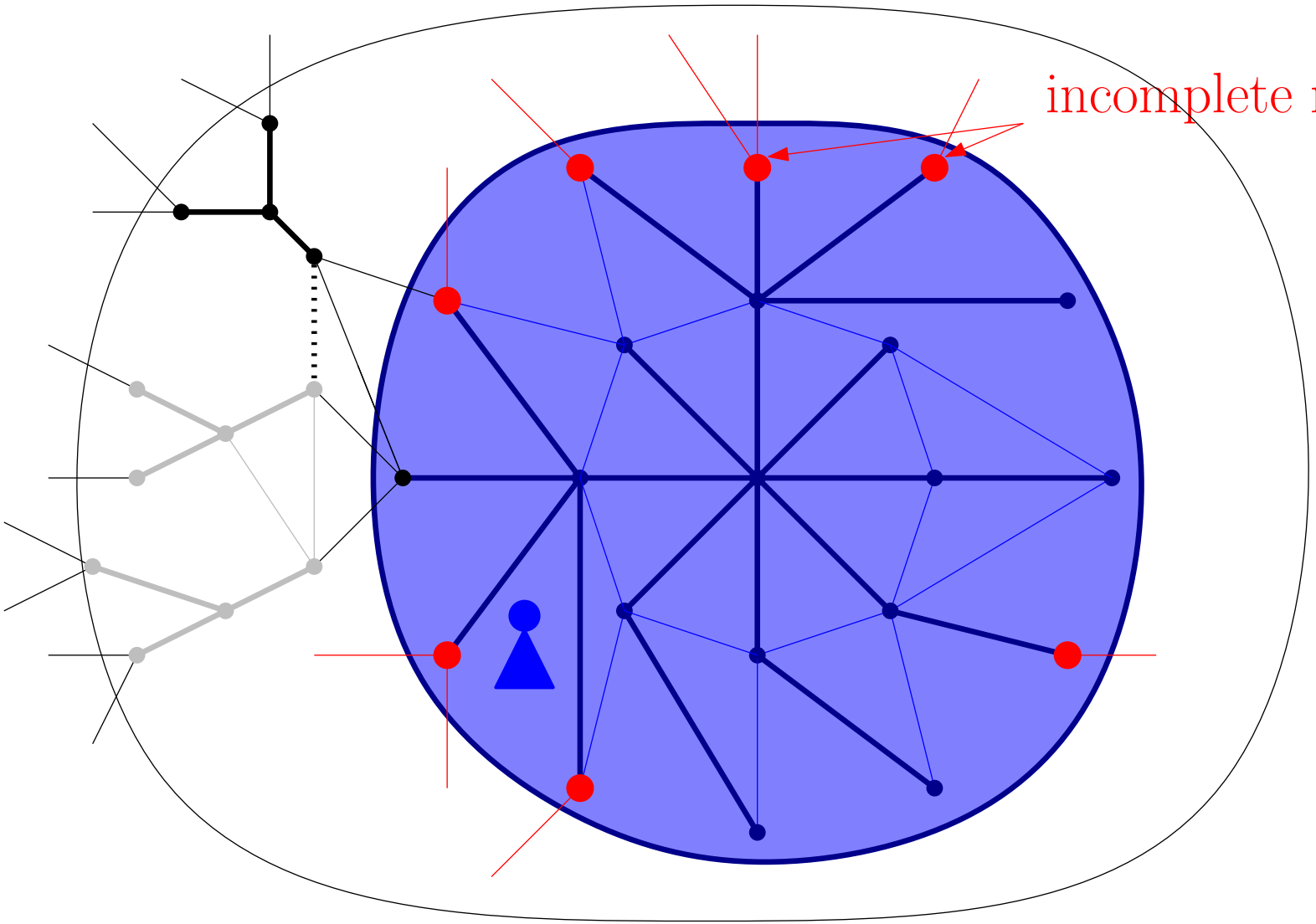
prune the tree, remove any complete tree

Execution of exploration with pruning



explore each pruned subtree one by one

Execution of exploration with pruning



Continue the same process with
the next incomplete node and so on

Rough idea of the analysis

Three kinds of edge traversal :

- moves between pruned subtrees
- moves inside pruned subtrees (in $O(e(h_i + l))$ since the pruned subtrees are disjoint)
- new edge exploration ($\leq 2e(h_i + l)$ since each of these edge is traversed once in each direction)

One of the key idea is that each pruned subtree has a size $\geq l/4$ and thus the moves between subtrees ($O(l)$ in order to move from one subtree to another one) is proportional to the number of moves inside the subtrees.

Complete code of our algorithm

Algorithm 2 `TreasureHunt`(x)

```
1  v := the current node;
2  M := ({v}, ∅); /* M is a global variable
3  repeat
4      Search(x);
```

Algorithm 3 `Search`(x)

```
1  v := the current node; m := |M|;
2  floor := ε_M(v); ceil := ⌊(1 + x) · floor⌋;
3  success := GlobalExpansion(1, ⊥);
4  floor := floor + 1; i := 0; l := ⌊(ceil − floor)/2⌋;
5  while l ≥ 1 and |M| < 2m and (i ≠ 1 or success = false) do
6      success := GlobalExpansion(l, m);
7      if success = true then
8          floor := floor + l; l := ⌊(ceil − floor)/2⌋;
9      else
10         ceil := floor + 2l − 1; l := ⌊l/2⌋;
11     M := B_floor(M, v);
12     i := i + 1;
```

Algorithm 4 `GlobalExpansion`(l, m)

```
1  v := the current node;
2  L := the array containing all the nodes of the boundary of M sorted in the order of
       the first visit through the DFS traversal of M from node v;
3  T := the tree produced by the DFS traversal of M from node v;
4  i := 1; b := m; T := ∅; /* T is a global variable                          */
5  while i ≤ |L| and (b ≥ 0 or b = ⊥) do
6      MoveTo(T, L[i]);
7      if l = 1 then
8          if b = ⊥ then
               /* We run CDFS(1, deg(L[i])) without using its return value.  */
9              (∗, ∗) := CDFS(1, deg(L[i]));
10         else
               /* We run CDFS(1, b) without using the second term of its
                  return value.                                              */
11             (b, ∗) := CDFS(1, b);
12     else
13         b := LocalExpansion(l, b);
14     i := i + 1;
15 MoveTo(T, v);
16 return the logical value of "b ≥ 0 or b = ⊥";
```

Algorithm 5 `CDFS`(l, b)

```
1  v := the current node; T := ({v}, ∅); bound := b;
2  if l > 0 then
3      Mark node v;
4      while node v is incomplete in M and bound ≥ 0 do
5          pt_1 := the smallest free port at node v in M;
6          Take port pt_1;
7          w := the current node;
8          pt_2 := the port by which the agent has just entered node w;
9          if v < w then
10             K := ({v, w}, {(v, w, pt_1, pt_2)});
11         else
12             K := ({v, w}, {(w, v, pt_2, pt_1)});
13         M := M ⊔ K; bound := bound − 1;
14         if w is not marked then
15             (bound, T') := CDFS(l − 1, bound);
16             T := T ⊔ T' ⊔ K;
17         Take port pt_2;
18     Unmark node v;
19 return (bound, T);
```

Algorithm 7 `Prune`(l)

```
1  v := the current node;
2  T_v := the tree of T containing node v;
3  T := T \ {T_v};
4  Root T_v at node v;
5  foreach node u of T_v such that d_{T_v}(u, v) = max{1, ⌊l/4⌋} do
6      T_u := the subtree of T_v rooted at u;
7      if ε_{T_u}(u) ≥ ⌊l/4⌋ − 1 then
8          T := T ∪ {T_u};
9          Remove from T_v all nodes that belong to T_u and all edges that are incident to
               a node of T_u;
10 T := T ∪ {T_v};
```

Algorithm 8 `Explore`(l, b)

```
1  bound := b; i := 1; v := the current node;
2  T := the tree of T containing node v;
3  V := array containing all the nodes of T sorted in the order of the first visit through
       the DFS traversal of T from node v;
4  while i ≤ |V| and bound ≥ 0 do
5      MoveTo(T, V[i]);
6      if node V[i] is incomplete in M then
7          (bound, T') := CDFS(⌊l/2⌋, bound);
8          T := T ∪ {T'};
9  return bound;
```

Algorithm 6 `LocalExpansion`(l, b)

```
1  bound := b; v := the current node;
2  if v is incomplete in M and no tree of T contains node v then
3      T := T ∪ {({v}, ∅)};
4  while IncompleteNodes(v, M, l) ∩ Nodes(T) ≠ ∅ and bound ≥ 0 do
5      u := the node with the smallest label in IncompleteNodes(v, M, l) ∩ Nodes(T);
6      MoveTo(M, u);
7      Prune(l);
8      bound := Explore(l, bound);
9      Remove from T every tree for which all the nodes are complete in M;
10     while there are two trees T and T' in T having a common node do
11         T'' := the spanning tree produced by the BFS traversal of T ⊔ T' from the
               node having the smallest label in T ⊔ T';
12         T := (T \ {T, T'}) ∪ {T''};
13     Execute in the reverse order all the edge traversals that have been made since the
           beginning of the current iteration of the while loop;
14 return bound;
```

Summary of results and open problem

Our result (ICALP 2021)

Treasure hunt on general graphs with cost $O(e(d) \log d)$.

Open problem

Is there a treasure hunt algorithm on general graphs with cost $O(e(d))$ (asymptotically optimal) ?

Thanks

Thanks for your attention !